

# Google App Engine Java и Google Web Toolkit: разработка Web-приложений

РАЗРАБОТКА  
Web-ПРИЛОЖЕНИЙ  
ДЛЯ Google App Engine

ИСПОЛЬЗОВАНИЕ  
ПРОГРАММНОГО ИНТЕРФЕЙСА  
СЛУЖБ Google App Engine

СОЗДАНИЕ Ajax-ПРИЛОЖЕНИЙ  
НА ОСНОВЕ Google Web Toolkit

КОМПОНЕНТЫ  
GUI-ИНТЕРФЕЙСА  
Google Web Toolkit

РАБОТА GWT-ПРИЛОЖЕНИЯ  
НА СТОРОНЕ СЕРВЕРА



**PRO**

ПРОФЕССИОНАЛЬНОЕ  
ПРОГРАММИРОВАНИЕ

Тимур Машнин

# **Google App Engine Java и Google Web Toolkit:**

**разработка  
Web-приложений**

Санкт-Петербург  
«БХВ-Петербург»  
2014

УДК 004.43:004.738.5  
ББК 32.973.26-018.2  
М38

**Машнин Т. С.**

М38 Google App Engine Java и Google Web Toolkit: разработка Web-приложений. — СПб.: БХВ-Петербург, 2014. — 352 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0828-5

Книга посвящена разработке Web-приложений для платформы Google App Engine и на основе фреймворка Google Web Toolkit на языке программирования Java и с использованием среды разработки Eclipse. Рассмотрено создание проектов и запуск GWT-приложений и приложений для Google App Engine. Описано использование программного интерфейса служб платформы Google App Engine, создание GUI-интерфейса на основе фреймворка Google Web Toolkit, оптимизация и интернационализация GWT-приложения. Показано применение фреймворков UiBinder и Activities and Places для разработки клиентской части GWT-приложения, а также фреймворков GWT RPC и RequestFactory для разработки серверной части GWT-приложения.

Материал книги сопровождается большим количеством примеров с подробным анализом исходных кодов.

*Для программистов*

УДК 004.43:004.738.5  
ББК 32.973.26-018.2

**Группа подготовки издания:**

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Екатерина Капалыгина</i>
Редактор	<i>Анна Кузьмина</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Марины Дамбиевой</i>

Подписано в печать 02.08.13.

Формат 70×100<sup>1/16</sup>. Печать офсетная. Усл. печ. л. 28,38.

Тираж 1000 экз. Заказ №

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"  
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-0828-5

© Машнин Т. С., 2014  
© Оформление, издательство "БХВ-Петербург", 2014

# Оглавление

<b>Введение .....</b>	<b>7</b>
Платформа Google App Engine .....	7
Фреймворк Google Web Toolkit.....	9
 <b>ЧАСТЬ I. ПЛАТФОРМА GOOGLE APP ENGINE.....</b>	<b>11</b>
 <b>Глава 1. Начало работы с Google App Engine.....</b>	<b>13</b>
Установка инструментов разработки.....	13
Создание проекта приложения .....	18
Запуск приложения из среды Eclipse .....	23
Развертывание приложения на платформе App Engine .....	25
Регистрация приложения .....	25
Страница администрирования приложением .....	26
Загрузка приложения в App Engine .....	31
Создание пользовательских разделов консоли администрирования.....	32
 <b>Глава 2. Журналирование приложения.....</b>	<b>34</b>
Библиотека <i>java.util.logging</i> .....	34
Библиотека Log4j.....	37
LogService API .....	38
 <b>Глава 3. Определение местоположения пользователя .....</b>	<b>42</b>
 <b>Глава 4. Аутентификация пользователей .....</b>	<b>45</b>
Ограничения доступа к ресурсам в дескрипторе web.xml.....	45
Программный интерфейс Users API.....	48
Аутентификация с помощью Google Accounts.....	49
Аутентификация с помощью OpenID.....	50
 <b>Глава 5. Использование JSP/JSF-страниц в GAE-приложении.....</b>	<b>54</b>
Технология JSP .....	54
Использование JSTL.....	57
Технология JSF .....	60

<b>Глава 6. Хранение данных приложения .....</b>	<b>65</b>
App Engine Datastore .....	65
Datastore API .....	67
Служба Remote API .....	80
JDO и JPA .....	82
JDO .....	82
JPA .....	91
Objectify .....	100
Twig .....	108
Slim3 .....	116
Google Cloud SQL .....	123
Google Cloud Storage и Blobstore .....	126
Google Cloud Storage .....	126
Blobstore .....	128
Сервис изображений .....	132
Служба Memcache .....	134
<b>Глава 7. Поддержка сессий и HTTPS .....</b>	<b>137</b>
Поддержка протокола HTTP/SSL .....	137
Использование сессий и cookie .....	138
<b>Глава 8. Сервисы сообщений Mail, XMPP и Channel .....</b>	<b>140</b>
Служба Mail .....	140
Отправка сообщений электронной почты .....	140
Получение сообщений электронной почты .....	141
Пример использования службы Mail .....	143
Служба XMPP .....	146
Отправка мгновенных сообщений .....	147
Получение мгновенных сообщений .....	147
Пример использования службы XMPP .....	150
Служба Channel .....	153
<b>Глава 9. Фильтры и обработка ошибок .....</b>	<b>159</b>
Фильтрация запросов и ответов .....	159
Обработка ошибок .....	161
<b>Глава 10. Разработка Backend-приложений .....</b>	<b>164</b>
<b>Глава 11. Использование протокола OAuth 2.0 для получения доступа к Google-сервисам .....</b>	<b>168</b>
Служба URL Fetch .....	172
<b>Глава 12. Запланированные задачи и очередь задач .....</b>	<b>175</b>
Служба Cron .....	175
Служба Task Queue .....	177
Очереди Push .....	177
Отложенные задачи <i>DeferredTask</i> .....	180
Очереди Pull .....	181

<b>Глава 13. Службы поиска Search и Prospective Search .....</b>	<b>183</b>
Служба Search .....	183
Служба Prospective Search.....	187
 <b>ЧАСТЬ II. ФРЕЙМВОРК GOOGLE WEB TOOLKIT .....</b>	<b>193</b>
 <b>Глава 14. Начало работы с Google Web Toolkit.....</b>	<b>195</b>
Установка плагинов фреймворка GWT .....	196
Создание проекта GWT-приложения.....	197
Структура проекта GWT-приложения .....	197
GWT-модули .....	198
Конфигурационный XML-файл определения GWT-модуля.....	200
Модель программирования фреймворка GWT .....	203
Запуск GWT-приложения в режиме разработки.....	208
Запуск GWT-приложения как Web-приложения .....	210
 <b>Глава 15. Компоненты графического интерфейса пользователя.....</b>	<b>214</b>
Кнопка <i>Button</i> .....	215
Отличие метода <i>setText()</i> от метода <i>setHTML()</i> .....	215
Обработчики событий кнопки .....	216
Определение свойств кнопки.....	222
Кнопка <i>PushButton</i> .....	222
Переключатель <i>RadioButton</i> .....	224
Флажок <i>CheckBox</i> .....	225
Компонент выбора даты <i>DatePicker</i> .....	226
Кнопка <i>ToggleButton</i> .....	229
Текстовое поле <i>TextBox</i> .....	231
Поле ввода пароля <i>PasswordTextBox</i> .....	232
Текстовая область <i>TextArea</i> .....	233
Гиперссылка <i>Hyperlink</i> .....	234
Гиперссылка <i>Anchor</i> .....	236
Список выбора <i>ListBox</i> .....	237
Компоненты <i>Cell Widgets</i> .....	238
Столбец <i>CellList</i> .....	239
Таблица <i>CellTable</i> .....	249
Таблица <i>DataGrid</i> .....	255
Дерево <i>CellTree</i> .....	257
Дерево <i>CellBrowser</i> .....	259
Панель меню <i>MenuBar</i> .....	260
Дерево <i>Tree</i> .....	263
Поле подсказки <i>SuggestBox</i> .....	265
Редактор текста <i>RichTextArea</i> .....	267
Таблица <i>FlexTable</i> .....	269
Таблица <i>Grid</i> .....	271
Всплывающие окна <i>PopupPanel</i> , <i>DecoratedPopupPanel</i> , <i>LoggingPopup</i> и <i>DialogBox</i> .....	273
Уведомление <i>NotificationMole</i> .....	276
Панели с закладками <i>TabPanel</i> и <i>TabLayoutPanel</i> .....	276
Загрузчик файлов <i>FileUpload</i> и форма <i>FormPanel</i> .....	279
Скрытое поле <i>Hidden</i> .....	280

Фрейм <i>Frame</i> .....	281
Изображение <i>Image</i> .....	281
Метка <i>Label</i> .....	283
Метка <i>HTML</i> .....	284
Метка <i>InlineHTML</i> .....	284
Метка <i>InlineLabel</i> .....	284
Медиакомпоненты <i>Audio</i> и <i>Video</i> .....	284
Компонент <i>Video</i> .....	284
Компонент <i>Audio</i> .....	285
Компонент <i>Canvas</i> .....	286
Панели компоновки .....	287
Панели <i>AbsolutePanel</i> , <i>RootPanel</i> , <i>RootLayoutPanel</i> и <i>LayoutPanel</i> .....	287
Панели <i>StackPanel</i> , <i>DecoratedStackPanel</i> и <i>StackLayoutPanel</i> .....	290
Панели <i>DockPanel</i> , <i>DockLayoutPanel</i> и <i>SplitLayoutPanel</i> .....	292
Панели <i>HorizontalPanel</i> и <i>VerticalPanel</i> .....	294
Панель <i>FlowPanel</i> .....	295
Панель <i>HTMLPanel</i> .....	295
Панель <i>HeaderPanel</i> .....	296
Панели <i>SimplePanel</i> , <i>DecoratorPanel</i> , <i>FocusPanel</i> и <i>SimpleLayoutPanel</i> .....	296
Панель <i>ScrollPanel</i> .....	298
Панель <i>DisclosurePanel</i> .....	299
Панель <i>CaptionPanel</i> .....	299
Изменение внешнего вида GWT-компонентов .....	300
Фреймворк <i>UiBinder</i> .....	304
<b>Глава 16. Интернационализация GWT-приложения .....</b>	<b>307</b>
Статическая интернационализация .....	308
Динамическая интернационализация .....	312
Интернационализация <i>UiBinder</i> .....	312
<b>Глава 17. Программный интерфейс JavaScript Native Interface .....</b>	<b>314</b>
<b>Глава 18. Оптимизация GWT-приложения .....</b>	<b>316</b>
Разделение кода (Code Splitting) .....	316
Отложенное связывание (Deferred Binding) .....	318
<b>Глава 19. Поддержка истории Web-браузера .....</b>	<b>321</b>
<b>Глава 20. Фреймворк Activities and Places .....</b>	<b>324</b>
<b>Глава 21. Взаимодействие GWT-приложения с сервером .....</b>	<b>330</b>
Фреймворк GWT RPC .....	330
Библиотека GWT HTTP Client .....	335
Фреймворк RequestFactory .....	341
<b>Литература .....</b>	<b>349</b>
<b>Предметный указатель .....</b>	<b>350</b>

# Введение

Корпорация Google среди своих многочисленных Web-продуктов, включающих в себя поисковые и рекламные сервисы, сервисы Gmail и Google Docs, различные инструменты коммуникации, публикации, разработки, анализа и картографии, предлагает платформу Google App Engine и фреймворк Google Web Toolkit.

## Платформа Google App Engine

Google App Engine (GAE) является PaaS-платформой (Platform as a Service, платформа как сервис) и представляет собой платформу облачного хостинга Web-приложений в инфраструктуре Google, содержащей центры данных и серверы. Платформа GAE избавляет от необходимости самостоятельной поддержки инфраструктуры развернутого сайта и заботы о его масштабировании. При этом платформа GAE сама регулирует использование серверов и центров данных инфраструктуры Google для обеспечения работы сайта. Развернутое Web-приложение может использовать бесплатно 500 Мбайт дискового пространства и бесплатно обслуживать около 5 млн запросов в месяц (квоты использования ресурсов можно посмотреть на странице по адресу <https://developers.google.com/appengine/docs/quotas>). Необходимость сайта в увеличении потребляемых ресурсов удовлетворяется автоматически за отдельную плату, при этом оплата производится только за фактически используемые дополнительные ресурсы. Таким образом, при росте проекта отпадает необходимость в аренде/покупке серверов и заказе VDS-серверов. Для работы Web-приложений платформа GAE обеспечивает среды выполнения Java, Python и Go.

Платформа GAE создает для выполнения Web-приложений "песочницы", которые накладывают ограничения на работу приложений.

- ◆ GAE-приложение может обмениваться данными с другими приложениями только с помощью HTTP/HTTPS-запросов и сообщений электронной почты. Поддержку HTTP/HTTPS-запросов обмена данными с другими серверами в Интернете обеспечивает служба GAE-платформы URL Fetch, а обмен сообщениями электронной почты обеспечивает GAE-служба Mail.



- ◆ GAE-приложение может взаимодействовать с дисковым пространством хостинга только путем чтения из тех ресурсных файлов, которые были загружены вместе с приложением. Для хранения и работы с данными GAE-приложение может использовать GAE-службы Blobstore, Images, Datastore и Memcache. Служба Blobstore обеспечивает работу приложения с большими двоичными объектами, размером до 2 Гбайт. Служба Images обеспечивает преобразования изображений, размером до 1 Мбайт, включая изменение размера, поворот, отражение, кадрирование и коррекцию контрастности и цветопередачи. Служба Datastore обеспечивает хранение данных приложения в двух типах хранилищ — High Replication Datastore (HRD) (по умолчанию) и Master/Slave Datastore (в настоящее время не рекомендуется). Выбор хранилища производится при регистрации приложения в GAE. Хранилище HRD — это хранилище с высоким уровнем репликации данных по нескольким центрам данных, а хранилище Master/Slave обеспечивает репликацию данных по системе "главное/подчиненное хранилище" и является менее затратным по сравнению с хранилищем HRD, которое, в свою очередь, обеспечивает большую доступность к данным. Служба Memcache обеспечивает хранение данных приложения объемом до 1 Мбайт в кэше памяти.
- ◆ При обработке клиентского запроса GAE-приложение не может создавать фоновые процессы и должно обрабатывать запрос в течение определенного времени — в настоящее время это 30 секунд. Если запрос не обрабатывается в течение этого времени, то возникает исключение. Размер запроса и ответа не должен превышать 10 Мбайт.

Кроме того, накладываются ограничения на размер отдельного файла приложения — до 10 Мбайт, на общий размер всех файлов приложения — до 150 Мбайт, и на общее количество файлов приложения — до 3000.

Помимо служб URL Fetch, Mail, Blobstore, Images, Datastore и Memcache платформа GAE предоставляет в пользование приложению службы Namespaces, OAuth, Task Queue, Cron, Users и XMPP.

Служба Namespaces позволяет разделить клиентов GAE-приложения по пространствам имен. Служба OAuth дает возможность клиенту предоставить доступ одного приложения к другому от своего имени. Служба Task Queue обеспечивает для приложения создание фоновых задач, не обрабатывающих клиентский запрос, и постановку их в очередь выполнения. Служба Cron позволяет создавать приложению запланированные задачи, выполняемые в конкретное время или регулярно через определенные периоды. Служба Users помогает приложению организовать аутентификацию/авторизацию клиента. Служба XMPP позволяет приложению организовать сервис мгновенных сообщений.

Кроме GAE-службы Datastore, обеспечивающей хранение данных приложения в нереляционной базе данных, организовать хранение данных GAE-приложения позволяют такие Google-сервисы, как Google Cloud SQL — хранение данных в реляционной базе данных, основанной на MySQL, и Google Cloud Storage — хранение больших объектов, размером до терабайта.

Среда выполнения Java платформы GAE включает в себя виртуальную машину JVM и Servlet-контейнер. Применение виртуальной машины JVM обеспечивает

создание GAE-приложений с использованием JVM-языков JavaScript, Ruby, Scala и др.

Использование классов платформы Java для GAE-приложений ограничено белым списком классов среды выполнения Java (<https://developers.google.com/appengine/docs/java/jrewhitelist?hl=ru-RU>).

Следуя ограничениям "песочницы" платформы GAE, Java-приложение не может создавать экземпляры классов `java.lang.ThreadGroup`, `java.lang.Thread`, `java.util.concurrent.ThreadPoolExecutor`, `java.util.Timer`, использовать классы записи библиотеки `java.io`. Методы `exit()`, `gc()`, `runFinalization()`, `runFinalizersOnExit()`, `inheritedChannel()`, `console()`, `load()`, `loadLibrary()`, `setSecurityManager()` класса `java.lang.System` не работают.

Среда выполнения Python платформы GAE содержит стандартную библиотеку Python, функции которой могут использоваться с учетом ограничений "песочницы", а также библиотеки Django, WebOb и PyYAML. В приложение могут добавляться и другие библиотеки, созданные на чистом языке Python.

Регистрация приложения в App Engine осуществляется с помощью Web-страницы консоли администратора по адресу <https://appengine.google.com/>, при этом на один аккаунт разработчика можно зарегистрировать до 10 приложений. В процессе регистрации для приложения создается поддомен домена **appspot.com** и формируется URL-адрес доступа к приложению в виде **http://[идентификатор приложения].appspot.com/**. После регистрации приложение можно привязать к домену, зарегистрированному в бизнес-приложении Google Apps (<http://www.google.com/apps/intl/ru/business/>). Само добавление приложения в App Engine после его регистрации осуществляется с помощью инструментов разработки GAE-приложений.

Для разработки приложений GAE-платформа предоставляет наборы Google App Engine SDK для Java, Python и Go, обеспечивающие эмуляцию соответствующих сред выполнения, работу GAE-служб и инструменты загрузки приложений в GAE-платформу. Для разработки Java-приложений предоставляется также Google-модуль для среды разработки Eclipse.

## Фреймворк Google Web Toolkit

Google Web Toolkit (GWT) обеспечивает создание RIA (Rich Internet Application) Ajax-приложений на основе Java-кода.

GWT-приложение разрабатывается на языке Java, и его код на стадии разработки содержит определение GUI-интерфейса, обработку событий интерфейса и работу с данными. Затем Java-код GWT-приложения, содержащий определение GUI-интерфейса и обработку его событий, компилируется в JavaScript-код Web-страницы клиента, а на стороне сервера остается лишь Java-код Web-сервисов, отвечающих за работу с данными. При этом JavaScript-код Web-страницы содержит Ajax-клиентов Web-сервисов.

GWT-компилятор Java-to-JavaScript создает на выходе набор версий приложения, обеспечивающий совместимость с различными Web-браузерами и интернационали-

зацию приложения. В результате запускаемый на стороне клиента код является компактным и быстро загружается, т. к. содержит только те ресурсы, которые требуются конкретным Web-браузером. Кроме того, для ускорения запуска приложения можно создавать точки разделения JavaScript-кода, в которых приложение будет производить Ajax-загрузку дополнительного кода.

Программный интерфейс JavaScript Native Interface (JSNI) GWT-платформы позволяет также при разработке встраивать напрямую JavaScript-код в Java-код приложения, что обеспечивает использование сторонних JavaScript-библиотек.

GWT-фреймворк обеспечивает поддержку HTML5-тегов `<canvas>`, `<audio>` и `<video>`.

Фреймворк Google Web Toolkit (GWT) содержит:

- ◆ GWT SDK — включает в себя Java-библиотеки программного интерфейса GWT-платформы, GWT-компилятор Java-кода в JavaScript-код, локальный сервер разработки, позволяющий запускать и отлаживать Java-код приложения без его компиляции в JavaScript-код;
- ◆ Speed Tracer — расширение Web-браузера Chrome, позволяющее анализировать производительность GWT-приложения;
- ◆ Google Plugin for Eclipse (GPE) — плагин, обеспечивающий разработку GWT-приложений в среде Eclipse;
- ◆ GWT Designer — плагин, обеспечивающий визуальное редактирование GUI-интерфейса GWT-приложения в среде Eclipse.

Для создания GUI-интерфейса приложения GWT-фреймворк предлагает набор готовых панелей и виджетов, из которых можно компоновать клиентскую часть приложения.

Серверную часть приложения, содержащую Web-сервисы данных для клиентской части, можно создавать с помощью библиотек GWT-RPC и RequestFactory GWT-фреймворка.

В данной книге рассматривается создание Java-приложений для развертывания на платформе Google App Engine и создание приложений на основе фреймворка Google Web Toolkit.



# ЧАСТЬ I

## Платформа Google App Engine

- Глава 1.** Начало работы с Google App Engine
- Глава 2.** Журналирование приложения
- Глава 3.** Определение местоположения пользователя
- Глава 4.** Аутентификация пользователей
- Глава 5.** Использование JSP/JSF-страниц в GAE-приложении
- Глава 6.** Хранение данных приложения
- Глава 7.** Поддержка сессий и HTTPS
- Глава 8.** Сервисы сообщений Mail, XMPP и Channel
- Глава 9.** Фильтры и обработка ошибок
- Глава 10.** Разработка Backend-приложений
- Глава 11.** Использование протокола OAuth 2.0 для получения доступа к Google-сервисам
- Глава 12.** Запланированные задачи и очередь задач
- Глава 13.** Службы поиска Search и Prospective Search

## ГЛАВА 1



# Начало работы с Google App Engine

## Установка инструментов разработки

Разработка на основе платформы Google App Engine (GAE) подразумевает создание Web-приложений. Для разработки Web-приложений на языке Java платформа GAE предлагает использовать набор Eclipse-плагинов, облегчающих организацию проекта, тестирование и развертывание приложения. Поэтому в качестве первого шага скачаем по адресу <http://www.eclipse.org/downloads/> последнюю версию среды Eclipse IDE for Java EE Developers, обеспечивающую все необходимые инструменты разработки Java Web-приложений. При этом предполагается, что набор JDK платформы Java уже установлен на компьютере (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>).

### *ПРИМЕЧАНИЕ*

При написании книги использовался релиз Indigo SR2 среды Eclipse.

После распаковки скачанного дистрибутива запустим среду Eclipse, закроем страницу приветствия **Welcome** кнопкой **Workbench** и в меню **Help** выберем команду **Install New Software** — в результате откроется окно мастера инсталляции Eclipse-плагинов (рис. 1.1).

Нажмем кнопку **Add** и в поле **Name** введем **Google**, а в поле **Location** — адрес установки Google-модуля, для Eclipse Indigo <http://dl.google.com/eclipse/plugin/3.7> (рис. 1.2).

Нажмем кнопку **OK**. После поиска в Интернете в окне мастера отобразится набор плагинов Google-модуля (рис. 1.3).

Плагин Google App Engine Tools for Android обеспечивает создание клиент-серверных приложений для платформы Android, в которых клиентская часть размещается в мобильном Android-устройстве, а серверная часть — в облаке GAE-платформы. Данный плагин требует предварительной установки плагина Android Development Tools (ADT) для разработки Android-приложений в среде Eclipse.

Плагин Google Plugin for Eclipse обеспечивает поддержку проектов GAE- и GWT-приложений в среде Eclipse.

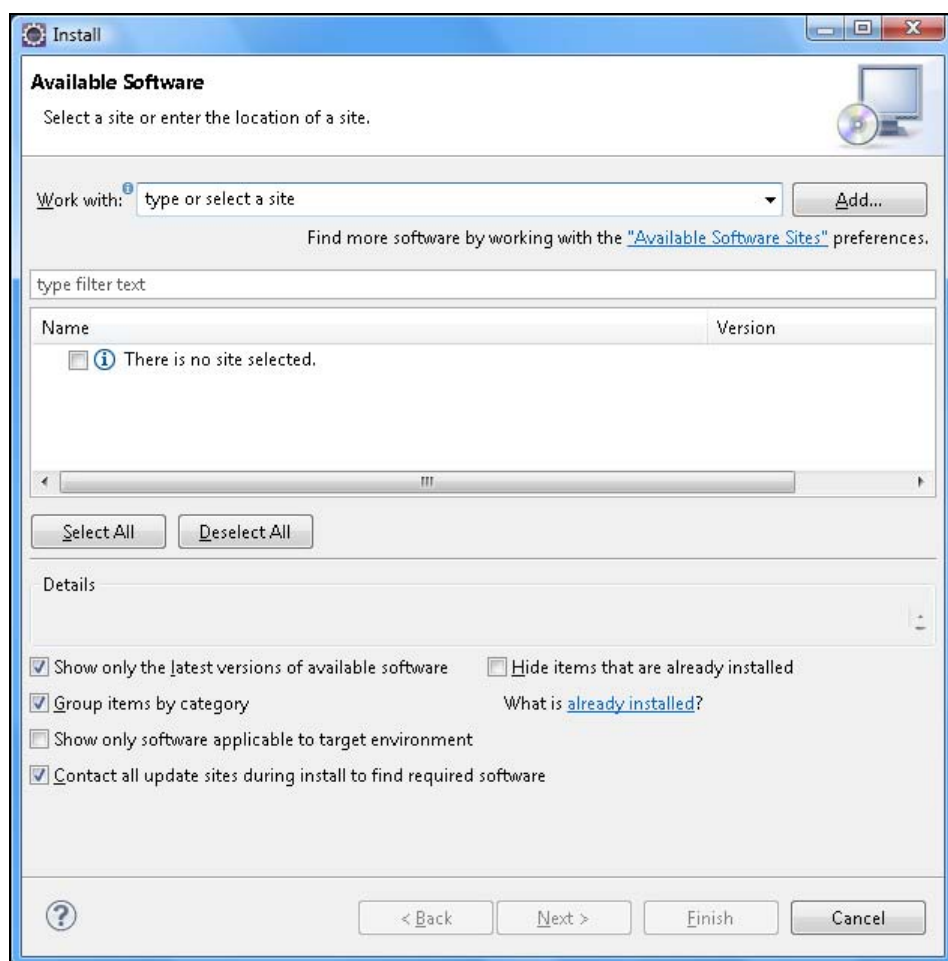


Рис. 1.1. Мастер установки Eclipse-плагинов

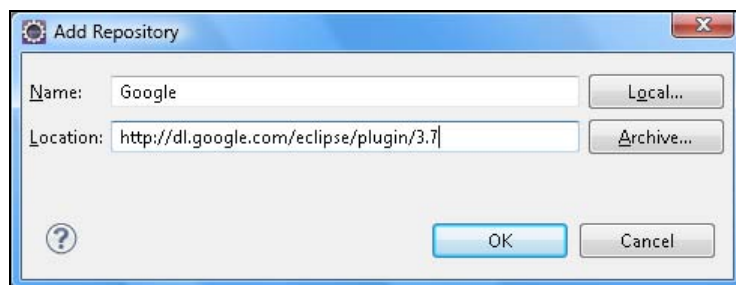


Рис. 1.2. Определение адреса установки Google-модуля

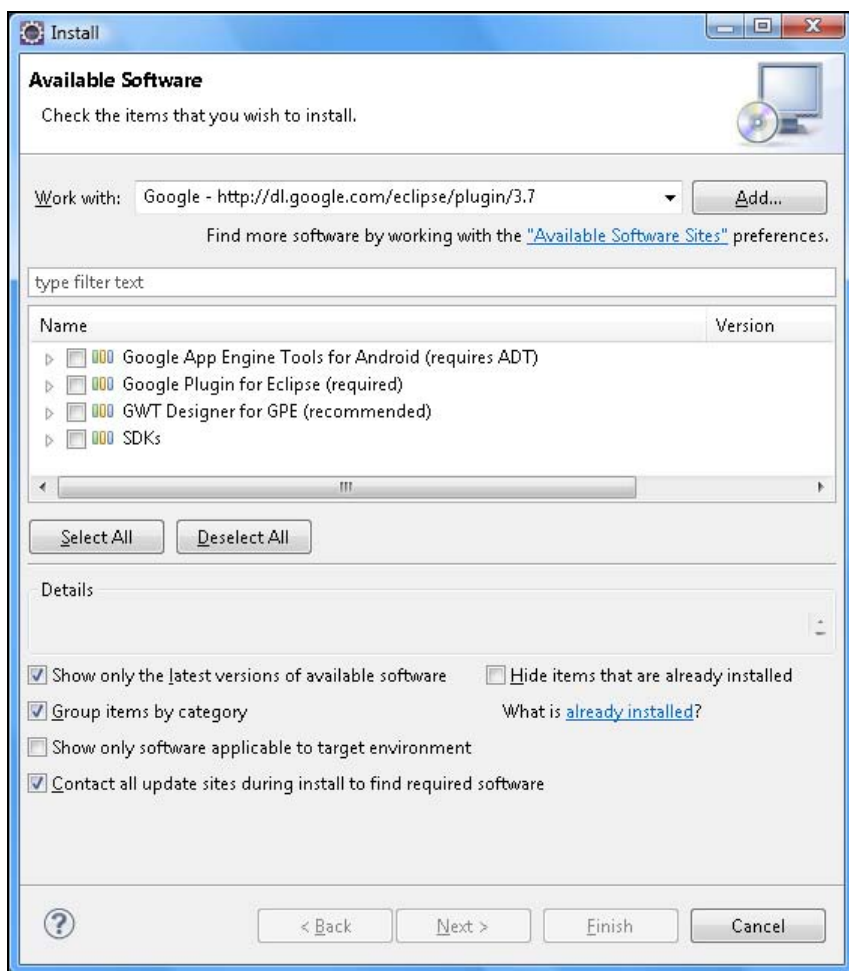

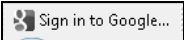


Рис. 1.3. Набор плагинов Google-модуля для среды Eclipse

Плагин GWT Designer for GPE предоставляет средства визуального редактирования GUI-интерфейса GWT-приложений.

Плагин SDKs обеспечивает установку наборов Google App Engine Java SDK и GWT SDK, необходимых для разработки GAE- и GWT-приложений в среде Eclipse.

Для начала отметим флажки **Google Plugin for Eclipse** и **Google App Engine Java SDK**, нажмем кнопку **Next** мастера и, следуя инструкциям, установим выбранные плагины с перезапуском среды Eclipse.

В результате установки плагина Google Plugin for Eclipse (GPE) в Workbench-окне среды Eclipse появятся кнопки **Google Services and Development Tools**  и , а в диалоговом окне команды **New | Other** меню **File** появятся разделы **Google** и **Google Web Toolkit** (рис. 1.4).

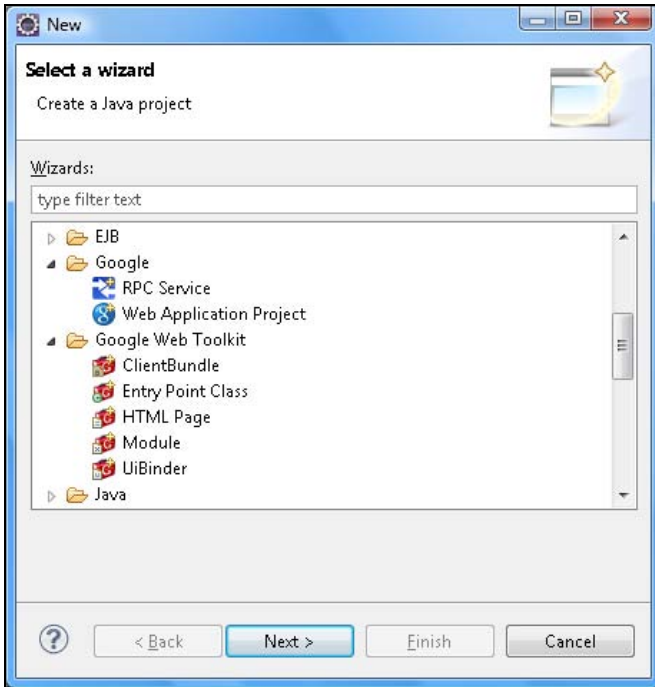


Рис. 1.4. Разделы Google и Google Web Toolkit  
окна команды New | Other меню File

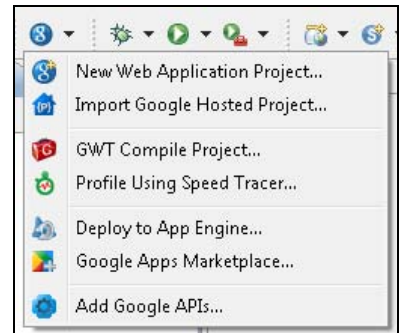


Рис. 1.5. Команды кнопки  
Google Services  
and Development Tools

Кнопка **Sign in to Google** позволяет пройти аутентификацию/авторизацию для доступа к таким Google-сервисам из среды Eclipse, как Project Hosting, Google SQL Service, Google Apps Marketplace и Google App Engine.

Сервис Project Hosting (<http://code.google.com/hosting/>) предоставляет возможность размещения открытых проектов в удаленном VCS-репозитории (Version Control System) для организации командной работы над ними. Данный сервис поддерживает такие системы контроля версий, как Subversion, Mercurial и Git.

Сервис Google SQL Service представляет собой Web-сервис, обеспечивающий создание, администрирование и использование MySQL базы данных для GAE-приложения.

Сервис Google Apps Marketplace (<http://www.google.com/enterprise/marketplace/>) является магазином Web-приложений, которые интегрируются с бизнес-приложением Google Apps (<http://www.google.com/enterprise/apps/business/>).

Раскрытая кнопка **Google Services and Development Tools** показывает набор команд (рис. 1.5):

- ◆ **New Web Application Project** — запускает мастер создания проекта приложения платформ GAE и GWT;
- ◆ **Import Google Hosted Project** — позволяет импортировать проект, размещенный на хостинге сервиса Project Hosting, в Workspace-пространство среды Eclipse;



- ◆ **GWT Compile Project** — запускает мастер компиляции Java-кода проекта GWT-приложения в JavaScript-код;
- ◆ **Profile Using Speed Tracer** — обеспечивает анализ производительности приложения с помощью инструмента Speed Tracer — расширения Web-браузера Google Chrome;
- ◆ **Deploy to App Engine** — запускает мастер развертывания приложения на GAE-платформе;
- ◆ **Google Apps Marketplace** — обеспечивает загрузку приложения в магазин Google Apps Marketplace;
- ◆ **Add Google APIs** — позволяет добавить в путь приложения программный интерфейс API различных Google-сервисов.

Раздел **Google** команды **New | Other** меню **File** среды Eclipse содержит мастера:

- ◆ **RPC Service** — на основе Entity-компонентов создает для GWT-приложения RequestFactory-сервис данных;
- ◆ **Web Application Project** — создает проекты GAE- и GWT-приложений.

Раздел **Google Web Toolkit** команды **New | Other** меню **File** среды Eclipse содержит мастера:

- ◆ **ClientBundle** — создает для GWT-приложения ClientBundle-интерфейс, группирующий ресурсы приложения, такие как изображения, CSS-стили, текстовые файлы и др., в единый модуль, что ускоряет загрузку и запуск клиентской части приложения;
- ◆ **Entry Point Class** — создает для GWT-приложения EntryPoint-класс — точку входа в GWT-модуль;
- ◆ **HTML Page** — создает для GWT-приложения HTML-страницу, обеспечивающую загрузку клиентской части приложения;
- ◆ **Module** — создает для GWT-приложения новый GWT-модуль;
- ◆ **UiBinder** — создает для GWT-приложения UiBinder-шаблон, декларативно определяющий GUI-интерфейс приложения.

Для использования в GAE-проекте страниц Java Server Pages (JSP) требуется Java-компилятор `javac.exe`, содержащийся в наборе Java SE Development Kit (JDK). По умолчанию среда Eclipse использует не набор JDK, а среду выполнения JRE, поэтому после установки плагинов GPE и GAE Java SDK, перед созданием GAE-проекта, в меню **Windows** среды Eclipse выберем команду **Preferences**, в диалоговом окне которой откроем раздел **Java | Installed JREs** и кнопкой **Add** добавим предварительно установленный набор JDK (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>) (рис. 1.6).

В настоящее время GAE-платформа поддерживает версии Java 5 и 6, поэтому необходимо проверить, чтобы уровень компиляции, установленный в разделе **Java | Compiler** диалогового окна команды **Preferences**, а также версия Java раздела **Project Facets** диалогового окна свойств GAE-проекта соответствовали версии Java раздела **Java | Installed JREs**.

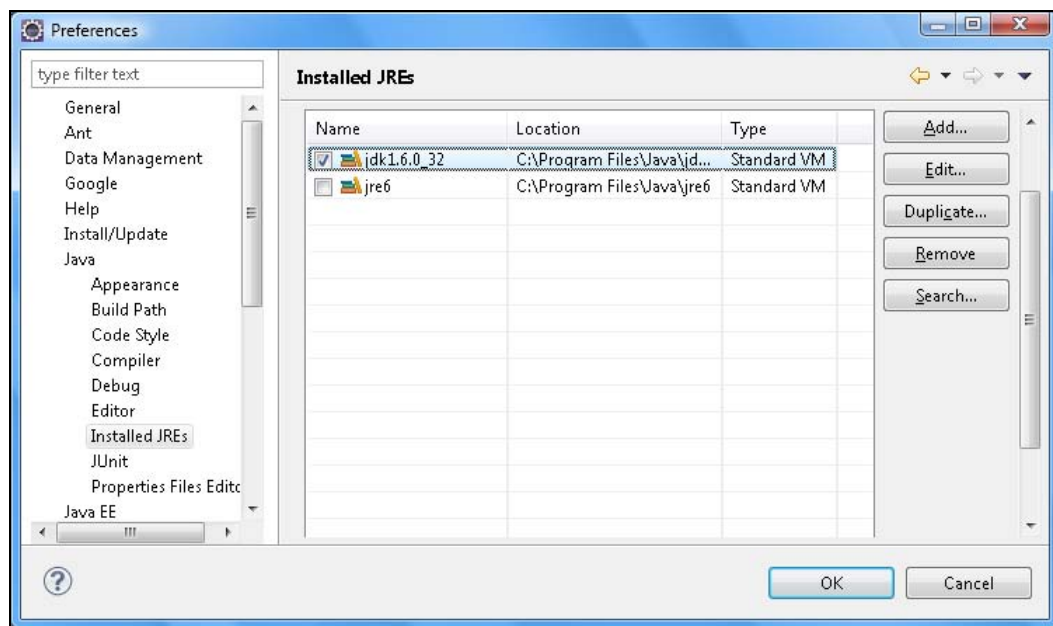


Рис. 1.6. Установка использования JDK по умолчанию

## Создание проекта приложения

Теперь все готово для создания проекта GAE-приложения. В меню **File** среды Eclipse выберем команду **New | Other | Google | Web Application Project** и нажмем кнопку **Next**. В окне мастера создания проекта, в поле **Project name** введем имя проекта, в поле **Package** — имя пакета проекта, отметим флажок **Use Google App Engine** и нажмем кнопку **Finish** (рис. 1.7).

В результате в Workspace-пространстве среды Eclipse будет сгенерирована основа проекта GAE-приложения.

В путь GAE-проекта будут добавлены библиотеки среды JRE и набора GAE Java SDK для локального запуска GAE-приложения, предоставляющего Servlet-контейнер Web-сервера Jetty и обеспечивающего эмуляцию работы GAE-служб.

Папка `src` сгенерированного GAE-проекта содержит пакет с классом сервлета, который расширяет класс `javax.servlet.http.HttpServlet` и отвечает за обработку HTTP-запросов к приложению (листинг 1.1). Сгенерированный Servlet-класс переопределяет метод `doGet()` класса `HttpServlet`, обрабатывающий HTTP GET-запрос, в котором в ответ на запрос записывается строка "Hello, world". Для вызова данного сервлета его класс указан в дескрипторе `web.xml` развертывания приложения с помощью тега `<servlet-class>`, при этом тег `<url-pattern>` указывает относительный URL-адрес вызова сервлета. Дескриптор `web.xml` располагается в папке `war\WEB-INF` GAE-проекта. Тег `<welcome-file>` дескриптора `web.xml` определяет страницу приветствия приложения.

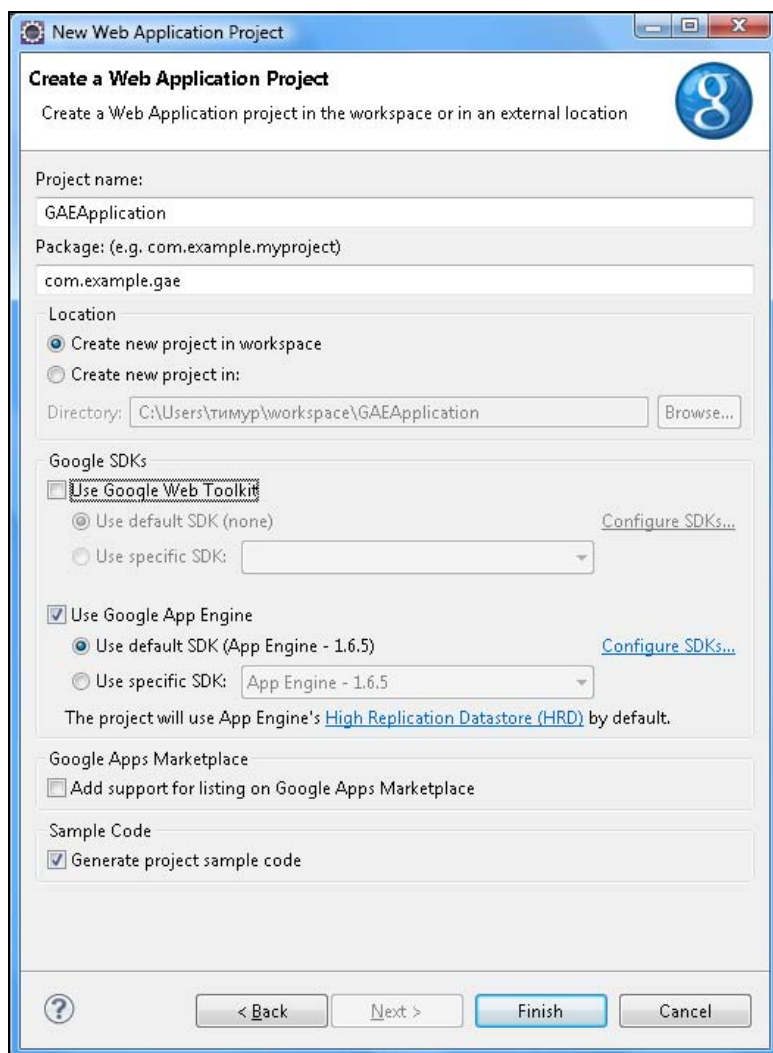


Рис. 1.7. GPE-мастер создания проекта

**Листинг 1.1. Код Servlet-класса GAE-приложения**

```
package com.example.gaeapplication;
import java.io.IOException;
import javax.servlet.http.*;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        resp.setContentType("text/plain");
        resp.getWriter().println("Hello, world");
    }
}
```

Сгенерированная страница `index.html` приветствия приложения каталога `war` GAE-проекта содержит ссылку на относительный URL-адрес вызова сервлета, указанный в теге `<url-pattern>` дескриптора `web.xml` (листинг 1.2).

**Листинг 1.2. Код страницы `index.html` приветствия GAE-приложения**

```
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  <title>Hello App Engine</title>
</head>
<body>
  <h1>Hello App Engine!</h1>
  <table>
    <tr>
      <td colspan="2" style="font-weight:bold;">Available Servlets:</td>
    </tr>
    <tr>
      <td><a href="gaeapplication">GAEApplication</a></td>
    </tr>
  </table>
</body>
</html>
```

Папка `src\META-INF` содержит конфигурационный файл `jdoconfig.xml`, определяющий для приложения конфигурацию механизма Java Data Objects (JDO) сохранения объектов в базе данных.

Каталог `war` GAE-проекта содержит все файлы, необходимые для развертывания приложения в Servlet-контейнере, поэтому в папку `war\WEB-INF\lib` включены библиотеки программного интерфейса API GAE-служб.

Конфигурационный файл `appengine-web.xml` папки `war\WEB-INF` GAE-проекта содержит настройки развертывания приложения на GAE-платформе. Файл `appengine-web.xml` указывает зарегистрированный GAE-идентификатор приложения, версию приложения, а также его список статических и ресурсных файлов и другие настройки работы приложения.

Изначально сгенерированный файл `appengine-web.xml` в теге `<application>` содержит пустой GAE-идентификатор, и для загрузки приложения в облако платформы GAE необходимо заполнить тег `<application>` GAE-идентификатором, полученным в результате регистрации приложения с помощью GAE-консоли **<https://appengine.google.com/>**.

Версия приложения указывается в теге `<version>` файла `appengine-web.xml` и необходима для создания новой версии приложения в облаке GAE-платформы или для замены приложения с тем же номером версии.

Для повышения эффективности GAE-платформа обслуживает отдельно статические файлы приложения, такие как изображения, CSS-стили, JavaScript-файлы

и др., которые могут обслуживаться простым HTTP-сервером, а не сервером приложений, вызывающим сервлеты и обрабатывающим JSP-страницы.

По умолчанию все файлы каталога war приложения, за исключением JSP-файлов и файлов папки WEB-INF, рассматриваются GAE-платформой как статические файлы. Отдельно указать статические файлы можно в файле appengine-web.xml с помощью тега:

```
<static-files>
  <include path="" expiration="" />
  <exclude path="" />
</static-files>
```

Здесь тег `<include>` указывает путь для включения файлов в список статических файлов, а тег `<exclude>` — путь для исключения файлов из списка статических файлов приложения. Атрибут `expiration` указывает время кэширования браузером статических файлов (по умолчанию 10 мин) в формате `d` (дни) `h` (часы) `m` (минуты) `s` (секунды).

Тег `<public-root>` файла appengine-web.xml позволяет переопределить корневой каталог для статических файлов приложения.

Все остальные файлы WAR-каталога приложения считаются ресурсными файлами и обслуживаются сервером приложений. Уточнить список ресурсных файлов приложения можно в файле appengine-web.xml с помощью тега:

```
<resource-files>
  <include path="" />
  <exclude path="" />
</resource-files>
```

Здесь тег `<include>` указывает путь для включения файлов в список ресурсных файлов, а тег `<exclude>` — путь для исключения файлов из списка ресурсных файлов приложения.

Указать кодировку приложения по умолчанию можно в файле appengine-web.xml с помощью тега:

```
<env-variables>
  <env-var name="DEFAULT_ENCODING" value="" />
</env-variables>
```

Файл appengine-web.xml может содержать также следующие теги.

- ◆ `<system-properties>` — посредством дочерних тегов `<property name="" value="" />` определяет системные свойства, которые могут быть доступны из Java-кода с помощью объекта `java.util.Properties`, получаемого методом `System.getProperties()`. Сгенерированный файл appengine-web.xml содержит свойство `java.util.logging.config.file` со значением `WEB-INF/logging.properties`, указывающим расположение конфигурационного файла для модуля ведения журнала `java.util.logging`. В конфигурационном файле `logging.properties` установлен уровень ведения журнала по умолчанию — `.level = WARNING`. Просматривать журнал приложения можно в консоли администрирования <https://>

**appengine.google.com/** в разделе **Main | Logs**. Выводить свои сообщения в журнал можно с помощью Java-кода:

```
private static final Logger log = Logger.getLogger([имя класса].class.getName());
log.info("");
log.warning("");
log.severe("");
```

Кроме того, GAE-платформа сама устанавливает ряд системных свойств при запуске приложения и JVM: `com.google.appengine.runtime.environment`, `com.google.appengine.runtime.version`, `com.google.appengine.application.id`, `file.separator`, `path.separator`, `line.separator`, `java.version`, `java.vendor`, `java.vendor.url`, `java.class.version`, `java.specification.version`, `java.specification.vendor`, `java.specification.name`, `java.vm.vendor`, `java.vm.name`, `java.vm.specification.version`, `java.vm.specification.vendor`, `java.vm.specification.name`, `user.dir`.

- ◆ `<ssl-enabled>` — с помощью значения `false` отключает использование для приложения протокола HTTPS.
- ◆ `<sessions-enabled>` — с помощью значения `true` включает использование сессий. При этом с помощью тега `<async-session-persistence enabled="true"/>` можно определить асинхронную запись данных сессии в хранилище.
- ◆ `<inbound-services>` — посредством дочерних тегов `<service>[служба]</service>` включает использование GAE-служб:
  - `channel_presence` — приложение получает уведомления о присоединении или отсоединении клиента от Channel-канала;
  - `mail` — приложение может получать электронные письма;
  - `xmpp_message` — приложение может получать мгновенные сообщения. При этом включение службы `xmpp_presence` позволяет приложению получать уведомления об изменении статуса клиента в службе XMPP, включение службы `xmpp_subscribe` — получать уведомления об изменении статуса подписки на сообщения между клиентом и приложением, включение службы `xmpp_error` позволяет приложению получать сообщения об ошибках XMPP-службы;
  - `warmup` — служба посылает запрос для предварительной инициализации нового экземпляра приложения до получения им клиентского запроса, что сокращает время ожидания клиентом ответа от нового экземпляра приложения. GAE-платформа создает новый экземпляр приложения, например, при увеличении количества клиентских запросов к существующему экземпляру приложения.
- ◆ `<warmup-requests-enabled>` — с помощью значения `false` отключает `warmup`-запросы.
- ◆ `<precompilation-enabled>` — с помощью значения `false` отключает оптимизацию загрузки классов. Служба `warmup` работает с приложением, имеющим постоян-

ный трафик с клиентом. Если же клиент посылает запрос к неработавшему приложению, этот запрос, называемый *loading-запросом*, инициализирует экземпляр приложения, и клиент будет ожидать некоторое время ответ на свой первый запрос. *Precompilation-оптимизация* ускоряет работу *loading-запроса* как минимум на 30%.

- ◆ `<admin-console>` — с помощью дочерних тегов `<page name="" url="" />` добавляет страницы администрирования приложением в консоль администрирования <https://appengine.google.com/>.
- ◆ `<static-error-handlers>` — с помощью дочерних тегов `<handler file="" error-code="" />` переопределяет страницы отображения ошибки клиенту по умолчанию, где код ошибки `error-code` может быть `over_quota`, `dos_api_denial` и `timeout`.
- ◆ `<threadsafe>` — с помощью значения `true` включает доставку параллельных запросов к серверу.

Перед запуском GAE-приложения из среды Eclipse удостоверимся, что версия Java раздела **Project Facets** диалогового окна свойств GAE-проекта соответствует версии среды JRE проекта. Для этого в окне **Project Explorer** щелкнем правой кнопкой мыши на узле проекта, в контекстном меню выберем команду **Properties** и раздел **Project Facets** ее диалогового окна (рис. 1.8).

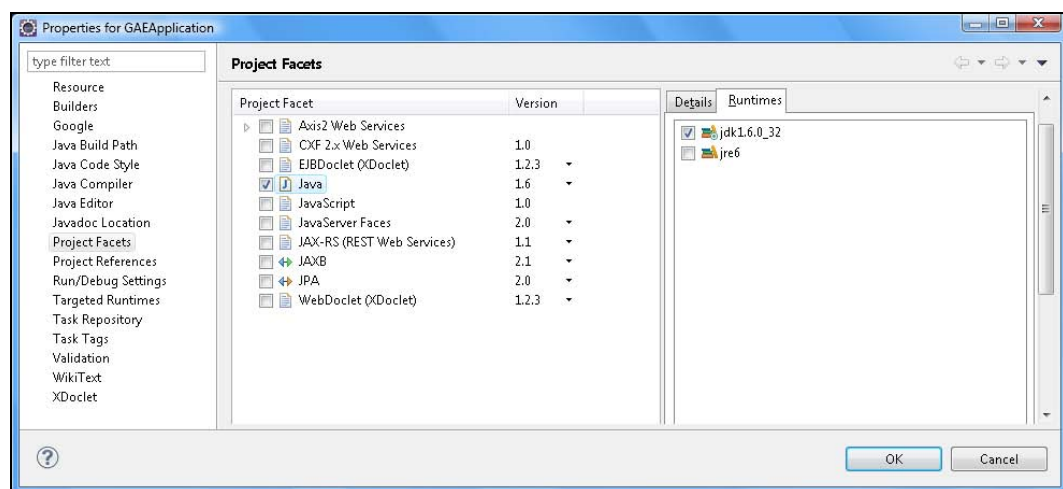


Рис. 1.8. Установка версии Java проекта GAE-приложения

## Запуск приложения из среды Eclipse

Для запуска GAE-приложения из среды Eclipse в окне **Project Explorer** щелкнем правой кнопкой мыши на узле проекта и в контекстном меню выберем команду **Run As | Web Application**. В результате в окно **Console** среды Eclipse будет выведено сообщение:

```
... AM com.google.apphosting.utils.jetty.JettyLogger info
INFO: Logging to JettyLogger(null) via com.google.apphosting.utils.jetty.JettyLogger
... AM com.google.apphosting.utils.config.AppEngineWebXmlReader readAppEngineWebXml
INFO: Successfully processed ...\\workspace\\GAEApplication\\war\\WEB-INF\\appengine-web.xml
... AM com.google.apphosting.utils.config.AbstractConfigXmlReader readConfigXml
INFO: Successfully processed ...\\workspace\\GAEApplication\\war\\WEB-INF\\web.xml
... PM com.google.appengine.tools.development.DevAppServerImpl start
INFO: The server is running at http://localhost:8888/
... com.google.appengine.tools.development.DevAppServerImpl start
INFO: The admin console is running at http://localhost:8888/_ah/admin
```

В выведенном сообщении указан адрес локального вызова приложения **http://localhost:8888/** и адрес локальной консоли администрирования **http://localhost:8888/\_ah/admin**.

Набрав в адресной строке Web-браузера адрес **http://localhost:8888/**, можно будет увидеть страницу приветствия приложения со ссылкой вызова сервлета приложения (рис. 1.9).

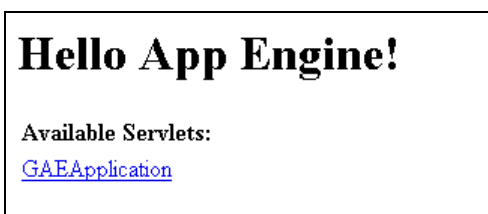


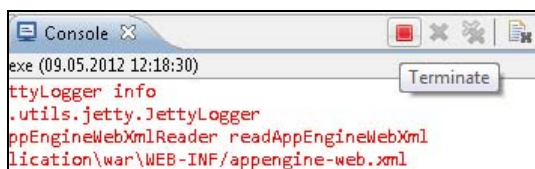
Рис. 1.9. Страница приветствия GAE-приложения

Набрав в адресной строке Web-браузера адрес **http://localhost:8888/\_ah/admin**, можно будет увидеть страницу консоли администрирования (рис. 1.10).



Рис. 1.10. Страница локальной консоли администрирования GAE-приложения



Рис. 1.11. Остановка GAE-приложения кнопкой **Terminate**

Остановить работающий сервер с GAE-приложением можно, нажав кнопку **Terminate** окна **Console** среды Eclipse (рис. 1.11).

## Развертывание приложения на платформе App Engine

### Регистрация приложения

Для того чтобы развернуть приложение на платформе GAE, его необходимо предварительно зарегистрировать. Для регистрации GAE-приложения откроем Web-браузер и попробуем войти в консоль администрирования по адресу <https://appengine.google.com/>. Однако сервис консоли администрирования является защищенным сервисом и его аутентификацию/авторизацию проходят лишь пользователи, имеющие свой Google-аккаунт. Поэтому предварительно откроется страница, приглашающая ввести логин/пароль Google-аккаунта или зарегистрировать его (рис. 1.12).

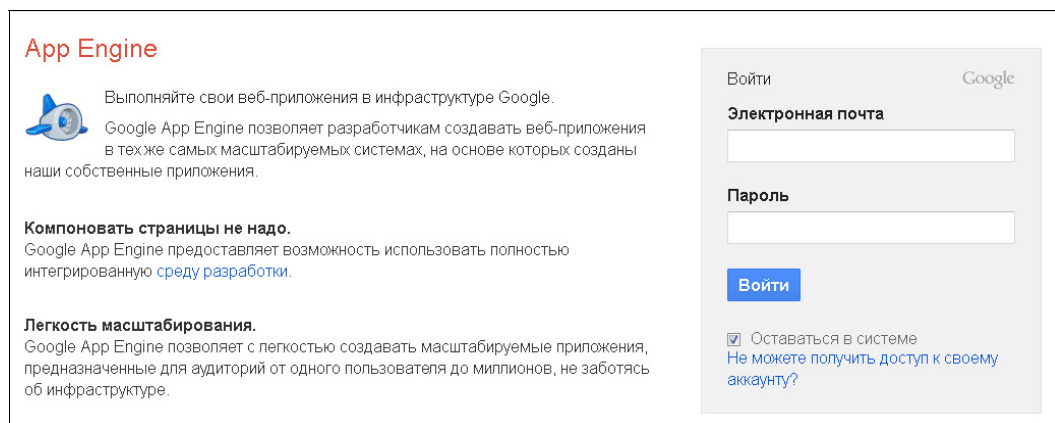


Рис. 1.12. Страница аутентификации консоли администрирования

После ввода логина/пароля и нажатия кнопки **Войти** откроется страница консоли администрирования.

Кнопка **Create Application** консоли администрирования позволяет зарегистрировать новое приложение. Страница, открывающаяся при нажатии кнопки **Create Application**, отображает уведомление об оставшемся количестве приложений, ко-

торые можно еще зарегистрировать на данный Google-аккаунт: максимальное число — 10. Поле **Application Identifier** предлагает ввести идентификатор приложения, при этом кнопка **Check Availability** позволяет проверить его уникальность. Поле **Application Title** предлагает ввести отображаемый при аутентификации заголовок приложения.

Раздел **Authentication Options (Advanced)** страницы регистрации GAE-приложения с помощью переключателей позволяет выбрать систему аутентификации пользователей, которая может быть использована GAE-приложением:

- ◆ **Open to all Google Accounts users (default)** — аутентификация с помощью Google-аккаунта;
- ◆ **Restricted to the following Google Apps domain** — аутентификация с помощью аккаунта домена бизнес-приложения Google Apps;
- ◆ **(Experimental) Open to all users with an OpenID Provider** — аутентификация с помощью идентификатора OpenID.

Раздел **Storage Options (Advanced)** страницы регистрации GAE-приложения с помощью переключателей **High Replication (default)** и **Master/Slave** позволяет выбрать тип хранилища данных приложения — распределенное хранилище или главное/подчиненное хранилище (в настоящее время не рекомендуется). Данный раздел после отмены хранилища Master/Slave был исключен.

Нажатие кнопки **Create Application** страницы регистрации GAE-приложения завершает регистрацию приложения.

Теперь при входе в консоль администрирования <https://appengine.google.com/> ее страница будет содержать ссылку на зарегистрированное приложение, при щелчке по которой откроется страница администрирования приложением.

## Страница администрирования приложением

Левая панель страницы администрирования приложением содержит разделы **Main**, **Data**, **Administration**, **Billing** и **Resources**.

Раздел **Main** содержит подразделы **Dashboard**, **Instances**, **Logs**, **Versions**, **Backends**, **Cron Jobs**, **Task Queues**, **Quota Details**.

Раздел **Dashboard** показывает общую информацию о работе приложения в App Engine, включающую в себя набор графиков потребления ресурсов приложением, информацию о работающих экземплярах приложения, статус оплаты хостинга приложения, таблицу потребления ресурсов приложением в пределах установленной квоты (<https://developers.google.com/appengine/docs/quotas>), информацию о URI-запросах к приложению и об их ошибках. Набор графиков потребления ресурсов приложением состоит из следующих графиков:

- ◆ **Requests/Second** — количество запросов в секунду в зависимости от времени, включая динамические, статические и кэшированные запросы;
- ◆ **Requests by Type/Second** — количество запросов в секунду в зависимости от времени с дифференциацией по типам запросов: динамическим, статическим (к статическим ресурсам приложения) и кэшированным;

- ◆ **Milliseconds/Request** — время ожидания ответа от приложения для динамического запроса в зависимости от времени;
- ◆ **Errors/Second** — количество ошибок приложения в секунду в зависимости от времени;
- ◆ **Bytes Received/Second** — количество байтов запроса в секунду в зависимости от времени;
- ◆ **Bytes Sent/Second** — количество байтов ответа в секунду в зависимости от времени;
- ◆ **CPU Seconds Used/Second** — загруженность процессора в зависимости от времени с дифференциацией по общей загруженности и загруженности, связанной с использованием API GAE-платформы;
- ◆ **Milliseconds Used/Second** — обработка запроса приложением в зависимости от времени;
- ◆ **Number of Quota DenialsSecond** — количество отклонений запросов к приложению из-за превышения установленной квоты (<https://developers.google.com/appengine/docs/quotas>);
- ◆ **Instances** — работа экземпляров приложения в зависимости от времени с дифференциацией по активным экземплярам и экземплярам учитываемых квот;
- ◆ **Memory Usage (MB)** — использование памяти в зависимости от времени.

Раздел **Instances** показывает информацию о работе экземпляров приложения — количество созданных экземпляров приложения, среднее количество запросов в секунду, среднюю задержку ответа и среднее количество памяти, занимаемой экземпляром. Для каждого экземпляра приложения отображается среднее количество запросов в секунду за последнюю минуту, средняя задержка ответа за последнюю минуту, количество успешно обработанных запросов, количество ошибок, время жизни экземпляра, количество памяти, ссылка на журнал, тип экземпляра — динамический или резидентный и кнопка **Shutdown** уничтожения экземпляра.

Экземпляр приложения состоит из работающего экземпляра среды выполнения, подключенных библиотек, кода приложения и памяти, выделенной для работы. Наличие нескольких экземпляров приложения обеспечивает параллельную обработку входящих запросов, при этом за создание новых экземпляров приложения при увеличении количества входящих запросов отвечает планировщик GAE-платформы. Повлиять на работу планировщика GAE-платформы можно с помощью установок раздела **Administration** | **Application Settings** | **Performance** консоли администрирования.

Экземпляр приложения называется *динамическим*, если его жизненный цикл регулируется планировщиком GAE-платформы, или *резидентным*, если работает все время, при этом за его создание отвечает разработчик — администратор приложения. Экземпляр приложения может быть, помимо динамического или резидентного, *внешним* (Frontend) или *фоновым* (Backend). Frontend-экземпляры по умолчанию обрабатывают входящие запросы, и их работа ограничена установленной квотой (<https://developers.google.com/appengine/docs/quotas>). Backend-экземпляры, по

сравнению с Frontend-экземплярами, не ограничены 30-секундным периодом обработки запроса и могут использовать большую память (до 1 Гбайт) и больше загружать процессор (до 4,8 ГГц). Соответственно работа Frontend- и Backend-экземпляров приложения отдельно учитывается в использовании установленной квоты. Создать резидентные Frontend-экземпляры можно, только если приложение имеет платный хостинг и с помощью установки **min idle instances** раздела **Administration | Application Settings | Performance** консоли администрирования. Backend-экземпляры приложения связаны со специальным Backend-сервером, создаваемым для приложения GAE-платформой. Приложение может иметь несколько Backend-серверов, с каждым из которых может быть связано несколько Backend-экземпляров. Свойства Backend-серверов приложения определяются в его конфигурационном `backends`-файле. В отличие от динамических Frontend-экземпляров, Backend-экземпляры не создаются автоматически планировщиком GAE-платформы, он может лишь запускать и останавливать динамический Backend-сервер, с которым связано определенное число Backend-экземпляров. Backend-серверы назначаются резидентными (по умолчанию) или динамическими с помощью конфигурационного `backends`-файла приложения.

Раздел **Logs** отображает записи в журнале приложения с возможностью фильтрации записей по уровню (списки **Debug**, **Info**, **Warning**, **Error**, **Critical**) и датам (раздел **Options**).

Раздел **Versions** показывает развернутые версии приложения. Версия приложения указывается в теге `<version>` конфигурационного файла `appengine-web.xml` приложения. При загрузке приложения с уже существующим номером версии код развернутого приложения заменяется, при загрузке приложения с другим номером версии создается новая версия приложения. Сделать запрос к конкретной версии приложения можно с помощью URL-адреса вида **`http://[номер_версии].[идентификатор_приложения].appspot.com/`**. При наличии нескольких версий приложения в верхней части консоли администрирования появится раскрывающийся список с номерами версий, позволяющий отображать информацию для конкретной версии приложения.

Кнопка **Make Default** раздела **Versions** позволяет назначить версию приложения версией по умолчанию, принимающей входящие запросы к приложению, кнопка **Delete** удаляет выбранную версию приложения.

С помощью кнопки **Add Traffic Split** раскрывающегося подраздела **Traffic Splitting** раздела **Versions** можно разделить входящий трафик по версиям приложения для тестирования пользовательских предпочтений.

Раздел **Backends** отображает информацию о работе Backend-серверов приложения: тип сервера — динамический или резидентный, время и Google-аккаунт развертывания сервера, класс классификации сервера, количество его Backend-экземпляров, опции, кнопки запуска, остановки и уничтожения сервера. Квота потребления ресурсов Backend-сервером определяется его классом — B1 (128 Мбайт, 600 МГц), B2 (по умолчанию, 256 Мбайт, 1,2 ГГц), B4 (512 Мбайт, 2,4 ГГц), B8 (1024 Мбайт, 4,8 ГГц). Свойства Backend-сервера определяются в конфигурационном `backends`-

файле приложения. Использование Backend-серверов приложением ограничено их общим количеством 5, с каждым из которых может быть связано не более 20 Backend-экземпляров, и общей используемой памятью не более 10 Гбайт.

Раздел **Cron Jobs** показывает запланированные задачи приложения, представляющие собой HTTP GET-вызовы определенных URL-адресов в заданное время. Вызываемые URL-адреса и их расписания вызова определяются в конфигурационном cron-файле приложения. Количество запланированных задач приложения ограничено 20.

Раздел **Task Queues** позволяет управлять очередями задач приложения, определенными в конфигурационном queue-файле приложения. Приложение в своем коде может добавлять задачи, представляющие собой фоновые HTTP-запросы, в очереди, объявленные в конфигурационном queue-файле приложения, а разработчик — администратор приложения в консоли администрирования может приостановить очередь, немедленно выполнить отдельную задачу, удалить задачу или всю очередь.

Раздел **Quota Details** показывает квоту использования ресурсов GAE-платформы, установленную для приложения. При превышении установленной квоты отдельного ресурса он станет недоступным для приложения до тех пор, пока квота не пополнится. Так, при превышении дневной квоты, она обновится в полночь по тихоокеанскому времени. При превышении квоты, связанной с ресурсами инициализации запроса к приложению, запрос возвратит код HTTP 403, при превышении квоты, связанной с другими ресурсами, будет сгенерировано исключение.

Раздел **Data** консоли администрирования содержит подразделы **Datastore Indexes**, **Datastore Viewer**, **Datastore Statistics**, **Blob Viewer**, **Prospective Search**, **Text Search**, **Datastore Admin**, **Memcache Viewer**.

Раздел **Datastore Indexes** показывает информацию о indexes-таблицах приложения. Индексы — это таблицы, содержащие результаты запросов приложения к Datastore-хранилищу GAE-платформы. GAE-платформа использует indexes-таблицы для быстрой обработки запросов приложения к Datastore-хранилищу данных. О требуемых для запросов приложения индексах GAE-платформа узнает при развертывании приложения из его конфигурационного datastore-indexes-файла, который создается автоматически при разработке приложения в процессе его тестирования.

Раздел **Datastore Viewer** позволяет просматривать данные приложения, хранящиеся в Datastore-хранилище GAE-платформы.

Раздел **Datastore Statistics** показывает информацию об использовании приложением ресурсов Datastore-хранилища GAE-платформы.

Раздел **Blob Viewer** позволяет просматривать данные приложения, хранящиеся в Blobstore-хранилище GAE-платформы.

Раздел **Prospective Search** показывает информацию о зарегистрированных приложением запросах на данные, которые соответствуют определенным критериям. Программный интерфейс Datastore API позволяет осуществлять запросы к статическим данным хранилища. Служба Prospective Search дает возможность зарегистри-

ровать запрос на данные, которые еще не были внесены в систему и добавление которых инициирует добавление соответствующей задачи в очередь Task Queues.

Раздел **Text Search** показывает индексы полнотекстового поиска приложения, использующего программный интерфейс Search API GAE-платформы. Программный интерфейс Search API позволяет организовать хранение и поиск документов аналогично поисковой машине Google.

Раздел **Datastore Admin** обеспечивает копирование и восстановление данных приложения.

Раздел **Memcache Viewer** дает возможность посмотреть статистику использования приложением Memcache-хранилища GAE-платформы, его содержимое и очистить кэш кнопкой **Flush Cache**.

Раздел **Administration** консоли администрирования содержит подразделы **Application Settings**, **Permissions**, **Blacklist**, **Admin Logs**.

Раздел **Application Settings** позволяет изменить настройки работы приложения:

- ◆ **Application Title** — заголовок приложения, отображаемый при аутентификации пользователя;
- ◆ **Cookie Expiration** — время жизни cookies, используемых для аутентификации пользователя приложения;
- ◆ **Authentication Options** — метод аутентификации пользователя приложения — с помощью Google-аккаунта или OpenID-идентификатора;
- ◆ **Frontend Instance Class** — класс экземпляров приложения, определяющий ограничения в использовании памяти и загрузке процессора: F1 (по умолчанию, 128 Мбайт, 600 МГц), F2 (256 Мбайт, 1,2 ГГц), F4 (512 Мбайт, 2,4 ГГц);
- ◆ **Max Idle Instances** — шкала, с помощью которой контролируется максимальное число работающих экземпляров приложения, создаваемых планировщиком GAE-платформы в ответ на увеличение количества входящих запросов. Количество работающих экземпляров приложения влияет на скорость расходования установленной квоты потребления ресурсов GAE-платформы;
- ◆ **Min Pending Latency** — шкала, с помощью которой контролируется минимальное время ожидания запроса в очереди экземпляра. В случае увеличения минимального времени ожидания запроса входящие запросы скорее будут помещаться в очереди запросов экземпляров, чем планировщик будет создавать новые экземпляры приложения;
- ◆ **Logs Retention** — максимальное число дней хранения журнала приложения и максимальный размер журнала приложения (для платного хостинга);
- ◆ **Built-ins** — с помощью кнопки включается и выключается консоль Datastore Admin копирования и восстановления данных приложения;
- ◆ **Domain Setup** — с помощью кнопки **Add Domain** позволяет связать приложение с доменом, зарегистрированным в бизнес-приложении Google Apps;
- ◆ **Disable Datastore Writes** — с помощью кнопки включает или выключает запись данных в Datastore-хранилище;

- ◆ **Disable or Delete Application** — отключает приложение с данным идентификатором с возможностью последующего удаления приложения. После удаления приложения его идентификатор все равно остается зарезервированным, и нельзя будет зарегистрировать новое приложение с тем же идентификатором;
- ◆ **Duplicate Application Settings** — регистрация нового приложения с текущими настройками.

Раздел **Permissions** с помощью ссылки **permanently prohibit code downloads** дает возможность запретить загрузку кода приложения, а также посредством поля **Email** и раскрывающегося списка **Role** позволяет дать разрешение на доступ к консоли администрирования другим пользователям со следующими ролями:

- ◆ **Owner** — полный доступ к консоли администрирования;
- ◆ **Developer** — доступ к консоли администрирования, за исключением изменений в разделе **Permissions**;
- ◆ **Viewer** — только просмотр.

Раздел **Blacklist** отображает "черный" список IP-адресов и подсетей, определенный в конфигурационном dos-файле приложения для предотвращения DoS-атак. Запросы с данных адресов и подсетей будут отклоняться до вызова кода приложения.

Раздел **Admin Logs** представляет журнал действий в консоли администрирования.

Раздел **Billing** консоли администрирования содержит подразделы **Billing Settings** и **Billing History**.

Раздел **Billing Settings** с помощью кнопки **Enable Billing** позволяет перевести хостинг приложения в статус платного с определением ежедневного бюджета.

Раздел **Billing History** представляет журнал ежедневного использования и оплаты ресурсов GAE-платформы.

Раздел **Resources** консоли администрирования с подразделами **Documentation**, **FAQ**, **Developer Forum**, **Downloads** и **System Status** обеспечивает доступ к документации, часто задаваемым вопросам, форуму, загрузкам и текущему состоянию GAE-платформы.

## Загрузка приложения в App Engine

После регистрации приложения для его загрузки в App Engine в среде Eclipse откроем в редакторе кода файл `appengine-web.xml` папки `war\WEB-INF` GAE-проекта и в тег `<application>` вставим зарегистрированный идентификатор приложения. Сохранив сделанные изменения, с помощью кнопки **Sign in to Google** пройдем аутентификацию/авторизацию для доступа к Google-сервисам, в окне **Project Explorer** щелкнем правой кнопкой мыши на узле проекта и в контекстном меню выберем команду **Google | Deploy to App Engine**, в диалоговом окне мастера нажмем кнопку **Deploy** (рис. 1.13) — в результате приложение будет развернуто на GAE-платформе.

В Web-браузере откроем консоль администрирования <https://appengine.google.com/> GAE-платформы, в разделе **Application** которой щелчком по ссылке приложения

откроем страницу администрирования приложения. В разделе **Application Settings** | **Application Default Version URL** страницы администрирования приложением будет указан URL-адрес запроса к приложению, открывающий его страницу приветствия.

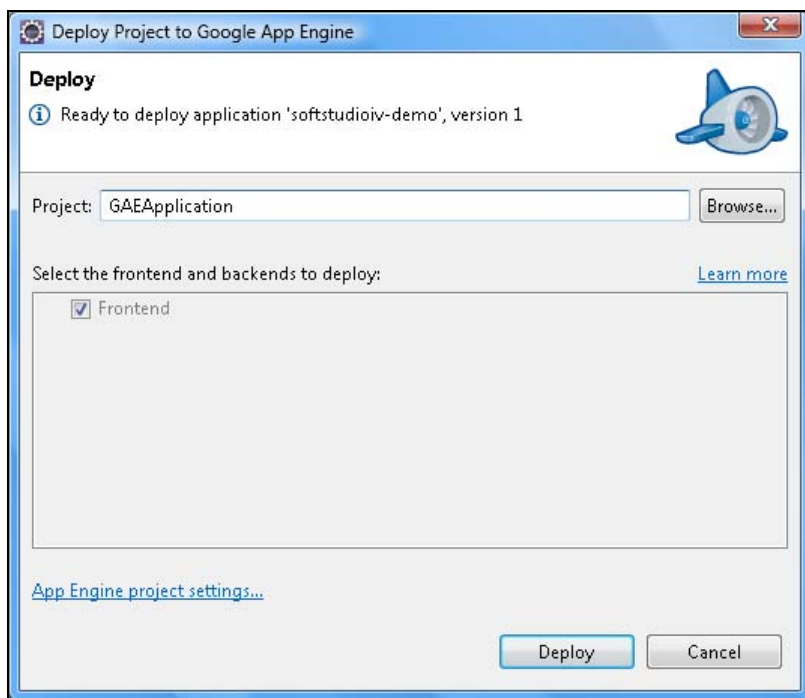


Рис. 1.13. Загрузка приложения из среды Eclipse в App Engine

## Создание пользовательских разделов консоли администрирования

Настройка конфигурационного файла `appengine-web.xml` приложения:

```
<admin-console>
  <page name="Custom Admin" url="/admin/customAdmin.jsp" />
</admin-console>
```

позволяет включить в страницу администрирования приложением собственные разделы.

Значение атрибута `name` тега `<page>` определяет имя подраздела, который появится в левой панели страницы администрирования в разделе **Custom**, находящемся ниже раздела **Billing**. Значение атрибута `url` указывает путь к ресурсу, открывающемуся в правом фрейме страницы администрирования при нажатии на ссылку подраздела.

Так как такие пользовательские страницы администрирования предназначены для использования администратором приложения, необходимо ограничить к ним дос-



туп для обычных пользователей. Сделать это можно с помощью настройки дескриптора web.xml приложения:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

## ГЛАВА 2



# Журналирование приложения

При разработке Web-приложений вывод и сохранение сообщений приложения на стороне сервера — важная часть процесса отладки и контроля работы приложения.

Платформа App Engine сама по себе автоматически обеспечивает журналирование входящих запросов к приложению, записывая время и дату запроса, его URL-адрес, HTTP-код ответа, время ожидания ответа, размер ответа, информацию о клиентском приложении, IP-адрес клиента, время загрузки процессора при обработке запроса, время загрузки процессора при вызове GAE API, оценочную стоимость 1000 таких запросов в долларах и др. Посмотреть журнал приложения можно в разделе **Main | Logs** консоли администрирования <https://appengine.google.com/> или загрузив журнал с помощью инструмента командной строки `appcfg.sh` набора GAE SDK.

Записи журнала приложения имеют двухуровневую структуру — в запись уровня запроса, содержащую информацию о запросе к приложению, вложены записи уровня приложения, содержащие информацию о работе экземпляра приложения, обрабатывающего запрос.

Раздел **Main | Logs** консоли администрирования обеспечивает фильтрацию записей журнала по уровню с помощью переключателя **Logs with minimum severity** и раскрывающегося списка, позволяющего выбрать уровень сообщения **Debug**, **Info**, **Warning**, **Error** и **Critical**, а также фильтрацию по дате и с помощью регулярных выражений в раскрывающемся разделе **Options**.

## Библиотека *java.util.logging*

Разработчик приложения в его коде также может организовать запись сообщений в журнал, используя поддерживаемый GAE-платформой модуль `java.util.logging`.

Объект `java.util.logging.Logger` GAE-платформы может быть получен с помощью статического метода `getLogger(String name)` класса `Logger`, принимающего в качестве аргумента имя Servlet-класса, обрабатывающего HTTP-запрос к приложению.

С помощью методов `info()`, `warning()` и `severe()` класса `Logger` можно осуществлять вывод сообщений уровня информации, предупреждения и ошибки. Методы

entering(String sourceClass, String sourceMethod, Object[] params) и exiting(String sourceClass, String sourceMethod, Object result) класса Logger обеспечивают запись информации о входящих параметрах вызываемого метода и результате его выполнения. Кроме того, GAE-платформа перехватывает стандартный вывод System.out.println() и System.err.println() и записывает его в качестве сообщений журнала с уровнем информации и предупреждения соответственно.

Для демонстрации работы модуля java.util.logging.Logger GAE-платформы создадим в среде Eclipse проект GAE-приложения, как описано в *главе 1*.

В меню **File** среды Eclipse выберем команду **New | Other | Google | Web Application Project** и нажмем кнопку **Next**. В окне мастера создания проекта, в поле **Project name** введем имя проекта, в поле **Package** — имя пакета проекта, отметим флажок **Use Google App Engine** и нажмем кнопку **Finish**.

В Eclipse-редакторе откроем Servlet-класс папки src GAE-проекта и изменим его код:

```
package com.example.gae;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.logging.Logger;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {

    private static final Logger log =
        Logger.getLogger(GAEApplicationServlet.class.getName());

    public void doGet(HttpServletRequest req, HttpServletResponse resp){
    try{
        System.out.println("Out");
        System.err.println("Err");
        log.info("Information");
        log.warning("Warning");
        log.severe("Error");
        log.entering(this.getClass().getName(), "doGet",
            new Object[]{req, resp});
        log.exiting(this.getClass().getName(), "doGet");
    }catch (Exception ex){
        System.err.println(ex.getMessage());
    }}}
```

Сохраним изменения и убедимся, что тег <application> файла appengine-web.xml папки war\WEB-INF GAE-проекта содержит зарегистрированный идентификатор приложения. В окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле проекта и в контекстном меню выберем команду **Google | Deploy to App Engine**, в диалоговом окне мастера нажмем кнопку **Deploy** — в результате приложение будет развернуто на GAE-платформе.

В адресной строке Web-браузера введем адрес приложения (раздел **Application Settings | Application Default Version URL** консоли администрирования) и на открывшейся странице приложения щелкнем по ссылке вызова сервлета приложения.

Откроем раздел **Main | Logs** консоли администрирования <https://appengine.google.com/> и увидим запись в журнале приложения, содержащую сообщение самой GAE-платформы о запросе к приложению и сообщения только уровня предупреждения и ошибки, сделанные в коде метода `doGet()` Servlet-класса приложения (рис. 2.1).

```

2012-05-19 18:10:19.656 /gaeapplication 200 1543ms 0kb Mozilla/5.0 (Windows NT 6.0; rv:12.0) Gecko/20100101 Firefox/12.0
176.51.59.7 - - [19/May/2012:18:10:19 -0700] "GET /gaeapplication HTTP/1.1" 200 0 "http://softstudioiv-
demo.appspot.com/" "Mozilla/5.0 (Windows NT 6.0; rv:12.0) Gecko/20100101 Firefox/12.0" "softstudioiv-
demo.appspot.com" ms=1543 cpu_ms=700 api_cpu_ms=0 cpm_usd=0.019490 loading_request=1
instance=00c61b117c720fa612f890ba2d9b047a8282

W 2012-05-19 18:10:19.644
[s~softstudioiv-demo/1.359026031734850818].<stderr>: Err

W 2012-05-19 18:10:19.646
com.example.gae.GAEApplicationServlet doGet: Warning

E 2012-05-19 18:10:19.646
com.example.gae.GAEApplicationServlet doGet: Error

I 2012-05-19 18:10:19.656
This request caused a new process to be started for your application, and thus caused your
application code to be loaded for the first time. This request may thus take longer and use more CPU
than a typical request for your application.

```

Рис. 2.1. Запись в журнал GAE-приложения, содержащая сообщения приложения уровня предупреждения и ошибки

Сообщения уровня информации не были записаны в журнал приложения. Произошло это в результате ограничения уровня журналирования, указанного в конфигурационном файле `logging.properties` папки `WEB-INF\logging.properties` GAE-приложения:

```
.level = WARNING
```

```

2012-05-19 18:46:03.115 /gaeapplication 200 1470ms 0kb Mozilla/5.0 (Windows NT 6.0;
D 2012-05-19 18:46:03.013 com.google.apphosting.utils.servlet.WarmupServlet init: Initiali
I 2012-05-19 18:46:03.021 [s~softstudioiv-demo/1.359026603091279665].<stdout>: Out
D 2012-05-19 18:46:03.090 com.example.gae.GAEApplicationServlet doGet: ENTRY GET
I 2012-05-19 18:46:03.090 com.example.gae.GAEApplicationServlet doGet: Information
W 2012-05-19 18:46:03.090 [s~softstudioiv-demo/1.359026603091279665].<stderr>: Err
W 2012-05-19 18:46:03.090 com.example.gae.GAEApplicationServlet doGet: Warning
E 2012-05-19 18:46:03.090 com.example.gae.GAEApplicationServlet doGet: Error
D 2012-05-19 18:46:03.102 com.example.gae.GAEApplicationServlet doGet: RETURN
I 2012-05-19 18:46:03.114 This request caused a new process to be started for your app

```

Рис. 2.2. Запись в журнал GAE-приложения, содержащая сообщения приложения всех уровней

Изменим уровень журналирования на:

```
.level = ALL
```

Сохраним сделанные изменения и заново загрузим приложение в App Engine. В Web-браузере вызовем сервлет приложения и обновим страницу администрирования приложением с разделом **Main | Logs**. В результате в журнале приложения появятся все сообщения, сделанные в коде приложения (рис. 2.2).

## Библиотека Log4j

Для записи в журнал приложения можно использовать и другие системы журналирования, например Log4j.

Для использования библиотеки Log4j скачаем ее с сайта Log4j-проекта (<http://logging.apache.org/log4j/>), в окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле GAE-проекта и в контекстном меню выберем команду **Properties**. В диалоговом окне команды **Properties** откроем вкладку **Libraries** раздела **Java Build Path** и кнопкой **Add External JARs** добавим в путь приложения библиотеку log4j-1.2.16.jar. Кроме того, скопируем данную библиотеку в папку war/WEB-INF/lib GAE-проекта для того, чтобы она загрузилась в App Engine.

При создании проекта GAE-приложения в среде Eclipse плагин GPE генерирует в папке src конфигурационный файл log4j.properties системы Log4j. По умолчанию данный файл содержит настройки вывода сообщений в консоль и настройки журналирования DataNucleus-плагины GAE-платформы, который обеспечивает реализацию JDO/JPA для Datastore-хранилища.

Добавим в конфигурационный файл log4j.properties установку уровня журналирования:

```
log4j.rootLogger=ALL, A1
```

Также изменим код Servlet-класса приложения для использования Log4j-системы:

```
package com.example.gae;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.*;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    private static final Logger log =
        Logger.getLogger(GAEApplicationServlet.class.getName());

    public void doGet(HttpServletRequest req, HttpServletResponse resp){
        try{
            System.out.println("Out");
            System.err.println("Err");
```

```

log.info("Information");
log.warn("Warning");
log.error("Error");
log.debug("Debug");
LogXF.entering(log, this.getClass().getName(), "doGet",
               new Object[]{req, resp});
LogXF.exiting(log, this.getClass().getName(), "doGet");
} catch (Exception ex) {
    System.err.println(ex.getMessage());
}}

```

Сохраним сделанные изменения и заново загрузим приложение в App Engine. В Web-браузере вызовем сервлет приложения и обновим страницу администрирования приложением с разделом **Main | Logs**. В результате в журнале приложения появятся все сообщения, сделанные в коде приложения с помощью Log4j-системы.

## LogService API

GAE-сервис журналирования LogService предоставляет программный интерфейс, обеспечивающий доступ из кода GAE-приложения к журналу приложения.

Программный интерфейс LogService API представлен пакетом `com.google.appengine.api.log`, интерфейс LogService которого как раз и обеспечивает доступ к журналу приложения с помощью метода `fetch(LogQuery query)`, отправляющего запрос к журналу и возвращающего коллекцию `LogQueryResult java.lang.Iterable <RequestLogs>` объектов `RequestLogs`, представляющих записи в журнале уровня запроса с вложенными в них записями уровня приложения:

```

package com.example.gae;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.log.AppLogLine;
import com.google.appengine.api.log.LogQuery;
import com.google.appengine.api.log.LogServiceFactory;
import com.google.appengine.api.log.RequestLogs;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        resp.setContentType("text/html");
        PrintWriter writer = resp.getWriter();

```

```
// Создание запроса к журналу приложения
LogQuery query = LogQuery.Builder.withDefaults();
// Модификация запроса с помощью методов класса LogQuery
query.includeAppLogs(true);
// Запрос к журналу приложения и последовательное извлечение его записей
for (RequestLogs record : LogServiceFactory.getLogService().fetch(query)) {
    writer.println("<br /> REQUEST LOG <br />");
    Calendar cal = Calendar.getInstance();
    cal.setTimeInMillis(record.getStartTimeUsec() / 1000);
    writer.println(String.format("<br />Date: %s", cal.getTime().toString()));
    // Извлечение информации из записи с помощью методов класса RequestLogs
    writer.println("<br/> Ver: " + record.getCombined()+"<br />");
    // Извлечение вложенных записей уровня приложения
    for (AppLogLine appLog : record.getAppLogLines()) {
        writer.println("<br />" + "APPLICATION LOG" + "<br />");
        // Извлечение информации из записи с помощью методов класса AppLogLine
        writer.println("<br />Level: "+appLog.getLogLevel()+"<br />");
        writer.println("Message: "+ appLog.getLogMessage()+"<br /> <br />");
    } } }
```

Запрос к журналу приложения представлен объектом `LogQuery`, создаваемым с помощью статического класса `LogQuery.Builder`. Как правило, для создания `LogQuery`-объекта используется статический метод `LogQuery.Builder.withDefaults()`, возвращающий объект `LogQuery` со свойствами по умолчанию, который затем модифицируется методами класса `LogQuery`:

- ◆ `batchSize(int batchSize)` — параметр, который может использоваться для группировки результатов запроса. Независимо от данного параметра запрос `LogService.fetch(LogQuery)` будет выполнен в полном объеме;
- ◆ `endTimeUsec(long endTimeUsec)` — фильтрация первой запрашиваемой записи журнала по времени в микросекундах, начиная с 1 января 1970 г.;
- ◆ `includeAppLogs(boolean includeAppLogs)` — если `true`, тогда в результат запроса к журналу будут включены записи уровня приложения;
- ◆ `includeIncomplete(boolean includeIncomplete)` — если `true`, тогда в результат запроса к журналу приложения включаются записи незавершенного запроса к приложению;
- ◆ `majorVersionIds(java.util.List<java.lang.String> versionIds)` — по умолчанию запрос к журналу возвращает записи журнала вызываемой версии приложения. Данный метод обеспечивает доступ к журналам других версий приложения, которые определяются в объекте `java.util.List` аргумента метода:

```
LogQuery query = LogQuery.Builder.withDefaults();
ArrayList<java.lang.String> list= new ArrayList<java.lang.String>();
list.add("[версия приложения]");
query.majorVersionIds(list);
```

- ◆ `minLogLevel(LogService.LogLevel minLogLevel)` — фильтрация запрашиваемых записей журнала по уровню сообщения, который устанавливается с помощью перечисления `LogService.LogLevel` с полями `DEBUG`, `ERROR`, `FATAL`, `INFO`, `WARN`;
- ◆ `offset(java.lang.String offset)` — устанавливает сдвиг позиции начальной запрашиваемой записи журнала. Сложность использования данного метода заключается в том, что параметр метода представляет собой строку в кодировке Base64;
- ◆ `startTimeUsec(long startTimeUsec)` — фильтрация последней запрашиваемой записи журнала по времени в микросекундах, начиная с 1 января 1970 г.

Класс `RequestLogs` позволяет извлечь информацию, содержащуюся в записи журнала, с помощью следующих методов:

- ◆ `getApiMcycles()` — возвращает число CPU-циклов, затраченных на работу с GAE API при обработке запроса к приложению;
- ◆ `getAppId()` — возвращает идентификатор приложения;
- ◆ `getAppLogLines()` — возвращает записи журнала уровня приложения в виде списка `java.util.List<AppLogLine>`;
- ◆ `getCombined()` — возвращает информацию о запросе к приложению в формате Apache Combined Log Format;
- ◆ `getCost()` — оценочная стоимость запроса в долларах;
- ◆ `getEndTimeUsec()` — время окончания обработки запроса в микросекундах, начиная с 1 января 1970 г.;
- ◆ `getHost()` — хост запроса;
- ◆ `getHttpVersion()` — HTTP-версия запроса;
- ◆ `getInstanceKey()` — возвращает идентификатор экземпляра приложения, обрабатывавшего запрос, или ноль, если запрос был к статическому ресурсу, не требовавшему обработки;
- ◆ `getIp()` — IP-адрес запроса;
- ◆ `getLatencyUsec()` — время ожидания ответа в микросекундах;
- ◆ `getMcycles()` — число CPU-циклов, затраченных на обработку запроса;
- ◆ `getMethod()` — HTTP-метод запроса;
- ◆ `getNickname()` — возвращает логин клиента запроса, если он прошел аутентификацию;
- ◆ `getOffset()` — возвращает позицию данной записи в журнале;
- ◆ `getPendingTimeUsec()` — время в микросекундах, проведенное запросом в очереди запросов экземпляра приложения;
- ◆ `getReferrer()` — URL-адрес источника запроса;
- ◆ `getReplicaIndex()` — Backend-экземпляр, обрабатывавший запрос, или -1 в его отсутствие;
- ◆ `getRequestId()` — идентификатор запроса;



- ◆ `getResource()` — относительный URL-адрес запрашиваемого ресурса;
- ◆ `getResponseSize()` — размер ответа в байтах;
- ◆ `getStartTimeUseC()` — время начала обработки запроса в микросекундах, начиная с 1 января 1970 г.;
- ◆ `getStatus()` — HTTP-код ответа;
- ◆ `getTaskName()` — имя задачи в случае, если запрос генерировался с помощью Task Queue API;
- ◆ `getTaskQueueName()` — имя очереди задач в случае, если запрос генерировался с помощью Task Queue API;
- ◆ `getUrlMapEntry()` — относительный URL-адрес статического ресурса, используемого для запроса;
- ◆ `getUserAgent()` — информация о клиентском приложении;
- ◆ `getVersionId()` — версия приложения.

Класс `AppLogLine` позволяет извлечь информацию, содержащуюся в записи журнала уровня приложения, с помощью следующих методов:

- ◆ `getLogLevel()` — уровень сообщения;
- ◆ `getLogMessage()` — само сообщение;
- ◆ `getTimeUseC()` — время записи в микросекундах, начиная с 1 января 1970 г.

Объект `LogService`, обеспечивающий доступ к журналу приложения с помощью метода `fetch(LogQuery query)`, создается посредством статического метода `getLogService()` класса-фабрики `LogServiceFactory`.

## ГЛАВА 3



# Определение местоположения пользователя

Вопрос, который решает практически любой разработчик сайта, — это определение местоположения пользователя. Платформа App Engine упрощает решение этой задачи, автоматически добавляя ряд специфических заголовков во входящий HTTP-запрос к GAE-приложению:

- ◆ X-AppEngine-Country — код страны, определяемый GAE-платформой по IP-адресу клиента;
- ◆ X-AppEngine-Region — имя региона страны клиента;
- ◆ X-AppEngine-City — имя города клиента;
- ◆ X-AppEngine-CityLatLong — широта и долгота города клиента.

Убедиться в наличии данных HTTP-заголовков запроса можно с помощью следующего кода Servlet-класса GAE-приложения:

```
package com.example.gae;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.util.Enumeration;
import java.util.logging.Logger;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {

    private static final Logger log =
        Logger.getLogger(GAEApplicationServlet.class.getName());

    public void doGet(HttpServletRequest req, HttpServletResponse resp){
        try{
            for (Enumeration e = req.getHeaderNames(); e.hasMoreElements();) {
                log.info("header: "+e.nextElement());
            }
        }
    }
}
```

```

}catch (Exception ex){
    System.err.println(ex.getMessage());
}
}

```

Загрузив GAE-приложение с данным Servlet-классом в App Engine с помощью команды **Deploy to App Engine** среды Eclipse и вызвав сервлет приложения в Web-браузере, в разделе **Main | Logs** консоли администрирования <https://appengine.google.com/> можно увидеть запись журнала приложения, содержащую перечень HTTP-заголовков запроса, включающий специфические GAE-заголовки (рис. 3.1).



Рис. 3.1. Запись журнала приложения с перечнем HTTP-заголовков запроса

Увидеть содержимое специфических GAE-заголовков запроса к приложению можно с помощью простого кода Servlet-класса GAE-приложения:

```

package com.example.gae;
import java.util.logging.Logger;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {

    private static final Logger log =
        Logger.getLogger(GAEApplicationServlet.class.getName());

    public void doGet(HttpServletRequest req, HttpServletResponse resp){
        try{
            String country=req.getHeader("X-AppEngine-Country");
            String region=req.getHeader("X-AppEngine-Region");
            String city=req.getHeader("X-AppEngine-City");
            String citylatlong=req.getHeader("X-AppEngine-CityLatLong");
            log.info("country: "+country);
            log.info("region: "+region);

```

```
log.info("city: "+city);  
log.info("citylatlong : "+citylatlong);  
}catch (Exception ex){  
    System.err.println(ex.getMessage());  
}}}
```

После загрузки GAE-приложения с данным Servlet-классом в App Engine и вызова сервлета приложения в Web-браузере, в разделе **Main | Logs** консоли администрирования <https://appengine.google.com/> можно увидеть запись журнала приложения, показывающую содержимое специфических GAE-заголовков запроса к приложению.

## ГЛАВА 4



# Аутентификация пользователей

Аутентификация/авторизация пользователей обычного Java Web-приложения может быть организована двумя способами — декларативным или программным. Декларативная аутентификация/авторизация обеспечивается Servlet-контейнером сервера приложений и основывается на определении ограничений в дескрипторе `web.xml` приложения или с помощью аннотаций для Servlet-контейнера с поддержкой библиотеки `javax.servlet.annotation` и регистрации пользователей в сервере приложений. Программная аутентификация/авторизация обеспечивается кодом приложения и основывается на извлечении логина и пароля, в случае базовой аутентификации, из HTTP-запроса к приложению и сравнении их с данными базы данных зарегистрированных пользователей для последующей авторизации.

Для GAE-приложения платформа App Engine с помощью службы Users автоматически обеспечивает проверку аутентификации пользователя в системе Google Accounts, OpenID или домена Google Apps. Выбор системы аутентификации для GAE-приложения производится при его регистрации или в разделе **Administration | Application Settings | Authentication Options** консоли администрирования <https://appengine.google.com/>. Ограничения же доступа к ресурсам приложения определяются в дескрипторе `web.xml` развертывания GAE-приложения.

## Ограничения доступа к ресурсам в дескрипторе `web.xml`

Если просто определить ограничения доступа к ресурсам приложения в дескрипторе `web.xml` без изменения кода GAE-приложения, тогда App Engine все равно автоматически произведет проверку прохождения клиентом аутентификации в Web-браузере и в случае ее отсутствия не предоставит доступ к защищенным ресурсам приложения.

Для определения ограничений доступа к ресурсам приложения в дескриптор `web.xml` развертывания GAE-приложения добавляется блок тегов:

```
<security-constraint>
  <web-resource-collection>
```

```
<url-pattern>[относительный путь защищенных ресурсов]</url-pattern>
</web-resource-collection>
<auth-constraint>
<role-name>[имя роли клиента, для которой разрешается доступ]</role-name>
</auth-constraint>
</security-constraint>
```

GAE-платформа автоматически поддерживает только две роли пользователя — "\*" (все пользователи) и "admin" (пользователь консоли администрирования <https://appengine.google.com/> с ролью Viewer, Owner или Developer раздела **Administration | Permissions** консоли). Такие теги, как `<security-role>` и `<login-config>` дескриптора `web.xml` не поддерживаются GAE-платформой.

В качестве примера создадим GAE-приложение в среде Eclipse, выбрав в меню **File** команду **New | Other | Google | Web Application Project**. Добавим в дескриптор `web.xml` папки `war\WEB-INF` GAE-проекта теги системы защиты:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/gaeapplication</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

Запустим GAE-приложение из среды Eclipse, в окне **Project Explorer** щелкнув правой кнопкой мыши на узле проекта и в контекстном меню выбрав команду **Run As | Web Application**. Откроем в Web-браузере страницу приложения по адресу <http://localhost:8888/> и попытаемся получить доступ к защищенному ресурсу, щелкнув на ссылке вызова сервлета приложения. В ответ откроется страница эмуляции аутентификации клиента (рис. 4.1).

После нажатия кнопки **Log In** Web-страницы аутентификации локальной средой выполнения будет создан объект `java.security.Principal` для введенного email-адреса, а затем вызван сервлет приложения.

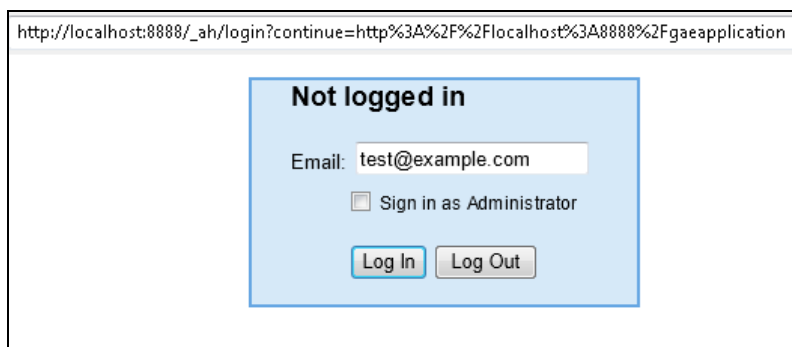


Рис. 4.1. Страница, эмулирующая аутентификацию клиента при локальном запуске GAE-приложения

Для загрузки GAE-приложения в App Engine в тег `<application>` файла `appengine-web.xml` папки `war\WEB-INF` GAE-проекта введем зарегистрированный идентификатор приложения. В окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле проекта и в контекстном меню выберем команду **Google | Deploy to App Engine**, в диалоговом окне мастера нажмем кнопку **Deploy** — в результате приложение будет развернуто на GAE-платформе.

Откроем консоль администрирования <https://appengine.google.com/> и на странице администрирования приложением в разделе **Administration | Application Settings | Authentication Options** в раскрывающемся списке выберем систему аутентификации **Google Accounts API** и нажмем кнопку **Save Settings**. В верхней части страницы администрирования приложением нажмем ссылку **Sign out** для выхода из системы Google Accounts, в адресной строке Web-браузера введем адрес приложения (раздел **Application Settings | Application Default Version URL** консоли администрирования) и на открывшейся странице приложения щелкнем по ссылке вызова сервлета приложения. В результате откроется страница аутентификации пользователей системы Google Accounts, на которой будет указан заголовок GAE-приложения (рис. 4.2).



Рис. 4.2. Страница аутентификации клиента GAE-приложения с помощью Google Accounts

После ввода данных Google-аккаунта и нажатия кнопки **Войти** будет произведен вызов сервлета приложения.

Откроем консоль администрирования <https://appengine.google.com/>, на странице администрирования приложением в разделе **Administration | Application Settings | Authentication Options** в раскрывающемся списке выберем систему аутентификации **Federated Login (OpenID)** и нажмем кнопку **Save Settings**. В адресной строке Web-браузера введем адрес приложения и на открывшейся странице приложения щелкнем по ссылке вызова сервлета приложения. В результате откроется страница ошибки (рис. 4.3), т. к. не была произведена аутентификация клиента в системе OpenID и GAE-платформа автоматически сделала вызов обработчика по адресу `/_ah/login_required`, который отсутствует.

**Error: NOT\_FOUND**

Рис. 4.3. Web-страница ошибки при отсутствии аутентификации клиента в системе OpenID и обработчика по адресу `/_ah/login_required`

Откроем в Web-браузере страницу по адресу:

```
[адрес приложения] / _ah/login_redir?claimedid=www.google.com/accounts/o8/id&continue=[адрес приложения]
```

В результате будет произведена аутентификация с помощью OpenID-идентификатора, используя Google-аккаунт, т. к. Google поддерживает протокол OpenID 2.0, обеспечивая аутентификацию в качестве провайдера OpenID. По запросу Google производит аутентификацию клиента с его Google-аккаунтом и возвращает OpenID-идентификатор. Теперь нажатие ссылки **Available Servlets** на странице приложения повлечет за собой удачный вызов сервлета приложения.

## Программный интерфейс Users API

Служба Users GAE-платформы предоставляет программный интерфейс, обеспечивающий в коде приложения проверку прохождения пользователем аутентификации, а также создание URL-адреса страницы аутентификации клиента и URL-адреса страницы выхода клиента из системы.

Программный интерфейс Users API представлен пакетом `com.google.appengine.api.users`, включающим в себя интерфейс `UserService`, классы `User` и `UserServiceFactory`, исключение `UserServiceFailureException`.

Интерфейс `UserService` с помощью методов `createLoginURL()` и `createLogoutURL()` обеспечивает создание URL-адреса страницы аутентификации клиента и URL-адреса страницы выхода клиента из системы, посредством методов `isUserAdmin()` и `isUserLoggedIn()` позволяет проверить, является ли пользователь разработчиком приложения и прошел ли пользователь аутентификацию, а методом `getCurrentUser()` возвращает объект `User`, содержащий информацию о пользователе, прошедшем аутентификацию.

Метод `getCurrentUser()` интерфейса `UserService` является аналогом метода `getUserPrincipal()` интерфейса `HttpServletRequest`. Однако объект `User`, возвращаемый методом `getCurrentUser()`, предоставляет больше информации о клиенте, по сравнению с объектом `java.security.Principal`, возвращаемым методом `getUserPrincipal()`. Интерфейс `Principal` предоставляет только метод `getName()`, возвращающий имя клиента, а класс `User` имеет следующие методы:

- ◆ `getNickname()` — возвращает логин клиента;
- ◆ `getAuthDomain()` — возвращает домен, отвечающий за аутентификацию;
- ◆ `getEmail()` — возвращает адрес электронной почты клиента;
- ◆ `getFederatedIdentity()` — возвращает OpenID-идентификатор клиента;
- ◆ `getUserId()` — возвращает идентификатор `User`-объекта, представляющего пользователя. Идентификатор `User ID` является уникальной характеристикой клиента, даже если он изменил свой email-адрес аккаунта, и является общим идентификатором клиента для всех GAE-приложений.

Объект `UserService` создается с помощью статического метода `getUserService()` класса-фабрики `UserServiceFactory`.



## Аутентификация с помощью Google Accounts

В качестве примера рассмотрим использование программного интерфейса Users API при аутентификации клиента системой Google Accounts.

Изменим код Servlet-класса GAE-приложения следующим образом:

```
package com.example.gaeapplication;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        // Создаем объект UserService
        UserService userService = UserServiceFactory.getUserService();
        // Получаем объект User
        User user = userService.getCurrentUser();
        PrintWriter writer = resp.getWriter();
        resp.setContentType("text/html");
        // Если клиент прошел аутентификацию, выводим данные клиента
        // и ссылку выхода из системы
        if (user != null) {
            writer.println("<a href=\"" + userService.createLogoutURL("/") + "\">sign out</a>");
            writer.println("<br/>Auth Domain: " + user.getAuthDomain() + "<br/>");
            writer.println("<br/>Email: " + user.getEmail() + "<br/>");
            writer.println("<br/>Federated Identity: " + user.getFederatedIdentity() + "<br/>");
            writer.println("<br/>Nick name: " + user.getNickname() + "<br/>");
            writer.println("<br/>User Id: " + user.getUserId() + "<br/>");
        }
        // Если клиент не проходил аутентификацию, выводим ссылку
        // его аутентификации
        else {
            writer.println("<a href=\"" + userService.createLoginURL(req.getRequestURI())
+ "\">sign in</a>");
        }
    }
}
```

В коде сервлета используются методы класса `User` для вывода информации о клиенте, прошедшем аутентификацию, и методы интерфейса `UserService` для создания ссылок выхода клиента из системы и его аутентификации в системе.

Методы `createLogoutURL()` и `createLoginURL()` принимают в качестве аргумента относительный URL-адрес перенаправления после выхода из системы и аутентификации клиента соответственно.

Сохраним сделанные изменения и запустим GAE-приложение из среды Eclipse, используя команду **Run As | Web Application** контекстного меню окна **Project Explorer**. Наберем в адресной строке Web-браузера адрес **http://localhost:8888/** — в результате откроется страница приветствия приложения. После нажатия ссылки вызова сервлета откроется страница эмуляции аутентификации клиента (см. рис. 4.1), после нажатия кнопки **Log In** которой будет произведен вызов сервлета и вывод информации о клиенте. В выведенной информации будут заполнены все поля, за исключением поля **Federated Identity**, т. к. аутентификация производилась с помощью Google-аккаунта, а не OpenID-идентификатора клиента. После нажатия ссылки **sign out** Web-страницы с информацией о клиенте будет открыта страница приветствия приложения, т. к. аргументом метода `createLogoutURL()` указан относительный URL-адрес `"/`. Далее все повторится с нажатием ссылки вызова сервлета.

После загрузки GAE-приложения в App Engine с помощью команды **Deploy to App Engine** приложение будет вести себя так же, как и при локальном запуске, за исключением открытия страницы Google-аутентификации клиента (см. рис. 4.2) вместо страницы ее эмуляции, при условии выбора в разделе **Administration | Application Settings | Authentication Options** консоли администрирования системы аутентификации **Google Accounts API**.

## Аутентификация с помощью OpenID

Система OpenID была создана Брэдом Фицпатриком, создателем сайта LiveJournal, в мае 2005 г. и первоначально называлась Yadis (Yet another distributed identity system). Система OpenID создавалась как открытая децентрализованная система идентификации, в которой каждый может получить и использовать свой OpenID-идентификатор или стать провайдером OpenID (<http://openid.net/get-an-openid/what-is-openid/>).

Система OpenID позволяет пользователю, получившему свой OpenID-идентификатор у OpenID-провайдера, использовать его для аутентификации на сторонних не связанных друг с другом сайтах.

Существует множество OpenID-провайдеров — это такие сервисы, как Google Accounts, Yahoo, LiveJournal и др. (<http://openid.net/get-an-openid/>). Для аутентификации пользователя на сайте с поддержкой OpenID требуется ввод OpenID URL-адреса пользователя, необходимого для запроса сайта к OpenID-провайдеру. После ввода OpenID URL-адреса сайт перенаправляет пользователя к OpenID-провайдеру, который запрашивает у пользователя подтверждение, действительно ли пользователь желает предоставить информацию о своей учетной записи. Если пользователь соглашается, тогда OpenID-провайдер перенаправляет пользователя обратно на сайт, передавая при этом информацию о пользователе.

Получить свой OpenID URL-адрес, связанный с Google-аккаунтом, можно по адресу <http://openid-provider.appspot.com/>. Каталог сайтов с поддержкой OpenID можно посмотреть по адресу <http://loginza.ru/openid-catalog>.

Для использования системы аутентификации OpenID в GAE-приложении необходимо в разделе **Administration | Application Settings | Authentication Options** консоли администрирования в раскрывающемся списке выбрать пункт **Federated Login**. Кроме того, при условии ограничения доступа к ресурсам приложения в дескрипторе web.xml необходимо реализовать обработчик по адресу `/_ah/login_required`, на который GAE-платформа автоматически перенаправит, если не была произведена аутентификация клиента с помощью OpenID-идентификатора.

В качестве примера рассмотрим использование программного интерфейса Users API и системы аутентификации клиента OpenID в GAE-приложении.

Изменим дескриптор web.xml, добавив теги ограничения доступа к ресурсам приложения и объявления обработчика по адресу `/_ah/login_required`:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/gaeapplication</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>

<servlet>
  <servlet-name>OpenIdServlet</servlet-name>
  <servlet-class>com.example.gaeapplication.OpenIdServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>OpenIdServlet</servlet-name>
  <url-pattern>/_ah/login_required</url-pattern>
</servlet-mapping>
```

В окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле Java-пакета GAE-проекта и в контекстном меню выберем команду **New | Other | Java | Class**, нажмем кнопку **Next**, введем имя класса `OpenIdServlet` и нажмем кнопку **Finish**. Изменим код класса `OpenIdServlet`

```
package com.example.gaeapplication;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashSet;
import javax.servlet.http.*;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
```

```

@SuppressWarnings("serial")
public class OpenIdServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        resp.setContentType("text/html");
        PrintWriter writer = resp.getWriter();
        UserService userService = UserServiceFactory.getUserService();
        // Создаем ссылки на странице для входа клиента
        // в учетную запись OpenID-провайдера
        String loginUrlGoogle = userService.createLoginURL("/gaeapplication", null,
"www.google.com/accounts/o8/id", new HashSet());
        writer.println("<a href=\"" + loginUrlGoogle + "\">Google</a> ");
        String loginUrlYahoo = userService.createLoginURL("/gaeapplication", null,
"yahoo.com", new HashSet());
        writer.println("<a href=\"" + loginUrlYahoo + "\">Yahoo</a> ");
        String loginUrlMyOpenId = userService.createLoginURL("/gaeapplication", null,
"myopenid.com", new HashSet());
        writer.println("<a href=\"" + loginUrlMyOpenId + "\">MyOpenId.com</a> ");
    }
}

```

Если пользователь GAE-приложения не прошел аутентификацию с помощью OpenID-идентификатора, тогда GAE-платформа автоматически вызовет сервлет `OpenIdServlet`, который отобразит на Web-странице ссылки OpenID-провайдеров для аутентификации клиента. После аутентификации пользователь будет перенаправлен по адресу `/gaeapplication` основного сервлета приложения.

Изменим код основного Servlet-класса GAE-приложения:

```

package com.example.gaeapplication;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.*;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        PrintWriter writer = resp.getWriter();
        resp.setContentType("text/html");
        if (user != null) {
            writer.println("<a href=\"" + userService.createLogoutURL("/") + "\">sign out</a>");
            writer.println("<br/>Auth Domain: " + user.getAuthDomain() + "<br/>");
            writer.println("<br/>Email: " + user.getEmail() + "<br/>");
        }
    }
}

```

```
writer.println("<br/>Federated Identity: "+ user.getFederatedIdentity() +"<br/>");  
writer.println("<br/>Nick name: "+ user.getNickname() +"<br/>");  
writer.println("<br/>User Id: "+ user.getUserId() +"<br/>");  
}}}
```

В коде основного Servlet-класса GAE-приложения используются методы класса `User` для вывода информации о клиенте, прошедшем аутентификацию с помощью OpenID-идентификатора, а также методы интерфейса `UserService` для создания ссылки выхода клиента из системы.

Локальная среда выполнения App Engine эмулирует аутентификацию только с помощью Google-аккаунта, поэтому для проверки работы примера необходимо загрузить GAE-приложение в App Engine, введя идентификатор приложения в тег `<application>` конфигурационного файла `appengine-web.xml` и воспользовавшись командой **Deploy to App Engine**.

## ГЛАВА 5



# Использование JSP/JSF-страниц в GAE-приложении

В ответ на запрос клиента GAE-приложение может динамически формировать HTML-контент непосредственно в методе `doGet()` или `doPost()` `HttpServlet`-сервлета приложения с помощью вывода `response.getWriter().println()`. Однако делать это можно и с помощью Java-технологий, в числе которых JSP (JavaServer Pages) и JSF (JavaServer Faces).

## Технология JSP

Технология JSP (JavaServer Pages) позволяет создавать динамические Web-странички, представляющие собой текстовые документы, которые содержат два типа информации — элементы HTML, SVG, XML и др., формирующие статический контент, и JSP-элементы, создающие динамический контент Web-странички.

Спецификацией JavaServer Pages для JSP-страниц допускаются файловые расширения `jsp`, `jspx` (фрагменты JSP-страниц) и `jspx` (XML-синтаксис), однако GAE-платформа без проблем работает только с расширением `jsp`.

При локальном запуске GAE-приложения с JSP-страницами среда выполнения App Engine Java SDK конвертирует JSP-страницу в Java-код, затем компилирует Java-код в Java-байт-код. При загрузке GAE-приложения с JSP-страницами в App Engine производится конвертация и компиляция и загружается только Java-байткод JSP-страниц, поэтому в случае замены статических HTML-страниц на JSP-страницы следует учитывать, что вызов HTML-страниц не потребляет ресурсы процессора, а вызов JSP-страниц будет всегда потреблять ресурсы процессора GAE-платформы. Кроме того, при загрузке в App Engine в папку `WEB-INF/lib` приложения добавляются 8 дополнительных библиотек общим объемом около 4 Мбайт, обеспечивающих обработку JSP-страниц. Убедиться в этом можно, посмотрев каталог [пользователь]\AppData\Local\Temp\appcfg.tmp\, создаваемый при загрузке GAE-приложения в App Engine.

Синтаксис JSP может быть двух несмешиваемых типов — стандартный и XML-синтаксис. Элементами JSP могут быть директивы, действия, скриптовые элементы

(скриплеты — фрагменты Java-кода, объявления переменных или методов Java-кода, вычисляемые выражения Java-кода), комментарии, видимые и невидимые клиентом, EL-выражения (Expression Language), являющиеся альтернативой скриптовым элементам, стандартные теги библиотеки JavaServer Pages Standard Tag Library (JSTL) и пользовательские теги.

В стандартном синтаксисе JSP-элементы отделяются от статических элементов страницы тегами `<% ... %>`, в XML-синтаксисе — тегами `<jsp: ... >` с пространством имен `xmlns:jsp="http://java.sun.com/JSP/Page"`. В стандартном синтаксисе корневым элементом JSP-страницы является тег-директива `<%@ page [attributes] %>`, определяющая свойства JSP-страницы с помощью атрибутов директивы. В XML-синтаксисе корневым элементом JSP-страницы является тег-действие `<jsp:root>`, указывающее, что документ — это JSP-страница. После тега `<jsp:root>` в JSP-странице с XML-синтаксисом обычно используется тег-действие `<jsp:output>` для определения свойств JSP-страницы.

Директива `<%@ include file="relativeURL" %>` или `<jsp:directive.include file="relativeURLspec"/>` позволяет вставить в JSP-страницу другую JSP-страницу, HTML-страницу или текстовый файл на стадии трансляции страницы. Включить ресурс в JSP-страницу можно также с помощью действия `<jsp:include>` только уже на стадии запроса к JSP-странице.

#### Конструкция из действий JSP-страницы

```
<jsp:element name="...">
  <jsp:attribute name="...">...</jsp:attribute>
  <jsp:body>...</jsp:body>
</jsp:element>
```

позволяет формировать элементы конечной Web-страницы. Например, теги JSP-страницы

```
<jsp:element name="h1">
  <jsp:attribute name="align">center</jsp:attribute>
  <jsp:body>Hello User</jsp:body>
</jsp:element>
```

сформируют следующий код конечной HTML-страницы:

```
<h1 align="center">Hello User</h1>
```

С помощью действия `<jsp:forward>` можно переадресовать клиентский запрос другому ресурсу (JSP-странице, сервлету и т. д.), при этом код исходной JSP-страницы после данного действия исполняться не будет. Например:

```
<jsp:forward page="login.jsp" >
  <jsp:param name="login" value="admin" />
</jsp:forward>
```

Тег `<% ... %>` (стандартный синтаксис) или тег `<jsp:scriptlet>... </jsp:scriptlet>` (XML-синтаксис) позволяет включать в JSP-страницу Java-код, а тег `<%= ... %>`

(стандартный синтаксис) или тег `<jsp:expression>... </jsp:expression>` (XML-синтаксис) — вставлять вычисляемые Java-выражения, которые после вычисления конвертируются в строки и вставляются в исходящий документ. Например, с помощью следующих JSP-элементов можно организовать проверку прохождения клиентом аутентификации непосредственно в JSP-странице:

```
<%@
page import="com.google.appengine.api.users.*"
%>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
if (user != null) {
%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out</jsp:body>
</jsp:element>
<%
}else{
%>
<jsp:element name="a">
    <jsp:attribute name="href"><%=userService.createLoginURL("/")%></jsp:attribute>
<jsp:body>sign in</jsp:body>
</jsp:element>
<%
}
%>
```

Тег `<%! ... %>` (стандартный синтаксис) или тег `<jsp:declaration> ... </jsp:declaration>` (XML-синтаксис) обеспечивает объявление переменных, используемых в Java-коде JSP-страницы.

Описание остальных JSP-элементов можно посмотреть на странице по адресу <http://java.sun.com/products/jsp/syntax/2.0/syntaxref20.html>.

Для создания JSP-страницы в проекте GAE-приложения в окне **Project Explorer** среды Eclipse IDE for Java EE Developers щелчком правой кнопкой мыши на узле **war** GAE-проекта и в контекстном меню выберем команду **New | Other | Web | JSP File**, нажмем кнопку **Next**, введем имя файла и нажмем кнопку **Next**. Откроется окно **Select JSP Template** задания шаблона JSP-страницы, в котором можно выбрать шаблон страницы JSF 1.x, JSP-страницы со стандартным синтаксисом или JSP-страницы с XML-синтаксисом, после чего нажать кнопку **Finish** для генерации основы JSP-страницы (рис. 5.1).



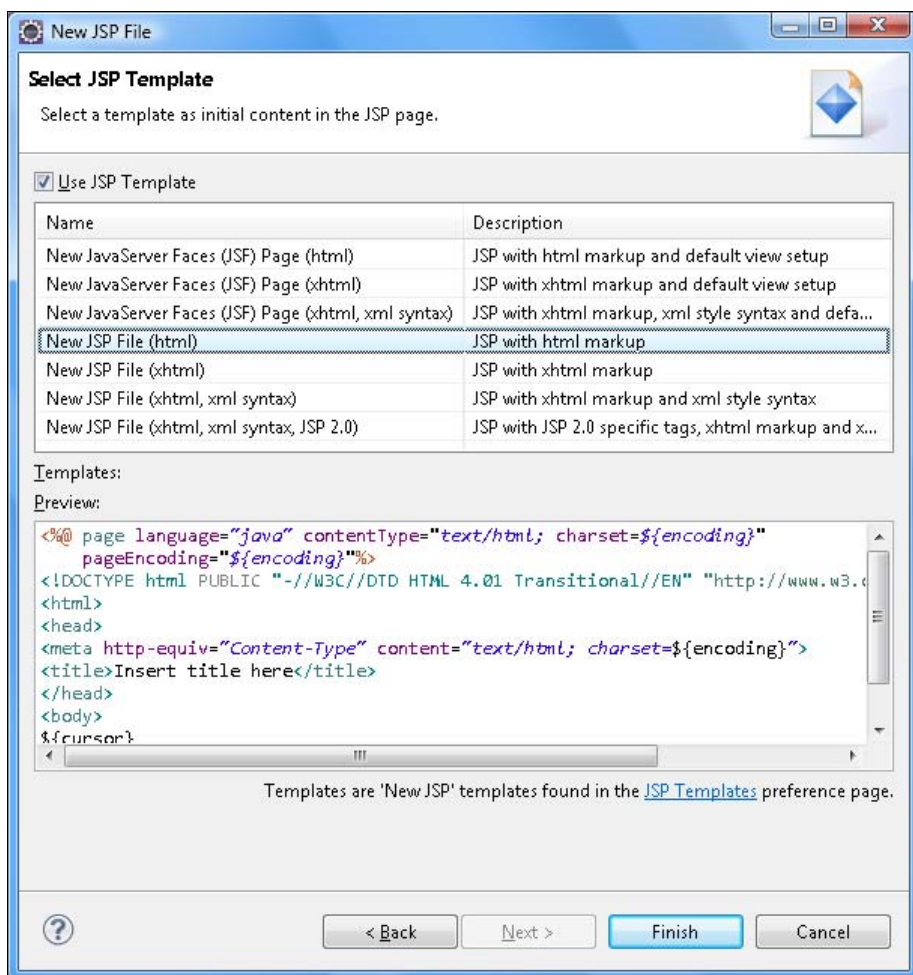


Рис. 5.1. Окно мастера создания JSP-файла с выбором JSP-шаблона

## Использование JSTL

Стандартные теги библиотеки JavaServer Pages Standard Tag Library (JSTL) представляют собой набор действий, которые могут быть использованы в JSP-странице.

JSTL-библиотека интегрирована с языком Expression Language, EL-выражения которого могут использоваться в JSTL-тегах как с помощью конструкции `${expr}`, так и посредством конструкции `#{expr}`.

Для того чтобы использовать JSTL-теги в JSP-страницах, необходимо применить директиву `taglib` (стандартный синтаксис) или атрибут `xmlns` (XML-синтаксис):

- ◆ `<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>` (стандартный синтаксис) или `<anyxml element xmlns:c="http://java.sun.com/jsp/jstl/core" />` (XML-синтаксис);

- ◆ `<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>` (стандартный синтаксис) или `<anyxMLElement xmlns:x= "http://java.sun.com/jsp/jstl/xml" />` (XML-синтаксис);
- ◆ `<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>` (стандартный синтаксис) или `<anyXMLElement xmlns:fmt = "http://java.sun.com/jsp/jstl/fmt" />` (XML-синтаксис);
- ◆ `<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>` (стандартный синтаксис) или `<anyXMLElement xmlns:sql = "http://java.sun.com/jsp/jstl/sql" />` (XML-синтаксис);
- ◆ `<%@taglib uri="http://java.sun.com/jstl/functions" prefix="fn" %>` (стандартный синтаксис) или `<anyXMLElement xmlns:fn ="http://java.sun.com/jsp/jstl/functions" />` (XML-синтаксис).

При этом тег с префиксом `c` обеспечивает использование базовой библиотеки JSTL, тег с префиксом `x` разрешает XML-обработку, тег с префиксом `fmt` обеспечивает форматирование и интернационализацию, тег с префиксом `sql` обеспечивает доступ к базам данных, тег с префиксом `fn` позволяет использовать JSTL-функции.

Описание тегов библиотеки JSTL можно посмотреть на странице по адресу <http://java.sun.com/products/jsp/jstl/reference/api/index.html>.

Для использования JSTL-библиотеки в GAE-приложении необходимо указать атрибут `isELIgnored="false"` в директиве `<%@ page [attributes] %>` (`<jsp:directive.page pageDirectiveAttrList/>`) JSP-страницы для принуждения Servlet-контейнера к распознаванию EL-выражений. Кроме того, для стандартного синтаксиса JSP-страницы URI-адрес `taglib`-библиотеки нужно указывать [http://java.sun.com/jstl/core\\_rt](http://java.sun.com/jstl/core_rt).

Поддержка языка Expression Language для JSTL включает в себя определение набора неявных объектов, обеспечивающих доступ ко всем видам данных приложения:

- ◆ `pageContext` — представляет объект `javax.servlet.jsp.PageContext`, обеспечивающий доступ к таким объектам, как `ServletRequest`, `ServletResponse`, `ServletConfig`, `ServletContext`, `HttpSession`, с помощью свойств `request`, `response`, `servletConfig`, `servletContext`, `session`;
- ◆ `param` — обеспечивает доступ к параметрам HTTP-запроса аналогично вызову метода `ServletRequest.getParameter(String)`;
- ◆ `paramValues` — обеспечивает доступ к параметрам HTTP-запроса аналогично вызову метода `ServletRequest.getParameterValues(String)`;
- ◆ `header` — обеспечивает доступ к HTTP-заголовкам аналогично вызову метода `ServletRequest.getHeader(String)`;
- ◆ `headerValues` — обеспечивает доступ к HTTP-заголовкам аналогично вызову метода `ServletRequest.getHeaders(String)`;
- ◆ `cookie` — обеспечивает доступ к cookies аналогично вызову метода `HttpServletRequest.getCookie(String)`;
- ◆ `initParam` — обеспечивает доступ к параметрам инициализации, указанным в дескрипторе `web.xml` с помощью тегов `<context-param>` `<param-name>` ... `</param-name>`

`<param-value> ... </param-value> </context-param>` аналогично вызову метода `ServletRequest.getInitParameter(String)`;

- ◆ `pageScope`, `requestScope`, `sessionScope`, `applicationScope` — обеспечивают доступ к объектам с областью видимости страницы, запроса, сессии и приложения, создаваемым средой выполнения и в коде JSP-страницы с помощью JSTL-тега `<c:set var="..." value="..." scope="[page|request|session|application]" />`.

Например, JSP-страница GAE-приложения со следующим кодом выведет параметры HTTP-запроса.

Стандартный синтаксис:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Standart</title>
</head>
<body>
<c:forEach var="parameter" items="${paramValues}">
  <ul>
    <li><b><c:out value="${parameter.key}"/></b>:</li>
    <c:forEach var="value" items="${parameter.value}">
      <c:out value="${value}"/>
    </c:forEach>
  </ul>
</c:forEach>
</body>
</html>
```

XML-синтаксис:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0" >
<jsp:directive.page contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1" session="false" isELIgnored="false" />
<jsp:output doctype-root-element="html"
doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
omit-xml-declaration="true" />
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>XML</title>
</head>
```

```
<body>
<c:forEach var="parameter" items="{paramValues}" xmlns:c=
"http://java.sun.com/jsp/jstl/core">
  <ul>
    <li><b><c:out value="{parameter.key}"/></b></li>
    <c:forEach var="value" items="{parameter.value}">
      <c:out value="{value}"/>
    </c:forEach>
  </ul>
</c:forEach>
</body>
</html>
</jsp:root>
```

Для HTTP-запроса [http://\[URL-адрес приложения\]/?first=1,2&second=3,4](http://[URL-адрес приложения]/?first=1,2&second=3,4) на Web-странице клиенту отобразится:

- second:  
3,4
- first:  
1,2

## Технология JSF

Технология JavaServer Faces (JSF) представляет собой платформу для создания интерфейса пользователя Web-приложений.

Спецификацию JSF-технологии последней версии 2.1 можно найти по адресу <http://www.jcp.org/en/jsr/detail?id=314>.

На основе платформы JSF интерфейс пользователя Web-приложения строится с помощью тегов, определяющих GUI-компоненты, при этом их свойства и функции определяются JavaBeans-компонентами, таким образом, реализуя архитектуру Model—View—Controller (MVC). Представлением данных в такой MVC-реализации служит страница с тегами JSF-компонентов, в качестве модели данных выступают JavaBeans-компоненты, а контроллером является среда выполнения JSF, отвечающая за интерпретацию и визуализацию представления, обработку событий JSF-интерфейса представления и передачу в него данных из модели.

До спецификации JSF 2.0 JSF-компоненты определялись с помощью библиотеки тегов JavaServer Pages для включения JSF-интерфейса в JSP-страницу или посредством библиотеки тегов Facelets для описания JSF-интерфейса в XHTML-странице. Начиная с версии JSF 2.0, библиотека Facelets определена библиотекой тегов по умолчанию для описания JSF-интерфейса.

Для связывания JavaBeans-компонентов, представляющих данные приложения, с JSF-компонентами используются аннотации пакета `javax.faces.bean.*`, с помощью которых определяется используемое в тегах имя JavaBeans-компонента и область его действия.

Две наиболее распространенных реализации спецификации JavaServer Faces — это проекты Mojarra (<http://javaserverfaces.java.net/>) и Apache MyFaces (<http://myfaces.apache.org/>).

Для использования JSF-технологии в GAE-приложении предварительно необходимо изменить конфигурацию его проекта.

Рассмотрим конфигурацию проекта GAE-приложения в среде Eclipse для JSF-реализации Mojarra.

Использование JSF-технологии требует поддержки класса `javax.naming.InitialContext`, который не поддерживается текущей версией GAE-платформы. Поэтому в папке `src` GAE-проекта необходимо создать класс `com.sun.faces.config.WebConfiguration` для версии Mojarra 2.0.4, исходный код которого можно скачать по адресу

<http://code.google.com/p/jsf-helloworld-2-appengine/source/browse/trunk/src/com/sun/faces/config/WebConfiguration.java?r=3>.

Также скачаем реализацию Mojarra 2.0.4 binary bundle по адресу <http://javaserverfaces.java.net/download.html> и реализацию EL API и EL Implementation языка Unified Expression Language по адресу <http://uel.java.net/download.html>, т. к. JSF-технология тесно интегрирована с языком выражений EL.

Добавим в путь приложения, используя раздел **Java Build Path | Libraries** диалогового окна **Properties** свойств GAE-проекта, библиотеки `jsf-api.jar` и `jsf-impl.jar` реализации Mojarra 2.0.4. Кроме того, скопируем в папку `war\WEB-INF\lib` GAE-проекта библиотеки `jsf-api.jar`, `jsf-impl.jar`, `el-api-1.1.jar` и `el-impl-1.1.jar`.

В конфигурационный файл `appengine-web.xml` добавим включение сессий:

```
<sessions-enabled>true</sessions-enabled>
```

А в дескриптор `web.xml` добавим определение JSF-конфигурации:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
<context-param>
  <param-name>com.sun.faces.expressionFactory</param-name>
  <param-value>com.sun.el.ExpressionFactoryImpl</param-value>
</context-param>
<context-param>
  <param-name>com.sun.faces.enableThreading</param-name>
  <param-value>>false</param-value>
</context-param>
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
```

```

    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>30</session-timeout>
</session-config>

```

В дескрипторе web.xml определим JSP-страницу приветствия:

```

<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

```

в которой сделаем перенаправление на JSF-страницу:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<% response.sendRedirect("index.jsf"); %>
</body>
</html>

```

Для создания JSF-страницы в окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле **war** GAE-проекта и в контекстном меню выберем команду **New | Other | Web | HTML File**, нажмем кнопку **Next**, введем имя файла **index.xhtml**, нажмем кнопку **Next**, в окне **Select HTML Template** мастера выберем шаблон страницы JSF 2.0 (рис. 5.2) и нажмем кнопку **Finish**.

Рассмотрим конфигурацию проекта GAE-приложения в среде Eclipse для JSF-реализации Apache MyFaces.

Скачаем реализацию MyFaces Core по адресу <http://myfaces.apache.org/download.html> и реализацию EL API и EL Implementation языка Unified Expression Language по адресу <http://uel.java.net/download.html>.

Скопируем в папку **war\WEB-INF\lib** GAE-проекта библиотеки **myfaces-api-x.x.x.jar**, **myfaces-impl-x.x.x.jar**, **commons-logging-x.x.x.jar**, **commons-beanutils-x.x.x.jar**, **commons-codec-x.x.x.jar**, **commons-collections-x.x.x.jar**, **commons-digester-x.x.x.jar**, **myfaces-bundle-x.x.x.jar** папки **lib** дистрибутива MyFaces, а также EL-библиотеки **el-api-1.1.jar** и **el-impl-1.1.jar**.

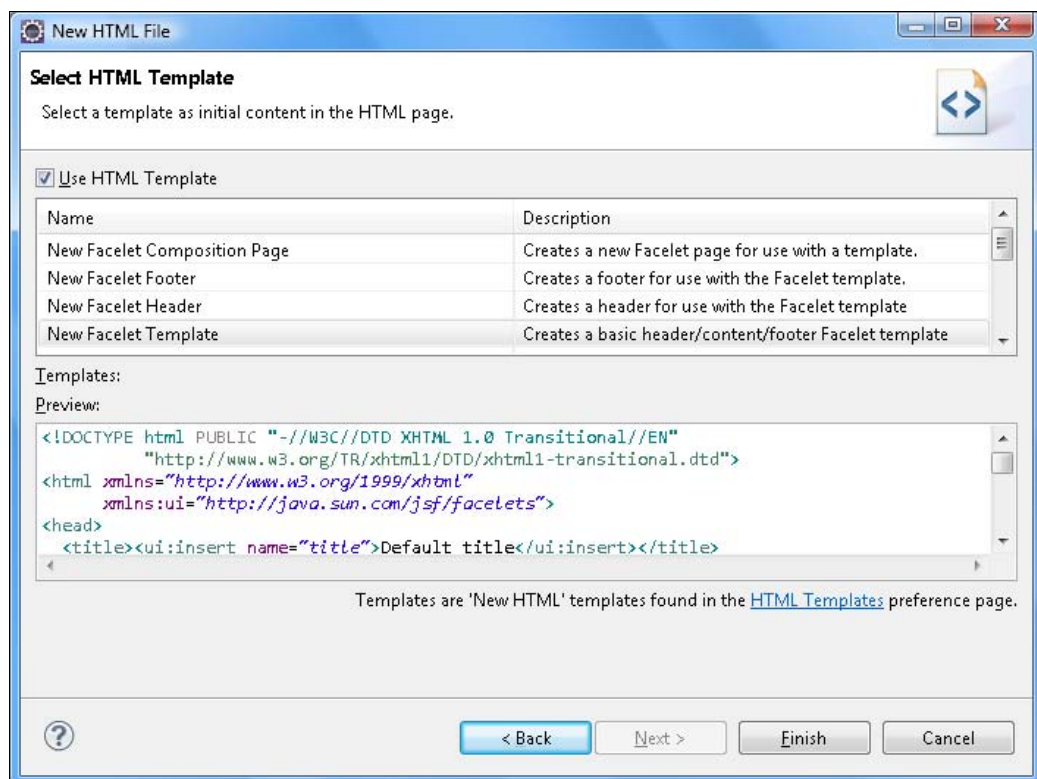


Рис. 5.2. Окно мастера создания JSF-страницы с выбором JSF-шаблона

В конфигурационный файл `appengine-web.xml` добавим включение сессий:

```
<sessions-enabled>true</sessions-enabled>
```

В дескриптор `web.xml` добавим определение JSF-конфигурации:

```
<context-param>
  <param-name>
    org.apache.myfaces.config.annotation.LifecycleProvider
  </param-name>
  <param-value>
    org.apache.myfaces.config.annotation.NoInjectionAnnotationLifecycleProvider
  </param-value>
</context-param>
<context-param>
  <param-name>org.apache.myfaces.SECRET</param-name>
  <param-value>NzYlNDMyMTA=</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

В дескрипторе web.xml определим JSP-страницу приветствия:

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

в которой сделаем перенаправление на JSF-страницу:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<% response.sendRedirect("index.jsf"); %>
</body>
</html>
```

Для создания JSF-страницы в окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле **war** GAE-проекта и в контекстном меню выберем команду **New | Other | Web | HTML File**, нажмем кнопку **Next**, введем имя файла **index.xhtml**, нажмем кнопку **Next**, в окне **Select HTML Template** мастера выберем шаблон страницы JSF 2.0 и нажмем кнопку **Finish**.



## ГЛАВА 6



# Хранение данных приложения

Для постоянного хранения данных приложения платформа App Engine предоставляет три вида хранилищ:

- ◆ App Engine Datastore — нереляционная распределенная база данных, обеспечивающая автоматическое масштабирование при увеличении объема данных. В данной системе хранения отсутствуют схемы, т. е. одна строка таблицы может иметь столбцы, отличные от столбцов другой строки таблицы;
- ◆ Google Cloud SQL — основанная на MySQL реляционная база данных;
- ◆ Google Cloud Storage и Blobstore — сервисы хранения больших, размером до 1 Тбайт, объектов.

Хранилище App Engine Datastore предоставляется приложению автоматически, и даже если код приложения не обращается к нему, GAE-платформа может сама использовать Datastore-хранилище для хранения служебных объектов приложения, таких как сессия.

Для использования GAE-приложения базы данных Google Cloud SQL предварительно необходимо пройти регистрацию в консоли Google APIs Console (<https://code.google.com/apis/console/>), создать экземпляр базы данных и установить доступ к нему для приложения.

Для хранения больших объектов GAE-приложения в коде приложения требуется применение программного интерфейса Blobstore API и/или Google Cloud Storage API.

Кроме того, платформа App Engine предоставляет службу Memcache, дающую возможность для GAE-приложения хранить данные в распределенном кэше памяти до Datastore-хранилища или вместо него, тем самым увеличивая производительность приложения.

## App Engine Datastore

Datastore-хранилище хранит объекты в виде Entity-сущностей, которые могут иметь одно или несколько свойств. Свойство Entity-сущности характеризуется именем и

может иметь одно или несколько значений, каждое из которых характеризуется типом. Сама Entity-сущность характеризуется видом и ключом-идентификатором, состоящим из вида сущности, идентификатора и дополнительно пути сущности в Datastore-иерархии. Например, объект сессии приложения хранится в виде сущности вида `_ah_SESSION` со свойствами `_expires` и `_values` (рис. 6.1).

**Edit Entity: `_ah_SESSION`**

Decoded entity key: `_ah_SESSION: name= ahsuQfo1GTwZAyz0W_Rtr0wTA`

Entity key: `ahNzfnNvZnRzdHVkaW9pdj1kZW1vcisLEgtrfYWhfU0VUOIPTilaX2Foc3VRZm8xR1R3WkF5ejBXX1J0cjB3VEEM`

Enter information for the entity below. If you'd like to change a property's type, set it to Null, save the entity, edit the entity again, and change the type.

**`_expires`**

value:

type:

**`_values`**

value: 120 bytes, SHA-1 = 25b1e3f186099a4e58a3eb6a53a92eb1aff9ca0e

type: blob

**Рис. 6.1.** Сущность сессии приложения Datastore-хранилища, как показано в разделе Datastore Viewer консоли администрирования

В Datastore-хранилище сущности одного вида могут иметь разные свойства, и свойство сущности может иметь значения с разными типами данных.

Существующее ограничение на максимальный размер сущности Datastore-хранилища — 1 Мбайт.

GAЕ-приложение может иметь доступ только к своим Entity-сущностям, используя запросы к Datastore-хранилищу, при этом запрос или группа запросов могут выполняться в пределах транзакции, гарантирующей откат к первоначальному состоянию в случае возникновения ошибки.

Для предоставления данных приложению в результате запроса к Datastore-хранилищу GAЕ-платформа использует индекс — таблицу, содержащую результаты запроса в определенном порядке. С каждым видом запроса приложения связан индекс, и соответственно запросы, не имеющие индекса, не обслуживаются. В процессе разработки GAЕ-приложения среда выполнения набора GAЕ SDK автоматически добавляет, если это необходимо, информацию об индексах приложения в его конфигурационный файл `datastore-indexes.xml` при тестировании запросов. При изменении хранящихся данных GAЕ-платформа производит автоматическое обновление соответствующих индексов, тем самым обеспечивая быстрый доступ приложения к хранящимся данным с помощью запросов.

Для хранения Java-объектов в Datastore-хранилище GAЕ-платформа обеспечивает поддержку программных интерфейсов Datastore API, Java Data Objects (JDO), Java Persistence API (JPA), Objectify, Twig и Slim3.

## Datastore API

Программный интерфейс Datastore API, обеспечивающий взаимодействие GAE-приложения с Datastore-хранилищем, представлен пакетом `com.google.appengine.api.datastore`.

Интерфейсы, классы и аннотации пакета `com.google.appengine.api.datastore` можно разделить на следующие группы для применения:

- ◆ `DatastoreServiceFactory`, `DatastoreService`, `AsyncDatastoreService` — обеспечивают синхронный и асинхронный доступ к Datastore-хранилищу;
- ◆ `Entity`, `Key`, `KeyFactory`, `KeyRange`, `EmbeddedEntity` — предназначены для создания Entity-сущностей;
- ◆ `Entities` — обеспечивает доступ к административной информации (метаданным) сущностей Datastore-хранилища приложения;
- ◆ `DatastoreServiceConfig`, `ReadPolicy` — позволяют определить конфигурацию взаимодействия приложения с Datastore-хранилищем;
- ◆ `Blob`, `Category`, `Email`, `GeoPt`, `IMHandle`, `Link`, `PhoneNumber`, `PostalAddress`, `Rating`, `ShortBlob`, `Text` — представляют специфические типы значений свойств Entity-сущностей;
- ◆ `DataTypeUtils` — обеспечивает проверку поддерживаемых типов значений свойств Entity-сущностей;
- ◆ `Query`, `PreparedQuery`, `PropertyProjection`, `RawValue`, `Cursor`, `FetchOptions`, `QueryResultList`, `QueryResultIterable`, `QueryResultIterator` — обеспечивают запросы к Datastore-хранилищу;
- ◆ `PostDelete`, `PostLoad`, `PostPut`, `PreDelete`, `PreGet`, `PrePut`, `PreQuery`, `DeleteContext`, `PreGetContext`, `PreQueryContext`, `PutContext` — позволяют создавать обработчики процесса взаимодействия с Datastore-хранилищем;
- ◆ `Transaction`, `TransactionOptions` — обеспечивают транзакции для операций с Datastore-хранилищем.

В качестве примера рассмотрим приложение, работающее с данными, которые можно сгруппировать в два набора:

- ◆ данные о пользователе приложения, представленные полями `id_user`, `user_nickname`, `user_email`;
- ◆ видео пользователя, сгруппированное в альбомы, каждый из которых характеризуется свойствами `id_album`, `id_user`, `video_album_name`, `video`, где свойство `video` является динамическим и описывает одну видеозапись альбома с помощью значений типов `com.google.appengine.api.datastore.Link` (адрес видео), `java.lang.String` (описание видео) и `java.util.Date` (дата добавления видео в альбом).

Вышеуказанные два набора данных определим как два вида Entity-сущностей `Users` и `UsersVideo`.

Такая конструкция была бы невозможна для традиционной реляционной базы данных — для нее нужно было бы определять таблицу для данных о пользователях

приложения, таблицу для альбомов видео со столбцами `id_album`, `id_user`, `video_album_name` и таблицу для видео со столбцами `id_video`, `id_user`, `id_album`, `video_url`, `video_desc`, `video_date`.

В качестве страницы приветствия приложения определим JSP-страницу, отвечающую за аутентификацию пользователя, добавление информации о новом пользователе в Datastore-хранилище, создание нового альбома видео, добавление нового видео в альбом и отображение существующих альбомов с видео пользователя. За добавление нового альбома и нового видео в альбом будет отвечать HTML-форма страницы приветствия, данные которой будут обрабатываться сервлетом приложения.

Для аутентификации пользователя и добавления информации о новом пользователе в Datastore-хранилище создадим следующий код JSP-страницы:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="com.google.appengine.api.datastore.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
if (user != null) {
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query qUsers = new Query("Users");
PreparedQuery pq = datastore.prepare(qUsers);
int auth=0;
for (Entity result : pq.asIterable()) {
    String id_user = (String) result.getProperty("id_user");
    if(id_user.equals(user.getUserId())){
        auth=1;
    }
}
if(auth==0){
    Entity Users = new Entity("Users");
    Users.setProperty("id_user", user.getUserId());
    Users.setProperty("user", user);
    datastore.put (Users);
}
%>
<div style="text-align:left">
```

```
<%
response.getWriter().println("Привет, " + user.getNickname());
%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{
    response.sendRedirect(userService.createLoginURL("/"));
}
%>
```

В приведенном коде объект `UserService` используется для получения объекта `User`, представляющего пользователя, прошедшего аутентификацию с помощью Google-аккаунта. При вызове приложения таким пользователем осуществляется запрос к Datastore-хранилищу, из которого извлекаются все Entity-сущности вида `Users` с проверкой на наличие в их числе сущности со свойством `id_user`, имеющим значение идентификатора клиента приложения. Если такая сущность не существует, создается новая сущность вида `Users` со свойствами `id_user` и `user`, которая заносится в Datastore-хранилище.

Для хранения информации о пользователях приложения применяются сущности вида `Users`, имеющие свойства `id_user` и `user`. Свойство `id_user` имеет тип данных `java.lang.String` и хранит идентификатор пользователя Google-аккаунта, который остается неизменным, даже если пользователь меняет email-адрес Google-аккаунта. Свойство `user` используется для хранения объектов `com.google.appengine.api.users.User`, содержащих логин, email-адрес и идентификатор пользователя Google-аккаунта.

С помощью встроенного JSP-объекта `response`, представляющего `HttpServletResponse`-объект, на Web-страницу приложения выводится строка приветствия пользователя приложения, в которой имя пользователя получается с помощью метода `getNickname()` класса `User`.

С помощью JSP-тега `<jsp:element>` на Web-странице приложения создается ссылка выхода пользователя из системы.

Если же приложение вызывает клиент, не прошедший аутентификацию с помощью Google-аккаунта, он перенаправляется на страницу аутентификации методом `sendRedirect(userService.createLoginURL([URL-адрес страницы после аутентификации]))` объекта `response`.

Запрос к Datastore-хранилищу по определенному виду сущностей создается путем создания экземпляра класса `Query` и последующего вызова метода `prepare(Query query)` объекта `DatastoreService`, обеспечивающего связь с Datastore-хранилищем и получаемого методом `getDatastoreService()` класса-фабрики `DatastoreServiceFactory`.

Метод `prepare()` подготавливает запрос для выполнения и возвращает объект `PreparedQuery`, который отвечает за выполнение запроса и возврат его результатов.

Набор сущностей, возвращаемый запросом к `Datastore`-хранилищу и содержащийся в объекте `PreparedQuery`, может быть представлен как итератор `java.util.Iterator<Entity>` или как список `java.util.List<Entity>`.

Новая сущность вида `Users` для нового пользователя приложения создается путем создания экземпляра класса `Entity`, с последующим его сохранением в `Datastore`-хранилище методом `put(Entity entity)` объекта `DatastoreService`. Свойства сущности определяются с помощью метода `setProperty(java.lang.String propertyName, java.lang.Object value)` `Entity`-сущности. Метод `put()` интерфейса `DatastoreService` создает новую сущность в `Datastore`-хранилище или обновляет уже существующую сущность при изменении ее свойств.

На JSP-странице приветствия приложения определим HTML-форму создания альбома и добавления видео в альбом, используя следующий код:

```
<div style="float:left; width:300px">
<p>Создать альбом:
<form action="/gaeapplication" name="create_album" method="post" >
<input type="hidden" name="htmlFormName" value="create_album"/>
<label>Название альбома:</label>
<input type="text" name="video_album_name" size="30" required>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
<%
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query qUsersVideo = new Query("UsersVideo");
qUsersVideo.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
user.getUserId()));
PreparedQuery pq = datastore.prepare(qUsersVideo);
java.util.List<Entity> listAlbums = pq.asList(FetchOptions.Builder.withDefaults());
%>
<p>Добавить видео:
<form action="/gaeapplication" name="add_video" method="post" >
<input type="hidden" name="htmlFormName" value="add_video"/>
В альбом:
<select name="video_album_name">
<%
for (Entity result : listAlbums) {
    String video_album_name = (String) result.getProperty("video_album_name");
%>
<option value="<%=video_album_name %>" ><%=video_album_name %></option>
<%
}
%>
</select>
```

```

<br>
<label>Введите адрес видео:</label>
<input type="text" name="video_url" size="30" required>
<br>
<label>Описание видео:</label><br>
<textarea rows="5" cols="45" name="video_desc"></textarea>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
</div>

```

В приведенном коде определены две HTML-формы — одна для создания альбома, а другая для добавления видео в альбом. Так как данные обеих форм посылаются методом POST и обрабатываются одним сервлетом, для их идентификации используется скрытое поле с атрибутом `name="htmlFormName"`.

В HTML-форме добавления видео в альбом используется раскрывающийся список существующих альбомов пользователя, который заполняется с помощью запроса к Datastore-хранилищу, возвращающего Entity-сущности вида `UsersVideo`.

Для хранения информации о видео пользователя приложения используются сущности вида `UsersVideo`, имеющие свойства `id_user`, `video_album_name`, `videodata_xxx`. Свойство `id_user` хранит идентификатор пользователя в виде строки, свойство `video_album_name` — имя альбома в виде строки. Свойств `videodata_xxx` у каждой `UsersVideo`-сущности может быть неограниченное количество, и имена данных свойств создаются путем добавления времени создания свойства к строке `"videodata_"`. Каждое свойство `videodata_xxx` хранит информацию об одном видео в виде коллекции `ArrayList` объектов `Link` (URL-адрес видео), `String` (описание видео) и `Date` (дата добавления видео).

Для заполнения раскрывающегося списка альбомов в HTML-форме создается запрос `Query` по сущностям вида `UsersVideo` с добавлением фильтра по свойству `id_user`, значение которого должно совпадать с идентификатором данного клиента приложения. Далее из свойств `video_album_name` `UsersVideo`-сущностей, полученных в результате запроса к Datastore-хранилищу, извлекаются значения с их добавлением в элементы `<option>` HTML-списка `<select>`, используя JSP-тег `<%=... %>`.

Для отображения существующих альбомов с видео пользователя на Web-странице используем следующий код JSP-страницы приветствия приложения:

```

<div style="float:left; margin-left:200px">
<p>Ваше видео:<br>
<%
for (Entity result : listAlbums) {
    java.util.Map map = result.getProperties();
    java.util.HashMap<String,String> mapProp=new java.util.HashMap<String,String>();
    for (Object key : map.keySet()) {
        if(key.toString().equals("video_album_name")){

```

```

        String value = map.get(key).toString();
        mapProp.put(key.toString(), value);
    }

    if(key.toString().substring(0, key.toString().indexOf("_")).equals("videodata")){
        String value = map.get(key).toString();
        mapProp.put(key.toString(), value);
    }
}
%>
Альбом:<br>
<%=mapProp.get("video_album_name") %><br>
<%
for (String key : mapProp.keySet()) {
    if(!key.equals("video_album_name")){
%>
<%=mapProp.get(key) %><br>
<%
    }
}}
%>
</p>
</div>
</body>
</html>

```

В приведенном коде используется полученный ранее набор UsersVideo-сущностей с идентификатором пользователя, из которых извлекаются все свойства в виде коллекций `java.util.Map` с последующим занесением свойств `video_album_name` и `videodata_xxx` в коллекции `java.util.HashMap`. Далее значения свойств `video_album_name` и `videodata_xxx` выводятся на Web-страницу с помощью JSP-тега `<%=... %>`.

Сервлет приложения, обрабатывающий данные HTML-форм страницы приветствия приложения, имеет следующий код:

```

package com.example.gae;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
import com.google.appengine.api.datastore.*;

@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {

```



```

public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
if (user != null) {
    String htmlFormName = req.getParameter("htmlFormName");
    if(htmlFormName.equals("create_album") &&
        (!req.getParameter("video_album_name").equals(""))){
        Entity UsersVideo = new Entity("UsersVideo");
        UsersVideo.setProperty("id_user", user.getUserId());
        UsersVideo.setProperty("video_album_name",
            req.getParameter("video_album_name"));
        datastore.put(UsersVideo);
    }
    if(htmlFormName.equals("add_video")){
        Query qUsersVideo = new Query("UsersVideo");
        qUsersVideo.addFilter("video_album_name", Query.FilterOperator.EQUAL,
            req.getParameter("video_album_name"));
        qUsersVideo.addFilter("id_user", Query.FilterOperator.EQUAL,
            user.getUserId());
        PreparedQuery pq = datastore.prepare(qUsersVideo);
        for (Entity result : pq.asIterable()) {
            result.setProperty("id_user", user.getUserId());
            result.setProperty("video_album_name",
                req.getParameter("video_album_name"));
            ArrayList listVideoProperty=new ArrayList();
            Link video_url=new Link(req.getParameter("video_url"));
            listVideoProperty.add(video_url);
            listVideoProperty.add(req.getParameter("video_desc"));
            Calendar cal = Calendar.getInstance();
            Date date=cal.getTime();
            listVideoProperty.add(date);
            String videoPropName="videodata_"+date.getTime();
            result.setProperty(videoPropName, listVideoProperty);
            datastore.put(result);
        }
    }
    resp.sendRedirect("/");
} else { resp.sendRedirect(userService.createLoginURL("/"));
}
}
}

```

В коде метода `doPost()` сервлета приложения, если HTTP POST-запрос имеет параметр `htmlFormName` со значением `create_album`, создается сущность вида `UsersVideo` со свойствами `id_user` и `video_album_name` с ее сохранением в Datastore-хранилище. Если же параметр `htmlFormName` HTTP POST-запроса к сервлету приложения имеет значение `add_video`, тогда из Datastore-хранилища извлекаются `UsersVideo`-сущности с указанным в HTTP-запросе именем альбома и идентификатором пользователя и для них добавляется свойство `videodata_xxx` с набором `ArrayList` значений, после чего

измененные UsersVideo-сущности обновляются в Datastore-хранилище методом `put()` интерфейса `DatastoreService`.

После загрузки приложения в App Engine страница приветствия приложения будет иметь вид, представленный на рис. 6.2, а раздел **Data | Datastore Viewer** консоли администрирования приложением — как показано на рис. 6.3.

Привет tmashnin  
[sign out](#)

Создать альбом

Ваше видео:

Название альбома:

Альбом:  
1  
[http://youtu.be/\_COXX0S-0ZM, Zadorov, Sun Jun 10 09:03:24 UTC 2012]  
Альбом:  
2  
[http://youtu.be/-i0hat7pdpk, Google I/O, Sun Jun 10 09:03:57 UTC 2012]

Отправить

Очистить

Добавить видео:

В альбом: 1

Введите а 1 видео:  
2

Описание видео:

Отправить

Очистить

Рис. 6.2. Страница приветствия GAE-приложения, использующего Datastore API

Query Create

By kind: UsersVideo kinds as of 0:00:06 ago

Options

UsersVideo Entities

< Prev 20 1-2 Next 20 >

ID/Name	id_user	video_album_name	videodata_1339319004026	videodata_1339319037681
<input type="checkbox"/> <a href="#">id=2003</a>	112827772722971382435	1	[u'http://youtu.be/_COXX0S-0ZM', u'Zadorov', datetime.datetime(2012, 6, 10, 9, 3, 24, 26000)]	<missing>
<input type="checkbox"/> <a href="#">id=8001</a>	112827772722971382435	2	<missing>	[u'http://youtu.be/-i0hat7pdpk', u'Google I/O', datetime.datetime(2012, 6, 10, 9, 3, 57, 681000)]

Delete

< Prev 20 1-2 Next 20 >

Рис. 6.3. Раздел **Data | Datastore Viewer** консоли администрирования, показывающий хранящиеся UsersVideo-сущности приложения

Если при создании запроса `Query` в конструкторе класса не указывать вид сущности, тогда запрос возвратит все сущности приложения, включая служебные сущности App Engine для данного приложения. Следующий код JSP-страницы выведет на

Web-страницу полный список сущностей приложения вместе с их свойствами, включая такие служебные сущности ведения статистики использования Datastore-хранилища, как `__Stat_Kind_IsRootEntity__`, `__Stat_Kind__`, `__Stat_PropertyName_Kind__`, `__Stat_PropertyType_Kind__`, `__Stat_PropertyType_PropertyName_Kind__`, `__Stat_PropertyType__`, `__Stat_Total__`:

```
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query q = new Query();
PreparedQuery p = datastore.prepare(q);
java.util.List<Entity> list=p.asList(FetchOptions.Builder.withDefaults());
for (Entity result : list) {
    java.util.Map map = result.getProperties();
    %>
    Kind Entity:
    <%=result.getKind() %><br>
    Properties:<br>
    <%
        for (Object key : map.keySet()) {
            String value = map.get(key).toString();
    %>
    <%=key.toString() %><br>
    <%=value %><br>
    <%
    }}
    %>
```

В вышеприведенном примере приложения обе сущности `Users` и `UsersVideo` являются корневыми и связаны друг с другом только через свои свойства. Однако можно и напрямую определить одну сущность как дочернюю другой сущности, связав их в одну группу сущностей. Сделать это можно при создании сущностей с помощью следующего кода:

```
Query qUsers = new Query("Users");
qUsers.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
user.getUserId()));
PreparedQuery pq = datastore.prepare(qUsers);
Entity Users = pq.asSingleEntity();
Entity UsersVideo = new Entity("UsersVideo", Users.getKey());
UsersVideo.setProperty("id_user", user.getUserId());
UsersVideo.setProperty("video_album_name", req.getParameter("video_album_name"));
datastore.put(UsersVideo);
```

Для того чтобы сущность `UsersVideo` стала дочерней по отношению к сущности `Users`, в вызов ее конструктора добавляется ключ-идентификатор сущности `Users`.

Теперь сущности приложения можно фильтровать по родительской сущности:

```
Query qUsers = new Query("Users");
qUsers.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
user.getUserId()));
```

```
PreparedQuery pqUsers= datastore.prepare(qUsers);
Entity Users = pqUsers.asSingleEntity();
Query q = new Query();
q.setAncestor(Users.getKey());
PreparedQuery p = datastore.prepare(q);
```

Объединение сущностей в иерархическую группу дает важное отличие при запросах к хранилищу Datastore (HRD). Если запрос не учитывает объединение сущностей в группу или сущности не объединены в группу, тогда запрос к хранилищу не гарантирует возврата актуальных данных из-за задержки между записью данных и их видимостью после репликации в хранилище (eventually consistent). Если же запрос осуществляется с фильтром по ключу родительской сущности, тогда запрос не будет возвращать данные до тех пор, пока вся группа сущностей не обновится, тем самым гарантируя возврат актуальных данных (strongly consistent).

Фильтровать запросы по ключу родительской сущности можно и с помощью указания ключа в конструкторе класса `Query`:

```
Query qUsers = new Query("Users");
qUsers.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
user.getUserId()));
PreparedQuery pqUsers= datastore.prepare(qUsers);
Entity Users = pqUsers.asSingleEntity();
Query q = new Query("UsersVideo", Users.getKey());
//q.setAncestor(Users.getKey());
PreparedQuery p = datastore.prepare(q);
```

В случае запроса с фильтрацией по ключу родительской сущности для гарантии актуальности данных хранилище Datastore (HRD) по умолчанию обеспечивает обращение только к первичному центру данных, и если он становится недоступным для чтения — запрос ожидает его доступность. Это не всегда удобно, и можно реализовать получение данных из реплик, хотя и нарушая гарантию актуальности данных, с помощью объектов `DatastoreServiceConfig` и `ReadPolicy`:

```
DatastoreServiceConfig con = DatastoreServiceConfig.Builder.withReadPolicy(new
ReadPolicy(ReadPolicy.Consistency.EVENTUAL));
```

Кроме того, можно установить крайний срок ожидания результатов запроса к Datastore-хранилищу в секундах:

```
DatastoreServiceConfig con =
DatastoreServiceConfig.Builder.withDefaults().deadline(10.0);
```

Запросы к Datastore-хранилищу можно также фильтровать по свойствам сущностей, при этом запрос будет возвращать не полные сущности, а только те их свойства, которые соответствуют фильтру:

```
Query q = new Query("UsersVideo");
q.addProjection(new PropertyProjection("video_album_name", String.class));
PreparedQuery p = datastore.prepare(q);
```

Класс `Cursor` программного интерфейса Datastore API позволяет организовать пролистывание хранящихся сущностей. Для этого в качестве первого шага из объекта

`PreparedQuery` извлекается список `QueryResultList<Entity>` сущностей с ограничением количества возвращаемых запросом сущностей:

```
QueryResultList<Entity> list = p.asQueryResultList(FetchOptions.Builder.withLimit(5));
```

Затем с помощью метода `getCursor()` объекта `QueryResultList<Entity>` получается объект `Cursor`, представляющий ссылку на конец списка сущностей.

Метод `toWebSafeString()` класса `Cursor` позволяет закодировать `Cursor`-объект в виде строки для передачи его как параметра HTTP-запроса, а метод `fromWebSafeString()` обеспечивает декодирование соответствующего параметра HTTP-запроса обратно в `Cursor`-объект.

Далее `Cursor`-объект может быть использован как указатель на начальную позицию запроса к `Datastore`-хранилищу:

```
QueryResultList<Entity> list = p.asQueryResultList(FetchOptions.Builder.withLimit(5).
    startCursor(cursor));
```

В результате будет получена следующая порция хранящихся сущностей, при этом объект `Query` для данного объекта `PreparedQuery` должен быть аналогичен объекту `Query`, используемому для извлечения списка, на окончание которого указывает курсор.

Как было сказано ранее, для обслуживания каждого вида запроса к `Datastore`-хранилищу GAE-платформа создает индекс — таблицу, содержащую результаты запроса в определенном порядке. При этом для большинства простых запросов создаются встроенные индексы, которые не отображаются в разделе **Data | Datastore Indexes** консоли администрирования. Для тех же запросов, для которых это необходимо, среда выполнения GAE SDK создает при тестировании приложения в папке `war\WEB-INF\appengine-generated` каталога GAE-проекта файл `datastore-indexes-auto.xml` с конфигурацией индексов. Например, для такого запроса:

```
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query qUsersVideo = new Query("UsersVideo");
qUsersVideo.addSort("video_album_name", Query.SortDirection.DESENDING);
qUsersVideo.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
    user.getUserId()));
PreparedQuery pq = datastore.prepare(qUsersVideo);
```

будет сгенерирована конфигурация индекса:

```
<datastore-indexes>
<!-- Used 1 time in query history -->
<datastore-index kind="UsersVideo" ancestor="false" source="auto">
  <property name="id_user" direction="asc"/>
  <property name="video_album_name" direction="desc"/>
</datastore-index>
</datastore-indexes>
```

После загрузки в App Engine приложения с определенными в конфигурационном файле индексами осуществить немедленный запрос к приложению не получится, т. к. после загрузки приложения требуется некоторое время для того, чтобы GAE-

платформа создала индекс согласно установленной конфигурации. В течение создания индекса в разделе **Data | Datastore Indexes** консоли администрирования в столбце **Status** строки индекса будет отображаться **Building**, а после создания индекса его статус сменится на **Serving**.

В папку `war\WEB-INF\` можно поместить и свой конфигурационный файл `datastore-indexes.xml`, определяющий индексы приложения. При этом если корневой тег `<datastore-indexes>` файла `datastore-indexes.xml` будет иметь атрибут `autoGenerate="false"`, тогда конфигурационный файл `datastore-indexes-auto.xml` будет игнорироваться, и информация об индексах будет отыскиваться GAE-платформой только в файле `datastore-indexes.xml`.

Для свойств Entity-сущностей существует особенность, связанная с тем, что если свойство имеет значение типа `com.google.appengine.api.datastore.Text` или `com.google.appengine.api.datastore.Blob`, оно ведет себя как непроиндексированное свойство. Например, запрос вида:

```
Entity UsersVideo = new Entity("UsersVideo");
Text id_user=new Text(user.getUserId());
UsersVideo.setProperty("id_user", id_user);
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query qUsersVideo = new Query("UsersVideo");
qUsersVideo.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
new Text(user.getUserId())));
PreparedQuery pq = datastore.prepare(qUsersVideo);
```

выполнен не будет из-за фильтрации по непроиндексированному свойству.

Определить свойство сущности непроиндексированным можно и программным способом, применяя к сущности метод `setUnindexedProperty(java.lang.String propertyName, java.lang.Object value)`.

В качестве значения свойства Entity-сущности можно определить объект `EmbeddedEntity`, который так же, как и объекты `Text` и `Blob`, не будет индексироваться. Объект `EmbeddedEntity`, как и объект `Entity`, представляет собой сущность со свойствами, которые определяются методами `setProperty()`, однако отличается от Entity-сущности тем, что не хранится в Datastore-хранилище самостоятельно, а только как свойство Entity-сущности и наличие Key-ключа для него не обязательно.

Проиндексированные свойства сущностей имеют свои записи во встроенных и пользовательских индексах GAE-платформы, и количество таких записей ограничено 5000 для отдельной сущности. Количество записей в индексах резко возрастает, если свойства сущности имеют по несколько значений, т. к. создаются отдельные записи для каждой перестановки значений.

Методы `asIterable()`, `asIterator()` и `asList()` класса `PreparedQuery` являются асинхронными, и их вызов не блокирует выполнения остального кода, в то время как методы `put()`, `get()` и `delete()` интерфейса `DatastoreService` являются синхронными и остальной код вынужден ожидать возврата данных методов. Применение синхронных обращений к Datastore-хранилищу необходимо, когда обрабатываемые методами сущности связаны между собой. В случае же несвязанных данных целесообразно

применять интерфейс `AsyncDatastoreService`, предоставляющий асинхронные методы `put()`, `get()` и `delete()`.

Объект `AsyncDatastoreService` получается с помощью метода `getAsyncDatastoreService()` класса-фабрики `DatastoreServiceFactory`. В отличие от интерфейса `DatastoreService`, методы интерфейса `AsyncDatastoreService` возвращают не `Entity`-сущности и `Key`-ключи, а объекты `java.util.concurrent.Future<Entity>` и `java.util.concurrent.Future<Key>`, из которых `Entity`-сущности и `Key`-ключи извлекаются методом `get()`.

Операции создания, обновления и удаления `Entity`-сущностей в `Datastore`-хранилище могут быть сгруппированы в транзакцию, гарантируя либо полное успешное выполнение всех операций транзакции, либо их полное невыполнение в случае возникновения ошибки в одной из операций.

В коде GAE-приложения транзакция начинается с создания объекта `Transaction` методом `beginTransaction(TransactionOptions options)` интерфейса `DatastoreService`. Параметром данного метода служит объект `TransactionOptions`, создаваемый с помощью класса-фабрики `TransactionOptions.Builder`.

По умолчанию все операции внутри транзакции применяются к сущностям одной группы, и в этом случае объект `Transaction` можно создавать методом `beginTransaction()` без параметров, что эквивалентно вызову метода `TransactionOptions.Builder.withDefaults()`. Если внутри такой транзакции попытаться использовать сущность другой группы, GAE-платформа выдаст ошибку:

```
can't operate on multiple entity groups in a single transaction
```

Если же есть необходимость работы с сущностями разных групп внутри транзакции, объект `Transaction` нужно создавать методом:

```
Transaction txn = datastore.beginTransaction(TransactionOptions.Builder.withXG(true));
```

Количество используемых групп сущностей внутри транзакции ограничено 5 разными группами.

Завершается транзакция вызовом метода `commit()` интерфейса `Transaction`.

Следующий код демонстрирует транзакцию, в которой видеоальбом одного пользователя перемещается к другому пользователю:

```
Query qUsers = new Query("Users");
qUsers.setFilter(new Query.FilterPredicate("id_user", Query.FilterOperator.EQUAL,
user.getUserId()));
PreparedQuery pqUsers = datastore.prepare(qUsers);
Entity Users = pqUsers.asSingleEntity();
Transaction txn = datastore.beginTransaction(TransactionOptions.Builder.withXG(true));
Query qUsersVideo = new Query("UsersVideo");
qUsersVideo.setAncestor(Users.getKey());
qUsersVideo.addFilter("video_album_name", Query.FilterOperator.EQUAL,
req.getParameter("video_album_name"));
qUsersVideo.addFilter("id_user", Query.FilterOperator.EQUAL, user.getUserId());
PreparedQuery pqUsersVideo = datastore.prepare(qUsersVideo);
```

```

for (Entity result : pqUsersVideo.asIterable()) {
    Query qOtherUsers = new Query("Users");
    qOtherUsers.addFilter("id_user", Query.FilterOperator.EQUAL, "18580476422013912411");
    PreparedQuery pqOtherUsers = datastore.prepare(qOtherUsers);
    Entity otherUsers = pqOtherUsers.asSingleEntity();
    Entity UsersVideo = new Entity("UsersVideo", otherUsers.getKey());
    UsersVideo.setProperty("id_user", otherUsers.getProperty("id_user"));
    UsersVideo.setProperty("video_album_name", result.getProperty("video_album_name"));
    datastore.put(UsersVideo);
    datastore.delete(result.getKey());
}
txn.commit();

```

Группировка операций Datastore-хранилища в транзакцию обеспечивает блокировку данных на время выполнения транзакции от попыток их параллельного изменения другими транзакциями. Для работы с актуальными данными, запросы в транзакции необходимо выполнять с фильтром по ключу родительской сущности.

Если выполняемая транзакция попала в блокировку от параллельного доступа к группе сущностей, ее можно повторить с помощью обработки исключения `java.util.ConcurrentModificationException`:

```

Transaction txn = datastore.beginTransaction();
try {
    ...
    txn.commit();
    break;
} catch (ConcurrentModificationException e) {
    // Повтор транзакции
} finally {
    if (txn.isActive()) {
        txn.rollback();
    }
}

```

Асинхронные методы интерфейса `AsyncDatastoreService` также могут быть сгруппированы в транзакцию:

```

AsyncDatastoreService datastore = DatastoreServiceFactory.getAsyncDatastoreService();
Future<Transaction> txn = datastore.beginTransaction();
...
txn.get().commitAsync();

```

## Служба Remote API

Служба Remote API платформы App Engine позволяет организовать совместное с GAE-приложением использование других служб GAE-платформы. В частности, можно обеспечить совместное использование данных Datastore-хранилища двумя разными приложениями, одно из которых может быть внешним по отношению к платформе App Engine.



Для реализации такой схемы в GAE-приложении подключается сервлет службы Remote API, а в другом приложении — клиенте создается заглушка сервлета.

Для подключения сервлета службы Remote API в дескриптор web.xml GAE-приложения добавляется настройка:

```
<servlet>
  <servlet-name>RemoteApiServlet</servlet-name>
  <servlet-class>com.google.apphosting.utils.remoteapi.RemoteApiServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>RemoteApiServlet</servlet-name>
  <url-pattern>/remote_api</url-pattern>
</servlet-mapping>
```

Для создания клиента службы Remote API в путь другого приложения добавляются библиотеки appengine-api.jar и appengine-remote-api.jar набора GAE SDK. Затем в коде приложения создается объект `com.google.appengine.tools.remoteapi.RemoteApiInstaller`, обеспечивающий связь с сервлетом `RemoteApiServlet`, который перенаправляет клиентские запросы соответствующим сервисам и возвращает ответы:

```
RemoteApiOptions options = new RemoteApiOptions()
    .server("[идентификатор GAE-приложения].appspot.com", 443)
    .remoteApiPath("/remote_api")
    .credentials(username, password);
RemoteApiInstaller installer = new RemoteApiInstaller();
installer.install(options);
try {
    // Код обращения к GAE-службам
}
} finally {
    installer.uninstall();
}
```

Объект `RemoteApiInstaller` создается на основе объекта `RemoteApiOptions`, обеспечивающего настройки соединения с сервлетом `RemoteApiServlet` GAE-приложения. В методе `server()` класса `RemoteApiOptions` указывается идентификатор GAE-приложения, соответствующий значению тега `<application>` файла `appengine-web.xml`, а также порт 443 платформы App Engine. В методе `remoteApiPath()` указывается относительный URL-адрес сервлета `RemoteApiServlet`. В методе `credentials()` — логин и пароль администратора GAE-приложения, определенного в разделе **Administration | Permissions** консоли администрирования.

Для рассмотренного ранее примера с сущностями `Users` и `UsersVideo` настольное Java-приложение, имеющее доступ к Datastore-хранилищу GAE-приложения, будет иметь следующий код:

```
package com.example.gae.client;
import com.google.appengine.api.datastore.*;
```

```
import com.google.appengine.tools.remoteapi.*;
import java.io.IOException;
public class RemoteClient {
public static void main(String[] args) throws IOException{
    String username = "[...]@gmail.com";
    String password = "[...]";
    RemoteApiOptions options = new RemoteApiOptions()
        .server("[...].appspot.com", 443)
        .remoteApiPath("/remote_api")
        .credentials(username, password);
    RemoteApiInstaller installer = new RemoteApiInstaller();
    installer.install(options);
    try {
        DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
        Query qUsersVideo = new Query("UsersVideo");
        PreparedQuery pq = datastore.prepare(qUsersVideo);
        java.util.List<Entity> listAlbums = pq.asList(FetchOptions.Builder.withDefaults());
        for (Entity result : listAlbums) {
String video_album_name = (String) result.getProperty("video_album_name");
            System.out.println(video_album_name );
        }
    } finally {
        installer.uninstall();
    }
}}
```

## JDO и JPA

Технологии Java Data Objects (JDO) и Java Persistence API (JPA) являются альтернативными моделями, обеспечивающими сохранение Java-объектов в базах данных. Относительно GAE-платформы, программные интерфейсы JDO и JPA предоставляет их реализация DataNucleus Access Platform (<http://www.datanucleus.org/>).

По сравнению с JPA технология JDO предоставляет большие возможности, в частности возможность взаимодействия с любыми типами баз данных — реляционными, объектными и файловыми, в то время как технология JPA обеспечивает взаимодействие только с реляционными базами данных. Так как Datastore-хранилище не является реляционной базой данных, ряд функций JPA для нее не поддерживается, хотя и для JDO относительно Datastore-хранилища существует ряд ограничений. В общем и целом можно сказать, что набор функций JPA представляет собой ограниченный набор функций JDO.

## JDO

Технология Java Data Objects (JDO) развивается в качестве проекта Apache JDO (<http://db.apache.org/jdo/index.html>).

Для того чтобы Java-объект сохранялся средой выполнения JDO в базе данных, он должен быть представлен аннотированным Java-классом, который компилируется

в байт-код с использованием инструмента JDO Enhancer (для GAE-платформы это DataNucleus Enhancer) для обеспечения синхронизации экземпляров класса с данными базы данных.

Для среды выполнения JDO предусмотрены два типа аннотированных Java-классов.

- ◆ Классы, экземпляры которых сохраняются в базе данных. Данные классы маркируются аннотацией `@PersistenceCapable`. Поля таких классов, предназначенные для сохранения, маркируются аннотацией `@Persistent`, а поля, не предназначенные для сохранения, — аннотацией `@NotPersistent`. Доступ к сохраняемым полям обеспечивается с помощью методов `get/set`.
- ◆ Классы, экземпляры которых не сохраняются в базе данных, но которые манипулируют сохраняемыми полями `PersistenceCapable`-классов напрямую без использования методов `get/set`. Данные классы маркируются аннотацией `@PersistenceAware`.

Модель программирования JDO в простом случае предусматривает использование всего двух интерфейсов — `javax.jdo.PersistenceManagerFactory` и `javax.jdo.PersistenceManager`.

Интерфейс `PersistenceManager` обеспечивает сохранение Java-объектов в базе данных, их извлечение, обновление и удаление из базы данных. Объект `PersistenceManager` получается с помощью фабрики `PersistenceManagerFactory`, используя метод `getPersistenceManager()`. В свою очередь объект `PersistenceManagerFactory` получается статическим методом `getPersistenceManagerFactory()` класса `javax.jdo.JDOHelper`.

Eclipse-плагин GPE при создании GAE-проекта автоматически обеспечивает создание конфигурационного JDO-файла `src\META-INF\jdoconfig.xml` и добавление необходимых JDO-библиотек в папку `war\WEB-INF\lib` проекта.

В рамках технологии JDO для создания сущности Datastore-хранилища необходимо создать `PersistenceCapable`-класс. Для рассмотренного ранее примера с сущностями `Users` и `UsersVideo`, сущность `Users` можно представить в виде класса:

```
package com.example.gae;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.PrimaryKey;
import javax.jdo.annotations.Persistent;
import com.google.appengine.api.datastore.Key;
import com.google.appengine.api.users.User;
@PersistenceCapable
public class Users {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;
    @Persistent
    private String id_user;
```

```

@Persistent
private User user;
public Users(String id_user, User user) {
    this.id_user = id_user;
    this.user = user;
}
public String getId_user() {
    return id_user;
}
public void setId_user(String id_user) {
    this.id_user = id_user;
}
public User getUser() {
    return user;
}
public void setUser(User user) {
    this.user = user;
}
public Key getKey() {
    return key;
}
}

```

### А сущность UsersVideo — в виде класса:

```

package com.example.gae;
import java.util.ArrayList;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;
import com.google.appengine.api.datastore.Key;
@PersistenceCapable
public class UsersVideo {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;
    @Persistent
    private String id_user;
    @Persistent
    private String video_album_name;
    @Persistent
    private ArrayList<String> videodata;
    public UsersVideo(String id_user, String video_album_name, ArrayList<String>
videodata) {
        this.id_user = id_user;
        this.video_album_name = video_album_name;
        this.videodata = videodata;
    }
}

```

```
public String getId_user() {
    return id_user;
}
public void setId_user(String id_user) {
    this.id_user = id_user;
}
public String getVideo_album_name() {
    return video_album_name;
}
public void setVideo_album_name(String video_album_name) {
    this.video_album_name = video_album_name;
}
public ArrayList<String> getVideodata() {
    return videodata;
}
public void setVideodata(ArrayList<String> videodata) {
    this.videodata = videodata;
}
public Key getKey() {
    return key;
}
}}
```

Для того чтобы экземпляры `PersistenceCapable`-классов сохранялись в `Datastore`-хранилище, они должны иметь поле — первичный ключ, как правило, в виде объекта `com.google.appengine.api.datastore.Key`. Первичный ключ также может быть типа `java.lang.Long`, `java.lang.String` или закодированной строки. Для первичных ключей типа `java.lang.Long` и `java.lang.String` экземпляры `PersistenceCapable`-классов не получится добавлять в группы сущностей, и за генерацию первичных ключей типа `java.lang.String` отвечает приложение. Первичный ключ в виде закодированной строки представляет собой преобразованный объект `Key` и объявляется с помощью аннотации `javax.jdo.annotations.Extension`:

```
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
@Extension(vendorName="datanucleus", key="gae.encoded-pk", value="true")
private String encodedKey;
```

В отличие от `Datastore API`, в рамках `JDO` нельзя динамически добавлять свойства для сущности и для свойства сущности с несколькими значениями — значения должны быть одного типа данных.

В `JDO` поле сущности объявляется неиндексируемым с помощью аннотации:

```
@Extension(vendorName="datanucleus", key="gae.unindexed", value="true")
```

Для JSP-страницы приветствия приложения, отвечающей за аутентификацию пользователя, добавление информации о новом пользователе в `Datastore`-хранилище, создание нового альбома видео, добавление нового видео в альбом и отображение существующих альбомов с видео пользователя, код изменится следующим образом:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="javax.jdo.*" %>
<%@ page import="com.example.gae.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
PersistenceManager pm = PMF.get().getPersistenceManager();
if (user != null) {
    Query query = pm.newQuery(Users.class);
    query.setFilter("this.id_user == id_user");
    query.declareParameters("String id_user");
    java.util.List<Users> results = (java.util.List<Users>)
query.execute(user.getId());
    if (results.isEmpty()) {
        Users users=new Users(user.getId(),user);
        pm.makePersistent(users);
    }
}%>
<div style="text-align:left">
<%
response.getWriter().println("Привет " + user.getNickname());
}%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{ response.sendRedirect(userService.createLoginURL("/")); }
}%>
<div style="float:left; width:300px">
<p>Создать альбом:
<form action="/gaeapplication" name="create_album" method="post" >
<input type="hidden" name="htmlFormName" value="create_album"/>
<label>Название альбома:</label>
<input type="text" name="video_album_name" size="30" required>

```

```

<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
<%
Query query = pm.newQuery(UsersVideo.class);
query.setFilter("this.id_user == id_user");
query.declareParameters("String id_user");
java.util.List<UsersVideo> results = (java.util.List<UsersVideo>)
query.execute(user.getUserId());
%>
<p>Добавить видео:
<form action="/gaeapplication" name="add_video" method="post" >
<input type="hidden" name="htmlFormName" value="add_video"/>
В альбом:
<select name="video_album_name">
<%
for (UsersVideo result : results) {
String video_album_name = result.getVideo_album_name();
%>
<option value="<%=video_album_name %>" ><%=video_album_name %></option>
<%
}
%>
</select>
<br>
<label>Введите адрес видео:</label>
<input type="text" name="video_url" size="30" required>
<br>
<label>Описание видео:</label><br>
<textarea rows="5" cols="45" name="video_desc"></textarea>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
</div>
<div style="float:left; margin-left:200px">
<p>Ваше видео:<br>
<%
for (UsersVideo result : results) {
String video_album_name=result.getVideo_album_name();
java.util.ArrayList<String> videodata=result.getVideodata();
%>
Альбом:<br>
<%=video_album_name%><br>
<%
for (String val : videodata) {
%>

```

```

<%=val %><br>
<%
}}
pm.close();
%>
</p>
</div>
</body>
</html>

```

В приведенном коде для создания объекта `PersistenceManager` используется класс приложения `PMF`, созданный с применением `Singleton`-шаблона для выдачи единственного экземпляра `PersistenceManagerFactory`, т. к. создание объекта `PersistenceManagerFactory` требует времени:

```

package com.example.gae;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
public final class PMF {
    private static final PersistenceManagerFactory pmfInstance =
        JDOHelper.getPersistenceManagerFactory("transactions-optional");
    private PMF() {}
    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }
}

```

Далее с помощью метода `newQuery()` интерфейса `PersistenceManager` создается запрос к хранилищу, для которого устанавливается фильтр с параметром `id_user`. Запрос выполняется методом `execute()` интерфейса `javax.jdo.Query`.

Запрос возвращает коллекцию экземпляров класса `Users`, и если в хранилище отсутствует `Users`-сущность со значением поля `id_user`, равным идентификатору клиента приложения — создается новая `Users`-сущность `Datastore`-хранилища с применением метода `makePersistent()` интерфейса `PersistenceManager`.

Сущности `UsersVideo` извлекаются из `Datastore`-хранилища также с помощью `JDO`-запросов с использованием фильтра по идентификатору клиента приложения.

В заключение объект `PersistenceManager` уничтожается методом `close()`.

Для сервлета приложения, обрабатывающего данные `HTML`-форм страницы при-  
ветствия приложения, код изменится следующим образом:

```

package com.example.gae;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import javax.jdo.PersistenceManager;
import javax.jdo.Query;
import javax.servlet.http.HttpServlet;

```



```

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        PersistenceManager pm = PMF.get().getPersistenceManager();
        if (user != null) {
            String htmlFormName = req.getParameter("htmlFormName");
            if (htmlFormName.equals("create_album") &&
                (!req.getParameter("video_album_name").equals(""))){
                UsersVideo album=new UsersVideo(user.getUserId(),
req.getParameter("video_album_name"),null);
                pm.makePersistent(album);
            }
            if (htmlFormName.equals("add_video")){
                Query query = pm.newQuery(UsersVideo.class);
                query = pm.newQuery(UsersVideo.class);
                query.setFilter("this.id_user==id_user && this.video_album_name == video_album_name
&& this.videodata == null");
                query.declareParameters("String id_user");
                query.declareParameters("String video_album_name");
                java.util.List<UsersVideo> results = (java.util.List<UsersVideo>)
query.execute(user.getUserId(), req.getParameter("video_album_name"));
                if (!results.isEmpty()){
                    for (UsersVideo result : results) {
                        ArrayList<String> videodata=new ArrayList<String>();
                        videodata.add(req.getParameter("video_url"));
                        videodata.add(req.getParameter("video_desc"));
                        Calendar cal = Calendar.getInstance();
                        Date date=cal.getTime();
                        videodata.add(date.toString());
                        result.setVideodata(videodata);
                        pm.makePersistent(result);
                    }
                }
            }
        }
    }
}
else{
    ArrayList<String> videodata=new ArrayList<String>();
    videodata.add(req.getParameter("video_url"));
    videodata.add(req.getParameter("video_desc"));
    Calendar cal = Calendar.getInstance();
    Date date=cal.getTime();
    videodata.add(date.toString());
}
}

```

```

        UsersVideo usersvideo=new
UsersVideo(user.getUserId(),req.getParameter("video_album_name"), videodata);
        pm.makePersistent(usersvideo);
    }}
    pm.close();
    resp.sendRedirect("/");
}else { resp.sendRedirect(userService.createLoginURL("/"));
}}
}

```

В приведенном коде при получении запроса к сервлету для обработки HTML-формы создания нового альбома создается новый экземпляр класса `UsersVideo` с его сохранением в Datastore-хранилище. Если же сервлет получает запрос на обработку формы добавления видео в альбом — идет проверка наличия `UsersVideo`-сущностей в Datastore-хранилище с заданным именем альбома и пустым полем `videodata`, которое заполняется данными HTML-формы. Если же `UsersVideo`-сущностей с заданным именем альбома и пустым полем `videodata` не обнаруживается, тогда создается новая `UsersVideo`-сущность.

В JDO для объединения сущностей в группу необходимо в класс родительской сущности добавить поле типа экземпляра класса дочерней сущности или типа коллекции экземпляров класса дочерней сущности. При этом запрос к дочерней сущности осуществляется по полю родительской сущности. В классе дочерней сущности можно также определить поле типа экземпляра класса родительской сущности с аннотацией `@Persistent(mappedBy = "...")`, где аргумент `mappedBy` указывает на поле типа экземпляра класса дочерней сущности в классе родительской сущности. При этом заполнение одного поля вызовет автоматическое заполнение другого поля.

Для создания встроенной сущности, в JDO встроенный класс необходимо промаркировать аннотацией:

```

@PersistenceCapable
@EmbeddedOnly

```

а поле типа экземпляра встроенного класса — аннотацией:

```

@Persistent
@Embedded

```

В отличие от Datastore API сохраняемые поля встроенного JDO-класса могут использоваться в фильтрах запросов.

В отличие от Datastore API для JDO-сущностей можно использовать наследование классов с применением JDO-стратегий `subclass-table` и `complete-table` (<http://www.datanucleus.org/products/accessplatform/jdo/orm/inheritance.html>).

Аналогом использования классов `DatastoreServiceConfig` и `ReadPolicy` программного интерфейса Datastore API, в JDO служит применение методов `addExtension()` и `setTimeoutMillis()` интерфейса `javax.jdo.Query`:

```

Query q = pm.newQuery([...].class);
q.addExtension("datanucleus.appengine.datastoreReadConsistency", "EVENTUAL");
q.setTimeoutMillis(...);

```

В отличие от Datastore API, технология JDO не поддерживает асинхронное взаимодействие с хранилищем.

Для использования курсора в JDO необходимо получить объект `com.google.appengine.api.datastore.Cursor` с помощью метода `getCursor()` класса `org.datanucleus.store.appengine.query.JDOPCursorHelper`, а затем определить его для объекта `javax.jdo.Query` посредством кода:

```
Map<String, Object> extensionMap = new HashMap<String, Object>();
extensionMap.put(JDOPCursorHelper.CURSOR_EXTENSION, cursor);
query.setExtensions(extensionMap);
query.setRange(0, [range]);
```

Группировка операций с Datastore-хранилищем в транзакцию в JDO осуществляется с применением интерфейса `javax.jdo.Transaction`, объект которого получается с помощью метода `currentTransaction()` интерфейса `PersistenceManager`:

```
PersistenceManager pm = PMF.get().getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{ tx.begin();
  ...
  tx.commit();
}
finally
{ if (tx.isActive())
  { tx.rollback(); }
}
```

Для работы с сущностями разных групп внутри транзакции, в конфигурационном файле `jdoconfig.xml` необходимо добавить свойство:

```
<property name="datanucleus.appengine.datastoreEnableXGTransactions" value="true"/>
```

## JPA

Технология Java Persistence API (JPA) является продуктом компании Oracle (<http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>).

Для того чтобы Java-объект сохранялся средой выполнения JPA в базе данных, он должен быть представлен аннотированным Java-классом, который компилируется в байткод с использованием инструмента `DataNucleus Enhancer` для обеспечения синхронизации экземпляров класса с данными базы данных.

Для среды выполнения JPA Java-класс маркируется аннотацией `@Entity` библиотеки `javax.persistence`.

Модель программирования JPA в простом случае предусматривает использование всего двух интерфейсов — `javax.persistence.EntityManagerFactory` и `javax.persistence.EntityManager`.

Интерфейс `EntityManager` обеспечивает сохранение Java-объектов в базе данных, их извлечение, обновление и удаление из базы данных. Объект `EntityManager` получается

с помощью фабрики `EntityManagerFactory` посредством метода `createEntityManager()`. В свою очередь объект `EntityManagerFactory` получается статическим методом `createEntityManagerFactory()` класса `javax.persistence.Persistence`.

Eclipse-плагин GPE при создании GAE-проекта автоматически обеспечивает добавление необходимых JPA-библиотек в папку `war\WEB-INF\lib` проекта. Для работы среды выполнения JPA требуется только создание конфигурационного JPA-файла `src\META-INF\persistence.xml`.

Таким образом, в рамках технологии JPA для создания сущности `Datastore`-хранилища необходимо создать `Entity`-класс. Для рассмотренного ранее примера с сущностями `Users` и `UsersVideo`, сущность `Users` можно представить в виде класса:

```
package com.example.gae;
import javax.persistence.*;
import com.google.appengine.api.users.User;
import com.google.appengine.api.datastore.Key;
@Entity
public class Users {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key key;
    private String id_user;
    private User user;
    public Users(String id_user, User user) {
        this.id_user = id_user;
        this.user = user;
    }
    public String getId_user() {
        return id_user;
    }
    public void setId_user(String id_user) {
        this.id_user = id_user;
    }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
    public Key getKey() {
        return key;
    }
}
```

**А сущность `UsersVideo` — в виде класса:**

```
import java.util.ArrayList;
import javax.persistence.*;
import com.google.appengine.api.datastore.Key;
```

```

@Entity
public class UsersVideo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key key;
    private String id_user;
    private String video_album_name;
    private ArrayList<String> videodata;
    public UsersVideo(String id_user, String video_album_name, ArrayList<String>
videodata) {
        this.id_user = id_user;
        this.video_album_name = video_album_name;
        this.videodata = videodata;
    }
    public String getId_user() {
        return id_user;
    }
    public void setId_user(String id_user) {
        this.id_user = id_user;
    }
    public String getVideo_album_name() {
        return video_album_name;
    }
    public void setVideo_album_name(String video_album_name) {
        this.video_album_name = video_album_name;
    }
    public ArrayList<String> getVideodata() {
        return videodata;
    }
    public void setVideodata(ArrayList<String> videodata) {
        this.videodata = videodata;
    }
    public Key getKey() {
        return key;
    }
}

```

Для того чтобы экземпляры Entity-классов сохранялись в Datastore-хранилище, они должны иметь поле — первичный ключ, как правило, в виде объекта `com.google.appengine.api.datastore.Key`. Первичный ключ также может быть типа `java.lang.Long`, `java.lang.String` или закодированной строки. Для первичных ключей типа `java.lang.Long` и `java.lang.String` экземпляры Entity-классов не получится добавлять в группы сущностей, и за генерацию первичных ключей типа `java.lang.String` отвечает приложение. Первичный ключ в виде закодированной строки представляет собой преобразованный объект `Key` и объявляется с помощью аннотации `org.datanucleus.jpa.annotations.Extension`:

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)

```

```
@Extension(vendorName = "datanucleus", key = "gae.encoded-pk", value = "true")
private String id;
```

В отличие от Datastore API, в рамках JPA нельзя динамически добавлять свойства для сущности и для свойства сущности с несколькими значениями — значения должны быть одного типа данных.

В JPA поле сущности объявляется несохраняемым с помощью аннотации `@Transient` и неиндексируемым — с помощью аннотации:

```
@Extension(vendorName="datanucleus", key="gae.unindexed", value="true")
```

Для JSP-страницы приветствия приложения, отвечающей за аутентификацию пользователя, добавление информации о новом пользователе в Datastore-хранилище, создание нового альбома видео, добавление нового видео в альбом и отображение существующих альбомов с видео пользователя, код изменится следующим образом:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="javax.persistence.*" %>
<%@ page import="com.example.gae.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
EntityManager em = EMF.get().createEntityManager();
if (user != null) {
Query query = em.createQuery("select users from Users users where
users.id_user=:id_user");
query.setParameter("id_user", user.getUserId());
java.util.List<Users> results = (java.util.List<Users>) query.getResultList();
if (results.isEmpty()) {
    Users users=new Users(user.getUserId(),user);
    em.persist(users);
}
%>
<div style="text-align:left">
<%
response.getWriter().println("Привет "+ user.getNickname());
%>
```

```

<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
} else { response.sendRedirect(userService.createLoginURL("/")); }
%>
<div style="float:left; width:300px">
<p>Создать альбом:
<form action="/gaeapplication" name="create_album" method="post" >
<input type="hidden" name="htmlFormName" value="create_album"/>
<label>Название альбома:</label>
<input type="text" name="video_album_name" size="30" required>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
<%
Query query = em.createQuery("select usersVideo from UsersVideo usersVideo where
usersVideo.id_user=:id_user");
query.setParameter("id_user", user.getUserId());
java.util.List<UsersVideo> results = (java.util.List<UsersVideo>) query.getResultList();
%>
<p>Добавить видео:
<form action="/gaeapplication" name="add_video" method="post" >
<input type="hidden" name="htmlFormName" value="add_video"/>
В альбом:
<select name="video_album_name">
<%
for (UsersVideo result : results) {
String video_album_name = result.getVideo_album_name();
%>
<option value="<%=video_album_name %>" ><%=video_album_name %></option>
<%
}
%>
</select>
<br>
<label>Введите адрес видео:</label>
<input type="text" name="video_url" size="30" required>
<br>
<label>Описание видео:</label><br>
<textarea rows="5" cols="45" name="video_desc"></textarea>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>

```

```

</form>
</p>
</div>
<div style="float:left; margin-left:200px">
<p>Ваше видео:<br>
<%
for (UsersVideo result : results) {
String video_album_name=result.getVideo_album_name();
java.util.ArrayList<String> videodata=result.getVideodata();
}%>
Альбом:<br>
<%=video_album_name%><br>
<%
for (String val : videodata) {
}%>
<%=val %><br>
<%
}}
em.close();
}%>
</p>
</div>
</body>
</html>

```

В приведенном коде для создания объекта `EntityManager` используется класс приложения EMF, созданный с применением Singleton-шаблона для выдачи единственного экземпляра `EntityManagerFactory`, т. к. создание объекта `EntityManagerFactory` требует времени:

```

package com.example.gae;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
public final class EMF {
    private static final EntityManagerFactory emfInstance =
        Persistence.createEntityManagerFactory("transactions-optional");
    private EMF() {}
    public static EntityManagerFactory get() {
        return emfInstance;
    }
}

```

Далее с помощью метода `createQuery()` интерфейса `EntityManager` создается запрос к хранилищу на извлечение `Users`-сущностей с фильтром по полю `id_user`. Запрос выполняется методом `getResultList()` интерфейса `javax.persistence.Query`.

Запрос возвращает коллекцию экземпляров класса `Users`, и если в хранилище отсутствует `Users`-сущность со значением поля `id_user`, равным идентификатору клиента приложения — создается новая `Users`-сущность `Datastore`-хранилища с применением метода `persist()` интерфейса `EntityManager`.



Сущности `UsersVideo` извлекаются из `Datastore`-хранилища также с помощью JPA-запросов с использованием фильтра по идентификатору клиента приложения.

В заключение объект `EntityManager` уничтожается методом `close()`.

Для сервлета приложения, обрабатывающего данные HTML-форм страницы приветствия приложения, код изменится следующим образом:

```
package com.example.gae;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import javax.persistence.*;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        EntityManager em = EMF.get().createEntityManager();
        if (user != null) {
            String htmlFormName = req.getParameter("htmlFormName");
            if(htmlFormName.equals("create_album") &&
                (!req.getParameter("video_album_name").equals(""))){
                UsersVideo album=new UsersVideo(user.getUserId(),
                    req.getParameter("video_album_name"),null);
                em.persist(album);
            }
            if(htmlFormName.equals("add_video")){
                Query query = em.createQuery("select usersVideo from UsersVideo usersVideo where
                usersVideo.id user=:id_user AND usersVideo.video_album_name=:video_album_name AND
                usersVideo.videodata=:videodata");
                query.setParameter("id_user", user.getUserId());
                query.setParameter("video_album_name", req.getParameter("video_album_name"));
                query.setParameter("videodata", null);
                java.util.List<UsersVideo> results = (java.util.List<UsersVideo>) query.getResultList();
                if(!results.isEmpty()){
                    for (UsersVideo result : results) {
                        ArrayList<String> videodata=new ArrayList<String>();
                        videodata.add(req.getParameter("video_url"));
                        videodata.add(req.getParameter("video_desc"));
                        Calendar cal = Calendar.getInstance();
```

```

        Date date=cal.getTime();
        videodata.add(date.toString());
        result.setVideodata(videodata);
        em.persist(result);
    }
} else {
    ArrayList<String> videodata=new ArrayList<String>();
    videodata.add(req.getParameter("video_url"));
    videodata.add(req.getParameter("video_desc"));
    Calendar cal = Calendar.getInstance();
    Date date=cal.getTime();
    videodata.add(date.toString());
    UsersVideo usersvideo=new UsersVideo(user.getUserId(),
        req.getParameter("video_album_name"), videodata);
    em.persist(usersvideo);
}
}
em.close();
resp.sendRedirect("/");
} else { resp.sendRedirect(userService.createLoginURL("/"));
}
}
}

```

В приведенном коде при получении запроса к сервлету для обработки HTML-формы создания нового альбома создается новый экземпляр класса `UsersVideo` с его сохранением в Datastore-хранилище. Если же сервлет получает запрос на обработку формы добавления видео в альбом — идет проверка наличия `UsersVideo`-сущностей в Datastore-хранилище с заданным именем альбома и пустым полем `videodata`, которое заполняется данными HTML-формы. Если же `UsersVideo`-сущности с заданным именем альбома и пустым полем `videodata` не обнаруживаются, тогда создается новая `UsersVideo`-сущность.

Для объявления сохраняемой единицы `transactions-optional` (см. класс EMF) конфигурационный файл `persistence.xml` будет иметь следующий код:

```

<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
    <persistence-unit name="transactions-optional">
        <provider> org.datanucleus.store.appengine.jpa.DatastorePersistenceProvider
    </provider>
    <properties>
        <property name="datanucleus.NontransactionalRead" value="true"/>
        <property name="datanucleus.NontransactionalWrite" value="true"/>
        <property name="datanucleus.ConnectionURL" value="appengine"/>
    </properties>
    </persistence-unit>
</persistence>

```

В JPA, также как и в JDO, для объединения сущностей в группу необходимо в класс родительской сущности добавить поле типа экземпляра класса дочерней сущности или типа коллекции экземпляров класса дочерней сущности. При этом запрос к дочерней сущности осуществляется по полю родительской сущности. В классе дочерней сущности можно также определить поле типа экземпляра класса родительской сущности с аннотацией `@Persistent(mappedBy = "...")`, где аргумент `mappedBy` указывает на поле типа экземпляра класса дочерней сущности в классе родительской сущности. При этом заполнение одного поля вызовет автоматическое заполнение другого поля:

```
@Entity
public class Parent{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key id;
    ...
    @OneToMany(targetEntity=Child.class, mappedBy="parent",
        fetch=FetchType.LAZY)
    private Set<Child> children = new HashSet<Child>();
    ...
}
@Entity
public class Child{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key id;
    ...
    @ManyToOne(fetch=FetchType.LAZY, targetEntity=Parent.class)
    private Parent parent;
    ...
}
```

Для создания встроенной сущности, в JPA встроенный класс необходимо промаркировать аннотацией `@Embeddable`, а поле типа экземпляра встроенного класса — аннотацией `@Embedded`.

В отличие от Datastore API, сохраняемые поля встроенного JPA-класса могут использоваться в фильтрах запросов.

В отличие от Datastore API, для JPA-сущностей можно использовать наследование классов с применением JPA-стратегий `TABLE_PER_CLASS` и `MAPPED_SUPERCLASS` (<http://www.datanucleus.org/products/accessplatform/jpa/orm/inheritance.html>).

Аналогом использования классов `DatastoreServiceConfig` и `ReadPolicy` программного интерфейса Datastore API в JPA служит применение метода `setHint()` интерфейса `javax.persistence.Query`:

```
Query q = em.createQuery("...");
q.setHint("datanucleus.appengine.datastoreReadConsistency", "EVENTUAL");
q.setHint("javax.persistence.query.timeout", ...);
```

В отличие от Datastore API, технология JPA не поддерживает асинхронное взаимодействие с хранилищем.

Для использования курсора в JPA необходимо получить объект `com.google.appengine.api.datastore.Cursor`, используя метод `getCursor()` класса `org.datanucleus.store.appengine.query.JPACursorHelper`, затем определить его для объекта `javax.persistence.Query` с помощью кода:

```
query.setHint(JPACursorHelper.CURSOR_HINT, cursor);
query.setMaxResults(...);
```

Группировка операций с Datastore-хранилищем в транзакцию в JPA осуществляется с применением интерфейса `javax.persistence.EntityTransaction`, объект которого получается с помощью метода `getTransaction()` интерфейса `EntityManager`:

```
EntityManager em = EMF.get().createEntityManager();
EntityTransaction txn = em.getTransaction();
txn.begin();
try {
    ...
    txn.commit();
} finally {
    if (txn.isActive()) {
        txn.rollback();
    }
}
```

Для работы с сущностями разных групп внутри транзакции в конфигурационный файл `persistence.xml` необходимо добавить свойство:

```
<property name="datanucleus.appengine.datastoreEnableXGTransactions" value="true"/>
```

## Objectify

Технология Objectify-Appengine (<http://code.google.com/p/objectify-appengine/>) представляет собой попытку соединения модели сохранения JPA с программным интерфейсом Datastore API. Технология Objectify с одной стороны оперирует аннотированными POJO-классами, а с другой стороны реализует все возможности Datastore API.

В рамках Objectify Java-класс, представляющий сущность Datastore-хранилища, маркируется как JPA-аннотациями, так и специфическими Objectify-аннотациями.

- ◆ Аннотация `@javax.persistence.Entity` может использоваться, а может и не использоваться для маркировки Java-класса сущности.
- ◆ Аннотация `@com.googlecode.objectify.annotation.Cached` при маркировке класса сущности обеспечивает кэширование полей для чтения в сервисе GAE Memcache. Такой кэш является общим для всех работающих экземпляров приложения. Срок кэширования указывается с помощью атрибута `@Cached(expirationSeconds=...)`. Можно также использовать сессионное кэширование экземпляра класса сущности:

```
ObjectifyOpts opts = new ObjectifyOpts().setSessionCache(true);  
Objectify ofy = ObjectifyService.begin(opts);
```

- ◆ Аннотация `@javax.persistence.Id` обязательна для маркировки поля, представляющего идентификатор сущности. Данное поле может иметь тип `Long`, `long` или `String`. За генерацию идентификаторов типа `long` или `String` отвечает приложение, нулевой идентификатор типа `Long` будет автоматически сгенерирован при сохранении сущности.
- ◆ Аннотация `@com.googlecode.objectify.annotation.Indexed` — необязательная аннотация, маркирующая индексируемые поля.
- ◆ Аннотация `@com.googlecode.objectify.annotation.Unindexed` позволяет определить неиндексируемое поле.
- ◆ Аннотация `@com.googlecode.objectify.annotation.NotSaved` маркирует не сохраняемое в хранилище поле.

#### ПРИМЕЧАНИЕ

Аннотации `@NotSaved`, `@Indexed` и `@Unindexed` можно использовать с условиями, представленными классами `Always`, `IfDefault`, `IfEmpty`, `IfEmptyString`, `IfFalse`, `IfNotEmpty`, `IfNotNull`, `IfNotZero`, `IfNull`, `IfTrue`, `IfZero`, `PojoIf`, `ValueIf` пакета `com.googlecode.objectify.condition`. Например, аннотация `@Unindexed(ifFalse.class)` `boolean flag` обеспечивает индексацию поля только в том случае, если его значение `true`.

- ◆ Аннотация `@javax.persistence.Transient` маркирует несохраняемое поле. Также не сохраняются статические и финальные поля.
- ◆ Аннотация `@com.googlecode.objectify.annotation.Parent` маркирует поле типа `com.google.appengine.api.datastore.Key`, представляющее ключ-идентификатор родительской сущности для данной группы сущностей.
- ◆ Аннотация `@com.googlecode.objectify.annotation.AlsoLoad` позволяет переименовывать поля класса сущности. Например, аннотация `@AlsoLoad("name1") String name2` будет ассоциировать ранее именованное поле `name1` с переименованным полем `name2`.
- ◆ Аннотация `@javax.persistence.Embedded` маркирует поле типа экземпляра встроенного класса.
- ◆ Аннотация `@com.googlecode.objectify.annotation.Serialized` представляет собой альтернативу аннотации `@Embedded` для маркировки полей типа экземпляра Java-класса, реализующего интерфейс `java.io.Serializable`. Однако поля такого встроенного класса не могут участвовать в фильтрах запросов, и внутри такого класса `Objectify`-аннотации будут игнорироваться. Преимуществом использования данной аннотации является возможность сохранения любых сериализуемых объектов и необязательность указания точного типа поля — тип поля можно указать как `Object`.
- ◆ Аннотация `@javax.persistence.PostLoad` маркирует метод без аргументов или с одним аргументом типа `Entity`, который вызывается после загрузки сущности из хранилища.

- ◆ Аннотация `@javax.persistence.PrePersist` маркирует метод без аргументов или с одним аргументом типа `Entity`, который вызывается перед загрузкой сущности в хранилище.

Класс сущности должен обязательно иметь конструктор без аргументов.

После создания Java-класса, представляющего сущность Datastore-хранилища, его необходимо зарегистрировать с помощью статического метода `register()` класса-фабрики `com.googlecode.objectify.ObjectifyService`:

```
ObjectifyService.register([имя класса].class);
```

После регистрации класса сущности все операции с ним выполняются с помощью методов интерфейса `com.googlecode.objectify.Objectify`, объект которого получается статическим методом `begin()` класса-фабрики `com.googlecode.objectify.ObjectifyService`:

- ◆ `get([имя класса].class, [имя экземпляра класса].id)` или `get([имя класса].class, new Long[] { [имя экземпляра класса].id, ... })` — извлекают сущность или сущности из хранилища;
- ◆ `put([имя экземпляра класса], ...)` — сохраняет сущность или сущности в хранилище;
- ◆ `delete(new Key<[имя класса]>([имя класса].class, [имя экземпляра класса].id), ...)` или `delete([имя экземпляра класса])` — удаляют сущность или сущности из хранилища.

Запросы к Datastore-хранилищу осуществляются с помощью объекта `com.googlecode.objectify.Query`, получаемого методом `query([имя класса сущности].class)` интерфейса `Objectify`. Метод `filter()` интерфейса `Query` позволяет установить фильтр запроса, метод `get()` — получить сущность в результате запроса, методы `fetch()` и `fetchKeys()` — получить коллекции сущностей и их ключей в результате запроса. Метод `ancestor(Object keyOrEntity)` интерфейса `Query` дает возможность установить фильтр запроса по ключу родительской сущности.

Для использования курсора необходимо получить объект `com.google.appengine.api.datastore.Cursor`, используя метод `getStartCursor()` объекта `com.google.appengine.api.datastore.QueryResultIterator`, получаемого методом `iterator()` интерфейса `Query`, затем определить его для объекта `Query` с помощью метода `startCursor()`. Метод `limit()` интерфейса `Query` позволяет ограничить количество возвращаемых результатов.

По умолчанию все `Objectify`-запросы `Query` являются асинхронными. Для выполнения асинхронных операций `get()`, `put()` и `delete()` необходимо использовать объект `com.googlecode.objectify.AsyncObjectify`, получаемый из объекта `Objectify` методом `async()`. При этом методы `get()`, `put()` и `delete()` будут возвращать объекты `com.googlecode.objectify.Result<T>`, из которых можно извлечь объекты `Future<T>` методом `getFuture()`.

Технология `Objectify` позволяет оперировать с полиморфной иерархией классов сущностей, при этом суперкласс должен быть промаркирован аннотацией `@Entity`, а его подклассы — аннотациями `@Subclass`.

Группировка операций с Datastore-хранилищем в транзакцию осуществляется путем получения объекта Objectify статическим методом `beginTransaction()` класса-фабрики `ObjectifyService`:

```
Objectify ofy = ObjectifyService.beginTransaction();
try
{
    ...
    ofy.getTxn().commit();
}
finally
{
    if (ofy.getTxn().isActive())
        ofy.getTxn().rollback();
}
```

Технология Objectify автоматически поддерживает работу с сущностями разных групп внутри транзакции.

Для создания в среде Eclipse GAE-приложений, использующих технологию Objectify, необходимо скачать библиотеку Objectify (<http://code.google.com/p/objectify-appengine/downloads/list>) и добавить ее в папку `war\WEB-INF\lib` каталога GAE-проекта, а также в путь приложения, используя команду **Build Path** контекстного меню окна **Project Explorer** среды Eclipse.

Для удобства работы с Objectify в среде Eclipse можно отключить инструмент DataNucleus Enhancer, убрав флажок **Enhancer** в окне **Properties** GAE-проекта.

Для рассмотренного ранее примера с сущностями `Users` и `UsersVideo`, сущность `Users` можно представить в виде класса:

```
package com.example.gae;
import javax.persistence.Id;
import com.google.appengine.api.users.User;
public class Users {
    @Id
    private Long id;
    private String id_user;
    private User user;
    public Users() { }
    public Users(String id_user, User user) {
        this.id_user = id_user;
        this.user = user;
    }
    public String getId_user() {
        return id_user;
    }
    public void setId_user(String id_user) {
        this.id_user = id_user;
    }
    public User getUser() {
        return user;
    }
}
```

```
public void setUser(User user) {
    this.user = user;
}
public Long getId() {
    return id;
}}
```

### А сущность UsersVideo — в виде класса:

```
package com.example.gae;
import java.util.ArrayList;
import javax.persistence.Id;
public class UsersVideo {
    @Id
    private Long id;
    private String id_user;
    private String video_album_name;
    private ArrayList<String> videodata;
    public UsersVideo() { }
    public UsersVideo(String id_user, String video_album_name, ArrayList<String>
videodata) {
        this.id_user = id_user;
        this.video_album_name = video_album_name;
        this.videodata = videodata;
    }
    public String getId_user() {
        return id_user;
    }
    public void setId_user(String id_user) {
        this.id_user = id_user;
    }
    public String getVideo_album_name() {
        return video_album_name;
    }
    public void setVideo_album_name(String video_album_name) {
        this.video_album_name = video_album_name;
    }
    public ArrayList<String> getVideodata() {
        return videodata;
    }
    public void setVideodata(ArrayList<String> videodata) {
        this.videodata = videodata;
    }
    public Long getId() {
        return id;
    }
}}
```

Для того чтобы избежать повторной регистрации сущностей, создадим класс, расширяющий класс `com.googlecode.objectify.util.DAOWBase`:



```
package com.example.gae;
import com.googlecode.objectify.util.DAOWBase;
import com.googlecode.objectify.*;
public class DAO extends DAOWBase
{static {
    ObjectifyService.register(Users.class);
    ObjectifyService.register(UsersVideo.class);
}}
```

Для JSP-страницы приветствия приложения, отвечающей за аутентификацию пользователя, добавление информации о новом пользователе в Datastore-хранилище, создание нового альбома видео, добавление нового видео в альбом и отображение существующих альбомов с видео пользователя, код изменится следующим образом:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="javax.persistence.*" %>
<%@ page import="com.googlecode.objectify.*" %>
<%@ page import="com.googlecode.objectify.Query" %>
<%@ page import="com.example.gae.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
DAO dao = new DAO();
Objectify ofy=dao.ofy();
if (user != null) {
    Query<Users> query = ofy.query(Users.class);
    query.filter("id_user", user.getUserId());
    java.util.List<Users> results = (java.util.List<Users>) query.list();
    if (results.isEmpty()) {
        Users users=new Users(user.getUserId(),user);
        ofy.put(users);
    }
}%>
<div style="text-align:left">
<%
response.getWriter().println("Привет "+ user.getNickname());
%>
```

```

<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{
    response.sendRedirect(userService.createLoginURL("/"));
}
%>
<div style="float:left; width:300px">
<p>Создать альбом:
<form action="/gaeapplication" name="create_album" method="post" >
<input type="hidden" name="htmlFormName" value="create_album"/>
<label>Название альбома:</label>
<input type="text" name="video_album_name" size="30" required>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
<%
Query<UsersVideo> query = ofy.query(UsersVideo.class);
query.filter("id_user", user.getUserId());
java.util.List<UsersVideo> results = (java.util.List<UsersVideo>)query.list();
%>
<p>Добавить видео:
<form action="/gaeapplication" name="add_video" method="post" >
<input type="hidden" name="htmlFormName" value="add_video"/>
В альбом:
<select name="video_album_name">
<%
for (UsersVideo result : results) {
    String video_album_name = result.getVideo_album_name();
%>
<option value="<%=video_album_name %>" ><%=video_album_name %></option>
<%
}
%>
</select>
<br>
<label>Введите адрес видео:</label>
<input type="text" name="video_url" size="30" required>
<br>
<label>Описание видео:</label><br>
<textarea rows="5" cols="45" name="video_desc"></textarea>
<p><input type="submit" value="Отправить">

```

```



```

Для сервлета приложения, обрабатывающего данные HTML-форм страницы приветствия приложения, код изменится следующим образом:

```

package com.example.gae;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
import com.googlecode.objectify.Objectify;
import com.googlecode.objectify.ObjectifyService;
import com.googlecode.objectify.Query;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();

```

```

Objectify ofy = ObjectifyService.begin();
if (user != null) {
    String htmlFormName = req.getParameter("htmlFormName");
    if(htmlFormName.equals("create_album") &&
        (!req.getParameter("video_album_name").equals(""))) {
        ArrayList<String> videodata=new ArrayList<String>();
        videodata.add("");
        UsersVideo album=new UsersVideo(user.getUserId(),
req.getParameter("video_album_name"),videodata);
        ofy.put(album);
    }
    if(htmlFormName.equals("add_video")){
        Query<UsersVideo> query = ofy.query(UsersVideo.class);
        query.filter("id_user",
            user.getUserId()).filter("video_album_name",
req.getParameter("video_album_name")).filter("videodata", "");
        java.util.List<UsersVideo> results = (java.util.List<UsersVideo>) query.list();
        if(!results.isEmpty()){
            for (UsersVideo result : results) {
                ArrayList<String> videodata=new ArrayList<String>();
                videodata.add(req.getParameter("video_url"));
                videodata.add(req.getParameter("video_desc"));
                Calendar cal = Calendar.getInstance();
                Date date=cal.getTime();
                videodata.add(date.toString());
                result.setVideodata(videodata);
                ofy.put(result);
            }
        }else{
            ArrayList<String> videodata=new ArrayList<String>();
            videodata.add(req.getParameter("video_url"));
            videodata.add(req.getParameter("video_desc"));
            Calendar cal = Calendar.getInstance();
            Date date=cal.getTime();
            videodata.add(date.toString());
            UsersVideo usersvideo=new
UsersVideo(user.getUserId(),req.getParameter("video_album_name"), videodata);
            ofy.put(usersvideo);
        }
    }
    resp.sendRedirect("/");
}else { resp.sendRedirect(userService.createLoginURL("/"));
}
}
}

```

## Twig

Технология Twig (<http://code.google.com/p/twig-persist/>) представляет собой попытку адаптации программного интерфейса Datastore API для работы с POJO-классами.

В рамках технологии Twig Java-класс, представляющий сущность, может быть создан без использования каких-либо аннотаций. Единственное требование к Java-классу — он должен иметь конструктор по умолчанию или конструктор без аргументов.

Однако программный интерфейс Twig предоставляет и аннотации пакета `com.vercer.engine.persist.annotation`, позволяющие добавить конфигурационную информацию к модели данных. При этом объект `com.vercer.engine.persist.ObjectDatastore`, обеспечивающий взаимодействие с Datastore-хранилищем, должен быть создан с помощью конструктора класса `com.vercer.engine.persist.annotation.AnnotationObjectDatastore`:

```
ObjectDatastore datastore = new AnnotationObjectDatastore();
```

Аннотации `@Child` и `@Parent` позволяют объединить сущности в одну группу. Аннотация `@Child` маркирует поле типа объекта дочерней сущности, а аннотация `@Parent` маркирует поле типа объекта родительской сущности.

Аннотация `@Key` маркирует поле ключа-идентификатора сущности.

Аннотация `@Type([имя класса].class)` дает возможность точно указать тип данных поля.

Аннотация `@Embed` маркирует поле типа объекта встроенной сущности.

Если при создании объекта `ObjectDatastore` в конструкторе класса `AnnotationObjectDatastore` указать `false`:

```
ObjectDatastore datastore = new AnnotationObjectDatastore(false);
```

тогда по умолчанию все поля будут неиндексируемыми. При этом с помощью аннотации `@Index` можно маркировать индексируемые поля.

Также с помощью аннотации `@Store(false)` можно указать несохраняемые поля.

Аннотация `@Activate([value])` позволяет указать глубину загрузки связанных сущностей.

Аннотация `@Independent` маркирует поле типа объекта сущности, не являющейся ни дочерней, ни родительской, ни встроенной.

После создания классов сущностей объект `ObjectDatastore` своими методами `store()`, `load()`, `find()`, `update()`, `refresh()` и `delete()` обеспечивает сохранение, загрузку, обновление и удаление сущностей. При применении методов `store()` и `find()` без аргументов, они возвращают объекты `StoreCommand` и `FindCommand`, которые позволяют указать экземпляр класса сущности, определить совместное хранение связанных сущностей, осуществить синхронный или асинхронный возврат ключа сущности, определить генерацию уникального ключа сущности, добавить сортировку и фильтр результатов запроса, сдвиг запроса, осуществить синхронный или асинхронный возврат результатов запроса:

◆ синхронная операция:

```
com.google.appengine.api.datastore.Key key = datastore.store()  
.instance([имя экземпляра])
```

```
.ensureUniqueKey()  
.batchRelated()  
.returnKeyNow();
```

◆ **асинхронная операция:**

```
java.util.concurrent.Future<com.google.appengine.api.datastore.Key> f =  
datastore.store()  
.instance([имя экземпляра])  
.ensureUniqueKey()  
.batchRelated()  
.returnKeyLater();
```

◆ **синхронный запрос:**

```
com.google.appengine.api.datastore.QueryResultIterator<[имя класса]> it =  
datastore.find()  
.type([имя класса].class)  
.addSort("...")  
.addFilter("...", com.google.appengine.api.datastore.Query.FilterOperator.EQUAL, ...)  
.withAncestor(...)  
.startFrom(...)  
.returnResultsNow();
```

◆ **асинхронный запрос:**

```
java.util.concurrent.Future<com.google.appengine.api.datastore.QueryResultIterator<  
[имя класса]>> f = datastore.find()  
.type([имя класса].class)  
.addSort("...")  
.addFilter("...", com.google.appengine.api.datastore.Query.FilterOperator.EQUAL, ...)  
.withAncestor(...)  
.startFrom(...)  
.returnResultsLater();
```

Для использования курсора необходимо получить объект `com.google.appengine.api.datastore.Cursor`, используя метод `getCursor()` объекта `QueryResultIterator`, затем определить его для запроса `find()` методом `continueFrom(cursor)`. Метод `maximumResults(int limit)` позволяет ограничить количество возвращаемых результатов для запроса `find()`.

Группировка операций с Datastore-хранилищем в транзакцию осуществляется путем получения объекта `com.google.appengine.api.datastore.Transaction` методом `beginTransaction()` интерфейса `ObjectDatastore`, с дальнейшим использованием аналогично Datastore API.

Для создания в среде Eclipse GAE-приложений, использующих технологию Twig, необходимо скачать библиотеки Twig и Google Collections (<http://code.google.com/p/twig-persist/wiki/IncludingTwig>) и добавить их в папку `war\WEB-INF\lib` каталога GAE-проекта, а также в путь приложения, используя команду **Build Path** контекстного меню окна **Project Explorer** среды Eclipse.

В библиотеке Google Collections полезен для использования класс `Lists`, позволяющий легко конвертировать итератор, возвращаемый запросом `find()` в список `List` с помощью статического метода `Lists.newArrayList([итератор])`.

Для удобства работы с Twig в среде Eclipse можно отключить инструмент **DataNucleus Enhancer**, убрав флажок **Enhancer** в окне **Properties** GAE-проекта.

Для рассмотренного ранее примера с сущностями `Users` и `UsersVideo`, сущность `Users` можно представить в виде класса:

```
package com.example.gae;
import com.google.appengine.api.users.User;
public class Users {
    private String id_user;
    private User user;
    public Users() { }
    public Users(String id_user, User user) {
        this.id_user = id_user;
        this.user = user;
    }
    public String getId_user() {
        return id_user;
    }
    public void setId_user(String id_user) {
        this.id_user = id_user;
    }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}}
```

**А сущность `UsersVideo` — в виде класса:**

```
package com.example.gae;
import java.util.ArrayList;
public class UsersVideo {
    private String id_user;
    private String video_album_name;
    private ArrayList<String> videodata;
    public UsersVideo() { }
    public UsersVideo(String id_user, String video_album_name, ArrayList<String>
videodata) {
        this.id_user = id_user;
        this.video_album_name = video_album_name;
        this.videodata = videodata;
    }
    public String getId_user() {
        return id_user;
    }
}
```

```

    public void setId_user(String id_user) {
        this.id_user = id_user;
    }
    public String getVideo_album_name() {
        return video_album_name;
    }
    public void setVideo_album_name(String video_album_name) {
        this.video_album_name = video_album_name;
    }
    public ArrayList<String> getVideodata() {
        return videodata;
    }
    public void setVideodata(ArrayList<String> videodata) {
        this.videodata = videodata;
    }
}

```

Для JSP-страницы приветствия приложения, отвечающей за аутентификацию пользователя, добавление информации о новом пользователе в Datastore-хранилище, создание нового альбома видео, добавление нового видео в альбом и отображение существующих альбомов с видео пользователя, код изменится следующим образом:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="com.vercer.engine.persist.*" %>
<%@ page import="com.vercer.engine.persist.annotation.*" %>
<%@ page import="com.google.common.collect.*" %>
<%@ page import="com.example.gae.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
ObjectDatastore datastore = new AnnotationObjectDatastore();
if (user != null) {
    java.util.Iterator<Users> usersIt =
        datastore.find().type(Users.class).addFilter("id_user",
            com.google.appengine.api.datastore.Query.FilterOperator.EQUAL,
            user.getUserId()).returnResultsNow();
    java.util.List<Users> results = Lists.newArrayList(usersIt);
    if (results.isEmpty()) {
        Users users=new Users(user.getUserId(),user);

```



```

datastore.store(users);
}
%>
<div style="text-align:left">
<%
response.getWriter().println("Привет "+ user.getNickname());
%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{
response.sendRedirect(userService.createLoginURL("/"));
}
%>
<div style="float:left; width:300px">
<p>Создать альбом:
<form action="/gaeapplication" name="create_album" method="post" >
<input type="hidden" name="htmlFormName" value="create_album"/>
<label>Название альбома:</label>
<input type="text" name="video_album_name" size="30" required>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
<%
java.util.Iterator<UsersVideo> usersVideoIt =
datastore.find().type(UsersVideo.class).addFilter("id_user",
com.google.appengine.api.datastore.Query.FilterOperator.EQUAL,
user.getUserId()).returnResultsNow();
java.util.List<UsersVideo> results = Lists.newArrayList(usersVideoIt);
%>
<p>Добавить видео:
<form action="/gaeapplication" name="add_video" method="post" >
<input type="hidden" name="htmlFormName" value="add_video"/>
В альбом:
<select name="video_album_name">
<%
for (UsersVideo result : results) {
String video_album_name = result.getVideo_album_name();
%>
<option value="<%=video_album_name %>" ><%=video_album_name %></option>
<%
}
%>

```

```

</select>
<br>
<label>Введите адрес видео:</label>
<input type="text" name="video_url" size="30" required>
<br>
<label>Описание видео:</label><br>
<textarea rows="5" cols="45" name="video_desc"></textarea>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
</div>
<div style="float:left; margin-left:200px">
<p>Ваше видео:<br>
<%
for (UsersVideo result : results) {
    String video_album_name=result.getVideo_album_name();
    java.util.ArrayList<String> videodata=result.getVideodata();
}%>
Альбом:<br>
<%=video_album_name%><br>
<%
for (String val : videodata) {
}%>
<%=val %><br>
<%
}}
}%>
</p>
</div>
</body>
</html>

```

Для сервлета приложения, обрабатывающего данные HTML-форм страницы при-  
ветствия приложения, код изменится следующим образом:

```

package com.example.gae;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
import com.vercer.engine.persist.*;

```

```

import com.vercer.engine.persist.annotation.*;
import com.google.common.collect.*;
import com.google.appengine.api.datastore.Query.FilterOperator;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
ObjectDatastore datastore = new AnnotationObjectDatastore();
        if (user != null) {
            String htmlFormName = req.getParameter("htmlFormName");
            if(htmlFormName.equals("create_album")&&
                (!req.getParameter("video_album_name").equals(""))){
                ArrayList<String> videodata=new ArrayList<String>();
                videodata.add("");
                UsersVideo album=new UsersVideo(user.getUserId(),
                    req.getParameter("video_album_name"),videodata);
                datastore.store(album);
            }
            if(htmlFormName.equals("add_video")){
                java.util.Iterator<UsersVideo> usersVideoIt =
datastore.find().type(UsersVideo.class)
                .addFilter("id_user", FilterOperator.EQUAL, user.getUserId())
                .addFilter("video_album_name", FilterOperator.EQUAL,
                    req.getParameter("video_album_name"))
                .addFilter("videodata", FilterOperator.EQUAL, "")
                .returnResultsNow();
                java.util.List<UsersVideo> results = Lists.newArrayList(usersVideoIt);
                if(!results.isEmpty()){
                    for (UsersVideo result : results) {
                        ArrayList<String> videodata=new ArrayList<String>();
                        videodata.add(req.getParameter("video_url"));
                        videodata.add(req.getParameter("video_desc"));
                        Calendar cal = Calendar.getInstance();
                        Date date=cal.getTime();
                        videodata.add(date.toString());
                        result.setVideodata(videodata);
                        datastore.update(result);
                    }
                }else{
                    ArrayList<String> videodata=new ArrayList<String>();
                    videodata.add(req.getParameter("video_url"));
                    videodata.add(req.getParameter("video_desc"));
                    Calendar cal = Calendar.getInstance();
                    Date date=cal.getTime();
                    videodata.add(date.toString());
                    UsersVideo usersvideo=new
                    UsersVideo(user.getUserId(), req.getParameter("video_album_name"), videodata);

```

```
        datastore.store(usersvideo);
    }}
    resp.sendRedirect("/");
} else { resp.sendRedirect(userService.createLoginURL("/"));
}}}
```

## Slim3

Технология Slim3 (<https://sites.google.com/site/slim3appengine/Home>) представляет собой MVC-фреймворк, созданный для платформы Google App Engine/Java и Datastore-хранилища.

Для того чтобы воспользоваться данным фреймворком, необходимо установить для среды Eclipse с установленным плагином Google Plugin for Eclipse (GPE) дополнительно Slim3-плагин. Для этого в среде Eclipse в меню **Help** выберем команду **Install New Software** и в поле **Work with** добавим адрес <http://slim3.googlecode.com/svn/updates/>, отметим флажок **Slim3** и нажмем кнопку **Next**. После установки Slim3-плагины и перезапуска среды Eclipse в команду **New | Other** добавится раздел **Slim3** с мастерами **GWT Module** и **Slim3 Project**.

После создания проекта Slim3-приложения с помощью мастера **Slim3 Project** дополнительно для тестирования Slim3-приложения в поле **Number of static imports needed for .\*** раздела **Preferences | Java | Code Style | Organize Imports** меню **Window** среды Eclipse нужно ввести 1. В разделе **Preferences | Java | Editor | Content Assist | Favorites** меню **Window** с помощью кнопки **New Type** добавить `org.hamcrest.CoreMatchers.*`, `org.junit.Assert.*` и `org.junit.matchers.JUnitMatchers.*`. Отметить флажок **Refresh automatically** в разделе **Preferences | General | Workspace**.

Для сборки проекта Slim3-приложения в окне **Project Explorer** щелкнем правой кнопкой мыши на узле **build.xml** проекта и в контекстном меню выберем команду **Run As | Ant Build**. В диалоговом окне **Ant Input Request** введем относительный путь запроса к приложению (например, `/gaeapplication/`) и нажмем кнопку **OK**. В результате после запуска приложения с помощью команды **Run As | Web Application** и ввода в адресной строке Web-браузера адреса <http://localhost:8888/gaeapplication/> откроется Web-страница приложения, отображающая:

```
Hello gaeapplication Index !!!
```

### ПРИМЕЧАНИЕ

Технология Slim3 поддерживает автоматическую перезагрузку измененных ресурсов приложения, поэтому после редактирования кода не требуется перезапуск приложения.

В результате выполнения Ant-задачи `gen-controller` файла `build.xml` в папке `src` проекта был сгенерирован пакет `xxx.controller.gaeapplication` с классом `IndexController`, являющимся надстройкой `Servlet`-класса. Фреймворк Slim3 обеспечивает автоматическое связывание классов-контроллеров с URL-адресами HTTP-запросов на основе имен пакетов контроллеров. В частности, сгенерированный класс `IndexController`

автоматически вызывается при наборе относительного адреса `/gaeapplication/` и в своем коде перенаправляет на JSP-страницу `index.jsp` папки `gaeapplication/` приложения.

Для создания модели — классов, представляющих данные приложения, откроем файл `build.xml` и в окне **Outline** среды Eclipse щелкнем правой кнопкой мыши на узле Ant-задачи `gen-model` >  `gen-model` и в контекстном меню выберем команду **Run As | Ant Build**. В диалоговом окне **Ant Input Request** введем имя класса `Users` и нажмем кнопку **ОК**. В результате в папке `src` проекта будет сгенерирован пакет `xxx.model` с классом `Users`, представляющим сущность Datastore-хранилища, а также пакет `xxx.meta` с классом `UsersMeta`, представляющим метамодель модели `Users` и обеспечивающим синхронизацию модели с данными Datastore-хранилища.

Добавим поля и конструкторы в класс `Users`:

```
private String id_user;
private User user;
public Users() { }
public Users(String id_user, User user) {
    this.id_user = id_user;
    this.user = user;
}
public String getId_user() {
    return id_user;
}
public void setId_user(String id_user) {
    this.id_user = id_user;
}
public User getUser() {
    return user;
}
public void setUser(User user) {
    this.user = user;
}
```

Аналогично создадим класс сущности `UsersVideo` и его метамодель `UsersVideoMeta`.

Добавим поля и конструкторы в класс `UsersVideo`:

```
private String id_user;
private String video_album_name;
private ArrayList<String> videodata;
public UsersVideo() { }
public UsersVideo(String id_user, String video_album_name, ArrayList<String> videodata)
{
    this.id_user = id_user;
    this.video_album_name = video_album_name;
    this.videodata = videodata;
}
```

```

public String getId_user() {
    return id_user;
}
public void setId_user(String id_user) {
    this.id_user = id_user;
}
public String getVideo_album_name() {
    return video_album_name;
}
public void setVideo_album_name(String video_album_name) {
    this.video_album_name = video_album_name;
}
public ArrayList<String> getVideodata() {
    return videodata;
}
public void setVideodata(ArrayList<String> videodata) {
    this.videodata = videodata;
}

```

**Изменим код JSP-страницы index.jsp папки war\gaeapplication, вызываемой IndexController-контроллером:**

```

<%@page pageEncoding="UTF-8" isELIgnored="false" session="false"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>
<%@taglib prefix="f" uri="http://www.slim3.org/functions"%>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="com.example.gae.model.*" %>
<%@ page import="com.example.gae.meta.*" %>
<%@ page import="org.slim3.datastore.Datastore" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
if (user != null) {
    UsersMeta u = UsersMeta.get();
    java.util.List<Users> results =
Datastore.query(u).filter(u.id_user.equal(user.getUserId())).asList();
    if (results.isEmpty()) {
        Users users=new Users(user.getUserId(),user);
        Datastore.put(users);
    }
}%>

```

```

<div style="text-align:left">
<%
response.getWriter().println("Привет "+ user.getNickname());
%>
<jsp:element name="a">
<jsp:attribute
name="href"><%=userService.createLogoutURL("/gaeapplication")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{
    response.sendRedirect(userService.createLoginURL("/gaeapplication"));
}
%>
<div style="float:left; width:300px">
<p>Создать альбом:
<form action="/gaeapplication" name="create_album" method="get" >
<input type="hidden" name="htmlFormName" value="create_album"/>
<label>Название альбома:</label>
<input type="text" name="video_album_name" size="30" required>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
<%
UsersVideoMeta uv = UsersVideoMeta.get();
java.util.List<UsersVideo> results =
Datastore.query(uv).filter(uv.id_user.equal(user.getUserId())) .asList();
%>
<p>Добавить видео:
<form action="/gaeapplication" name="add_video" method="get" >
<input type="hidden" name="htmlFormName" value="add_video"/>
В альбом:
<select name="video_album_name">
<%
for (UsersVideo result : results) {
    String video_album_name = result.getVideo_album_name();
%>
<option value="<%=video_album_name %>" ><%=video_album_name %></option>
<%
}
%>
</select>
<br>
<label>Введите адрес видео:</label>

```

```

<input type="text" name="video_url" size="30" required>
<br>
<label>Описание видео:</label><br>
<textarea rows="5" cols="45" name="video_desc"></textarea>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
</div>
<div style="float:left; margin-left:200px">
<p>Ваше видео:<br>
<%
for (UsersVideo result : results) {
    String video_album_name=result.getVideo_album_name();
    java.util.ArrayList<String> videodata=result.getVideodata();
}%>
Альбом:<br>
<%=video_album_name%><br>
<%
for (String val : videodata) {
}%>
<%=val %><br>
<%
}}
}%>
</p>
</div>
</body>
</html>

```

Для обработки данных HTML-форм страницы `index.jsp` изменим код класса `IndexController`:

```

package com.example.gae.controller.gaeapplication;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import org.slim3.controller.Controller;
import org.slim3.controller.Navigation;
import org.slim3.datastore.Datastore;
import org.slim3.util.RequestLocator;
import com.example.gae.meta.UsersVideoMeta;
import com.example.gae.model.UsersVideo;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
public class IndexController extends Controller {
    @Override

```



```

public Navigation run() throws Exception {
    UserService userService = UserServiceFactory.getUserService();
    User user = userService.getCurrentUser();
    if (user != null) {
String htmlFormName = RequestLocator.get().getParameter("htmlFormName");
if(htmlFormName!=null&&htmlFormName.equals("create_album")&&(!RequestLocator.get().
getParameter("video_album_name").equals(""))){
    ArrayList<String> videodata=new ArrayList<String>();
    videodata.add("");
    UsersVideo album=new UsersVideo(user.getUserId(),
RequestLocator.get().getParameter("video_album_name"),videodata);
    Datastore.put(album);
}
if(htmlFormName!=null&&htmlFormName.equals("add_video")){
    UsersVideoMeta uv = UsersVideoMeta.get();
    java.util.List<UsersVideo> results =
Datastore.query(uv).filter(uv.id_user.equal(user.getUserId()),
uv.video_album_name.equal(RequestLocator.get().getParameter("video_album_name")),
uv.videodata.equal("")).asList();
    if(!results.isEmpty()){
        for (UsersVideo result : results) {
            ArrayList<String> videodata=new ArrayList<String>();
            videodata.add(RequestLocator.get().getParameter("video_url"));
            videodata.add(RequestLocator.get().getParameter("video_desc"));
            Calendar cal = Calendar.getInstance();
            Date date=cal.getTime();
            videodata.add(date.toString());
            result.setVideodata(videodata);
            Datastore.put(result);
        }
    }else{
        ArrayList<String> videodata=new ArrayList<String>();
        videodata.add(RequestLocator.get().getParameter("video_url"));
        videodata.add(RequestLocator.get().getParameter("video_desc"));
        Calendar cal = Calendar.getInstance();
        Date date=cal.getTime();
        videodata.add(date.toString());
        UsersVideo usersvideo=new
UsersVideo(user.getUserId(),RequestLocator.get().getParameter("video_album_name"),
videodata);
        Datastore.put(usersvideo);
    }
    return forward("index.jsp");
}else { return forward(userService.createLoginURL("/gaeapplication"));
}}

```

Для устранения ошибки "class x.y.z.MyClass cannot be cast to class x.y.z.MyClass" закомментируем Hot-фильтр дескриптора web.xml:

```

<!--
<filter>
<filter-name>HotReloadingFilter</filter-name>
<filter-class>org.slim3.controller.HotReloadingFilter</filter-class>
</filter>
-->
<!--
<filter-mapping>
<filter-name>HotReloadingFilter</filter-name>
<url-pattern>/*</url-pattern>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
-->

```

В рамках технологии Slim3 классы сущностей маркируются аннотацией `@Model`. Поле, представляющее ключ-идентификатор сущности, маркируется аннотацией `@Attribute(primaryKey=true)`. Поля, не предназначенные для сохранения в Datastore-хранилище, маркируются аннотацией `@Attribute(persistent=false)`.

Поле типа объекта сериализуемого класса или типа байтового массива с аннотацией `@Attribute(lob=true)` сохраняется в виде Blob-значения. Поле типа строки с аннотацией `@Attribute(lob=true)` сохраняется в виде Text-значения.

Неиндексируемые поля маркируются аннотацией `@Attribute(unindexed=true)`.

В технологии Slim3 за взаимодействие с Datastore-хранилищем отвечает класс `org.slim3.datastore.Datastore`.

Для объединения сущностей в одну группу используется метод `allocateId()` класса `Datastore`:

```

import com.google.appengine.api.datastore.Key;
import org.slim3.datastore.Datastore;
...
Key parentKey = ...;
Key childKey = Datastore.allocateId(parentKey, Child.class);

```

Методы `put()`, `putAsync()`, `get()`, `getAsync()`, `delete()` и `deleteAsync()` класса `Datastore` обеспечивают синхронное и асинхронное сохранение, обновление, извлечение и удаление сущностей в Datastore-хранилище.

Запросы к Datastore-хранилищу осуществляются с помощью метода `query()` класса `Datastore`:

```

List<[имя сущности]> list = Datastore.query(экземпляр сущности)
    .filter(...)
    .sort()
    .offset(...)
    .limit(...)
    .asList();

```

Запрос с фильтром по ключу родительской сущности выполняется с помощью метода:

```
Key ancestorKey = ...;  
List<Child> list = Datastore.query(Child.class, ancestorKey).asList();
```

Для использования курсора результаты запроса к хранилищу возвращают в виде коллекции `org.slim3.datastore.S3QueryResultList<T>`:

```
S3QueryResultList<...> results = Datastore.query([...].class)  
.limit(...).asQueryResultList();
```

Затем получают параметры курсора:

```
String encodedCursor = results.getEncodedCursor();  
String encodedFilters = results.getEncodedFilters();  
String encodedSorts = results.getEncodedSorts();
```

И передают их в запрос:

```
S3QueryResultList<...> results2 = Datastore.query([...].class)  
.encodedCursor(encodedCursor)  
.encodedFilters(encodedFilters)  
.encodedSorts(encodedSorts)  
.limit(...)  
.asQueryResultList();
```

Группировка операций с Datastore-хранилищем в транзакцию осуществляется путем получения объекта `com.google.appengine.api.datastore.Transaction` методом `beginTransaction()` класса `Datastore`, с дальнейшим использованием аналогично Datastore API. Технология Slim3 автоматически поддерживает работу с сущностями разных групп внутри транзакции.

## Google Cloud SQL

Для использования сервиса Google Cloud SQL необходимо открыть консоль Google APIs Console и создать проект (рис. 6.4).

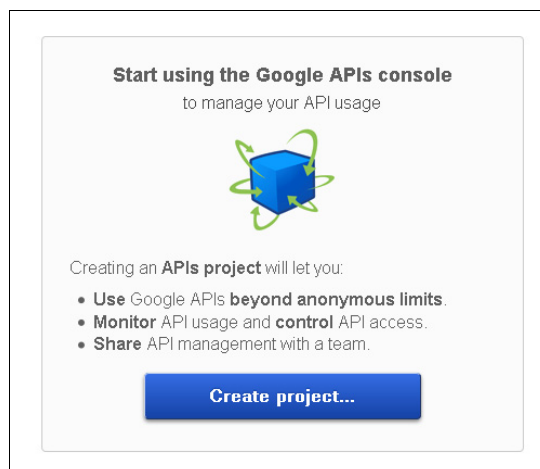


Рис. 6.4. Страница консоли Google APIs Console

 Freebase API		<input type="checkbox"/> OFF	Courtesy limit: 100,000 requests/day
 Google Affiliate Network API		<input type="checkbox"/> OFF	Courtesy limit: 1,000 requests/day
 Google Cloud SQL		<a href="#">Request access...</a>	<a href="#">Pricing</a>
 Google Cloud Storage		<input type="checkbox"/> OFF	<a href="#">Pricing</a>

Рис. 6.5. Страница консоли Google APIs Console с перечнем сервисов

После нажатия кнопки **Create project** откроется страница с перечнем Google-сервисов (рис. 6.5).

После нажатия ссылки **Request access...** откроется страница, на которой нужно ввести информацию для запроса на получение доступа к сервису Google Cloud SQL. Если введенная и отправленная информация с данной страницы убедит команду Google, вместо ссылки **Request access...** появится кнопка **ON/OFF** включения сервиса. При этом необходимо учитывать, что данный сервис является платным.

После включения сервиса, в левой панели выбора проекта консоли необходимо нажать на пункт **Google Cloud SQL** и создать экземпляр базы данных.

К созданному экземпляру базы данных требуется разрешить доступ для GAE-приложений, используя раздел **Authorized applications** консоли Google APIs Console.

Для создания схемы базы данных и ее таблиц можно использовать панель **SQL Prompt** консоли Google APIs Console, позволяющей вводить и выполнять SQL-выражения.

#### ПРИМЕЧАНИЕ

При создании схемы базы данных Cloud SQL и ее таблиц нужно учитывать, что их имена чувствительны к регистру.

Для соединения с сервисом необходимо использовать драйвер `com.google.appengine.api.rdbms.AppEngineDriver`, который нужно зарегистрировать с помощью статического метода:

```
java.sql.DriverManager.registerDriver(new AppEngineDriver());
```

Далее можно устанавливать соединение с базой данных:

```
java.sql.Connection c = java.sql.DriverManager.getConnection("jdbc:google:rdbms://instance_name/database", "user", "password");
```

После установки соединения можно использовать программный интерфейс JDBC API для взаимодействия с базой данных.

При разработке GAE-приложения, хранящего данные в базе Cloud SQL, в среде Eclipse с GPE-плагином раздел **Google | App Engine** диалогового окна **Properties** свойств GAE-проекта позволяет с помощью флажка **Enable Google Cloud SQL** включить использование базы данных Cloud SQL (рис. 6.6). При этом необходимо

с помощью ссылок **Configure** ввести данные для локального экземпляра базы данных и для экземпляра базы данных GAE-платформы — зарегистрированные в консоли Google APIs Console имя экземпляра базы данных, имя схемы базы данных, логин и пароль доступа к базе данных.

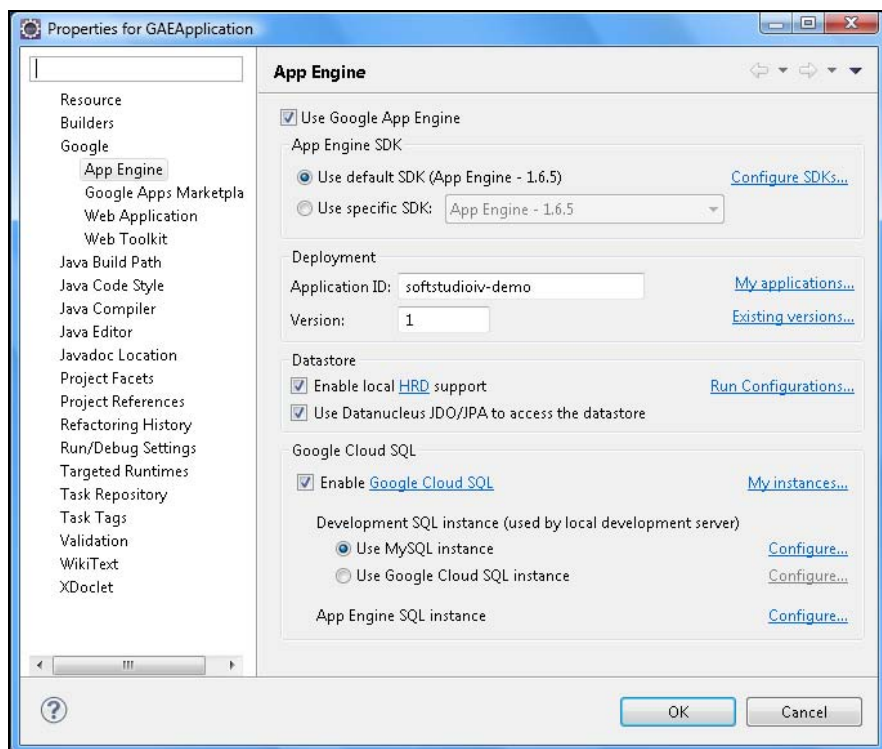


Рис. 6.6. Включение использования базы данных Cloud SQL в свойствах проекта

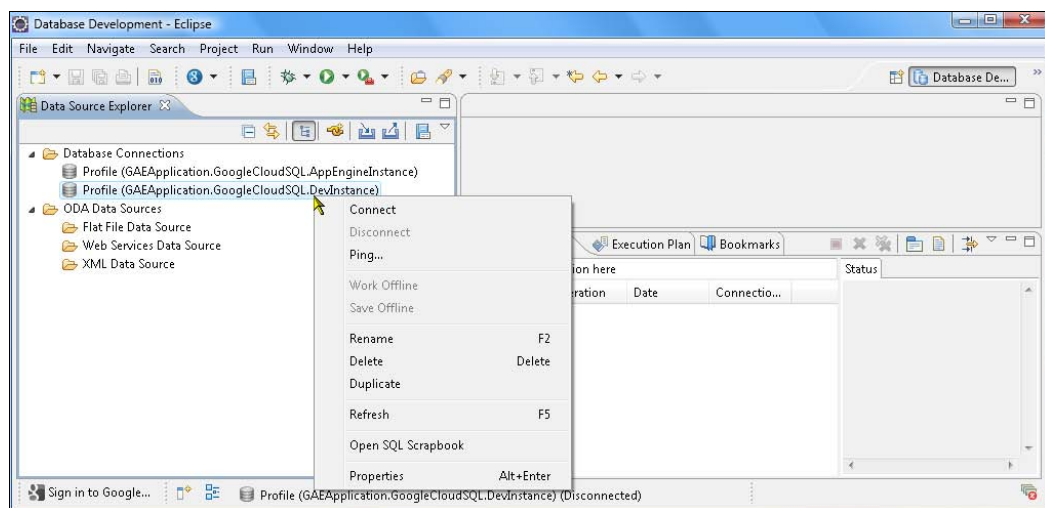


Рис. 6.7. Перспектива Database Development среды Eclipse IDE for Java EE Developers

Использование среды Eclipse IDE for Java EE Developers с GPE-плагином дает возможность применять перспективу **Database Development**, которая упрощает управление соединениями с базой данных (рис. 6.7).

Кроме того, раздел **Project Facets** диалогового окна **Properties** свойств GAE-проекта позволяет с помощью флажка **JPA** включить поддержку технологии Java Persistence API (JPA) для GAE-проекта.

## Google Cloud Storage и Blobstore

### Google Cloud Storage

Для использования сервиса Google Cloud Storage его необходимо активировать в консоли Google APIs Console с помощью кнопки **ON/OFF** списка сервисов. При этом следует учитывать, что сервис Google Cloud Storage является платным.

После активации сервиса необходимо создать экземпляр (bucket) хранилища Google Cloud Storage. Сделать это можно, воспользовавшись страницей менеджера Google Cloud Storage Manager, ссылка на которую присутствует на странице активированного сервиса Google Cloud Storage консоли Google APIs Console (рис. 6.8).

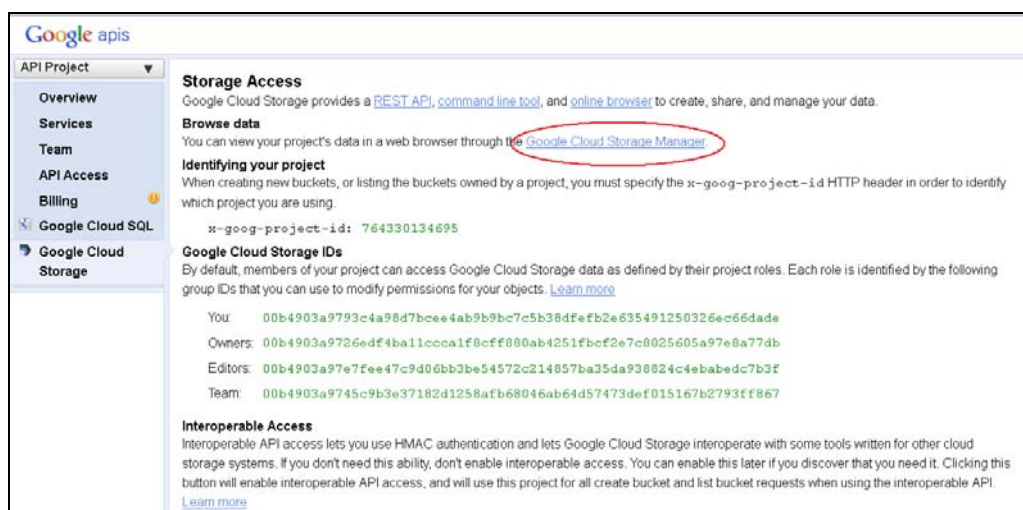


Рис. 6.8. Страница активированного сервиса Google Cloud Storage консоли Google APIs Console

Кнопка **New Bucket** менеджера Google Cloud Storage Manager позволяет создать новый экземпляр хранилища (рис. 6.9).



Рис. 6.9. Страница менеджера Google Cloud Storage Manager

После создания bucket-контейнера требуется разрешить доступ к нему для GAE-приложения. Для этого в консоли администрирования приложением в разделе **Administration | Application Settings** скопируем строку **Service Account Name** и вставим ее в поле **Add a teammate** раздела **Team** консоли **Google APIs Console**.

Доступ к хранилищу Cloud Storage может осуществляться в коде приложения с помощью программного интерфейса RESTful API или посредством программного интерфейса Google Cloud Storage Java API.

Для организации взаимодействия с хранилищем Cloud Storage с помощью программного интерфейса Google Cloud Storage Java API в качестве первого шага необходимо создать объект `com.google.appengine.api.files.FileService`, используя метод `getFileService()` класса-фабрики `com.google.appengine.api.files.FileServiceFactory`:

```
FileService fileService = FileServiceFactory.getFileService();
```

Затем требуется определить параметры сохраняемого объекта для экземпляра класса `com.google.appengine.api.files.GSFileOptions.GSFileOptionsBuilder`:

```
GSFileOptionsBuilder optionsBuilder = new GSFileOptionsBuilder()
    .setBucket("[имя зарегистрированного bucket-контейнера]")
    .setKey("[имя сохраняемого файла]")
    .setMimeType("text/html")
    .setAcl("public_read")
    .addUserMetadata("field", "value");
```

После этого объект создается методом `createNewGSFile()` интерфейса `FileService`:

```
com.google.appengine.api.files.AppEngineFile writableFile =
fileService.createNewGSFile(optionsBuilder.build());
```

После создания объекта в bucket-контейнере в него можно записать данные. Для этого требуется открыть канал записи:

```
boolean lock = false;
com.google.appengine.api.files.FileWriteChannel writeChannel =
fileService.openWriteChannel(writableFile, lock);
```

и произвести запись данных посредством метода `int write(java.nio.ByteBuffer src, java.lang.String sequenceKey)` интерфейса `FileWriteChannel` или стандартными методами записи в канал библиотеки `java.io`.

После записи данных перед их чтением сохраняемый объект необходимо закрыть:

```
writeChannel.closeFinally();
```

Для обновления данных в финализированном объекте требуется создать новый объект с тем же именем и перезаписать его.

Для чтения данных из объекта bucket-контейнера последовательность действий та же, что и при записи — необходимо открыть канал и прочитать из него:

```
String filename = "/gs/[имя зарегистрированного bucket-контейнера]/[имя сохраняемого файла]";
AppEngineFile readableFile = new AppEngineFile(filename);
```

```
FileReadChannel readChannel = fileService.openReadChannel(readableFile, false);
BufferedReader reader = new BufferedReader(Channels.newReader(readChannel, "UTF8"));
String line = reader.readLine();
readChannel.close();
```

## Blobstore

Взаимодействовать с Blobstore-хранилищем GAE-платформы можно двумя способами. Первый способ — это использование библиотеки `com.google.appengine.api.files` аналогично взаимодействию с хранилищем Cloud Storage. Второй способ — использование программного интерфейса Blobstore API.

При использовании библиотеки `com.google.appengine.api.files` в качестве первого шага создается объект `com.google.appengine.api.files.FileService` с помощью метода `getFileService()` класса-фабрики `com.google.appengine.api.files.FileServiceFactory`:

```
FileService fileService = FileServiceFactory.getFileService();
```

Затем создается объект `com.google.appengine.api.files.AppEngineFile`:

```
AppEngineFile file = fileService.createNewBlobFile("[mime-type]");
```

Далее открывается канал записи:

```
boolean lock = false;
com.google.appengine.api.files.FileWriteChannel writeChannel =
fileService.openWriteChannel(file, lock);
```

в который производится запись данных посредством метода `int write(java.nio.ByteBuffer src, java.lang.String sequenceKey)` интерфейса `FileWriteChannel` или стандартных методов записи в канал библиотеки `java.io`.

Для доступа к созданному объекту нужно сохранить его путь:

```
String path = file.getFullPath();
```

После записи данных перед их чтением созданный объект необходимо закрыть:

```
writeChannel.closeFinally();
```

Для чтения данных из Blobstore-хранилища последовательность действий та же, что и при записи — необходимо открыть канал и прочитать из него:

```
file = new AppEngineFile(path);
lock = false;
FileReadChannel readChannel = fileService.openReadChannel(file, lock);
...
readChannel.close();
```

Применение программного интерфейса Blobstore API для взаимодействия с Blobstore-хранилищем предполагает совместную работу HTML-форм и Java-кода, использующего библиотеку `com.google.appengine.api.blobstore`.

В качестве примера рассмотрим GAE-приложение, Web-страница которого отображает HTML-форму загрузки изображения в Blobstore-хранилище и показывает все изображения, хранящиеся для данного пользователя.



JSP-страница приветствия приложения будет иметь следующий код:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="com.google.appengine.api.datastore.*" %>
<%@ page import="com.google.appengine.api.blobstore.*" %>
<%@ page import="com.google.appengine.api.images.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>App Engine Application</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
if (user != null) {
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query qUsers = new Query("Users");
PreparedQuery pq = datastore.prepare(qUsers);
int auth=0;
for (Entity result : pq.asIterable()) {
    String id_user = (String) result.getProperty("id_user");
    if(id_user.equals(user.getUserId())){
        auth=1;
    }
}
if(auth==0){
    Entity Users = new Entity("Users");
    Users.setProperty("id_user", user.getUserId());
    Users.setProperty("user", user);
    datastore.put(Users);}
%>
<div style="text-align:left">
<%
response.getWriter().println("Привет " + user.getNickname());
%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{
response.sendRedirect(userService.createLoginURL("/"));
}
%>
```

```

<%
BlobstoreService blobstoreService = BlobstoreServiceFactory.getBlobstoreService();
%>
<div style="float:left; width:300px">
<p>Загрузите изображение:
<form action="<%= blobstoreService.createUploadUrl("/gaeapplication") %>" method="post"
enctype="multipart/form-data">
<input type="file" name="myImage" required><br><br>
<p><input type="submit" value="Отправить">
<input type="reset" value="Очистить"></p>
</form>
</p>
</div>
<div style="float:left; margin-left:200px">
<p>Ваши изображения:<br>
<%
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query qUsersImage = new Query("UsersImage");
qUsersImage.addFilter("id_user", Query.FilterOperator.EQUAL, user.getUserId());
PreparedQuery pq = datastore.prepare(qUsersImage);
java.util.List<Entity> list = pq.asList(FetchOptions.Builder.withDefaults());
BlobInfoFactory fact=new BlobInfoFactory();
for (Entity result : list) {
    BlobKey blobkey=(BlobKey) result.getProperty("blobKey");
    ImagesService imagesService = ImagesServiceFactory.getImagesService();
    String url=imagesService.getServingUrl(blobkey,100,true);
%>
<!--

-->

<br>
<%
}
%>
</p>
</div>
</body>
</html>

```

В коде JSP-страницы приветствия приложения производится регистрация нового пользователя с помощью создания Entity-сущности Users Datastore-хранилища. Форма загрузки изображения страницы обрабатывается действием blobstoreService.createUploadUrl(), обеспечивающим загрузку файла непосредственно в Blobstore-хранилище, после чего формируется запрос по URL-адресу, указанному в качестве аргумента метода createUploadUrl(). Данный запрос содержит в качестве параметра ключ сохраненного файла, обеспечивающий к нему дальнейший доступ. Сервлет, отвечающий за обработку такого запроса, будет иметь следующий код:

```

package com.example.gae;
import java.io.IOException;
import java.util.Map;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.blobstore.*;
import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
        BlobstoreService blobstoreService = BlobstoreServiceFactory.getBlobstoreService();
        if (user != null) {
            Map<String, java.util.List<BlobKey>> blobs = blobstoreService.getUploads(req);
            java.util.List<BlobKey> blobsKey = blobs.get("myImage");
            if (blobsKey.isEmpty()) {
                resp.sendRedirect("/");
            } else {
                for (BlobKey result : blobsKey) {
                    Entity UsersImage = new Entity("UsersImage");
                    UsersImage.setProperty("id_user", user.getUserId());
                    UsersImage.setProperty("blobKey", result);
                    datastore.put(UsersImage);
                    //resp.sendRedirect("/serve?blob-key=" + result.getKeyString());
                }
                resp.sendRedirect("/");
            } else { resp.sendRedirect(userService.createLoginURL("/"));
        }
    }
}

```

В коде сервлета ключ `BlobKey`, соответствующий полю HTML-формы загрузки изображения, извлекается из запроса и сохраняется в качестве свойства `Entity`-сущности `Datastore`-хранилища.

Если раскомментировать строку:

```
//resp.sendRedirect("/serve?blob-key=" + result.getKeyString());
```

то будет вызван сервлет:

```

package com.example.gae;
import java.io.IOException;

```

```

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.blobstore.BlobKey;
import com.google.appengine.api.blobstore.BlobstoreService;
import com.google.appengine.api.blobstore.BlobstoreServiceFactory;
@SuppressWarnings("serial")
public class ServeImage extends HttpServlet {
    private BlobstoreService blobstoreService =
        BlobstoreServiceFactory.getBlobstoreService();
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException {
        BlobKey blobKey = new BlobKey(req.getParameter("blob-key"));
        blobstoreService.serve(blobKey, res);
    }
}
//web.xml:
<servlet>
    <servlet-name>ServeImage</servlet-name>
    <servlet-class>com.example.gae.ServeImage</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ServeImage</servlet-name>
    <url-pattern>/serve</url-pattern>
</servlet-mapping>

```

Данный сервлет извлекает ключ файла из запроса, извлекает файл по ключу из Blobstore-хранилища и передает его в составе ответа клиенту. При этом Web-браузер отобразит сохраненное изображение в его исходном состоянии.

Однако это не всегда удобно, поэтому в оставшейся части кода JSP-страницы приложения из Datastore-хранилища извлекаются ключи хранящихся в Blobstore-хранилище изображений, на основе которых изображения трансформируются с помощью Google-сервиса изображений и уже измененные изображения отображаются на Web-странице.

## Сервис изображений

Для использования сервиса изображений в качестве первого шага необходимо создать объект `com.google.appengine.api.images.ImagesService`, используя статический метод `getImagesService()` класса-фабрики `com.google.appengine.api.images.ImagesServiceFactory`.

Метод `getServingUrl()` интерфейса `ImagesService` позволяет получить на основе ключа `BlobKey` URL-адрес автоматически трансформированного изображения с измененными размерами и определить его в качестве значения атрибута `src` HTML-тега `<img>`.

В качестве значения атрибута `src` HTML-тега `<img>` можно определить и запрос к сервлету, передавая с ним ключ хранящегося в Blobstore-хранилище изображения. При этом вызываемый сервлет будет иметь следующий код:

```

import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.blobstore.*;
import com.google.appengine.api.images.*;
@SuppressWarnings("serial")
public class ServeImageByte extends HttpServlet {
    private BlobstoreService blobstoreService =
        BlobstoreServiceFactory.getBlobstoreService();
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException {
        BlobKey blobKey = new BlobKey(req.getParameter("blob-key"));
        BlobInfoFactory fact=new BlobInfoFactory();
        BlobInfo binf=fact.loadBlobInfo(blobKey);
        long size=binf.getSize();
        byte[] imgData= blobstoreService.fetchData(blobKey, 0, size);
        ImagesService imagesService = ImagesServiceFactory.getImagesService();
        Image oldImage = ImagesServiceFactory.makeImage(imgData);
        Transform resize = ImagesServiceFactory.makeResize(100, 100);
        Image newImage = imagesService.applyTransform(resize, oldImage);
        byte[] newImageData = newImage.getImageData();
        res.setContentType("image/jpeg");
        res.setContentLength(newImageData.length);
        res.getOutputStream().write(newImageData);
        res.getOutputStream().flush();
        res.getOutputStream().close();
    }
}
//web.xml:
<servlet>
    <servlet-name>ServeImageByte</servlet-name>
    <servlet-class>com.example.gae.ServeImageByte</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ServeImageByte</servlet-name>
    <url-pattern>/servebyte</url-pattern>
</servlet-mapping>

```

В коде сервлета из запроса извлекается ключ `BlobKey` и на его основе получается объект `BlobInfo`, содержащий информацию о хранящемся изображении. Объект `BlobInfo` используется для получения размера хранящегося изображения, который фигурирует в качестве аргумента метода `fetchData()` интерфейса `BlobstoreService` чтения данных изображения из хранилища. Далее для преобразования полученных данных используется интерфейс `ImagesService`, и измененные данные изображения записываются в ответ сервлета.

Помимо изменения размеров методы `makeComposite()`, `makeCrop()`, `makeHorizontalFlip()`, `makeVerticalFlip()`, `makeRotate()` и `makeImFeelingLucky()` класса-фабрики `ImagesServiceFactory` обеспечивают композицию, кадрирование, отражение, вращение и улучшение качества изображения.

## Служба Memcache

Для сохранения данных GAE-приложения в распределенном кэше памяти платформы App Engine можно использовать библиотеку программного интерфейса JCACHE (Java Temporary Caching API) или программный интерфейс самой службы Memcache Java API.

При использовании программного интерфейса JCACHE, представленного пакетом `net.sf.jsr107`, в качестве первого шага необходимо получить объект фабрики кэша и создать экземпляр кэша:

```
Cache cache;
try {
    CacheFactory cacheFactory = CacheManager.getInstance().getCacheFactory();
    cache = cacheFactory.createCache(Collections.emptyMap());
} catch (CacheException e) {
}
```

где `Collections.emptyMap()` — набор свойств кэша.

Далее экземпляр кэша с помощью своих методов `put(key, value)` и `get(key)` позволяет сохранять и извлекать данные в виде пар "ключ — значение".

Метод `putAll(java.util.Map map)` обеспечивает сохранение экземпляра таблицы `java.util.HashMap`, метод `remove(key)` — удаление пары из кэша, метод `clear()` — очистку кэша.

Метод `containsKey(key)` позволяет узнать наличие пары в кэше, метод `isEmpty()` — проверить заполненность кэша, метод `size()` — узнать размер кэша.

Набор свойств кэша можно определить в виде таблицы `java.util.HashMap`, передав ее в качестве аргумента методу `createCache()`. При этом в таблицу можно добавить следующие свойства:

- ◆ `GCacheFactory.EXPIRATION_DELTA`, `GCacheFactory.EXPIRATION_DELTA_MILLIS` или `GCacheFactory.EXPIRATION` — указывают желаемое время хранения данных в кэше в секундах, миллисекундах или `java.util.Date`, хотя реальное время хранения данных в кэше зависит от работы платформы App Engine;
- ◆ `MemcacheService.SetPolicy.SET_ALWAYS` (по умолчанию), `MemcacheService.SetPolicy.ADD_ONLY_IF_NOT_PRESENT` или `MemcacheService.SetPolicy.REPLACE_ONLY_IF_PRESENT` — значение добавляется или заменяет существующее, значение добавляется при его отсутствии или значение только заменяет существующее.

Программный интерфейс Memcache Java API представлен пакетом `com.google.appengine.api.memcache`.

В отличие от JCACHE программный интерфейс Memcache Java API позволяет сохранять данные в GAE-кэше как синхронно, так и асинхронно. Для синхронного взаимодействия с GAE-кэшем нужно создать объект `MemcacheService`, а для асинхронной работы с GAE-кэшем — объект `AsyncMemcacheService`:

```
MemcacheService syncCache = MemcacheServiceFactory.getMemcacheService();  
AsyncMemcacheService asyncCache = MemcacheServiceFactory.getAsyncMemcacheService();
```

Далее методы объекта `MemcacheService` или объекта `AsyncMemcacheService` обеспечивают сохранение пары "ключ — значение", извлечение значения по ключу и удаление данных из GAE-кэша, при этом методы интерфейса `AsyncMemcacheService` возвращают объект `java.util.concurrent.Future`, зафиксировать который можно методом `get()`:

- ◆ `get(java.lang.Object key)` или `getAll(java.util.Collection<T> keys)` — извлекают значение по ключу;
- ◆ `getIdentifiable(java.lang.Object key)` или `getIdentifiables(java.util.Collection<T> keys)` — извлекают объект, который может быть использован в качестве аргумента `oldValue` метода `putIfUntouched(java.lang.Object key, MemcacheService.IdentifiableValue oldValue, java.lang.Object newValue, Expiration expires)`, сохраняющего значение `newValue` только в том случае, если не было другого сохранения с момента получения значения `oldValue`;
- ◆ `contains(java.lang.Object key)` — проверяет наличие пары в кэше;
- ◆ `put(java.lang.Object key, java.lang.Object value, Expiration expires, MemcacheService.SetPolicy policy)` или `put(java.lang.Object key, java.lang.Object value, Expiration expires)`, или `putAll(java.util.Map<T, ?> values, Expiration expires, MemcacheService.SetPolicy policy)`, или `putAll(java.util.Map<T, ?> values, Expiration expires)`, или `putAll(java.util.Map<T, ?> values)` — сохраняют пару "ключ — значение" в кэше, где объект `Expiration` создается статическим методом `byDeltaMillis(int milliDelay)` или `byDeltaSeconds(int secondsDelay)`, а объект `MemcacheService.SetPolicy` представляет собой перечисление с полями `ADD_ONLY_IF_NOT_PRESENT`, `REPLACE_ONLY_IF_PRESENT` и `SET_ALWAYS` (по умолчанию);
- ◆ `putIfUntouched(java.lang.Object key, MemcacheService.IdentifiableValue oldValue, java.lang.Object newValue, Expiration expires)` или `putIfUntouched(java.lang.Object key, MemcacheService.IdentifiableValue oldValue, java.lang.Object newValue)`, или `putIfUntouched(java.util.Map<T, MemcacheService.CasValues> values)`, или `putIfUntouched(java.util.Map<T, MemcacheService.CasValues> values, Expiration expiration)` — сохраняют значение `newValue` только в том случае, если не было другого сохранения с момента получения значения `oldValue`;
- ◆ `delete(java.lang.Object key)` или `delete(java.lang.Object key, long millisNoReAdd)`, или `deleteAll(java.util.Collection<T> keys)`, или `deleteAll(java.util.Collection<T> keys, long millisNoReAdd)` — удаляют пару из кэша;
- ◆ `increment(java.lang.Object key, long delta)` или `increment(java.lang.Object key, long delta, java.lang.Long initialValue)`, или `incrementAll(java.util.Collection<T> keys, long delta)`, или `incrementAll(java.util.Collection<T> keys, long delta, java.lang.Long initialValue)`, или `incrementAll(java.util.Map<T, java.lang.Long> offsets)`, или `incrementAll(java.util.Map<T, java.lang.Long> offsets, java.lang.Long initialValue)` — первый метод извлекает, второй увеличивает, а третий сохраняет неотрицательное значение типа `Byte`, `Short`, `Integer` или `Long`;

- ◆ `clearAll()` — очищает кэш;
- ◆ `getStatistics()` — возвращает статистику кэша в виде объекта `Stats` с методами:
  - `getBytesReturnedForHits()` — размер запросов `get()` и `contains()`;
  - `getHitCount()` — количество успешных запросов `get()` и `contains()`;
  - `getItemCount()` — количество пар в кэше;
  - `getMaxTimeWithoutAccess()` — прошедшее время с момента последнего запроса;
  - `getMissCount()` — количество неудачных запросов `get()` и `contains()`;
  - `getTotalItemBytes()` — размер кэша.



## ГЛАВА 7



# Поддержка сессий и HTTPS

## Поддержка протокола HTTP/SSL

Платформа App Engine автоматически поддерживает включение соединения клиента с приложением по протоколу HTTPS для поддоменов домена **appspot.com**. Для этого достаточно лишь указать в дескрипторе развертывания приложения `web.xml`, какие относительные URL-адреса приложения должны быть защищены HTTPS-протоколом:

```
<?xml version="1.0" encoding="utf-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
<servlet>
  <servlet-name>GAEApplication</servlet-name>
  <servlet-class>com.example.gae.GAEApplicationServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>GAEApplication</servlet-name>
  <url-pattern>/gaeapplication</url-pattern>
</servlet-mapping>
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/gaeapplication</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
</web-app>
```

В данном случае соединение со страницей приветствия приложения происходит по обычному HTTP-протоколу, а вызов сервлета приложения производится с использованием HTTPS-протокола.

При локальном запуске GAE-приложение в среде Eclipse включение HTTPS-протокола игнорируется, поэтому, для того чтобы увидеть HTTPS-соединение, необходимо загрузить GAE-приложение в App Engine (рис. 7.1).



Рис. 7.1. Вызов сервлета GAE-приложения по HTTPS-протоколу

Включение протокола HTTPS для конкретных URL-адресов приложения производится в дескрипторе `web.xml`, однако в конфигурационном файле `appengine-web.xml` также существует настройка, дублирующая включение или выключение использования HTTPS-протокола:

```
<ssl-enabled>false</ssl-enabled>
```

При использовании данной опции со значением `false` в дескрипторе `web.xml` тег `<transport-guarantee>` должен иметь значение `NONE` или ограничения по протоколу не должно быть вовсе.

При использовании данной опции со значением `true` в дескрипторе `web.xml` тег `<transport-guarantee>` может иметь значение `INTEGRAL` или `CONFIDENTIAL`.

## Использование сессий и cookie

Протокол HTTP является протоколом, не сохраняющим состояния. Для решения проблемы запоминания состояния при перемещении по Web-страницам приложения применяют сессию. GAE-платформа поддерживает сессии, используя реализацию программного интерфейса `HttpSession API Servlet-контейнера`.

По умолчанию поддержка сессий в App Engine отключена. Для того чтобы включить сессии, необходимо добавить в конфигурационный файл `appengine-web.xml` приложения настройку:

```
<sessions-enabled>true</sessions-enabled>
```

Для создания объекта `javax.servlet.http.HttpSession` сессии в коде сервлета приложения нужно вызвать метод `getSession()` интерфейса `javax.servlet.http.HttpServletRequest`:

```
HttpSession session = req.getSession(true);
```

Наличие аргумента `true` обеспечивает создание новой сессии, если она не существует.

При вызове данного метода в Datastore-хранилище создается сущность `_ah_SESSION` со свойствами `_expires` типа `int` — UNIX-время окончания действия сессии и `_values`

типа `blob` — хэш-данные сессии. Кроме того, в Web-браузере клиента записываются `cookies JSESSIONID` — идентификатор сессии. Если в Web-браузере отключить запись `cookies`, тогда при каждом обращении к сервлету, содержащему метод `getSession(true)`, будет создаваться новый объект `HttpSession` и новая сущность `_ah_SESSION`. При наличии `cookies JSESSIONID` время действия сессии будет определяться временем жизни `cookies`, которое настраивается в консоли администрирования в разделе **Administration | Application Settings | Cookie Expiration**.

Настройка `<async-session-persistence enabled="true"/>` конфигурационного файла `appengine-web.xml` приложения обеспечивает асинхронную запись данных сессии в Datastore-хранилище и тем самым уменьшает время ожидания ответа при запросе к приложению.

Методы `setAttribute(java.lang.String name, java.lang.Object value)` и `getAttribute(java.lang.String name)` интерфейса `HttpSession` позволяют сохранять и извлекать из объекта сессии различные значения:

```
String ip=req.getRemoteAddr();
session.setAttribute("ip", ip);
resp.getWriter().println("ip: "+session.getAttribute("ip"));
```

При этом необходимо учитывать, что сохраняться будут только объекты, реализующие интерфейс `java.io.Serializable`.

Сохраняться значения в объекте сессии будут с использованием `cookies`, поэтому если удалить `cookies` из Web-браузера клиента, удалятся и все сохраненные значения.

Метод `setMaxInactiveInterval(int interval)` интерфейса `HttpSession` никакого влияния на GAE-сессию оказывать не будет, а метод `invalidate()` интерфейса `HttpSession` будет уничтожать сущность `_ah_SESSION` Datastore-хранилища, но не `cookies JSESSIONID` Web-браузера.

Так как создание сессии App Engine связано с потреблением ресурсов Datastore-хранилища и все равно основано на `cookies` Web-браузера, имеет смысл напрямую записывать и извлекать значения из `cookies` в коде сервлета приложения:

```
// Запись cookies:
String ip=req.getRemoteAddr();
Cookie user_ip=new Cookie("user_ip",ip);
resp.addCookie(user_ip);
// Чтение cookies:
Cookie[] cookies=req.getCookies();
for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getName().equals("user_ip"))
        resp.getWriter().println("ip: "+cookie.getValue()); } }
```

## ГЛАВА 8



# Сервисы сообщений Mail, XMPP и Channel

Платформа App Engine позволяет отправлять и получать сообщения в приложении с помощью GAE-сервисов Mail и XMPP. Служба Mail обеспечивает отправку и получение сообщений электронной почты, а служба XMPP дает возможность организовать для приложения чат общения с пользователем.

## Служба Mail

### Отправка сообщений электронной почты

GAE-платформа обеспечивает отправку сообщений по электронной почте из приложения с помощью программного интерфейса JavaMail API.

#### **ПРИМЕЧАНИЕ**

Набор GAE SDK изначально уже содержит библиотеки `javax.mail` программного интерфейса JavaMail API, необходимые для разработки GAE-приложения, отправляющего и получающего сообщения по электронной почте.

Отправка электронного сообщения в коде приложения начинается с создания объекта `javax.mail.Session`, который обеспечивает связь с поставщиком протокола и связан с объектом `java.util.Properties`, позволяющим хранить информацию о почтовом сервере, логине, пароле и др. Объект почтовой сессии `Session` может быть создан с помощью статического метода `getDefaultInstance(Properties props, Authenticator authenticator)`, возвращающего `Session`-объект по умолчанию для среды выполнения:

```
Properties props = new Properties();  
Session session = Session.getDefaultInstance(props, null);
```

#### **ПРИМЕЧАНИЕ**

GAE-платформа не позволяет конфигурировать объект `Session` для отправки сообщений через другие почтовые службы, помимо GAE-службы Mail.

После создания объекта сессии можно создать объект сообщения `javax.mail.Message`. Класс `Message` является абстрактным классом, поэтому для создания `Message`-объекта необходимо воспользоваться его подклассом `javax.mail.internet.MimeMessage`:

```
MimeMessage message = new MimeMessage(session);
```

Далее определяются содержимое сообщения, его отправитель и получатель:

```
message.setContent("...", "text/plain");
message.setSubject("...");
Address addressFrom = new InternetAddress("nicknameFrom@gmail.com", "NameFrom");
message.setFrom(addressFrom);
Address addressTo = new InternetAddress("nicknameTo@gmail.com", "NameTo");
message.addRecipient(type, addressTo);
```

Здесь type — тип получателя:

- ◆ Message.RecipientType.TO — получатель;
- ◆ Message.RecipientType.CC — копия;
- ◆ Message.RecipientType.BCC — скрытая копия.

И в заключение сообщение отправляется статическим методом `send(Message msg)` класса `javax.mail.Transport`:

```
Transport.send(message);
```

Для отправки сообщения с вложением необходимо использовать классы `Multipart`, `MimeMultipart`, `BodyPart` и `MimeBodyPart` для определения содержимого сообщения:

```
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(addressFrom));
message.addRecipient(Message.RecipientType.TO, new InternetAddress(addressTo));
message.setSubject("...");
// Основная часть:
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("...");
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
// Вложение:
messageBodyPart = new MimeBodyPart();
javax.activation.DataSource source = new javax.activation.FileDataSource(filename);
messageBodyPart.setDataHandler(new javax.activation.DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);
// Добавление частей в сообщение:
message.setContent(multipart);
// Отправка сообщения:
Transport.send(message);
```

## Получение сообщений электронной почты

Для получения сообщений электронной почты GAE-приложением в первую очередь необходимо включить работу GAE-службы Mail по доставке сообщений. Для этого нужно добавить в конфигурационный файл приложения настройку:

```
<inbound-services>
  <service>mail</service>
</inbound-services>
```

Теперь клиент приложения может написать сообщение электронной почты по адресу:

anything@app-id.appspotmail.com

Задача GAE-службы Mail состоит в том, чтобы направить входящее сообщение приложению в виде HTTP POST-запроса по адресу:

/\_ah/mail/[address]

где [address] — полный адрес электронной почты приложения в виде **anything@app-id.appspotmail.com**. Поэтому, для того чтобы приложение смогло обработать входящее сообщение, в него требуется добавить сервлет-обработчик, связанный с URL-адресом HTTP-запроса /\_ah/mail/[address]:

```
<servlet>
  <servlet-name>MailReceiverServlet</servlet-name>
  <servlet-class>com.application.MailReceiverServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MailReceiverServlet</servlet-name>
  <url-pattern>/_ah/mail/*</url-pattern>
</servlet-mapping>
```

В коде сервлета-обработчика входящих сообщений электронной почты в качестве первого шага создаются объекты `javax.mail.Session` и `javax.mail.internet.MimeMessage`:

```
Properties props = new Properties();
Session session = Session.getDefaultInstance(props, null);
MimeMessage message = new MimeMessage(session, req.getInputStream());
```

Здесь объект `MimeMessage` создается путем чтения и разбора данных из входящего MIME-потока.

Далее из объекта `MimeMessage` извлекается содержимое сообщения:

```
Object content = message.getContent();
if (content instanceof String) {
  System.out.println("From : " + message.getFrom()[0]);
  System.out.println("Subject : " + message.getSubject());
  System.out.print("Message : ");
  InputStream stream = message.getInputStream();
  while (stream.available() != 0) {
    System.out.print((char) stream.read());
  }
} else if (content instanceof Multipart) {
for (int i = 0; i < ((Multipart) content).getCount(); i++) {
  BodyPart bodyPart = ((Multipart) content).getBodyPart(i);
  InputStream stream = bodyPart.getInputStream();
  BufferedReader br = new BufferedReader(new InputStreamReader(stream));
  while (br.ready()) {
    System.out.println(br.readLine());
  }
}
```

Существующие ограничения использования GAE-службы Mail — это размер исходящих сообщений электронной почты не должен превышать 1 Мбайт, а размер входящих сообщений — 10 Мбайт.

## Пример использования службы Mail

В качестве примера рассмотрим приложение, имеющее страницу, на которой пользователь может послать сообщение электронной почты другому пользователю. Приложение также имеет обработчик входящих сообщений приложению.

Страница приветствия приложения будет иметь следующий код:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ page import="com.google.appengine.api.users.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
if (user != null) {
%>
<div style="text-align:left">
<%
response.getWriter().println("Hello " + user.getNickname());
%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
</jsp:element>
</div><br>
<%
}else{
    response.sendRedirect(userService.createLoginURL("/"));
}
%>
<label>Send email:</label><br>
<form action="/gaeapplication" name="send_msg" method="post" >
<label>Email address:</label><br>
<input type="text" name="addressTo" size="30" required><br>
<label>Email subject:</label><br>
```

```

<input type="text" name="msg_subject" size="30" ><br>
<label>Email text:</label><br>
<textarea rows="5" cols="45" name="msg_text"></textarea>
<p><input type="submit" value="Submit">
<input type="reset" value="Reset"></p>
</form>
</body>
</html>

```

В коде JSP-страницы приветствия приложения производится аутентификация пользователя с помощью Google-аккаунта. Также страница содержит форму для заполнения содержимого сообщения электронной почты, которая обрабатывается сервлетом приложения, связанным с URL-адресом `/gaeapplication`:

```

<servlet>
  <servlet-name>GAEApplication</servlet-name>
  <servlet-class>com.example.gae.GAEApplicationServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>GAEApplication</servlet-name>
  <url-pattern>/gaeapplication</url-pattern>
</servlet-mapping>
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

Аутентификация пользователя с помощью Google-аккаунта производится для того, чтобы отправителем сообщения был адрес электронной почты Google-аккаунта:

```

package com.example.gae;
import java.io.IOException;
import java.util.Properties;
import javax.servlet.http.*;
import com.google.appengine.api.users.*;
import javax.mail.*;
import javax.mail.internet.*;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
  public void doPost(HttpServletRequest req, HttpServletResponse resp)
  throws IOException {
    UserService userService = UserServiceFactory.getUserService();
    User user = userService.getCurrentUser();
    Properties props = new Properties();
    Session session = Session.getDefaultInstance(props, null);
    MimeMessage message = new MimeMessage(session);
    try {
      message.setContent(req.getParameter("msg_text"), "text/plain");
      message.setSubject(req.getParameter("msg_subject"));
    }
  }
}

```



```

        Address addressFrom = new InternetAddress(user.getEmail());
        message.setFrom(addressFrom);
        Address addressTo = new InternetAddress(req.getParameter("addressTo"));
        message.addRecipient(Message.RecipientType.TO, addressTo);
        Transport.send(message);
    } catch (MessagingException e) {
        e.printStackTrace();
    }
    resp.sendRedirect("/");
}}

```

**Сервлет-обработчик входящих сообщений будет иметь следующий код:**

```

package com.example.gae;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Properties;
import javax.mail.BodyPart;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.Session;
import javax.mail.internet.MimeMessage;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@SuppressWarnings("serial")
public class MailReceiverServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        Properties props = new Properties();
        Session session = Session.getDefaultInstance(props, null);
        try {
            MimeMessage message = new MimeMessage(session, req.getInputStream());
            Object content = message.getContent();
            if (content instanceof String) {
                System.out.println("From : " + message.getFrom()[0]);
                System.out.println("Subject : " + message.getSubject());
                System.out.print("Message : ");
                InputStream stream = message.getInputStream();
                while (stream.available() != 0) {
                    System.out.print((char) stream.read());
                }
            } else if (content instanceof Multipart) {
                for (int i = 0; i < ((Multipart) content).getCount(); i++) {
                    BodyPart bodyPart = ((Multipart) content).getBodyPart(i);
                    InputStream stream = bodyPart.getInputStream();
                }
            }
        }
    }
}

```

```

        BufferedReader br = new BufferedReader(new InputStreamReader(stream));
        while (br.ready()) {
            System.out.println(br.readLine());
        }
    }
} catch (MessagingException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}}}
```

logging.properties:

```
.level = ALL
```

appengine-web.xml:

```

<inbound-services>
  <service>mail</service>
</inbound-services>
```

web.xml:

```

<servlet>
<servlet-name>MailReceiverServlet</servlet-name>
<servlet-class>com.example.gae.MailReceiverServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>MailReceiverServlet</servlet-name>
<url-pattern>/_ah/mail/*</url-pattern>
</servlet-mapping>
```

Теперь приложение может получать сообщения электронной почты, отправленные с любого почтового сервиса, и отображать их в журнале приложения, доступном в разделе **Main | Logs** консоли администрирования.

## Служба XMPP

Служба XMPP платформы App Engine обеспечивает для GAE-приложения, использующего программный интерфейс XMPP API, получение и отправку мгновенных сообщений от клиента и клиенту, находящемуся в системе XMPP-сервера, такой как Google Talk, Jabber и др. Например, пользователь, который запустил и вошел в систему клиентского приложения Google Talk (<http://www.google.com/talk/intl/ru/>) или Jabber.Ru (<http://www.miranda-im.org/download/>), может отправить сообщение GAE-приложению на JID-адрес **app\_id@appspot.com** или **anything@app\_id.appspotchat.com**, который присваивается приложению GAE-сервисом XMPP, и получить ответное сообщение.

Платформа App Engine не является сама по себе XMPP-сервером, она лишь обеспечивает своей XMPP-службой перенаправление входящего мгновенного сообщения приложению в виде HTTP POST-запроса и отправку мгновенного сообщения от приложения XMPP-серверу.

Например, с помощью программного интерфейса XMPP API можно реализовать GAE-приложение, которое будет представлять собой XMPP-бот переводчика, получающего от клиента сообщение на одном языке и в ответ отправляющего сообщение с переводом на другой язык, или GAE-приложение, служащее мостом между обычным Web-клиентом и XMPP-клиентом.

## Отправка мгновенных сообщений

Программный интерфейс XMPP API обеспечивает для GAE-приложения четыре вида исходящих действий — запрос статуса клиента, отправка приглашения в чат, отправка сообщения и отправка статуса приложения. Для выполнения данных действий необходимо создать объект `com.google.appengine.api.xmpp.XMPPService`, отвечающий за взаимодействие с GAE-службой XMPP. Объект `XMPPService` создается статическим методом `getXMPPService()` класса-фабрики `com.google.appengine.api.xmpp.XMPPServiceFactory`:

```
XMPPService xmppService = XMPPServiceFactory.getXMPPService();
```

После создания объекта `XMPPService` отправка приглашения осуществляется с помощью метода `sendInvitation()`, проверка статуса — с помощью метода `getPresence()`, отправка сообщения — методом `sendMessage()`, а отправка статуса приложения — методом `sendPresence()`:

```
// Отправка приглашения:
JID jid = new JID("someone@jabber.ru");
xmppService.sendInvitation(jid);
// Проверка доступности и статуса:
Presence presence = xmppService.getPresence(jid);
Boolean flag = presence.isAvailable();
String status = presence.getStatus();
// Отправка сообщения:
Message msg = new MessageBuilder()
    .withRecipientJids(jid)
    .withBody("")
    .build();
xmppService.sendMessage(msg);
// Отправка статуса приложения:
xmppService.sendPresence(jid, PresenceType.AVAILABLE, PresenceShow.NONE, "My status");
```

## Получение мгновенных сообщений

GAE-приложение может быть доступно для получения следующих типов входящих сообщений:

- ◆ входящие мгновенные сообщения. Требуют включения в файл `appengine-web.xml` настройки:

```
<inbound-services>
  <service> xmpp_message </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/message/chat/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения о том, что пользователь хочет подписаться на наличие приложения. Требуют включения в файл appengine-web.xml настройки:

```
<inbound-services>
  <service> xmpp_subscribe </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/subscription/subscribe/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения о том, что пользователь авторизовал приложение. Требуют включения в файл appengine-web.xml настройки:

```
<inbound-services>
  <service> xmpp_subscribe </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/subscription/subscribed/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения о том, что пользователь аннулировал подписку на наличие приложения. Требуют включения в файл appengine-web.xml настройки:

```
<inbound-services>
  <service> xmpp_subscribe </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/subscription/unsubscribe/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения о том, что пользователь отверг запрос на авторизацию приложения или отменил ранее предоставленные подписки. Требуют включения в файл appengine-web.xml настройки:

```
<inbound-services>
  <service> xmpp_subscribe </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/subscription/unsubscribed/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения о том, что пользователь в он-лайне. Требуют включения в файл `appengine-web.xml` настройки:

```
<inbound-services>
  <service> xmpp_presence </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/presence/available/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения о том, что пользователь недоступен. Требуют включения в файл `appengine-web.xml` настройки:

```
<inbound-services>
  <service> xmpp_presence </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/presence/unavailable/ </url-pattern>
</servlet-mapping>
```

- ◆ ответы на запросы текущего статуса пользователя. Требуют включения в файл `appengine-web.xml` настройки:

```
<inbound-services>
  <service> xmpp_presence </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/presence/probe/ </url-pattern>
</servlet-mapping>
```

- ◆ сообщения об ошибках. Требуют включения в файл `appengine-web.xml` настройки:

```
<inbound-services>
  <service> xmpp_error </service>
</inbound-services>
```

и связывания сервлета-обработчика с адресом:

```
<servlet-mapping>
  <servlet-name>ApplicationServlet</servlet-name>
  <url-pattern> /_ah/xmpp/error/ </url-pattern>
</servlet-mapping>
```

Для вышеуказанных видов сообщений программный интерфейс XMPP API предоставляет возможность разбора входящих сообщений в коде сервлета-обработчика:

```
XMPPService xmpp = XMPPServiceFactory.getXMPPService();
// Разбор входящего сообщения сервиса xmpp_message:
Message message = xmpp.parseMessage(req);
JID fromJid = message.getFromJid();
String body = message.getBody();
// Разбор входящего сообщения сервиса xmpp_presence:
Presence presence = xmpp.parsePresence(req);
String fromJid = presence.getFromJid().getId().split("/")[0];
String status = presence.getStatus();
// Разбор входящего сообщения сервиса xmpp_subscribe:
Subscription sub = xmpp.parseSubscription(req);
String fromJid = sub.getFromJid().getId().split("/")[0];
String type = sub.getSubscriptionType().toString();
```

Существующие ограничения использования GAE-службы XMPP — это размер входящих и исходящих сообщений не должен превышать 100 килобайт.

## Пример использования службы XMPP

В качестве примера рассмотрим простое приложение, служащее мостом между обычным Web-клиентом и XMPP-клиентом.

На странице приветствия приложения размещена форма ввода сообщения для определенного JID-адреса и фрейм, отображающий ответные сообщения:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <title>XMPP Application</title>
  </head>
  <body>
    <IFRAME SRC="xmpp.jsp" WIDTH="200" HEIGHT="200" NAME="iframe" SCROLLING="auto"
    FRAMEBORDER=1></IFRAME>
    <form action="/send" method="post" >
      Jid:<br>
      <input type="text" name="jid" size="30"/><br><br>
      Text:<br>
      <textarea rows="5" cols="45" name="text"></textarea>
```

```

        <p><input type="submit" value="Submit">
        <input type="reset" value="Reset"></p>
    </form>
</body>
</html>

```

**web.xml:**

```

<welcome-file-list>
<welcome-file>index.html</welcome-file>
</welcome-file-list>

```

**В коде сервлета, обрабатывающего форму, из запроса извлекается JID-адрес и текст сообщения, на основе которых формируется и отправляется XMPP-сообщение:**

```

package com.example.gae;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.xmpp.JID;
import com.google.appengine.api.xmpp.Message;
import com.google.appengine.api.xmpp.MessageBuilder;
import com.google.appengine.api.xmpp.MessageType;
import com.google.appengine.api.xmpp.XMPPService;
import com.google.appengine.api.xmpp.XMPPServiceFactory;
@SuppressWarnings("serial")
public class XMPPSend extends HttpServlet {
    private static final XMPPService xmppService = XMPPServiceFactory.getXMPPService();
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        String jidString=req.getParameter("jid");
        JID jid = new JID(jidString);
        String text=req.getParameter("text");
        Message message = new MessageBuilder()
            .withRecipientJids(jid)
            .withMessageType(MessageType.NORMAL)
            .withBody(text)
            .build();
        xmppService.sendMessage(message);
        resp.sendRedirect("/");}}

```

**web.xml:**

```

<servlet>
    <servlet-name>XMPPSend</servlet-name>
    <servlet-class>com.example.gae.XMPPSend</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>XMPPSend</servlet-name>

```

```
<url-pattern>/send</url-pattern>
</servlet-mapping>
```

В коде сервлета, обрабатывающего входящее сообщение, производится разбор ответного сообщения с сохранением его содержимого в Datastore-хранилище:

```
package com.example.gae;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.datastore.*;
import com.google.appengine.api.xmpp.Message;
import com.google.appengine.api.xmpp.XMPPService;
import com.google.appengine.api.xmpp.XMPPServiceFactory;
@SuppressWarnings("serial")
public class XMPPReceive extends HttpServlet {
    private static final XMPPService xmppService = XMPPServiceFactory.getXMPPService();
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        Message message = xmppService.parseMessage(req);
        String jid = message.getFromJid().getId().split("/")[0];
        String body = message.getBody();
        DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
        Query qUsersMessage = new Query("UsersMessage");
        PreparedQuery pq = datastore.prepare(qUsersMessage);
        java.util.List<Entity> list = pq.asList(FetchOptions.Builder.withDefaults());
        if(list.isEmpty()){
            Entity UsersMessage = new Entity("UsersMessage");
            UsersMessage.setProperty("jid", jid);
            UsersMessage.setProperty("text", body);
            datastore.put(UsersMessage);
        }else{
            for (Entity result : list) {
                result.setProperty("jid", jid);
                result.setProperty("text", body);
                datastore.put(result);
            }
        }
    }
}
```

#### web.xml:

```
<servlet>
    <servlet-name>XMPPReceive</servlet-name>
    <servlet-class>com.example.gae.XMPPReceive</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>XMPPReceive</servlet-name>
    <url-pattern>/_ah/xmpp/message/chat</url-pattern>
</servlet-mapping>
```



appengine-web.xml:

```
<inbound-services>
  <service>xmpp_message</service>
</inbound-services>
```

В коде JSP-страницы, включаемой в страницу приветствия приложения в виде фрейма, из Datastore-хранилища извлекаются и отображаются ответные сообщения:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ page import="com.google.appengine.api.users.*" %>
<%@ page import="com.google.appengine.api.datastore.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<script type="text/javascript" language="javascript">
    setInterval ("location.reload(true);", 1000);
</script>
<%
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Query qUsersMessage = new Query("UsersMessage");
    PreparedQuery pq = datastore.prepare(qUsersMessage);
    java.util.List<Entity> list = pq.asList(FetchOptions.Builder.withDefaults());
    for (Entity result : list) {
        String jid = (String) result.getProperty("jid");
        String text = (String) result.getProperty("text");
    }
%>
From:<br>
<%=jid %><br>
Text:<br>
<%=text %><br>
<%
    }
%>
</body>
</html>
```

## Служба Channel

При разработке Web-приложения может появиться задача отправки данных клиентскому браузеру со стороны Web-сервера без клиентского запроса. Такое взаимодействие между сервером и клиентом основывается на постоянном или длитель-

ном HTTP-соединении, поэтому приложения, использующие данную технологию, часто называют *приложениями реального времени*.

Технологии, обеспечивающие с помощью долгого HTTP-соединения для Web-сервера отправку данных Web-браузеру без прямого со стороны клиента запроса, объединены под названием Comet-технологии.

Реализации модели Comet относятся к двум основным категориям — технологии потока (streaming) и технологии длинного опроса (long polling).

Модель длинного опроса (long polling) представляет собой схему, в которой браузер клиента подключается к серверу и ожидает, пока сервер не пошлет ему данные. После получения сообщения от сервера клиент снова подключается к серверу. В случае реализации потока (streaming), после получения данных от сервера не происходит переподключения, а соединение между браузером и сервером остается открытым.

Выбор между двумя моделями зависит от того, как часто клиент должен получать данные от сервера. Для торгового клиента Forex правильным решением будет реализация модели потока, а для организации чата возможно применение длинных опросов.

Альтернативой Comet-технологиям являются технологии HTML5 EventSource и WebSocket, Java Applets и Adobe Flash.

Для использования модели Comet существуют готовые коммерческие и открытые решения (<http://cometdaily.com/maturity.html>).

Платформа App Engine предлагает свое решение для организации длинных опросов — службу Channel, которая позволяет создать постоянный канал сообщений между Web-страницей на стороне клиента и приложением на стороне GAE-платформы.

Для создания клиента GAE-приложения, обслуживающего длинные опросы, платформа App Engine предоставляет JavaScript-библиотеку, находящуюся по относительному адресу `/_ah/channel/jsapi`, для использования которой в HTML-код страницы, отображаемой клиенту, необходимо вставить строку:

```
<script type="text/javascript" src="/_ah/channel/jsapi"></script>
```

Для открытия канала сообщений между Web-страницей на стороне клиента и приложением на стороне GAE-платформы канал необходимо идентифицировать. Поэтому в качестве первого шага Channel-клиент должен отправить запрос GAE-приложению для получения идентификатора канала. После получения идентификатора канала на стороне клиента нужно создать JavaScript-объект `goog.appengine.Channel(token)`, где `token` — идентификатор (маркер) канала сообщений:

```
var channel = new goog.appengine.Channel('[token]');
```

и открыть канал методом `open()` объекта `Channel`, возвращающего JavaScript-объект `goog.appengine.Socket`:

```
var socket = channel.open();
```

свойства которого необходимо определить:

- ◆ `socket.onopen` — указывает JavaScript-функцию, вызываемую, когда сокет готов получать сообщения;
- ◆ `socket.onmessage` — указывает JavaScript-функцию, вызываемую при получении сообщения. Данная функция имеет единственный параметр — объект `message`, поле `data` которого возвращает строку сообщения;
- ◆ `socket.onerror` — указывает JavaScript-функцию, вызываемую при возникновении ошибки. Данная функция имеет единственный параметр — объект `error`, содержащий поля `description` и `code`;
- ◆ `socket.onclose` — указывает JavaScript-функцию, вызываемую при закрытии сокета, который закрывается методом `close()` объекта `Socket`.

Время жизни маркера канала составляет 2 часа, после этого вызываются методы `socket.onerror` и `socket.onclose` сокета, в которых нужно обеспечить запрос на получение нового маркера канала.

На стороне платформы App Engine в коде сервлета GAE-приложения создается объект `com.google.appengine.api.channel.ChannelService`, обеспечивающий использование службы Channel:

```
ChannelService channelService = ChannelServiceFactory.getChannelService();
```

Метод `createChannel(java.lang.String clientId)` интерфейса `ChannelService` обеспечивает создание идентификатора (маркера) канала, который необходимо отправить клиенту. Маркер канала создается на основе идентификатора клиента.

Метод `sendMessage(ChannelMessage message)` интерфейса `ChannelService` позволяет послать сообщение в канал. Объект `com.google.appengine.api.channel.ChannelMessage` представляет сообщение канала и создается с помощью конструктора:

```
public ChannelMessage(java.lang.String clientId, java.lang.String message)
```

где `clientId` — идентификатор клиента, на основе которого был создан маркер канала, а `message` — строка сообщения размером не более 32 Кбайт в кодировке UTF-8.

Соответственно, если клиент знает идентификатор другого клиента, он может послать сообщение GAE-приложению, которое перешлет его в канал другого клиента.

GAE-приложение может также получать уведомления от службы Channel, когда клиент присоединяется к каналу или отсоединяется от него. Для этого в конфигурационный файл `appengine-web.xml` приложения должна быть включена настройка:

```
<inbound-services>
  <service>channel_presence</service>
</inbound-services>
```

Уведомления будут приходить в виде POST-запросов по адресам `/_ah/channel/connected/` и `/_ah/channel/disconnected/`.

В коде сервлетов, обрабатывающих данные уведомления, может быть использован метод `parsePresence(HttpServletRequest request)` интерфейса `ChannelService`, возвращающий объект `com.google.appengine.api.channel.ChannelPresence`. При этом метод

`clientId()` класса `ChannelPresence` позволит выяснить, от какого клиента пришло уведомление.

В качестве примера рассмотрим простое приложение, представляющее собой авто-бот, который отвечает посылаемым сообщением (рис. 8.1).

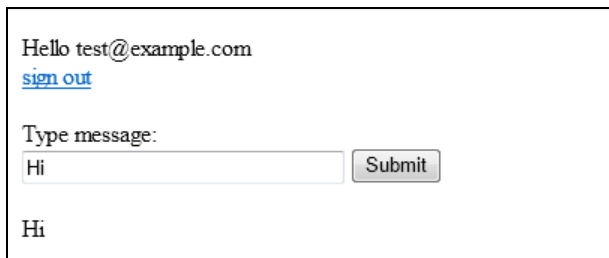


Рис. 8.1. Страница приложения, отвечающего посылаемым сообщением

Страница приветствия такого приложения содержит код аутентификации пользователя с помощью Google-аккаунта, поле и кнопку отправки сообщения, а также код открытия и использования Channel-канала:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ page import="com.google.appengine.api.users.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<script type="text/javascript" src="/scripts/jquery-1.7.2.min.js" ></script>
<script type="text/javascript" src="/_ah/channel/jsapi"></script>
<title>Insert title here</title>
</head>
<body>
// Аутентификация пользователя:
<%
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
if (user != null) {
%>
<div style="text-align:left">
<%
response.getWriter().println("Hello " + user.getNickname());
%>
<jsp:element name="a">
<jsp:attribute name="href"><%=userService.createLogoutURL("/")%></jsp:attribute>
<jsp:body>sign out
</jsp:body>
```

```

</jsp:element>
</div><br>
<%
}else{
response.sendRedirect(userService.createLoginURL("/"));
}
%>
// Ввод сообщения:
Type message:
<br>
<input id="input" type="text" size="30" name="message"/>
<input id="submit" type="button" value="Submit"/>
<br>
// Блоки получения сообщения:
<div id="opened"></div>
<div id="message"></div>
<div id="error"></div>
<div id="close"></div>
<script type="text/javascript">
$(document).ready(function() {
// Получение маркера канала:
var token='';
$.ajax({
    type: 'GET',
    url: '/gaeapplication',
    async:false,
    success: function(data) {
        result = data.getElementsByTagName('result')[0].firstChild.data;
        if(result!=""){
            token=result;
        }
    }
});
// Открытие канала:
var channel = new goog.appengine.Channel(token);
var socket = channel.open();
socket.onopen = onOpened;
socket.onmessage = onMessage;
socket.onerror = onError;
socket.onclose = onClose;
// Отправка сообщения:
$('#submit').click(function() {
    var message=$('#input').val();
    $.post("/gaeapplication", { message: message });
});
});
// Функции обработки входящих сообщений:
function onOpened(){ }
```

```

function onError(error){
    $('#error').append('<p> Code: '+error.code+'</p><p> Description: '+error.description+'</p>');
}
function onClose(){ }
function onMessage(message){
    $('#message').append('<p>'+message.data+'</p>');
}
</script>
</body>
</html>

```

**Сервлет GAE-приложения, выдающий маркер канала и посылающий ответное сообщение, будет иметь следующий код:**

```

package com.example.gae;
import java.io.IOException;
import javax.servlet.http.*;
import com.google.appengine.api.channel.*;
import com.google.appengine.api.users.*;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    // Выдача маркера канала:
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        ChannelService channelService = ChannelServiceFactory.getChannelService();
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        String token = channelService.createChannel(user.getUserId());
        resp.setContentType("text/xml");
        resp.getWriter().println("<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>");
        resp.getWriter().println("<response>");
        resp.getWriter().println("<method>getURL</method>");
        resp.getWriter().println("<result>"+token+"</result>");
        resp.getWriter().println("</response>");
    }
    // Получение и отправка сообщения:
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        ChannelService channelService = ChannelServiceFactory.getChannelService();
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        String messageIn=req.getParameter("message");
        ChannelMessage messageOut=new ChannelMessage(user.getUserId(), messageIn);
        channelService.sendMessage(messageOut);
    }
}

```

## ГЛАВА 9



# Фильтры и обработка ошибок

## Фильтрация запросов и ответов

Servlet-контейнер GAE-платформы предоставляет возможность фильтрации входящих запросов и исходящих ответов приложения с помощью программного интерфейса Filter API, включающего в себя интерфейсы `Filter`, `FilterChain` и `FilterConfig` библиотеки `javax.servlet`.

Фильтр приложения создается аналогично сервлету приложения. Класс фильтра должен реализовывать интерфейс `javax.servlet.Filter` с определением его методов:

- ◆ `void init(FilterConfig filterConfig)` — вызывается Servlet-контейнером после создания экземпляра фильтра для его инициализации. Методу передается в качестве аргумента объект `FilterConfig`, позволяющий получить имя фильтра в дескрипторе `web.xml` с помощью метода `getFilterName()`, значение параметра инициализации дескриптора `web.xml` с помощью метода `getInitParameter(java.lang.String name)`, контекст сервлета методом `getServletContext()`;
- ◆ `void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)` — вызывается Servlet-контейнером при каждом запросе/ответе, связанном с фильтром. Аргумент метода — объект `FilterChain` обеспечивает передачу управления следующему фильтру с помощью вызова метода `chain.doFilter(request, response)`. Вызов фильтров в цепочке производится в том же порядке, в каком фильтры определены в дескрипторе `web.xml`;
- ◆ `void destroy()` — вызывается Servlet-контейнером при уничтожении экземпляра фильтра для освобождения занятых им ресурсов.

В дескрипторе `web.xml` приложения фильтр объявляется с использованием тегов:

```
<filter>
  <filter-name>MyFilter</filter-name>
  <filter-class>myapplication.MyFilter</filter-class>
  <init-param>
    <param-name>param</param-name>
    <param-value>value</param-value>
```

```
</init-param>  
</filter>
```

С относительным адресом приложения фильтр связывается аналогично сервлету приложения:

```
<filter-mapping>  
  <filter-name>MyFilter</filter-name>  
  <url-pattern>/path</url-pattern>  
</filter-mapping>
```

Для GAE-платформы особенность состоит в том, что фильтр нельзя связать со статическими ресурсами приложения, т.к. они не обрабатываются Servlet-контейнером GAE-платформы.

Как правило, фильтры Servlet-контейнера используются для аутентификации-авторизации пользователя, журналирования и аудита приложения, конвертации, шифрования и сжатия данных, локализации приложения и др.

В качестве примера рассмотрим фильтр локализации приложения.

Для объявления фильтра добавим следующие теги в дескриптор web.xml приложения:

```
<filter>  
  <filter-name>FilterLocale</filter-name>  
  <filter-class>com.example.gae.FilterLocale</filter-class>  
</filter>  
<filter-mapping>  
  <filter-name>FilterLocale</filter-name>  
  <url-pattern>/</url-pattern>  
</filter-mapping>  
<welcome-file-list>  
  <welcome-file>index.jsp</welcome-file>  
</welcome-file-list>
```

Теперь фильтр локализации будет вызываться при обращении к JSP-странице приветствия приложения.

Класс `FilterLocale` фильтра будет иметь следующий код:

```
package com.example.gae;  
import java.io.IOException;  
import javax.servlet.*;  
import javax.servlet.http.*;  
public class FilterLocale implements Filter{  
  private FilterConfig filterConfig = null;  
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain  
    filterChain)  
    throws IOException, ServletException {  
    final HttpServletRequest httpRequest = (HttpServletRequest) request;  
    final HttpServletResponse httpResponse = (HttpServletResponse) response;
```



```
String country=httpRequest.getHeader("X-AppEngine-Country");
    if(country.equals("RU")){
        httpResponse.sendRedirect("/ru/index.jsp");
    }else{
        httpResponse.sendRedirect("/en/index.jsp");
    }
}
public void init(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
}
public void destroy() {
    this.filterConfig = null;
}
}}
```

В коде класса фильтра производится проверка GAE-заголовка `X-AppEngine-Country` HTTP-запроса, и если запрос происходит из России — он перенаправляется JSP-странице папки `ru` приложения, во всех остальных случаях запрос перенаправляется JSP-странице папки `en`.

## Обработка ошибок

При возникновении различного рода ошибок и исключительных ситуаций в работе приложения GAE-платформой производится отправка определенного HTTP-кода состояния и генерация страницы ошибки для клиента. С помощью настроек дескриптора `web.xml` и конфигурационного файла `appengine-web.xml` можно определить отображение своих страниц при возникновении ошибок и исключительных ситуаций.

Для дескриптора `web.xml` это теги:

```
<error-page>
    <error-code>[HTTP-код ошибки или Java-класс исключения]</error-code>
    <location>[URL-путь ресурса для отображения]</location>
</error-page>
```

Для файла `appengine-web.xml` это теги:

```
<static-error-handlers>
<handler file="[HTML-файл для отображения]" error-code="[код ошибки]"/>
. . .
</static-error-handlers>
```

Здесь код ошибки может быть:

- ◆ `over_quota` — превышение установленной квоты расходования ресурсов GAE-платформы;
- ◆ `dos_api_denial` — блокирование в случае угрозы DoS-атак;
- ◆ `timeout` — превышение порога ожидания ответа.

При этом в файле `appengine-web.xml` можно настроить отображение только статических ресурсов, в отличие от дескриптора `web.xml`, позволяющего определить обработчик ошибки.

В качестве примера рассмотрим приложение, которое имеет сервлет-обработчик, отправляющий сообщение электронной почты в случае возникновения ошибки.

Для настройки такого сервлета добавим в дескриптор web.xml теги:

```
<servlet>
  <servlet-name>SendErrorServlet</servlet-name>
  <servlet-class>com.example.gae.SendErrorServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SendErrorServlet</servlet-name>
  <url-pattern>/error</url-pattern>
</servlet-mapping>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/error</location>
</error-page>
```

Сервлет-обработчик SendErrorServlet будет иметь следующий код:

```
package com.example.gae;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.Properties;
import javax.servlet.http.*;
import com.google.appengine.api.users.*;
import javax.mail.*;
import javax.mail.internet.*;
@SuppressWarnings("serial")
public class SendErrorServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {
    UserService userService = UserServiceFactory.getUserService();
    User user = userService.getCurrentUser();
    Properties props = new Properties();
    Session session = Session.getDefaultInstance(props, null);
    MimeMessage message = new MimeMessage(session);
    try {
      Throwable t = (Throwable) req.getAttribute("javax.servlet.error.exception");
      StringWriter sw = new StringWriter();
      PrintWriter pw = new PrintWriter(sw, true);
      t.printStackTrace(pw);
      String text = sw.toString();
      message.setContent(text, "text/plain");
      message.setSubject(t.toString());
      Address addressFrom = new InternetAddress(user.getEmail());
      message.setFrom(addressFrom);
```

```
Address addressTo = new InternetAddress("admin_mail");
message.addRecipient(Message.RecipientType.TO, addressTo);
Transport.send(message);
} catch (MessagingException e) {
    e.printStackTrace();
}
resp.sendRedirect("/errorPages/exception.jsp");
}}
```

## ГЛАВА 10



# Разработка Backend-приложений

Как уже было сказано в *главе 1*, платформа App Engine может создавать два типа экземпляров приложения — экземпляр GAE-приложения может быть *внешним* (Frontend) или *фоновым* (Backend).

Платформа App Engine по умолчанию создает Frontend-экземпляры приложения, которые по умолчанию обрабатывают входящие запросы, и их работа ограничена установленной квотой. В частности, бесплатная квота для Frontend-экземпляров приложения составляет 28 часов.

В разделе **Administration | Application Settings | Frontend Instance Class** консоли администрирования для приложения можно установить тип создаваемых Frontend-экземпляров:

- ◆ F1 (по умолчанию) — ограничение по памяти 128 Мбайт, ограничение по частоте процессора 600 МГц, плата за час \$0,08;
- ◆ F2 — ограничение по памяти 256 Мбайт, ограничение по частоте процессора 1,2 ГГц, плата за час \$0,16;
- ◆ F4 — ограничение по памяти 512 Мбайт, ограничение по частоте процессора 2,4 ГГц, плата за час \$0,32.

Frontend-экземпляры приложения могут быть динамическими или резидентными.

Frontend-экземпляр приложения называется *динамическим*, если его жизненный цикл регулируется планировщиком GAE-платформы, или *резидентным*, если работает все время, при этом за его создание отвечает администратор приложения. Создать резидентные Frontend-экземпляры можно, только если приложение имеет платный хостинг и с помощью установки **min idle instances** раздела **Administration | Application Settings | Performance** консоли администрирования.

Для того чтобы GAE-платформа создала Backend-экземпляры приложения, их нужно определить в конфигурационном файле `backends.xml` папки WEB-INF приложения, который имеет следующий синтаксис:

```
<?xml version="1.0" encoding="utf-8"?>
<backends>
  <backend name="...">
```

```
<class>...</class>
<instances>...</instances>
<max-concurrent-requests>...</max-concurrent-requests>
<options>
  <fail-fast>true</fail-fast>
  <dynamic>true</dynamic>
  <public>true</public>
</options>
</backend>
. . .
</backends>
```

Атрибут `name` определяет имя Backend-сервера, тег `<class>` указывает тип Backend-сервера:

- ◆ B1 — ограничение по памяти 128 Мбайт, ограничение по частоте процессора 600 МГц, плата за час \$0,08;
- ◆ B2 (по умолчанию) — ограничение по памяти 256 Мбайт, ограничение по частоте процессора 1200 МГц, плата за час \$0,16;
- ◆ B4 — ограничение по памяти 512 Мбайт, ограничение по частоте процессора 2400 МГц, плата за час \$0,32;
- ◆ B8 — ограничение по памяти 1024 Мбайт, ограничение по частоте процессора 4800 МГц, плата за час \$0,64.

Бесплатная квота для Backend-экземпляров приложения составляет 9 часов.

Тег `<instances>` указывает количество от 1 до 20 создаваемых экземпляров Backend-сервера.

Тег `<max-concurrent-requests>` определяет максимальное количество запросов, которые каждый экземпляр Backend-сервера может обрабатывать одновременно.

Тег `<dynamic>` указывает, является ли Backend-сервер динамическим или резидентным (по умолчанию). В отличие от динамических Frontend-экземпляров Backend-экземпляры не создаются автоматически планировщиком GAE-платформы, он может лишь запускать и останавливать динамический Backend-сервер, с которым связано определенное число Backend-экземпляров.

Тег `<fail-fast>` отключает очередь запросов, так что занятый Backend-экземпляр в ответ на запрос возвратит HTTP-код 503.

Тег `<public>` определяет, сможет ли Backend-сервер вызываться внешним HTTP-запросом или только приложением (по умолчанию).

Backend-экземпляры, по сравнению с Frontend-экземплярами, не ограничены 30-секундным периодом обработки запроса и могут использовать большую память (до 1 Гбайт) и больше загружать процессор (до 4,8 ГГц). Соответственно работа Frontend- и Backend-экземпляров приложения отдельно учитывается в использовании установленной квоты. Backend-экземпляры приложения связаны со специальным Backend-сервером, создаваемом для приложения GAE-платформой. Приложе-

ние может иметь несколько Backend-серверов, с каждым из которых может быть связано несколько Backend-экземпляров. Свойства Backend-серверов приложения определяются в его конфигурационном `backends`-файле, кроме того, Backend-серверы назначаются резидентными (по умолчанию) или динамическими с помощью конфигурационного `backends`-файла приложения.

Резидентный Backend-сервер запускается и останавливается с помощью консоли администрирования (рис. 10.1), а динамический Backend-сервер — GAE-платформой при получении запроса и после окончания работы через несколько минут.

Your changes have been saved.


Backend	Deployed	Class (Memory, CPU)	Instances	Options	Start/Stop	Delete
<a href="#">backend</a>  <a href="#">instances</a>   14.91 MBytes   java   api_version: 1.0	0:07:52 ago by	B1 (126MB, 600Mhz)	1	dynamic public	<input type="button" value="Stop"/>	<input type="button" value="Delete"/>

Рис. 10.1. Управление резидентным Backend-сервером в консоли администрирования

Backend-экземпляры могут запускать те же самые ресурсы приложения, что и Frontend-экземпляры — для этого HTTP-запрос должен иметь вид:

`http://[instance].[backend_name].[app_id].appspot.com/resource`

Если не указан номер Backend-экземпляра, вызывается первый доступный экземпляр.

Программным способом URL-адрес запроса к Backend-экземпляру может быть получен с помощью программного интерфейса `Backends API`:

```
import com.google.appengine.api.backends.BackendService;
import com.google.appengine.api.backends.BackendServiceFactory;
BackendService backendsService = BackendServiceFactory.getBackendService();
String url = backendsService.getBackendAddress([backend_name], [instance]);
```

В отличие от Frontend-экземпляра код, выполняемый в Backend-экземпляре, может исполняться в фоновом потоке:

```
import com.google.appengine.api.ThreadManager;
Thread thread = ThreadManager.createBackgroundThread(new Runnable() {
    public void run() {
        try {
            while (true) {
                . . .
            }
        } catch (InterruptedException ex) {
            throw new RuntimeException("Interrupted in loop:", ex);
        }
    }
});
thread.start();
```

Существующие ограничения для использования Backend-экземпляров — это максимальное количество Backend-серверов для приложения — 5, максимальное количество Backend-экземпляров для Backend-сервера — 20, максимальный объем занимаемой памяти — 10 гигабайт.

Таким образом, для создания Backend-приложения в качестве первого шага необходимо создать в папке WEB-INF конфигурационный файл `backends.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<backends>
  <backend name="backend">
    <class>Bl</class>
    <options>
      <dynamic>true</dynamic>
      <public>true</public>
    </options>
  </backend>
</backends>
```

Затем в сервлете приложения нужно перенаправить вызов ресурсу, код которого должен выполняться Backend-экземпляром:

```
package com.example.gae;
import java.io.IOException;
import javax.servlet.http.*;
import com.google.appengine.api.backends.BackendService;
import com.google.appengine.api.backends.BackendServiceFactory;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {
    BackendService backendsService = BackendServiceFactory.getBackendService();
    String url = backendsService.getBackendAddress("backend");
    resp.sendRedirect(resp.encodeRedirectURL("http://" + url + "/backend/custom.jsp"));
  }
}
```

## ГЛАВА 11



# Использование протокола OAuth 2.0 для получения доступа к Google-сервисам

Протокол OAuth обеспечивает доступ третьей стороне, в нашем случае GAE-приложению, к Web-приложению от имени пользователя, который уже авторизован для работы с данным Web-приложением.

Доступ третьей стороне с помощью протокола OAuth предоставляется в три этапа:

1. Третья сторона запрашивает OAuth-службу для получения маркера запроса.
2. Пользователь перенаправляется на Web-страницу, на которой он разрешает доступ третьей стороне к Web-приложению от своего имени.
3. Третья сторона обменивает у OAuth-службы маркер запроса на маркер доступа, с помощью которого осуществляется доступ к Web-приложению от имени пользователя.

Если сравнивать протокол OAuth с протоколом Open ID, то протокол Open ID обеспечивает для самого пользователя аутентификацию на многих сайтах с помощью одного идентификатора Open ID, а протокол OAuth позволяет пользователю авторизировать один сайт для получения данных пользователя от другого сайта.

Многочисленные продукты Google поддерживают протокол OAuth 2.0 для аутентификации и авторизации. Для того чтобы Web-приложение получило возможность использовать протокол OAuth 2.0 для доступа к Google-сервисам от имени пользователя, оно должно быть зарегистрировано в консоли APIs Console (<https://code.google.com/apis/console/>).

Для регистрации Web-приложения необходимо открыть раздел **API Access** левой панели **API Project** и нажать кнопку **Create an OAuth 2.0 client ID** (рис. 11.1).

Далее в диалоговом окне **Create Client ID** вводится информация, отображаемая пользователю при запросе разрешения доступа к Google-сервисам от его имени (рис. 11.2), тип и идентификатор приложения (рис. 11.3).

После нажатия кнопки **Create client ID** диалогового окна на странице консоли отобразятся три необходимых параметра Client ID, Client secret и Redirect URIs для доступа к службе Google OAuth 2.0.



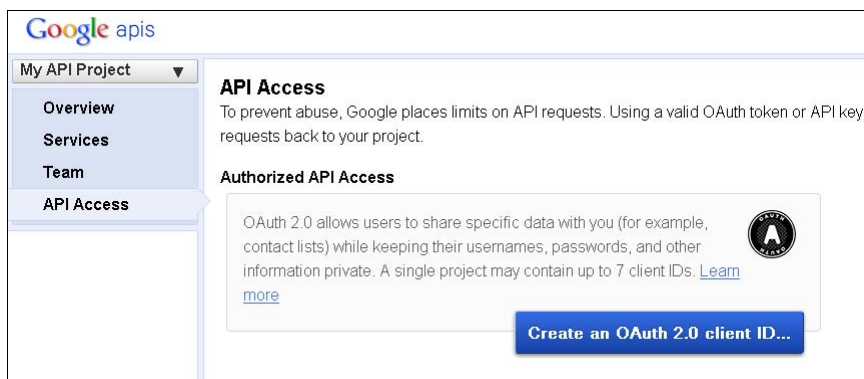


Рис. 11.1. Создание идентификаторов доступа к OAuth-службе

Рис. 11.2. Ввод информации, отображаемой пользователю при запросе разрешения доступа к Google-сервисам от его имени

Для получения маркера запроса (кода авторизации) у OAuth-службы Web-приложение должно отправить HTTP GET-запрос по адресу **https://accounts.google.com/o/oauth2/auth** со следующими параметрами:

- ◆ `response_type` — значение `code`;
- ◆ `client_id` — значение из консоли APIs Console;
- ◆ `redirect_uri` — значение из консоли APIs Console;
- ◆ `scope` — список идентификаторов Google-сервисов, к которым будет запрашиваться доступ;
- ◆ `state` — любое значение, используемое приложением при получении ответа;
- ◆ `access_type` — значение `online` (по умолчанию) указывает, что доступ запрашивается, когда пользователь использует Web-браузер, или `offline`;

**Create Client ID**

**Client ID Settings**

**Application type**

- ☒ Web application  
Accessed by web browsers over a network.
- ☐ Service account  
Calls Google APIs on behalf of your application instead of an end-user. [Learn more](#)
- ☐ Installed application  
Runs on a desktop computer or handheld device (like Android or iPhone).

**Your site or hostname** [\(more options\)](#)  
For example: `www.example.com` or `localhost`

**Redirect URI**

[Learn more](#)

Рис. 11.3. Определение типа и идентификатора приложения

◆ `approval_prompt` — значение `force` (разрешение на доступ запрашивается у пользователя каждый раз) или `auto` (по умолчанию).

Например, сервлет GAE-приложения, запрашивающий код авторизации для доступа к сервису Google Books API от имени пользователя, будет иметь следующий код:

```
package com.example.gae;
import java.io.IOException;
import javax.servlet.http.*;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        resp.sendRedirect(resp.encodeRedirectURL("https://accounts.google.com/o/oauth2/
auth?scope=https://www.googleapis.com/auth/books&redirect_uri=https://softstudioiv-
demo.appspot.com/oauth2callback&response_type=code&client_id=764330134695.apps.
googleusercontent.com&approval_prompt=force"));
    }
}
```

После получения разрешения пользователя на доступ к Google-сервису Web-приложения (рис. 11.4) OAuth-служба посылает HTTP GET-запрос с кодом авторизации по адресу, указанному в параметре **Redirect URIs** консоли APIs Console.

После получения кода авторизации Web-приложение должно отправить HTTP POST-запрос OAuth-службе для получения маркера доступа по адресу **https://accounts.google.com/o/oauth2/token** со следующими параметрами:

- ◆ code — код авторизации;
- ◆ client\_id, client\_secret, redirect\_uri — значения из консоли APIs Console;
- ◆ grant\_type — значение authorization\_code.

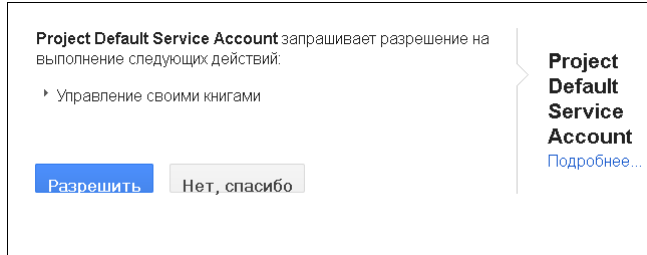


Рис. 11.4. Страница получения разрешения у пользователя на доступ Web-приложения к сервису Google Books API

В ответ будет получен HTTP POST-запрос, содержащий JSON-данные в виде:

```
{
  "access_token": "...",
  "expires_in": ...,
  "token_type": "Bearer"
}
```

Например, сервлет GAE-приложения, обрабатывающий GET-запрос с кодом авторизации, посылающий POST-запрос для получения маркера доступа и считывающий ответ, содержащий маркер доступа, будет иметь следующий код:

```
package com.example.gae;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.URL;
import java.net.URLConnection;
import java.net.URLEncoder;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@SuppressWarnings("serial")
public class AuthServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        String code=req.getParameter("code");
        URL url = new URL("https://accounts.google.com/o/oauth2/token");
        String content = "code=" + URLEncoder.encode(code, "UTF-8")
            + "&client_id="
            + URLEncoder.encode("764330134695.apps.googleusercontent.com", "UTF-8")
```

```
+ "&client_secret="
+ URLEncoder.encode("tSN9Q2iug8s34XiZyG6ANDoT", "UTF-8")
+ "&redirect_uri="
+ URLEncoder.encode("https://softstudioiv-demo.appspot.com/oauth2callback", "UTF-8")
+ "&grant_type="
+ URLEncoder.encode("authorization_code", "UTF-8");
// POST-запрос:
URLConnection conn = url.openConnection();
conn.setDoOutput(true);
OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream());
    wr.write(content);
    wr.flush();
// POST-ответ:
BufferedReader rd = new BufferedReader(new InputStreamReader(conn.getInputStream()));
    StringBuilder sb = new StringBuilder();
    String line=rd.readLine();
    while (line != null) {
        sb.append(line + "\n");
        line = rd.readLine();
    }
    String data = sb.toString();
    resp.getWriter().write(data);
    wr.close();
    rd.close();
}}
```

В результате Web-приложение может осуществлять запросы к Google-сервису от имени пользователя с HTTP-заголовком `Authorization:`, содержащим маркер доступа `access_token`.

Демонстрационная страница использования протокола OAuth 2.0 для доступа к Google-сервисам находится по адресу <https://code.google.com/oauthplayground/>.

## Служба URL Fetch

В рассмотренном ранее примере сервлет GAE-приложения после получения кода авторизации выступает как обычный Java-клиент, запрашивая и получая ресурс по URL-адресу с помощью библиотеки `java.net`. Такое взаимодействие с сетью в платформе App Engine обеспечивает служба URL Fetch, выступающая как посредник и использующая HTTP-прокси сервер. При этом служба URL Fetch накладывает ограничения на размер запроса и ответа — до 1 Мбайт и не поддерживает постоянное HTTP-соединение, завершая его после получения ресурса и не позволяя создавать соединения через `java.net.Socket`.

Базовым классом библиотеки `java.net` является класс `URL`, экземпляр которого создается на основе строки URL-адреса с помощью конструктора класса. Для кодировки строки URL-адреса предварительно можно использовать класс `URI`:

```
URI uri = new URI(...);
URL url = uri.toURL();
```

Для кодировки параметров запроса может быть использован статический метод `public static String encode(String s, String enc)` класса `URLEncoder`.

После создания объекта `URL`, для извлечения ресурса непосредственно из URL-адреса открывается и считывается входящий поток, используя метод `openStream()` класса `URL`:

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(url.openStream()));
String inputLine;
while ((inputLine = in.readLine()) != null)
    ...
in.close();}
```

Для отправки запроса по URL-адресу необходимо создать объект `URLConnection` методом `openConnection()` класса `URL`, установить метод запроса с помощью метода `setRequestMethod()` класса `URLConnection`, открыть исходящий поток методом `getOutputStream()` класса `URLConnection` и записать в него сообщение:

```
URLConnection connection = (URLConnection) url.openConnection();
connection.setDoOutput(true);
connection.setRequestMethod("POST");
OutputStreamWriter writer = new OutputStreamWriter(connection.getOutputStream());
writer.write([message]);
writer.close();
```

Помимо библиотеки `java.net` организовать Java-клиента обмена данными по URL-адресу позволяет программный интерфейс `URL Fetch API`, представленный пакетом `com.google.appengine.api.urlfetch`. В этом случае для использования службы `URL Fetch` в качестве первого шага необходимо создать объект `URLFetchService` статическим методом `public static URLFetchService getURLFetchService()` класса-фабрики `URLFetchServiceFactory`. После этого методы `fetch()` и `fetchAsync()` интерфейса `URLFetchService` обеспечивают синхронную и асинхронную отправку запросов и получение ответов.

В качестве аргументов методы интерфейса `URLFetchService` могут использовать объект `java.net.URL` или объект `com.google.appengine.api.urlfetch.HTTPRequest`, создаваемый с помощью конструктора:

```
public HTTPRequest(java.net.URL url, HTTPMethod method,
    FetchOptions fetchOptions)
```

где объект `com.google.appengine.api.urlfetch.HTTPMethod` представляет собой перечисление с полями `DELETE`, `GET`, `HEAD`, `POST` и `PUT`, а объект `com.google.appengine.api.urlfetch.FetchOptions` создается с помощью статических методов класса-фабрики `FetchOptions.Builder`. В частности, метод `withDeadline(double deadline)` позволяет уста-

новить временной порог запроса, а метод `allowTruncate()` — определить усечение слишком больших ответов без возникновения исключительной ситуации.

Методы `setHeader(HTTPHeader header)` и `setPayload(byte[] payload)` класса `HTTPRequest` обеспечивают определение заголовка и тела запроса. При этом объект `com.google.appengine.api.urlfetch.HTTPHeader` создается с помощью конструктора класса:

```
public HTTPHeader(java.lang.String name, java.lang.String value)
```

Методы интерфейса `URLFetchService` возвращают объект `com.google.appengine.api.urlfetch.HTTPResponse`, представляющий ответ запроса, метод `public byte[] getContent()` которого позволяет извлечь запрашиваемый ресурс.

## ГЛАВА 12



# Запланированные задачи и очередь задач

Servlet-контейнер платформы App Engine обеспечивает вызов сервлета GAE-приложения для быстрой обработки HTTP-запроса и выдачи ответа клиенту. GAE-службы Cron и Task Queue дополняют данную схему работы возможностью выполнять приложением задачи вне клиентского запроса.

Служба Cron позволяет определить вне кода приложения с помощью конфигурационного файла `cron.xml` HTTP GET-вызов относительного URL-адреса приложения в конкретное время регулярно через заданные интервалы времени.

Служба Task Queue дает возможность в коде сервлета приложения добавить в очередь относительные URL-адреса приложения, которые затем будут вызываться GAE-платформой вне клиентского запроса. В отличие от службы Cron, служба Task Queue позволяет оперировать всем набором методов HTTP-запроса, а не только методом GET. Также с помощью службы Task Queue можно добавлять в очередь сообщения, которые затем могут извлекаться из нее и обрабатываться как самим GAE-приложением, так и клиентом вне платформы App Engine, использующим программный интерфейс Task Queue REST API.

## Служба Cron

Служба Cron обеспечивает для GAE-приложения выполнение всех возможных запланированных задач, таких как обновление данных приложения в хранилище или кэше, рассылки электронной почты и др. При этом запланированные задачи представлены относительными URL-адресами приложения, которые с помощью дескриптора `web.xml` могут быть связаны с сервлетами, выполняющими код задач.

Определить расписание HTTP GET-запросов GAE-платформой по заданным URL-адресам позволяет конфигурационный файл `cron.xml` приложения. GAE-приложение с бесплатным хостингом может определить до 10 запланированных задач, с платным хостингом — до 100 запланированных задач.

Конфигурационный файл `cron.xml` создается в папке `war\WEB-INF` приложения, и его корневым элементом служит тег `<cronentries>` (рис. 12.1).

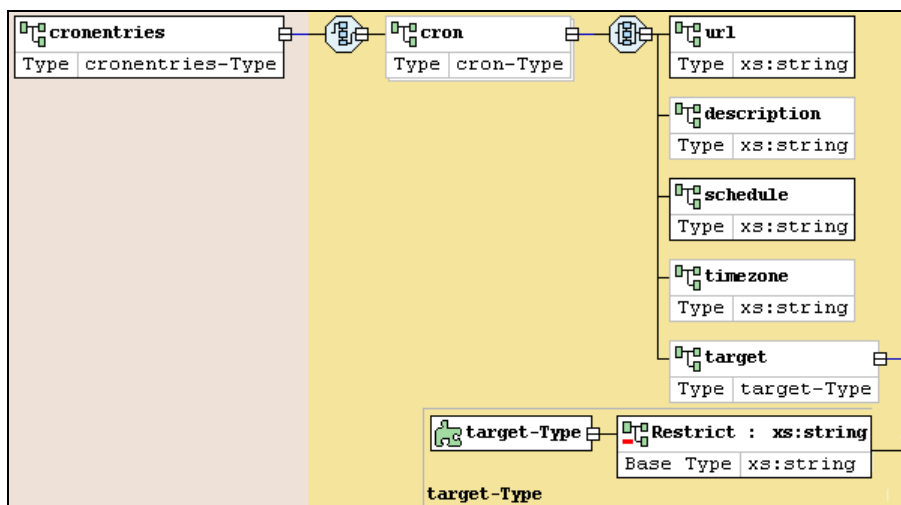


Рис. 12.1. Общая схема конфигурационного файла cron.xml определения запланированных задач GAE-приложения

Каждая запланированная задача определяется внутри тега `<cron>` с помощью его дочерних тегов `<url>`, `<description>`, `<schedule>`, `<timezone>` и `<target>`.

Обязательный тег `<url>` указывает URL-адрес HTTP GET-запроса.

Обязательный тег `<schedule>` указывает расписание вызова GAE-платформой URL-адреса в формате:

[порядок] [дни недели] ["of" [месяцы]] [время дня]

где **порядок** — порядковый номер дней недели месяцев, вызываемых в указанное время дня. Например, расписание:

1st monday of sep,oct,nov 17:00

означает вызов URL-адреса каждый первый понедельник сентября, октября и ноября в пять вечера.

Или же расписание может быть определено в формате:

every [интервал] [hours|mins|minutes] ["from" (time) "to" (time)]

Например, расписание:

every 5 minutes from 10:00 to 14:00

означает вызов URL-адреса каждые пять минут с 10 до 14 часов ежедневно.

Дополнительный тег `<description>` содержит текстовое описание запланированной задачи, отображаемое в панели администрирования.

Дополнительный тег `<timezone>` указывает TZ-имя временной зоны расписания ([http://en.wikipedia.org/wiki/List\\_of\\_zoneinfo\\_time\\_zones](http://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones)) (по умолчанию UTC).

Дополнительный тег `<target>` указывает версию GAE-приложения, для которой будет выполняться расписание.



После развертывания GAE-приложения его запланированные задачи будут отображаться в разделе **Main | Cron Jobs** консоли администрирования приложением (рис. 12.2).

<a href="#">Main</a> <a href="#">Dashboard</a> <a href="#">Instances</a> <a href="#">Logs</a> <a href="#">Versions</a> <a href="#">Backends</a> <a href="#">Cron Jobs</a> <a href="#">Task Queues</a> <a href="#">Quota Details</a> <a href="#">Endpoints</a>	<p>Below are scheduled tasks (cron jobs) for the application. Cron jobs are defined in <code>cron.yaml</code> (Python and Go) or <code>cron.xml</code> (Java).  <a href="#">Learn more about cron.</a></p> <p>« Prev 10   Next 10 »      Order by: <a href="#">Job Description</a>   <a href="#">Job ETA</a></p> <table> <tr> <th>Cron Job</th><th>Schedule/Last Run/Last Status (All times are UTC)</th></tr> <tr> <td>/gaeapplication Repopulate the cache every 2 minutes</td><td>every 2 minutes (UTC) 2012/07/21 11:27:17 <b>on time Success</b></td></tr> </table>	Cron Job	Schedule/Last Run/Last Status (All times are UTC)	/gaeapplication Repopulate the cache every 2 minutes	every 2 minutes (UTC) 2012/07/21 11:27:17 <b>on time Success</b>
Cron Job	Schedule/Last Run/Last Status (All times are UTC)				
/gaeapplication Repopulate the cache every 2 minutes	every 2 minutes (UTC) 2012/07/21 11:27:17 <b>on time Success</b>				

**Рис. 12.2.** Запланированная задача обновления кэша приложения в консоли администрирования

Для того чтобы отменить ранее определенные запланированные задачи, необходимо загрузить GAE-приложение с конфигурационным файлом `cron.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries/>
```

## Служба Task Queue

Служба Task Queue позволяет в коде сервлета добавить в очередь задач HTTP-вызовы относительных URL-адресов приложения, исполняемые платформой App Engine вне клиентского запроса. Кроме того, служба Task Queue дает возможность добавлять в очередь данные, которые затем могут быть извлечены и обработаны как в самом GAE-приложении, так и внешним клиентом.

Создавать очереди HTTP-вызовов позволяет Push-конфигурация службы Task Queue, а организовывать очереди сообщений — Pull-конфигурация службы Task Queue.

Просмотр статистики использования и управление очередями приложения может быть выполнено в разделе **Main | Task Queues** консоли администрирования (<https://appengine.google.com/>).

## Очереди Push

Взаимодействие кода GAE-приложения со службой Task Queue обеспечивает программный интерфейс Task Queue API, представленный пакетом `com.google.appengine.api.taskqueue`.

Для Push-конфигурации служба Task Queue предоставляет очередь по умолчанию, объект которой может быть создан методом `getDefaultQueue()` класса-фабрики `com.google.appengine.api.taskqueue.QueueFactory`:

```
Queue queue = QueueFactory.getDefaultQueue();
```

Однако конфигурационный файл `queue.xml` папки `war\WEB-INF` приложения позволяет создавать свои очереди и переопределять очередь по умолчанию. Корневым элементом XML-файла `queue.xml` служит тег `<queue-entries>`, дочерние теги `<queue>` которого описывают очереди приложения для службы Task Queue (рис. 12.3 и 12.4).

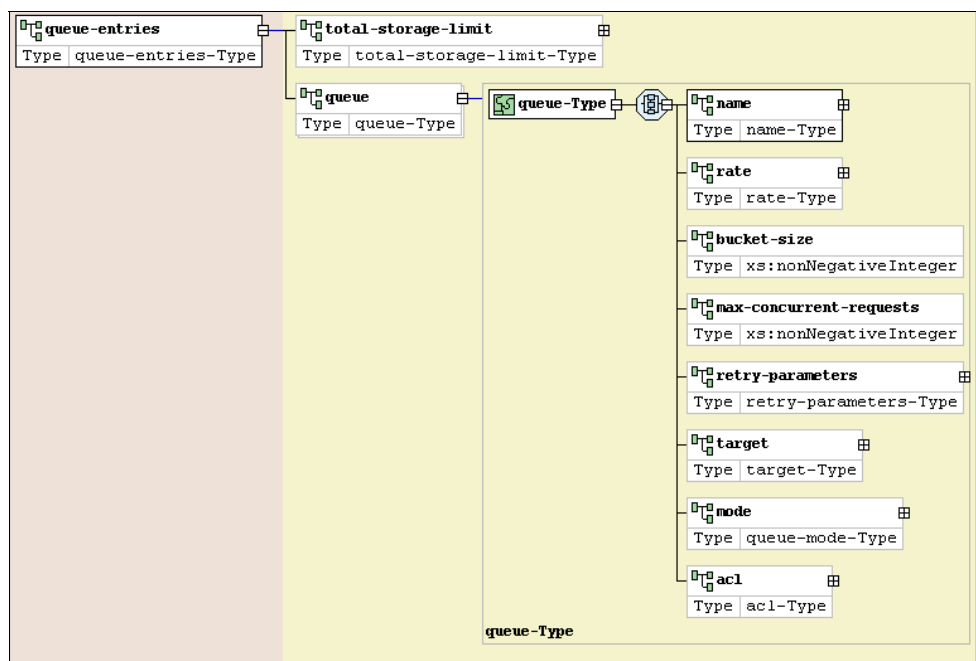


Рис. 12.3. Общая схема конфигурационного файла `queue.xml` определения очередей службы Task Queue

Тег `<total-storage-limit>` файла `queue.xml` позволяет установить ограничение для общего потребления памяти всеми очередями приложения в формате `[значение]B|K|M|G|T`. Для бесплатного GAE-хостинга по умолчанию такое ограничение установлено в размере 500 Мбайт. Рекомендуется устанавливать данное ограничение как защиту от неконтролируемого потребления памяти в результате ошибки приложения.

Каждая очередь приложения может быть определена с помощью следующих дочерних тегов тега `<queue>`:

- ◆ `<name>` — имя очереди, которое используется при создании объекта очереди методом `getQueue(java.lang.String queueName)` класса-фабрики `QueueFactory`. Если имя очереди "default", тогда данные теги переопределяют очередь по умолчанию службы Task Queue;
- ◆ `<bucket-size>` — максимальное количество задач для контейнера задач очереди (по умолчанию 5);
- ◆ `<rate>` — частота, с которой задачи добавляются в контейнер задач очереди (по умолчанию 5 задач в секунду) в формате `[значение]/s|m|h|d`. Как только контей-

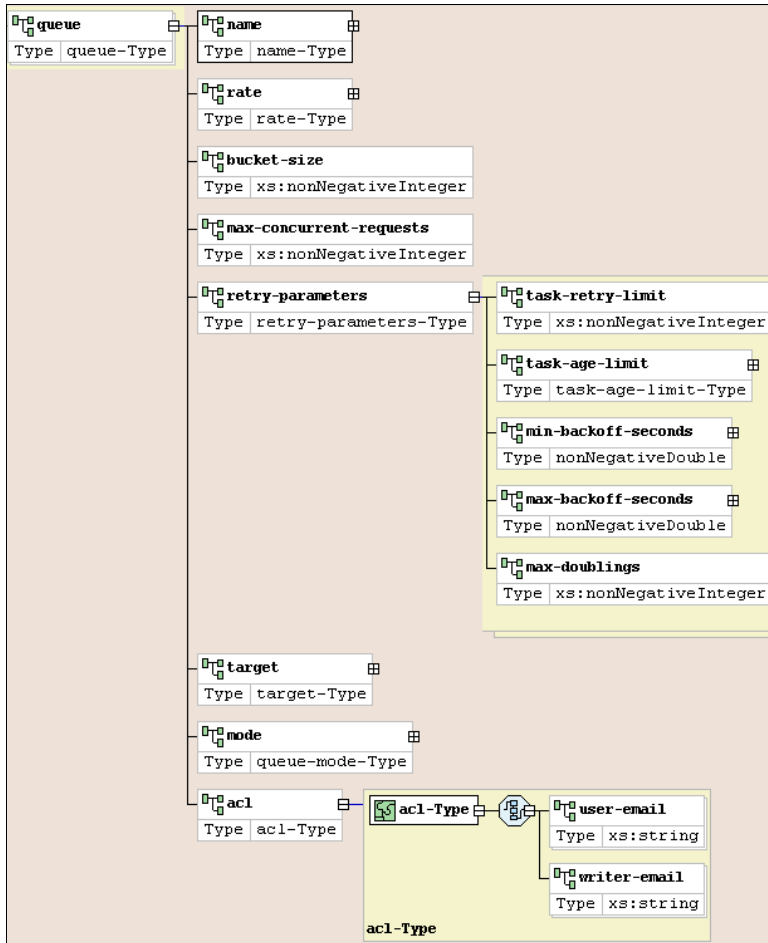


Рис. 12.4. Элементы конфигурационного файла queue.xml описания очереди службы Task Queue

нер наполнится, его задачи начинают выполняться. Задачи очереди должны быть выполнены в течение 10 мин от первоначального клиентского запроса, иначе возникает исключительная ситуация `com.google.apphosting.api.DeadlineExceededException`;

- ◆ `<max-concurrent-requests>` — максимальное количество задач, которые могут выполняться параллельно;
- ◆ `<retry-parameters>` — если задачу очереди не удалось выполнить, служба Task Queue будет повторять попытки ее выполнения. Данный тег определяет параметры повторных попыток выполнения задачи с помощью следующих дочерних тегов:
  - `<task-retry-limit>` — максимальное количество повторных попыток;
  - `<task-age-limit>` — период времени для повторных попыток в формате [значение]s|m|h|d;

- `<min-backoff-seconds>` — минимальное количество секунд между повторными попытками;
- `<max-backoff-seconds>` — максимальное количество секунд между повторными попытками;
- `<max-doublings>` — максимальное количество удвоения интервала между повторными попытками, пока интервал не станет константой  $2 \times (\text{max-doublings} - 1) \times \text{min-backoff-seconds}$ ;
- ◆ `<target>` — версия приложения, для которой будут выполняться задачи;
- ◆ `<mode>` — конфигурация службы Task Queue, значение по умолчанию `push`;
- ◆ `<acl>` — только для Pull-конфигурации службы Task Queue определяет внешних потребителей задач очереди.

После создания объекта очереди `com.google.appengine.api.taskqueue.Queue` методом `getDefaultQueue()` или методом `getQueue(java.lang.String queueName)` класса-фабрики `com.google.appengine.api.taskqueue.QueueFactory` добавить HTTP-вызов в очередь можно методом `add(TaskOptions taskOptions)` интерфейса `Queue`.

Объект `com.google.appengine.api.taskqueue.TaskOptions` определяет параметры задачи и создается с помощью статических методов класса-фабрики `TaskOptions.Builder`. Как правило, для Push-очереди объект `TaskOptions` создается статическим методом `withUrl(java.lang.String url)` класса-фабрики `TaskOptions.Builder`, указывающим вызываемый относительный URL-адрес приложения, связанный в дескрипторе `web.xml` с сервлетом приложения. Далее параметры задачи определяются методами класса `TaskOptions`. В частности:

- ◆ метод `method(TaskOptions.Method method)` определяет метод HTTP-запроса;
- ◆ метод `param(java.lang.String name, java.lang.String value)` добавляет параметры запроса;
- ◆ метод `header(java.lang.String headerName, java.lang.String value)` добавляет заголовок к запросу. Используя данный метод, можно вызвать Backend-экземпляр приложения и тем самым уйти от ограничения в 10 мин на выполнение задачи:
 

```
header("Host", BackendServiceFactory.getBackendService().getBackendAddress("backend", 1))
```
- ◆ метод `countdownMillis(long countdownMillis)` устанавливает задержку выполнения задачи;
- ◆ метод `etaMillis(long etaMillis)` устанавливает время выполнения задачи, совместимое с `System.currentTimeMillis()`.

## Отложенные задачи *DeferredTask*

С помощью интерфейса `com.google.appengine.api.taskqueue.DeferredTask` в Push-очередь можно добавить не HTTP-вызов URL-адреса, связанного с обработчиком приложения, а сразу исполняемый метод класса приложения. Для этого класс приложения должен реализовывать интерфейс `DeferredTask` с определением его метода `run()`, который и будет выполняться в очереди:

```
import com.google.appengine.api.taskqueue.DeferredTask;
public class SampleDeferredTask implements DeferredTask{
    private String key;
    private String value;

    public SampleDeferredTask (String key, String value){
        ...
    }
    @Override
    public void run() {
        ...
    }
}
Queue queue = QueueFactory.getDefaultQueue();
TaskOptions taskOptions = TaskOptions.Builder.withPayload(
    new SampleDeferredTask(key, value));
queue.add(taskOptions);
```

Так как при добавлении `DeferredTask`-объекта в очередь он сериализуется, необходимо контролировать сериализуемость объектов класса.

## Очереди Pull

Pull-конфигурация службы Task Queue позволяет добавлять в очередь данные, которые затем могут быть из нее извлечены.

Для Pull-конфигурации не существует очереди по умолчанию, поэтому Pull-очередь должна быть описана в конфигурационном файле `queue.xml` папки `war\WEB-INF` приложения. При этом тег `<mode>` файла `queue.xml` должен содержать значение `pull`, а теги `<bucket-size>`, `<max-concurrent-requests>`, `<rate>`, `<task-age-limit>`, `<min-backoff-seconds>`, `<max-backoff-seconds>`, `<max-doublings>` и `<target>` не могут использоваться, т. к. они предназначены только для определения Push-очерей.

С помощью тега `<acl>` файла `queue.xml` и его дочерних тегов `<user-email>` и `<writer-email>` для Pull-очереди определяются внешние потребители данных из очереди, использующие для доступа к очереди программный интерфейс Task Queue REST API (<https://developers.google.com/appengine/docs/java/taskqueue/rest>) и имеющие Google-аккаунт. Адрес электронной почты Google-аккаунта, указанный в теге `<user-email>`, дает право на операции List, Get, Lease, Delete и Update программного интерфейса REST API, а адрес электронной почты Google-аккаунта, указанный в теге `<writer-email>`, дополнительно дает право на операцию Insert программного интерфейса REST API.

Для использования Pull-очереди в самом GAE-приложении может применяться библиотека `com.google.appengine.api.taskqueue`. В этом случае объект очереди `com.google.appengine.api.taskqueue.Queue` создается методом `getQueue(java.lang.String queueName)` класса-фабрики `com.google.appengine.api.taskqueue.QueueFactory`. После этого данные добавляются в очередь методом `add(TaskOptions taskOptions)` интерфейса `Queue`, где объект `TaskOptions` создается методом `TaskOptions.Builder.withMethod(TaskOptions.Method.PULL)`.

Сами данные добавляются методами `payload(byte[] payload)`, `payload(byte[] payload, java.lang.String contentType)`, `payload(java.lang.String payload)` или `payload(java.lang.String payload, java.lang.String charset)` класса `TaskOptions`.

Из Pull-очереди данные могут быть извлечены методом `leaseTasks(LeaseOptions options)` интерфейса `Queue`, при этом до завершения процесса обработки данные становятся недоступными для других потребителей. Объект `LeaseOptions` создается с помощью статических методов класса-фабрики `LeaseOptions.Builder`, например методом `withLeasePeriod(long lease, java.util.concurrent.TimeUnit unit)`, где `lease` — время аренды задачи (максимум одна неделя), а `unit` — формат значения `lease`. Затем конфигурация аренды задач очереди определяется методами класса `LeaseOptions`. В частности:

- ◆ метод `countLimit(long countLimit)` указывает количество арендуемых задач, максимум 1000 задач;
- ◆ метод `deadlineInSeconds(java.lang.Double deadlineInSeconds)` определяет максимальное время ожидания Lease-запроса;
- ◆ методы `tag(byte[] tag)`, `tag(java.lang.String tag)` и `groupByTag()` позволяют отфильтровать и сгруппировать извлечение задач очереди по тегам, которые присоединяются к задаче методом `tag(byte[] tag)` или `tag(java.lang.String tag)` класса `TaskOptions` при добавлении задачи в очередь.

Метод `leaseTasks()` интерфейса `Queue` возвращает список `java.util.List<TaskHandle>` объектов `TaskHandle`, из которых данные извлекаются методом `getPayload()`.

Потребитель задачи Pull-очереди может удалить ее после обработки методом `deleteTask()`.

Для создания внешнего Java-потребителя задач Pull-очереди может использоваться Java-библиотека Google APIs Client Library for Java ([http://code.google.com/p/google-api-java-client/wiki/APIs#TaskQueue\\_API](http://code.google.com/p/google-api-java-client/wiki/APIs#TaskQueue_API)), упрощающая доступ к Pull-очереди с помощью программного интерфейса Task Queue REST API.

## ГЛАВА 13



# Службы поиска Search и Prospective Search

Службы Search и Prospective Search платформы App Engine позволяют организовать полнотекстовый поиск по документам, которыми оперирует приложение.

С помощью службы Search документы приложения индексируются с возможностью последующих запросов к индексам.

Служба Prospective Search дает возможность заранее сформировать запрос, относительно которого будут проверяться входящие документы приложения, и автоматически добавляет соответствующие запросу документы в очередь Task Queue, которая отвечает за HTTP POST-вызов обработчика приложения.

## Служба Search

Для применения службы Search платформы App Engine данные приложения, предназначенные для участия в полнотекстовом поиске, обертываются объектом `com.google.appengine.api.search.Document`, представляющим документ приложения, который добавляется в индекс `com.google.appengine.api.search.Index` службы Search.

Объект `Document` создается с помощью класса-фабрики `com.google.appengine.api.search.Document.Builder`, экземпляр которого в свою очередь создается статическим методом `newBuilder()` класса `Document`. Метод `addField(Field field)` класса-фабрики `Document.Builder` добавляет в документ поле, содержащее данные приложения, а метод `build()` создает документ.

Поле документа `com.google.appengine.api.search.Field` создается с помощью класса-фабрики `com.google.appengine.api.search.Field.Builder`, экземпляр которого в свою очередь создается статическим методом `newBuilder()` класса `Field`. Поле `Field` может обертывать данные приложения типа `TEXT` (простой текст), `HTML` (текст в формате HTML), `ATOM` (целостная строка), `NUMBER` (число типа `int`, `long` и `float`), `DATE` (дата типа `java.util.Date`) и `GEO_POINT` (местоположение типа `com.google.appengine.api.search.GeoPoint`). Поле `Field` именуется с помощью метода `setName(java.lang.String name)` класса-фабрики `Field.Builder`, а данные приложения добавляются в поле документа соответствующими методами `setText(java.lang.String text)`, `setHTML(java.lang.String html)`, `setAtom(java.lang.String atom)`, `setNumber(double number)`, `setDate(java.util.Date`

date) и `setGeoPoint(GeoPoint geoPoint)`. Метод `build()` класса-фабрики `Field.Builder` завершает создание поля.

Создание индекса службы Search начинается с создания объекта `com.google.appengine.api.search.IndexSpec`, определяющего конфигурацию индекса. Объект `IndexSpec` создается с помощью класса-фабрики `com.google.appengine.api.search.IndexSpec.Builder`, экземпляр которого в свою очередь создается статическим методом `newBuilder()` класса `IndexSpec`. Метод `setName(java.lang.String name)` класса-фабрики `IndexSpec.Builder` определяет имя индекса, метод `setConsistency(Consistency consistency)` — режим согласованности индекса `Consistency.GLOBAL` (гарантирует при запросе включение всех последних изменений в индексе) или `Consistency.PER_DOCUMENT` (более производительный режим), а метод `build()` завершает создание объекта `IndexSpec`.

Для создания индекса на основе объекта `IndexSpec` создается объект `com.google.appengine.api.search.SearchService`, обеспечивающий связь со службой Search, метод `getIndex(IndexSpec spec)` которого возвращает объект `com.google.appengine.api.search.Index`:

```
SearchService searchService = SearchServiceFactory.getSearchService();
IndexSpec spec = IndexSpec.newBuilder()
    .setName("nameIndex")
    .setConsistency(Consistency.PER_DOCUMENT)
    .build();
Index index = searchService.getIndex(spec);
```

После создания индекса документ добавляется к нему с помощью метода `add(Document... documents)`, `add(java.lang.Iterable<Document> documents)`, `addAsync(Document... document)` или `addAsync(java.lang.Iterable<Document> documents)`.

Для осуществления полнотекстового поиска в индексе используются методы `search(Query query)`, `search(java.lang.String query)`, `searchAsync(Query query)` или `searchAsync(java.lang.String query)`, где объект `com.google.appengine.api.search.Query`, представляющий запрос, создается с помощью класса-фабрики `com.google.appengine.api.search.Query.Builder`, экземпляр которого в свою очередь создается статическим методом `newBuilder()` класса `Query`. Метод `setOptions(QueryOptions options)` класса-фабрики определяет параметры запроса, а метод `build(java.lang.String queryString)` завершает создание объекта `Query`.

Объект `QueryOptions` создается с помощью класса-фабрики `com.google.appengine.api.search.QueryOptions.Builder`, экземпляр которого в свою очередь создается статическим методом `newBuilder()` класса `QueryOptions`. Методы `setLimit()`, `setFieldsToReturn()`, `setFieldsToSnippet()`, `setSortOptions()` и `setCursor()` класса-фабрики `QueryOptions.Builder` устанавливают количество возвращаемых запросом результатов, возвращаемые запросом поля документа, сортировку результатов запроса и курсор поиска. Метод `build()` класса-фабрики `QueryOptions.Builder` завершает создание объекта `QueryOptions`.

Метод `search()` интерфейса `com.google.appengine.api.search.Index` возвращает список `com.google.appengine.api.search.Results<ScoredDocument>` соответствующих запросу до-



кументов, из которых методом `getField(java.lang.String name)` класса `Document` могут быть извлечены поля с последующим извлечением данных методами `get()` класса `Field`.

В качестве примера рассмотрим простое GAE-приложение, в котором пользователи оставляют свои комментарии и могут выполнить полнотекстовый поиск по всем комментариям.

Страница приветствия такого приложения содержит две формы — одну для ввода комментариев, а другую — для ввода строки поиска:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <title>Hello App Engine</title>
  </head>
  <body style="text-align:center">
// Форма отправки комментариев
    <form method="post" action="/gaeapplication" >
    <p>
      Comments:
      <input name="comments" type="text" size="30">
    </p>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
    </form>
// Форма поиска
    <form method="get" action="/gaeapplication" >
    <p>
      Search:
      <input name="query" type="text" size="30">
    </p>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
    </form>
  </body>
</html>
```

Метод сервлета приложения обработки POST-запросов сохраняет комментарии в Datastore-хранилище и добавляет их в индекс службы Search, а метод обработки GET-запросов на основе строки запроса осуществляет поиск в индексе и возвращает найденные комментарии пользователю:

```
package com.example.gae;
import java.io.IOException;
import javax.servlet.http.*;
import com.google.appengine.api.search.*;
import com.google.appengine.api.search.Index;
```

```

import com.google.appengine.api.datastore.*;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
        Entity UserComments = new Entity("UserComments");
        String text=(String)req.getParameter("comments");
        UserComments.setProperty("comments", text);
        datastore.put(UserComments);
        Document document = Document.newBuilder().addField(Field.newBuilder()
            .setName("comments")
            .setText(text))
            .build();
        IndexSpec spec = IndexSpec.newBuilder()
            .setName("comments")
            .setConsistency(Consistency.PER_DOCUMENT)
            .build();
        SearchService searchService = SearchServiceFactory.getSearchService();
        Index index = searchService.getIndex(spec);
        index.add(document);
        resp.sendRedirect("/");
    }

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        String query=(String)req.getParameter("query");
        IndexSpec spec = IndexSpec.newBuilder()
            .setName("comments")
            .setConsistency(Consistency.PER_DOCUMENT)
            .build();
        SearchService searchService = SearchServiceFactory.getSearchService();
        Index index = searchService.getIndex(spec);
        Results<ScoredDocument> results = index.search(query);
        resp.setContentType("text/plain");
        resp.getWriter().println("Search results:");
        for (ScoredDocument document : results) {
            String text=document.getOnlyField("comments").getText();
            resp.getWriter().println(text);
        }
    }
}

```

#### web.xml:

```

<servlet>
    <servlet-name>GAEApplication</servlet-name>
    <servlet-class>com.example.gae.GAEApplicationServlet</servlet-class>
</servlet>

```

```
<servlet-mapping>
  <servlet-name>GAEApplication</servlet-name>
  <url-pattern>/gaeapplication</url-pattern>
</servlet-mapping>
```

## Служба Prospective Search

С помощью службы Prospective Search можно заранее зарегистрировать запрос, относительно которого будут проверяться входящие документы приложения. При соответствии параметров нового документа приложения параметрам зарегистрированного запроса служба Prospective Search автоматически добавляет новый документ в очередь Task Queue, которая отвечает за HTTP POST-вызов обработчика приложения по адресу `/_ah/prospective_search`.

Примером приложения, основанным на службе Prospective Search, может служить сервис Google Alerts (<http://www.google.com/alerts>), обеспечивающий доставку уведомлений при появлении новых результатов поиска, статей или сообщений по заданному запросу (рис. 13.1).

Рис. 13.1. Страница сервиса Google Alerts

Для применения службы Prospective Search в коде GAE-приложения в качестве первого шага регистрируется запрос посредством метода `subscribe()` интерфейса `com.google.appengine.api.prospectivesearch.ProspectiveSearchService`. При этом объект `ProspectiveSearchService` создается с помощью статического метода `getProspectiveSearchService()` класса-фабрики `com.google.appengine.api.prospectivesearch.ProspectiveSearchServiceFactory`:

```
ProspectiveSearchService prospectiveSearch =
ProspectiveSearchServiceFactory.getProspectiveSearchService();
```

Метод `subscribe()` имеет следующие параметры:

- ◆ `java.lang.String topic` — тема подписки на проверку соответствия входящих данных заданному запросу;

- ◆ `java.lang.String subId` — идентификатор подписки;
- ◆ `long leaseDurationSeconds` — время действия подписки в секундах;
- ◆ `java.lang.String query` — строка запроса для поиска во входящих документах;
- ◆ `java.util.Map<java.lang.String, FieldType> schema` — набор полей входящих документов приложения, в которых будет осуществляться поиск.

Входящие документы приложения, которые будут проверяться на соответствие заданному запросу, должны быть представлены объектами `com.google.appengine.api.datastore.Entity`, а поля, в которых будет осуществляться поиск, — свойствами объектов `Entity`.

При создании нового документа — `Entity`-объекта приложения — для проверки его соответствия зарегистрированной подписке применяется метод `match()` интерфейса `ProspectiveSearchService`, принимающий в качестве аргумента `Entity`-объект.

Если новый `Entity`-объект соответствует подписке, служба `Prospective Search` отправляет HTTP POST-запрос с параметром `document` по адресу `/_ah/prospective_search` сервлета-обработчика приложения. При этом `Entity`-объект может быть извлечен из POST-запроса методом `getDocument(HttpServletRequest matchCallbackPost)` интерфейса `ProspectiveSearchService`.

В качестве примера рассмотрим простое GAE-приложение, на странице которого пользователь вводит комментарии, и если они содержат заранее заданное слово, служба `Channel` отправляет уведомление пользователю.

Страница приветствия такого приложения содержит две формы — одну для ввода запроса службы `Prospective Search`, а другую — для отправки комментариев:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script type="text/javascript" src="/scripts/jquery-1.7.2.min.js" ></script>
<script type="text/javascript" src="/_ah/channel/jsapi"></script>
<title>Insert title here</title>
</head>
<body>
<!-- Ввод сообщения:-->
Type message:
<br>
<input id="input_msg" type="text" size="30" name="message"/>
<input id="submit_msg" type="button" value="Submit"/>
<br>
<!-- Ввод запроса -->
Type query:
<br>
<input id="input_query" type="text" size="30" name="message"/>
<input id="submit_query" type="button" value="Submit"/>
```

```

<br>
<!-- Блоки получения сообщения:-->
<div id="opened"></div>
<div id="message"></div>
<div id="error"></div>
<div id="close"></div>
<script type="text/javascript">
$(document).ready(function() {
// Получение маркера канала и отправка запроса:
var token='';
$('#submit_query').click(function() {
    var query=$('#input_query').val();
    $.ajax({
        type: 'GET',
        url: '/gaeapplication',
        data:{ query: query },
        async:false,
        success: function(data) {
result = data.getElementsByTagName('result')[0].firstChild.data;
if(result!=""){
    token=result;
    // Открытие канала:
    var channel = new goog.appengine.Channel(token);
    var socket = channel.open();
    socket.onopen = onOpened;
    socket.onmessage = onMessage;
    socket.onerror = onError;
    socket.onclose = onClose;
    $('#input_query').val("");
}}});
});
// Отправка сообщения:
$('#submit_msg').click(function() {
    var message=$('#input_msg').val();
    $.post("/gaeapplication", { message: message });
    $('#input_msg').val("");
});});
// Функции обработки входящих сообщений:
function onOpened(){
}
function onError(error){
    $('#error').append('<p> Code: '+error.code+'</p><p> Description: '+error.description+'</p>');
}
function onClose(){ }
function onMessage(message){
    $('#message').append('<p>'+message.data+'</p>');
}

```

```

</script>
</body>
</html>

```

Сервлет GAE-приложения в методе-обработчике GET-запросов выдает маркер Channel-канала и регистрирует подписку службы Prospective Search, а в методе-обработчике POST-запросов принимает входящие сообщения от пользователя и ставит их на проверку соответствия зарегистрированной подписке, а также принимает уведомления от службы Prospective Search и перенаправляет их пользователю:

```

package com.example.gae;
import java.io.IOException;
import java.util.*;
import javax.servlet.http.*;
import com.google.appengine.api.channel.*;
import com.google.appengine.api.datastore.*;
import com.google.appengine.api.prospectivesearch.*;
@SuppressWarnings("serial")
public class GAEApplicationServlet extends HttpServlet {
    // Выдача маркера канала и создание запроса:
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        ChannelService channelService = ChannelServiceFactory.getChannelService();
        String query = (String)req.getParameter("query");
        String topic = "Comments";
        String token = channelService.createChannel("Comments");
        resp.setContentType("text/xml");
        resp.getWriter().println("<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>");
        resp.getWriter().println("<response>");
        resp.getWriter().println("<method>getURL</method>");
        resp.getWriter().println("<result>"+token+"</result>");
        resp.getWriter().println("</response>");

        ProspectiveSearchService prospectiveSearch =
            ProspectiveSearchServiceFactory.getProspectiveSearchService();
        String subscriptionId = query;
        long leaseTimeInMilliseconds = 24 * 60 * 60 * 1000;
        Map<String, FieldType> schema = new HashMap<String, FieldType>();
        schema.put("comments", FieldType.STRING);
        prospectiveSearch.subscribe(topic,
                                   subscriptionId,
                                   leaseTimeInMilliseconds,
                                   query,
                                   schema);
    }

    // Получение клиентских сообщений и уведомлений
    // от службы Prospective Search:
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {

```

```
ProspectiveSearchService prospectiveSearch =
ProspectiveSearchServiceFactory.getProspectiveSearchService();
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
if (req.getParameter("message")!=null) {
    Entity UserComments = new Entity("UserComments");
    String messageIn=(String)req.getParameter("message");
    UserComments.setProperty("comments", messageIn);
    datastore.put(UserComments);
    String topic = "Comments";
    prospectiveSearch.match(UserComments, topic);
}

if (req.getParameter("document")!=null) {
    Entity matchedEntity =
ProspectiveSearchServiceFactory.getProspectiveSearchService().getDocument(req);
String message=(String)matchedEntity.getProperty("comments");
ChannelService channelService = ChannelServiceFactory.getChannelService();
ChannelMessage messageOut=new ChannelMessage("Comments", message);
channelService.sendMessage(messageOut);}}
```



## **ЧАСТЬ II**

# **Фреймворк Google Web Toolkit**

- Глава 14.** Начало работы с Google Web Toolkit
- Глава 15.** Компоненты графического интерфейса пользователя
- Глава 16.** Интернационализация GWT-приложения
- Глава 17.** Программный интерфейс JavaScript Native Interface
- Глава 18.** Оптимизация GWT-приложения
- Глава 19.** Поддержка истории Web-браузера
- Глава 20.** Фреймворк Activities and Places
- Глава 21.** Взаимодействие GWT-приложения с сервером



## ГЛАВА 14



# Начало работы с Google Web Toolkit

Google Web Toolkit (GWT) (<https://developers.google.com/web-toolkit/>) представляет собой фреймворк для разработки RIA (Rich Internet Application) Ajax-приложений на основе Java-кода.

С помощью GWT Web-приложение разрабатывается на языке Java, и его код на стадии разработки содержит определение GUI-интерфейса, обработку его событий и работу с данными. Затем Java-код GWT-приложения, содержащий определение GUI-интерфейса и обработку его событий, компилируется в JavaScript-код Web-страницы клиента, а на стороне сервера остается лишь Java-код Web-сервисов, отвечающих за работу с данными. При этом JavaScript-код Web-страницы содержит Ajax-клиентов Web-сервисов.

Таким образом, фреймворк GWT позволяет написать клиентскую часть Web-приложения на языке Java, а затем откомпилировать ее в оптимизированный JavaScript-код, корректно работающий в большинстве Web-браузерах, освобождая разработчика от задач оптимизации и кросс-браузерности приложения.

Так как Java-код клиента приложения в итоге компилируется в JavaScript-код, существуют особенности использования конструкций языка Java для написания клиентской части приложения. В частности, из-за того что JavaScript-интерпретатор является однопоточным, GWT-компилятор Java-кода в JavaScript-код игнорирует ключевое слово `synchronized` Java-кода. Также не поддерживаются динамическая загрузка классов и Java-финализация. Кроме того, для разработки клиентской части GWT-приложения может быть использован лишь ограниченный набор библиотек платформы Java (<https://developers.google.com/web-toolkit/doc/latest/RefJreEmulation>).

Фреймворк Google Web Toolkit (GWT) содержит:

- ◆ GWT SDK — набор, содержащий Java-библиотеки программного интерфейса GWT-фреймворка, GWT-компилятор Java-кода в JavaScript-код, локальный сервер разработки, позволяющий запускать и отлаживать Java-код приложения без его компиляции в JavaScript-код;
- ◆ Speed Tracer — расширение Web-браузера Chrome, позволяющее анализировать производительность GWT-приложения;

- ◆ Google Plugin for Eclipse (GPE) — плагин, обеспечивающий разработку GWT-приложений в среде Eclipse;
- ◆ GWT Designer — плагин, обеспечивающий визуальное редактирование GUI-интерфейса GWT-приложения в среде Eclipse.

## Установка плагинов фреймворка GWT

Для начала работы с GWT-фреймворком в среде Eclipse необходимо установить плагины GWT SDK, GPE и GWT Designer.

Откроем среду Eclipse IDE for Java EE Developers (<http://www.eclipse.org/downloads/>) и в меню **Help** выберем опцию **Install New Software**, нажмем кнопку **Add** поля **Work with** и введем адрес установки, указанный на странице <https://developers.google.com/eclipse/docs/download?hl=ru>, отметим флажки плагинов и нажмем кнопку **Next** (рис. 14.1).

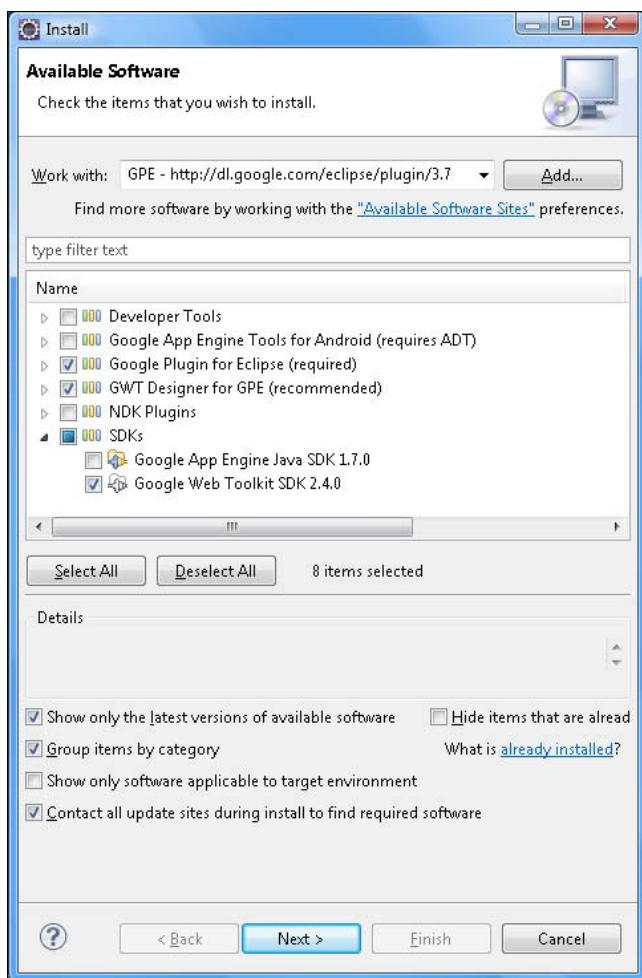


Рис. 14.1. Установка плагинов фреймворка GWT

## Создание проекта GWT-приложения

После установки плагинов фреймворка GWT и перезагрузки среды Eclipse в меню **File** выберем опцию **New | Other | Google | Web Application Project**, нажмем кнопку **Next**, введем имя проекта и имя пакета приложения, сбросим флажок **Use Google App Engine** и нажмем кнопку **Finish** (рис. 14.2). В результате будет создан проект GWT-приложения.

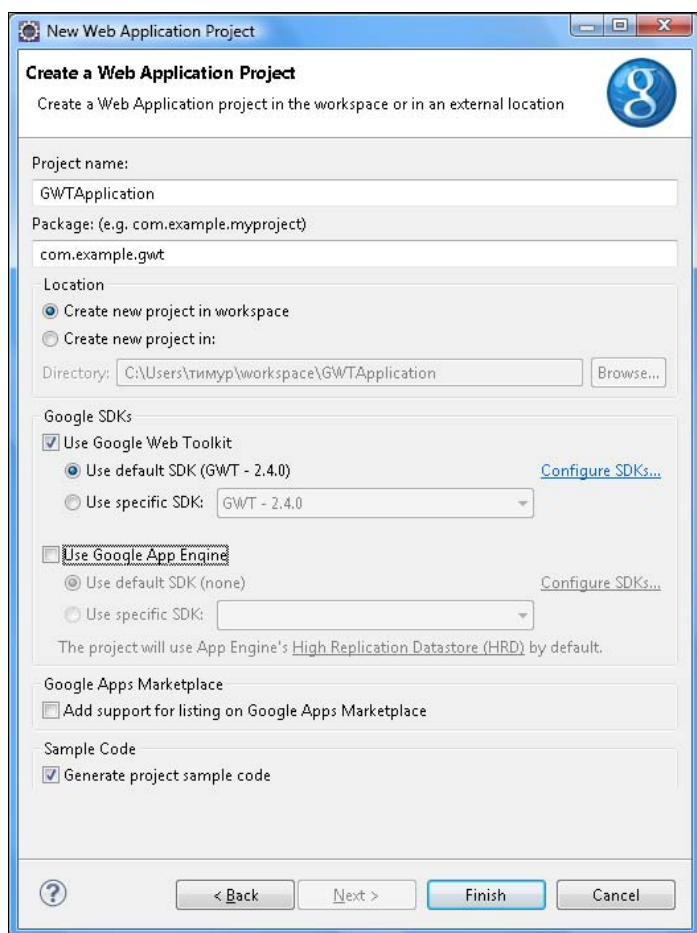


Рис. 14.2. Мастер создания проекта GWT-приложения

## Структура проекта GWT-приложения

Проект GWT-приложения состоит из трех основных каталогов — `src`, `war` и `test`.

Каталог `src` содержит исходный Java-код как клиентской, так и серверной части приложения, каталог `war` — набор файлов для развертывания в Web-сервере приложений, каталог `test` предназначен для JUnit-тестов приложения.

Каталог `src` сгенерированного средой Eclipse GWT-проекта включает в себя четыре пакета — пакет приложения, определенный при создании проекта, и пакеты с расширениями `.client`, `.server` и `.shared`.

Пакет приложения содержит конфигурационный XML-файл `.gwt.xml`, описывающий конфигурацию GWT-модуля в XML-формате с корневым тегом `<module>`.

## GWT-модули

Каждое GWT-приложение организовано в виде модулей, представляющих определенную функциональность. GWT-фреймворк предоставляет набор стандартных GWT-модулей, которые необходимо наследовать GWT-модулю приложения. GWT-модуль приложения как минимум должен наследовать модуль `com.google.gwt.user.User`, содержащий базовую функциональность GWT-фреймворка. Кроме того, GWT-модуль приложения может наследовать модуль темы, например `com.google.gwt.user.theme.clean.Clean`, обеспечивающий стили для GUI-компонентов приложения. Наследование объявляется в конфигурационном файле GWT-модуля приложения с помощью тега `<inherits>`.

Также конфигурационный файл GWT-модуля содержит объявление точки входа в GWT-модуль — класса, реализующего интерфейс `com.google.gwt.core.client.EntryPoint` и определяющего его метод `public void onModuleLoad()`, который вызывается при загрузке GWT-модуля. Точка входа объявляется с помощью тега `<entry-point>` конфигурационного файла GWT-модуля.

Тег `<source>` конфигурационного файла GWT-модуля объявляет пакеты, содержащие Java-код, предназначенный для компиляции в JavaScript-код для выполнения на стороне клиента.

Таким образом, GWT-модуль приложения представляет единицу функциональности клиентской части GWT-приложения, а его конфигурационный XML-файл `.gwt.xml` определяет конфигурацию компиляции Java-кода клиентской части в JavaScript-код.

Отдельно GWT-модуль приложения можно создать в среде Eclipse с помощью команды **New | Other | Google Web Toolkit | Module** контекстного меню окна **Project Explorer**.

После компиляции Java-кода клиентской части GWT-приложения в JavaScript-код GWT-модуль появится в каталоге `war` GWT-проекта в виде папки с именем модуля, содержащей файлы `.cache.html`, `.gwt.rpc`, `.nocache.js` и `hosted.html`, а также папку `gwt` CSS-стилей унаследованной модулем темы.

Загрузка GWT-модуля приложения на стороне клиента осуществляется с помощью HTML-страницы каталога `war` приложения, а именно с помощью ее тега `<script>`, атрибут `src` которого указывает для загрузки JavaScript-файл `.nocache.js` модуля.

После загрузки JavaScript-файл `.nocache.js` GWT-модуля определяет конфигурацию Web-браузера клиента, тем самым реализуя механизм отложенного связывания

Deferred Binding GWT-фреймворка, и создает iframe-фрейм, в который загружает подходящий выявленной конфигурации HTML-файл `.cache.html` модуля.

HTML-файл `.cache.html` GWT-модуля представляет собой HTML-обертку JavaScript-кода клиентской части GWT-приложения. JavaScript-код обернут в HTML-файл для повышения кросс-браузерности приложения. HTML-файлы `.cache.html` GWT-модуля именованы согласно MD5-суммам их содержимого, что гарантирует корректное их кэширование Web-браузером.

Файл `.gwt.rpc` GWT-модуля содержит определения Java-типов, реализующих интерфейс `java.io.Serializable` и предназначенных для сериализации при взаимодействии клиентской части GWT-приложения с его серверной частью.

Файл `hosted.html` загружается скриптом `.nocache.js` вместо файла `.cache.html` при вызове приложения с параметром `?gwt.codesvr=127.0.0.1:9997`. При этом файл `hosted.html` загружает в Web-браузер GWT-плагин Development Mode, обеспечивающий связь Web-браузера с сервером разработки GWT-фреймворка.

При проектировании GWT-приложения для решения проблемы, каким образом организовать разделение логики приложения для разных групп пользователей, необходимо определить — создавать ли несколько GWT-приложений или несколько отдельных GWT-модулей, или несколько GWT-модулей, объединенных наследованием в главный GWT-модуль.

Решить данную проблему с помощью нескольких точек входа `EntryPoint` в один GWT-модуль не получится, т. к. при загрузке GWT-модуля его точки входа будут вызываться последовательно, в том же самом порядке, в каком они объявлены в конфигурационном файле GWT-модуля.

Разделение проекта на несколько GWT-приложений возможно, если разные группы пользователей не используют общую HTTP-сессию и не объединены общими данными на уровне приложения. Проблема использования общей HTTP-сессии и данных может быть решена разделением приложения на разные GWT-модули, каждый со своей точкой входа `EntryPoint` и соответственно своим относительным URL-адресом, однако тогда нельзя будет использовать стандартные GWT-методы навигации между модулями, а задача разделения общего состояния клиента между модулями должна решаться на уровне сервера. Поэтому оптимальный путь организации проекта GWT-приложения для тесно связанных групп пользователей — это создание одного GWT-модуля с одной точкой входа `EntryPoint`. При этом задача модульности приложения может быть решена путем создания нескольких GWT-модулей без точек входа, объединенных наследованием в главный GWT-модуль с точкой входа.

В случае одного GWT-модуля с одной точкой входа `EntryPoint` задача разделения логики приложения для разных групп пользователей может решаться с помощью механизма отложенного связывания Deferred Binding GWT-фреймворка, когда в HTML-страницу загружается только тот JavaScript-код приложения, который соответствует определенным параметрам клиента. Проблема быстрой загрузки большого JavaScript-кода приложения в этом случае может быть решена посредством механизма разделения кода Code Splitting GWT-фреймворка.

В HTML-страницу приложения можно включить с помощью тегов `<script>` несколько GWT-модулей, каждый со своей точкой входа `EntryPoint`, однако при этом каждый GWT-модуль будет загружаться независимо, включая в себя избыточный код GWT-библиотек, и может в дальнейшем конфликтовать с другими GWT-модулями при обработке событий. Поэтому рекомендуется создавать один GWT-модуль с одной точкой входа `EntryPoint`.

GWT-приложение нужно рассматривать как одну HTML-страницу с одним URL-адресом, в которой события пользователя и самого приложения изменяют ее содержимое, используя механизм Ajax для взаимодействия с сервером. При этом многостраничность приложения может быть реализована путем организации логических страниц, представляющих определенное содержимое HTML-страницы приложения, в виде панелей с управлением их видимостью и использованием механизма `History` GWT-фреймворка для навигации по посещенным страницам, или может быть реализована на основе GWT MVP фреймворка `Activities and Places`.

## Конфигурационный XML-файл определения GWT-модуля

Структура файла `.gwt.xml` определяется его DTD-схемой `gwt-module.dtd` набора GWT SDK.

Корневым элементом XML-файла GWT-модуля служит тег `<module>`, дополнительный атрибут `rename-to` которого определяет имя GWT-модуля и соответственно имя папки GWT-модуля в каталоге `war` после компиляции Java-кода в JavaScript-код и имя JavaScript-файла `.nocache.js` GWT-модуля.

Если атрибут `rename-to` тега `<module>` отсутствует, тогда имя папки GWT-модуля в каталоге `war` после компиляции Java-кода в JavaScript-код и имя JavaScript-файла `.nocache.js` GWT-модуля соответствуют имени файла `.gwt.xml`.

Корневой тег `<module>` имеет дочерние теги `<inherits>`, `<source>`, `<public>`, `<super-source>`, `<entry-point>`, `<stylesheet>`, `<script>`, `<servlet>`, `<replace-with>`, `<generate-with>`, `<define-property>`, `<extend-property>`, `<set-property>`, `<set-property-fallback>`, `<clear-configuration-property>`, `<define-configuration-property>`, `<extend-configuration-property>`, `<set-configuration-property>`, `<property-provider>`, `<define-linker>`, `<add-linker>`, `<collapse-all-properties>`, `<collapse-property>`.

Тег `<inherits>` с помощью атрибута `name` указывает GWT-модуль, содержимое которого наследуется данным GWT-модулем. Как правило, GWT-модуль приложения наследует библиотечные GWT-модули и GWT-модуль, определяющий стили по умолчанию для GUI-компонентов приложения. Фреймворк GWT предоставляет библиотечные модули для наследования:

- ◆ `com.google.gwt.user.User` — предоставляет базовую функциональность GWT-фреймворка;
- ◆ `com.google.gwt.http.HTTP` — обеспечивает использование библиотеки `com.google.gwt.http.client` GWT API, позволяющей делать на стороне клиента HTTP-запросы и обрабатывать соответствующие ответы;

- ◆ `com.google.gwt.json.JSON` — обеспечивает применение библиотеки `com.google.gwt.json.client` GWT API, упрощающей использование JSON-формата данных при HTTP-запросах на стороне клиента;
- ◆ `com.google.gwt.junit.JUnit` — обеспечивает использование платформы тестирования JUnit;
- ◆ `com.google.gwt.xml.XML` — обеспечивает применение библиотеки `com.google.gwt.xml.client` GWT API, упрощающей использование XML-формата данных на стороне клиента.

GWT-модули стилей по умолчанию для GUI-компонентов приложения, предоставляемые GWT-фреймворком:

- ◆ `com.google.gwt.user.theme.clean.Clean` (рис. 14.3, а);
- ◆ `com.google.gwt.user.theme.standard.Standard` (рис. 14.3, б);
- ◆ `com.google.gwt.user.theme.chrome.Chrome` (рис. 14.3, в);
- ◆ `com.google.gwt.user.theme.dark.Dark` (рис. 14.3, г).

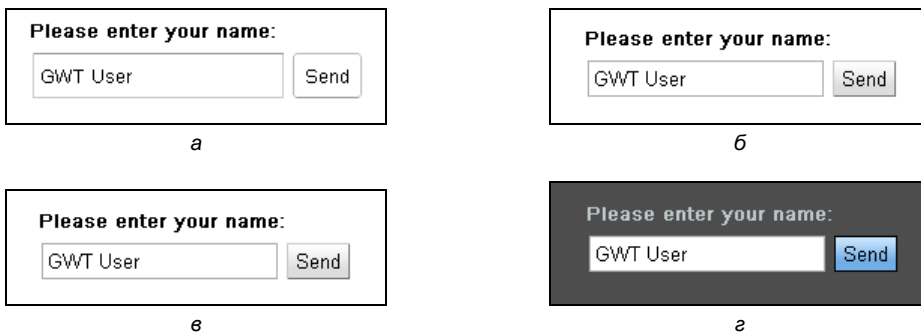


Рис. 14.3. Вид GUI-компонентов

Тег `<source>` с помощью атрибута `path` указывает расширение главного пакета приложения (по умолчанию `client`), содержащее Java-код, предназначенный для компиляции в JavaScript-код. Атрибуты `includes` и `excludes` указывают шаблоны включения и исключения ресурсов для компиляции; атрибут `defaultexcludes` (значение `yes` (по умолчанию) или `no`) указывает использование шаблонов по умолчанию для исключения ресурсов из компиляции (<http://ant.apache.org/manual/dirtasks.html#defaultexcludes>); атрибут `casesensitive` (значение `true` (по умолчанию) или `false`) указывает чувствительность к регистру при определении значений атрибутов тега.

Тег `<public>` с помощью атрибута `path` указывает расширение главного пакета приложения, содержащее ресурсы для их добавления при компиляции Java-to-JavaScript в конечную папку GWT-модуля каталога `war`. Тег `<public>` также может содержать атрибуты `includes`, `excludes`, `defaultexcludes` и `casesensitive`.

Тег `<super-source>` с помощью атрибута `path` указывает расширение главного пакета приложения, содержащее пользовательские Java-классы, используемые на стороне

клиента и предназначенные для компиляции Java-to-JavaScript. Тег `<super-source>` также может содержать атрибуты `includes`, `excludes`, `defaultexcludes` и `casesensitive`.

Тег `<entry-point>` с помощью атрибута `class` указывает точку входа в GWT-модуль — класс, реализующий интерфейс `com.google.gwt.core.client.EntryPoint` и определяющий его метод `public void onModuleLoad()`, который вызывается при загрузке GWT-модуля.

Тег `<stylesheet>` с помощью атрибута `src` указывает файл CSS-стилей для включения в GWT-модуль. Для включения дополнительного файла CSS-стилей с помощью тега `<stylesheet>` нужно в пакете приложения, содержащем файл `.gwt.xml`, создать папку `public` и поместить в нее файл CSS-стилей, имя которого затем указать в качестве значения атрибута `src` тега `<stylesheet>`. После компиляции Java-to-JavaScript файл CSS-стилей появится в папке GWT-модуля и будет автоматически загружаться в HTML-страницу приложения.

Тег `<script>` с помощью атрибута `src` указывает JavaScript-файл для включения в GWT-модуль. Для включения дополнительного JavaScript-файла с помощью тега `<script>` нужно в пакете приложения, содержащем файл `.gwt.xml`, создать папку `public` и поместить в нее JavaScript-файл, имя которого затем указать в качестве значения атрибута `src` тега `<script>`. После компиляции Java-to-JavaScript JavaScript-файл появится в папке GWT-модуля и будет автоматически загружаться в HTML-страницу приложения.

Тег `<servlet>` с помощью атрибутов `path` и `class` в режиме разработки (`hosted mode`) связывает с модулем сервлет, URL-путь которого также нужно указать в дескрипторе `WEB-INF/web.xml` каталога `war`.

Тег `<add-linker>` с помощью атрибута `name` указывает пользовательский компоновщик `com.google.gwt.core.ext.Linker` для GWT-компилятора. Атрибут `name` тега `<add-linker>` должен соответствовать атрибуту `name` тега `<define-linker>`, регистрирующему пользовательский компоновщик `com.google.gwt.core.ext.Linker` для GWT-компилятора.

Тег `<define-linker>` с помощью атрибутов `name` и `class` регистрирует пользовательский компоновщик `com.google.gwt.core.ext.Linker` для GWT-компилятора.

Тег `<define-property>` с помощью атрибутов `name` и `values` определяет свойство и его возможные значения, разделенные запятой, для дальнейшего использования в дочерних тегах условий `<when>` тегов `<replace-with-class>` и `<generate-with-class>` механизма отложенного связывания `Deferred Binding`.

Тег `<set-property>` с помощью атрибутов `name` и `value` фиксирует значение свойства, ранее определенное тегом `<define-property>`.

Тег `<property-provider>` альтернативно статическому тегу `<set-property>` определяет с помощью вложенного тега `<![CDATA[...]]>` фрагмент JavaScript-кода, который динамически возвращает значение для свойства, ранее определенного тегом `<define-property>`. Атрибут `name` тега `<property-provider>` указывает имя свойства, а дополнительный атрибут `generator` может указывать генератор `com.google.gwt.core.ext.Generator` исходного кода.



Тег `<replace-with>` с помощью атрибута `class` указывает класс, которым нужно заменить при выполнении определенных условий. Условия замены определяются с помощью дочерних тегов:

- ◆ `when-type-is` — с помощью атрибута `class` указывает единственный Java-тип замены;
- ◆ `when-property-is` — с помощью атрибутов `name` и `value` указывает значение свойства, при котором производится замена;
- ◆ `when-type-assignable` — с помощью атрибута `class` указывает Java-тип, к которому приводятся типы замены;
- ◆ `all`, `any`, `none` — комбинируют теги `<when>` в выражение.

Тег `<generate-with>` с помощью атрибута `class` указывает класс генератора `com.google.gwt.core.ext.Generator` исходного кода, который должен использоваться GWT-компилятором при выполнении определенных условий. Условия определяются с помощью дочерних тегов `<when-type-is>`, `<when-property-is>`, `<when-type-assignable>` и `<all>`, `<any>`, `<none>`.

Тег `<define-configuration-property>` с помощью атрибутов `name` и `is-multi-valued` определяет конфигурационное свойство для генератора `Generator` и компоновщика `Linker`.

Тег `<set-property-fallback>` с помощью атрибутов `name` и `value` устанавливает резервное значение свойства.

Тег `<set-configuration-property>` с помощью атрибутов `name` и `value` фиксирует значение конфигурационного свойства для генератора `Generator` и компоновщика `Linker`.

Тег `<extend-property>` с помощью атрибутов `name` и `values` расширяет значения ранее определенного свойства, атрибут `fallback-value` указывает резервное значение свойства.

Тег `<collapse-property>` с помощью атрибутов `name` и `values` определяет эквивалентные значения свойства.

Тег `<collapse-all-properties>` с помощью атрибута `value="true"` устанавливает все значения свойств эквивалентными.

Тег `<extend-configuration-property>` с помощью атрибутов `name` и `value` расширяет значение ранее определенного конфигурационного свойства.

Тег `<clear-configuration-property>` удаляет ранее определенные значения конфигурационного свойства, имя которого указано с помощью атрибута `name`.

## Модель программирования фреймворка GWT

Каталог `src` сгенерированного средой Eclipse GWT-проекта, помимо главного пакета приложения, содержит пакеты с расширениями `.client`, `.server` и `.shared`.

Пакеты с расширениями `.client` и `.shared` определены в конфигурационном файле `.gwt.xml` тегом `<source>` для компиляции в JavaScript-код. Следовательно, пакет с расширением `.client` проекта GWT-приложения содержит код, выполняемый на

стороне клиента. Пакет с расширением `.shared` проекта GWT-приложения содержит код, используемый как на стороне клиента, так и на стороне сервера. Сгенерированный средой Eclipse код пакета `.shared` состоит из класса, обеспечивающего проверку введенных пользователем данных.

Сгенерированный средой Eclipse код пакета `.client` состоит из главного класса приложения и интерфейсов — интерфейса Web-сервиса, поставляющего данные клиенту, и интерфейса, обеспечивающего асинхронный вызов Web-сервиса на стороне клиента.

Главный класс GWT-приложения, обеспечивающий точку входа в GWT-модуль, объявляется в конфигурационном файле `.gwt.xml` тегом `<entry-point>` и реализует интерфейс `com.google.gwt.core.client.EntryPoint` с определением его метода `onModuleLoad()`. Метод `onModuleLoad()` автоматически вызывается при загрузке GWT-модуля, и поэтому именно в нем инициализируется GUI-интерфейс приложения.

Отдельно главный класс GWT-приложения можно создать в среде Eclipse с помощью команды **New | Other | Google Web Toolkit | Entry Point Class** контекстного меню окна **Project Explorer**.

GUI-интерфейс GWT-приложения конструируется из объектов классов, базовым классом которых является класс `com.google.gwt.user.client.ui.Widget`, при этом их компоновка осуществляется с помощью панелей, базовым классом которых является класс `com.google.gwt.user.client.ui.Panel`. Внешний вид по умолчанию Widget-компонентов задается GWT-модулем темы, наследуемым GWT-модулем приложения, и может переопределяться с помощью CSS-стилей.

Для того чтобы определить CSS-стиль элемента HTML-страницы, необходимо установить для HTML-элемента его класс (атрибут `class` элемента) или идентификатор (атрибут `id` элемента), используемые в качестве селекторов CSS-стилей. По умолчанию каждому Widget-компоненту присваивается класс с именем `gwt-[имя Java-класса]`, например для кнопки класс будет `.gwt-Button`.

Для того чтобы присвоить идентификатор Widget-компоненту, нужно создать в коде HTML-страницы приложения блок `<div id="id-widget"/>` с идентификатором и связать его с Widget-компонентом:

```
com.google.gwt.user.client.DOM.setElementAttribute(new someWidget().getElement(), "id",  
" id-widget");
```

Далее CSS-стиль Widget-компонента задается с помощью селектора:

```
gwt-[имя Java-класса]{ ... }
```

или

```
#id-widget { ... }
```

Можно переопределить имя CSS-класса по умолчанию для Widget-компонента с помощью метода `setStyleName(java.lang.String style)` суперкласса `com.google.gwt.user.client.ui.UIObject` или добавить CSS-класс, используя метод `addStyleName(java.lang.String style)` суперкласса `UIObject`.

Кроме того, можно и программным способом "на лету" изменять стиль Widget-компонента:

```
myWidget.getElement().setAttribute("style", "background-color:red;");
```

или

```
myWidget.getElement().getStyle().setProperty("backgroundColor", "red");
```

Как правило, CSS-стили определяются в отдельных файлах, которые могут быть связаны с GWT-модулем приложения несколькими способами.

- ◆ Файл CSS-стилей может быть включен в HTML-страницу приложения с помощью атрибута href тега <link>:

```
<link rel="stylesheet" href="custom.css" type="text/css"/>
```

- ◆ Связать файл CSS-стилей с GWT-модулем можно в его конфигурационном файле .gwt.xml, используя тег <stylesheet>.

- ◆ С помощью интерфейса `com.google.gwt.resources.client.ClientBundle`, гарантирующего кэширование ресурсов приложения. Интерфейс GWT-приложения, расширяющий интерфейс `ClientBundle`, можно создать в среде Eclipse с помощью команды **New | Other | Google Web Toolkit | ClientBundle** контекстного меню окна **Project Explorer**, добавив, например, в созданный интерфейс код:

```
import com.google.gwt.core.client.GWT;
import com.google.gwt.resources.client.*;

public interface ApplicationResources extends ClientBundle {
    public static final ApplicationResources INSTANCE =
        GWT.create(ApplicationResources.class);
    @Source("custom.css")
    public CssResource css();}

В данном случае файл CSS-стилей custom.css размещен вместе с интерфейсом ApplicationResources и имеет, например, следующий код:
```

```
@external .gwt-Button;
.gwt-Button {
    background: blue;
}
```

```
Здесь класс .gwt-Button указан как внешний для предотвращения обфускации1 его имени при компиляции Java-to-JavaScript.
```

Для работы интерфейса `ApplicationResources` в конфигурационном файле .gwt.xml приложения необходимо добавить наследование:

```
<inherits name="com.google.gwt.resources.Resources" />
```

А в методе `onModuleLoad()` главного класса приложения — включение файла CSS-стилей:

```
ApplicationResources.INSTANCE.css().ensureInjected();
```

---

<sup>1</sup> Обфускация — это запутывание текста вставками, которые не отображаются на экране монитора.

- ◆ Связать файл CSS-стилей с GWT-модулем можно также с помощью тега `<ui:style>` шаблона `UiBinder`. Фреймворк `UiBinder` GWT позволяет декларативным способом с помощью XML-разметки компоновать `Widget`-компоненты и `HTML`-компоненты в один составной `Widget`-компонент. `UiBinder`-компонент можно создать в среде Eclipse посредством команды **New | Other | Google Web Toolkit | UiBinder** контекстного меню окна **Project Explorer**. Затем в XML-файле `UiBinder`-компонента определить тег `<ui:style>`, например:

```
<ui:style src="custom.css" />
```

Здесь файл CSS-стилей `custom.css` размещен вместе с файлами `UiBinder`-компонента и имеет, например, следующий код:

```
@external .gwt-Button;
.gwt-Button { background: red; }
```

где класс `.gwt-Button` указан как внешний для предотвращения обфускации его имени при компиляции `Java-to-JavaScript`.

В `HTML`-странице приложения необходимо предусмотреть контейнер для `UiBinder`-компонента:

```
<td id="customWidget"></td>
```

А также добавить код инициализации `UiBinder`-компонента в метод `onModuleLoad()` `EntryPoint`-точки входа в GWT-модуль:

```
CustomWidget custom=new CustomWidget();
RootPanel.get("customWidget").add(custom);
```

GUI-интерфейс GWT-приложения имеет основную корневую панель компоновки, представленную классом `com.google.gwt.user.client.ui.RootPanel`, экземпляр которого получается статическим методом `RootPanel.get()`. Такая основная корневая панель связана с тегом `<body>` `Web`-страницы GWT-приложения. Основная корневая `RootPanel`-панель может содержать другие корневые `RootPanel`-панели, связанные с элементами `Web`-страницы GWT-приложения и экземпляры которых получают с помощью статического метода `RootPanel.get(String id)`, где `id` — идентификатор `HTML`-элемента. `Widget`-компоненты и `Panel`-панели добавляются в `RootPanel`-панели с помощью метода `add()` класса `Panel`:

```
RootPanel panel = RootPanel.get("nameWidgetContainer");
if (panel) panel.add(new someWidget());
```

Визуальное редактирование GUI-интерфейса GWT-приложения обеспечивает инструмент `GWT Designer` GWT-фреймворка.

Для того чтобы воспользоваться инструментом `GWT Designer`, в окне **Project Explorer** среды Eclipse щелчком правой кнопкой мыши на узле главного класса GWT-приложения пакета `.client` и в контекстном меню выберем опцию **Open With | GWT Designer** — в результате главный класс откроется в графическом визуальном GWT-редакторе, вкладка **Source** которого отображает исходный `Java`-код класса, а вкладка **Design** позволяет визуальное редактирование GUI-интерфейса GWT-приложения (рис. 14.4).

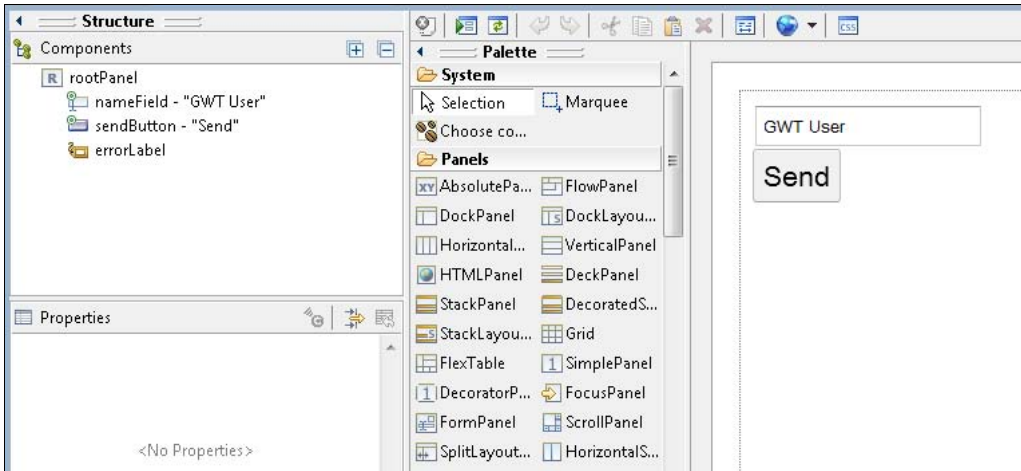


Рис. 14.4. Редактор GWT Designer GUI-интерфейса GWT-приложения

Palette-палитра GWT-редактора обеспечивает наполнение GUI-интерфейса GWT-компонентами, а окно **Properties** — редактирование их свойств.

Инструмент GWT Designer также обеспечивает визуальное редактирование XML-файла UiBinder-компонента.

Для создания клиент-серверного взаимодействия обмена данными фреймворк GWT предлагает три фреймворка — GWT RPC, HTTP client и Request Factory.

Сгенерированный код при создании проекта GWT-приложения в среде Eclipse основан на фреймворке GWT RPC и содержится в пакетах с расширениями .client и .server каталога src проекта.

В сгенерированном коде главный класс GWT-приложения обеспечивает взаимодействие с Web-сервисом сервера с помощью Proxu-объекта, создаваемого с помощью статического метода `create()` класса `com.google.gwt.core.client.GWT`:

```
private final GreetingServiceAsync greetingService = GWT.create(GreetingService.class);
```

Интерфейс Web-сервиса `GreetingService`, определенный на стороне клиента, расширяет интерфейс `com.google.gwt.user.client.rpc.RemoteService` и объявляет методы Web-сервиса, вызываемые клиентом. Аннотация `com.google.gwt.user.client.rpc.RemoteServiceRelativePath` используется для указания относительного пути сервлета на стороне сервера, реализующего интерфейс Web-сервиса.

Интерфейс `GreetingServiceAsync`, определенный на стороне клиента и обеспечивающий асинхронный вызов Web-сервиса, создан на основе интерфейса Web-сервиса `GreetingService` и дополняет методы Web-сервиса аргументом — объектом `com.google.gwt.user.client.rpc.AsyncCallback`. Интерфейс `AsyncCallback` предоставляет методы `onSuccess()` и `onFailure()`, автоматически вызываемые в случае успешного завершения асинхронного вызова Web-сервиса и в случае возникновения ошибки соответственно. Результат вызова Web-сервиса содержится как аргумент метода `onSuccess()`.

Пакет с расширением `.server` проекта GWT-приложения содержит код, выполняемый на стороне сервера. Сгенерированный средой Eclipse код папки `.server` состоит из сервлета, расширяющего класс `com.google.gwt.user.server.rpc.RemoteServiceServlet` и реализующего интерфейс Web-сервиса `GreetingService`. Класс `RemoteServiceServlet` обеспечивает десериализацию входящих клиентских запросов и сериализацию ответов клиенту.

Каталог `war` проекта GWT-приложения содержит папку `WEB-INF`, а также HTML-страницу GWT-приложения и файлы CSS-стилей, связанные с HTML-страницей с помощью HTML-тега `<link>`.

HTML-страница GWT-приложения содержит элементы, идентификаторы которых связаны с `RootPanel`-панелями.

Папка `WEB-INF` содержит дескриптор `web.xml` развертывания GWT-приложения на стороне сервера, а также папку `lib` с библиотекой `gwt-servlet.jar`, обеспечивающей классы механизма GWT RPC удаленного вызова процедур.

Дескриптор `web.xml` содержит объявления сервлета, представляющего Web-сервис GWT-приложения, относительного пути сервлета и главной Web-страницы GWT-приложения.

## Запуск GWT-приложения в режиме разработки

Режим разработки фреймворка GWT (`Development Mode` или `Hosted Mode`) дает возможность запускать GWT-приложение без компиляции Java-кода его клиентской части в JavaScript-код. Для этого набор GWT SDK предусматривает встроенный Web-сервер `Jetty`, а для Web-браузера предусмотрен GWT-плагин `Developer Plugin`, обеспечивающий связь между исполняемым Java-байт-кодом в сервере разработки и JavaScript-кодом Web-браузера.

Для запуска GWT-приложения в отладочном режиме без компиляции Java-кода в JavaScript-код в окне **Project Explorer** среды Eclipse щелчком правой кнопкой мыши на узле GWT-проекта и в контекстном меню выберем опцию **Run As | Web Application**. В результате откроется окно **Development Mode** с URL-адресом загрузки GWT-приложения (рис. 14.5).

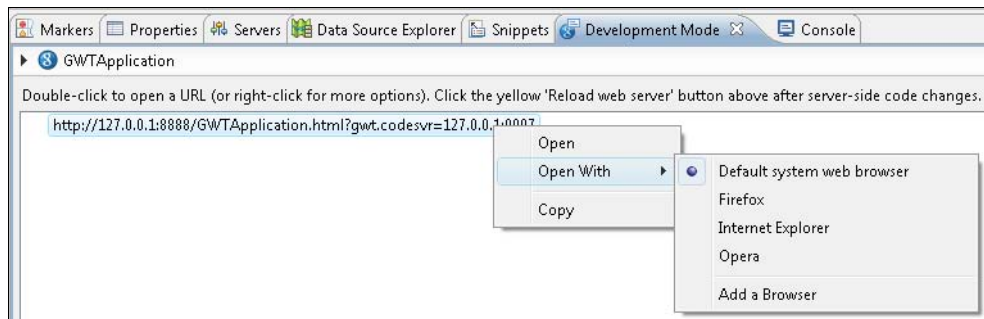


Рис. 14.5. Запуск GWT-приложения в режиме `Development Mode`

В окне **Development Mode** щелкнем правой кнопкой мыши на URL-адресе загрузки GWT-приложения и выберем Web-браузер. В первый раз Web-браузер отобразит запрос на загрузку GWT-плагина, который обеспечивает связь между Java-байт-кодом в сервере разработки и JavaScript-кодом Web-браузера (рис. 14.6).



Рис. 14.6. Установка GWT-плагина Developer Plugin

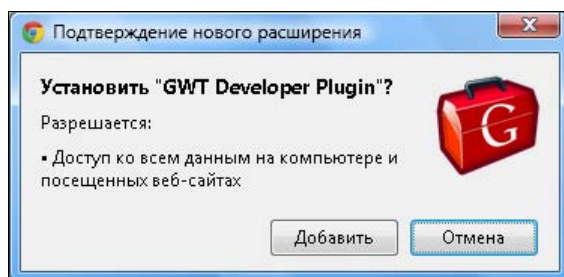


Рис. 14.7. Диалоговое окно разрешения на установку GWT-плагина Web-браузера Chrome

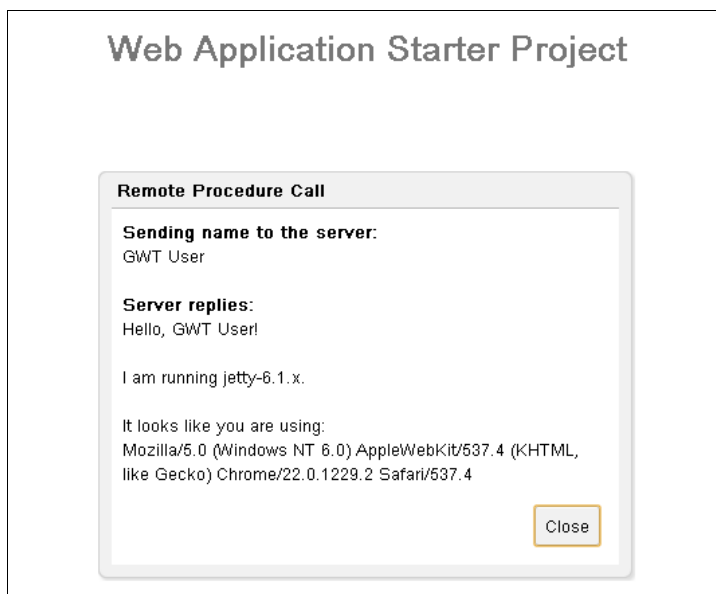


Рис. 14.8. Web-страница GWT-приложения

Для Web-браузера Chrome скачаем плагин, откроем настройки Web-браузера **Инструменты | Расширения** и перетащим мышью файл плагина на страницу расширений — в результате появится диалоговое окно разрешения на установку плагина (рис. 14.7).

После установки GWT-плагина Web-браузер отобразит Web-страницу GWT-приложения (рис. 14.8).

## Запуск GWT-приложения как Web-приложения

Для запуска GWT-приложения как Web-приложения необходимо откомпилировать Java-код его клиентской части в JavaScript-код и развернуть его в сервере приложений.

Для компиляции Java-кода GWT-приложения в JavaScript-код в окне **Project Explorer** среды Eclipse нажмем правой кнопкой мыши на узле GWT-проекта и в контекстном меню выберем команду **Google | GWT Compile**. В результате откроется диалоговое окно мастера GWT-компиляции (рис. 14.9).

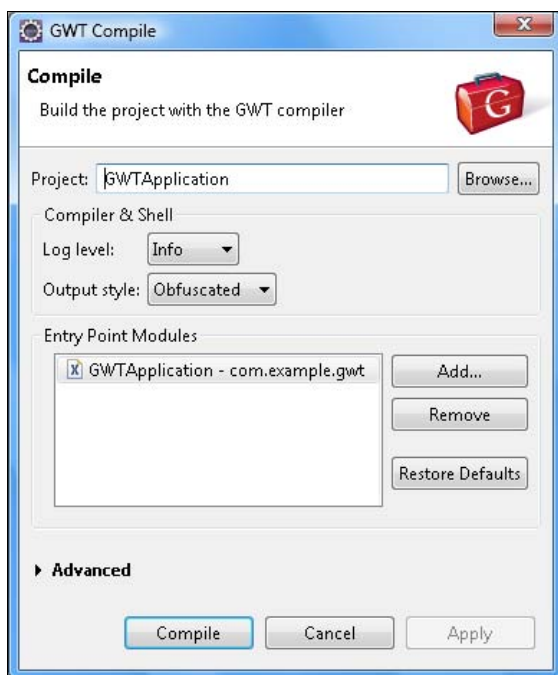


Рис. 14.9. Мастер компиляции Java-кода GWT-приложения в JavaScript-код

Раскрывающийся список **Output style** мастера GWT-компиляции предлагает выбрать способ генерации JavaScript-кода:

- ◆ **Obfuscated** — нечитаемый, сжатый код;
- ◆ **Pretty** — JavaScript-код с осмысленными именами;
- ◆ **Detailed** — JavaScript-код с полными именами.



После нажатия кнопки **Compile** мастера GWT-компиляции будет запущен процесс компиляции Java-to-JavaScript, отображаемый в окне **Console** среды Eclipse. По завершению процесса компиляции в каталоге war GWT-проекта появится папка с именем GWT-модуля, содержащая HTML- и JavaScript-код, а также сопутствующие ресурсы стороны клиента.

Для развертывания GWT-приложения в сервере Tomcat скопируем каталог war GWT-проекта в папку webapps каталога сервера Tomcat, запустим сервер с помощью инструмента startup.bat папки bin и в адресной строке Web-браузера введем адрес **http://127.0.0.1:8080/war/** — в результате откроется Web-страница GWT-приложения.

Самый простой способ упаковать Eclipse-проект GWT-приложения в архивный WAR-файл Web-приложения — перевести проект в статус динамического Web-модуля. Для этого в окне **Project Explorer** среды Eclipse нажмем правой кнопкой мыши на узле GWT-проекта и в контекстном меню выберем команду **Properties**. В диалоговом окне **Properties** в разделе **Project Facets** отметим флажок **Dynamic Web Module** и нажмем кнопку **OK** (рис. 14.10).

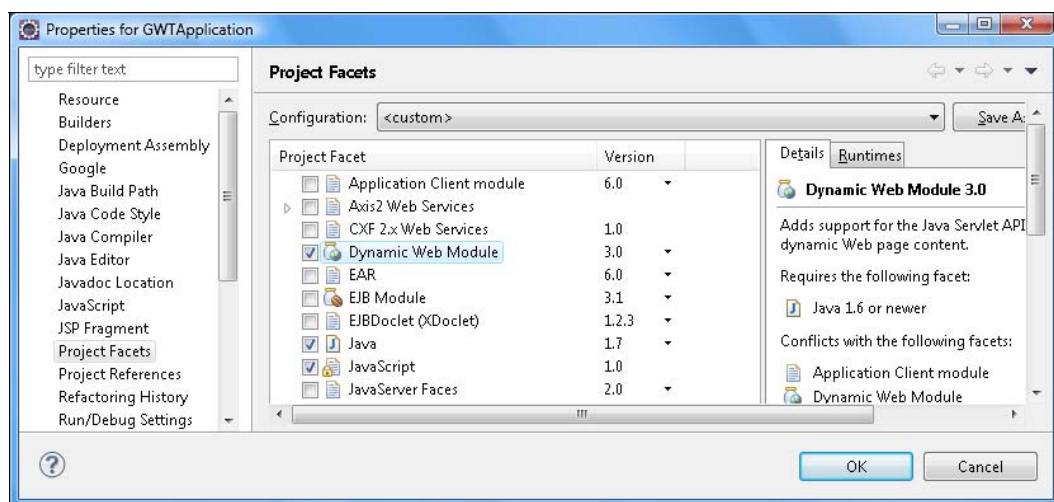


Рис. 14.10. Добавление свойств динамического Web-модуля GWT-проекту

В результате в GWT-проект добавится каталог WebContent, в который нужно скопировать содержимое каталога war GWT-проекта.

В окне **Project Explorer** среды Eclipse нажмем правой кнопкой мыши на узле GWT-проекта и в контекстном меню выберем команду **Export | WAR file**. В диалоговом окне мастера кнопкой **Browse** выберем каталог для размещения готового WAR-файла и нажмем кнопку **Finish** (рис. 14.11).

В результате средой Eclipse будет собран WAR-файл GWT-приложения, готовый для развертывания в сервере приложений.

Инструмент Speed Tracer фреймворка GWT позволяет анализировать производительность GWT-приложения.

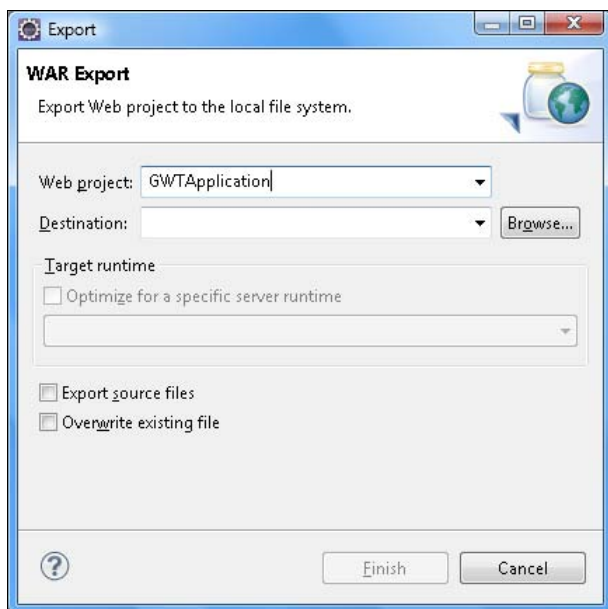


Рис. 14.11. Экспорт GWT-проекта в WAR-файл

Инструмент Speed Tracer представляет собой расширение Web-браузера Chrome, поэтому, для того чтобы начать использовать инструмент Speed Tracer, необходимо установить версию не меньше 18 Web-браузера Chrome (<http://dev.chromium.org/getting-involved/dev-channel#TOC-Subscribing-to-a-channel>).

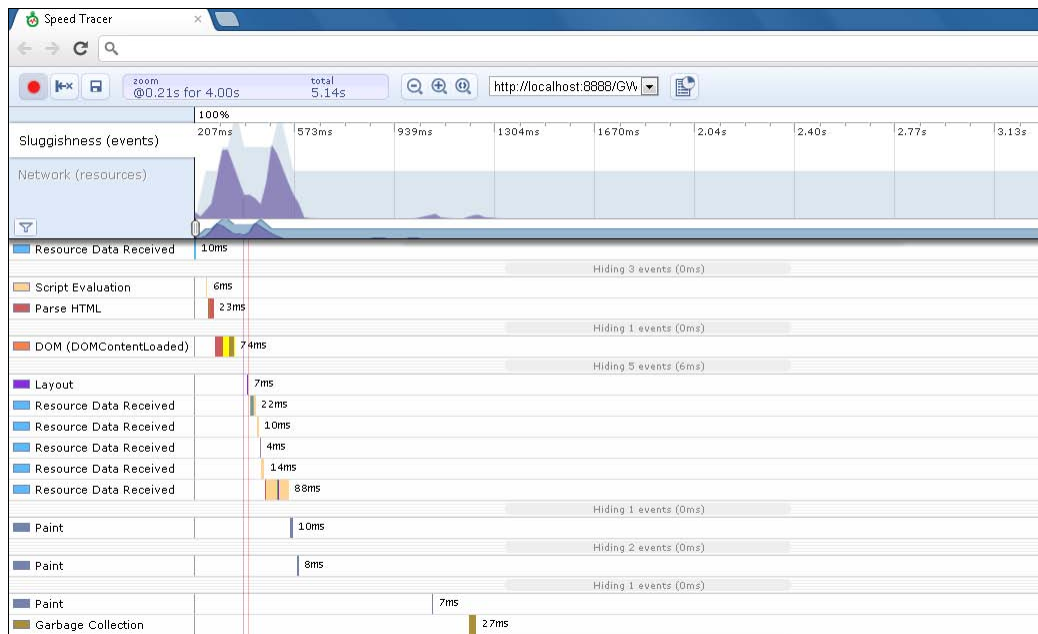


Рис. 14.12. Анализ работы GWT-приложения инструментом Speed Tracer

После установки Web-браузера Chrome откроем его и на странице **<https://developers.google.com/web-toolkit/speedtracer/get-started>** нажмем ссылку **Install Speed Tracer**.

После установки расширения Speed Tracer в Web-браузере Chrome в меню **Настройки** выберем опцию **Инструменты | Расширения** и для расширения Speed Tracer отметим флажок **Разрешить доступ к URL файла** для разрешения доступа к GWT-приложению.

В окне **Project Explorer** среды Eclipse нажмем правой кнопкой мыши на узле GWT-проекта и в контекстном меню выберем опцию **Google | Profile Using Speed Tracer**. В появившемся диалоговом окне нажмем кнопку **Profile**. В результате в Web-браузере Chrome откроется Web-страница GWT-приложения и Web-страница инструмента Speed Tracer с анализом работы GWT-приложения (рис. 14.12).

## ГЛАВА 15



# Компоненты графического интерфейса пользователя

Компоненты графического интерфейса пользователя GWT-приложения создаются в Java-коде, который затем компилируется в JavaScript-код, поэтому в отличие от графических Java GUI-библиотек, таких как AWT/Swing, SWT/JFace и JavaFX, отображение GWT-компонентов основано не на пиксельной графике, а на динамически создаваемых HTML-элементах.

GUI-компоненты GWT-приложения могут создаваться в Java-коде с помощью низкоуровневой GWT-библиотеки `com.google.gwt.dom.client` (DOM-библиотека) или высокоуровневой GWT-библиотеки `com.google.gwt.user.client.ui` (Widget-библиотека).

Компоненты DOM-библиотеки и Widget-библиотеки имеют общие корни в виде иерархии:

```
java.lang.Object
├── extended by com.google.gwt.core.client.JavaScriptObject
│   └── extended by com.google.gwt.core.client.JavaScriptObject
│       └── extended by com.google.gwt.dom.client.Element
```

Классы компонентов DOM-библиотеки напрямую расширяют класс `Element`, а компоненты Widget-библиотеки основаны на встроенном объекте `com.google.gwt.user.client.Element`, который может быть получен методом `getElement()` суперкласса Widget-библиотеки `UIObject`. При этом класс `com.google.gwt.user.client.Element` расширяет класс `com.google.gwt.dom.client.Element`.

Низлежащие `JavaScriptObject`-объекты GWT GUI-компонентов максимально обеспечивают кросс-браузерность там, где это возможно, основываясь на HTML-элементах, одинаково воспринимаемых различными Web-браузерами.

Как правило, GUI-интерфейс GWT-приложения строится из Widget-компонентов библиотеки `com.google.gwt.user.client.ui`, базовым классом которых является класс `com.google.gwt.user.client.ui.Widget`, при этом Widget-компоненты komponуются с помощью панелей, базовым классом которых является класс `com.google.gwt.user.client.ui.Panel`. Eclipse-плагин GWT Designer также обеспечивает визуальное создание GUI-интерфейса GWT-приложения на основе Widget-библиотеки.

## Кнопка *Button*

GUI-компонент `Button` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента строку надписи кнопки и обработчик событий щелчка кнопки.

Класс `Button` расширяет классы `ButtonBase`, `FocusWidget`, `Widget` и `UIObject`.

По умолчанию `Button`-компонент имеет CSS-класс `.gwt-Button`.

Надпись кнопки может устанавливаться методом `setText(java.lang.String text)`, принимающим в качестве аргумента простую текстовую строку, или методом `setHTML([java.lang.String html] или [SafeHtml html])`, принимающим в качестве аргумента текстовую строку, которая может включать в себя HTML-теги разметки текста, или объект `SafeHtml`, обеспечивающий безопасный HTML-код.

### Отличие метода *setText()* от метода *setHTML()*

Отличие использования метода `setText()` от метода `setHTML()` заключается в том, что метод `setText()` не воспринимает включенные в строку аргумента HTML-теги разметки текста, такие как `<i>`, `<em>`, `<b>` и др., а метод `setHTML()` следует им.

Например, кнопка, созданная с помощью следующего кода:

```
final Button sendButton = new Button();
sendButton.setHTML("<i><b>hello</b></i>");
```

будет иметь вид, как на рис. 15.1, *а*.

А кнопка, созданная кодом:

```
final Button sendButton = new Button();
sendButton.setText("<i><b>hello</b></i>");
```

будет иметь вид, как на рис. 15.2, *б*.

*а**б*

Рис. 15.1. Разные виды кнопок

Вместо метода `setHTML(java.lang.String html)` рекомендуется применять метод `setHTML(SafeHtml html)`, оперирующий объектом `com.google.gwt.safehtml.shared.SafeHtml`, который обеспечивает предотвращение атаки Cross-Site-Scripting (XSS), проводя санитарную обработку обернутой текстовой строки с HTML-тегами и гарантируя невыполнение включенного в нее JavaScript-кода в Web-браузере.

Метод `setHTML(java.lang.String html)` в конечном итоге после компиляции в JavaScript-код основывается на использовании свойства HTML-элементов `innerHTML`, которое при определении значения не вызывает немедленное выполнение тега `<script>`.

Поэтому следующий код

```
final Button sendButton = new Button();
String html="<script>alert()</script>";
sendButton.setHTML(html);
```

не вызовет появление Alert-сообщения. Однако следующий код

```
final Button sendButton = new Button();
String html="<a href='javascript:alert();'>click me!</a>";
sendButton.setHTML(html);
```

приведет к появлению ссылки в кнопке, при нажатии на которой будет выполнен встроенный JavaScript-код с отображением Alert-сообщения.

Использование объекта `SafeHtml` предотвращает такого рода действия.

Объект `SafeHtml` может быть получен с помощью класса `com.google.gwt.safehtml.shared.SafeHtmlBuilder`:

```
SafeHtmlBuilder builder = new SafeHtmlBuilder();
SafeHtml htmlSafe = builder.appendHtmlConstant(html).toSafeHtml();
```

Или с помощью класса `com.google.gwt.safehtml.shared.SafeHtmlUtils`:

```
SafeHtml htmlSafe = SafeHtmlUtils.fromString(html);
```

или

```
SafeHtml htmlSafe = SafeHtmlUtils.fromSafeConstant(html);
```

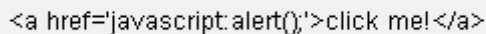
Или с помощью класса `com.google.gwt.safehtml.shared.SimpleHtmlSanitizer`:

```
SafeHtml htmlSafe = SimpleHtmlSanitizer.sanitizeHtml(html);
```

С использованием объекта `SafeHtml` код:

```
final Button sendButton = new Button();
String html="<a href='javascript:alert();'>click me!</a>";
SafeHtml htmlSafe = SimpleHtmlSanitizer.sanitizeHtml(html);
sendButton.setHTML(htmlSafe);
```

отобразит кнопку в виде, представленном на рис. 15.2.



```
<a href='javascript:alert();'>click me!</a>
```

Рис. 15.2. Кнопка, созданная с использованием объекта `SafeHtml`

## Обработчики событий кнопки

Класс `Button` наследует от класса `FocusWidget` следующие методы, позволяющие определить для кнопки различные обработчики событий.

Обработчики щелчка мышью:

- ◆ `addClickHandler(ClickHandler handler)` — присоединяет к кнопке обработчик событий щелчка кнопки:

```
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.Window;
sendButton.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        Window.alert("Click");
    }
});
```

- ◆ `addDoubleClickHandler(DoubleClickHandler handler)` — присоединяет к кнопке обработчик событий двойного щелчка кнопки:

```
import com.google.gwt.event.dom.client.DoubleClickEvent;
import com.google.gwt.event.dom.client.DoubleClickHandler;
import com.google.gwt.user.client.Window;
sendButton.addDoubleClickHandler( new DoubleClickHandler () {
    public void onDoubleClick(DoubleClickEvent event) {
        Window.alert("Double Click");
    }
});
```

Метод `click()` класса `Button` позволяет программным способом сгенерировать событие щелчка кнопки.

Обработчики перетаскивания мышью:

- ◆ `addDragEndHandler(DragEndHandler handler)` — присоединяет к кнопке обработчик окончания перетаскивания;
- ◆ `addDragEnterHandler(DragEnterHandler handler)` — присоединяет к кнопке обработчик события входа перетаскивания другого компонента в область данного компонента;
- ◆ `addDragHandler(DragHandler handler)` — присоединяет к кнопке обработчик события перетаскивания;
- ◆ `addDragLeaveHandler(DragLeaveHandler handler)` — присоединяет к кнопке обработчик события выхода перетаскивания другого компонента из области данного компонента;
- ◆ `addDragOverHandler(DragOverHandler handler)` — присоединяет к кнопке обработчик события прохождения перетаскивания другого компонента через область данного компонента;
- ◆ `addDragStartHandler(DragStartHandler handler)` — присоединяет к кнопке обработчик начала перетаскивания;
- ◆ `addDropHandler(DropHandler handler)` — присоединяет к кнопке обработчик события сбрасывания в результате перетаскивания другого компонента в область данного компонента.

Следующий код демонстрирует работу обработчиков перетаскивания компонентов мышью:

```
package com.example.gwt.client;
import com.google.gwt.core.client.*;
```

```
import com.google.gwt.dom.client.Style.*;
import com.google.gwt.event.dom.client.*;
import com.google.gwt.user.client.ui.*;
import com.google.gwt.user.client.*;

public class GWTApplication implements EntryPoint {
public void onModuleLoad() {
    final Button button1 = new Button("Button1");
    final RootPanel panel1=RootPanel.get("button1Container");
    panel1.add(button1);
    button1.getElement().setDraggable(Element.DRAGGABLE_TRUE);
    panel1.getElement().getStyle().setPosition(Position.ABSOLUTE);

    final Button button2 = new Button("Button2");
    final RootPanel panel2=RootPanel.get("button2Container");
    panel2.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel2.add(button2,100,0);
    button2.getElement().setDraggable(Element.DRAGGABLE_TRUE);

    button1.addDragEndHandler(new DragEndHandler () {
        public void onDragEnd(DragEndEvent event) {
            double x= event.getNativeEvent().getClientX();
            double y= event.getNativeEvent().getClientY();
            panel1.getElement().getStyle().setTop(y, com.google.gwt.dom.client.Style.Unit.PX);
            panel1.getElement().getStyle().setLeft(x,
com.google.gwt.dom.client.Style.Unit.PX);
            Window.alert("DragEndEvent");
        }
    });

    button1.addDragEnterHandler( new DragEnterHandler () {
        public void onDragEnter(DragEnterEvent event) {
            //Window.alert("DragEnterEvent");
        }
    });

    button1.addDragHandler( new DragHandler () {
        public void onDrag(DragEvent event) {
            //Window.alert("DragEvent");
        }
    });

    button1.addDragLeaveHandler(new DragLeaveHandler () {
        public void onDragLeave(DragLeaveEvent event) {
            //Window.alert("DragLeaveEvent");
        }
    });

    button1.addDragOverHandler(new DragOverHandler () {
        public void onDragOver(DragOverEvent event) {
            //Window.alert("DragOverEvent");
        }
    });
}
```



```
button1.addDragStartHandler(new DragStartHandler () {
    public void onDragStart(DragStartEvent event) {
        //Window.alert("DragStartEvent");
    }
});

button1.addDropHandler(new DropHandler () {
    public void onDrop(DropEvent event) {
        //Window.alert("DropEvent");
    }
});
}}
```

### Обработчики событий сенсорного экрана:

- ◆ **addGestureChangeHandler(GestureChangeHandler handler)** — присоединяет к кнопке обработчик жеста сенсорного экрана:

```
button.addGestureChangeHandler(new GestureChangeHandler () {
    public void onGestureChange(GestureChangeEvent event) {
        Window.alert("GestureChangeEvent ");
    }
});
```

- ◆ **addGestureEndHandler(GestureEndHandler handler)** — присоединяет к кнопке обработчик окончания жеста сенсорного экрана:

```
button.addGestureEndHandler(new GestureEndHandler () {
    public void onGestureEnd(GestureEndEvent event) {
        Window.alert("GestureEndEvent ");
    }
});
```

- ◆ **addGestureStartHandler(GestureStartHandler handler)** — присоединяет к кнопке обработчик начала жеста сенсорного экрана:

```
button.addGestureStartHandler(new GestureStartHandler () {
    public void onGestureStart(GestureStartEvent event) {
        Window.alert("GestureStartEvent ");
    }
});
```

- ◆ **addTouchCancelHandler(TouchCancelHandler handler)** — присоединяет к кнопке обработчик прекращения прикосновения к сенсорному экрану:

```
button.addTouchCancelHandler(new TouchCancelHandler() {
    public void onTouchCancel(TouchCancelEvent event) {
        Window.alert("TouchCancelEvent");
    }
});
```

- ◆ **addTouchEndHandler(TouchEndHandler handler)** — присоединяет к кнопке обработчик окончания прикосновения к сенсорному экрану:

```
button.addTouchEndHandler(new TouchEndHandler() {
    public void onTouchEnd(TouchEndEvent event) {
        Window.alert("TouchEndEvent");
    }
});
```

- ◆ `addTouchMoveHandler(TouchMoveHandler handler)` — присоединяет к кнопке обработчик движения прикосновения к сенсорному экрану:

```
button.addTouchMoveHandler(new TouchMoveHandler() {  
    public void onTouchMove(TouchMoveEvent event) {  
        Window.alert("TouchMoveEvent");  
    }  
});
```

- ◆ `addTouchStartHandler(TouchStartHandler handler)` — присоединяет к кнопке обработчик начала прикосновения к сенсорному экрану:

```
button.addTouchStartHandler(new TouchStartHandler() {  
    public void onTouchStart(TouchStartEvent event) {  
        Window.alert("TouchStartEvent");  
    }  
});
```

### Обработчики фокуса:

- ◆ `addBlurHandler(BlurHandler handler)` — присоединяет к кнопке обработчик потери фокуса:

```
button.addBlurHandler(new BlurHandler () {  
    public void onBlur(BlurEvent event) {  
        Window.alert("BlurEvent");  
    }  
});
```

- ◆ `addFocusHandler(FocusHandler handler)` — присоединяет к кнопке обработчик наведения фокуса. Навести фокус на кнопку можно, например, с помощью клавиши `<Tab>`:

```
button.addFocusHandler(new FocusHandler () {  
    public void onFocus(FocusEvent event) {  
        Window.alert("FocusEvent");  
    }  
});
```

### Обработчики событий клавиатуры:

- ◆ `addKeyDownHandler(KeyDownHandler handler)` — присоединяет к кнопке обработчик события нажатия клавиши. Инициировать данное событие можно, например, наведя фокус на кнопку с помощью клавиши `<Tab>` и нажав клавишу:

```
button.addKeyDownHandler(new KeyDownHandler () {  
    public void onKeyDown(KeyDownEvent event) {  
        Window.alert("KeyDownEvent");  
    }  
});
```

- ◆ `addKeyPressHandler(KeyPressHandler handler)` — присоединяет к кнопке обработчик события нажатия и освобождения клавиши:

```
button.addKeyPressHandler(new KeyPressHandler () {  
    public void onKeyPress(KeyPressEvent event) {  
        Window.alert("KeyPressEvent");  
    }  
});
```

- ◆ `addKeyUpHandler(KeyUpHandler handler)` — присоединяет к кнопке обработчик события освобождения клавиши:

```
button.addKeyUpHandler(new KeyUpHandler () {  
    public void onKeyUp(KeyUpEvent event) {  
        Window.alert("KeyUpEvent");  
    }  
});
```

### Обработчики событий мыши:

- ◆ `addMouseDownHandler(MouseDownHandler handler)` — присоединяет к кнопке обработчик события нажатия клавиши мыши:

```
button.addMouseDownHandler(new MouseDownHandler() {  
    public void onMouseDown(MouseDownEvent event) {  
        Window.alert("MouseDownEvent");  
    }  
});
```

- ◆ `addMouseMoveHandler(MouseMoveHandler handler)` — присоединяет к кнопке обработчик события движения курсора мыши в области компонента:

```
button.addMouseMoveHandler(new MouseMoveHandler() {  
    public void onMouseMove(MouseMoveEvent event) {  
        Window.alert("MouseMoveEvent");  
    }  
});
```

- ◆ `addMouseOutHandler(MouseOutHandler handler)` — присоединяет к кнопке обработчик события выхода курсора мыши из области компонента:

```
button.addMouseOutHandler(new MouseOutHandler() {  
    public void onMouseOut(MouseOutEvent event) {  
        Window.alert("MouseOutEvent");  
    }  
});
```

- ◆ `addMouseOverHandler(MouseOverHandler handler)` — присоединяет к кнопке обработчик события проведения курсором мыши над областью компонента:

```
button.addMouseOverHandler(new MouseOverHandler() {  
    public void onMouseOver(MouseOverEvent event) {  
        Window.alert("MouseOverEvent");  
    }  
});
```

- ◆ `addMouseUpHandler(MouseUpHandler handler)` — присоединяет к кнопке обработчик события освобождения кнопки мыши:

```
button.addMouseUpHandler(new MouseUpHandler() {  
    public void onMouseUp(MouseUpEvent event) {  
        Window.alert("MouseUpEvent");  
    }  
});
```

- ◆ `addMouseWheelHandler(MouseWheelHandler handler)` — присоединяет к кнопке обработчик события прокручивания колесика мыши:

```
button.addMouseWheelHandler(new MouseWheelHandler() {  
    public void onMouseWheel(MouseWheelEvent event) {  
        Window.alert("MouseWheelEvent");  
    }  
});
```

## Определение свойств кнопки

Метод `setAccessKey(char key)`, унаследованный от класса `FocusWidget`, позволяет поставить в соответствие программной кнопке клавишу для активации кнопки при наведенном фокусе:

```
button.setAccessKey('C');
button.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        Window.alert("ClickEvent");
    }
});
```

В приведенном коде при наведенном фокусе на кнопке Alert-сообщение появляется при нажатии комбинации клавиш `<Alt>+<C>`.

Метод `setEnabled(boolean enabled)` класса `FocusWidget` позволяет деактивировать кнопку:

```
button.setEnabled(false);
```

Метод `setFocus(boolean focused)` класса `FocusWidget` дает возможность программным способом навести фокус на кнопку:

```
button.setFocus(true);
```

Методы `setHeight(java.lang.String height)`, `setPixelSize(int width, int height)`, `setSize(java.lang.String width, java.lang.String height)` и `setWidth(java.lang.String width)`, унаследованные от класса `UIObject`, устанавливают размеры кнопки:

```
button.setSize("200px", "30px");
```

С помощью метода `setVisible(boolean visible)` класса `UIObject` можно сделать кнопку невидимой:

```
button.setVisible(false);
```

Метод `setTitle(java.lang.String title)` класса `UIObject` позволяет определить всплывающую подсказку для кнопки (рис. 15.3):

```
button.setTitle("This is the button");
```

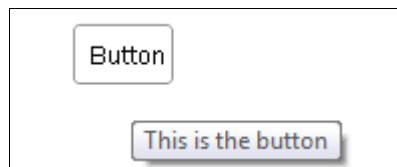


Рис. 15.3. Всплывающая подсказка для кнопки

## Кнопка *PushButton*

Отличие `PushButton`-компонента от `Button`-компонента заключается в том, что для `PushButton`-компонента программным способом можно разделить отображение программной кнопки при нажатой кнопке мыши и отображение программной кнопки при освобожденной кнопке мыши.

GUI-компонент `PushButton` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента строку одной надписи кнопки, строку надписи нажатой кнопки и строку надписи отжатой кнопки, изображение, изображение нажатой кнопки и изображение отжатой кнопки, обработчик событий щелчка кнопки:

```
final PushButton pushButton = new PushButton("pushButton");
```

или

```
import com.google.gwt.user.client.ui.Image;
final Image image = new Image();
image.setUrl("image.jpg");    // Изображение размещено в том же каталоге,
                               // что и HTML-страница приложения
final PushButton pushButton = new PushButton(image);
```

или

```
final PushButton pushButton = new PushButton("Up", "Down");
```

или

```
import com.google.gwt.user.client.ui.Image;
final Image imageUp = new Image();
imageUp.setUrl("imageUp.jpg");
final Image imageDown = new Image();
imageDown.setUrl("imageDown.jpg");
final PushButton pushButton = new PushButton(imageUp, imageDown);
```

При использовании конструктора класса `PushButton` с одной надписью кнопки в качестве аргумента, визуальное поведение `PushButton`-компонента все равно отличается от `Button`-компонента — состояния нажатой кнопки и отжатой кнопки визуально различны. Достигается этот эффект с помощью CSS-стилей кнопки по умолчанию.

Класс `PushButton` расширяет классы `CustomButton`, `ButtonBase`, `FocusWidget`, `Widget` и `UIObject`.

Для `PushButton`-компонента по умолчанию определены следующие CSS-классы:

- ◆ `.gwt-PushButton-up` — стиль отжатой кнопки;
- ◆ `.gwt-PushButton-down` — стиль нажатой кнопки;
- ◆ `.gwt-PushButton-up-hovering` — стиль отжатой кнопки при проведении над ней курсором мыши;
- ◆ `.gwt-PushButton-down-hovering` — стиль нажатой кнопки при проведении над ней курсором мыши;
- ◆ `.gwt-PushButton-up-disabled` — стиль отжатой деактивированной кнопки;
- ◆ `.gwt-PushButton-down-disabled` — стиль нажатой деактивированной кнопки;
- ◆ `.gwt-PushButton .html-face` — стиль надписи кнопки.

Надпись отжатой кнопки также может устанавливаться методом `setText(java.lang.String text)` класса `CustomButton`, принимающим в качестве аргумен-

та простую текстовую строку, или методом `setHTML([java.lang.String html]` или `[SafeHtml html])` класса `CustomButton`, принимающим в качестве аргумента текстовую строку, которая может включать в себя HTML-теги разметки текста, или объект `SafeHtml`, обеспечивающий безопасный HTML-код (см. разд. "Отличие метода `setText()` от метода `setHTML()`" ранее в этой главе).

**Методы** `getUpFace()`, `getDownFace()`, `getUpHoveringFace()`, `getDownHoveringFace()`, `getUpDisabledFace()` и `getDownDisabledFace()`, унаследованные от класса `CustomButton`, позволяют установить надписи и изображения кнопки в состояниях, которые соответствуют CSS-классам `.gwt-PushButton-up/down/up-hovering/down-hovering/up-disabled/down-disabled` кнопки:

```
pushButton.getDownHoveringFace().setText("Down-Hover");
pushButton.getDownDisabledFace().setText("Down-Dis");
pushButton.getDownFace().setText("DownFace");
pushButton.getUpHoveringFace().setText("Up-Hover");
pushButton.getUpFace().setText("UpFace");
pushButton.getUpDisabledFace().setText("Up_Dis");
```

Класс `PushButton` наследует от класса `FocusWidget` методы, позволяющие определить для кнопки различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств кнопки (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Переключатель *RadioButton*

Переключатель `RadioButton` создается с помощью конструктора класса, который может принимать в качестве аргумента имя группы переключателей и надпись переключателя:

```
RadioButton rb1 = new RadioButton("radioGroup", "radio1");
RadioButton rb2 = new RadioButton("radioGroup", "radio2");
final RootPanel panel=RootPanel.get("buttonContainer");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(rb1,150,200);
panel.add(rb2,250,200);
```

Объединение переключателей `RadioButton` в одну группу обеспечивает возможность выбора только одного переключателя из группы.

Отдельно определить имя группы для переключателя позволяет метод `setName(java.lang.String name)` класса `RadioButton`.

Класс `RadioButton` расширяет классы `CheckBox`, `ButtonBase`, `FocusWidget`, `Widget` и `UIObject`.

Для `RadioButton`-компонента по умолчанию определен CSS-класс `.gwt-RadioButton`.

С помощью метода `addValueChangeHandler(ValueChangeHandler<java.lang.Boolean> handler)` класса `CheckBox` для переключателя `RadioButton` можно определить обработчик события его выбора:

```
rb.addValueChangeListener(new ValueChangeListener<java.lang.Boolean>() {  
    public void onValueChange(ValueChangeEvent<java.lang.Boolean> event) {  
        Window.alert("ValueChangeEvent");  
    }  
});
```

Метод `setAccessKey(char key)` класса `CheckBox` позволяет определить для переключателя `RadioButton` клавишу активации `<Alt>+<key>`, метод `setEnabled(boolean enabled)` класса `CheckBox` — деактивировать переключатель, метод `setFocus(boolean focused)` — навести фокус, метод `setFormValue(java.lang.String value)` — определить значение атрибута `value` соответствующего компоненту HTML-элемента `<input type="radio" name="" value="">`.

Надпись переключателя может устанавливаться методом `setText(java.lang.String text)` класса `CheckBox`, принимающим в качестве аргумента простую текстовую строку, или методом `setHTML([java.lang.String html] или [SafeHtml html])`, принимающим в качестве аргумента текстовую строку, которая может включать в себя HTML-теги разметки текста, или объект `SafeHtml`, обеспечивающий безопасный HTML-код (см. разд. "Отличие метода `setText()` от метода `setHTML()`" ранее в этой главе).

Метод `setValue(java.lang.Boolean value)` класса `CheckBox` дает возможность сделать выбор переключателя программным способом:

```
rb.setValue(true);
```

Метод `setWordWrap(false)` класса `CheckBox` отменяет перенос слов в надписи переключателя.

Класс `RadioButton` также наследует от класса `FocusWidget` методы, позволяющие определить для переключателя различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств переключателя (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Флажок *CheckBox*

Флажок `CheckBox` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента строку надписи флажка в виде объекта `String` или объекта `SafeHtml`.

Отдельно надпись флажка может устанавливаться методом `setText(java.lang.String text)` класса `CheckBox`, принимающим в качестве аргумента простую текстовую строку, или методом `setHTML([java.lang.String html] или [SafeHtml html])`, принимающим в качестве аргумента текстовую строку, которая может включать в себя HTML-теги разметки текста, или объект `SafeHtml`, обеспечивающий безопасный HTML-код (см. разд. "Отличие метода `setText()` от метода `setHTML()`" ранее в этой главе).

Класс `CheckBox` расширяет классы `ButtonBase`, `FocusWidget`, `Widget` и `UIObject`.

Для `CheckBox`-компонента по умолчанию определены CSS-классы `.gwt-CheckBox` и `.gwt-CheckBox-disabled` (стиль деактивированного компонента).

С помощью метода `addValueChangeHandler(ValueChangeHandler<java.lang.Boolean> handler)` класса `CheckBox` для флажка можно определить обработчик события его выбора:

```
final CheckBox cb = new CheckBox("CheckBox");
cb.addValueChangeHandler(new ValueChangeHandler<java.lang.Boolean>() {
    public void onValueChange(ValueChangeEvent<java.lang.Boolean> event) {
        Window.alert("ValueChangeEvent");
    }
});
```

Отличие использования метода `addValueChangeHandler()` для флажка `CheckBox` от применения метода для переключателя `RadioButton` заключается в том, что для флажка обработчик событий `ValueChangeHandler` будет срабатывать всякий раз, когда пользователь установит флажок или сбросит его — для переключателя отменить его выбор нельзя.

Метод `setAccessKey(char key)` класса `CheckBox` позволяет определить для флажка клавишу активации `<Alt>+<key>`, метод `setEnabled(boolean enabled)` класса `CheckBox` — деактивировать флажок, метод `setFocus(boolean focused)` — навести фокус, метод `setFormValue(java.lang.String value)` — определить значение атрибута `value` соответствующего компоненту HTML-элемента `<input type="checkbox" name="" value="">`.

Метод `setValue(java.lang.Boolean value)` класса `CheckBox` дает возможность поставить или убрать флажок программным способом:

```
cb.setValue(true);
```

Соответственно проверить, поставлен флажок или нет, можно методом `getValue()`, возвращающим `true` или `false`.

Метод `setWordWrap(false)` класса `CheckBox` отменяет перенос слов в надписи флажка.

Класс `CheckBox` также наследует от класса `FocusWidget` методы, позволяющие определить для флажка различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств флажка (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Компонент выбора даты *DatePicker*

Компонент `DatePicker` обеспечивается пакетом `com.google.gwt.user.datepicker.client` и представляет собой составной компонент — панель календаря для выбора даты (рис. 15.4).

Компонент `DatePicker` создается с помощью конструктора класса без аргументов:

```
import com.google.gwt.user.datepicker.client.*;
final DatePicker datePicker = new DatePicker();
```

Класс `DatePicker` расширяет классы `Composite`, `Widget` и `UIObject`.

Для компонента `DatePicker` по умолчанию определены следующие CSS-классы:

◆ `.gwt-DatePicker` — общий стиль компонента (границы, курсор);

◆ `.datePickerMonthSelector` — стиль компонента выбора месяца 



- ◆ `.datePickerMonth` — стиль компонента месяца 

2012 Aug
----------

;
- ◆ `.datePickerPreviousButton` — стиль кнопки 

«
---

 компонента выбора месяца;
- ◆ `.datePickerNextButton` — стиль кнопки 

»
---

 компонента выбора месяца;
- ◆ `.datePickerDays` — стиль компонента выбора дня (рис. 15.5);

« 2012 Aug »						
M	T	W	T	F	S	S
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Рис. 15.4. Компонент DatePicker

M	T	W	T	F	S	S
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Рис. 15.5. Стиль выбора дня

- ◆ `.datePickerWeekdayLabel` — стиль компонента дней недели до выходных 

M	T	W	T	F
---	---	---	---	---

;
- ◆ `.datePickerWeekendLabel` — стиль компонента выходных дней недели 

S	S
---	---

;
- ◆ `.datePickerDay` — стиль компонента дней (рис. 15.6);
- ◆ `.datePickerDayIsToday` — стиль компонента текущего дня 

17
----

;
- ◆ `.datePickerDayIsWeekend` — стиль компонента выходных дней (рис. 15.7);

30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Рис. 15.6. Стиль дней



4	5
11	12
18	19
25	26
1	2
8	9

Рис. 15.7. Стиль выходных дней

- ◆ `.datePickerDayIsFiller` — стиль компонента дней другого месяца 

30	31
----	----

;
- ◆ `.datePickerDayIsValue` — стиль компонента выбранного дня;
- ◆ `.datePickerDayIsDisabled` — стиль деактивированного компонента;
- ◆ `.datePickerDayIsHighlighted` — стиль компонента дня при наведении курсора мыши;
- ◆ `.datePickerDayIsValueAndHighlighted` — стиль компонента значения даты при ее выборе.

**Методы** `addShowRangeHandler(ShowRangeHandler<java.util.Date> handler)` и `addShowRangeHandlerAndFire(ShowRangeHandler<java.util.Date> handler)` класса `DatePicker` позволяют определить для компонента обработчики событий отображения диапазона дней для выбора. Данные обработчики срабатывают при первоначальном отображении компонента и при нажатии кнопок ,  компонента выбора месяца:

```
datePicker.addShowRangeHandler(new ShowRangeHandler<Date>(){
    public void onShowRange(ShowRangeEvent<Date> event) {
        Window.alert("ShowRangeEvent");
    }
});

datePicker.addShowRangeHandlerAndFire(new ShowRangeHandler<Date>(){
    public void onShowRange(ShowRangeEvent<Date> event) {
        Window.alert("ShowRangeEventAndFire");
    }
});
```

**Метод** `addHighlightHandler(HighlightHandler<java.util.Date> handler)` класса `DatePicker` позволяет определить для компонента обработчик события подсвечивания даты, например, при наведении курсора мыши на дату календаря:

```
datePicker.addHighlightHandler(new HighlightHandler<Date>(){
    public void onHighlight(HighlightEvent<Date> event) {
        Window.alert("HighlightEvent");
    }
});
```

**Метод** `addValueChangeHandler(ValueChangeHandler<java.util.Date> handler)` класса `DatePicker` позволяет определить для компонента обработчик события выбора даты календаря:

```
datePicker.addValueChangeHandler(new ValueChangeHandler<Date>(){
    public void onValueChange(ValueChangeEvent<Date> event) {
        Window.alert("ValueChangeEvent");
    }
});
```

**Методы** `getCurrentMonth()`, `getFirstDate()`, `getLastDate()`, `getHighlightedDate()` и `getValue()` класса `DatePicker` возвращают отображаемый компонентом месяц, первую дату отображаемого диапазона, последнюю дату отображаемого диапазона, подсвеченную дату и выбранную дату:

```
datePicker.addValueChangeHandler(new ValueChangeHandler<Date>(){
    public void onValueChange(ValueChangeEvent<Date> event) {
        Window.alert(datePicker.getCurrentMonth().toString());
        Window.alert(datePicker.getFirstDate().toString());
        Window.alert(datePicker.getLastDate().toString());
        Window.alert(datePicker.getHighlightedDate().toString());
        Window.alert(datePicker.getValue().toString());
    }
});
```

**Методы** `setCurrentMonth(java.util.Date month)` и `setValue(java.util.Date newValue)` класса `DatePicker` дают возможность программным способом определить отображаемый месяц и выбранную дату.

Класс `DatePicker` также наследует от класса `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Кнопка *ToggleButton*

Отличие `ToggleButton`-компонента от `PushButton`-компонента заключается в том, что для `ToggleButton`-компонента состояние нажатой кнопки фиксируется, в то время как нажатая `PushButton`-кнопка сразу становится отжатой при освобождении кнопки мыши.

GUI-компонент `ToggleButton` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента строку одной надписи кнопки, строку надписи нажатой кнопки и строку надписи отжатой кнопки, изображение, изображение нажатой кнопки и изображение отжатой кнопки, обработчик событий щелчка кнопки:

```
final ToggleButton toggleButton = new ToggleButton("toggleButton");
```

или

```
import com.google.gwt.user.client.ui.Image;
final Image image = new Image();
image.setUrl("image.jpg"); // Изображение размещено в том же каталоге,
                             // что и HTML-страница приложения
final ToggleButton toggleButton = new ToggleButton(image);
```

или

```
final ToggleButton toggleButton = new ToggleButton("Up", "Down");
```

или

```
import com.google.gwt.user.client.ui.Image;
final Image imageUp = new Image();
imageUp.setUrl("imageUp.jpg");
final Image imageDown = new Image();
imageDown.setUrl("imageDown.jpg");
final ToggleButton toggleButton = new ToggleButton(imageUp, imageDown);
```

Класс `ToggleButton` расширяет классы `CustomButton`, `ButtonBase`, `FocusWidget`, `Widget` и `UIObject`.

Для `ToggleButton`-компонента по умолчанию определены следующие CSS-классы:

- ◆ `.gwt-ToggleButton-up` — стиль отжатой кнопки;
- ◆ `.gwt-ToggleButton-down` — стиль нажатой кнопки;
- ◆ `.gwt-ToggleButton-up-hovering` — стиль отжатой кнопки при проведении над ней курсором мыши;
- ◆ `.gwt-ToggleButton-down-hovering` — стиль нажатой кнопки при проведении над ней курсором мыши;
- ◆ `.gwt-ToggleButton-up-disabled` — стиль отжатой деактивированной кнопки;

- ◆ `.gwt-ToggleButton-down-disabled` — стиль нажатой деактивированной кнопки;
- ◆ `.gwt-ToggleButton.html-face` — стиль надписи кнопки.

Программным способом зафиксировать кнопку в нажатом состоянии можно методом `setDown(boolean down)` или методом `setValue(java.lang.Boolean value)` класса `ToggleButton`. При этом метод `setDown()` имеет приоритет перед методом `setValue()` — при одновременном использовании методов:

```
toggleButton.setValue(true);
toggleButton.setDown(false);
```

кнопка останется отжатой.

Проверить состояние кнопки можно методом `isDown()` или методом `getValue()` класса `ToggleButton`.

Метод `addValueChangeHandler(ValueChangeHandler<java.lang.Boolean> handler)` класса `ToggleButton` позволяет определить для компонента обработчик события изменения состояния кнопки:

```
toggleButton.addValueChangeHandler(new ValueChangeHandler<java.lang.Boolean>() {
    public void onValueChange(ValueChangeEvent<java.lang.Boolean> event) {
        Window.alert("ValueChangeEvent");
    }
});
```

Надпись отжатой кнопки также может устанавливаться методом `setText(java.lang.String text)` класса `CustomButton`, принимающим в качестве аргумента простую текстовую строку, или методом `setHTML([java.lang.String html] или [SafeHtml html])` класса `CustomButton`, принимающим в качестве аргумента текстовую строку, которая может включать в себя HTML-теги разметки текста, или объект `SafeHtml`, обеспечивающий безопасный HTML-код (см. разд. "Отличие метода `setText()` от метода `setHTML()`" ранее в этой главе).

Методы `getUpFace()`, `getDownFace()`, `getUpHoveringFace()`, `getDownHoveringFace()`, `getUpDisabledFace()` и `getDownDisabledFace()`, унаследованные от класса `CustomButton`, позволяют установить надписи и изображения кнопки в состояниях, которые соответствуют CSS-классам `.gwt-ToggleButton-up/down/up-hovering/down-hovering/up-disabled/down-disabled` кнопки:

```
toggleButton.getDownHoveringFace().setText("Down-Hover");
toggleButton.getDownDisabledFace().setText("Down-Dis");
toggleButton.getDownFace().setText("DownFace");
toggleButton.getUpHoveringFace().setText("Up-Hover");
toggleButton.getUpFace().setText("UpFace");
toggleButton.getUpDisabledFace().setText("Up_Dis");
```

Класс `ToggleButton` наследует от класса `FocusWidget` методы, позволяющие определить для кнопки различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств кнопки (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Текстовое поле *TextBox*

GUI-компонент `TextBox` представляет собой поле для ввода пользователем текстовой строки и создается с помощью конструктора класса без аргументов.

Класс `TextBox` расширяет классы `TextBoxBase`, `ValueBoxBase<java.lang.String>`, `FocusWidget`, `Widget` и `UIObject`.

Для `TextBox`-компонента по умолчанию определены CSS-классы `.gwt-TextBox` и `.gwt-TextBox-readonly` (стиль для не редактируемого поля).

Метод `setMaxLength(int length)` класса `TextBox` позволяет ограничить количество вводимых символов, а метод `setVisibleLength(int length)` класса `TextBox` устанавливает ширину поля в видимых символах, при этом метод `setSize()` класса `UIObject` имеет приоритет перед методом `setVisibleLength()`:

```
final TextBox tb = new TextBox();
tb.setMaxLength(10);
//tb.setSize("20px", "10px");
tb.setVisibleLength(50);
```

Метод `getValue()`, унаследованный от класса `TextBoxBase`, позволяет получить введенную строку текста. Введенную строку также можно получить методом `getText()`, унаследованным от класса `ValueBoxBase`.

Методы `addChangeHandler(ChangeHandler handler)` и `addValueChangeHandler(ValueChangeHandler<java.lang.String> handler)` класса `ValueBoxBase<java.lang.String>` дают возможность определить для поля обработчики события изменения строки поля, которые срабатывают при нажатии клавиши `<Enter>`:

```
tb.addChangeHandler(new ChangeHandler() {
    public void onChange(ChangeEvent event) {
        Window.alert(tb.getValue());
    }
});
tb.addValueChangeHandler(new ValueChangeHandler<java.lang.String>() {
    public void onValueChange(ValueChangeEvent<java.lang.String> event) {
        Window.alert(tb.getValue());
    }
});
```

Метод `cancelKey()` класса `ValueBoxBase` заглушает событие клавиатуры и тем самым позволяет организовать фильтрацию вводимых символов:

```
tb.addKeyPressHandler(new KeyPressHandler() {
    public void onKeyPress(KeyPressEvent event) {
        if (!Character.isDigit(event.getCharCode())) {
            ((TextBox) event.getSource()).cancelKey();
        }
    }
});
```

В данном коде разрешается ввод только цифр.

Метод `getCursorPos()` класса `ValueBoxBase` позволяет получить позицию курсора в поле, а методы `getSelectedText()` и `getSelectionLength()` — выделенные символы строки поля и количество выделенных символов строки поля.

Переопределение метода `onBrowserEvent(Event event)` класса `ValueBoxBase` в конструкторе класса `TextBox` позволяет обрабатывать различные события браузера:

```
final TextBox tb = new TextBox(){
@Override
public void onBrowserEvent(Event event){
    super.onBrowserEvent(event);
    if(event.getType()=="blur"){
        Window.alert(this.getText());
    }
};
```

В данном коде обработчик события срабатывает при потере фокуса `TextBox`-компонентом.

Метод `selectAll()` класса `ValueBoxBase` выделяет всю строку поля, метод `setCursorPos(int pos)` устанавливает позицию курсора в поле, метод `setSelectionRange(int pos, int length)` выделяет часть строки, методы `setText(java.lang.String text)` и `setValue(java.lang.String text)` устанавливают текст поля, метод `setAlignment(ValueBoxBase.TextAlignment align)` устанавливает выравнивание текста в поле:

```
tb.setAlignment(ValueBoxBase.TextAlignment.CENTER);
```

Метод `setReadOnly(boolean readOnly)` класса `ValueBoxBase` дает возможность исключить редактирование поля.

Класс `TextBox` также наследует от класса `FocusWidget` методы, позволяющие определить для поля различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств поля (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Поле ввода пароля *PasswordTextBox*

GUI-компонент `PasswordTextBox` представляет собой поле для ввода пользователем текстовой строки пароля и создается с помощью конструктора класса без аргументов.

Класс `PasswordTextBox` расширяет классы `TextBox`, `TextBoxBase`, `ValueBoxBase<java.lang.String>`, `FocusWidget`, `Widget` и `UIObject`.

Для `PasswordTextBox`-компонента по умолчанию определены CSS-классы `.gwt-PasswordTextBox` и `.gwt-PasswordTextBox-readonly` (стиль для нередатируемого поля).

Компонент `PasswordTextBox` отличается от компонента `TextBox` только тем, что вводимые в `PasswordTextBox`-поле символы маскируются символами •, во всем остальном свойства `PasswordTextBox`-компонента аналогичны свойствам `TextBox`-компонента.

## Текстовая область *TextArea*

GUI-компонент `TextArea` представляет собой поле для ввода пользователем текстовых строк и создается с помощью конструктора класса без аргументов.

Класс `TextArea` расширяет классы `TextBoxBase`, `ValueBoxBase<java.lang.String>`, `FocusWidget`, `Widget` и `UIObject`.

Для `TextArea`-компонента по умолчанию определены CSS-классы `.gwt-TextArea` и `.gwt-TextArea-readonly` (стиль для нередактируемой области).

Компонент `TextArea` отличается от компонента `TextBox` возможностью ввода не одной строки, а нескольких текстовых строк.

Метод `setVisibleLines(int lines)` класса `TextArea` позволяет ограничить количество видимых текстовых строк, а метод `setCharacterWidth(int width)` класса `TextArea` ограничивает ширину компонента количеством вводимых символов в одной строке, при этом метод `setSize()` класса `UIObject` имеет приоритет перед методами `setVisibleLines()` и `setCharacterWidth()`:

```
final TextArea ta = new TextArea();
ta.setCharacterWidth(5);
ta.setVisibleLines(2);
ta.setSize("200px", "200px");
```

При превышении установленной высоты компонента количеством введенных строк у текстовой области автоматически появляется вертикальная полоса прокрутки.

Метод `getValue()`, унаследованный от класса `TextBoxBase`, позволяет получить введенный текст. Введенный текст также можно получить методом `getText()`, унаследованным от класса `ValueBoxBase`.

Методы `addChangeHandler(ChangeHandler handler)` и `addValueChangeHandler(ValueChangeHandler<java.lang.String> handler)` класса `ValueBoxBase<java.lang.String>` дают возможность определить для текстовой области обработчики события изменения текста, которые срабатывают при потере фокуса текстовой областью:

```
ta.addChangeHandler(new ChangeHandler() {
    public void onChange(ChangeEvent event) {
        Window.alert(ta.getValue());
    }
});
ta.addValueChangeHandler(new ValueChangeHandler<java.lang.String>() {
    public void onValueChange(ValueChangeEvent<java.lang.String> event) {
        Window.alert(ta.getValue());
    }
});
```

Метод `cancelKey()` класса `ValueBoxBase` заглушает событие клавиатуры и тем самым позволяет организовать фильтрацию вводимых символов:

```
ta.addKeyPressHandler(new KeyPressHandler() {
    public void onKeyPress(KeyPressEvent event) {
```

```

        if (!Character.isDigit(event.getCharCode())) {
            ((TextArea) event.getSource()).cancelKey();
        }
    });
}

```

В данном коде разрешен ввод только цифр.

Метод `getCursorPos()` класса `TextArea` позволяет получить позицию курсора в текстовой области, а методы `getSelectedText()` класса `ValueBoxBase` и `getSelectionLength()` класса `TextArea` — выделенные символы текста и количество выделенных символов текста.

Переопределение метода `onBrowserEvent(Event event)` класса `ValueBoxBase` в конструкторе класса `TextArea` позволяет обрабатывать различные события браузера:

```

final TextArea ta = new TextArea () {
    @Override
    public void onBrowserEvent(Event event) {
        super.onBrowserEvent(event);
        switch (event.getTypeInt()) {
            case Event.ONPASTE:
                Window.alert(this.getText());
                break;
        }
    }
};
ta.sinkEvents(Event.ONPASTE);

```

В данном коде обработчик события срабатывает при вставке текста в `TextArea`-компонент.

Метод `selectAll()` класса `ValueBoxBase` выделяет весь текст компонента, метод `setCursorPos(int pos)` устанавливает позицию курсора в тексте, метод `setSelectionRange(int pos, int length)` выделяет часть текста, методы `setText(java.lang.String text)` и `setValue(java.lang.String text)` устанавливают текст компонента, метод `setAlignment(ValueBoxBase.TextAlignment align)` устанавливает выравнивание текста в компоненте:

```
ta.setAlignment(ValueBoxBase.TextAlignment.CENTER);
```

Метод `setReadOnly(boolean readOnly)` класса `ValueBoxBase` дает возможность исключить редактирование текстовой области.

Класс `TextArea` также наследует от класса `FocusWidget` методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Гиперссылка *Hyperlink*

Компонент `Hyperlink` предназначен для совместного использования с механизмом истории `History` фреймворка GWT и обеспечивает создание ссылок на различные внутренние состояния GWT-приложения.




GUI-компонент `Hyperlink` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента строку надписи ссылки и маркер истории:

```
public void onModuleLoad() {
    Hyperlink link = new Hyperlink("link to foo", "foo");
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(link, 110,29);
    link.setSize("100px", "20px");
    link.setStyleName("my-HyperLink");
    History.addValueChangeHandler(new ValueChangeHandler<String>() {
        public void onValueChange(ValueChangeEvent<String> event) {
            String historyToken = event.getValue();
            if (historyToken.equals("foo")) {
                Window.alert("Foo State");
            }
            panel.getElement().getStyle().setProperty("backgroundColor", "red");
        }
    });
    if (historyToken.equals("init")) {
        Window.alert("Init State");
        panel.getElement().getStyle().setProperty("backgroundColor", "blue");
    }
}
History.newItem("init",true);
}
```

В приведенном коде при создании объекта `Hyperlink` маркер `"foo"` автоматически добавляется в стек объекта `History` с возможностью его дальнейшей обработки в обработчике событий изменения значения объекта `History`. После создания объекта `Hyperlink` и соответственно добавления маркера `"foo"`, в стек объекта `History` добавляется маркер `"init"`, поэтому GWT-приложение изначально открывается с URL-адресом, имеющим расширение в виде `#init`.

При нажатии `Hyperlink`-ссылки к URL-адресу GWT-приложения вместо маркера `#init` добавляется маркер `#foo`, при этом срабатывает метод `onValueChange()` обработчика `ValueChangeHandler` событий объекта `History`, в котором приложение переводится в новое состояние, характеризующееся красным цветом фона контейнера `Hyperlink`-ссылки.

Вернуться обратно к URL-адресу с маркером `#init` позволяет кнопка  Web-браузера, при этом вновь срабатывает метод `onValueChange()` обработчика `ValueChangeHandler` событий объекта `History`, в котором приложение переводится в стартовое состояние, характеризующееся синим цветом фона контейнера `Hyperlink`-ссылки.

Перейти снова к URL-адресу с маркером `#foo` можно с помощью кнопки  Web-браузера или щелчка `Hyperlink`-ссылки.

HTML-страница GWT-приложения содержит код:

```
<iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
style="position:absolute;width:0;height:0;border:0"></iframe>
<h1>Web Application Starter Project</h1>
<div id="container"></div>
```

Здесь фрейм `__gwt_historyFrame` обеспечивает поддержку механизма истории History, а блок `container` служит контейнером для Hyperlink-ссылки.

Файл CSS-стилей HTML-страницы GWT-приложения содержит код:

```
#container{
    background-color: blue;
    width:300px;
    height:300px;
}
.my-HyperLink a,.my-HyperLink a:visited{
    color: white;
}
.my-HyperLink a:hover{
    color: white;
}
```

Таким образом, объект `Hyperlink` обеспечивает автоматическое добавление маркера в стек объекта `History`, а щелчок на `Hyperlink`-ссылке переводит ее маркер в текущий маркер объекта `History`.

Класс `Hyperlink` расширяет классы `Widget` и `UIObject`.

Для `Hyperlink`-компонента по умолчанию определен CSS-класс `.gwt-Hyperlink`.

Отдельно определить надпись гиперссылки позволяют методы `setHTML()` и `setText()` класса `Hyperlink`, а установить маркер истории — метод `setTargetHistoryToken(java.lang.String targetHistoryToken)` класса `Hyperlink`.

Класс `Hyperlink` также наследует от класса `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Гиперссылка *Anchor*

Компонент `Anchor` является оберткой HTML-элемента `<a>` и создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента надпись ссылки, ее адрес и значение атрибута `target` тега `<a>`.

Класс `Anchor` расширяет классы `FocusWidget`, `Widget` и `UIObject` и по умолчанию для `Anchor`-компонента определен CSS-класс `.gwt-Anchor`.

Метод `setHorizontalAlignment(HasHorizontalAlignment.HorizontalAlignmentConstant align)` класса `Anchor` определяет горизонтальное выравнивание компонента, метод `setHref(java.lang.String href)` устанавливает адрес ссылки, методы `setHTML()` и

`setText()` — надпись ссылки, метод `setTarget()` указывает, где открывается ссылка (`_blank`, `_parent`, `_self`, `_top`), метод `setWordWrap(boolean wrap)` включает перенос слов надписи ссылки.

Класс `Anchor` также наследует от класса `FocusWidget` методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Список выбора *ListBox*

GUI-компонент `ListBox` представляет собой список выбора элементов и создается с помощью конструктора класса без аргументов или конструктора `ListBox(boolean isMultipleSelect)` класса, в случае аргумента `true` создающего список с возможностью множественного выбора элементов. По умолчанию создается список с возможностью выбора только одного элемента.

Класс `ListBox` расширяет классы `FocusWidget`, `Widget` и `UIObject`.

Для `ListBox`-компонента по умолчанию определен CSS-класс `.gwt-ListBox`.

Наполнить список элементами позволяет метод `addItem(java.lang.String item)` класса `ListBox`. При этом установить значение элемента можно методом `setValue(int index, java.lang.String value)` или методом `addItem(java.lang.String item, java.lang.String value)` класса `ListBox`. Нумерация элементов в списке начинается с 0.

С помощью метода `addChangeHandler(ChangeHandler handler)` класса `ListBox` для списка можно определить обработчик события выбора элемента:

```
final ListBox lb = new ListBox();
lb.addItem("item1");
lb.addItem("item2");
lb.setValue(1, "myItem");
lb.addItem("item3");
lb.addItem("item4");
lb.addItem("item5");
lb.setVisibleItemCount(5);
lb.addChangeHandler(new ChangeHandler() {
    public void onChange(ChangeEvent event) {
        int index=lb.getSelectedIndex();
        Window.alert(index+"");
        Window.alert(lb.getItemText(index));
        Window.alert(lb.getValue(index));
    }
});
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(lb,100,20);
```

Метод `getSelectedIndex()` класса `ListBox` позволяет получить номер выбранного элемента, метод `getItemText(int index)` — отображаемую надпись выбранного элемента, метод `getValue(int index)` — значение выбранного элемента.

По умолчанию создается выпадающий список с отображением только одного элемента. Создать полностью отображаемый список позволяет метод `setVisibleItemCount(int visibleItems)` класса `ListBox`.

Метод `clear()` класса `ListBox` удаляет все элементы из списка, метод `insertItem(java.lang.String item, int index)` вносит элемент в список по заданному номеру, метод `isItemSelected(int index)` проверяет выбор элемента, метод `removeItem(int index)` удаляет элемент списка, методы `setItemSelected(int index, boolean selected)` и `setSelectedIndex(int index)` позволяют программным способом выбрать элемент списка, метод `setItemText(int index, java.lang.String text)` — изменить отображаемую надпись элемента списка.

Класс `ListBox` также наследует от класса `FocusWidget` методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Компоненты *Cell Widgets*

Фреймворк GWT предлагает набор компонентов `CellList`, `CellTable`, `DataGrid`, `CellTree` и `CellBrowser` пакета `com.google.gwt.user.cellview.client`, с помощью одного объекта обеспечивающих отображение большого набора данных. `Cell`-компоненты избавляют от необходимости для каждого элемента набора данных создавать свой GUI-компонент со своим обработчиком событий — `Cell`-компонент представляет сразу весь набор данных, и его обработчик событий обрабатывает события каждого элемента набора данных.

`Cell`-компонент является составным компонентом, т. к. за отображение каждого элемента набора данных отвечает объект ячейки `com.google.gwt.cell.client.Cell<C>`, являющийся основой `Cell`-компонента.

Фреймворк GWT предоставляет большой набор классов `Cell`-ячеек пакета `com.google.gwt.cell.client` для создания на их основе `Cell`-компонентов:

- ◆ `TextCell` — не редактируемая ячейка, отображающая текст;
- ◆ `ClickableTextCell` — текстовая ячейка, при щелчке на которой вызывается ее обработчик событий `ValueUpdater`;
- ◆ `EditTextCell` — редактируемая текстовая ячейка;
- ◆ `TextInputCell` — ячейка для ввода текста;
- ◆ `ActionCell<C>` — ячейка кнопки с обработчиком событий;
- ◆ `ButtonCell` — ячейка кнопки;
- ◆ `CheckboxCell` — ячейка флажка;

- ◆ `SelectionCell` — ячейка выпадающего меню выбора;
- ◆ `DateCell` — ячейка отображения даты;
- ◆ `DatePickerCell` — ячейка выбора даты;
- ◆ `ImageCell` — ячейка отображения изображения по URL-адресу;
- ◆ `ImageResourceCell` — ячейка отображения статического изображения;
- ◆ `ImageLoadingCell` — ячейка отображения изображения по URL-адресу с индикатором загрузки;
- ◆ `NumberCell` — ячейка отображения чисел;
- ◆ `CompositeCell<C>` — составная ячейка;
- ◆ `IconCellDecorator<C>` — ячейка значка для другой ячейки.

Для создания `Cell`-компонента необходимо в первую очередь создать объект `Cell`-ячейки, затем на его основе создать `Cell`-компонент, определить набор данных и добавить его в `Cell`-компонент с помощью метода `setRowData()` либо поставщика данных `ListDataProvider` или `AsyncDataProvider`.

## Столбец `CellList`

Класс `CellList` представляет столбец ячеек и расширяет класс `AbstractHasData<T>` пакета `com.google.gwt.user.cellview.client`, а также классы `Composite`, `Widget` и `UIObject` пакета `com.google.gwt.user.client.ui`.

В самом простом случае компонент `CellList` создается с помощью конструктора класса `CellList(Cell<T> cell)`:

```
public class GWTApplication implements EntryPoint {
    private static final List<String> Items= Arrays.asList("Item1", "Item2", "Item3");
    public void onModuleLoad() {
        TextCell textCell = new TextCell();
        CellList<String> cellList = new CellList<String>(textCell);
        cellList.setRowData(Items);
        final RootPanel panel=RootPanel.get("container");
        panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
        panel.add(cellList,100,50);
    }
}
```

В приведенном коде набор данных определяется в виде `List`-списка текстовых строк, `Cell`-ячейка — как объект класса `TextCell` и набор данных связывается с `CellList`-компонентом с помощью метода `setRowData()` класса `AbstractHasData<T>`. В результате набор данных отображается в виде набора HTML-строк:

```
Item1
Item2
Item3
```

По умолчанию CSS-стиль `CellList`-компонента определяется файлом `CellList.css` пакета `com.google.gwt.user.cellview.client` библиотеки `gwt-user.jar`. Переопределить

стиль по умолчанию CellList-компонента позволяет конструктор класса `CellList(Cell<T> cell, CellList.Resources resources)`, принимающий в качестве аргумента объект `com.google.gwt.user.cellview.client.CellList.Resources`.

Интерфейс `CellList.Resources` расширяет интерфейс `com.google.gwt.resources.client.ClientBundle`, гарантирующий кэширование ресурсов GWT-приложения. Для создания объекта `CellList.Resources` создадим интерфейс `CellListResource`, расширяющий интерфейс `CellList.Resources`:

```
package com.example.gwt.client;
import com.google.gwt.user.cellview.client.CellList;
public interface CellListResource extends CellList.Resources{
    public interface CellListStyle extends CellList.Style {};
    @Source({"CellList.css"})
    CellListStyle cellListStyle();
};
```

Также создадим файл `CellList.css`, который разместим вместе с интерфейсом `CellListResource`:

```
.cellListWidget {
    background: gray;
}
.cellListEvenItem {
    cursor: pointer;
    zoom: 1;
}
.cellListOddItem {
    cursor: pointer;
    zoom: 1;
}
.cellListKeyboardSelectedItem {
    background: #ffc;
}
@sprite .cellListSelectedItem {
    gwt-image: 'cellListSelectedBackground';
    background-color: red;
    color: white;
    height: auto;
    overflow: visible;
}
```

В файле `CellList.css` переопределены общий фон CellList-компонента и фон выбранного мышью элемента.

В заключение создадим CellList-компонент с учетом интерфейса `CellListResource`:

```
TextCell textCell = new TextCell();
CellListResource resource = GWT.create(CellListResource.class);
```

```
CellList<String> cellList = new CellList<String>(textCell, resource);  
cellList.setRowData(Items);
```

CellList-компонент можно также создать с помощью конструктора `CellList(Cell<T> cell, ProvidesKey<T> keyProvider)`, принимающего в качестве аргумента объект `com.google.gwt.view.client.ProvidesKey<T>`.

Интерфейс `ProvidesKey` обеспечивает ключи для элементов набора данных и используется совместно с моделью выбора элементов набора данных.

Определить модель выбора для CellList-компонента позволяет метод `setSelectionModel(SelectionModel<? super T> selectionModel)` класса `AbstractHasData<T>`, где интерфейс `com.google.gwt.view.client.SelectionModel<T>` реализуется набором классов пакета `com.google.gwt.view.client`, в том числе классами `SingleSelectionModel` и `MultiSelectionModel`.

Класс `SingleSelectionModel` обеспечивает выбор только одного элемента, который возвращается методом `T getSelectedObject()`. Метод класса `setSelected(T item, boolean selected)` обеспечивает программный выбор элемента, а метод `addSelectionChangeHandler(SelectionChangeEvent.Handler handler)` определяет обработчик события выбора элемента. Метод `getKey(T item)` возвращает ключ элемента.

Поставщик ключей `ProvidesKey` может быть определен для модели выбора `SingleSelectionModel` в конструкторе класса `SingleSelectionModel(ProvidesKey<T> keyProvider)`.

Класс `MultiSelectionModel` обеспечивает множественный выбор элементов, которые возвращаются методом `java.util.Set<T> getSelectedSet()`. Методы `setSelected()`, `addSelectionChangeHandler()` и `getKey()` работают аналогично классу `SingleSelectionModel`.

Поставщик ключей `ProvidesKey` может быть определен для модели выбора `MultiSelectionModel` в конструкторе класса `MultiSelectionModel(ProvidesKey<T> keyProvider)`.

Следующий код демонстрирует работу поставщика ключей `ProvidesKey`:

```
public class GWTApplication implements EntryPoint {  
    private static final List<String> Items= Arrays.asList("Item1", "Item2", "Item3");  
    public void onModuleLoad() {  
        TextCell textCell = new TextCell();  
        ProvidesKey<String> keyProvider = new ProvidesKey<String>() {  
            private int id=0;  
            public Object getKey(String item) {  
                if (item.equals("Item1")) {  
                    id=1;  
                }  
                if (item.equals("Item2")) {  
                    id=2;  
                }  
                if (item.equals("Item3")) {  
                    id=3;  
                }  
            }  
        }  
    }  
}
```

```

        return id;
    });
    CellList<String> cellList = new CellList<String>(textCell, keyProvider);
    final SingleSelectionModel<String> selectionModel = new
    SingleSelectionModel<String>(keyProvider);
    cellList.setSelectionModel(selectionModel);
    selectionModel.addSelectionChangeHandler(new SelectionChangeEvent.Handler() {
        public void onSelectionChange(SelectionChangeEvent event) {
            String selected = selectionModel.getSelectedObject();
            if (selected != null) {
                Window.alert("You selected: " + selectionModel.getKey(selected));
            }
        }
    });
    cellList.setRowData(Items);
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(cellList,100,50);
}

```

В отсутствие объекта `ProvidesKey` в качестве аргумента конструктора класса `SingleSelectionModel`, при выборе элемента обработчик события отображает сам элемент. Если же присоединить объект `ProvidesKey` к модели выбора `SingleSelectionModel`, тогда при выборе элемента обработчик события отображает порядковый номер элемента, определенный поставщиком ключей.

Метод `setEmptyListWidget(Widget widget)` класса `CellList` позволяет установить `Widget`-компонент, отображаемый в случае пустого набора данных:

```

public class GWTApplication implements EntryPoint {
    private static final List<String> Items= new ArrayList<String>();
    public void onModuleLoad() {
        Items.add("Item1");
        Items.add("Item2");
        Items.add("Item3");
        TextCell textCell = new TextCell();
        CellList<String> cellList = new CellList<String>(textCell);
        cellList.setEmptyListWidget(new Label("List Empty"));
        Items.clear();
        cellList.setRowData(Items);
        final RootPanel panel=RootPanel.get("container");
        panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
        panel.add(cellList,100,50);
    }
}

```

Метод `setLoadingIndicator(Widget widget)` класса `CellList` определяет `Widget`-компонент, отображаемый при загрузке набора данных.

Метод `setValueUpdater(ValueUpdater<T> valueUpdater)` класса `CellList` позволяет определить обработчик события изменения элемента данных ячейки:



```

public class GWTApplication implements EntryPoint {
private static final List<String> Items= new ArrayList<String>();
private int index;
public void onModuleLoad() {
    Items.add("Item1");
    Items.add("Item2");
    Items.add("Item3");
    EditTextCell textCell = new EditTextCell();
    ProvidesKey<String> keyProvider = new ProvidesKey<String>() {
        private int id;
        public Object getKey(String item) {
            if (item.equals("Item1")) {
                id=0;
            }
            if (item.equals("Item2")) {
                id=1;
            }
            if (item.equals("Item3")) {
                id=2;
            }
            return id;
        }
    };
    final CellList<String> cellList = new CellList<String>(textCell, keyProvider);
    final SingleSelectionModel<String> selectionModel = new
    SingleSelectionModel<String>(keyProvider);
    cellList.setSelectionModel(selectionModel);
    selectionModel.addSelectionChangeHandler(new SelectionChangeEvent.Handler() {
        public void onSelectionChange(SelectionChangeEvent event) {
            String selected = selectionModel.getSelectedObject();
            if (selected != null) {
                index= new Integer(""+ selectionModel.getKey(selected));
            }
        }
    });
    ValueUpdater<String> valueUpdater = new ValueUpdater<String>() {
    public void update(String newValue) {
        Items.remove(index);
        Items.add(index, newValue);
        cellList.setRowData(Items);
    };
    cellList.setValueUpdater(valueUpdater);
    cellList.setRowData(Items);
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(cellList,100,50);
}
}

```

В приведенном коде обработчик ValueUpdater изменяет набор данных с учетом нового значения поля EditTextCell.

**Метод** `addCellPreviewHandler(CellPreviewEvent.Handler<T> handler)` **класса** `AbstractHasData` позволяет определить для `CellList`-компонента обработчик, перехватывающий события ячеек компонента:

```
final CellList<String> cellList = new CellList<String>(textCell);
cellList.addCellPreviewHandler(new CellPreviewEvent.Handler<String>() {
public void onCellPreview(CellPreviewEvent<String> event) {
    if ("mousedown".equals(event.getNativeEvent().getType())) {
        Element cellElement = event.getNativeEvent().getEventTarget().cast();
        cellElement.setInnerText("Over");
    }
}}});
```

В приведенном коде при нажатии на ячейке изменяется ее содержимое.

**Метод** `addLoadingStateChangeHandler>LoadingStateChangeEvent.Handler handler)` **класса** `AbstractHasData` позволяет определить для `CellList`-компонента обработчик загрузки компонента:

```
cellList.addLoadingStateChangeHandler(new LoadingStateChangeEvent.Handler() { @Override
    public void onLoadingStateChanged>LoadingStateChangeEvent event) {
        if(event.getLoadingState() == LoadingState.LOADED) {
            Window.alert("Rows: " + cellList.getRowCount());
        }
    }
});
```

**Метод** `addRangeChangeHandler(RangeChangeEvent.Handler handler)` **класса** `AbstractHasData` позволяет определить для `CellList`-компонента обработчик события изменения диапазона отображения набора данных:

```
cellList.addRangeChangeHandler(new RangeChangeEvent.Handler () {
    @Override
    public void onRangeChange(RangeChangeEvent event) {
        Window.alert("Range: " + event.getNewRange());
    }
});
cellList.setVisibleRange(1, 2);
```

**Отображаемый диапазон набора данных устанавливается методом** `setVisibleRange(int start, int length)` **класса** `AbstractHasData`.

**Метод** `addRowCountChangeHandler(RowCountChangeEvent.Handler handler)` **класса** `AbstractHasData` позволяет определить для `CellList`-компонента обработчик события изменения количества строк списка:

```
cellList.addRowCountChangeHandler(new RowCountChangeEvent.Handler () {
    @Override
    public void onRowCountChange(RowCountChangeEvent event) {
        Window.alert("Rows: " + event.getNewRowCount());
    }
});
cellList.setRowCount(2);
```

**Количество строк списка устанавливается методом** `setRowCount(int count)` **класса** `AbstractHasData`.

Переопределение метода `onBrowserEvent(Event event)` класса `AbstractHasData` позволяет обрабатывать различные события браузера для `CellList`-компонента.

Метод `redraw()` класса `AbstractHasData` перерисовывает `CellList`-компонент с учетом уже определенного набора данных, метод `redrawRow(int absRowIndex)` перерисовывает заданную строку `CellList`-компонента, метод `setFocus(true)` наводит фокус.

Метод `setPageSize(int pageSize)` класса `AbstractHasData` устанавливает количество отображаемых строк списка, а метод `setPageStart(int pageStart)` — индекс первой отображаемой строки:

```
public class GWTApplication implements EntryPoint {
    private static final List<String> Items= new ArrayList<String>();
    public void onModuleLoad() {
        for (int i = 0; i < 200; i++) {
            Items.add("Item " + i);
        }
        final TextCell textCell = new TextCell();
        final CellList<String> cellList = new CellList<String>(textCell);
        cellList.setRowData(Items);
        cellList.setPageSize(10);
        final Button button = new Button("Next Page");
        button.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                int start=cellList.getVisibleRange().getStart();
                int pageSize=cellList.getPageSize();
                start=start+pageSize;
                cellList.setRowData(Items);
                cellList.setVisibleRange(start, pageSize);
            }
        });
        final RootPanel panel=RootPanel.get("container");
        panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
        panel.add(cellList,100,50);
        panel.add(button,100,250);
    }
}
```

Приведенный код демонстрирует простейший способ постраничного отображения списка.

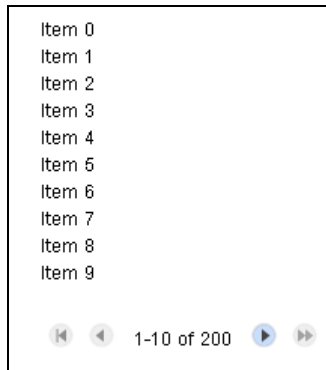
Организовать постраничное отображение списка можно также с помощью компонента `com.google.gwt.user.cellview.client.SimplePager` (рис. 15.8):

```
final TextCell textCell = new TextCell();
final CellList<String> cellList = new CellList<String>(textCell);
ListDataProvider<String> dataProvider = new ListDataProvider<String>();
List<String> data = dataProvider.getList();
for (int i = 0; i < 200; i++) {
    data.add("Item " + i);
}
```

```

dataProvider.addDataDisplay(cellList);
final SimplePager pager = new SimplePager();
pager.setDisplay(cellList);
pager.setPageSize(10);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(cellList,100,50);
panel.add(pager,100,250);

```



**Рис. 15.8.** Постраничное отображение списка с помощью компонента SimplePager

**Метод** `setKeyboardPagingPolicy(HasKeyboardPagingPolicy.KeyboardPagingPolicy policy)` класса `AbstractHasData` позволяет организовать перелистывание страниц списка с помощью клавиатуры:

```

cellList.setPageSize(5);
cellList.setKeyboardPagingPolicy(
    HasKeyboardPagingPolicy.KeyboardPagingPolicy.CHANGE_PAGE);
cellList.addRangeChangeListener(new RangeChangeEvent.Handler () {
    @Override
    public void onRangeChange(RangeChangeEvent event) {
        Window.alert("Range: " + event.getNewRange());
    }
});

```

В случае политики `CHANGE_PAGE` после наведения фокуса на список и использовании клавиши `<↓>` или `<PgDn>`, для того чтобы дойти до конца страницы, срабатывает обработчик `RangeChangeEvent.Handler` и открывается новая страница списка.

В случае политики `INCREASE_RANGE` при достижении конца страницы отображаемый диапазон увеличивается на 30 строк.

**Метод** `setKeyboardSelectionPolicy(HasKeyboardSelectionPolicy.KeyboardSelectionPolicy policy)` класса `AbstractHasData` позволяет связать выбор элементов списка с помощью клавиатуры с установленной моделью выбора `CellList`-компонента:

```

cellList.setKeyboardSelectionPolicy( KeyboardSelectionPolicy.BOUND_TO_SELECTION);
final SingleSelectionModel<String> selectionModel = new SingleSelectionModel<String>();
cellList.setSelectionModel(selectionModel);

```

```
selectionModel.addSelectionChangeHandler(new SelectionChangeEvent.Handler() {
public void onSelectionChange(SelectionChangeEvent event) {
    String selected = selectionModel.getSelectedObject();
    if (selected != null) {
        Window.alert("You selected: " + selected);
    }
}}});
```

В приведенном коде обработчик `SelectionChangeEvent.Handler` срабатывает как при выборе элемента списка мышью, так и при выборе с помощью клавиш `<↑>` и `<↓>`.

Метод `setKeyboardSelectedRow(int row)` класса `AbstractHasData` дает возможность выбрать элемент списка с помощью клавиатуры.

Метод `setKeyboardSelectionHandler(CellPreviewEvent.Handler<T> keyboardSelectionReg)` класса `AbstractHasData` определяет для `CellList`-компонента обработчик, перехватывающий события выбора клавиатурой.

Класс `CellList` также наследует от класса `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

Определить набор данных для `CellList`-компонента можно также с помощью поставщиков данных `ListDataProvider` или `AsyncDataProvider`, обеспечивающих автоматическое обновление набора данных в случае его изменения для `CellList`-компонента:

```
public class GWTApplication implements EntryPoint {
private int index;
public void onModuleLoad() {
    final EditTextCell textCell = new EditTextCell();
    final CellList<String> cellList = new CellList<String>(textCell);
    final ListDataProvider<String> dataProvider = new ListDataProvider<String>();
    List<String> data = dataProvider.getList();
    for (int i = 0; i < 10; i++) {
        data.add("Item " + i);
    }
    dataProvider.addDataDisplay(cellList);
    final SingleSelectionModel<String> selectionModel = new SingleSelectionModel<String>();
    cellList.setSelectionModel(selectionModel);
    selectionModel.addSelectionChangeHandler(new SelectionChangeEvent.Handler() {
        public void onSelectionChange(SelectionChangeEvent event) {
            String selected = selectionModel.getSelectedObject();
            if (selected != null) {
                List<String> list = dataProvider.getList();
                index=list.indexOf(selected);
            }
        }
    });
    ValueUpdater<String> valueUpdater = new ValueUpdater<String>() {
public void update(String newValue) {
    List<String> list = dataProvider.getList();
    list.set(index, newValue);
}
}};
```

```

cellList.setValueUpdater(valueUpdater);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(cellList,100,50);
}}

```

В приведенном коде поставщик данных `ListDataProvider` обеспечивает автоматическое обновление отображаемого набора данных при его редактировании.

**Асинхронный поставщик данных `AsyncDataProvider` обеспечивает получение данных с сервера и заполнение ими страниц `CellList`-компонента:**

```

public class GWTApplication implements EntryPoint {
private int index;
private static class DataProvider extends AsyncDataProvider<String> {
@Override
protected void onRangeChanged(HasData<String> display) {
    final Range range = display.getVisibleRange();
    new Timer() {
        @Override
        public void run() {
            int start = range.getStart();
            int length = range.getLength();
            List<String> newData = new ArrayList<String>();
            for (int i = start; i < start + length; i++) {
                newData.add("Item " + i);
            }
            updateRowData(start, newData);
        }
    }.schedule(3000);
}}
public void onModuleLoad() {
    final EditTextCell textCell = new EditTextCell();
    final CellList<String> cellList = new CellList<String>(textCell);
    cellList.setPageSize(10);
    final DataProvider dataProvider = new DataProvider();
    dataProvider.addDataDisplay(cellList);
    final SimplePager pager = new SimplePager();
    pager.setDisplay(cellList);
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(cellList,100,50);
    panel.add(pager,100,250);
}}

```

**Переопределение метода `onBrowserEvent(Cell.Context context, Element parent, java.lang.String value, NativeEvent event, ValueUpdater<java.lang.String> valueUpdater)` `Cell`-ячейки** дает возможность организовать обработку событий элементов `Cell`-компонента:

```

EditTextCell textCell = new EditTextCell(){
@Override
public void onBrowserEvent(com.google.gwt.cell.client.Cell.Context context,
com.google.gwt.dom.client.Element parent, String value, NativeEvent event,
ValueUpdater<String> valueUpdater) {
    super.onBrowserEvent(context, parent, value, event, valueUpdater);
    if(event.getType().equals("blur")){
        Window.alert("Blur Cell");
    }
}};

```

Для отображения набора данных пользовательского Java-типа необходимо создать пользовательскую Cell-ячейку:

```

private static class ApplicationData {
    private final String str;
    private final int id;
    public ApplicationData(String str, int id) {
        this.str = str;
        this.id = id;
    }
}
private static class ApplicationCell extends AbstractCell<ApplicationData> { @Override
    public void render(Context context, ApplicationData object,
        SafeHtmlBuilder sb) {
        if (object != null) {
            sb.appendEscaped(object.str);
        }
    }
}

```

Компонент ValuePicker<T> обертывает список CellList<T> с возможностью обработки выбора элемента списка:

```

final TextCell textCell = new TextCell();
final CellList<String> cellList = new CellList<String>(textCell);
final ListDataProvider<String> dataProvider = new ListDataProvider<String>();
List<String> data = dataProvider.getList();
for (int i = 0; i < 10; i++) {
    data.add("Item " + i);
}
dataProvider.addDataDisplay(cellList);
ValuePicker<String> vp=new ValuePicker<String>(cellList);
vp.addValueChangeHandler(new ValueChangeHandler<String>() {
    public void onValueChange(ValueChangeEvent<String> event) {
        Window.alert(event.getValue());
    }
});

```

## Таблица CellTable

Компонент CellTable обеспечивает отображение таблицы данных и является составным компонентом, который создается на основе колонок с заголовками. В свою очередь колонки таблицы создаются на основе Cell-ячеек.

Так как каждая колонка таблицы связана со своим набором данных, модель данных для всей таблицы удобно создавать в виде класса, каждое поле которого представляет набор данных для отдельной колонки таблицы.

Компонент `CellTable` создается с помощью конструктора класса без аргументов или конструктора, принимающего в качестве аргумента число отображаемых строк страницы таблицы, объект `CellTable.Resources`, поставщика ключей `ProvidesKey` и `Widget`-компонент, отображаемый при загрузке таблицы.

Класс `CellTable` расширяет классы `AbstractCellTable<T>` и `AbstractHasData<T>` пакета `com.google.gwt.user.cellview.client`, а также классы `Composite`, `Widget` и `UIObject` пакета `com.google.gwt.user.client.ui`.

По умолчанию CSS-стиль `CellTable`-компонента определяется файлом `CellTable.css` пакета `com.google.gwt.user.cellview.client` библиотеки `gwt-user.jar` фреймворка GWT. С помощью интерфейса `CellTable.Resources` можно переопределить CSS-стиль `CellTable`-компонента по умолчанию (см. разд. "Столбец `CellList`" ранее в этой главе).

Интерфейс `ProvidesKey` обеспечивает ключи для элементов набора данных и используется совместно с моделью выбора элементов набора данных (см. разд. "Столбец `CellList`" ранее в этой главе).

Колонка таблицы создается как экземпляр класса `com.google.gwt.user.cellview.client.Column<T,C>`, где `T` — Java-тип класса модели данных, а `C` — Java-тип поля класса модели данных. При этом `Column`-объект создается с помощью конструктора класса `Column(Cell<C> cell)` с переопределением метода:

```
public abstract C getValue(T object)
```

в котором устанавливается возврат значения поля класса модели данных таблицы:

```
private static class ApplicationData {
    private final String str;
    private final Date date;
    private final int id;
    public ApplicationData(String str, Date date, int id) {
        this.str = str;
        this.date = date;
        this.id = id;
    }
}

Column<ApplicationData, Date> dateColumn = new Column<ApplicationData, Date>(dateCell) {
    @Override
    public Date getValue(ApplicationData object) {
        return object.date;
    }
};
```

После создания колонок они добавляются в `CellTable`-компонент с помощью метода `addColumn(Column<T,?> col, java.lang.String headerString)` класса `AbstractCellTable`, формируя таблицу:

```
public class GWTApplication implements EntryPoint {
    private static class ApplicationData {
        private final String str;
```



```

private final Date date;
private final int id;
public ApplicationData(String str, Date date, int id) {
    this.str = str;
    this.date = date;
    this.id = id;
}
}
public void onModuleLoad() {
    CellTable<ApplicationData> table = new CellTable<ApplicationData>();
    TextColumn<ApplicationData> strColumn =
        new TextColumn<ApplicationData>() {
            @Override
            public String getValue(ApplicationData object) {
                return object.str;
            }
        };
    DateCell dateCell = new DateCell();
    Column<ApplicationData, Date> dateColumn = new Column<ApplicationData, Date>(dateCell) {
        @Override
        public Date getValue(ApplicationData object) {
            return object.date;
        }
    };
    NumberCell idCell = new NumberCell();
    Column<ApplicationData, Number> idColumn = new Column<ApplicationData, Number>(idCell) {
        @Override
        public Number getValue(ApplicationData object) {
            return object.id;
        }
    };
    table.addColumn(strColumn, "String");
    table.addColumn(dateColumn, "Date");
    table.addColumn(idColumn, "Id");
    final List<ApplicationData> DATA = Arrays.asList(new ApplicationData("A Item", new
    Date(), 1), new ApplicationData("B Item", new Date(), 2), new ApplicationData("C Item",
    new Date(), 3));
    ListDataProvider<ApplicationData> dataProvider = new
    ListDataProvider<ApplicationData>();
    dataProvider.addDataDisplay(table);
    final List<ApplicationData> list = dataProvider.getList();
    for (ApplicationData data : DATA) {
        list.add(data);
    }
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(table,100,50);
}
}

```

**С помощью метода** `addColumn(Column<T,?> col, java.lang.String headerString, java.lang.String footerString)` **класса** `AbstractCellTable` **для колонки таблицы можно определить не только заголовок, но и заключение (footer).**

Заголовок и заключение колонки таблицы можно определить не только в виде строки, но и как Cell-ячейку, используя метод `addColumn(Column<T,?> col, Header<?> header, Header<?> footer)` класса `AbstractCellTable`. При этом объект `Header` создается с помощью конструктора `Header(Cell<H> cell)` класса, который имеет методы:

- ◆ `onBrowserEvent(Cell.Context context, Element elem, NativeEvent event)` — метод для переопределения в конструкторе класса с целью обработки событий браузера;
- ◆ `onPreviewColumnSortEvent(Cell.Context context, Element elem, NativeEvent event)` — переопределение метода позволяет перехватывать события сортировки колонки;
- ◆ `setHeaderStyleNames(java.lang.String styleNames)` — устанавливает CSS-стиль;
- ◆ `setUpdater(ValueUpdater<H> updater)` — определяет обработчик события изменения содержимого ячейки.

Методы `setVerticalAlignment(HasVerticalAlignment.VerticalAlignmentConstant align)` и `setHorizontalAlignment(HasHorizontalAlignment.HorizontalAlignmentConstant align)` класса `Column` позволяют определить вертикальное и горизонтальное выравнивания ячеек в колонке:

```
column.setVerticalAlignment(HasAlignment.ALIGN_BOTTOM);
column.setHorizontalAlignment(HasAlignment.ALIGN_RIGHT);
```

Метод `setSortable(boolean sortable)` класса `Column` включает опцию сортировки для колонки таблицы, при этом для самой таблицы необходимо определить обработчик сортировки, используя метод `addColumnSortHandler(ColumnSortEvent.Handler handler)` класса `AbstractCellTable`:

```
strColumn.setSortable(true);
idColumn.setSortable(true);
ColumnSortEvent.ListHandler<ApplicationData> columnSortHandler = new
ColumnSortEvent.ListHandler<ApplicationData>(list);
columnSortHandler.setComparator(strColumn, new Comparator<ApplicationData>() {
public int compare(ApplicationData o1, ApplicationData o2) {
    if (o1 == o2) {
        return 0;
    }
    if (o1 != null) {
        return (o2 != null) ? o1.str.compareTo(o2.str) : 1;
    }
    return -1;
}
});
columnSortHandler.setComparator(idColumn, new Comparator<ApplicationData>() {
public int compare(ApplicationData o1, ApplicationData o2) {
    if (o1 == o2) {
        return 0;
    }
    if (o1 != null) {
        return (o2 != null) ? new Integer(o1.id).compareTo(o2.id) : 1;
    }
}
```

```

        return -1;
    }
});
table.addColumnSortHandler(columnSortHandler);

```

При включенной опции сортировки для колонки таблицы и при определенном обработчике сортировки щелчок заголовка колонки активирует сортировку ее элементов (рис. 15.9).



▼ String	Date	Id
C Item	Sunday, 2012 August 26	3
B Item	Sunday, 2012 August 26	2
A Item	Sunday, 2012 August 26	1

Рис. 15.9. Таблица CellTable с включенной опцией сортировки

Метод `setDefaultSortAscending(boolean isAscending)` класса `Column` меняет порядок сортировки элементов колонки таблицы.

Метод `setFieldUpdater(FieldUpdater<T,C> fieldUpdater)` класса `Column` позволяет определить для колонки таблицы обработчик событий изменения содержимого ячеек колонки:

```

final CellTable<ApplicationData> table = new CellTable<ApplicationData>();
EditTextCell textCell=new EditTextCell();
Column<ApplicationData, String> strColumn = new Column<ApplicationData,
String>(textCell) {
    @Override
    public String getValue(ApplicationData object) {
        return object.str;
    }
};
strColumn.setFieldUpdater(new FieldUpdater<ApplicationData, String>() {
    @Override
    public void update(int index, ApplicationData object, String value) {
        Window.alert("Update");
        object.str = value;
        table.redraw();
    }
});

```

В приведенном коде обработчик `FieldUpdater` срабатывает по окончании редактирования `EditTextCell`-ячеек колонки таблицы.

Метод `setCellStyleNames(java.lang.String styleNames)` класса `Column` устанавливает CSS-стиль колонки таблицы, а переопределение метода `onBrowserEvent(Cell.Context`

context, Element elem, T object, NativeEvent event) обеспечивает обработку событий браузера.

В целом для таблицы метод `addColumnStyleName(int index, java.lang.String styleName)` класса `CellTable` добавляет CSS-стиль для колонки с указанным порядковым номером, переопределение методов `onTableBodyChange(TableSectionElement newTBody)`, `onTableFootChange(TableSectionElement newTFoot)` и `onTableHeadChange(TableSectionElement newTHead)` позволяет обрабатывать события изменения элементов таблицы, метод `setEmptyTableWidget(Widget widget)` устанавливает Widget-компонент, отображаемый в случае пустой таблицы, метод `setLoadingIndicator(Widget widget)` устанавливает Widget-компонент, отображаемый при загрузке таблицы.

Комбинация методов `setWidth(java.lang.String width, boolean isFixedLayout)` и `setColumnWidth(Column<T,?> column, double width, Style.Unit unit)` класса `CellTable` позволяет регулировать ширину таблицы и ее колонок.

Таблица с фиксированной шириной и относительными размерами колонок:

```
table.setWidth("250px", true);
table.setColumnWidth(strColumn, 30.0, Unit.PCT);
table.setColumnWidth(dateColumn, 50.0, Unit.PCT);
table.setColumnWidth(idColumn, 20.0, Unit.PCT);
```

или

```
table.setWidth("100%", true);
table.setColumnWidth(strColumn, 30.0, Unit.PCT);
table.setColumnWidth(dateColumn, 50.0, Unit.PCT);
table.setColumnWidth(idColumn, 20.0, Unit.PCT);
```

Таблица с фиксированными размерами колонок:

```
table.setWidth("auto", true);
table.setColumnWidth(strColumn, 150.0, Unit.PX);
table.setColumnWidth(dateColumn, 200.0, Unit.PX);
table.setColumnWidth(idColumn, 150.0, Unit.PX);
```

Метод `addRowHoverHandler(RowHoverEvent.Handler handler)` класса `AbstractCellTable` определяет для таблицы обработчик события движения курсора мыши по строкам таблицы:

```
table.addRowHoverHandler(new RowHoverEvent.Handler(){
    public void onRowHover(RowHoverEvent event) {
        Window.alert("Hover");
    }
});
```

Метод `clearColumnWidth(Column<T,?> column)` класса `AbstractCellTable` сбрасывает установку ширины колонки таблицы, метод `flush()` сбрасывает все незаконченные изменения таблицы с ее немедленным отображением, метод `getColumn(int col)` возвращает объект `Column` по его порядковому номеру, метод `getColumnCount()` возвращает число колонок таблицы, метод `getColumnIndex(Column<T,?> column)` возвращает порядковый номер колонки, метод `getColumnWidth(Column<T,?> column)` возвращает

установленную ширину колонки, метод `getKeyboardSelectedColumn()` возвращает порядковый номер выбранной колонки, метод `getKeyboardSelectedSubRow()` возвращает порядковый номер выбранной подстроки, методы `redrawFooters()` и `redrawHeaders()` перерисовывают заголовок и футер колонки, методы `setAutoFooterRefreshDisabled(boolean disabled)` и `setAutoHeaderRefreshDisabled(boolean disabled)` включают или отключают автоматическое обновление заголовка и футера при изменении содержимого ячейки.

Метод `insertColumn(int beforeIndex, Column<T,?> col, java.lang.String headerString, java.lang.String footerString)` класса `AbstractCellTable` позволяет вставить колонку в таблицу перед колонкой с определенным номером, метод `removeColumn(Column<T,?> col)` или метод `removeColumn(int index)` удаляет колонку таблицы.

Методы `setKeyboardSelectedColumn(int column)` и `setKeyboardSelectedRow(int row, int subrow, boolean stealFocus)` класса `AbstractCellTable` позволяют программным способом выбрать колонку и строку таблицы.

Метод `setTableBuilder(CellTableBuilder<T> tableBuilder)` класса `AbstractCellTable` устанавливает объект `com.google.gwt.user.cellview.client.CellTableBuilder`, отвечающий за отображение строк таблицы. Интерфейс `CellTableBuilder` имеет реализацию в виде класса `DefaultCellTableBuilder<T>`, метод `buildRowImpl(T rowValue, int absRowIndex)` которого позволяет создавать строки таблицы.

Класс `CellTable` также наследует от класса `AbstractHasData` методы, определяющие различные обработчики и свойства таблицы (см. разд. "Столбец `CellList`" ранее в этой главе). От класса `UIObject` класс `CellTable` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Таблица *DataGrid*

Отличие таблицы `DataGrid` от таблицы `CellTable` заключается в том, что при установленной высоте таблицы `DataGrid`, недостаточной для полного отображения данных, между заголовками и футерами колонок таблицы появляется полоса прокрутки (рис. 15.10). Кроме того, таблицу `DataGrid` необходимо добавлять в `Layout`-панель:

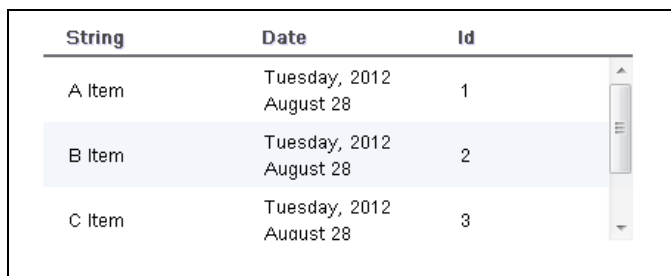
```
public class GWTApplication implements EntryPoint {
    private static class ApplicationData {
        private final String str;
        private final Date date;
        private final int id;
        public ApplicationData(String str, Date date, int id) {
            this.str = str;
            this.date = date;
            this.id = id;
        }
    }

    public void onModuleLoad() {
        DataGrid<ApplicationData> table = new DataGrid<ApplicationData>();
        TextColumn<ApplicationData> strColumn = new TextColumn<ApplicationData>() {
            @Override
```

```

    public String getValue(ApplicationData object) {
        return object.str;
    };
};
DateCell dateCell = new DateCell();
Column<ApplicationData, Date> dateColumn = new Column<ApplicationData, Date>(dateCell) {
    @Override
    public Date getValue(ApplicationData object) {
        return object.date;
    };
};
NumberCell idCell = new NumberCell();
Column<ApplicationData, Number> idColumn = new Column<ApplicationData, Number>(idCell) {
    @Override
    public Number getValue(ApplicationData object) {
        return object.id;
    };
};
table.addColumn(strColumn, "String");
table.addColumn(dateColumn, "Date");
table.addColumn(idColumn, "Id");
final List<ApplicationData> DATA = Arrays.asList(new ApplicationData("A Item", new
Date(), 1),
    new ApplicationData("B Item", new Date(), 2),
    new ApplicationData("C Item", new Date(), 3),
    new ApplicationData("D Item", new Date(), 4) );
ListDataProvider<ApplicationData> dataProvider = new
ListDataProvider<ApplicationData>();
dataProvider.addDataDisplay(table);
final List<ApplicationData> list = dataProvider.getList();
for (ApplicationData data : DATA) {
    list.add(data);
}
DockLayoutPanel p = new DockLayoutPanel(Unit.PX);
p.add(table);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(p,200,50);
p.setSize("400px", "150px");
//table.setHeight("150px");
}
}

```



String	Date	Id
A Item	Tuesday, 2012 August 28	1
B Item	Tuesday, 2012 August 28	2
C Item	Tuesday, 2012 August 28	3

Рис. 15.10. Таблица DataGrid

## Дерево *CellTree*

Компонент *CellTree* представляет иерархию узлов и создается с помощью конструктора класса, принимающего в качестве аргумента объект `com.google.gwt.view.client.TreeViewModel`, скрытый корневой узел дерева и объект `com.google.gwt.user.cellview.client.CellTree.Resources`.

Класс *CellTree* расширяет класс *AbstractCellTree* пакета `com.google.gwt.user.cellview.client`, а также классы *Composite*, *Widget* и *UIObject* пакета `com.google.gwt.user.client.ui`.

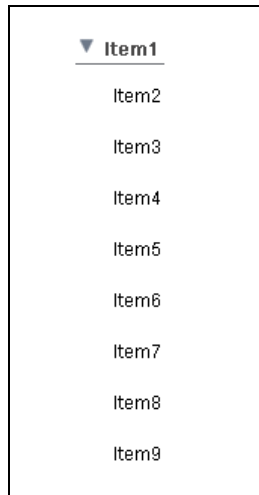
По умолчанию CSS-стиль *CellTree*-компонента определяется файлом *CellTree.css* пакета `com.google.gwt.user.cellview.client` библиотеки *gwt-user.jar* фреймворка GWT. С помощью интерфейса *CellTree.Resources* можно переопределить CSS-стиль *CellTree*-компонента по умолчанию (см. разд. "Столбец *CellList*" ранее в этой главе).

Объект *TreeViewModel* описывает дерево как набор узлов, которые могут иметь дочерние узлы, и узлов-листьев, которые не могут иметь дочерние узлы. Объект *TreeViewModel* создается как экземпляр класса, реализующего интерфейс *TreeViewModel* с определением его методов `<T> TreeViewModel.NodeInfo<?> getNodeInfo(T value)` и `boolean isLeaf(java.lang.Object value)`:

```
public class GWTApplication implements EntryPoint {
    private static class ApplicationTreeModel implements TreeViewModel {
        public <T> NodeInfo<?> getNodeInfo(T value) {
            if(value.toString().equals("Item0")){
                ListDataProvider<String> dataProvider = new ListDataProvider<String>();
                List<String> data = dataProvider.getList();
                data.add("Item1");
                return new DefaultNodeInfo<String>(dataProvider, new TextCell());
            }else if(value.toString().equals("Item1")){
                ListDataProvider<String> dataProvider = new ListDataProvider<String>();
                List<String> data = dataProvider.getList();
                for (int i = 2; i < 10; i++) {
                    data.add("Item" + i);
                }
                return new DefaultNodeInfo<String>(dataProvider, new TextCell());
            }else{
                ListDataProvider<String> dataProvider = new ListDataProvider<String>();
                return new DefaultNodeInfo<String>(dataProvider, new TextCell());
            }
        }
        public boolean isLeaf(Object value) {
            boolean flag=true;
            String temp=value.toString().substring(4, value.toString().length());
            int i=Integer.parseInt(temp);
            if(i == 0) flag=false;
            if(i == 1) flag=false;
            return flag;
        }
    }
}
```

```
public void onModuleLoad() {
    TreeViewModel model = new ApplicationTreeModel();
    final CellTree tree = new CellTree(model, "Item0");
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(tree,200,10);
}
```

В приведенном коде класс `ApplicationTreeModel` описывает дерево, корневой узел `Item0` которого имеет дочерний узел `Item1`. Узел `Item1` в свою очередь имеет набор узлов-листьев (рис. 15.11).



**Рис. 15.11.** Дерево, описываемое классом `ApplicationTreeModel`

Метод `getNodeInfo()` интерфейса `TreeViewModel` возвращает объект `TreeViewModel.NodeInfo<T>`, обеспечивающий информацию о дочерних узлах. Интерфейс `TreeViewModel.NodeInfo<T>` имеет реализацию в виде класса `TreeViewModel.DefaultNodeInfo<T>`, экземпляр которого создается с помощью конструктора класса, принимающего в качестве аргумента объект поставщика данных `AbstractDataProvider<T>`, `Cell`-ячейку, отвечающую за отображение данных, модель выбора `SelectionModel` и обработчик изменения содержимого ячейки `ValueUpdater<T>`.

Метод `isLeaf()` интерфейса `TreeViewModel` обеспечивает информацию о типе узла дерева — может ли узел иметь дочерние узлы, или узел является листом.

Переопределение метода `onBrowserEvent(Event event)` класса `CellTree` обеспечивает обработку событий браузера для `CellTree`-компонента.

Метод `getRootTreeNode()` класса `CellTree` возвращает объект `TreeNode`, представляющий корневой узел дерева. Метод `setChildOpen(int index, boolean open)` интерфейса `TreeNode` позволяет программным способом раскрывать узел дерева, имеющий дочерние узлы.

Методы `addCloseHandler(CloseHandler<TreeNode> handler)` и `addOpenHandler(OpenHandler<TreeNode> handler)` класса `AbstractCellTree` определяют для `CellTree`-компонента об-



работчики закрытия и раскрытия узла дерева, а метод `setKeyboardSelectionPolicy` (`HasKeyboardSelectionPolicy.KeyboardSelectionPolicy policy`) позволяет связать выбор узлов дерева с помощью клавиатуры с установленной моделью выбора `CellTree`-компонента.

От класса `UIObject` класс `CellTree` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Дерево `CellBrowser`

Отличие дерева `CellBrowser` от дерева `CellTree` состоит в том, что дерево `CellBrowser` отображается в виде двух панелей: в левой панели демонстрируется само дерево, а в правой панели — дочерние узлы при раскрытии узла левой панели. При этом, если установленная высота компонента недостаточна для полного отображения данных, в правой панели появляется полоса прокрутки (рис. 15.12). Кроме того, дерево `CellBrowser` необходимо добавлять в `Layout`-панель:

```
public class GWTApplication implements EntryPoint {
    private static class ApplicationTreeModel implements TreeViewModel {
        public <T> NodeInfo<?> getNodeInfo(T value) {
            if(value.toString().equals("Item0")){
                ListDataProvider<String> dataProvider = new ListDataProvider<String>();
                List<String> data = dataProvider.getList();
                data.add("Item1");
                return new DefaultNodeInfo<String>(dataProvider, new TextCell());
            }else if(value.toString().equals("Item1")){
                ListDataProvider<String> dataProvider = new ListDataProvider<String>();
                List<String> data = dataProvider.getList();
                for (int i = 2; i < 10; i++) {
                    data.add("Item" + i);
                }
                return new DefaultNodeInfo<String>(dataProvider, new TextCell());
            }else{
                ListDataProvider<String> dataProvider = new ListDataProvider<String>();
                return new DefaultNodeInfo<String>(dataProvider, new TextCell());
            }
        }
        public boolean isLeaf(Object value) {
            boolean flag=true;
            String temp=value.toString().substring(4, value.toString().length());
            int i=Integer.parseInt(temp);
            if(i == 0) flag=false;
            if(i == 1) flag=false;
            return flag;
        }
    }
    public void onModuleLoad() {
        TreeViewModel model = new ApplicationTreeModel();
        final CellBrowser tree = new CellBrowser.Builder<String>(model, "Item0").build();
        tree.setDefaultColumnWidth(100);
    }
}
```

```

DockLayoutPanel p = new DockLayoutPanel(Unit.PX);
p.add(tree);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(p,200,50);
p.setSize("400px", "150px");
}}

```

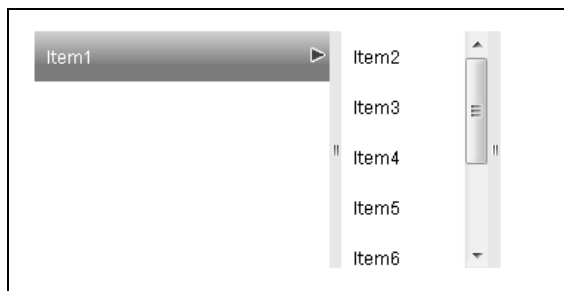


Рис. 15.12. Дерево CellBrowser

Компонент `CellBrowser` создается с помощью класса-фабрики `CellBrowser.Builder<T>`.

Экземпляр класса `CellBrowser.Builder<T>` создается посредством конструктора класса, принимающего в качестве аргумента объект `com.google.gwt.view.client.TreeViewModel` и скрытый корневой узел дерева.

Метод `build()` класса `CellBrowser.Builder` создает компонент `CellBrowser`, метод `loadingIndicator(Widget widget)` определяет `Widget`-компонент, отображаемый при загрузке дерева, метод `pageSize(int pageSize)` устанавливает количество отображаемых узлов дерева на одной странице, метод `resources(CellBrowser.Resources resources)` определяет объект `CellBrowser.Resources`, с помощью которого можно переопределить CSS-стиль `CellBrowser`-компонента по умолчанию.

С помощью методов `setDefaultColumnWidth(int width)` и `setMinimumColumnWidth(int minWidth)` класса `CellBrowser` можно регулировать ширину правой панели компонента.

## Панель меню *MenuBar*

Компонент `MenuBar` представляет панель, содержащую элементы меню, с помощью которых можно активировать различные команды или открывать другие панели меню.

Компонент `MenuBar` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента флаг, указывающий ориентацию меню (по умолчанию создается горизонтальная панель меню).

Класс `MenuBar` расширяет классы `Widget` и `UIObject`.

Для MenuBar-компонента по умолчанию определены CSS-классы:

- ◆ .gwt-MenuBar — общий стиль компонента;
- ◆ .gwt-MenuBar-horizontal — стиль горизонтального меню;
- ◆ .gwt-MenuBar-vertical — стиль вертикального меню;
- ◆ .gwt-MenuBar .gwt-MenuItem — стиль элементов меню;
- ◆ .gwt-MenuBar .gwt-MenuItem-selected — стиль выбранного элемента меню;
- ◆ .gwt-MenuBar .gwt-MenuItemSeparator — стиль разделителя элементов меню;
- ◆ .gwt-MenuBar .gwt-MenuItemSeparator .menuSeparatorInner — стиль внутреннего компонента разделителя элементов меню;
- ◆ .gwt-MenuBarPopup, .gwt-MenuBarPopup .menuPopupTopLeft, .gwt-MenuBarPopup .menuPopupTopLeftInner, .gwt-MenuBarPopup .menuPopupTopCenter, .gwt-MenuBarPopup .menuPopupTopCenterInner, .gwt-MenuBarPopup .menuPopupTopRight, .gwt-MenuBarPopup .menuPopupTopRightInner, .gwt-MenuBarPopup .menuPopupMiddleLeft, .gwt-MenuBarPopup .menuPopupMiddleLeftInner, .gwt-MenuBarPopup .menuPopupMiddleCenter, .gwt-MenuBarPopup .menuPopupMiddleCenterInner, .gwt-MenuBarPopup .menuPopupMiddleRight, .gwt-MenuBarPopup .menuPopupMiddleRightInner, .gwt-MenuBarPopup .menuPopupBottomLeft, .gwt-MenuBarPopup .menuPopupBottomLeftInner, .gwt-MenuBarPopup .menuPopupBottomCenter, .gwt-MenuBarPopup .menuPopupBottomCenterInner, .gwt-MenuBarPopup .menuPopupBottomRight, .gwt-MenuBarPopup .menuPopupBottomRightInner — стили подменю.

Панель MenuBar наполняется элементами меню с помощью метода `addItem()`, принимающего в качестве аргумента объект `com.google.gwt.user.client.ui.MenuItem`, представляющий элемент меню, или принимающего все те аргументы, которые может принимать конструктор класса `MenuItem`:

- ◆ `MenuItem(SafeHtml html, MenuBar subMenu), MenuItem(java.lang.String text, boolean asHTML, MenuBar subMenu), MenuItem(java.lang.String text, MenuBar subMenu)` — создается элемент меню, открывающий подменю (рис. 15.13):

```
Command command = new Command()
{ public void execute() {
    Window.alert("Command Fired");
}};
MenuBar menuMain = new MenuBar();
MenuBar menuSub = new MenuBar();
MenuItem item1 = new MenuItem("One", command);
MenuItem item2 = new MenuItem("Two", command);
menuMain.addItem("SubMenu", menuSub);
menuSub.addItem(item1);
menuSub.addItem(item2);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(menuMain,200,50);
```

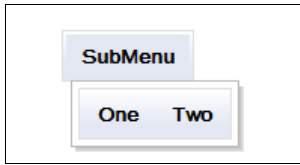


Рис. 15.13. Меню  
с подменю

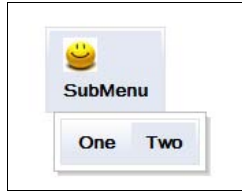


Рис. 15.14. Подменю  
со значком



Рис. 15.15. Меню  
с разделителем

или со значком (рис. 15.14):

```
Command command = new Command()
{ public void execute() {
    Window.alert("Command Fired");
}};
MenuBar menuMain = new MenuBar();
MenuBar menuSub = new MenuBar();
MenuItem item1 = new MenuItem("One", command);
MenuItem item2 = new MenuItem("Two", command);
final String labelSub = "<img src='/images/image.jpg' height='25px'
width='25px' />SubMenu";
menuMain.addItem(labelSub, true, menuSub);
menuSub.addItem(item1);
menuSub.addItem(item2);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(menuMain,200,50);
```

- ◆ MenuItem(SafeHtml html, Scheduler.ScheduledCommand cmd), MenuItem(java.lang.String text, boolean asHTML, Scheduler.ScheduledCommand cmd), MenuItem(java.lang.String text, Scheduler.ScheduledCommand cmd) — создается элемент меню для вызова команды.

Команда элемента меню создается путем реализации интерфейса

com.google.gwt.core.client.Command с определением его метода execute().

Метод addCloseHandler(CloseHandler<PopupPanel> handler) класса MenuBar определяет для главной панели меню обработчик события закрытия панели подменю:

```
menuMain.addCloseHandler(new CloseHandler<PopupPanel>() {
    public void onClose(CloseEvent<PopupPanel> event) {
        Window.alert("Close");
    }
});
```

Методы addSeparator() и addSeparator(MenuItemSeparator separator) класса MenuBar позволяют добавить в панель разделитель элементов меню. Разделители добавляются в панель меню в том же самом порядке, в каком метод addSeparator() применяется к объекту MenuBar (рис. 15.15):

```
menuMain.addItem("Item", false, command);
menuMain.addSeparator();
menuMain.addItem(labelSub, true, menuSub);
```

Объект `MenuItemSeparator` создается с помощью конструктора `MenuItemSeparator()` класса, наследующего от класса `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

Метод `clearItems()` класса `MenuBar` удаляет все элементы панели меню, метод `closeAllChildren(boolean focus)` закрывает меню и все подменю, метод `focus()` наводит на панель фокус, метод `getItemIndex(MenuItem item)` возвращает порядковый номер элемента меню, метод `getSeparatorIndex(MenuItemSeparator item)` возвращает порядковый номер разделителя элементов меню, методы `insertItem(MenuItem item, int beforeIndex)`, `insertSeparator(int beforeIndex)` и `insertSeparator(MenuItemSeparator separator, int beforeIndex)` позволяют вставлять элемент и разделитель в определенное место панели, а методы `removeItem(MenuItem item)` и `removeSeparator(MenuItemSeparator separator)` удаляют элемент и разделитель, методы `selectItem(MenuItem item)`, `moveSelectionDown()` и `moveSelectionUp()` позволяют программным способом выбирать и перемещать выбор элементов меню, переопределение метода `onBrowserEvent(Event event)` дает возможность обработки событий браузера, метод `setAnimationEnabled(false)` отключает плавное открытие подменю, метод `setAutoOpen(true)` включает открытие подменю при наведении курсора мыши.

От класса `UIObject` класс `MenuBar` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

Метод `setEnabled(boolean enabled)` класса `MenuItem` деактивирует элемент меню, методы `setText(java.lang.String text)`, `setHTML(SafeHtml html)` и `setHTML(java.lang.String html)` устанавливают надпись элемента меню, метод `setScheduledCommand(Scheduler.ScheduledCommand cmd)` определяет команду элемента меню, метод `setSubMenu(MenuBar subMenu)` определяет подменю.

От класса `UIObject` класс `MenuItem` также наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Дерево *Tree*

Компонент `Tree` позволяет создавать дерево элементов, представленных `Widget`-компонентами, и создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента объект `com.google.gwt.user.client.ui.Tree.Resources`, позволяющий определить значки для элементов дерева (рис. 15.16):

```
public class GWTApplication implements EntryPoint {
    interface ApplicationResources extends Tree.Resources {
        @Source("image.jpg")
        ImageResource treeOpen();
        @Source("image.jpg")
        ImageResource treeClosed();
        @Source("image.jpg")
        ImageResource treeLeaf();
    }
}
```

```

public void onModuleLoad() {
    Tree.Resources images = (Tree.Resources)GWT.create(ApplicationResources.class);
    Tree tree = new Tree(images, true);
    TreeItem root = new TreeItem();
    root.setText("root");
    root.addTextItem("item0");
    root.addTextItem("item1");
    root.addTextItem("item2");
    tree.addItem(root);
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(tree,200,50);
}
}

```

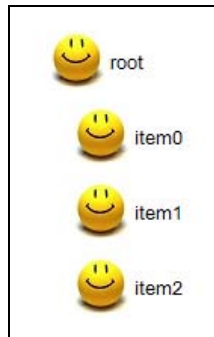


Рис. 15.16. Дерево со значками

Класс `Tree` расширяет классы `Widget` и `UIObject`.

Для компонента `Tree` по умолчанию определены CSS-классы:

- ◆ `.gwt-Tree` — общий стиль компонента;
- ◆ `.gwt-Tree .gwt-TreeItem` — стиль элемента дерева;
- ◆ `.gwt-Tree .gwt-TreeItem-selected` — стиль выбранного элемента дерева.

Дерево `Tree` наполняется элементами с помощью методов:

- ◆ `add(Widget widget)`, `addItem(SafeHtml itemHtml)`, `addItem(TreeItem item)`, `addItem(Widget widget)` — определяют корневой элемент дерева;
- ◆ `insertItem(int beforeIndex, SafeHtml itemHtml)`, `insertItem(int beforeIndex, TreeItem item)`, `insertItem(int beforeIndex, Widget widget)`, `insertTextItem(int beforeIndex, java.lang.String itemText)` — вставляют корневой элемент дерева;
- ◆ `remove(Widget w)`, `removeItem(TreeItem item)`, `removeItems()`, `clear()` — удаляют корневые элементы дерева.

Объект `TreeItem`, представляющий элемент дерева, создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента флаг, указывающий, является ли элемент корневым, HTML-код надписи элемента дерева, `Widget`-компонент.

Методы `addItem(SafeHtml itemHtml)`, `addItem(TreeItem item)`, `addItem(Widget widget)`, `addTextItem(java.lang.String itemText)`, `insertItem(int beforeIndex, SafeHtml itemHtml)`, `insertItem(int beforeIndex, TreeItem item)`, `insertItem(int beforeIndex, Widget widget)`, `insertTextItem(int beforeIndex, java.lang.String itemText)`, `remove()`, `removeItem(TreeItem item)`, `removeItems()` класса `TreeItem` позволяют добавить, вставить или удалить дочерние элементы для данного элемента дерева.

Методы `setHTML(SafeHtml html)`, `setHTML(java.lang.String html)`, `setText(java.lang.String text)` класса `TreeItem` определяют надпись элемента дерева.

Метод `isSelected()` класса `TreeItem` позволяет проверить, выбран ли данный элемент дерева, метод `getChild(int index)` возвращает дочерний элемент, а методы `getChildCount()` и `getChildIndex(TreeItem child)` возвращают общее количество дочерних элементов и порядковый номер дочернего элемента.

От класса `UIObject` класс `TreeItem` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

Метод `addBlurHandler(BlurHandler handler)` класса `Tree` определяет для дерева обработчик события потери фокуса, методы `addCloseHandler(CloseHandler<TreeItem> handler)` и `addOpenHandler(OpenHandler<TreeItem> handler)` — обработчики сворачивания и раскрытия иерархии элементов, метод `addFocusHandler(FocusHandler handler)` — обработчик наведения фокуса, методы `addKeyDownHandler(KeyDownHandler handler)`, `addKeyPressHandler(KeyPressHandler handler)`, `addKeyUpHandler(KeyUpHandler handler)`, `addMouseDownHandler(MouseDownHandler handler)`, `addMouseMoveHandler(MouseMoveHandler handler)`, `addMouseOutHandler(MouseOutHandler handler)`, `addMouseOverHandler(MouseOverHandler handler)`, `addMouseUpHandler(MouseUpHandler handler)`, `addMouseWheelHandler(MouseWheelHandler handler)` — обработчики клавиатуры и мыши.

Метод `addSelectionHandler(SelectionHandler<TreeItem> handler)` класса `Tree` определяет для дерева обработчик выбора элемента дерева.

Метод `getItem(int index)` класса `Tree` возвращает корневой элемент дерева, метод `getItemCount()` возвращает количество корневых элементов дерева, метод `getSelectedItem()` возвращает выбранный элемент, переопределение метода `onBrowserEvent(Event event)` позволяет обрабатывать события браузера, метод `setAnimationEnabled(false)` выключает плавное разворачивание иерархии элементов, метод `setSelectedItem(TreeItem item)` программным способом выбирает элемент дерева, метод `treeItemIterator()` возвращает итератор корневых элементов дерева.

От класса `UIObject` класс `Tree` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Поле подсказки *SuggestBox*

Компонент `SuggestBox` представляет собой текстовое поле, при вводе в которое отображается список подсказок.

Компонент `SuggestBox` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента

объект `com.google.gwt.user.client.ui.SuggestOracle` и объект `com.google.gwt.user.client.ui.TextBoxBase`.

Класс `SuggestBox` расширяет классы `Composite`, `Widget` и `UIObject`.

Для компонента `SuggestBox` по умолчанию определен CSS-класс `.gwt-SuggestBox`.

Если компонент `SuggestBox` создается с помощью конструктора класса без аргументов, тогда он создается на основе объектов `MultiWordSuggestOracle` и `TextBox`, являющихся реализациями абстрактных классов `SuggestOracle` и `TextBoxBase`.

Объект `MultiWordSuggestOracle` обеспечивает хранение подсказок и возврат отсортированного списка подсказок, основываясь на строках запросов, которые формируются при вводе текстовой строки в поле `TextBox`.

Объект `MultiWordSuggestOracle` возвращается методом `getSuggestOracle()` класса `SuggestBox` или создается с помощью конструктора класса:

- ◆ `MultiWordSuggestOracle()` — введенная текстовая строка делится на слова, которые комбинируются с разделителем в виде пробела и тем самым образуют набор строк запросов;

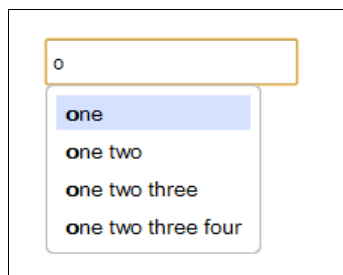
- ◆ `MultiWordSuggestOracle(java.lang.String whitespaceChars)` — указаны разделители для комбинации слов в запросы:

```
MultiWordSuggestOracle oracle = new MultiWordSuggestOracle(".,");
```

В данном случае из строки `"foo.bar"` будут созданы запросы `"foo"`, `"bar"`, `"foo.bar"`, `"foo...bar"`, `"foo, bar"`.

Объект `MultiWordSuggestOracle` заполняется подсказками с помощью методов `add(java.lang.String suggestion)` и `addAll(java.util.Collection<java.lang.String> collection)` класса `MultiWordSuggestOracle` (рис. 15.17):

```
MultiWordSuggestOracle oracle = new MultiWordSuggestOracle();
oracle.add("one");
oracle.add("one two");
oracle.add("one two three");
oracle.add("one two three four");
SuggestBox box = new SuggestBox(oracle);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(box,200,50);
```



**Рис. 15.17.** Поле с подсказками `SuggestBox`



Создание своей реализации абстрактного класса `SuggestOracle` позволяет организовать запросы подсказок к серверной части приложения с помощью определения метода `requestSuggestions(SuggestOracle.Request request, SuggestOracle.Callback callback)`, где объект `SuggestOracle.Request` с помощью метода `getQuery()` возвращает строку запроса, после чего должен быть сформирован объект `SuggestOracle.Response`, содержащий подсказки и создаваемый с помощью конструктора `SuggestOracle.Response(java.util.Collection<? extends SuggestOracle.Suggestion> suggestions)`. После формирования объекта `SuggestOracle.Response` он должен быть передан методу `onSuggestionsReady(SuggestOracle.Request request, SuggestOracle.Response response)` объекта `SuggestOracle.Callback`.

Методы `addKeyDownHandler(KeyDownHandler handler)`, `addKeyPressHandler(KeyPressHandler handler)`, `addKeyUpHandler(KeyUpHandler handler)` класса `SuggestBox` позволяют определить для `SuggestBox`-компонента обработчики событий клавиатуры, метод `addSelectionHandler(SelectionHandler<SuggestOracle.Suggestion> handler)` — обработчик события выбора подсказки:

```
final SuggestBox box = new SuggestBox(oracle);
box.addSelectionHandler(new SelectionHandler<SuggestOracle.Suggestion> ()
{ public void onSelection(SelectionEvent<SuggestOracle.Suggestion> event) {
    Window.alert(box.getValue());
}});
```

Метод `addValueChangeHandler(ValueChangeHandler<java.lang.String> handler)` — обработчик изменения содержимого поля.

Методы `getText()` или `getValue()`, `setText(java.lang.String text)` или `setValue(java.lang.String newValue)` возвращают и определяют строку поля `SuggestBox`, метод `refreshSuggestionList()` обновляет список подсказок, метод `setLimit(int limit)` ограничивает список подсказок.

От класса `UIObject` класс `SuggestBox` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Редактор текста *RichTextArea*

Компонент `RichTextArea` создается с помощью конструктора класса без аргументов.

Класс `RichTextArea` расширяет классы `FocusWidget`, `Widget` и `UIObject`.

По умолчанию `RichTextArea`-компонент имеет CSS-класс `.gwt-RichTextArea`.

Для создания редактора `RichTextArea` с панелью инструментов в пакет `.client` GWT-проекта скопируем файлы `RichTextToolbar.java` и `RichTextToolbar_Strings.properties`, а также файлы изображений, расположенные в папке `samples\Showcase\src\com\google\gwt\sample\showcase\client\content\text` дистрибутива GWT SDK.

Произведем замену пакета класса `RichTextToolbar` на пакет приложения и в главном классе приложения добавим код:

```
public class GWTApplication implements EntryPoint {
public void onModuleLoad() {
```

```

final RichTextArea area = new RichTextArea();
area.ensureDebugId("cwRichText-area");
area.setSize("100%", "14em");
RichTextToolbar toolbar = new RichTextToolbar(area);
toolbar.ensureDebugId("cwRichText-toolbar");
toolbar.setWidth("100%");
area.addKeyPressHandler(new KeyPressHandler() {
public void onKeyPress(KeyPressEvent event) {
    char keyCode = event.getCharCode();
    if (keyCode == KeyCodes.KEY_ENTER) {
        Window.alert(area.getHTML());
    }
}});
Grid grid = new Grid(2, 1);
grid.setStyleName("cw-RichText");
grid.setWidget(0, 0, toolbar);
grid.setWidget(1, 0, area);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add( grid,200,50);
}}

```

В результате страница приложения отобразит редактор текста `RichTextArea` с панелью инструментов (рис. 15.18).

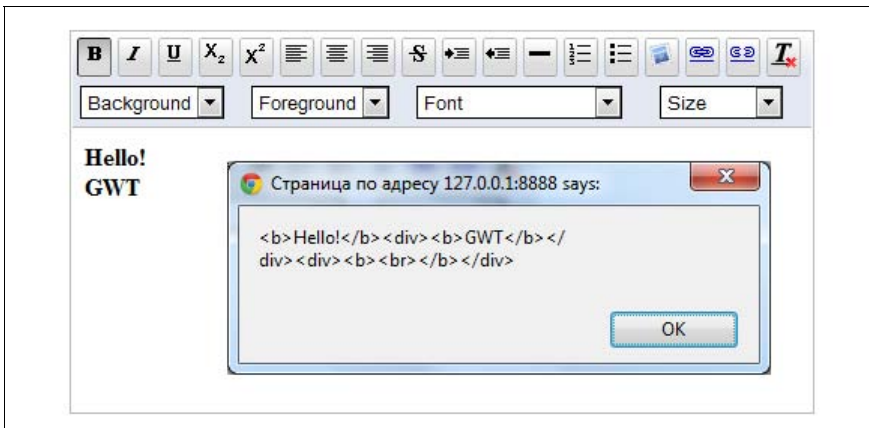


Рис. 15.18. Редактор текста `RichTextArea` с панелью инструментов

Компонент `RichTextArea` форматирует текст с помощью HTML-разметки и методы `setHTML(SafeHtml html)`, `setHTML(java.lang.String html)` и `getHTML()` класса `RichTextArea` устанавливают и возвращают текст `RichTextArea`-компонента с HTML-разметкой. Методы `setText(java.lang.String text)` и `getText()` класса `RichTextArea` устанавливают и возвращают текст `RichTextArea`-компонента без HTML-разметки.

Класс `RichTextArea` наследует от класса `FocusWidget` методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики собы-

тий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки").

## Таблица *FlexTable*

Компонент `FlexTable` позволяет создавать таблицу с динамическим добавлением ячеек, которые могут объединять строки и столбцы (рис. 15.19):

```
Label lb1 = new Label("Label1");
Label lb2 = new Label("Label2");
Label lb3 = new Label("Label3");
Label lb4 = new Label("Label4");
FlexTable t = new FlexTable();
t.setWidget(0, 0, lb1);
t.setWidget(1, 0, lb2);
t.setWidget(2, 0, lb3);
t.setWidget(0, 1, lb4);
t.setText(3, 5, "Text");
FlexTable.FlexCellFormatter cellFormatter = t.getFlexCellFormatter();
cellFormatter.setRowSpan(0, 1, 2);
cellFormatter.setAlignment(1, 1, HasHorizontalAlignment.ALIGN_LEFT,
HasVerticalAlignment.ALIGN_MIDDLE);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(t,200,50);
```

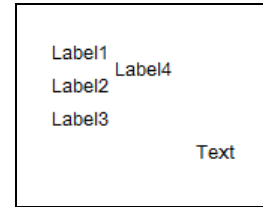


Рис. 15.19. Таблица `FlexTable`

Компонент `FlexTable` создается с помощью конструктора класса без аргументов.

Класс `FlexTable` расширяет классы `HTMLTable`, `Panel`, `Widget` и `UIObject`.

Таблица `FlexTable` заполняется ячейками и их содержимым с помощью следующих методов:

- ◆ `addCell(int row)` — добавляет пустую ячейку в существующий ряд таблицы;
- ◆ `insertCell(int beforeRow, int beforeColumn)` — вставляет пустую ячейку в указанную позицию;
- ◆ `insertRow(int beforeRow)` — вставляет пустой ряд в указанную позицию;
- ◆ `setHTML(int row, int column, SafeHtml html)` и `setHTML(int row, int column, java.lang.String html)` — определяют HTML-содержимое конкретной ячейки с созданием ячейки, если она не существует;
- ◆ `setText(int row, int column, java.lang.String text)` — определяет текст конкретной ячейки с созданием ячейки, если она не существует;
- ◆ `setWidget(int row, int column, Widget widget)` — определяет `Widget`-компонент конкретной ячейки с созданием ячейки, если она не существует;
- ◆ `removeAllRows()` — удаляет все ряды таблицы;

- ◆ `removeCell(int row, int col)` — удаляет ячейку;
- ◆ `removeCells(int row, int column, int num)` — удаляет несколько ячеек;
- ◆ `removeRow(int row)` — удаляет ряд;
- ◆ `clear()` — удаляет все Widget-компоненты таблицы без удаления текста ячеек;
- ◆ `clear(boolean clearInnerHTML)` — удаляет все Widget-компоненты таблицы с удалением текста ячеек;
- ◆ `clearCell(int row, int column)` — очищает ячейку;
- ◆ `remove(Widget widget)` — удаляет Widget-компонент.

Внешний вид `FlexTable`-компонента устанавливается следующими методами:

- ◆ `setBorderWidth(int width)` — устанавливает толщину рамки таблицы и включает отображение рамок ячеек;
- ◆ `setCellPadding(int padding)` и `setCellSpacing(int spacing)` — задают отступ и интервал для всех ячеек таблицы.

Кроме того, формат таблицы `FlexTable` может быть определен путем получения объекта `FlexTable.FlexCellFormatter` посредством метода `getFlexCellFormatter()` класса `FlexTable` с последующим применением методов классов `FlexTable.FlexCellFormatter` и `HTMLTable.CellFormatter`:

- ◆ `setColSpan(int row, int column, int colSpan)` и `setRowSpan(int row, int column, int rowSpan)` — устанавливают объединение ячейкой столбцов и рядов таблицы;
- ◆ `addStyleName(int row, int column, java.lang.String styleName)` — добавляет CSS-стиль для определенной ячейки;
- ◆ `setAlignment(int row, int column, HasHorizontalAlignment.HorizontalAlignmentConstant hAlign, HasVerticalAlignment.VerticalAlignmentConstant vAlign)`, `setHorizontalAlignment(int row, int column, HasHorizontalAlignment.HorizontalAlignmentConstant align)`, `setVerticalAlignment(int row, int column, HasVerticalAlignment.VerticalAlignmentConstant align)` — устанавливают горизонтальное и вертикальное выравнивания ячейки;
- ◆ `setHeight(int row, int column, java.lang.String height)` и `setWidth(int row, int column, java.lang.String width)` — устанавливают размеры ячейки;
- ◆ `setWordWrap(int row, int column, boolean wrap)` — определяет перенос слов текста ячейки.

## Методы

```

addClickHandler(ClickHandler handler)
addDoubleClickHandler(DoubleClickHandler handler)
addDragEndHandler(DragEndHandler handler)
addDragEnterHandler(DragEnterHandler handler)
addDragHandler(DragHandler handler)
addDragLeaveHandler(DragLeaveHandler handler)
addDragOverHandler(DragOverHandler handler)

```

```
addDragStartHandler(DragStartHandler handler)
addDropHandler(DropHandler handler)
```

класса `HTMLTable` позволяют определить для таблицы `FlexTable` обработчики событий.

От класса `UIObject` класс `FlexTable` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Таблица *Grid*

Таблица `Grid` отличается от таблицы `FlexTable` тем, что оперирует не отдельными ячейками, а целиком рядами и колонками таблицы:

```
Label lb1 = new Label("Label1");
Label lb2 = new Label("Label2");
Label lb3 = new Label("Label3");
Label lb4 = new Label("Label4");
Grid g = new Grid(4, 6);
g.setWidget(0, 0, lb1);
g.setWidget(1, 0, lb2);
g.setWidget(2, 0, lb3);
g.setWidget(0, 1, lb4);
g.setText(3, 5, "Text");
g.setBorderWidth(2);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(g,200,50);
```

Компонент `Grid` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргументов количество рядов и колонок таблицы.

Класс `Grid` расширяет классы `HTMLTable`, `Panel`, `Widget` и `UIObject`.

Таблица `Grid` заполняется рядами и колонками, а также их содержимым с помощью следующих методов:

- ◆ `resize(int rows, int columns)`, `resizeColumns(int columns)` и `resizeRows(int rows)` — изменяют количество колонок и рядов таблицы;
- ◆ `insertRow(int beforeRow)` и `removeRow(int row)` — первый метод вставляет, а второй удаляет ряд;
- ◆ `clearCell(int row, int column)` — очищает ячейку;
- ◆ `setHTML(int row, int column, SafeHtml html)` и `setHTML(int row, int column, java.lang.String html)` — определяют HTML-содержимое конкретной ячейки;
- ◆ `setText(int row, int column, java.lang.String text)` — определяет текст конкретной ячейки;
- ◆ `setWidget(int row, int column, Widget widget)` — определяет `Widget`-компонент конкретной ячейки;

- ◆ `clear()` — удаляет все Widget-компоненты таблицы без удаления текста ячеек;
- ◆ `clear(boolean clearInnerHTML)` — удаляет все Widget-компоненты таблицы с удалением текста ячеек;
- ◆ `remove(Widget widget)` — удаляет Widget-компонент.

Внешний вид Grid-компонента устанавливается следующими методами:

- ◆ `setBorderWidth(int width)` — устанавливает толщину рамки таблицы и включает отображение рамок ячеек;
- ◆ `setCellPadding(int padding)` и `setCellSpacing(int spacing)` — задают отступ и интервал для всех ячеек таблицы.

Кроме того, формат таблицы Grid может быть определен с помощью объекта `HTMLTable.CellFormatter`, который может быть получен методом `getCellFormatter()` класса `HTMLTable`, с последующим применением методов класса `HTMLTable.CellFormatter`:

- ◆ `addStyleName(int row, int column, java.lang.String styleName)` — добавляет CSS-стиль для определенной ячейки;
- ◆ `setAlignment(int row, int column, HasHorizontalAlignment.HorizontalAlignmentConstant hAlign, HasVerticalAlignment.VerticalAlignmentConstant vAlign), setHorizontalAlignment(int row, int column, HasHorizontalAlignment.HorizontalAlignmentConstant align), setVerticalAlignment(int row, int column, HasVerticalAlignment.VerticalAlignmentConstant align)` — устанавливают горизонтальное и вертикальное выравнивания ячейки;
- ◆ `setHeight(int row, int column, java.lang.String height)` и `setWidth(int row, int column, java.lang.String width)` — устанавливают размеры ячейки;
- ◆ `setWordWrap(int row, int column, boolean wrap)` — определяет перенос слов текста ячейки.

## Методы

```
addClickHandler(ClickHandler handler)
addDoubleClickHandler(DoubleClickHandler handler)
addDragEndHandler(DragEndHandler handler)
addDragEnterHandler(DragEnterHandler handler)
addDragHandler(DragHandler handler)
addDragLeaveHandler(DragLeaveHandler handler)
addDragOverHandler(DragOverHandler handler)
addDragStartHandler(DragStartHandler handler)
addDropHandler(DropHandler handler)
```

класса `HTMLTable` позволяют определить для таблицы Grid обработчики событий.

От класса `UIObject` класс `Grid` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Всплывающие окна *PopupPanel*, *DecoratedPopupPanel*, *LoggingPopup* и *DialogBox*

Компонент `PopupPanel` создается с помощью конструктора класса, который может быть конструктором без аргументов или принимать в качестве аргумента флаг, определяющий, будет ли всплывающее окно автоматически закрываться при потере фокуса, а также флаг, определяющий, будет ли всплывающее окно модальным, т. е. блокирующим GUI-интерфейс:

```
Button btn = new Button("Popup");
final PopupPanel popup=new PopupPanel(false, true);
popup.setWidget(new Label("Popup"));
popup.setGlassEnabled(true);
btn.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        int x=event.getClientX();
        int y=event.getClientY();
        popup.setPopupPosition(x, y);
        popup.show();
    }
});
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(btn,200,50);
```

Класс `PopupPanel` расширяет классы `SimplePanel`, `Panel`, `Widget` и `UIObject`, и по умолчанию для `PopupPanel`-компонента определены CSS-классы:

- ◆ `.gwt-PopupPanel` — общий стиль компонента;
- ◆ `.gwt-PopupPanel .popupContent` — стиль содержимого компонента;
- ◆ `.gwt-PopupPanelGlass` — стиль фоновой маски компонента.

Содержимое `PopupPanel`-компонента определяется с помощью метода `setWidget(Widget w)` класса `PopupPanel`.

Метод `addAutoHidePartner(Element partner)` класса `PopupPanel` определяет компонент, при щелчке на котором всплывающее окно не закрывается автоматически:

```
Button btn = new Button("Popup");
final PopupPanel popup=new PopupPanel(true);
popup.setWidget(new Label("Popup"));
final PopupPanel popup1=new PopupPanel(true);
popup1.setWidget(new Label("Popup1"));
popup.addAutoHidePartner(popup1.getElement());
popup.setGlassEnabled(true);
btn.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        int x=event.getClientX();
        int y=event.getClientY();
```

```

        popup.setPopupPosition(x, y);
        popup.show();
        popup1.setPopupPosition(x+50, y+50);
        popup1.show();
    });
    final RootPanel panel=RootPanel.get("container");
    panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
    panel.add(btn,200,50);

```

В приведенном коде при щелчке на кнопке `btn` открываются два окна. При щелчке на окне `popup` окно `popup1` закрывается, а при щелчке на окне `popup1` окно `popup` не закрывается.

Метод `addCloseHandler(CloseHandler<PopupPanel> handler)` класса `PopupPanel` определяет для всплывающего окна обработчик события закрытия окна, метод `center()` отображает компонент по центру окна браузера, методы `show()`, `showRelativeTo(UIObject target)` и `hide()` открывают и закрывают окно, метод `setAnimationEnabled(true)` включает плавное появление и закрытие окна, метод `setAutoHideEnabled(boolean autoHide)` включает или отключает автоматическое закрытие окна при потере фокуса, метод `setGlassEnabled(boolean enabled)` включает или отключает появление фоновой маски окна, метод `setGlassStyleName(java.lang.String glassStyleName)` устанавливает CSS-стиль фоновой маски окна, методы `setHeight(java.lang.String height)` и `setWidth(java.lang.String width)` устанавливают размеры содержимого окна, метод `setModal(boolean modal)` включает или отключает модальность окна, метод `setPopupPosition(int left, int top)` определяет расположение окна, метод `setTitle(java.lang.String title)` определяет всплывающую подсказку окна.

От класса `UIObject` класс `PopupPanel` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

Класс `DecoratedPopupPanel` расширяет класс `PopupPanel` и представляет всплывающее окно, которое отличается от `PopupPanel`-компонента тем, что его содержимое компонуется с помощью сетки 3×3, в которой `Widget`-компонент окна расположен в центре. Это дает возможность закруглить углы всплывающего окна.

Кроме того, для компонента `DecoratedPopupPanel` существенно расширен список CSS-классов, определенных по умолчанию. В частности, если устанавливать размеры компонента методами `setHeight()` и `setWidth()`, для корректного отображения окна необходимо определить CSS-стиль:

```

.gwt-DecoratedPopupPanel .popupMiddleCenter {
    height: 100%;
    width: 100%;
}

```

Класс `com.google.gwt.logging.client.LoggingPopup` расширяет класс `PopupPanel` и представляет окно для вывода сообщений журнала приложения.

Хотя класс `LoggingPopup` и является расширением класса `PopupPanel`, окно `LoggingPopup` по сути не является всплывающим, т. к. компонент `LoggingPopup` отображается сразу



после своего создания как дочерний компонент окна корневого `Logger`-объекта (рис. 15.20):

◆ файл `.gwt.xml`:

```
<inherits name="com.google.gwt.logging.Logging"/>
```

◆ Java:

```
final LoggingPopup popup=new LoggingPopup();
popup.setWidget(new Label("Message"));
```

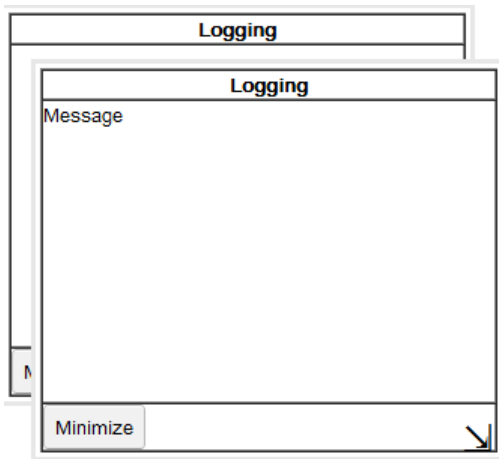


Рис. 15.20. Окно `LoggingPopup`

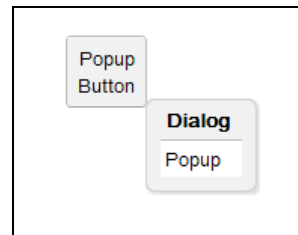


Рис. 15.21. Окно `DialogBox`

При этом содержимое корневого окна `LoggingPopup` определяется с помощью объекта `java.util.logging.Logger`:

```
Logger logger = Logger.getLogger("Logger");
logger.log(Level.INFO, "Message");
```

Класс `DialogBox` расширяет класс `DecoratedPopupPanel` и представляет окно с заголовком, которое может перетаскиваться мышью (рис. 15.21):

```
Button btn = new Button("Popup Button");
final DialogBox dialog=new DialogBox();
dialog.add(new Label("Popup"));
dialog.setAutoHideEnabled(true);
dialog.setText("Dialog");
btn.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        int x=event.getClientX();
        int y=event.getClientY();
        dialog.setPopupPosition(x, y);
        dialog.show();
    }
});
```

```
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(btn,200,50);
```

Для компонента `DialogBox` также существенно расширен список CSS-классов, определенных по умолчанию.

Заголовок компонента `DialogBox` определяется с помощью методов `setHTML(SafeHtml html)`, `setHTML(java.lang.String html)` и `setText(java.lang.String text)` класса `DialogBox`. Методы `setHeight()` и `setWidth()` устанавливают размеры не содержимого компонента `DialogBox`, а размеры самого окна. Переопределение метода `onBrowserEvent(Event event)` обеспечивает обработку событий браузера.

## Уведомление *NotificationMole*

Компонент `NotificationMole` обеспечивает отображение пользователю сообщения с анимацией его появления и задержкой:

```
NotificationMole nm = new NotificationMole();
nm.setMessage("Message");
nm.setAnimationDuration(500);
nm.showDelayed(100);
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(nm,200,50);
```

Класс `NotificationMole` расширяет классы `Composite`, `Widget` и `UIObject`, и его экземпляр создается с помощью конструктора класса без аргументов.

Метод `setMessage(java.lang.String message)` класса `NotificationMole` определяет текст сообщения, метод `showDelayed(int delay)` отображает сообщение после указанной задержки в миллисекундах, а метод `setAnimationDuration(int duration)` устанавливает в миллисекундах анимацию раскрытия сообщения, метод `hide()` закрывает сообщение.

## Панели с закладками *TabPanel* и *TabLayoutPanel*

Компонент `TabPanel` создается с помощью конструктора класса без аргументов:

```
TabPanel tp = new TabPanel();
tp.add(new HTML("<h1>Tab1</h1>"), "Tab1");
tp.add(new HTML("<h1>Tab2</h1>"), "Tab2");
tp.add(new HTML("<h1>Tab3</h1>"), "Tab3");
tp.selectTab(0);
tp.addSelectionHandler(new SelectionHandler<Integer>() {
    public void onSelection(SelectionEvent<Integer> event) {
        Window.alert("Tab: " + event.getSelectedItemId());
    }
});
```

```
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(tp,200,50);
```

Класс `TabPanel` расширяет классы `Composite`, `Widget` и `UIObject`, и по умолчанию для `TabPanel`-компонента определены CSS-классы `.gwt-TabPanel` и `.gwt-TabPanelBottom`.

Панель `TabPanel` наполняется закладками с помощью методов:

- ◆ `add(Widget w)` — закладка с `Widget`-компонентом без надписи;
- ◆ `add(Widget w, java.lang.String tabText)`, `add(Widget w, java.lang.String tabText, boolean asHTML)`, `add(Widget w, Widget tabWidget)` — закладка с `Widget`-компонентом и надписью;
- ◆ `insert(Widget widget, java.lang.String tabText, boolean asHTML, int beforeIndex)`, `insert(Widget widget, java.lang.String tabText, int beforeIndex)`, `insert(Widget widget, Widget tabWidget, int beforeIndex)` — вставляет закладку;
- ◆ `remove(int index)`, `remove(Widget widget)` — удаляет закладку;
- ◆ `clear()` — очищает панель от закладок.

Метод `addBeforeSelectionHandler(BeforeSelectionHandler<java.lang.Integer> handler)` класса `TabPanel` позволяет определить обработчик, вызываемый перед открытием закладки, например, чтобы предотвратить ее выбор:

```
tp.addBeforeSelectionHandler(new BeforeSelectionHandler<Integer>() {
    public void onBeforeSelection(BeforeSelectionEvent<Integer> event) {
        if (event.getItem().intValue() == 1) {
            event.cancel();
        }
    }
});
```

Метод `addSelectionHandler(SelectionHandler<java.lang.Integer> handler)` класса `TabPanel` определяет обработчик события выбора закладки.

Метод `getDeckPanel()` класса `TabPanel` возвращает компонент `DeckPanel`, представляющий нижнюю часть панели, а метод `getTabBar()` — компонент `TabBar`, представляющий верхнюю часть панели.

Метод `selectTab(int index)` класса `TabPanel` позволяет программным способом открыть закладку.

От класса `UIObject` класс `TabPanel` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

Компонент `TabLayoutPanel` создается с помощью конструктора класса, принимающего в качестве аргумента высоту верхней части панели. Кроме того, компонент `TabLayoutPanel` требует добавления в `Layout`-панель:

```
TabLayoutPanel tp = new TabLayoutPanel(1.5, Unit.EM);
tp.add(new HTML("<h1>Tab1</h1>"), "Tab1");
tp.add(new HTML("<h1>Tab2</h1>"), "Tab2");
tp.add(new HTML("<h1>Tab3</h1>"), "Tab3");
tp.selectTab(0);
```

```

tp.addSelectionHandler(new SelectionHandler<Integer>() {
    public void onSelection(SelectionEvent<Integer> event) {
        Window.alert("Tab: " + event.getSelectedItem());
    }
});
tp.addBeforeSelectionHandler(new BeforeSelectionHandler<Integer>() {
    public void onBeforeSelection(BeforeSelectionEvent<Integer> event) {
        if (event.getItem().intValue() == 1) {
            event.cancel();
        }
    }
});
DockLayoutPanel p = new DockLayoutPanel(Unit.PX);
p.add(tp);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(p,200,50);
p.setSize("400px", "150px");

```

Класс `TabLayoutPanel` расширяет классы `ResizeComposite`, `Composite`, `Widget` и `UIObject`, и по умолчанию для `TabPanel`-компонента определены CSS-классы:

- ◆ `.gwt-TabLayoutPanel` — общий стиль панели;
- ◆ `.gwt-TabLayoutPanel .gwt-TabLayoutPanelTabs` — стиль верхней части;
- ◆ `.gwt-TabLayoutPanel .gwt-TabLayoutPanelTab` — стиль отдельной закладки;
- ◆ `.gwt-TabLayoutPanel .gwt-TabLayoutPanelTabInner` — стиль внутренней части закладки;
- ◆ `.gwt-TabLayoutPanel .gwt-TabLayoutPanelContent` — стиль содержимого нижней части закладок.

Панель `TabLayoutPanel` наполняется закладками с помощью методов:

- ◆ `add(Widget w)` — закладка с `Widget`-компонентом без надписи;
- ◆ `add(Widget child, SafeHtml html), add(Widget w, java.lang.String tabText), add(Widget w, java.lang.String tabText, boolean asHTML), add(Widget w, Widget tabWidget)` — закладка с `Widget`-компонентом и надписью;
- ◆ `insert(Widget child, SafeHtml html, int beforeIndex), insert(Widget widget, java.lang.String tabText, boolean asHTML, int beforeIndex), insert(Widget widget, java.lang.String tabText, int beforeIndex), insert(Widget widget, Widget tabWidget, int beforeIndex)` — вставляет закладку;
- ◆ `remove(int index), remove(Widget widget)` — удаляет закладку;
- ◆ `clear()` — очищает панель от закладок.

Метод `addBeforeSelectionHandler(BeforeSelectionHandler<java.lang.Integer> handler)` класса `TabLayoutPanel` позволяет определить обработчик, вызываемый перед открытием закладки, а метод `addSelectionHandler(SelectionHandler<java.lang.Integer> handler)` определяет обработчик события выбора закладки.

Методы `selectTab(int index)` и `selectTab(Widget child)` класса `TabLayoutPanel` позволяют программным способом открыть закладку.

Метод `setAnimationDuration(int duration)` класса `TabLayoutPanel` дает возможность установить продолжительность анимации открытия закладки, а метод `setAnimationVertical(boolean isVertical)` — определить, будет ли это вертикальная или горизонтальная анимация.

Методы `setTabHTML(int index, SafeHtml html)`, `setTabHTML(int index, java.lang.String html)` и `setTabText(int index, java.lang.String text)` класса `TabLayoutPanel` определяют надпись закладки.

От класса `UIObject` класс `TabLayoutPanel` наследует методы определения свойств компонента (см. разд. *"Определение свойств кнопки"* ранее в этой главе).

Основное отличие компонента `TabPanel` от компонента `TabLayoutPanel` заключается в том, что компонент `TabPanel` предназначен для отображения в браузере в режиме Quirks Mode (режим обратной совместимости), а компонент `TabLayoutPanel` — в режиме Standards Mode (режим соответствия стандартам).

## Загрузчик файлов *FileUpload* и форма *FormPanel*

Компонент `FileUpload` представляет собой обертку простого HTML-элемента `<input type="file">` и для отправки выбранного файла на сервер должен использоваться как дочерний компонент компонента `FormPanel`.

Компонент `FormPanel` представляет собой обертку HTML-элемента `<form>` и служит родительским компонентом для `Widget`-компонентов, представляющих текстовые поля, переключатели, флажки, списки выбора, загрузчики файлов и скрытые поля.

Класс `FormPanel` расширяет классы `SimplePanel`, `Panel`, `Widget` и `UIObject`, и его экземпляр создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента имя фрейма `<iframe>`, который должен быть обновлен после завершения отправки формы:

```
final FormPanel form = new FormPanel();
form.setAction("/formAction");
form.setEncoding(FormPanel.ENCODING_MULTIPART);
form.setMethod(FormPanel.METHOD_POST);
VerticalPanel vpanel = new VerticalPanel();
form.setWidget(vpanel);
final FileUpload upload = new FileUpload();
upload.setName("file");
Button submit=new Button("Submit");
vpanel.add(upload);
vpanel.add(submit);
submit.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        Window.alert("Submit go");
        form.submit();
    }
});
```

```

form.addSubmitHandler(new FormPanel.SubmitHandler() {
    public void onSubmit(FormPanel.SubmitEvent event) {
        if (upload.getFilename().equals("")) {
            Window.alert("Select File!!!");
            event.cancel();
        }
    }
});
form.addSubmitCompleteHandler(new FormPanel.SubmitCompleteHandler() {
    public void onSubmitComplete(FormPanel.SubmitCompleteEvent event) {
        Window.alert(event.getResults());
    }
});
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(form,200,50);

```

Метод `addSubmitCompleteHandler(FormPanel.SubmitCompleteHandler handler)` класса `FormPanel` определяет для формы обработчик события завершения отправки формы, который используется для обработки ответа от сервера, метод `addSubmitHandler(FormPanel.SubmitHandler handler)` определяет для формы обработчик события начала отправки формы, который используется для проверки правильности заполнения пользователем элементов формы.

Метод `reset()` класса `FormPanel` очищает все поля формы, методы `setAction(SafeUri url)` и `setAction(java.lang.String url)` определяют адрес отправки формы, метод `setEncoding(java.lang.String encodingType)` указывает значение атрибута `enctype` тега `<form>`, метод `setMethod(java.lang.String method)` указывает HTML-метод GET или POST отправки данных формы, метод `submit()` отправляет данные формы на сервер.

Класс `FileUpload` расширяет классы `Widget` и `UIObject`, и его экземпляр создается с помощью конструктора класса без аргументов.

Метод `getFilename()` класса `FileUpload` возвращает имя файла, выбранного для загрузки на сервер, метод `setName(java.lang.String name)` устанавливает значение атрибута `name` тега `<input type="file">`, переопределение метода `onBrowserEvent(Event event)` в конструкторе класса позволяет обрабатывать события браузера.

## Скрытое поле *Hidden*

Компонент `Hidden` представляет скрытое поле и создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента имя поля и его значение.

Класс `Hidden` расширяет классы `Widget` и `UIObject`.

Метод `setDefaultValue(java.lang.String defaultValue)` класса `Hidden` определяет значение поля по умолчанию, метод `setID(java.lang.String id)` устанавливает HTML-идентификатор поля, метод `setName(java.lang.String name)` устанавливает имя поля, а метод `setValue(java.lang.String value)` — его значение.

## Фрейм *Frame*

Компонент `Frame` является оберткой HTML-элемента `<iframe>` и создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента URL-адрес отображаемого фреймом ресурса:

```
Frame frame = new Frame("/frame.html");
```

По умолчанию для компонента `Frame` определен CSS-класс `.gwt-Frame`.

Класс `Frame` расширяет классы `Widget` и `UIObject`.

Методы `setUrl(SafeUri url)`, `setUrl(java.lang.String url)` класса `Frame` определяют URL-адрес отображаемого фреймом ресурса.

## Изображение *Image*

Компонент `Image` обеспечивает отображение изображения по заданному URL-адресу.

Класс `Image` расширяет классы `Widget` и `UIObject`, и по умолчанию для компонента `Image` определен CSS-класс `.gwt-Image`.

Компонент `Image` создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента URL-адрес изображения, размеры маски, обрезающей изображение, и объект `com.google.gwt.resources.client.ImageResource`.

С помощью объектов `ImageResource` изображения приложения могут быть объединены в один файл, чтобы избежать многочисленных запросов к серверу для загрузки изображений страницы приложения. Для этого в пакете `.client` приложения необходимо создать `ClientBundle`-интерфейс, в котором определены объекты `ImageResource`:

```
import com.google.gwt.resources.client.ClientBundle;
import com.google.gwt.resources.client.ImageResource;
public interface ApplicationResources extends ClientBundle {
    @Source("image1.jpg")
    ImageResource image1();
    @Source("image2.jpg")
    ImageResource image2();
}
```

Объект `ClientBundle` обеспечивает доступ к отдельным изображениям:

```
ApplicationResources resources = GWT.create(ApplicationResources.class);
Image img1 = new Image(resources.image1());
Image img2 = new Image(resources.image2());
VerticalPanel vpanel = new VerticalPanel();
vpanel.add(img1);
vpanel.add(img2);
```

```
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(vpanel,200,50);
```

**Переопределение метода** `onBrowserEvent(Event event)`, а также методы

```
addClickHandler(ClickHandler handler)
addDoubleClickHandler(DoubleClickHandler handler)
addDragEndHandler(DragEndHandler handler)
addDragEnterHandler(DragEnterHandler handler)
addDragHandler(DragHandler handler)
addDragLeaveHandler(DragLeaveHandler handler)
addDragOverHandler(DragOverHandler handler)
addDragStartHandler(DragStartHandler handler)
addDropHandler(DropHandler handler)
addGestureChangeHandler(GestureChangeHandler handler)
addGestureEndHandler(GestureEndHandler handler)
addGestureStartHandler(GestureStartHandler handler)
addMouseDownHandler(MouseDownHandler handler)
addMouseMoveHandler(MouseMoveHandler handler)
addMouseOutHandler(MouseOutHandler handler)
addMouseOverHandler(MouseOverHandler handler)
addMouseUpHandler(MouseUpHandler handler), addMouseWheelHandler(MouseWheelHandler
handler), addTouchCancelHandler(TouchCancelHandler handler),
addTouchEndHandler(TouchEndHandler handler)
addTouchMoveHandler(TouchMoveHandler handler)
addTouchStartHandler(TouchStartHandler handler)
```

класса `Image` обеспечивают обработку событий браузера.

**Методы** `addLoadHandler(LoadHandler handler)` и `addErrorHandler(ErrorHandler handler)` класса `Image` обеспечивают обработку загрузки и ошибки загрузки изображения:

```
img.addLoadHandler(new LoadHandler() {
    public void onLoad(LoadEvent event) {
        Window.alert(""+img.getHeight());
        Window.alert(""+img.getWidth());
    }
});
```

**Метод** `setAltText(java.lang.String altText)` класса `Image` устанавливает альтернативный изображению текст, **метод** `setResource(ImageResource resource)` — объект `ImageResource`, **методы** `setUrl(SafeUri url)` и `setUrl(java.lang.String url)` — адрес изображения, **методы** `setUrlAndVisibleRect(SafeUri url, int left, int top, int width, int height)`, `setUrlAndVisibleRect(java.lang.String url, int left, int top, int width, int height)` и `setVisibleRect(int left, int top, int width, int height)` — маску изображения.

От класса `UIObject` класс `Image` наследует методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).



## Метка *Label*

Компонент `Label` обеспечивает отображение простого текста и создается с помощью конструктора класса, который может быть конструктором без аргументов, или принимать в качестве аргумента текст метки, флаг, определяющий перенос слов, объект `com.google.gwt.i18n.client.HasDirection.Direction.LTR` или `RTL`, указывающий направление текста слева направо или справа налево, объект `com.google.gwt.i18n.shared.DirectionEstimator`, динамически обеспечивающий оценку направления текста.

Для отображения текста компонент `Label` использует тег `<div>` блочной верстки.

Класс `Label` расширяет классы `LabelBase<java.lang.String>`, `Widget` и `UIObject`, и по умолчанию для компонента `Label` определен CSS-класс `.gwt-Label`.

### Методы

```
addClickHandler(ClickHandler handler)
addDoubleClickHandler(DoubleClickHandler handler)
addDragEndHandler(DragEndHandler handler)
addDragEnterHandler(DragEnterHandler handler)
addDragHandler(DragHandler handler)
addDragLeaveHandler(DragLeaveHandler handler)
addDragOverHandler(DragOverHandler handler)
addDragStartHandler(DragStartHandler handler)
addDropHandler(DropHandler handler)
addGestureChangeHandler(GestureChangeHandler handler)
addGestureEndHandler(GestureEndHandler handler)
addGestureStartHandler(GestureStartHandler handler)
addMouseDownHandler(MouseDownHandler handler)
addMouseMoveHandler(MouseMoveHandler handler)
addMouseOutHandler(MouseOutHandler handler)
addMouseOverHandler(MouseOverHandler handler)
addMouseUpHandler(MouseUpHandler handler)
addMouseWheelHandler(MouseWheelHandler handler)
addTouchCancelHandler(TouchCancelHandler handler)
addTouchEndHandler(TouchEndHandler handler)
addTouchMoveHandler(TouchMoveHandler handler)
addTouchStartHandler(TouchStartHandler handler)
```

класса `Label` обеспечивают обработку событий браузера.

Метод `setText(java.lang.String text)` класса `Label` определяет текст метки, метод `setHorizontalAlignment(HasHorizontalAlignment.HorizontalAlignmentConstant align)` класса `LabelBase` — горизонтальное выравнивание текста метки, метод `setWordWrap(boolean wrap)` включает перенос слов текста, метод `setDirectionEstimator(boolean enabled)` — оценку направления текста.

## Метка *HTML*

Класс `HTML` расширяет класс `Label` и обеспечивает отображение текста с HTML-разметкой.

Для компонента `HTML` по умолчанию определен CSS-класс `.gwt-HTML`.

Для отображения текста компонент `HTML` использует тег `<div>` блочной верстки.

Дополнительно к методам класса `Label` класс `HTML` предлагает методы `setHTML(SafeHtml html)` и `setHTML(java.lang.String html)` определения текста с HTML-разметкой для метки.

## Метка *InlineHTML*

Класс `InlineHTML` расширяет класс `HTML` и также обеспечивает отображение текста с HTML-разметкой, однако в отличие от компонента `HTML`, компонент `InlineHTML` для отображения текста использует тег `<span>`, не включающий разметку параграфа.

Для компонента `InlineHTML` по умолчанию определен CSS-класс `.gwt-InlineHTML`.

## Метка *InlineLabel*

Класс `InlineLabel` расширяет класс `Label` и обеспечивает отображение простого текста без разметки, однако в отличие от компонента `Label`, компонент `InlineLabel` для отображения текста использует тег `<span>`, не включающий разметку параграфа.

Для компонента `InlineLabel` по умолчанию определен CSS-класс `.gwt-InlineLabel`.

## Медиакомпоненты *Audio* и *Video*

### Компонент *Video*

Компонент `Video` является оберткой HTML5-элемента `<video>` и создается с помощью статического метода `createIfSupported()` класса `Video`, который возвращает объект `Video`, если браузер поддерживает HTML5-элемент `<video>`:

```
final Video video = Video.createIfSupported();
String url="video.ogv";
video.setSrc(url);
video.setAutoplay(true);
video.setControls(true);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(video,200,50);
```

Класс `com.google.gwt.media.client.Video` расширяет классы `MediaBase`, `FocusWidget`, `Widget` и `UIObject`.

Также как и HTML5-элемент `<video>`, компонент `Video` поддерживает видеоформаты `WebM` и `Theora`.

Методы `getVideoHeight()` и `getVideoWidth()` класса `Video` возвращают размеры видео, метод `setPoster(java.lang.String url)` устанавливает заставку видео.

Методы `addCanPlayThroughHandler(CanPlayThroughHandler handler)`, `addEndedHandler(EndedHandler handler)` и `addProgressHandler(ProgressHandler handler)` класса `MediaBase` определяют для компонента обработчики воспроизведения контента.

Методы `addSource(java.lang.String url)`, `addSource(java.lang.String url, java.lang.String type)` и `setSrc(java.lang.String url)`, `removeSource(SourceElement source)` класса `MediaBase` определяют для компонента контент для воспроизведения.

Метод `setCurrentTime(double time)` класса `MediaBase` устанавливает, а метод `getCurrentTime()` возвращает текущую позицию воспроизведения, методы `setDefaultPlaybackRate(double rate)` и `setPlaybackRate(double rate)`, `getDefaultPlaybackRate()` и `getPlaybackRate()` — скорость воспроизведения, метод `getDuration()` — общую продолжительность воспроизведения, метод `getInitialTime()` — начальную позицию воспроизведения, методы `setVolume(double volume)` и `getVolume()` — громкость воспроизведения, метод `load()` иницирует загрузку контента, метод `pause()` приостанавливает воспроизведение, метод `play()` запускает воспроизведение, метод `setAutoplay(boolean autoplay)` определяет автовоспроизведение контента после его загрузки, метод `setControls(true)` включает отображение элементов контроля плеера, метод `setLoop(true)` включает цикличность воспроизведения, метод `setMuted(boolean muted)` отключает звук.

Класс `Video` наследует от класса `FocusWidget` методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Компонент *Audio*

Компонент `Audio` является оберткой HTML5-элемента `<audio>` и создается с помощью статического метода `createIfSupported()` класса `Audio`, который возвращает объект `Audio`, если браузер поддерживает HTML5-элемент `<audio>`:

```
final Audio audio = Audio.createIfSupported();
String url="muzic.mp3";
audio.setSrc(url);
audio.setAutoplay(true);
audio.setControls(true);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(audio,200,50);
```

Класс `com.google.gwt.media.client.Audio` расширяет классы `MediaBase`, `FocusWidget`, `Widget` и `UIObject`.

Также как и HTML5-элемент `<audio>`, компонент `Audio` поддерживает аудиоформаты MP3, WAV и OGG.

От класса `MediaBase` класс `Audio` наследует методы определения свойств воспроизведения контента (см. компонент `Video`), от класса `FocusWidget` класс `Audio` наследует методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики событий кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Компонент `Canvas`

Компонент `Canvas` является оберткой HTML5-элемента `<canvas>` и создается с помощью статического метода `createIfSupported()` класса `Canvas`, который возвращает объект `Canvas`, если браузер поддерживает HTML5-элемент `<canvas>`:

```
Canvas canvas = Canvas.createIfSupported();
canvas.setCoordinateSpaceHeight(500);
canvas.setCoordinateSpaceWidth(500);
Context2d ctx=canvas.getContext2d();
ctx.setFont("20pt Calibri");
ctx.setFillStyle("red");
ctx.fillText("\Sample String\"",20,20);
ctx.setFillStyle( CssColor.make("blue"));
ctx.strokeText("\Sample String\"",20,20);
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(canvas,200,50);
```

**Класс** `com.google.gwt.canvas.client.Canvas` расширяет классы `FocusWidget`, `Widget` и `UIObject`.

**Метод** `getContext2d()` класса `Canvas` возвращает объект `com.google.gwt.canvas.dom.client.Context2d`, обеспечивающий создание 2D-графики.

**Методы** `setCoordinateSpaceHeight(int height)` и `setCoordinateSpaceWidth(int width)` класса `Canvas` устанавливают размер холста для рисования.

**Методы** `toDataURL()` и `toDataURL(java.lang.String type)` класса `Canvas` возвращают созданную графику в виде закодированной base64 URL-строки (по умолчанию в формате PNG).

**Класс** `Context2d` обеспечивает создание 2D-графики с помощью методов `arc()`, `beginPath()`, `closePath()`, `fill()`, `lineTo()`, `moveTo()`, `stroke()`, `arcTo()`, `bezierCurveTo()`, `createLinearGradient()`, `drawImage()`, `fillRect()`, `fillText()`, `quadraticCurveTo()`, `setFillStyle()`, `setFont()`, `setLineCap()`, `setLineJoin()`, `setLineWidth()`, `setStrokeStyle()`, `setTextAlign()`, `setTextBaseline()`, `strokeRect()`, `strokeText()`, а также различные эффекты с помощью методов `rotate()`, `scale()`, `setGlobalAlpha()`, `setGlobalCompositeOperation()`, `setShadowBlur()`, `setShadowColor()`, `setShadowOffsetX()`, `setShadowOffsetY()`, `setTransform()`, `transform()`, `translate()`.

От класса `FocusWidget` класс `Canvas` наследует методы, позволяющие определить для компонента различные обработчики событий (см. разд. "Обработчики событий

кнопки" ранее в этой главе), а также наследует от классов `FocusWidget` и `UIObject` методы определения свойств компонента (см. разд. "Определение свойств кнопки" ранее в этой главе).

## Панели компоновки

Для компоновки GUI-интерфейса фреймворк GWT предлагает набор панелей, которые в своей основе используют HTML-элементы для расположения GUI-компонентов на странице приложения.

Корневым классом GWT-панелей является класс `Panel`, дающий начало двум ветвям панелей:

- ◆ `AbsolutePanel`, `RootPanel`, `DockPanel`, `HorizontalPanel`, `VerticalPanel`, `DeckPanel`, `FlowPanel`, `HTMLPanel`, `StackPanel`, `DecoratedStackPanel`, `HeaderPanel`, `FlexTable`, `Grid`, `SimplePanel`, `DecoratorPanel`, `FocusPanel`, `FormPanel`, `PopupPanel`, `DecoratedPopupPanel`, `LoggingPopup`, `ScrollPanel`;
- ◆ `DeckLayoutPanel`, `DockLayoutPanel`, `SplitLayoutPanel`, `LayoutPanel`, `RootLayoutPanel`, `ResizeLayoutPanel`, `SimpleLayoutPanel`.

Для компоновки GUI-компонентов существуют также панели `StackLayoutPanel` и `TabLayoutPanel`, не берущие свое начало от класса `Panel`.

Основное отличие `LayoutPanel`-панелей от `Panel`-панелей заключается в том, что `LayoutPanel`-панели имеют более предсказуемое поведение, обеспечивают отслеживание изменения размеров и предназначены для отображения в браузере в режиме `Standards Mode` (режим соответствия стандартам), в то время как `Panel`-панели — в режиме `Quirks Mode` (режим обратной совместимости).

Кроме того, такие панели, как `DeckLayoutPanel`, `DockLayoutPanel`, `LayoutPanel`, `RootLayoutPanel`, `SplitLayoutPanel`, `StackLayoutPanel`, `TabLayoutPanel`, обеспечивают анимацию изменения компоновки.

## Панели `AbsolutePanel`, `RootPanel`, `RootLayoutPanel` и `LayoutPanel`

Панель `AbsolutePanel` обеспечивает абсолютное позиционирование своих дочерних компонентов, при этом необходимо устанавливать точные размеры самой панели:

```
AbsolutePanel panel = new AbsolutePanel();
panel.add(new Label("Item1"), 100, 100);
panel.add(new Label("Item2"), 100, 130);
panel.add(new Label("Item3"), 100, 160);
panel.setSize("200px", "300px");
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,100);
panelR.add(panel,200,50);
```

Панель `AbsolutePanel` создается с помощью конструктора класса без аргументов.

Класс `AbsolutePanel` расширяет классы `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панель `AbsolutePanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget w)` и `add(Widget w, int left, int top)` — добавляют компонент в панель;
- ◆ `insert(Widget w, int beforeIndex)` и `insert(Widget w, int left, int top, int beforeIndex)` — вставляют компонент в определенную позицию;
- ◆ `remove(Widget w)` — удаляет компонент из панели;
- ◆ `setWidgetPosition(Widget w, int left, int top)` — определяет для компонента расположение.

Класс `RootPanel` расширяет класс `AbsolutePanel`, и его экземпляр создается с помощью статических методов класса:

- ◆ `get()` — возвращает панель, являющуюся оберткой HTML-элемента `<body>`.
- ◆ `get(java.lang.String id)` — возвращает панель, являющуюся оберткой HTML-элемента с указанным идентификатором.

В отличие от панели `AbsolutePanel` для панели `RootPanel` не требуется указание ее точных размеров.

Панель `RootLayoutPanel` является оберткой HTML-элемента `<body>`, и ее нельзя ассоциировать с HTML-элементом с указанным идентификатором.

Панель `RootLayoutPanel` можно получить только с помощью статического метода `get()` класса, обеспечивающего получение единственного экземпляра класса.

Класс `RootLayoutPanel` расширяет классы `LayoutPanel`, `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панель `RootLayoutPanel` заполняется компонентами с помощью методов класса `LayoutPanel`:

- ◆ `add(Widget widget)` — добавляет компонент;
- ◆ `insert(Widget widget, int beforeIndex)` — вставляет компонент в определенную позицию;
- ◆ `remove(Widget w)` — удаляет компонент.

Расположение дочерних компонентов панелей `LayoutPanel` и `RootLayoutPanel` определяют следующие методы класса `LayoutPanel`:

- ◆ `setWidgetBottomHeight(Widget child, double bottom, Style.Unit bottomUnit, double height, Style.Unit heightUnit)` — определяет позицию нижней границы компонента на вертикальной оси и высоту компонента;
- ◆ `setWidgetLeftRight(Widget child, double left, Style.Unit leftUnit, double right, Style.Unit rightUnit)` — определяет позиции левой и правой границ компонента на горизонтальной оси;
- ◆ `setWidgetLeftWidth(Widget child, double left, Style.Unit leftUnit, double width, Style.Unit widthUnit)` — определяет позицию левой границы компонента на горизонтальной оси и ширину компонента;

- ◆ `setWidthRightWidth(Widget child, double right, Style.Unit rightUnit, double width, Style.Unit widthUnit)` — определяет позицию правой границы компонента на горизонтальной оси и ширину компонента;
- ◆ `setWidthTopBottom(Widget child, double top, Style.Unit topUnit, double bottom, Style.Unit bottomUnit)` — определяет позиции верхней и нижней границ компонента на вертикальной оси;
- ◆ `setWidthTopHeight(Widget child, double top, Style.Unit topUnit, double height, Style.Unit heightUnit)` — определяет позицию верхней границы компонента на вертикальной оси и высоту компонента.

Метод `animate(int duration)` класса `LayoutPanel` обеспечивает анимацию изменения компоновки:

```
final RootLayoutPanel panel = RootLayoutPanel.get();
final Label l1 = new Label("Item1");
final Label l2 = new Label("Item2");
final Label l3 = new Label("Item3");
panel.add(l1);
panel.add(l2);
panel.add(l3);
panel.setWidthTopHeight(l1, 100, Unit.PX, 20, Unit.PX);
panel.setWidthTopHeight(l2, 130, Unit.PX, 20, Unit.PX);
panel.setWidthTopHeight(l3, 160, Unit.PX, 20, Unit.PX);
Button btn = new Button("Button");
btn.setWidth("50px");
panel.add(btn);
panel.setWidthTopHeight(btn, 200, Unit.PX, 30, Unit.PX);
btn.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        panel.setWidthLeftWidth(l1, 100, Unit.PX, 50, Unit.PX);
        panel.setWidthLeftWidth(l2, 100, Unit.PX, 50, Unit.PX);
        panel.setWidthLeftWidth(l3, 100, Unit.PX, 50, Unit.PX);
        panel.animate(5000);
    }
});
```

Метод `setWidthVisible(Widget child, boolean visible)` класса `LayoutPanel` включает или отключает отображение определенного компонента.

Панель `LayoutPanel` создается с помощью конструктора класса без аргументов, и, как правило, ее использование основано на создании экземпляра класса, заполнении панели дочерними компонентами и добавлении панели в панель `RootLayoutPanel`. Однако панель `LayoutPanel` можно добавить и в панель `RootPanel`, указав ее точные размеры с помощью метода `setSize()`:

```
panel.setSize("200px", "300px");
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

## Панели *StackPanel*, *DecoratedStackPanel* и *StackLayoutPanel*

Панель *StackPanel* компонует дочерние Widget-компоненты в вертикальный стек с заголовками, при щелчке на которых отображается соответствующий Widget-компонент.

Компонент *StackPanel* создается с помощью конструктора класса без аргументов (рис. 15.22):

```
StackPanel panel = new StackPanel();
panel.add(new Label("Item1"), "Item1");
panel.add(new Label("Item2"), "Item2");
panel.add(new Label("Item3"), "Item3");
panel.setWidth("200px");
final RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

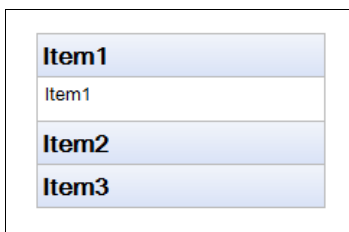


Рис. 15.22. Панель *StackPanel*

Класс *StackPanel* расширяет классы *ComplexPanel*, *Panel*, *Widget* и *UIObject*, и по умолчанию для компонента *StackPanel* определены CSS-классы:

- ◆ *.gwt-StackPanel* — общий стиль панели;
- ◆ *.gwt-StackPanel .gwt-StackPanelItem* — стиль невыбранного компонента;
- ◆ *.gwt-StackPanel .gwt-StackPanelItem-selected* — стиль выбранного компонента;
- ◆ *.gwt-StackPanel .gwt-StackPanelContent* — стиль содержимого компонента.

Панель *StackPanel* заполняется компонентами с помощью следующих методов:

- ◆ *add(Widget w)*, *add(Widget w, SafeHtml stackHtml)*, *add(Widget w, java.lang.String stackText)*, *add(Widget w, java.lang.String stackText, boolean asHTML)* — добавляют Widget-компонент с заголовком или без него;
- ◆ *insert(Widget w, int beforeIndex)* — вставляет Widget-компонент в определенную позицию стека;
- ◆ *remove(int index)* и *remove(Widget child)* — удаляют Widget-компонент из стека.

Методы *setStackText(int index, SafeHtml html)*, *setStackText(int index, java.lang.String text)*, *setStackText(int index, java.lang.String text, boolean asHTML)* класса *StackPanel* определяют заголовок Widget-компонента, а метод *showStack(int index)* отображает определенный Widget-компонент.



Класс `DecoratedStackPanel` расширяет класс `StackPanel` и панель `DecoratedStackPanel` отличается от панели `StackPanel` тем, что обертывает свои компоненты в сетку 2×3 — это дает возможность закруглить верхние углы элементов стека.

Кроме того, для компонента `DecoratedStackPanel` существенно расширен список CSS-классов, определенных по умолчанию.

Панель `StackLayoutPanel` создается с помощью конструктора класса, принимающего в качестве аргумента CSS-единицу измерения высоты заголовка компонента:

```
StackLayoutPanel panel = new StackLayoutPanel(Unit.PX);
panel.add(new Label("Item1"), "Item1", 30);
panel.add(new Label("Item2"), "Item2", 30);
panel.add(new Label("Item3"), "Item3", 30);
// panel.setSize("200px", "300px");
// RootPanel panelR=RootPanel.get("container");
// panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
// panelR.add(panel,200,50);
RootLayoutPanel rp = RootLayoutPanel.get();
rp.add(panel);
// rp.setWidgetLeftRight(panel, 200, Unit.PX, 900, Unit.PX);
// rp.setWidgetTopBottom(panel, 200, Unit.PX, 200, Unit.PX);
```

Класс `StackLayoutPanel` расширяет классы `ResizeComposite`, `Composite`, `Widget` и `UIObject`. Расширение класса `ResizeComposite` обеспечивает отслеживание изменения размеров.

По умолчанию для компонента `StackLayoutPanel` определены CSS-классы:

- ◆ `.gwt-StackLayoutPanel` — общий стиль панели;
- ◆ `.gwt-StackLayoutPanel .gwt-StackLayoutPanelHeader` — стиль заголовка компонента;
- ◆ `.gwt-StackLayoutPanel .gwt-StackLayoutPanelHeader-hovering` — стиль заголовка компонента при наведении курсора мыши;
- ◆ `.gwt-StackLayoutPanel .gwt-StackLayoutPanelContent` — стиль содержимого компонента.

Панель `StackLayoutPanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget w)`, `add(Widget widget, SafeHtml header, double headerSize)`, `add(Widget widget, java.lang.String header, boolean asHtml, double headerSize)`, `add(Widget widget, java.lang.String header, double headerSize)`, `add(Widget widget, Widget header, double headerSize)` — добавляют компонент с заголовком или без него;
- ◆ `clear()` — удаляет все компоненты;
- ◆ `insert(Widget child, SafeHtml html, double headerSize, int beforeIndex)`, `insert(Widget child, java.lang.String text, boolean asHtml, double headerSize, int beforeIndex)`, `insert(Widget child, java.lang.String text, double headerSize, int beforeIndex)`, `insert(Widget child, Widget header, double headerSize, int beforeIndex)` — вставляют компонент в определенную позицию стека;
- ◆ `remove(int index)` и `remove(Widget child)` — удаляют определенный компонент из стека.

**Методы** `addBeforeSelectionHandler(BeforeSelectionHandler<java.lang.Integer> handler)` и `addSelectionHandler(SelectionHandler<java.lang.Integer> handler)` класса `StackLayoutPanel` определяют обработчики выбора компонента панели.

**Метод** `setAnimationDuration(int duration)` класса `StackLayoutPanel` устанавливает продолжительность анимации открытия компонента при щелчке на заголовке.

**Методы** `setHeaderHTML(int index, SafeHtml html)`, `setHeaderHTML(int index, java.lang.String html)`, `setHeaderText(int index, java.lang.String text)` класса `StackLayoutPanel` устанавливают заголовок компонента, а методы `showWidget(int index)` и `showWidget(Widget child)` — отображают компонент.

## Панели *DockPanel*, *DockLayoutPanel* и *SplitLayoutPanel*

Панель `DockPanel` компонует дочерние компоненты в верхней, нижней, левой, правой и центральной областях панели:

```
DockPanel panel = new DockPanel();
panel.add(new Label("Item5"), DockPanel.NORTH);
panel.add(new Label("Item6"), DockPanel.SOUTH);
panel.add(new Label("Item1"), DockPanel.LINE_START);
panel.add(new Label("Item3"), DockPanel.LINE_END);
panel.add(new Label("Item2"), DockPanel.WEST);
panel.add(new Label("Item4"), DockPanel.EAST);
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

Панель `DockPanel` создается с помощью конструктора класса без аргументов, и класс `DockPanel` расширяет классы `CellPanel`, `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панель `DockPanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget widget, DockPanel.DockLayoutConstant direction)` — добавляет компонент в определенную область панели, которая определяется полем класса `DockPanel`:
  - `CENTER` — центральная область;
  - `EAST` — правая область;
  - `LINE_END` — конец ряда компоновки;
  - `LINE_START` — начало ряда компоновки;
  - `NORTH` — верхняя область;
  - `SOUTH` — нижняя область;
  - `WEST` — левая область;
- ◆ `remove(Widget w)` — удаляет компонент из панели.

Методы `setCellHeight(Widget w, java.lang.String height)` и `setCellWidth(Widget w, java.lang.String width)` класса `DockPanel` определяют размеры ячейки, выделенной под компонент.

Методы `setCellHorizontalAlignment(Widget w, HasHorizontalAlignment.HorizontalAlignmentConstant align)` и `setCellVerticalAlignment(Widget w, HasVerticalAlignment.VerticalAlignmentConstant align)` класса `DockPanel` определяют выравнивание компонента внутри ячейки, а методы `setHorizontalAlignment(HasHorizontalAlignment.HorizontalAlignmentConstant align)` и `setVerticalAlignment(HasVerticalAlignment.VerticalAlignmentConstant align)` определяют выравнивание по умолчанию для компонентов панели.

Метод `setBorderWidth(int width)` класса `CellPanel` устанавливает толщину рамки ячейки, а метод `setSpacing(int spacing)` — интервал между ячейками.

Панель `DockLayoutPanel` создается с помощью конструктора класса, принимающего в качестве аргумента единицу измерения размеров дочерних компонентов.

```
DockLayoutPanel panel = new DockLayoutPanel(Unit.PX);
panel.addNorth(new Label("Item1"), 50);
panel.addSouth(new Label("Item3"), 50);
panel.addEast(new Label("Item5"), 50);
panel.addWest(new Label("Item6"), 50);
panel.add(new Label("Item2"));
// panel.setSize("150px", "150px");
// RootPanel panelR=RootPanel.get("container");
// panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
// panelR.add(panel,200,50);
RootLayoutPanel rp = RootLayoutPanel.get();
rp.add(panel);
rp.setWidgetLeftRight(panel, 200, Unit.PX, 1000, Unit.PX);
rp.setWidgetTopBottom(panel, 200, Unit.PX, 280, Unit.PX);
```

Класс `DockLayoutPanel` расширяет классы `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панель `DockLayoutPanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget widget)`, `addEast(Widget widget, double size)`, `addLineEnd(Widget widget, double size)`, `addLineStart(Widget widget, double size)`, `addNorth(Widget widget, double size)`, `addSouth(Widget widget, double size)`, `addWest(Widget widget, double size)` — добавляют компонент в центральную, левую, правую, верхнюю, нижнюю области панели, в начало и конец ряда компоновки;
- ◆ `insertEast(Widget widget, double size, Widget before)`, `insertLineEnd(Widget widget, double size, Widget before)`, `insertLineStart(Widget widget, double size, Widget before)`, `insertNorth(Widget widget, double size, Widget before)`, `insertSouth(Widget widget, double size, Widget before)`, `insertWest(Widget widget, double size, Widget before)` — вставляют компонент в определенную позицию области;
- ◆ `remove(Widget w)` — удаляет компонент из панели.

Метод `animate()` класса `DockLayoutPanel` обеспечивает анимацию изменения компоновки, метод `setWidgetHidden(Widget widget, boolean hidden)` — включает или отключает

чает отображение компонента, метод `setWidgetSize(Widget widget, double size)` переопределяет размер компонента (кроме центрального).

Класс `SplitLayoutPanel` расширяет класс `DockLayoutPanel` и обеспечивает отображение разделителей между своими дочерними компонентами, которые пользователь может перемещать:

```
SplitLayoutPanel panel = new SplitLayoutPanel();
panel.addNorth(new Label("Item1"), 50);
panel.addSouth(new Label("Item3"), 50);
panel.addEast(new Label("Item5"), 50);
panel.addWest(new Label("Item6"), 50);
panel.add(new Label("Item2"));
RootLayoutPanel rp = RootLayoutPanel.get();
rp.add(panel);
rp.setWidgetLeftRight(panel, 200, Unit.PX, 1000, Unit.PX);
rp.setWidgetTopBottom(panel, 200, Unit.PX, 280, Unit.PX);
```

Панель `SplitLayoutPanel` создается с помощью конструктора класса без аргументов, который подразумевает, что размеры дочерних компонентов измеряются в единицах `Style.Unit.PX`, или конструктора, принимающего в качестве аргумента толщину разделителей в пикселах.

По умолчанию для панели `SplitLayoutPanel` определены CSS-классы

```
.gwt-SplitLayoutPanel, .gwt-SplitLayoutPanel .gwt-SplitLayoutPanel-HDragger,
.gwt-SplitLayoutPanel .gwt-SplitLayoutPanel-VDragger.
```

Метод `setWidgetMinSize(Widget child, int minSize)` класса `SplitLayoutPanel` определяет минимальный размер дочернего компонента.

## Панели *HorizontalPanel* и *VerticalPanel*

Панели `HorizontalPanel` и `VerticalPanel` обеспечивают размещение своих дочерних компонентов по горизонтальной и вертикальной осям.

Панели `HorizontalPanel` и `VerticalPanel` создаются с помощью конструктора класса без аргументов, и классы `HorizontalPanel` и `VerticalPanel` расширяют классы `CellPanel`, `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панели `HorizontalPanel` и `VerticalPanel` заполняются компонентами с помощью следующих методов:

- ◆ `add(Widget w)` — добавляет компонент в панель;
- ◆ `insert(Widget w, int beforeIndex)` — вставляет компонент в определенную позицию;
- ◆ `remove(Widget w), remove(int index)` — удаляют компонент из панели.

Методы `setHorizontalAlignment(HasHorizontalAlignment.HorizontalAlignmentConstant align)` и `setVerticalAlignment(HasVerticalAlignment.VerticalAlignmentConstant align)` классов `HorizontalPanel` и `VerticalPanel` определяют выравнивание по умолчанию для компонентов панели.

Метод `setBorderWidth(int width)` класса `CellPanel` устанавливает толщину рамки ячейки компонента, а метод `setSpacing(int spacing)` — интервал между ячейками панели.

Методы `setCellHeight(Widget w, java.lang.String height)` и `setCellWidth(Widget w, java.lang.String width)` класса `CellPanel` определяют размеры ячейки компонента, методы `setCellHorizontalAlignment(Widget w, HasHorizontalAlignment.HorizontalAlignmentConstant align)` и `setCellVerticalAlignment(Widget w, HasVerticalAlignment.VerticalAlignmentConstant align)` определяют выравнивание компонента внутри ячейки

## Панель *FlowPanel*

Панель `FlowPanel` размещает свои дочерние компоненты в горизонтальные ряды и создается с помощью конструктора класса без аргументов:

```
FlowPanel panel = new FlowPanel();
panel.setSize("300px", "300px");
for (int i = 0; i < 30; i++) {
    CheckBox checkbox = new CheckBox(" " + i);
    panel.add(checkbox);
}
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel, 200, 50);
```

Если перед добавлением компонентов в панель не указать размер панели — будет создан один вертикальный столбец с дочерними компонентами.

Класс `FlowPanel` расширяет классы `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панель `FlowPanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget w)` — добавляет компонент в панель;
- ◆ `insert(Widget w, int beforeIndex)` — вставляет компонент в определенную позицию;
- ◆ `remove(Widget w), remove(int index)` — удаляют компонент из панели;
- ◆ `clear()` — очищает панель.

## Панель *HTMLPanel*

Панель `HTMLPanel` обеспечивает обертывание HTML-кода с последующим добавлением дочерних компонентов в HTML-элементы с определенными идентификаторами:

```
String html = "<div id='one'></div><div id='two'></div>";
HTMLPanel panel = new HTMLPanel(html);
panel.setSize("200px", "120px");
panel.add(new Label("Item1"), "one");
panel.add(new Label("Item2"), "two");
```

```
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

Класс `HTMLPanel` расширяет классы `ComplexPanel`, `Panel`, `Widget` и `UIObject`.

Панель `HTMLPanel` создается с помощью конструктора класса, принимающего в качестве аргумента HTML-разметку и корневой тег HTML-разметки.

Панель `HTMLPanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget widget)`, `add(Widget widget, Element elem)`, `add(Widget widget, java.lang.String id)`, `addAndReplaceElement(Widget widget, Element toReplace)`, `addAndReplaceElement(Widget widget, java.lang.String id)` — добавляют компонент;
- ◆ `remove(Widget w)`, `remove(int index)` — удаляют компонент из панели.

## Панель *HeaderPanel*

Панель `HeaderPanel` размещает свои дочерние компоненты в трех областях — заголовке, центральной области и "подвале":

```
HeaderPanel panel = new HeaderPanel();
panel.setSize("200px", "200px");
panel.setHeaderWidget(new Label("Header"));
panel.setContentWidget(new Label("Content"));
panel.setFooterWidget(new Label("Footer"));
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

Класс `HeaderPanel` расширяет классы `Panel`, `Widget` и `UIObject`, и его экземпляр создается с помощью конструктора без аргументов.

Панель `HeaderPanel` заполняется компонентами с помощью следующих методов:

- ◆ `add(Widget w)`, `setContentWidget(Widget w)`, `setFooterWidget(Widget w)`, `setHeaderWidget(Widget w)` — добавляют компоненты и распределяют их по трем областям панели;
- ◆ `remove(Widget w)` — удаляет компонент из панели.

## Панели *SimplePanel*, *DecoratorPanel*, *FocusPanel* и *SimpleLayoutPanel*

Панель `SimplePanel` может содержать только один компонент и создается с помощью конструктора класса без аргументов или конструктора класса, принимающего в качестве аргумента дочерний компонент.

Класс `SimplePanel` расширяет классы `Panel`, `Widget` и `UIObject`.

Методы `add(Widget w)` и `setWidget(Widget w)` класса `SimplePanel` определяют дочерний компонент панели, а метод `remove(Widget w)` удаляет компонент из панели.

Класс `DecoratorPanel` расширяет класс `SimplePanel` и обеспечивает обертывание своего содержимого в сетку 3×3, в которой `Widget`-компонент панели расположен в центре. Это дает возможность закруглить углы панели и стилизовать углы и границы панели.

Кроме того, для панели `DecoratedPanel` существенно расширен список CSS-классов, определенных по умолчанию. В частности, если устанавливать размеры панели методами `setHeight()` и `setWidth()`, для корректного отображения панели необходимо определить CSS-стиль:

```
.gwt-DecoratorPanel .middleCenter {  
    height: 100%;  
    width: 100%;  
}
```

Панель `DecoratedPanel` также может содержать только один компонент и создается с помощью конструктора класса без аргументов или конструктора класса, принимающего в качестве аргумента стили для рядов (`Left`, `Center` и `Right`) сетки панели.

Класс `FocusPanel` расширяет класс `SimplePanel` и обеспечивает обработку событий браузера с помощью методов:

```
addBlurHandler(BlurHandler handler)  
addClickHandler(ClickHandler handler)  
addDoubleClickHandler(DoubleClickHandler handler)  
addDragEndHandler(DragEndHandler handler)  
addDragEnterHandler(DragEnterHandler handler)  
addDragHandler(DragHandler handler)  
addDragLeaveHandler(DragLeaveHandler handler)  
addDragOverHandler(DragOverHandler handler)  
addDragStartHandler(DragStartHandler handler)  
addDropHandler(DropHandler handler)  
addFocusHandler(FocusHandler handler)  
addGestureChangeHandler(GestureChangeHandler handler)  
addGestureEndHandler(GestureEndHandler handler)  
addGestureStartHandler(GestureStartHandler handler)  
addKeyDownHandler(KeyDownHandler handler)  
addKeyPressHandler(KeyPressHandler handler)  
addKeyUpHandler(KeyUpHandler handler)  
addMouseDownHandler(MouseDownHandler handler)  
addMouseMoveHandler(MouseMoveHandler handler)  
addMouseOutHandler(MouseOutHandler handler)  
addMouseOverHandler(MouseOverHandler handler)  
addMouseUpHandler(MouseUpHandler handler)  
addMouseWheelHandler(MouseWheelHandler handler)  
addTouchCancelHandler(TouchCancelHandler handler)  
addTouchEndHandler(TouchEndHandler handler)  
addTouchMoveHandler(TouchMoveHandler handler)  
addTouchStartHandler(TouchStartHandler handler)
```

Класс `SimpleLayoutPanel` расширяет класс `SimplePanel` и обеспечивает вызов метода `RequiresResize.onResize()` своего дочернего компонента при изменении размеров.

Панель `SimpleLayoutPanel` создается с помощью конструктора класса без аргументов:

```
SimpleLayoutPanel panel = new SimpleLayoutPanel();
panel.setSize("200px", "200px");
panel.add(new Label("Content"));
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
//RootLayoutPanel rp = RootLayoutPanel.get();
//rp.add(panel);
//rp.setWidgetLeftRight(panel, 200, Unit.PX, 1000, Unit.PX);
//rp.setWidgetTopBottom(panel, 200, Unit.PX, 280, Unit.PX);
```

## Панель *ScrollPanel*

Панель `ScrollPanel` расширяет класс `SimplePanel` и обеспечивает обертывание своего дочернего компонента в панель с полосами прокруток:

```
ScrollPanel panel = new ScrollPanel();
panel.setSize("100px", "100px");
Label lb=new Label("Content");
lb.setSize("300px", "300px");
panel.add(lb);
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

Панель `ScrollPanel` может содержать только один компонент и создается с помощью конструктора класса без аргументов или конструктора класса, принимающего в качестве аргумента дочерний компонент.

Класс `ScrollPanel` позволяет управлять полосами прокруток с помощью следующих методов:

- ◆ `addScrollHandler(ScrollHandler handler)` — определяет обработчик события прокрутки;
- ◆ `getHorizontalScrollPosition()`, `getMaximumHorizontalScrollPosition()`, `getMaximumVerticalScrollPosition()`, `getMinimumHorizontalScrollPosition()`, `getMinimumVerticalScrollPosition()`, `getVerticalScrollPosition()` — возвращают данные о полосах прокрутки;
- ◆ `scrollToBottom()`, `scrollToLeft()`, `scrollToRight()`, `scrollToTop()`, `setHorizontalScrollPosition(int position)`, `setVerticalScrollPosition(int position)` — устанавливают позицию прокрутки;
- ◆ `setAlwaysShowScrollBars(boolean alwaysShow)` — регулирует отображение полос прокрутки.



## Панель *DisclosurePanel*

Панель `DisclosurePanel` имеет заголовок, при щелчке на котором открывается содержимое панели.

Класс `DisclosurePanel` расширяет классы `Composite`, `Widget` и `UIObject`, и его экземпляр создается с помощью конструктора класса без аргументов или конструктора, принимающего в качестве аргумента строку заголовка панели и изображения для открытия или закрытия содержимого панели:

```
DisclosurePanel panel = new DisclosurePanel("Header");
Label lb=new Label("Content");
panel.add(lb);
panel.setSize("150px", "150px");
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

По умолчанию для панели `DisclosurePanel` определены CSS-классы `.gwt-DisclosurePanel`, `.gwt-DisclosurePanel-open`, `.gwt-DisclosurePanel-closed`.

Содержимое панели `DisclosurePanel` устанавливается с помощью методов `add(Widget w)`, `clear()`, `remove(Widget w)`, `setContent(Widget content)`, а заголовок — методом `setHeader(Widget headerWidget)`.

Методы `addCloseHandler(CloseHandler<DisclosurePanel> handler)` и `addOpenHandler(OpenHandler<DisclosurePanel> handler)` класса `DisclosurePanel` определяют обработчики закрытия и открытия содержимого панели, а метод `setOpen(boolean isOpen)` программным способом открывает или закрывает содержимое панели.

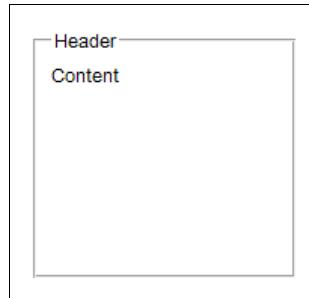
Метод `setAnimationEnabled(boolean enable)` класса `DisclosurePanel` включает или отключает анимацию открытия и закрытия содержимого панели.

## Панель *CaptionPanel*

Панель `CaptionPanel` имеет рамку, в верхней части которой находится заголовок панели (рис. 15.23).

```
CaptionPanel panel = new CaptionPanel("Header");
Label lb=new Label("Content");
panel.add(lb);
panel.setSize("150px", "150px");
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel,200,50);
```

Класс `CaptionPanel` расширяет классы `Composite`, `Widget` и `UIObject`, и его экземпляр создается с помощью конструктора класса без аргументов или конструктора, принимающего в качестве аргумента заголовок панели.



**Рис. 15.23.** Панель  
CaptionPanel

Содержимое панели `CaptionPanel` устанавливается с помощью методов `add(Widget w)`, `clear()`, `remove(Widget w)`, `setContentWidget(Widget w)`, а заголовок — методами `setCaptionHTML(java.lang.String html)`, `setCaptionHTML(SafeHtml html)`, `setCaptionText(java.lang.String text)`.

## Изменение внешнего вида GWT-компонентов

В первую очередь, т. к. класс `Widget` расширяет класс `UIObject`, с помощью методов класса `UIObject` можно изменять размеры GWT-компонентов, включать или выключать их видимость, а также добавлять всплывающую подсказку.

Методы `setHeight(java.lang.String height)`, `setPixelSize(int width, int height)`, `setSize(java.lang.String width, java.lang.String height)` и `setWidth(java.lang.String width)`, унаследованные от класса `UIObject`, устанавливают размеры компонента, а с помощью метода `setVisible(boolean visible)` класса `UIObject` можно сделать компонент невидимым или видимым.

Метод `setTitle(java.lang.String title)` класса `UIObject` позволяет определить всплывающую подсказку для компонента.

Однако основным инструментом изменения внешнего вида GWT-компонентов является применение к ним CSS-стилей.

Для изменения общего вида сразу всех компонентов GWT-фреймворк предлагает набор из четырех тем — `Clean`, `Standard`, `Chrome` и `Dark`, которые подключаются в конфигурационном файле GWT-модуля приложения с помощью тега `<inherits>`:

```
<inherits name='com.google.gwt.user.theme.clean.Clean' />
<!-- <inherits name='com.google.gwt.user.theme.standard.Standard' /> -->
<!-- <inherits name='com.google.gwt.user.theme.chrome.Chrome' /> -->
<!-- <inherits name='com.google.gwt.user.theme.dark.Dark' /> -->
```

Для этого достаточно раскомментировать нужный модуль.

Темы `Clean`, `Standard`, `Chrome` и `Dark` GWT-фреймворка также созданы с использованием CSS-стилей, файлы которых расположены в папках `clean\public\gwt\clean`, `standard\public\gwt\standard`, `chrome\public\gwt\chrome` и `dark\public\gwt\dark` каталога `com\google\gwt\user\theme` библиотеки `gwt-user.jar`.

Например, для кнопки `Button` в каждой теме установлены следующие CSS-стили:

◆ тема **Clean**:

```
.gwt-Button {
    margin: 0;
    padding: 5px 7px;
    text-decoration: none;
    cursor: pointer;
    cursor: hand;
    font-size: small;
    background: url("images/hborder.png") repeat-x 0px -2077px;
    border: 1px solid #bbb;
    border-bottom: 1px solid #a0a0a0;
    border-radius: 3px;
    -moz-border-radius: 3px;
}
.gwt-Button:active {
    border: 1px inset #ccc;
}
.gwt-Button:hover {
    border-color: #939393;
}
.gwt-Button[disabled] {
    cursor: default;
    color: #888;
}
.gwt-Button[disabled]:hover {
    border: 1px outset #ccc;
}
```

◆ тема **Standard**:

```
.gwt-Button {
    margin: 0;
    padding: 3px 5px;
    text-decoration: none;
    font-size: small;
    cursor: pointer;
    cursor: hand;
    background: url("images/hborder.png") repeat-x 0px -27px;
    border: 1px outset #ccc;
}
.gwt-Button:active {
    border: 1px inset #ccc;
}
.gwt-Button:hover {
    border-color: #9cf #69e #69e #7af;
}
```

```
.gwt-Button[disabled] {  
    cursor: default;  
    color: #888;  
}  
.gwt-Button[disabled]:hover {  
    border: 1px outset #ccc;  
}
```

◆ тема Chrome:

```
.gwt-Button {  
    margin: 0;  
    padding: 3px 5px;  
    text-decoration: none;  
    font-size: small;  
    cursor: pointer;  
    cursor: hand;  
    background: url("images/hborder.png") repeat-x 0px -27px;  
    border: 1px outset #ccc;  
}  
.gwt-Button:active {  
    border: 1px inset #ccc;  
}  
.gwt-Button:hover {  
    border-color: #9cf #69e #69e #7af;  
}  
.gwt-Button[disabled] {  
    cursor: default;  
    color: #888;  
}  
.gwt-Button[disabled]:hover {  
    border: 1px outset #ccc;  
}
```

◆ тема Dark:

```
.gwt-Button {  
    margin: 0;  
    padding: 3px 5px;  
    text-decoration: none;  
    font-size: small;  
    cursor: pointer;  
    cursor: hand;  
    background: url("images/hborder.png") repeat-x 0px -27px;  
    border: 1px outset #000;  
}  
.gwt-Button:active {  
    border: 1px inset #000;  
}
```

```
.gwt-Button:hover {
    border: 1px outset #24d3ff;
}
.gwt-Button[disabled] {
    cursor: default;
    color: #777;
}
.gwt-Button[disabled]:hover {
    border: 1px outset #000;
}
```

Используя CSS-классы, установленные для GWT-компонентов по умолчанию, можно переопределять CSS-стили компонентов в CSS-файлах приложения.

Для связывания CSS-файлов с GWT-модулем приложения можно использовать разные способы:

- ◆ с помощью тега `<link>` HTML-страницы приложения:

```
<link rel="stylesheet" href="path/custom.css" type="text/css"/>
```

- ◆ с помощью тега `<stylesheet>` конфигурационного файла `.gwt.xml` GWT-модуля:

```
<stylesheet src="css-url" />
```

где атрибут `src` может содержать абсолютный URL-адрес CSS-файла или его относительный путь в папке `public` пакета приложения файла `.gwt.xml`;

- ◆ с помощью интерфейса `com.google.gwt.resources.client.ClientBundle`:

```
import com.google.gwt.core.client.GWT;
import com.google.gwt.resources.client.*;
public interface ApplicationResources extends ClientBundle {
    public static final ApplicationResources INSTANCE =
        GWT.create(ApplicationResources.class);
    @Source("custom.css")
    public CssResource css();
}
```

Конфигурационный файл `.gwt.xml`:

```
<inherits name="com.google.gwt.resources.Resources" />
```

Метод `onModuleLoad()`:

```
ApplicationResources.INSTANCE.css().ensureInjected();
```

- ◆ с помощью тега `<ui:style>` шаблона `UiBinder`:

```
<ui:style src="custom.css" />
```

В теге `<ui:style>` шаблона `UiBinder` можно и непосредственно определять CSS-стили:

```
<ui:style>
@external .gwt-Button;
```

```
.gwt-Button {
    background: blue;
}
</ui:style>
```

Помимо CSS-классов, установленных по умолчанию, для GWT-компонента также можно определить свой CSS-селектор — ID-селектор или селектор класса.

Для использования ID-селектора GWT-компонент должен иметь свой HTML-идентификатор.

Для того чтобы присвоить идентификатор GWT-компоненту, в HTML-страницу приложения нужно добавить блок `<div id="id-widget"/>` и связать его с GWT-компонентом:

```
com.google.gwt.user.client.DOM.setElementAttribute(new someWidget().getElement(), "id",
" id-widget");
```

Для переопределения CSS-класса по умолчанию GWT-компонента используется метод `setStyleName(java.lang.String style)` суперкласса `com.google.gwt.user.client.ui.UIObject`.

Для добавления CSS-класса применяется метод `addStyleName(java.lang.String style)` суперкласса `UIObject`.

Метод `setAttribute()` класса `com.google.gwt.dom.client.Element` дает возможность определить CSS-стиль GWT-компонента прямо в Java-коде:

```
someWidget.getElement().setAttribute("style", "some-property:value;");
```

Методы класса `com.google.gwt.dom.client.Style` позволяют программным способом определять CSS-свойства GWT-компонента:

```
someWidget.getElement().getStyle().setProperty("some-property", "value");
```

## Фреймворк UiBinder

С помощью фреймворка `UiBinder` GWT можно декларативным способом, используя XML-разметку, компоновать `Widget`-компоненты и HTML-компоненты в один составной `Widget`-компонент.

`UiBinder`-компонент создается в среде Eclipse с помощью команды **New | Other | Google Web Toolkit | UiBinder** контекстного меню окна **Project Explorer** (рис. 15.24).

В результате в пакете `.gwt.client` генерируется XML-файл `.ui.xml` и Java-файл с кодом класса, расширяющего класс `com.google.gwt.user.client.ui.Composite`.

После этого в метод `onModuleLoad()` достаточно добавить код:

```
UiBinderWidget w=new UiBinderWidget();
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(w,100,50);
```

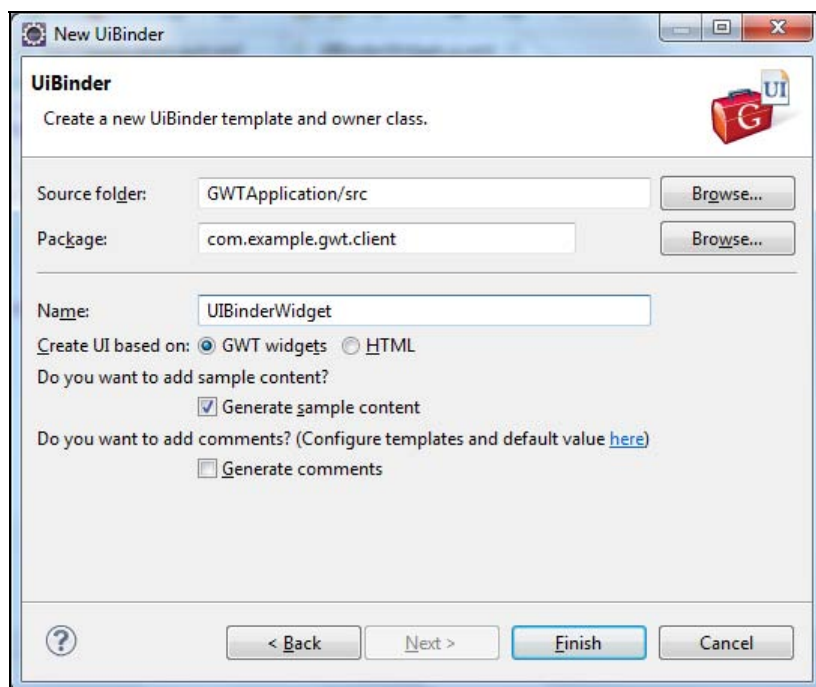


Рис. 15.24. Мастер создания шаблона UiBinder

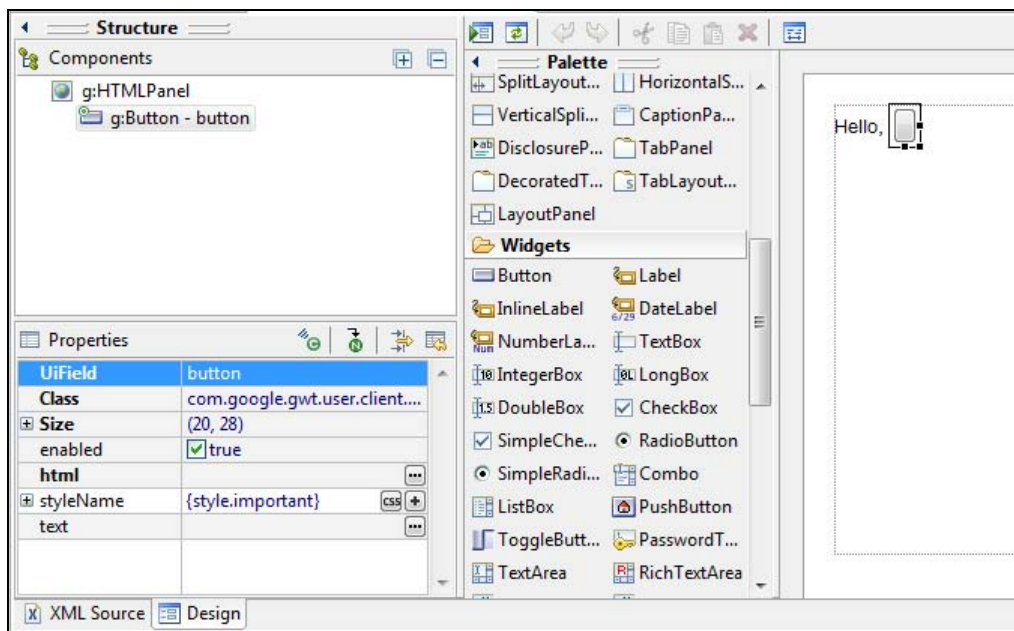


Рис. 15.25. Визуальный редактор шаблона UiBinder

Заполнить шаблон UiBinder компонентами можно с помощью визуального редактора. Для этого XML-файл `.ui.xml` шаблона UiBinder необходимо открыть в редакторе WindowBuilder, щелкнув правой кнопкой мыши в окне **Project Explorer** среды Eclipse и в контекстном меню выбрав команду **Open With | WindowBuilder Editor**. В результате XML-файл `.ui.xml` откроется в окне редактора с двумя вкладками — **XML Source** и **Design**. При выборе вкладки **Design** откроется окно с рабочей областью, палитрой компонентов, окном редактирования свойств компонентов и структуры шаблона (рис. 15.25).

При перетаскивании компонентов из палитры в рабочую область, редактировании их свойств и сохранении изменений автоматически изменяется XML- и Java-код UiBinder-шаблона приложения.



## ГЛАВА 16



# Интернационализация GWT-приложения

Интернационализация (i18n) представляет собой адаптацию приложения к различным языкам, упрощая его использование в различных регионах мира.

При создании текстовых данных приложения лучше всего использовать кодировку UTF-8, обеспечивающую отображение любых символов для любых языков.

Для использования кодировки UTF-8 в GWT-приложении в первую очередь необходимо удостовериться в наличии объявления кодировки UTF-8 на HTML-странице приложения:

```
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
```

Также при разработке приложения в среде Eclipse в свойствах проекта и файлов в разделе **Resources** должна быть указана кодировка UTF-8 (рис. 16.1).

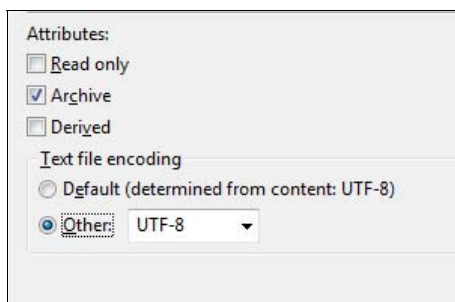


Рис. 16.1. Определение кодировки UTF-8 в свойствах проекта и файлов приложения

При использовании в приложении базы данных, она должна быть настроена на работу с кодировкой UTF-8, и соединение с ней должно осуществляться в кодировке UTF-8.


Интернационализацию GWT-приложения обеспечивает модуль `com.google.gwt.i18n.i18n` GWT-фреймворка, который наследуется модулем `com.google.gwt.user.User`. Так как GWT-модуль приложения, как правило, наследует модуль `User`, объявлять наследование модуля `i18n` не требуется.

Интернационализация GWT-приложения может быть реализована двумя способами — статически или динамически.

## Статическая интернационализация

Визуальный редактор GWT Designer фреймворка GWT упрощает реализацию статической интернационализации GWT-приложения, обеспечивая выделение текстовых строк приложения в отдельные файлы свойств (пар "ключ — значение"), представляющих разные локализации приложения, а также связывание свойств с GWT-компонентами приложения.

Для того чтобы воспользоваться редактором GWT Designer, в окне **Project Explorer** среды Eclipse щелкнем правой кнопкой мыши на узле `EntryPoint`-класса GWT-приложения пакета `.client` и в контекстном меню выберем опцию **Open With | GWT Designer** — в результате `EntryPoint`-класс откроется в графическом визуальном GWT-редакторе, вкладка **Source** которого отображает исходный Java-код класса, а вкладка **Design** позволяет визуальное редактирование GUI-интерфейса GWT-приложения.

В панели инструментов редактора GWT Designer нажмем кнопку  **Externalize Strings** для выделения текстовых строк приложения в отдельные файлы свойств. В результате откроется диалоговое окно мастера, в котором кнопкой **New** создадим файл свойств и интерфейс приложения, расширяющий интерфейс `com.google.gwt.i18n.client.Constants` и обеспечивающий связывание свойств с GWT-компонентами приложения (рис. 16.2).

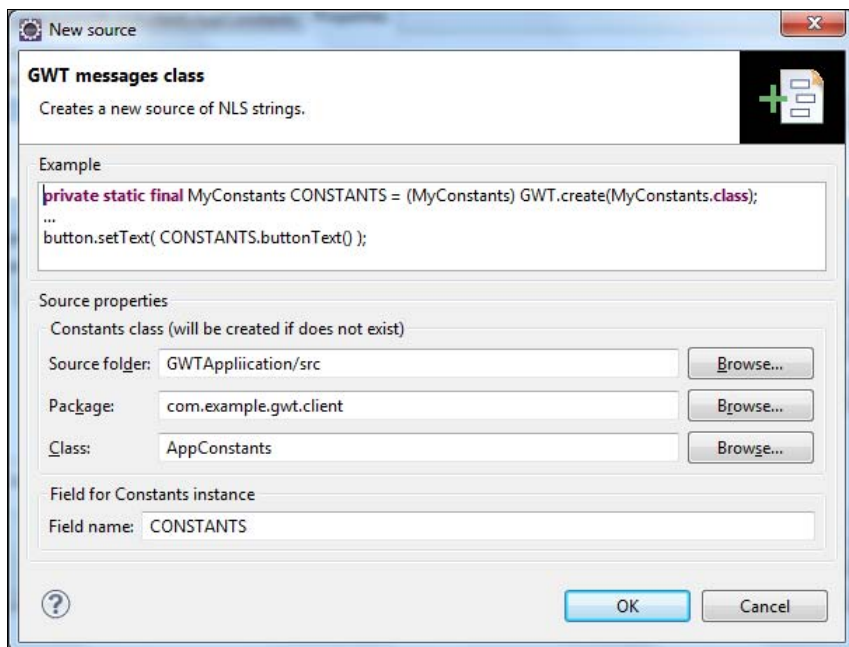


Рис. 16.2. Создание ресурсов интернационализации GWT-приложения

В окне мастера **Externalize strings** отметим флажки текстовых строк приложения, предназначенных для выделения в отдельные файлы свойств, и нажмем кнопки **Externalize** и **OK** (рис. 16.3).

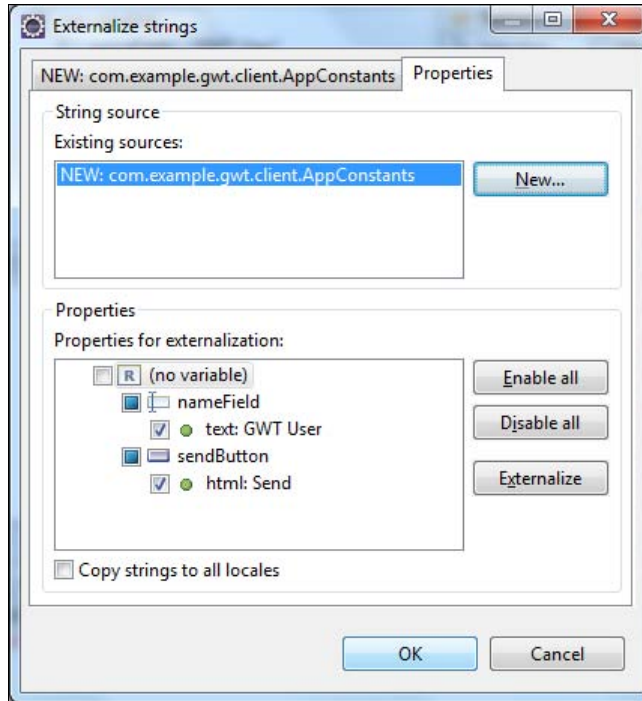


Рис. 16.3. Выделение текстовых строк GWT-приложения в отдельные файлы свойств

В результате в пакете `.client` GWT-приложения будет создан интерфейс `AppConstants`:


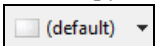
```
import com.google.gwt.i18n.client.Constants;
public interface AppConstants extends Constants {
    String nameField_text();
    String sendButton_html();
}
```

Файл свойств `AppConstants.properties`:

```
#GWT variable: CONSTANTS
nameField_text=GWT User
sendButton_html=Send
```

В `EntryPoint`-класс будет добавлен код:

```
private static final AppConstants CONSTANTS = GWT.create(AppConstants.class);
final Button sendButton = new Button(CONSTANTS.sendButton_html());
final TextBox nameField = new TextBox();
nameField.setText(CONSTANTS.nameField_text());
```

В панели инструментов редактора GWT Designer вместо кнопки  появится кнопка , нажав которую вы увидите окно с таблицей свойств файла `AppConstants.properties` (рис. 16.4).

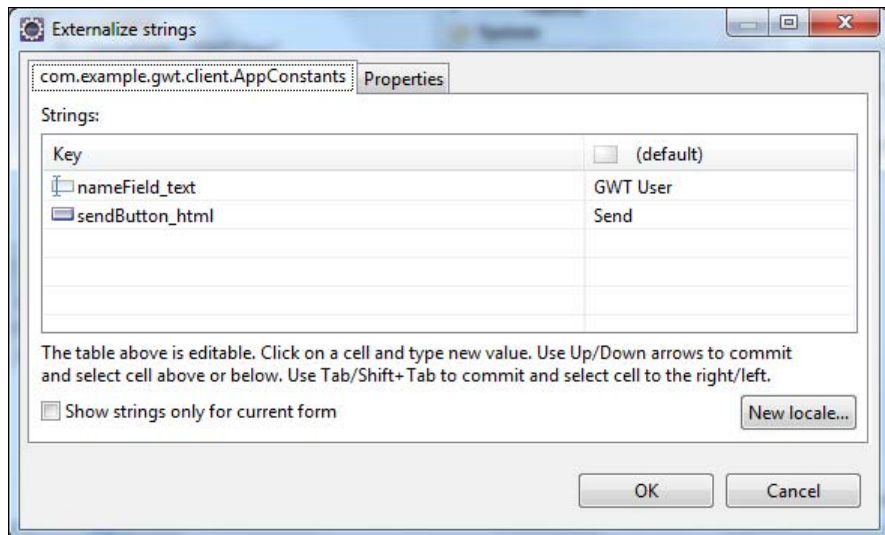


Рис. 16.4. Таблица выделенных текстовых строк GWT-приложения

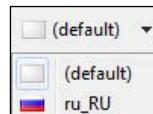
Нажав кнопку **New locale**, создадим новый файл свойств, представляющий русскую локализацию приложения (рис. 16.5).

В окне с таблицей выделенных текстовых строк GWT-приложения отредактируем строки созданной локализации и нажмем кнопку **ОК** (рис. 16.6).

В результате будет создан файл `AppConstants_ru_RU.properties`:

```
#GWT variable: CONSTANTS
nameField_text=GWT Пользователь
sendButton_html=Послать
```

И в панели инструментов редактора GWT Designer появится элемент меню с русской локализацией, при выборе которого в рабочей области редактора GWT Designer отобразится GUI-интерфейс приложения с соответствующей локализацией.



Кроме того, файл `.gwt.xml` GWT-модуля дополнится строками:

```
<inherits name="com.google.gwt.i18n.I18N"/>
<extend-property name="locale" values="ru_RU"/>
```

Теперь при запуске приложения с помощью выбора команды **Run As | Web Application** и добавлении параметра `?locale=ru_RU` в строку URL-адреса GWT-приложения GUI-интерфейс приложения отобразится в русской локализации (рис. 16.7).

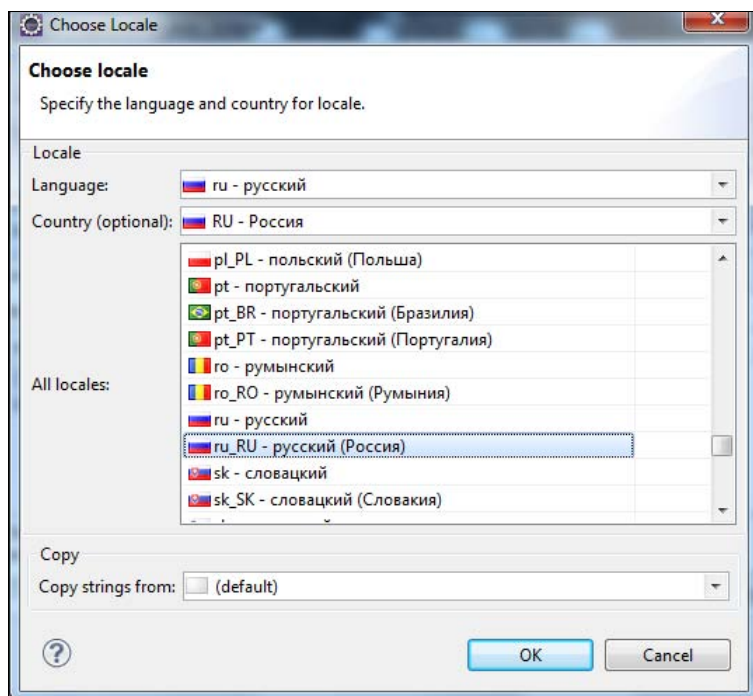


Рис. 16.5. Создание файла свойств GWT-локализации приложения

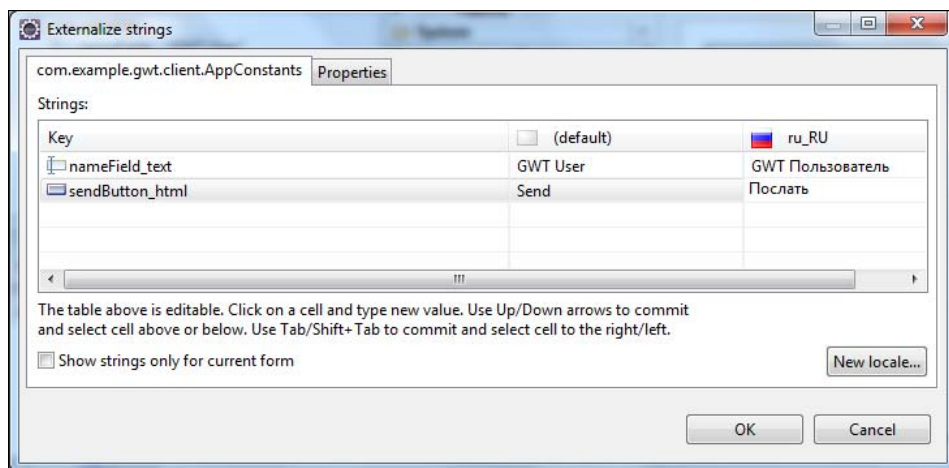


Рис. 16.6. Редактирование строк локализации GWT-приложения

**Please enter your name:**

Рис. 16.7. GUI-интерфейс GWT-приложения в русской локализации

Выбор локализации также можно сделать с помощью тега `<meta>` HTML-страницы приложения:

```
<meta name="gtw:property" content="locale=ru_RU">
```

При компиляции Java-to-JavaScript GWT-приложения с помощью выбора команды **Google | GWT Compile** для GWT-модуля будут сгенерированы разные HTML-файлы для различных локализаций приложения. При запуске GWT-приложения JavaScript-код файла `gtwapplication.nocache.js` определяет значение свойства `locale` и загружает кэшированный HTML-файл с нужной локализацией, таким образом реализуя механизм отложенного связывания `Deferred Binding` фреймворка GWT.

Установить значение свойства `locale`, помимо использования тега `<meta>` и URL-параметра, можно и в файле `.gtw.xml` GWT-модуля:

```
<extend-property name="locale" values="ru_RU"/>
<set-property name="locale" value="ru_RU"/>
```

## Динамическая интернационализация

Класс `com.google.gwt.i18n.client.Dictionary` фреймворка GWT дает возможность программным способом определять текстовые строки GWT-приложения, используя JavaScript-объекты HTML-страницы приложения.

Для применения класса `Dictionary` на HTML-странице приложения определим JavaScript-объект:

```
<script type="text/javascript">
    var Locale = {
        sendButton : "Послать",
        nameField: "GWT Пользователь"
    };
</script>
```

И дополним код `EntryPoint`-класса GWT-модуля:

```
Dictionary locale = Dictionary.getDictionary("Locale");
String sendButtonStr = locale.get("sendButton");
String nameFieldStr = locale.get("nameField");
final Button sendButton = new Button(sendButtonStr);
final TextBox nameField = new TextBox();
nameField.setText(nameFieldStr);
```

В результате свойства JavaScript-объекта HTML-страницы приложения будут связаны с текстовыми строками GWT-компонентов.

## Интернационализация UiBinder

В пакете `.client` GWT-приложения создадим интерфейс `AppConstants`:

```
import com.google.gwt.i18n.client.Messages;
public interface AppConstants extends Messages {
```

```
String nameField_text();
String sendButton_html();
}
```

А также два файла свойств локализаций. Первый — `AppConstants.properties`:

```
#GWT variable: CONSTANTS
nameField_text=GWT User
sendButton_html=Send
```

Второй — `AppConstants_ru_RU.properties`:

```
#GWT variable: CONSTANTS
nameField_text=GWT Пользователь
sendButton_html=Послать
```

В файле `.gwt.xml` GWT-модуля определим локализацию:

```
<inherits name="com.google.gwt.i18n.I18N"/>
<extend-property name="locale" values="ru_RU"/>
<set-property name="locale" value="ru_RU"/>
```

#### ПРИМЕЧАНИЕ

Установить локализацию также можно с помощью тега `<meta>` и URL-параметра.

Создадим в среде Eclipse UiBinder-компонент с помощью команды **New | Other | Google Web Toolkit | UiBinder** контекстного меню окна **Project Explorer**.

Дополним код созданного UiBinder-компонента:

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
  xmlns:g="urn:import:com.google.gwt.user.client.ui">
  <ui:with field='i18n' type='com.example.gwt.client.AppConstants' />
  <ui:style>
    .important {
      font-weight: bold;
    }
  </ui:style>
  <g:HTMLPanel>
    <g:Label text="{i18n.nameField_text}" width="81px"/>
    <g:Button styleName="{style.important}" ui:field="button" text="{i18n.sendButton_html}" />
  </g:HTMLPanel>
</ui:UiBinder>
```

Используем UiBinder-компонент в `EntryPoint`-классе:

```
UiBinderWidget w=new UiBinderWidget();
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(w,100,50);
```

В результате при запуске GWT-приложения GUI-компоненты будут отображаться в указанной локализации.

## ГЛАВА 17



# Программный интерфейс JavaScript Native Interface

Программный интерфейс JavaScript Native Interface (JSNI) позволяет использовать JavaScript-код непосредственно в Java-коде `EntryPoint`-класса GWT-модуля.

Например, чтобы применить JQuery Ajax-запрос к серверу с последующим использованием полученных таким способом данных в Java-коде, нужно включить JQuery-библиотеку в HTML-страницу приложения с помощью тега:

```
<script type="text/javascript" src="jquery.js" ></script>
```

или строки:

```
<script src="jquery.js"/>
```

XML-файла `.gwt.xml` GWT-модуля, правда при этом нужно поместить JQuery-библиотеку в папку `public` пакета XML-файла `.gwt.xml` GWT-модуля и откомпилировать приложение командой **Google | GWT Compile**.

Далее в `EntryPoint`-класс включим код:

```
public class GWTApplication implements EntryPoint { |
    public native String ajax() /*-{
        var data="";
        $wnd.jquery.ajax({ | |
            async:false,
            type: "GET",
            url: "/data",
            dataType: "xml",
            success: (function(xml) {
                $wnd.jquery(xml).find("result").each(function () {
                    data=$wnd.jquery(this).text();
                }); })
        });
    });
    return data;
} */;
public void onModuleLoad() {
    String data=ajax();
    Window.alert(data);
}}
```



Объявление JavaScript-функций в Java-коде с помощью интерфейса JSNI должно содержать ключевое слово `native`, а тело функции должно быть заключено в комментарии `/*- {...}-*/;`.

Вызов таким способом объявленной JavaScript-функции производится аналогично вызову Java-функции.

JavaScript-объекты `window` и `document` обозначаются в JSNI как `$wnd` и `$doc`.

Так как язык JavaScript является нетипизированным, а язык Java — строго типизированным, при передаче данных из JavaScript-кода в Java-код в режиме разработки без компиляции Java-to-JavaScript могут возникать исключения при запуске приложения. Кроме того, существуют ограничения на типы передаваемых данных, в частности нельзя использовать тип данных `long`.

Интерфейс JSNI также позволяет вызывать Java-функции из JavaScript-кода, используя синтаксис:

```
[instance-expr.]@class-name::method-name(param-signature) (arguments)
```

Для доступа к Java-коду GWT-модуля из внешнего JavaScript-кода вызов Java-функции обертывается в функцию `$entry()`, а для доступа к функции используется объект `$wnd`:

```
public static int myFunctionJava(int param) { ... }

public static native void myFunction() /*-{
    $wnd.myFunctionJava = $entry(@mypackage.UtilityClass:: myFunctionJava(I));
}-*/
```

## ГЛАВА 18



# Оптимизация GWT-приложения

## Разделение кода (Code Splitting)

Разработка приложения на основе фреймворка GWT предполагает создание полностью Ajax-приложения с одной HTML-страницей и одним GWT-модулем. Однако GWT-компиляция Java-to-JavaScript большого приложения соответственно генерирует JavaScript-файлы большого размера, что приводит для пользователя к заметной задержке при начальном открытии страницы приложения.

Для решения этой проблемы фреймворк GWT предлагает технологию разделения кода Dead-for-now (DFN) (код не нужный в данный момент), при использовании которой первоначальные JavaScript-файлы приложения при GWT-компиляции разделяются на несколько JavaScript-файлов, загружаемых приложением по требованию.

Чтобы определить код, предназначенный для выделения в отдельный файл с отложенной загрузкой, его обертывают методом `GWT.runAsync()`, создающим точку разделения кода:

```
GWT.runAsync(new RunAsyncCallback() {  
    public void onFailure(Throwable caught) {  
        // Обработка ошибки загрузки кода  
    }  
    public void onSuccess() {  
        // Выделенный код  
    }  
});
```

Как правило, такую конструкцию вставляют в обработчик событий GWT-компонента, иницилируя таким способом загрузку кода по требованию, вызванную действиями пользователя. При этом нужно учитывать, что событие обработчика, иницированное перед загрузкой выделенного кода, не передается в выделенный код и поэтому такое событие необходимо сохранить в отдельной переменной.

Для использования DFN-техники существует шаблон Async Package Pattern, в котором выделенный код организуется в отдельный Java-пакет. Такой Java-пакет в ка-

честве единственной точки входа имеет Java-класс с приватным конструктором и статическим методом, содержащим конструкцию `GWT.runAsync()`, в методе `onSuccess()` которой создается экземпляр Java-класса.

Таким образом, при применении DFN-техники JavaScript-код приложения разделяется на:

- ◆ первоначально загружаемый код (Initial Fragments);
- ◆ выделенный код, на который могут ссылаться несколько методов `runAsync()` (Leftovers Fragments);
- ◆ выделенный код, связанный с определенной точкой разделения (Exclusive Fragments).

После создания точек разделения с помощью конструкций `GWT.runAsync()` необходимо запустить GWT-компиляцию Java-to-JavaScript. В среде Eclipse GWT-компиляция запускается командой **Google | GWT Compile**. При этом для компилятора можно определить опцию `-compileReport`, указывающую создание отчета компиляции (рис. 18.1).

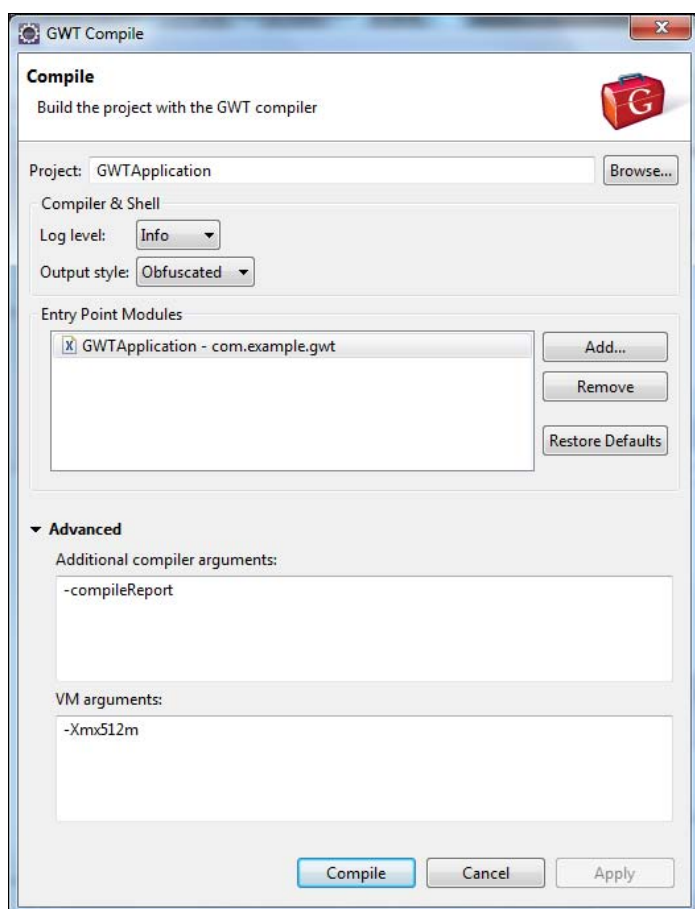


Рис. 18.1. Запуск GWT-компилятора с опцией `-compileReport`

После окончания процесса компиляции в папке GWT-модуля каталога war будет создана папка deferredjs, содержащая папки с разделенным JavaScript-кодом, предназначенным для загрузки в браузеры: gecko1\_8, opera, ie6, ie8, ie9, safari.

Чтобы посмотреть отчет компиляции, содержащий информацию о разделенном коде, нужно открыть в браузере страницу index.html папки extras/modulename/soycReport/compile-report/ каталога проекта.

При наличии нескольких точек разделения в коде приложения для улучшения производительности приложения можно указать точку разделения, которая должна быть загружена приложением первой.

Для этого в метод runAsync() добавляется аргумент в виде имени класса:

```
GWT.runAsync(SomeClass.class, new RunAsyncCallback() {  
    ...  
})
```

А в файл GWT-модуля добавляется свойство:

```
<extend-configuration-property name="compiler.splitpoint.initial.sequence"  
    value="mypackage.SomeClass"/>
```

## Отложенное связывание (Deferred Binding)

Отложенное связывание (Deferred Binding) — это технология фреймворка GWT, при использовании которой в процессе GWT-компиляции Java-to-JavaScript для различных Web-браузеров генерируется своя версия набора кэшированных страниц и каждая такая версия, в свою очередь, может разделяться на несколько версий кода в зависимости от параметров, определенных в файле .gwt.xml GWT-модуля. При этом в процессе запуска приложения на стороне клиента загружается только одна нужная версия кода.

Пример отложенного связывания — это интернационализация GWT-приложения, при которой в файле GWT-модуля указываются локализации приложения, для каждой локализации генерируется своя версия приложения, а в Web-браузер загружается только та версия, которая соответствует значению параметра locale. Генерация в процессе GWT-компиляции нескольких версий приложения, каждая из которых предназначена для своей версии Web-браузера, — это тоже отложенное связывание, при котором в Web-браузер загружается версия приложения в соответствии со значением параметра user.agent.

Воспользоваться техникой отложенного связывания можно двумя способами — с помощью замены типов и используя генерацию кода.

Для применения замены типов в приложении создается интерфейс:

```
public interface MyType {  
    void doSomething();  
}
```

Затем создаются различные реализации интерфейса MyTypeImp1, MyTypeImp2 и т. д.

В методе `onModuleLoad()` `EntryPoint`-класса приложения создается объект `MyType`:

```
MyType myType = (MyType)GWT.create(MyType.class);
myType.doSomething();
```

В конце файла GWT-модуля определяется свойство `mytype` и его использование:

```
<define-property name="mytype" values="type1,type2" />
<property-provider name="mytype">
<![CDATA[
    return __gwt_getMetaProperty("mytype");
]]>
</property-provider>

<replace-with class="com.example.client.MyTypeImpl">
    <when-type-is class="com.example.client.MyType" />
    <when-property-is name="mytype" value="type1" />
</replace-with>

<replace-with class="com.example.client.MyTypeImpl2">
    <when-type-is class="com.example.client.MyType" />
    <when-property-is name="mytype" value="type2" />
</replace-with>
```

В HTML-файле приложения устанавливается значение свойства `mytype`:

```
<meta name="gwt:module" content="js/gwt=com.example.client.MyType"/>
<meta name="gwt:property" content="mytype=type2"/>
```

Таким образом, значение свойства `mytype` определяется динамически из тега `<meta>` и, исходя из значения свойства `mytype`, производится замена типа `MyType` его конкретной реализацией.

В качестве примера рассмотрим создание формы загрузки файлов с возможностью выбора сразу нескольких файлов для загрузки, используя отложенное связывание GWT-компонента `FileUpload` для браузеров с поддержкой атрибута `multiple` тега `<input type="file">`.

В `EntryPoint`-классе приложения создадим форму загрузки файлов:

```
final FormPanel form = new FormPanel();
form.setAction("/formAction");
form.setEncoding(FormPanel.ENCODING_MULTIPART);
form.setMethod(FormPanel.METHOD_POST);
VerticalPanel vpanel = new VerticalPanel();
form.setWidget(vpanel);
final FileUpload upload = (FileUpload)GWT.create(FileUpload.class);
upload.setName("file");
Button submit=new Button("Submit");
vpanel.add(upload);
vpanel.add(submit);
```

```

submit.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        Window.alert("Submit go");
        form.submit();
    }
});
form.addSubmitHandler(new FormPanel.SubmitHandler() {
    public void onSubmit(FormPanel.SubmitEvent event) {
        if (upload.getFilename().equals("")) {
            Window.alert("Select File!!!");
            event.cancel();
        }
    }
});
form.addSubmitCompleteHandler(new FormPanel.SubmitCompleteHandler() {
    public void onSubmitComplete(FormPanel.SubmitCompleteEvent event) {
        Window.alert(event.getResults());
    }
});
final RootPanel panel=RootPanel.get("container");
panel.getElement().getStyle().setPosition(Position.ABSOLUTE);
panel.add(form,200,50);

```

**Создадим класс, расширяющий класс FileUpload:**

```

package com.example.gwt.client;
public class FileUploadMulti extends com.google.gwt.user.client.ui.FileUpload{
    FileUploadMulti(){
        super();
        this.getElement().setAttribute("multiple", "true");
    }
}

```

**В файле GWT-модуля определим условие замены:**

```

<replace-with class="com.example.gwt.client.FileUploadMulti">
  <when-type-is class="com.google.gwt.user.client.ui.FileUpload" />
  <any>
    <when-property-is name="user.agent" value="safari"/>
    <when-property-is name="user.agent" value="gecko1_8" />
  </any>
</replace-with>

```

**Для применения отложенного связывания с генерацией кода создается генератор кода — класс, расширяющий класс `com.google.gwt.core.ext.Generator`, а в файле GWT-модуля определяется условие использования генератора:**

```

<generate-with class="com.example.client.MyGenerator">
  <when-type-assignable class="com.example.client.MyType" />
</generate-with>

```

## ГЛАВА 19



# Поддержка истории Web-браузера

Для Ajax-приложений с одной HTML-страницей, содержимое которой изменяется в ответ на действия пользователя и другие события, существует вопрос использования кнопок Web-браузера **Назад** и **Вперед**.

Фреймворк GWT предлагает решение этой проблемы с помощью класса `com.google.gwt.user.client.History`.

Класс `History` имеет статический метод `newItem(java.lang.String historyToken)`, при использовании которого в конец URL-адреса приложения добавляется фрагмент `#historyToken`. Причем при каждом добавлении нового URL-фрагмента он записывается в историю Web-браузера, так что становится возможна навигация между URL-адресами с фрагментами с помощью кнопок Web-браузера.

Для обработки события изменения URL-фрагмента существует статический метод `addValueChangeHandler(ValueChangeHandler<java.lang.String> handler)` класса `History`, в котором можно связывать URL-фрагменты с определенными состояниями GUI-интерфейса приложения.

В качестве простого примера рассмотрим GWT-приложение, GUI-интерфейс которого содержит меню перехода на различные страницы приложения. При выборе элемента меню в конец URL-адреса приложения добавляется новый фрагмент, а в обработчике события изменения URL-фрагмента изменяется содержимое HTML-страницы приложения, как бы представляющее новую страницу. При этом кнопки Web-браузера позволяют перемещаться между уже открытыми страницами приложения.

HTML-страница приложения имеет два блока: один — для меню, а другой — для содержимого страницы:

```
<div id="menu"></div>
<div id="content"></div>
```

Кроме того, HTML-страница приложения должна содержать код поддержки истории Web-браузера:

```
<iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
style="position:absolute;width:0;height:0;border:0"></iframe>
```

В методе `onModuleLoad()` `EntryPoint`-класса приложения создается меню и определяется обработка событий изменения URL-фрагмента адреса приложения:

```
// Содержимое первой страницы
final HTML content=new HTML("Page1");
RootPanel.get("content").add(content);
// Меню приложения
final MenuBar pages = new MenuBar(true);
// Команда выбора первого элемента меню и добавление первого элемента меню
Command cmd1 = new Command() {
    public void execute() {
        History.newItem("page1");
    }
};
pages.addItem("Page1", cmd1);
// Команда выбора второго элемента меню и добавление второго элемента меню
Command cmd2 = new Command() {
    public void execute() {
        History.newItem("page2");
    }
};
pages.addItem("Page2", cmd2);
// Команда выбора третьего элемента меню и добавление
// третьего элемента меню
Command cmd3 = new Command() {
    public void execute() {
        History.newItem("page3");
    }
};
pages.addItem("Page3", cmd3);
// Добавление меню в панель меню
MenuBar menu = new MenuBar();
menu.addItem("Pages", pages);
// Добавление панели меню в HTML-блок меню
RootPanel.get("menu").add(menu);
// Добавление первоначального URL-фрагмента
History.newItem("page1");
// Обработка событий изменения URL-фрагмента
History.addValueChangeListener(new ValueChangeListener<java.lang.String>() {
    public void onValueChange(ValueChangeEvent<String> event) {
        if(event.getValue().equals("page1")){
            content.setHTML("Page1");
        }
        if(event.getValue().equals("page2")){
            content.setHTML("Page2");
        }
        if(event.getValue().equals("page3")){
            content.setHTML("Page3");
        }
    }
});
```



Класс `History` также имеет статические методы:

- ◆ `back()` и `forward()` — программным способом осуществляют навигацию по истории Web-браузера;
- ◆ `encodeHistoryToken(java.lang.String historyToken)` — кодирует строку для приемлемого использования в качестве URL-фрагмента;
- ◆ `fireCurrentHistoryState()` — вызывает обработчик `ValueChangeListener`;
- ◆ `getToken()` — возвращает текущий URL-фрагмент.

## ГЛАВА 20



# Фреймворк Activities and Places

Фреймворк Activities and Places создан на основе механизма History фреймворка GWT и обеспечивает добавление URL-фрагментов к адресной строке для страниц GWT-приложения, связывая их с историей Web-браузера.

Фреймворк Activities and Places оперирует объектами `Place` и `Activity`.

Объект `Place` представляет определенное состояние GUI-интерфейса — страницу GWT-приложения и может быть конвертирован в URL-фрагмент адреса приложения. Объект `Place` создается с помощью класса, расширяющего класс `com.google.gwt.place.shared.Place`:

```
class OnePlace extends Place {
    private String page;
    public OnePlace(String token) {
        this.page = token;
    }
    public String getContent() {
        return page;
    }
    public static class Tokenizer implements PlaceTokenizer<OnePlace> {
        @Override
        public String getToken(OnePlace place) {
            return place.getContent();
        }
    }
    @Override
    public OnePlace getPlace(String token) {
        return new OnePlace(token);
    }
}

class TwoPlace extends Place {
    private String page;
    public TwoPlace(String token) {
        this.page = token;
    }
}
```

```

public String getContent() {
    return page;
}
public static class Tokenizer implements PlaceTokenizer<TwoPlace> {
    @Override
    public String getToken(TwoPlace place) {
        return place.getContent();
    }
    @Override
    public TwoPlace getPlace(String token) {
        return new TwoPlace(token);
    }
}
}
}

```

Здесь две страницы приложения `OnePlace` и `TwoPlace` связываются с URL-фрагментами с помощью встроенных классов `Tokenizer`, реализующих интерфейс `com.google.gwt.place.shared.PlaceTokenizer<P extends Place>`. Кроме того, для синхронизации URL-фрагментов с объектами `OnePlace` и `TwoPlace` необходимо создать интерфейс приложения, расширяющий интерфейс `com.google.gwt.place.shared.PlaceHistoryMapper`. Данный интерфейс приложения маркируется аннотацией `@WithTokenizers` для автоматической генерации реализации интерфейса при вызове метода `GWT.create(Class)`:

```

@WithTokenizers({OnePlace.Tokenizer.class, TwoPlace.Tokenizer.class})
interface AppPlaceHistoryMapper extends PlaceHistoryMapper
{
}

```

Для представления GUI-интерфейса каждой страницы приложения создается составной Widget-компонент:

```

class ViewOne extends Composite {
    SimplePanel viewPanel = new SimplePanel();
    HTML content=new HTML();
    public ViewOne() {
        viewPanel.add(content);
        initWidget(viewPanel);
    }
    public void setContent(String str) {
        content.setHTML(str);
    }
}
class ViewTwo extends Composite {
    SimplePanel viewPanel = new SimplePanel();
    HTML content=new HTML();
    public ViewTwo() {
        viewPanel.add(content);
        initWidget(viewPanel);
    }
    public void setContent(String str) {
        content.setHTML(str);
    }
}

```

Объект `Activity` отвечает за загрузку GUI-интерфейса страницы приложения, а также за отображение сообщений пользователю при переходе на другую страницу приложения и при закрытии окна Web-браузера. Объект `Activity` создается с помощью класса, расширяющего класс `com.google.gwt.activity.shared.AbstractActivity`:

```
class OneActivity extends AbstractActivity {
    private String page;
    public OneActivity(OnePlace place) {
        this.page = place.getContent();
    }
    @Override
    public void start(AcceptsOneWidget containerWidget, com.google.gwt.event.shared.EventBus
    eventBus) {
        ViewOne view = new ViewOne();
        view.setContent(page);
        containerWidget.setWidget(view);
    }
    @Override
    public String mayStop() {
        return "Stop this activity?";
    }
}

class TwoActivity extends AbstractActivity {
    private String page;
    public TwoActivity(TwoPlace place) {
        this.page = place.getContent();
    }
    @Override
    public void start(AcceptsOneWidget containerWidget,
    com.google.gwt.event.shared.EventBus eventBus) {
        ViewTwo view = new ViewTwo();
        view.setContent(page);
        containerWidget.setWidget(view);
    }
    @Override
    public String mayStop() {
        return "Stop this activity?";
    }
}
```

Для того чтобы связать страницы приложения `OnePlace` и `TwoPlace` с объектами `OneActivity` и `TwoActivity`, необходимо создать класс приложения, реализующий интерфейс `com.google.gwt.activity.shared.ActivityMapper` с определением его метода `Activity getActivity(Place place)`:

```
class AppActivityMapper implements ActivityMapper {
    public AppActivityMapper() {
        super();
    }
}
```

```

@Override
public Activity getActivity(Place place) {
    if (place instanceof OnePlace)
        return new OneActivity((OnePlace) place);
    else if (place instanceof TwoPlace)
        return new TwoActivity((TwoPlace) place);
    return null;
}
}

```

После создания Place-компонентов и Activity-компонентов в EntryPoint-классе приложения создаются объекты:

- ◆ `com.google.gwt.activity.shared.ActivityManager` — отвечает за запуск методов жизненного цикла объектов Activity в ответ на получение событий изменения страницы приложения;
- ◆ `com.google.gwt.place.shared.PlaceHistoryHandler` — отслеживает события изменения страницы приложения и истории Web-браузера и синхронизирует их;
- ◆ `com.google.gwt.place.shared.PlaceController` — с помощью метода `goTo(Place newPlace)` обеспечивает переход на другую страницу приложения.

```

public class GWTApplication implements EntryPoint {
    private Place defaultPlace = new OnePlace("Page1");
    private SimplePanel appWidget = new SimplePanel();

    public void onModuleLoad() {
        com.google.web.bindery.event.shared.EventBus eventBus = new SimpleEventBus();
        final PlaceController placeController = new PlaceController(eventBus);
        ActivityMapper activityMapper = new AppActivityMapper();
        ActivityManager activityManager = new ActivityManager(activityMapper, eventBus);
        activityManager.setDisplay(appWidget);
        AppPlaceHistoryMapper historyMapper= GWT.create(AppPlaceHistoryMapper.class);
        PlaceHistoryHandler historyHandler = new PlaceHistoryHandler(historyMapper);
        historyHandler.register(placeController, eventBus, defaultPlace);
        historyHandler.handleCurrentHistory();
        final MenuBar pages = new MenuBar(true);
        Command cmd1 = new Command() {
            public void execute() {
                placeController.goTo(new OnePlace("Page1"));
            }
        };
        pages.addItem("Page1", cmd1);
        Command cmd2 = new Command() {
            public void execute() {
                placeController.goTo(new TwoPlace("Page2"));
            }
        };
        pages.addItem("Page2", cmd2);
        MenuBar menu = new MenuBar();
        menu.addItem("Pages", pages);
    }
}

```

```
DockPanel panel = new DockPanel();
panel.add(menu, DockPanel.NORTH);
panel.add(appWidget, DockPanel.CENTER);
RootPanel panelR=RootPanel.get("container");
panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
panelR.add(panel, 200, 50);
}}
```

### HTML-страница приложения:

```
<!doctype html>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
<link type="text/css" rel="stylesheet" href="GWTAP.css">
<title>Web Application Starter Project</title>
<script type="text/javascript" language="javascript"
src="gwtap/gwtap.nocache.js"></script>
</head>
<body>
<iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
style="position:absolute;width:0;height:0;border:0"></iframe>
<noscript>
<div style="width: 22em; position: absolute; left: 50%; margin-left: -11em; color: red;
background-color: white; border: 1px solid red; padding: 4px; font-family: sans-serif">
Your web browser must have JavaScript enabled
in order for this application to display correctly.
</div>
</noscript>
<h1>Web Application Starter Project</h1>
<div id="container"></div>
</body>
</html>
```

### XML-файл GWT-модуля:

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='gwtap'>
<inherits name='com.google.gwt.user.User' />
<inherits name='com.google.gwt.user.theme.clean.Clean' />
<inherits name="com.google.gwt.activity.Activity" />
<inherits name="com.google.gwt.place.Place" />
<entry-point class='com.example.gwt.client.GWTAP' />
<source path='client' />
<source path='shared' />
</module>
```

В данном примере навигацию между страницами приложения обеспечивает меню **Pages** с двумя элементами — **Page1** и **Page2**, а также кнопки Web-браузера. При

щелчке элемента меню или кнопки Web-браузера изменяется URL-фрагмент адреса приложения — **#OnePlace:Page1** или **#TwoPlace:Page2**, а также появляется сообщение пользователю (рис. 20.1).

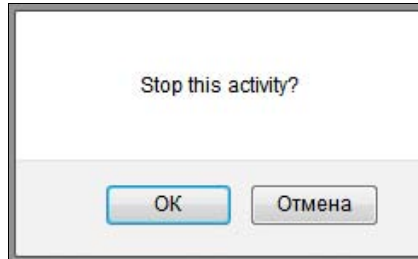


Рис. 20.1. Сообщение пользователю

Навигацию между страницами приложения также можно организовать с помощью гиперссылок `Hyperlink`, при этом не нужно будет применять метод `goTo()` класса `PlaceController`:

```
public class GWTApplication implements EntryPoint {
    private Place defaultPlace = new OnePlace("Page1");
    private SimplePanel appWidget = new SimplePanel();

    public void onModuleLoad() {
        com.google.web.bindery.event.shared.EventBus eventBus = new SimpleEventBus();
        final PlaceController placeController = new PlaceController(eventBus);
        ActivityMapper activityMapper = new AppActivityMapper();
        ActivityManager activityManager = new ActivityManager(activityMapper, eventBus);
        activityManager.setDisplay(appWidget);
        AppPlaceHistoryMapper historyMapper= GWT.create(AppPlaceHistoryMapper.class);
        PlaceHistoryHandler historyHandler = new PlaceHistoryHandler(historyMapper);
        historyHandler.register(placeController, eventBus, defaultPlace);
        historyHandler.handleCurrentHistory();
        Hyperlink link1 = new Hyperlink("Page1", historyMapper.getToken(new OnePlace("Page1")));
        Hyperlink link2 = new Hyperlink("Page2", historyMapper.getToken(new TwoPlace("Page2")));
        DockPanel panel = new DockPanel();
        HorizontalPanel hp=new HorizontalPanel();
        hp.add(link1);
        hp.add(link2);
        panel.add(hp, DockPanel.NORTH);
        panel.add(appWidget,DockPanel.SOUTH);
        RootPanel panelR=RootPanel.get("container");
        panelR.getElement().getStyle().setPosition(Position.ABSOLUTE);
        panelR.add(panel,200,50);
    }
}
```

## ГЛАВА 21



# Взаимодействие GWT-приложения с сервером

## Фреймворк GWT RPC

Фреймворк GWT RPC обеспечивает взаимодействие клиентской части GWT-приложения с его серверной частью.

Фреймворк GWT RPC основывается на Java-сервлетах, обеспечивая сериализацию передаваемых объектов с помощью отложенного связывания Deferred Binding.

При использовании фреймворка GWT RPC реализуется общая концепция Web-сервисов, подразумевающая создание интерфейса сервиса, класса реализации интерфейса сервиса и клиентского класса-заглушки вызова сервиса. При этом отличие фреймворка GWT RPC от технологии Web-сервисов заключается в том, что клиентская часть сервиса GWT RPC в итоге компилируется в JavaScript-код и в процессе передачи данных не используется SOAP-протокол.

Для создания сервиса GWT RPC на стороне клиента создается интерфейс сервиса с объявлением вызываемых методов, а также асинхронный интерфейс, основанный на интерфейсе сервиса и используемый для вызова сервиса. Явным образом реализовывать клиентскую заглушку при этом не требуется, т. к. код заглушки автоматически генерируется GWT-компилятором. На стороне сервера создается класс реализации интерфейса сервиса.

Интерфейс сервиса должен расширять интерфейс `com.google.gwt.user.client.rpc.RemoteService`, который не имеет никаких методов и служит лишь маркером:

```
import com.google.gwt.user.client.rpc.RemoteService;
public interface MyService extends RemoteService {
    String methodServer(String arg) throws IllegalArgumentException;
}
```

Асинхронный интерфейс переопределяет вызываемый метод, передавая возвращаемый сервисом объект в объект `com.google.gwt.user.client.rpc.AsyncCallback<T>`:

```
import com.google.gwt.user.client.rpc.AsyncCallback;
public interface MyServiceAsync {
    void methodServer(String arg, AsyncCallback<String> callback)
        throws IllegalArgumentException;
}
```



Такая схема предотвращает блокировку GUI-интерфейса клиента при взаимодействии с сервером.

Объект `AsyncCallback<T>` создается путем создания класса, как правило, анонимного, который реализует интерфейс `AsyncCallback<T>` с определением его методов `void onFailure(java.lang.Throwable caught)` и `void onSuccess(T result)`.

Метод `onFailure()` автоматически вызывается в случае возникновения ошибки взаимодействия с сервером, а метод `onSuccess()` — в случае удачного завершения асинхронного вызова сервиса.

Класс реализации интерфейса сервиса на стороне сервера представляет собой сервлет, расширяющий класс `com.google.gwt.user.server.rpc.RemoteServiceServlet`. Расширение класса `RemoteServiceServlet` дает то, что класс сервиса на первый взгляд ничем не напоминает сервлет. В нем не надо переопределять метод `doPost()` обработки HTTP POST-запроса, а требуется просто определить методы интерфейса сервиса:

```
import com.example.gwt.client.MyService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
@SuppressWarnings("serial")
public class MyServiceImpl extends RemoteServiceServlet implements
MyService {
    public String methodServer(String arg) throws IllegalArgumentException {
        return "...";
    }
}
```

За сценой при получении HTTP POST-запроса сервером вызывается метод `doPost(HttpServletRequest req, HttpServletResponse resp)` сервлета, который инициирует вызов метода `onBeforeRequestDeserialized(java.lang.String serializedRequest)` класса `RemoteServiceServlet`, проверяющего сериализованную версию тела запроса, затем идет вызов метода `processCall(java.lang.String payload)` класса `RemoteServiceServlet`, в котором вызывается собственно метод сервиса, после этого вызывается метод `onAfterResponseSerialized(java.lang.String serializedResponse)` класса `RemoteServiceServlet`, проверяющий сериализованную версию ответа, и ответ возвращается клиенту.

Так как класс сервиса все же является сервлетом, его необходимо объявить в дескрипторе `war/WEB-INF/web.xml` приложения:

```
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>com.example.gwt.server.MyServiceImpl</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>/gwtapplication/path</url-pattern>
</servlet-mapping>
```

Значение элемента `<url-pattern>` состоит из двух частей. Первая часть — это имя GWT-модуля, а вторая — относительный путь сервлета.

Относительный путь сервлета указывается также в интерфейсе сервиса с помощью аннотации `com.google.gwt.user.client.rpc.RemoteServiceRelativePath`:

```
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
@RemoteServiceRelativePath("path")
public interface MyService extends RemoteService {
    String methodServer(String arg) throws IllegalArgumentException;
}
```

Кроме того, для работы сервлета требуется наличие библиотеки `gwt-servlet.jar` в каталоге `war\WEB-INF\lib` приложения.

Для вызова сервлета на стороне клиента в методе `onModuleLoad()` `EntryPoint`-класса создается объект сервиса с помощью метода `GWT.create()`:

```
MyServiceAsync myService = GWT.create(MyService.class);
```

Затем производится вызов методов сервиса с обработкой результатов в объекте `AsyncCallback<T>`:

```
myService.greetServer(arg, new AsyncCallback<String>() {
    public void onFailure(Throwable caught) {
        . . .
    }
    public void onSuccess(String result) {
        . . .
    }
});
```

Хорошей практикой является кэширование объекта сервиса для предотвращения его повторного создания при многократных вызовах сервиса. Для этого объект сервиса создается в виде поля `EntryPoint`-класса:

```
private final MyServiceAsync myService = GWT.create(MyService.class);
```

Использование фреймворка GWT RPC подразумевает передачу данных между JavaScript-кодом клиента и Java-кодом сервера. И там, и там используются объекты, поэтому фреймворк GWT RPC обеспечивает сериализацию и десериализацию объектов для тела HTTP POST-запроса и HTTP-ответа.

Так как на стороне клиента работает JavaScript-код, GWT RPC-сериализация отличается от Java-сериализации, основанной на использовании интерфейса `java.io.Serializable`. Тип данных может быть сериализован фреймворком GWT RPC, если он соответствует следующим требованиям:

- ◆ тип данных является `char`, `byte`, `short`, `int`, `long`, `boolean`, `float`, `double`, `String`, `Date`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Float`, `Double`;
- ◆ тип данных — перечисление или массив сериализуемых типов;
- ◆ тип данных представляет собой сериализуемый класс или имеет, по крайней мере, один сериализуемый подкласс. Класс является сериализуемым, если он реализует интерфейс `com.google.gwt.user.client.rpc.IsSerializable` или интерфейс `java.io.Serializable`, все его нефинальные и невременные поля должны быть сериализуемыми, а также класс должен иметь конструктор по умолчанию;

- ♦ тип данных — класс, имеющий свой класс-сериализатор, на стороне клиента расширяющий класс `com.google.gwt.user.client.rpc.CustomFieldSerializer<T>`, на стороне сервера расширяющий класс `com.google.gwt.user.server.rpc.ServerCustomFieldSerializer<T>`. `CustomFieldSerializer`-класс должен находиться вместе с сериализуемым классом и иметь имя `[имя сериализуемого класса]_CustomFieldSerializer`, `ServerCustomFieldSerializer`-класс должен находиться на стороне сервера и иметь имя `[имя сериализуемого класса]_ServerCustomFieldSerializer`;
- ♦ тип данных не является коллекцией объектов `java.lang.Object`, т. к. класс `Object` не сериализуется.

В процессе GWT-компиляции фреймворк GWT RPC обеспечивает генерацию файла политики сериализации, который содержит список типов данных приложения, разрешенных для сериализации.

Помимо стандартного взаимодействия клиентской части GWT-приложения с его серверной частью, фреймворк GWT RPC позволяет создавать защищенное от атак типа "подделка межсайтовых запросов" Cross-Site Request Forgery (XSRF или CSRF) клиент-серверное приложение.

XSRF-защита фреймворка GWT RPC основана на включении определенного XSRF-маркера в каждый HTTP-вызов сервиса. По умолчанию XSRF-маркер генерируется как MD5-хэш cookie-сессии.

Для включения XSRF-защиты клиент-серверного GWT RPC-приложения необходимо дополнить конфигурацию дескриптора `web.xml`, класс сервиса должен расширять класс `com.google.gwt.user.server.rpc.XsrfProtectedServiceServlet`, а интерфейс сервиса — расширять интерфейс `com.google.gwt.user.client.rpc.XsrfProtectedService`.

Так как клиентская часть приложения получает XSRF-маркер путем вызова метода `getNewXsrfToken()` RPC-сервиса `com.google.gwt.user.client.rpc.XsrfTokenService`, при этом передавая cookie-сессию, в дескрипторе `web.xml` необходимо определить сервлет RPC-сервиса и контекстный параметр cookie-сессии:

```
<servlet-name>xsrf</servlet-name>
<servlet-class>
    com.google.gwt.user.server.rpc.XsrfTokenServiceServlet
</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>xsrf</servlet-name>
    <url-pattern>/gwtapplication/xsrf</url-pattern>
</servlet-mapping>
<context-param>
    <param-name>gwt.xsrf.session_cookie_name</param-name>
    <param-value>JSESSIONID</param-value>
</context-param>
```

Здесь значение элемента `<url-pattern>` состоит из двух частей. Первая часть — это имя GWT-модуля, а вторая — относительный путь сервлета `XsrfTokenServiceServlet`.

Интерфейс сервиса расширяет интерфейс `XsrfProtectedService`, который в свою очередь расширяет интерфейс `RemoteService`:

```
import com.google.gwt.user.client.rpc.XsrfProtectedService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
@RemoteServiceRelativePath("greet")
public interface GreetingService extends XsrfProtectedService {
    String greetServer(String name) throws IllegalArgumentException;
}
```

Другой способ XSRF-маркировки интерфейса сервиса — это использование аннотации `com.google.gwt.user.server.rpc.XsrfProtect`:

```
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.server.rpc.XsrfProtect;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
@XsrfProtect
@RemoteServiceRelativePath("greet")
public interface GreetingService extends RemoteService {
    String greetServer(String name) throws IllegalArgumentException;
}
```

Класс реализации сервиса должен расширять класс `XsrfProtectedServiceServlet`, который в свою очередь расширяет классы `AbstractXsrfProtectedServiceServlet` и `RemoteServiceServlet`:

```
import com.example.gwt.client.GreetingService;
import com.example.gwt.shared.FieldVerifier;
import com.google.gwt.user.server.rpc.XsrfProtectedServiceServlet;
@SuppressWarnings("serial")
public class GreetingServiceImpl extends XsrfProtectedServiceServlet implements
GreetingService {
    public String greetServer(String input) throws IllegalArgumentException {
        if (!FieldVerifier.isValidName(input)) {
            throw new IllegalArgumentException(
                "Name must be at least 4 characters long");
        }
        String serverInfo = getServletContext().getServerInfo();
        String userAgent = getThreadLocalRequest().getHeader("User-Agent");
        input = escapeHtml(input);
        userAgent = escapeHtml(userAgent);
        return "Hello, " + input + "!<br><br>I am running " + serverInfo
            + ".<br><br>It looks like you are using:<br>" + userAgent;
    }
    private String escapeHtml(String html) {
        if (html == null) {
            return null;
        }
        return html.replaceAll("&", "&amp;").replaceAll("<", "&lt;").replaceAll(">", "&gt;");
    }
}
```

В `EntryPoint`-классе для вызова XSRF-защищенного сервиса нужно установить cookie-сессию и получить XSRF-маркер:

```
import com.google.gwt.user.client.rpc.*;
...
String id="...";
com.google.gwt.user.client.Cookies.setCookie("JSESSIONID", id);
XsrfTokenServiceAsync xsrf = (XsrfTokenServiceAsync)GWT.create(XsrfTokenService.class);
((ServiceDefTarget)xsrf).setServiceEntryPoint(GWT.getModuleBaseURL() + "xsrf");
xsrf.getNewXsrfToken(new AsyncCallback<XsrfToken>() {
public void onSuccess(XsrfToken token) {
GreetingServiceAsync rpc = (GreetingServiceAsync)GWT.create(GreetingService.class);
((HasRpcToken) rpc).setRpcToken(token);
rpc.greetServer(textToServer, new AsyncCallback<String>() {
public void onFailure(Throwable caught) {
dialogBox.setText("Remote Procedure Call - Failure");
serverResponseLabel.addStyleName("serverResponseLabelError");
serverResponseLabel.setHTML(SERVER_ERROR);
dialogBox.center();
closeButton.setFocus(true);
}
public void onSuccess(String result) {
dialogBox.setText("Remote Procedure Call");
serverResponseLabel.removeStyleName("serverResponseLabelError");
serverResponseLabel.setHTML(result);
dialogBox.center();
closeButton.setFocus(true);
}});
}
public void onFailure(Throwable caught) {
try {
throw caught;
} catch (RpcTokenException e) {
} catch (Throwable e) {
}}
});
```

## Библиотека GWT HTTP Client

Библиотека GWT HTTP Client представлена пакетом `com.google.gwt.http.client` и обеспечивает создание HTTP DELETE, GET, HEAD, POST, PUT асинхронных запросов клиентской части GWT-приложения.

Для использования библиотеки GWT HTTP Client в файл GWT-модуля необходимо включить наследование:

```
<inherits name="com.google.gwt.http.HTTP" />
```

Главным классом пакета `com.google.gwt.http.client` является класс `RequestBuilder`, позволяющий отправлять HTTP-запрос указанным методом по указанному адресу и получать ответ асинхронным обработчиком. Экземпляр класса `RequestBuilder` создается с помощью конструктора:

```
RequestBuilder(RequestBuilder.Method httpMethod, java.lang.String url)
```

принимającego в качестве аргументов метод запроса и адрес запроса.

Метод запроса указывается с помощью полей `DELETE`, `GET`, `HEAD`, `POST`, `PUT` внутреннего класса `RequestBuilder.Method`, а адрес запроса может быть создан с использованием класса `com.google.gwt.http.client.URL` или класса `com.google.gwt.http.client.UrlBuilder`.

Класс `URL` обеспечивает URL-кодировку адресной строки с помощью статических методов `encode(java.lang.String decodedURL)`, `encodePathSegment(java.lang.String decodedURLComponent)` и `encodeQueryString(java.lang.String decodedURLComponent)`.

Класс `UrlBuilder` позволяет конструировать URL-адрес из компонентов с помощью набора методов:

- ◆ `removeParameter(java.lang.String name)` — удаляет параметр из адреса;
- ◆ `setHash(java.lang.String hash)` — устанавливает хэш-фрагмент адреса;
- ◆ `setHost(java.lang.String host)` — устанавливает хост адреса;
- ◆ `setParameter(java.lang.String key, java.lang.String... values)` — устанавливает параметры адреса;
- ◆ `setPath(java.lang.String path)` — устанавливает путь адреса;
- ◆ `setPort(int port)` — устанавливает порт адреса;
- ◆ `setProtocol(java.lang.String protocol)` — устанавливает протокол адреса;
- ◆ `buildString()` — кодирует адрес в строку.

Экземпляр класса `UrlBuilder` создается с помощью конструктора класса без аргументов.

Метод `sendRequest(java.lang.String requestData, RequestCallback callback)` класса `RequestBuilder` посылает HTTP-запрос. В качестве аргумента данный метод принимает параметры запроса и асинхронный обработчик ответа.

При использовании метода `POST` параметры запроса кодируются в строку в виде `параметр=значение&параметр=значение...`, при этом с помощью метода `setHeader(java.lang.String header, java.lang.String value)` класса `RequestBuilder` устанавливается заголовок `Content-type` запроса `application/x-www-form-urlencoded`.

Асинхронный обработчик ответа должен реализовывать интерфейс `com.google.gwt.http.client.RequestCallback` с определением его методов `onError(Request request, java.lang.Throwable exception)` — обработка ошибки запроса, `onResponseReceived(Request request, Response response)` — обработка результатов запроса.

Класс `com.google.gwt.http.client.Response` обеспечивает доступ к деталям HTTP-ответа с помощью методов:

- ◆ `getHeader(java.lang.String header)`, `getHeaders()` и `getHeadersAsString()` — извлекают заголовки ответа;
- ◆ `getStatusCode()` — возвращает код статуса ответа;
- ◆ `getStatusText()` — возвращает текст статуса ответа;
- ◆ `getText()` — возвращает данные ответа.

**Методы** `setPassword(java.lang.String password)`, `setUser(java.lang.String user)` и `setTimeoutMillis(int timeoutMillis)` класса `RequestBuilder` позволяют добавить в запрос логин, пароль и время ожидания ответа.

В качестве примера рассмотрим модификацию сгенерированной в среде Eclipse основы GWT-проекта использования фреймворка GWT RPC на применение библиотеки GWT HTTP Client.

При использовании библиотеки GWT HTTP Client интерфейсы `GreetingService` и `GreetingServiceAsync` не нужны, а класс `GreetingServiceImpl` можно изменить следующим образом:

```
import java.io.IOException;
import javax.servlet.http.*;
@SuppressWarnings("serial")
public class GreetingServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp){
        String input=req.getParameter("name");
        String serverInfo = getServletContext().getServerInfo();
        String userAgent = req.getHeader("User-Agent");
        try {
            resp.getWriter().println( "Hello, " + input + "<br><br>I am running " + serverInfo +
            ".<br><br>It looks like you are using:<br>" + userAgent);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**web.xml:**

```
<servlet>
<servlet-name>greetServlet</servlet-name>
<servlet-class>com.example.gwt.server.GreetingServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>greetServlet</servlet-name>
<url-pattern>/greet</url-pattern>
</servlet-mapping>
```

В `EntryPoint`-классе следующий код будет обеспечивать взаимодействие с сервлетом приложения:

```
String url = "/greet";
RequestBuilder builder = new RequestBuilder(RequestBuilder.POST, URL.encode(url));
StringBuffer postData = new StringBuffer();
```

```

postData.append(URL.encode("name")).append("=").append(URL.encode(textToServer));
builder.setHeader("Content-type", "application/x-www-form-urlencoded");
try {
    uilder.sendRequest(postData.toString(), new RequestCallback() {
        public void onError(Request request, Throwable exception) { }
        public void onResponseReceived(Request request, Response response) {
            if (200 == response.getStatusCode()) {
                String result=response.getText();
                dialogBox.setText("Remote Procedure Call");
                serverResponseLabel.removeStyleName("serverResponseLabelError");
                serverResponseLabel.setHTML(result);
                dialogBox.center();
                closeButton.setFocus(true);
            } else { }
        }
    });
} catch (RequestException e) {
}

```

При обмене с сервером данных в XML-формате фреймворк GWT предлагает использовать библиотеку `com.google.gwt.xml.client` для разбора и генерации XML-документов на стороне клиента.

Для использования библиотеки `com.google.gwt.xml.client` в файл GWT-модуля необходимо включить наследование:

```
<inherits name="com.google.gwt.xml.XML" />
```

Основным классом пакета `com.google.gwt.xml.client` является класс `XMLParser`, обеспечивающий с помощью метода `parse(java.lang.String contents)` разбор XML-документа, а с помощью метода `createDocument()` — его создание. Оба метода возвращают объект `Document`, своими методами позволяющий извлекать и наполнять XML-документ элементами:

- ◆ `CDATASection createCDATASection(java.lang.String data)` — создает секцию CDATA;
- ◆ `Comment createComment(java.lang.String data)` — создает комментарий;
- ◆ `DocumentFragment createDocumentFragment()` — создает контейнер элементов;
- ◆ `Element createElement(java.lang.String tagName)` — создает элемент;
- ◆ `ProcessingInstruction createProcessingInstruction(java.lang.String target, java.lang.String data)` — создает инструкцию;
- ◆ `Text createTextNode(java.lang.String data)` — создает текстовый элемент;
- ◆ `Element getDocumentElement()` — возвращает элемент документа;
- ◆ `Element getElementById(java.lang.String elementId)` — возвращает элемент с указанным идентификатором;
- ◆ `NodeList getElementsByTagName(java.lang.String tagName)` — возвращает элементы с указанным именем;
- ◆ `Node importNode(Node importedNode, boolean deep)` — импортирует узел в документ.



В рассмотренном примере в случае возврата сервлетом ответа на HTTP-запрос в XML-формате, код сервлета изменится следующим образом:

```
import java.io.*;
import javax.servlet.http.*;
@SuppressWarnings("serial")
public class GreetingServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp) {
        String input=req.getParameter("name");
        String serverInfo = getServletContext().getServerInfo();
        String userAgent = req.getHeader("User-Agent");
        try {
            resp.setContentType("text/xml");
            PrintWriter out = resp.getWriter();
            out.println("<?xml version='1.0'?>");
            out.println("<greeting language='en_US'?>");
            out.println("Hello, " + input + "!, I am running " + serverInfo
                + ".It looks like you are using:" + userAgent);
            out.println("</greeting>");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Для разбора ответа сервлета в EntryPoint-классе можно использовать следующий код:

```
String url = "/greet";
RequestBuilder builder = new RequestBuilder(RequestBuilder.POST, URL.encode(url));
StringBuffer postData = new StringBuffer();
postData.append(URL.encode("name")).append("=").append(URL.encode(textToServer));
builder.setHeader("Content-type", "application/x-www-form-urlencoded");
try {
    builder.sendRequest(postData.toString(), new RequestCallback() {
        public void onError(Request request, Throwable exception) {
        }
        public void onResponseReceived(Request request, Response response) {
            if (200 == response.getStatusCode()) {
                String result=response.getText();
                Document resultXML = XMLParser.parse(result);
                Text greetingNode = (Text)
resultXML.getElementsByTagName("greeting").item(0).getFirstChild();
                String greetingText = greetingNode.getData();
                dialogBox.setText("Remote Procedure Call");
                serverResponseLabel.removeStyleName("serverResponseLabelError");
                serverResponseLabel.setHTML(greetingText);
                dialogBox.center();
                closeButton.setFocus(true);
            }
        }
    });
}
```

```

    } else {
    }
  });
} catch (RequestException e) { }
```

При обмене с сервером данных в JSON-формате фреймворк GWT предлагает использовать библиотеку `com.google.gwt.json.client` для разбора и генерации JSON-документов на стороне клиента.

Для использования библиотеки `com.google.gwt.json.client` в файл GWT-модуля необходимо включить наследование:

```
<inherits name="com.google.gwt.json.JSON" />
```

Основным классом пакета `com.google.gwt.json.client` является класс `JSONParser`, обеспечивающий с помощью метода `parseStrict(java.lang.String jsonString)` разбор JSON-документа. Данный метод возвращает объект `JSONValue`. Так как JSON-документ представляет собой набор пар "свойство — значение", для дальнейшего разбора JSON-документа из объекта `JSONValue` получается объект `JSONObject` с использованием метода `isObject()` класса `JSONValue`.

Объект `JSONObject` содержит JSON-свойства, извлечь которые можно методом `get(java.lang.String key)`. Данный метод возвращает объект `JSONValue`, своими методами позволяющий извлекать значения свойств различных типов:

- ◆ `JSONArray isArray()` — возвращает массив;
- ◆ `JSONBoolean isBoolean()` — возвращает логическое значение;
- ◆ `JSONNumber isNumber()` — возвращает число;
- ◆ `JSONString isString()` — возвращает строку.

В рассмотренном ранее примере в случае возврата сервлетом ответа на HTTP-запрос в JSON-формате, код сервлета изменится следующим образом:

```

import java.io.*;
import javax.servlet.http.*;
@SuppressWarnings("serial")
public class GreetingServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp) {
        String input=req.getParameter("name");
        String serverInfo = getServletContext().getServerInfo();
        String userAgent = req.getHeader("User-Agent");
        try {
            PrintWriter out = resp.getWriter();
            out.println("{}");
            out.println("{\"greeting\":\"Hello, " + input + "!, I am running " +
                serverInfo + ".It looks like you are using:" + userAgent+"\""}");
            out.println("{}");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Для разбора ответа сервлета в `EntryPoint`-классе можно использовать следующий код:

```
String url = "/greet";
RequestBuilder builder = new RequestBuilder(RequestBuilder.POST, URL.encode(url));
StringBuffer postData = new StringBuffer();
postData.append(URL.encode("name")).append("=").append(URL.encode(textToServer));
builder.setHeader("Content-type", "application/x-www-form-urlencoded");
try {
builder.sendRequest(postData.toString(), new RequestCallback() {
public void onError(Request request, Throwable exception) { }
public void onResponseReceived(Request request, Response response) {
if (200 == response.getStatusCode()) {
String result=response.getText();
JSONValue value = JSONParser.parseStrict(result);
JSONObject greetingObj = value.isObject();
JSONString greetingStr = greetingObj.get("greeting").isString();
String greetingText = greetingStr.stringValue();
dialogBox.setText("Remote Procedure Call");
serverResponseLabel.removeStyleName("serverResponseLabelError");
serverResponseLabel.setHTML(greetingText);
dialogBox.center();
closeButton.setFocus(true);
} else { }
});
} catch (RequestException e) { }
```

## Фреймворк RequestFactory

Фреймворк `RequestFactory` представляет собой альтернативу фреймворку `GWT RPC` для организации взаимодействия клиентской части GWT-приложения с его серверной частью.

Если при использовании фреймворка `GWT RPC` реализуется общая концепция Web-сервисов, подразумевающая создание интерфейса сервиса, класса реализации интерфейса сервиса и клиентского класса-заглушки вызова сервиса, то при применении фреймворка `RequestFactory` создается модель данных "Entity-сущность, заглушка сущности `EntityProxy`, интерфейс `RequestFactory` и интерфейс `RequestContext`". Поэтому фреймворк `GWT RPC` называют сервис-ориентированным, а фреймворк `RequestFactory` — ориентированным на данные.

Entity-сущность представляет сохраняемые данные приложения и может содержать методы соединения с базой данных:

```
package data;
import java.io.Serializable;
import javax.persistence.*;
import javax.validation.constraints.*;
```

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import java.util.List;
@Entity
@Table(name = "PERSON")
public class Person implements Serializable {
    @Id
    @GeneratedValue
    @Column(name = "PERSONID")
    private Long Id;
    @Column(name = "FIRSTNAME")
    @NotNull
    @Size(min=3, max=16)
    private String firstName;
    @Column(name = "FAMILYNAME")
    @NotNull
    @Size(min=3, max=16)
    private String familyName;
    @Version
    @Column(name = "VERSION")
    private Integer version;
    public Long getId() {
        return Id;
    }
    public void setId(Long id) {
        this.Id = id;
    }
    public void setfirstName(String input){
        firstName=input;
    }
    public void setfamilyName(String input){
        familyName=input;
    }
    public String getfirstName(){
        return firstName;
    }
    public String getfamilyName(){
        return familyName;
    }
    public Integer getVersion() {
        return this.version;
    }
    public void setVersion(Integer version) {
        this.version = version;
    }
}
```

```
@SuppressWarnings("unchecked")
public static List<Person> getAllPerson() {
    org.hsqldb.server.Server server=new org.hsqldb.server.Server();
    org.hsqldb.persist.HsqlProperties p = new org.hsqldb.persist.HsqlProperties();
    p.setProperty("server.database.0", "/db/testdb");
    p.setProperty("server.dbname.0", "TESTDB");
    p.setProperty("ifexists", true);
    p.setProperty("shutdown", true);
    try {
        server.setProperties(p);
    } catch (Exception e) {
        e.printStackTrace();
    }
    server.setLogWriter(null);
    server.setErrWriter(null);
    server.start();
    Configuration conf=new Configuration();
    conf.addAnnotatedClass(Person.class);
    SessionFactory sessionFactory = conf.configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    List<Person> allPerson=null;
    try {
        transaction = session.beginTransaction();
        allPerson=session.createQuery("select person from Person person").list();
        transaction.commit();
    } catch (Exception e) {
        transaction.rollback();
        e.printStackTrace();
    }
    server.shutdown();
    session.close();
    server.stop();
    return allPerson;
}

public static Person findPerson(Long id) {
    if (id == null) {
        return null;
    }
    org.hsqldb.server.Server server=new org.hsqldb.server.Server();
    org.hsqldb.persist.HsqlProperties p = new org.hsqldb.persist.HsqlProperties();
    p.setProperty("server.database.0", "/db/testdb");
    p.setProperty("server.dbname.0", "TESTDB");
    p.setProperty("ifexists", true);
    p.setProperty("shutdown", true);
    try {
        server.setProperties(p);
```

```

    } catch (Exception e) {
        e.printStackTrace();
    }
    server.setLogWriter(null);
    server.setErrWriter(null);
    server.start();
    Configuration conf=new Configuration();
    conf.addAnnotatedClass(Person.class);
    SessionFactory sessionFactory = conf.configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    Person person=null;
    try {
        transaction = session.beginTransaction();
        person = (Person) session.get(Person.class, id);
        transaction.commit();
    } catch (Exception e) {
        transaction.rollback();
        e.printStackTrace();
    }
    server.shutdown();
    session.close();
    server.stop();
    return person;
}

public void persist() {
    org.hsqldb.server.Server server=new org.hsqldb.server.Server();
    org.hsqldb.persist.HsqlProperties p = new org.hsqldb.persist.HsqlProperties();
    p.setProperty("server.database.0", "/db/testdb");
    p.setProperty("server.dbname.0", "TESTDB");
    p.setProperty("ifexists", true);
    p.setProperty("shutdown", true);
    try {
        server.setProperties(p);
    } catch (Exception e) {
        e.printStackTrace();
    }
    server.setLogWriter(null);
    server.setErrWriter(null);
    server.start();
    Configuration conf=new Configuration();
    conf.addAnnotatedClass(Person.class);
    SessionFactory sessionFactory = conf.configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    try {
        transaction = session.beginTransaction();
        Person person = (Person) session.get(Person.class, this.Id);

```

```

        this.version=person.getVersion();
        person.setfirstName(this.firstName);
        person.setfamilyName(this.familyName);
        transaction.commit();
    } catch (Exception e) {
        transaction.rollback();
        e.printStackTrace();
    }
    server.shutdown();
    session.close();
    server.stop();
}}

```

Реализовать код сохранения данных можно не в самой Entity-сущности, а в отдельном классе, расширяющем класс `com.google.web.bindery.requestfactory.shared.Locator<T,I>`.

На стороне клиента создается заглушка сущности:

```

package test.client;
import com.google.web.bindery.requestfactory.shared.EntityProxy;
import com.google.web.bindery.requestfactory.shared.ProxyFor;
@ProxyFor(data.Person.class)
public interface PersonProxy extends EntityProxy {
    Long getId();
    void setId(Long id);
    void setfirstName(String input);
    void setfamilyName(String input);
    String getfirstName();
    String getfamilyName();
    Integer getVersion();
    void setVersion(Integer version);
}

```

Фреймворк `RequestFactory` автоматически связывает свойства Entity-сущности на стороне сервера и `EntityProxy`-заглушки на стороне клиента.

Для встроенных объектов с аннотацией `@Embedded` создается интерфейс-заглушка, расширяющая интерфейс `com.google.web.bindery.requestfactory.shared.ValueProxy`.

Для связывания клиентского и серверного кода создается интерфейс-фабрика:

```

package test.client;
import com.google.web.bindery.requestfactory.shared.RequestFactory;
public interface PersonRequestFactory extends RequestFactory{
    PersonRequest personRequest();
}

```

А также заглушка сервиса:

```

package test.client;
import com.google.web.bindery.requestfactory.shared.*;

```

```
import java.util.List;
@Service(data.Person.class)
public interface PersonRequest extends RequestContext{
    Request<List<PersonProxy>> getAllPerson();
    InstanceRequest<PersonProxy, Void> persist();
}
```

При использовании Locator-класса в аннотации @ProxyFor(value = [].class, locator = [].class) и @Service(value = [].class, locator = [].class) добавляется определение Locator-класса.

Фреймворк RequestFactory накладывает ограничения на передаваемые типы данных:

- ◆ BigDecimal, BigInteger, Boolean, Byte, Enum, Character, Date, Double, Float, Integer, Long, Short, String, Void и их коллекции List<T>, Set<T>;
- ◆ классы, расширяющие классы EntityProxy, ValueProxy и их коллекции List<T>, Set<T>.

В EntryPoint-классе для вызова серверной части приложения создаются объекты RequestFactory и RequestContext, с помощью которого создается объект EntityProxy. Для объекта EntityProxy устанавливаются свойства, после чего он передается на сервер:

```
EventBus eventBus = new SimpleEventBus();
PersonRequestFactory requestFactory = GWT.create(PersonRequestFactory.class);
requestFactory.initialize(eventBus);
PersonRequest request = requestFactory.personRequest();
PersonProxy newPerson = request.create(PersonProxy.class);
newPerson.setId(object.getId());
newPerson.setfamilyName(value);
newPerson.setfirstName(object.getfirstName());
request.persist().using(newPerson).fire();
```

Для работы фреймворка RequestFactory в файл GWT-модуля необходимо включить наследование:

```
<inherits name='com.google.web.bindery.requestfactory.RequestFactory' />
```

Кроме того, в папку WEB-INF\lib проекта приложения нужно добавить библиотеки requestfactory-server.jar, validator-api-1.0.0.GA.jar и библиотеки набора инструментов Hibernate Validator (<http://www.hibernate.org/subprojects/validator/download>).

В дескрипторе web.xml приложения требуется указать сервлет RequestFactoryServlet:

```
<servlet>
<servlet-name>gwtRequest</servlet-name>
<servlet-
class>com.google.web.bindery.requestfactory.server.RequestFactoryServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>gwtRequest</servlet-name>
```



```
<url-pattern>/gwtRequest</url-pattern>  
</servlet-mapping>
```

Перед вызовом сервера фреймворк RequestFactory осуществляет проверку Entity-сущности, поэтому в папке `war\WEB-INF\classes\test\client` требуется наличие класса `PersonRequestFactoryDeobfuscatorBuilder.class`.

Получить класс `PersonRequestFactoryDeobfuscatorBuilder` можно, упаковав пакет приложения в `jar`-файл с помощью команды:

```
jar cf factory.jar test/* data/*
```

С последующей обработкой его инструментом `com.google.web.bindery.requestfactory.apt.ValidationTool`:

```
java -cp requestfactory-apt.jar;requestfactory-server.jar;factory.jar  
com.google.web.bindery.requestfactory.apt.ValidationTool factory.jar  
com.example.gwt.client.PersonRequestFactory
```

# Литература

1. Ahammad S. Google Web Toolkit 2 Application Development Cookbook. — Birmingham: Packt Publishing, 2010.
2. Chaganti P. Google Web Toolkit GWT Java AJAX Programming: A step-by-step to Google Web Toolkit for creating Ajax applications fast. — Birmingham: Packt Publishing, 2007.
3. Cooper R. GWT in Practice. — Manning Publications, 2008.
4. De Jonge A. Essential App Engine: Building High-Performance Java Apps with Google App Engine. — Addison-Wesley Professional, 2011.
5. Dewsbury R. Google Web Toolkit Applications. — Addison-Wesley Professional, 2007.
6. Guermeur D., Unruh A. Google App Engine Java and GWT Application Development. — Birmingham: Packt Publishing, 2010.
7. Kereki F. Essential GWT: Building for the Web with Google Web Toolkit 2. — Addison-Wesley Professional, 2010.
8. Sanderson D. Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure. — O'Reilly Media, 2009.
9. Smeets B., Boness U., Bankras R. Beginning Google Web Toolkit: From Novice to Professional. — Apress, 2008.

# Предметный указатель

## A

Activities and Places 324  
App Engine Datastore 65

## B

Blobstore 128

## C

Channel, служба 154  
Cookies 139  
Cron, служба 175

## D

Datastore API 67

## G

Google Apps Marketplace, сервис 16  
Google Cloud SQL 123  
Google Web Toolkit (GWT) 195  
GWT Designer, редактор 308  
GWT HTTP Client, библиотека 335  
GWT RPC 330

## I

ID-селектор 304

## J

Java Data Objects (JDO) 82  
Java Persistence API (JPA) 82

JavaMail API 140  
JavaScript Native Interface (JSNI) 314  
JavaServer Pages (JSP) 54  
JavaServer Pages Standard Tag Library (JSTL)  
57

## L

Log4j, библиотека 37  
LogService, сервис 38

## M

Memcache Java API 134

## O

OAuth, протокол 168  
Objectify-Appengine 100  
OpenID 50

## P

Project Hosting, сервис 16  
Prospective Search, служба 187

## R

Remote API, служба 80  
RequestFactory 341

## S

Search, служба 183  
Slim3 116  
Speed Tracer, инструмент 212

**Т**

Task Queue, служба 177  
Twig 108

**U**

UiBinder 304  
Users API 48

**X**

XMPP, служба 146

**A**

Аудиокомпонент 285  
Аутентификация/авторизация  
пользователей 45

**B**

Видеокомпонент 284  
Выбор даты 226

**Г**

Гиперссылка 234, 236

**Д**

Дерево 257, 259, 263

**З**

Загрузчик файлов 279

**И**

Изображение 281  
Интернационализация 307  
История Web-браузера 321

**К**

Кнопка 215, 222, 229  
◊ обработки событий 216  
◊ свойства 222

**Компонент:**

◊ AbsolutePanel 287  
◊ Anchor 236  
◊ Audio 285  
◊ Button 215  
◊ Canvas 286  
◊ CaptionPanel 299  
◊ Cell 238  
◊ CellBrowser 259  
◊ CellList 239  
◊ CellTable 249  
◊ CellTree 257  
◊ CheckBox 225  
◊ DataGrid 255  
◊ DatePicker 226  
◊ DecoratedPanel 297  
◊ DecoratedPopupPanel 274  
◊ DecoratedStackPanel 291  
◊ DialogBox 276  
◊ DisclosurePanel 299  
◊ DockLayoutPanel 293  
◊ DockPanel 292  
◊ FileUpload 279  
◊ FlexTable 269  
◊ FlowPanel 295  
◊ FocusPanel 297  
◊ FormPanel 279  
◊ Frame 281  
◊ Grid 271  
◊ HeaderPanel 296  
◊ Hidden 280  
◊ HorizontalPanel 294  
◊ HTMLPanel 295  
◊ Hyperlink 234  
◊ Image 281  
◊ Label 283  
◊ LayoutPanel 289  
◊ ListBox 237  
◊ LoggingPopup 274  
◊ MenuBar 260  
◊ NotificationMole 276  
◊ PasswordTextBox 232  
◊ PopupPanel 273  
◊ PushButton 222  
◊ RadioButton 224  
◊ RichTextArea 267  
◊ RootLayoutPanel 288  
◊ RootPanel 288  
◊ ScrollPanel 298  
◊ SimpleLayoutPanel 298

Компонент (*npod.*):

- ◇ SimplePanel 296
- ◇ SplitLayoutPanel 294
- ◇ StackLayoutPanel 291
- ◇ StackPanel 290
- ◇ SuggestBox 265
- ◇ TabLayoutPanel 277
- ◇ TabPanel 276
- ◇ TextArea 233
- ◇ TextBox 231
- ◇ ToggleButton 229
- ◇ Tree 263
- ◇ VerticalPanel 294
- ◇ Video 284

## М

Местоположение пользователя 42

Метка 283, 284

## О

Обработка ошибок 161

Окно всплывающее 273

Очередь 177

## П

Панель:

- ◇ компоновки 287
- ◇ меню 260
- ◇ с закладками 276

Переключатель 224

Плагины, установка 196

Поле:

- ◇ ввода пароля 232
- ◇ подсказки 265
- ◇ скрытое 280
- ◇ текстовое 231

Приложение:

- ◇ загрузка в App Engine 31
- ◇ запуск из среды Eclipse 23
- ◇ регистрация 25
- ◇ создание проекта 18

## Р

Разделение кода 316

Редактор текста 267

## С

Связывание, отложенное 318

Сессия 138

Сообщение:

- ◇ запись в журнал 34
- ◇ мгновенное:
  - отправка 147
  - получение 147
- ◇ электронной почты:
  - отправка 140
  - получение 141

Список выбора 237

Столбец 239

## Т

Таблица 249, 255, 269, 271

Текстовая область 233

## У

Уведомление 276

Установка инструментов разработки 13

## Ф

Фильтрация запросов и ответов 159

Флажок 225

Форма 279

Фрейм 281

## Х

Холст 286

Хранение данных 65

## Я

Ячейка 238