Оглавление

О учебнике

Это неофициальный перевод руководства по использованию инструмента Google Web Toolkit. Учебник создан по личной инициативе переводчиков без содействия компании Google. Сам проект Google Web Toolkit находится по этому адресу (http://code.google.com/webtoolkit/).

Учебник написан по версии GWT 1.6, для GWT версии 2.0 или выше желательно использовать официальную документацию так как версия GWT 2.0 имеет значительные отличия от первой версии.[1]

Вы уважаемый читатель можете внести свой вклад в данный учебник прислав свои замечания и пожелания или дополнения к данному учебнику по адресу указанному на сайте http://gocha.org (раздел Контакты).

Перевод осуществлен Камневым Георгием и Павлом Вязанкиным (pavel.vyazankin)

Учебник GWT

Это учебник по основам Google Web Toolkit (GWT), пошаговые уроки помогут вам создать учебное приложение просмотра состояния акций, данное приложение мы назовем "StockWatcher". Как нам построить приложение, вы узнаете из каждой части этого учебника GWT, в том числе библиотеку виджетов, работу в режиме отладки, вызов удаленных процедур Remote Procedure Calls, интернационализации, а также многое другое.

1 На сайте http://gocha.org имеются дополнительные материалы связанные с новой версией GWT

Установка GWT

Установка JAVA SDK

Если вы не имеете свежую версию JAVA SDK (версия 1.5 или выше) установленную на вашем компьютере, то скачайте и установите с сайта SUN (http://java.sun.com/javase/downloads/).

Скачайте GWT

Скачайте GWT дистрибутив для вашей операционной системы (http://code.google.com/webtoolkit/download.html)

Распакуйте GWT дистрибут

На Windows, вы можете распаковать файлы из gwt-windows-1.4.60.zip с помощью программы WinZip (или 7zip) (http://www.winzip.com/)

На Mac, вы можете распаковать дистрибут с помощью командой строки:

```
tar xvzf gwt-mac-1.4.60.tar.gz
```

На Linux, используйте команду:

```
tar xvf gwt-linux-1.4.60.tar.bz2
```

Все, GWT не требует использования программы установки, все файлы необходимые для запуска и работы GWT расположены в распакованном каталоге.

Добавьте GWT каталог с приложениями в ваши «пути»

Для использования утилит командной строки необходимо добавить каталог с распакованным GWT дистрибутивом в системные пути.

Для Windows пользователей выполните следующие действия:

- Правый клик на Мой-компьютер, и выберете Свойства
- Откройте вкладку Дополнительно
- Кликнете на кнопке Переменные окружения
- Выберете PATH из пользовательских переменных окружения и нажмите редактировать
- Вставьте в конец значения переменной точку с запятой и затем полный путь к каталогу, где распакован дистрибутив GWT (например: C:\gwt-windows-1.4.61\)

На Mac или Linux вам необходимо отредактировать файл .profile или .bash_profile в вашей домашней директории.

Для примера, если вы распаковали GWT в /home/user/gwt-linux-1.4.61/, то добавьте в

профиль следующие строчки:

```
PATH=$PATH:/home/user/gwt-linux-1.4.61/
export PATH
```

После редактирования вам необходимо выйти из сеанса пользователя и зайти снова, чтобы настройки профиля возымели эффект.

Создание GWT проекта

Давайте начнем, создадим директорию куда положим наш GWT проект. Создаем директорию с именем StockWatcher под главной директорий GWT (gwt-windows-1.4.60). Создавая наш проект мы будем использовать утилиты командной строки, которые поставляются с GWT. Эти утилиты генерируют файлы и поддиректории необходимы для старта.

Создание Eclipse проекта с projectCreator

Если вы используете Eclipse, то запустите первой утилиту projectCreator. Если вы не используете Eclipse, то переходите к следующему пункту.

Утилита projectCreator генерирует Eclipse проект нашего приложения. Откройте командую строку и перейдите к каталогу StockWatcher, который вы создали. Запустите команду:

```
projectCreator –eclipse StockWatcher –out StockWatcher
```

После того как утилита отработает, появится новый подкаталог StockWatcher, с подкаталогами src для исходных текстов, и test для модульного тестирования, а так же файлы Eclipse (.project и .classpath).

Создание GWT проекта с applicationCreator

Следующая утилита applicationCreator сгенерирует минимальное приложение GWT. Утилите необходимо указать полное имя главного (входной точки, оно же main класс) класса. Используйте GWT рекомендации структуры проекта. Данный класс всегда должен быть внутри под пакета client. Для приложения StockWatcher имя нашего главного класса будет: com.google.gwt.sample.stockwatcher.client.StockWatcher.

Утилита applicationCreator может также с генерировать конфигурационный файл Eclipse для запуска режима отладки.

Укажите флаг -eclipse следующий перед имением Eclipse проекта.

В командной строке, перейдите к каталогу StockWatcher. Если вы используете Eclipse, выполните команду:

```
applicationCreator –eclipse StockWatcher –out StockWatcher
com.google.gwt.sample.stockwatcher.client.StockWatcher
```

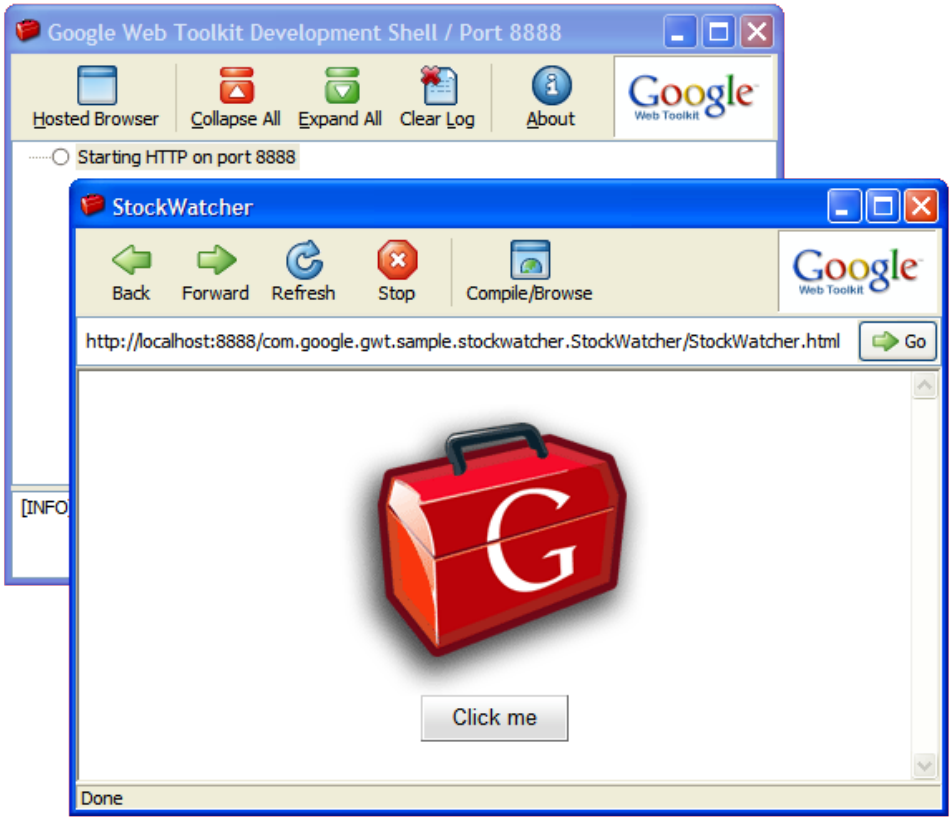
Если вы используете другую IDE, то опустите флаг -eclipse и имя проекта.

```
applicationCreator –out StockWatcher
com.google.gwt.sample.stockwatcher.client.StockWatcher
```

Теперь загляните во внутрь директории StockWatcher. Вы уведите что applicationCreator сгенерировал два скрипта: StockWatcher-compile и StockWatcher-shell. Эти скрипты помогут для распространения (deploy) и запуска приложения в режиме хоста (host-mode).

Тестирование генерированного проекта

Попробуем запустить приложение в режиме хоста. Для этого запускаем скрипт StockWatcher-shell. Вы увидите два окна. Первое окно Development shell, содержит лог запуска приложения и там же появляются сообщения об ошибках. Второе окно содержит встроенный браузер, где вы можете наблюдать работу вашего приложения в режиме хоста, взаимодействовать с ним. В данном режиме работа приложения будет точно такой же как и в конечном, развернутом приложении.

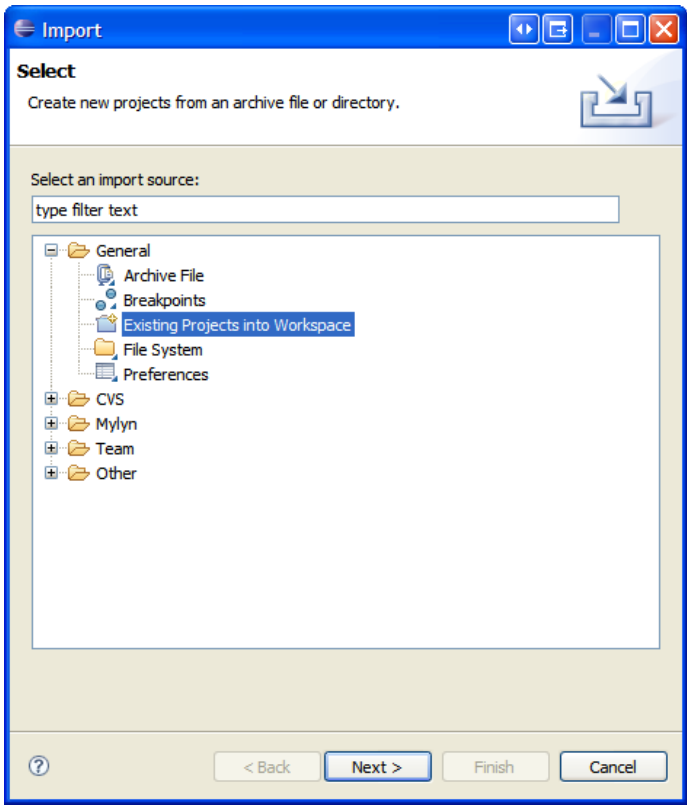


Приложение в режиме хоста означает следующее: что приложение не транслирует в конечный Javascript код. Оно заменяется виртуальной машиной JAVA и исполняется как скомпилированный байт код Java. GWT встраивает данный код в браузер. Это дает возможность отладки приложения используя исходный код в вашей IDE, как любого другого Java приложения. Цикл разработки принимает стандартный вид «кодирование, тестирование, отладка», что даст прирост в скорости разработки приложения.

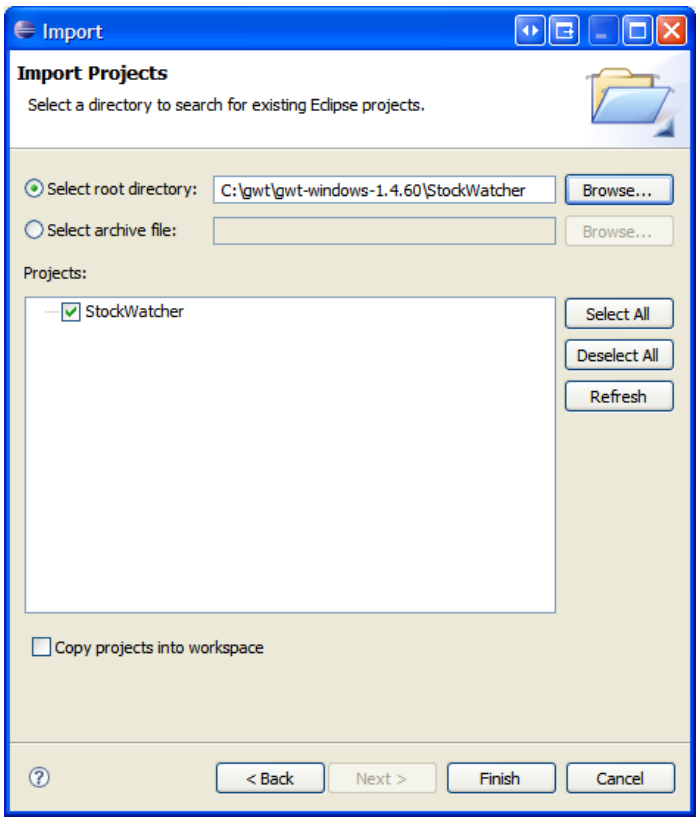
applicationCreator может генерировать конфигурацию проекта для последующей отладки его в режиме хоста, с помощью Eclipse. Что бы использовать эту возможность испортируйте проект в Eclipse.

Импорт проекта в Eclipse

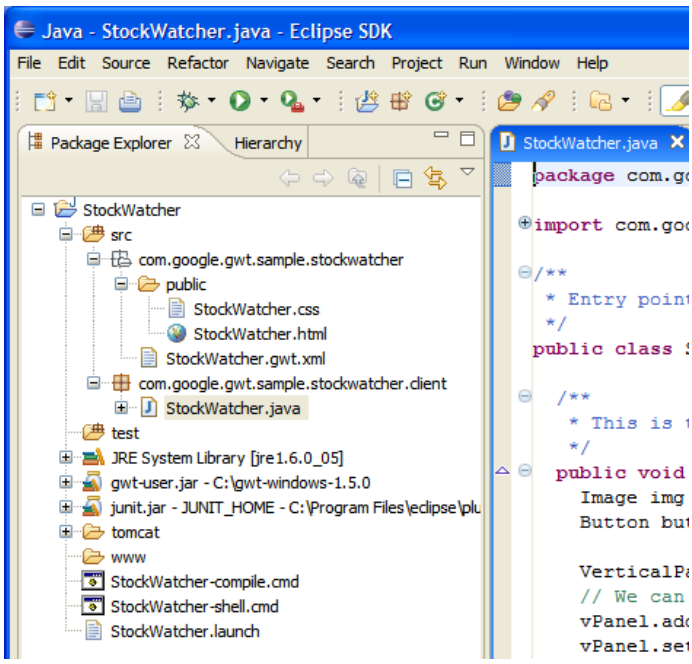
Что бы открыть проект в Eclipse, запустите сначала Eclipse и щелкните меню File → Import. Выберете Existsing Project into Workspace на первом окне мастера.



Затем выберети директорию которая содержит файл .project



Щелкните Finish и вы увидете что GWT проект появиться в Eclipse.



Для запуска проекта в режиме хоста выберете проект в Package Exp lorer и кликнете зеленую кнопку запуска в панели инструментов (или выберете в меню Run → Run).

Внутри GWT проекта

Теперь вы имеете начальную версию приложения StockWatcher, готового к запуску, давайте поговорим о GWT проекте. Вернитей назад к директории StockWatcher, утилита applicationCreator создала директории с вложенным каталогом src где и содержится исходный код приложения.

Модули

Наше приложение содержится в пакете src/com/google/gwt/sample/stockwatcher, давайте начнем с этого места. Найдите один файл под названием StockWatcher.gwt.xml:

```
<module>
      <!-- Inherit the core Web Toolkit stuff. -->
      <inherits name='com.google.gwt.user.User'/>
      <!-- Inherit the default GWT style sheet. You can change -->
      <!-- the theme of your GWT application by uncommenting -->
      <!-- any one of the following lines. -->
      <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"/> -->
      <!-- Other module inherits -->
      <!-- Specify the app entry point class. -->
      <entry-point
class='com.google.gwt.sample.stockwatcher.client.StockWatcher'/>
      <!-- Specify the application specific style sheet. -->
      <stylesheet src='StockWatcher.css' />
</module>
```

Это GWT модуль. Модуль содержит настройки (определены в XML файле) для GWT приложения или библиотеки. Наше недавно созданное приложение содержит два тега

<inherits>, тег <entry-point> и тег stylesheet. Тег inherits сообщает GWT от каких модулей зависит ваше приложение. Это может как один так и несколько встроенных модулей или другие GWT модули которые вы разрабатываете. Необходим только один встроенный модуль чтоб создать GWT приложение: com.google.gwt.user.User, которые содержит основные GWT библиотеки. По умолчанию добавлен еще один модуль com.google.get.user.theme.standard.Standard, он содержит стили по умолчанию используемые в приложении.

Тег entry-point сообщает GWT, что данный класс является запускаемым. Данный класс должен реализовывать интерфейс Entry Point, в котором определен один метод onModuleLoad(). Метод onModuleLoad() необходим для приложения, чтобы сделать такие действия как создание пользовательского интерфейса и регистрации обработчиков событий. В нашем приложении данный класс является: com.google.gwt.sample.stockwatcher.client.StockWatcher.

HTML хост страницы

StockWatcher.html это хост страница нашего приложения. Хост страница содержит GWT приложение. Это обычный HTML файл, содержащий тег script, указывающий на запуск вашего приложения. Этот скрипт указывает на полное имя модуля и заканчивается суффиксом nocache.js:

```
<script language="javascript" src =
"com.google.gwt.sample.stockwatcher.StockWatcher.nocache.js"></script>
```

Мы будем неоднократно просматривать StockWatcher.html. По началу стоит использовать GWT приложения с использованием указанного суффикса.

CSS стили

Наши приложения также включает в себя таблицы стилей CSS, которые содержат ряд основных стилей, используемых в приложение. Вы можете добавить или удалить стиль как в традиционных веб — приложениях.

В дополнение к таблице стилей, созданные applicationCreator, вы можете использовать любой из визуальных тем, что GWT обеспечивает. Визуальные темы добавляются по умолчанию для всех виджетов GWT, что ваше приложение публикует.

Добавление GWT в существующею WEB страницу

Один из фактов, что вы должны знать, хост страница приложения не обязана быть только HTML файлом, созданной с нуля. Гибкая архитектура GWT дает возможность для интеграции GWT с существующими веб-страницами, в том числе использование технологий сервера как PHP, PERL, JAVA и т.д.

Допустим, например, что ваш существующий веб-сайт был построен с PHP. Вы хотите добавить на стороне клиента поведение в одной из ваших страниц, используя GWT. Нет проблем. Просто добавьте <script> тег, указывающий на GWT приложение, внутри вашего PHP страницы - тега <head>:

```
<html>
<head>
```

```
<title>Wrapper PHP for StockWatcher</title>
<script language='javascript'
src='com.google.gwt.sample.stockwatcher.StockWatcher.nocache.js'></script>
</head>
<body>
<?php
echo 'PHP version: ' . phpversion();
?>
</body>
</html>
```

Добавив одну строчку в эту PHP, страницу вы сделали поддержу GWT. Вы можете добавить виджеты GWT стороны клиента к страницам так же, как вы может со статическими хост страницами. GWT не должен явно поддерживать PHP для этого, чтобы работать. Фактически, так как GWT работает на стороне клиента (в отличие от стороны сервера - PHP), GWT даже не знает о факте, что хост страница была динамически произведена с PHP. К GWT это - только другая хост страница. Это означает, что GWT может легко работать с любой технологией стороны сервера, которую вы используете, пусто будет хоть PHP, Рубин, или даже ASP.NET.

Проектирование приложения

Теперь о генерированном нами скелета приложения, мы использовали утилиты projectCreator и applicationCreator, давайте спроектируем наше конечное приложение. Приложение StockWatcher будет весьма просто просто просматривать состояния виртуальных акций. Конечный вид приложения будет таким:

Давайте поговорим о возможностях которые нам нужны от программы:

Пользовательский интерфейс

Список акций

Акции будут отображаться в виде таблицы. Мы хотим отображать названия акций, текущую цену, и ее изменение за сегодня (абсолютную величину изменения и процентное величину). У нас должно быть AJAX приложение, мы не хотим заставлять пользователя кликать на кнопке обновить, что бы увидеть новые данные по акциям. Нам необходимо что бы данная таблица обновлялась сама автоматически и периодично. В каждый момент обновления, мы будем наблюдать дату последнего обновления, что бы заверить пользователя в том что цены текущие.

Первоначально приложение будет полностью на клиентской стороне, мы будем генерировать случайные данные по акциям. Мы не рекомендуем использовать эти данные что бы делать какие либо инвестиции, или другие серьезные действия. Лучше воспользуйтесь таким средством как Google Finance.

Добавление/удаление акций (наименования)

Пользователю необходима возможность добавлять акции в его список. Пользователь вводит название акции в текстовое поле и нажимает кнопку добавить. Приложение проверяет правильность введенного названия акции. Для примера мы проверяем что название акции состоит из букв или цифр в количестве от одной до пяти. Если пользователь ввел правильное название, то он увидит название акции в списке, иначе получит сообщение о неправильном вводе названия.

Так же мы добавим еще колонку, которая будет содержать кнопку удалить соответствующую акцию из списка. Когда пользователь нажимает на кнопку, то соответствующая акция немедленно удаляется из списка.

Клиентский код

Ядро GWT это компилятор, который конвертирует исходный кода JAVA в JAVASCRIPT, трансформируя ваше рабоче приложение в эквивалентное JAVASCRIPT приложение. Однако надо учитывать ограничения JAVASCRIPT исполняемого в среде браузера, которые не поддерживает GWT. Мы должные помнить о этих ограничениях при написании исходного кода программы StockWatcher.

Поддержка языка (JAVA)

GWT поддерживает семантику языка JAVA версии 1.5. Вы можете использовать любы собственные типы данных, классы (и прочее) в вашем приложении. Однако JAVASCRIPT не поддерживает 64-битные целые числа и числа с плавающей точкой, одинарной точности, так что следует избегать использование переменных long или ключевого слова strictfp в исходном тексте JAVA. JAVASCRIPT (и его расширения, GWT) не поддерживает многопоточность, так что не используйте слово synchronized, и не делайте вызовов методов Object.wait(), Object.notify() и Object.notifyAll(). Так же, GWT не поддерживает механизмы рефлексии, хотя есть возможность узнать имя класса, для указанного объекта, через метод GWT.getTypeName(Object). JAVASCRIPT также имеет встроенный сборщик мусора (GC), но JAVASCRIPT не поддерживает финализаторов, так что Java финализаторы определенные в

программе не будут выполнены в web режиме.

Поддержка библиотеки времени исполнения

JAVA 2 SE и JAVA EE библиотеки поставляют большое кол-во классов, но не вся их функциональность возможна в браузере. GWT поддерживает часть JAVA платформы, которая определена в пакетах: java.lang и java.util. GWT частично эмулирует регулярные выражения JAVA, синтаксис регулярных выражений используемых в GWT не идентичен JAVA регулярных выражений. GWT использует возможности регулярных выражений JAVASCRIPT.

Обеспечение GWT совместимости

При написании приложения используйте языковые возможности (фичи языка) и классы которые поддерживает GWT. Что бы найти несовместимость (классов/языка) в процессе разработки вашего приложения, отлаживайте его при помощи Хост — режима, если будет задействована не поддерживаемая JAVA функциональность, то вы увидите в окне отладки соответствующие сообщения.

Создание пользовательского интерфейса

На данный момент мы решили, что StockWatcher собирается делать, и мы обсуждали, как пользоваться GWT компилятором и создавать собственные проекты GWT. Мы начнем создавать приложение с пользовательским интерфейсом.

GWT пользовательский интерфейс состоит из виджетев и панелей. Виджеты предоставляют общие пользовательские инструменты как кнопки, текстовые поля и деревья. Панели как DocPanel, HorizontalPanel, и RootPanel, содержат виджеты и определяют как они должны располагаться в окне браузера.

Виджеты и панели работают одинаково на всех браузерах. Используя их, вы избавляете необходимости писать код специализированный для каждого браузера. GWT имеет полный набор виджетов доступных "из коробки", но вы не ограничены этим набором виджетов. Есть несколько способов создания пользовательских виджетов самим. Вы также можете найти хороший выбор из сторонних библиотек виджетов в сети.

Главная панель (RootPanel)

На верхнем уровне иерархии любой GWT пользовательский интерфейс, всегда содержит главную панель (RootPanel). RootPanel всегда обертывания фактический элемент HTML хост страницы. По умолчанию RootPanel обертывает элемент <body> HTML хост страницы. Вы можете получить ссылку на эту RootPanel через вызов метода RootPanel.get(). Вы также можете указать RootPanel и для других HTML элементов страницы. Просто укажите ID атрибут интересующего HTML элемента в параметре метода RootPanel.get(String).

Таким образом, у вас есть два варианта построения интерфейса вашего GWT приложения. Вы можете построить частей интерфейса вашего приложения с использованием статического HTML и просто вставить GWT виджеты и панелей в именованные элементы (ID) HTML. Эта функция особенно полезна для встраивание GWT в имеющихся приложения. Кроме того, хост-страница может содержать пустые элементы <body>, и в этом случае вы получаете ссылку умолчанию на RootPanel, а затем создать весь интерфейс с GWT используя виджеты.

Переходим к StockWatcher, используем именованные (ID) элементы HTML. Переходим к файлу хост страницы и открываем ее (src/com/google/gwt/sample/stockwatcher/public/StockWatcher.html) и изменяем содержимое как указано ниже

```
<!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>StockWatcher</title>
      <script type="text/javascript" language="javascript"
      src="com.google.gwt.sample.stockwatcher.StockWatcher.nocache.js"></script>
      </head>
<body>
      <h1>Stock Watcher</h1>
      <div id="stockList"></div>
      </body>
</html>
```

Этот код определяет статическое содержание страницы. Как и в обычном HTML, тег <title> - название нашей страницы, которая появляется на окне или вкладке. Тег <script> - ссылка на исходный код JavaScript, что GWT будет генерировать. Наконец, тег будет содержать наши GWT виджеты. Мы уставили ему (тегу div) ID атрибут, чтобы он был нам доступен из GWT через RootPanel.

Кроме того, в начале файла указан тип HTML 4.01 — в данном случае это указывает использовать «стандартный режим» рендер-движка GWT, которая обеспечивает лучшую кросс-браузерную совместимость. Если вы удалите эту строку, то рендер — движок будет работать в режиме обратной совместимости, что означает, что все старые ошибки браузера будет по-прежнему присутствуют. В некоторых случаях, вы можете использовать режим обратной совместимости, если вы интегрируете с уже готовым приложением, которое использует конкретные ошибки браузера в настоящее время, но сейчас мы будем придерживаться стандартного режима.

Панели и Виджеты

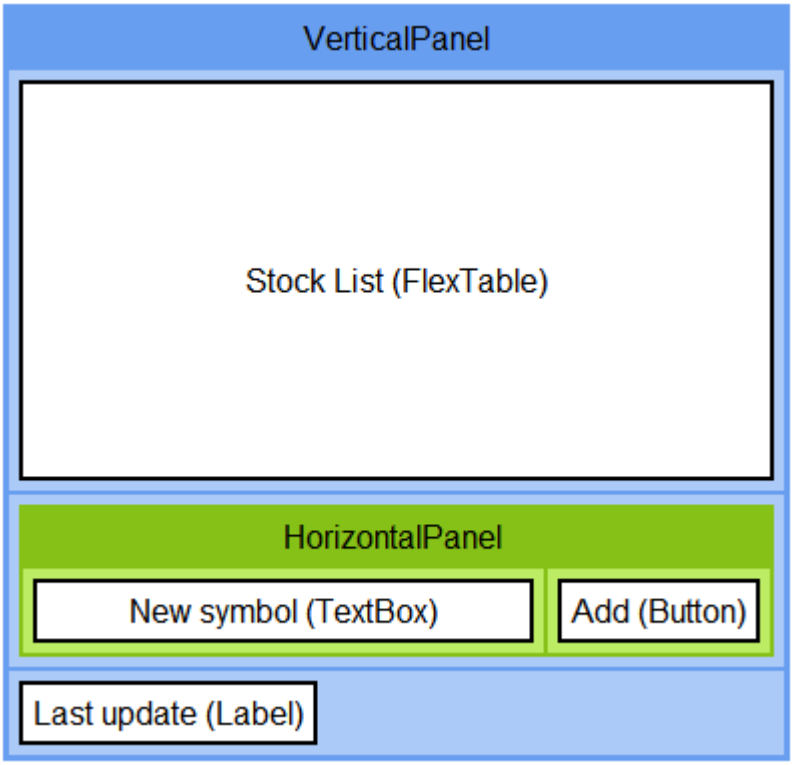
Теперь нам нужно использовать GWT панели и виджеты для построения динамичной части пользовательского интерфейса. В начале данного руководства мы показали вам, как будет окончательно выглядеть приложение StockWatcher. Список акций отображается в виде таблицы, с текстовым полем названия акции и кнопки добавить, а так же дата последнего обновления списка. Поскольку элементы пользовательского интерфейса уложены вертикально, мы смотрим галерею виджетов и обнаруживаем, что VerticalPanel это именно то, что нам нужно. Мы будем использовать GWT VerticalPanel с тремя дочерними виджетами.

Он первым (вверху) дочерним виджетом это сам список акций. Поскольку это таблица, мы опять обратимся к галерее и найдем HTMLTable, которая выглядит многообещающе. Тем не менее, он помечен как абстрактный поэтому мы должны найти подходящий подкласс. Grid (Сетка) не будет работать, поскольку она не позволяет удалить строки из середины таблицы (нам это будет нужно - функция удаления акций из списка). FlexTable, с другой стороны, есть методом removeRow(INT). Он также имеет методы для определения содержания каких-либо клеток (определенных индексов строк и столбцов), автоматически расширяет таблицу, если это необходимо. Это должно пригодиться.

Второй дочерний виджет нашей вертикальной панели (VerticalPanel), должен будет содержать текстовое поле с названием акции и кнопку Добавить. Мы хотим что бы эти два дочерних элемента (текстовое поле и кнопка) отображались на одной линии, поэтому мы

будем нуждаться в другой вложенной панели. Для этого мы будем использовать HorizontalPanel с дочерними элементами: текстовое поле (TextBox) и кнопка (Button).

Наконец, третий дочерний виджет нашей вертикальной панели (VerticalPanel) будет дата последнего обновления списка акций, мы будем показывать в простой ярлык. Виджет Label (метка) предназначен для отображения динамичного текста, не являющихся HTML текстом (обычный текст).



Теперь переходим к коду:

Откройте главный класс (EntryPoint — входная точка) приложения StockWatcher (src/com/google/gwt/sample/stockwatcher/client/StockWatcher.java) и замените содержимое следующим кодом

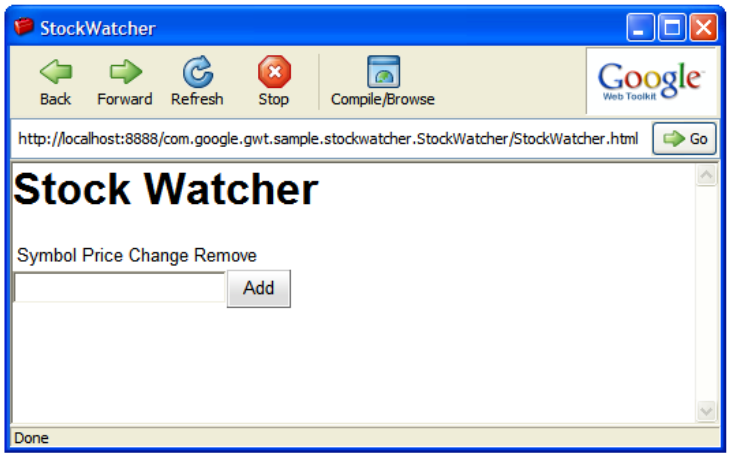
```
package com.google.gwt.sample.stockwatcher.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.FlexTable;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class StockWatcher implements EntryPoint {
   private VerticalPanel mainPanel = new VerticalPanel();
   private FlexTable stocksFlexTable = new FlexTable();
   private HorizontalPanel addPanel = new HorizontalPanel();
   private TextBox newSymbolTextBox = new TextBox();
   private Button addButton = new Button("Add");
   private Label lastUpdatedLabel = new Label();

   public void onModuleLoad() {
        // Настраиваем заголовок таблицы цен
        stocksFlexTable.setText(0, 0, "Symbol");
        stocksFlexTable.setText(0, 1, "Price");
```

```
        stocksFlexTable.setText(0, 2, "Change");
        stocksFlexTable.setText(0, 3, "Remove");
        // Панель добавления новой акции
        addPanel.add(newSymbolTextBox);
        addPanel.add(addButton);
        // Главная панель
        mainPanel.add(stocksFlexTable);
        mainPanel.add(addPanel);
        mainPanel.add(lastUpdatedLabel);
        // Добавляем главную панель к HTML элементу с ID "stockList"
        RootPanel.get("stockList").add(mainPanel);
        // Переводим фокус на текстовое поле
        newSymbolTextBox.setFocus(true);
    }
}
```

Что мы делаем здесь, это просто строим интерфейс с GWT виджетами и панелями. Мы работаем снизу-вверх, сначала создаем каждый виджет / панель через поля класса. Затем в методе onModuleLoad () (который, как вы помните, это куда мы помещаем код запуска нашего приложения), мы устанавливаем заголовок виджета FlexTable, а затем добавляем в панели дочерние элементы. Последний шаг заключается в том, чтобы добавить нашу внешнюю вертикальную панель (VerticalPanel) к главной панели (RootPanel), которая должна содержаться в элементе с именем stockList на нашей хост-странице.

Проверка интерфейса

Пришло время проверить наши изменения. Сохраните отредактированный файл и запустить StockWatcher в хост режиме (нажмите кнопку Выполнить в Eclipse, или запустить StockWatcher-shell скрипт, если вы используете другую IDE). Вы должны увидеть наши примитивные формы StockWatcher, они появляться в хост режиме браузера.



Как вы можете видеть, на текущем шаге StockWatcher наверняка не завоюет дизайнерские награды. Это нормально. Мы добавим некоторые стили позже с использованием CSS. Что в StockWatcher действительно хватает, так это интерактивность: интерфейс фактически не делает ничего ... все еще. Давайте исправить, добавив, что некоторые слушателей событий (event listeners).

Добавление слушателей событий

Как и во многих фреймворках пользовательских интерфейсов, GWT является событие-ориентированным. Это означает, что весь код выполняется в ответ на некоторые другие

события. Зачастую, это событие вызвало со стороны пользователя, который используется мышь или клавиатуру для взаимодействия с программой.

Интерфейс слушателя

Прежде чем вы сможете реагировать на события, сначала надо сообщить GWT, в каких типах событий вы заинтересованы. Это известно как подписаться на события. Чтобы подписаться на события, вы передаете реализацию интерфейса слушателя события виджету. Интерфейс слушателя определяет один или несколько методов, которые виджет можно позже вызвать случае, когда произойдет событие. С точки зрения виджета, мы говорим что виджет публикует события которые доставляются (перенаправляются) любым слушателям.

Подписка на события

Существуют различные интерфейсы слушателей в GWT. Например, интерфейс ClickListener является простым обработчиком событий кликов мыши. Он содержит только один метод, onClick (Widget). Этот метод будет вызван, когда пользователь нажимает на виджет, который в свою очередь опубликовал событие кликов мыши. Один из таких виджетов является класс Button. Для просмотра этого в действии, вернемся к нашему примеру StockWatcher. У нас уже есть кнопки для добавления акций в список. Теперь мы будем обрабатывать события кликов посредством реализации интерфейса ClickListener.

Для этого нам необходимо знать точное название метода - addClickListener (ClickListener). Давайте двигаться вперед, и доработаем код метода onModuleLoad():

```
public void onModuleLoad() {
    // set up stock list watch list
    stocksFlexTable.setText(0, 0, "Symbol");
    stocksFlexTable.setText(0, 1, "Price");
    stocksFlexTable.setText(0, 2, "Change");
    stocksFlexTable.setText(0, 3, "Remove");
    // set up event listeners for adding a new stock
    addButton.addClickListener(new ClickListener() {
         public void onClick(Widget sender) {
              addStock();
         }
    });
    // assemble Add Stock panel
    addPanel.add(newSymbolTextBox);
    addPanel.add(addButton);
    // assemble main panel
    mainPanel.add(stocksFlexTable);
    mainPanel.add(addPanel);
    mainPanel.add(lastUpdatedLabel);
    // add the main panel to the HTML element with the id "stockList"
    RootPanel.get("stockList").add(mainPanel);
    // move cursor focus to the text box
    newSymbolTextBox.setFocus(true);
}
private void addStock() {
    // executed when user clicks the addButton
}
```

Вам также необходимо добавить пару новых инструкций импорта в StockWatcher.java:

```
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.Widget;
```

Вы заметите, что мы использовали анонимный внутренний класс для реализации ClickListener. В нем мы определили метод onClick (Widget), в котором делегируем вызов метода, addStock (). В данном методе мы будем проверять написание имени акции и

добавлять ее в общий список акций.

Для небольших приложений, с относительно небольшим числом событий, можно использовать анонимные внутренние классы, это требует минимальных усилий. Однако, если мы подпишемся на много событий из большого числа издателей, то этот подход будет неэффективным, поскольку это может привести к созданию множества отдельных объектов слушателей. В этом случае, лучше иметь класс реализующий интерфейс слушателя и использовать один метод для обработки событий происходящих от разных издателей. Когда происходит событие, то у соответствующего подписчика вызывается метод, где параметр sender указывает издателя создавшего событие, поэтому вы будете знать, кто вызвал событие. Это позволяет более эффективно использовать память, но требует немного больше кода, как показано в следующем примере:

```
public class ListenerExample extends Composite implements ClickListener {
   private FlowPanel fp = new FlowPanel();
   private Button b1 = new Button("Button 1");
   private Button b2 = new Button("Button 2");
   public ListenerExample() {
        initWidget(fp);
        fp.add(b1);
        fp.add(b2);
        b1.addClickListener(this);
        b2.addClickListener(this);
   }
   public void onClick(Widget sender) {
        if (sender == b1) {
             // handle b1 being clicked
        } else if (sender == b2) {
             // handle b2 being clicked
   }
   }
}
```

Адаптеры слушателей

Сейчас, вернемся к StockWatcher. Как только мы реализуем addStock () в следующем разделе, пользователь сможет нажать на кнопку Добавить, чтобы добавить акцию в список. Для удобства, давайте также дадим ему возможность использовать клавиатуру и при нажатии ENTER внутри текстового поля newSymbolTextBox, акция будет добавляться в список. Чтобы сделать это, мы подпишемся на события клавиатуры для виджета newSymbolTextBox , для этого мы вызовем его метод addKeyboardListener() и передадим в качестве параметра ссылку на реализацию KeyboardListener.

Глядя на KeyboardListener интерфейс, мы видим, что он содержит три метода:

- onKeyDown(Widget, char, int) Вызывается когда пользователь нажимает на «физическую» клавишу.
- onKey Press(Widget, char, int) Вызывается когда пользователь печатает текст на клавиатуре.
- onKeyUp (Widget, char, int) Вызывается когда пользователь отпускает на «физическую» клавишу.

В нашем примере это нам действительно нужен всего лишь метод onKeyDown (Widget, char, int) слушателя KeyboardListener. Что нам нужно, так это класс, который позволяет нам принимать конкретные события в которых мы заинтересованы и игнорировать все остальные. Как выясняется, GWT имеет класс - адаптер для этой цели. Адаптеры просто пустые реализации того или иного интерфейса. Просто создадим подкласс класса адаптера и

реализуем прием события на которые хотим подписаться.

Для StockWatcher, мы добавим инструкцию импорта KeyboardListenerAdapter:

```
import com.google.gwt.user.client.ui.KeyboardListenerAdapter;
```

Реализуем его подкласс и передадим ссылку в вызове метода addKeyboardListener(KeyboardListener) в onModuleLoad() следующим образом:

```
// set up event listeners for adding a new stock
addButton.addClickListener(new ClickListener() {
public void onClick(Widget sender) {
addStock();
}
});
newSymbolTextBox.addKeyboardListener(new KeyboardListenerAdapter() {
@Override
public void onKeyDown(Widget sender, char keyCode, int modifiers) {
if (keyCode == KEY_ENTER) {
addStock();
}
}
});
// assemble Add Stock panel
addPanel.add(newSymbolTextBox);
addPanel.add(addButton);
```

Наш слушатель событий готов к работе. Следующим шагом будет добавить код в наш пустой метод addStock() - добавление акции к списку акций, когда происходит соответствующее событие.

Реализация функциональности клиентской стороны

Наш пример StockWatcher далек от красоты. Напомним, мы уже разработали дизайн - концепт и реализовали пользовательский интерфейс GWT с помощью виджетов и панелей, а так же подписали свои куски кода на события ввода. Теперь мы готовый написать клиентский код, который делал что нибудь с нашим приложением.

Проверка введенных данных

Когда пользователь первый раз запускает StockWatcher, ему придется добавить акции ведя их название в текстовое поле, что бы просматривать ее состояние. Прежде чем добавить название акции, мы должны убедиться в том, что такое название акции существует. Если нет, мы предупреждаем его через окно.

Прежде всего надо добавить код к addStock() для получения названия новой акции. Мы используем метод getText () класса TextBox для получения текста. GWT виджеты и панели содержат богатый набор свойств и методов, многие из которых обращаются непосредственно к методам HTML DOM элементов. После того как мы преобразовали пользовательский ввод в стандартную форму, мы будем использовать регулярные выражения для проверки правильности написания названий акций (не забывайте использование регулярных выражений в Java и JavaScript имеют один и тот-же смысл). Вот обновленный метод addStock ():

```
private void addStock() {
final String symbol = newSymbolTextBox.getText().toUpperCase().trim();
newSymbolTextBox.setFocus(true);
// symbol must be between 1 and 10 chars that are numbers, letters, or dots
if (!symbol.matches("^[0-9a-zA-Z\\.]{1,10}$"))
```

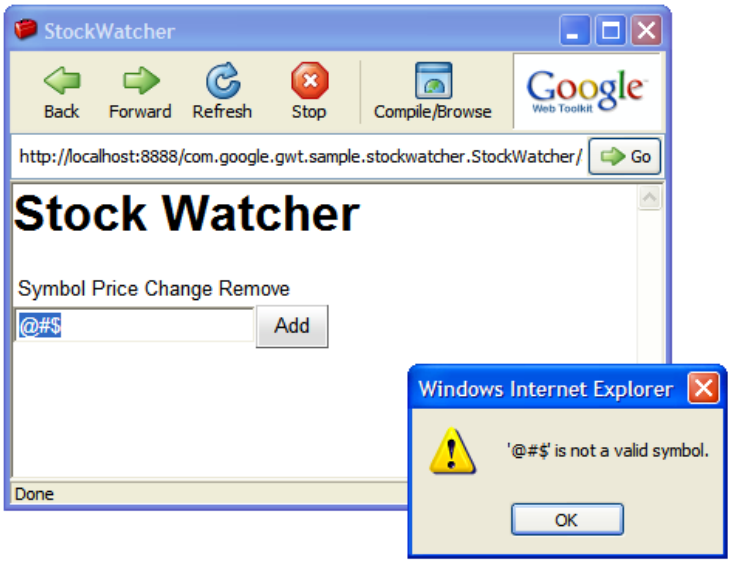
```
{
Window.alert("'" + symbol + "' is not a valid symbol.");
newSymbolTextBox.selectAll();
return;
}
newSymbolTextBox.setText("");
// now we need to add the stock to the list...
}
```

Поскольку вызывается метод Window.alert(String), так же необходимо добавить конструкцию импорта:

```
import com.google.gwt.user.client.Window;
```

На данный момент, вы можете пойти дальше и перекомпилировать и запустить StockWatcher приложения для проверки пользовательского ввода.

Если у вас уже открыт хост браузер, то вам не надо перезагружать его. Просто нажмите кнопку "Обновить" на панели инструментов, чтобы перезагрузить ваш новый GWT код.



Управление списоком акций

Хотя приложение StockWatcher получило возможность проверить название акций, оно все равно не оправдает свое назначение: пользователь еще не может добавлять акции в свой список. Давайте двигаться вперед и работать над этой функциональностью сейчас. Как вы помните, наш интерфейс содержит виджет FlexTable с именем stocksFlexTable, который будет содержать список акций. Каждая строка в списке будут содержать название акции, ее текущую цену, значение на сколько изменилось цена, и кнопка для удаления этой акции из списка. Теперь, давайте просто сосредоточиться на добавление и удаление акции в/из списка, и не будем беспокоиться по поводу установления и изменение цен акций.

Во-первых, мы нуждаемся в структуре данных для хранения названий акций которые в текущий момент наблюдает пользователь. Давайте использовать стандартный список Java - (ArrayList):

```
public class StockWatcher implements EntryPoint {
private VerticalPanel mainPanel = new VerticalPanel();
private FlexTable stocksFlexTable = new FlexTable();
private HorizontalPanel addPanel = new HorizontalPanel();
private TextBox newSymbolTextBox = new TextBox();
private Button addButton = new Button("Add");
private Label lastUpdatedLabel = new Label();
```

```
private ArrayList<String> stocks = new ArrayList<String>();
```

Не забудьте добавить инструкцию импорта:

```
import java.util.ArrayList;
```

Переходим к следующей части. Добавьте следующий код в конец метода addStock().

```
// don't add the stock if it's already in the watch list
if (stocks.contains(symbol))
return;
// add the stock to the list
int row = stocksFlexTable.getRowCount();
stocks.add(symbol);
stocksFlexTable.setText(row, 0, symbol);
// add button to remove this stock from the list
Button removeStock = new Button("x");
removeStock.addClickListener(new ClickListener(){
public void onClick(Widget sender) {
int removedIndex = stocks.indexOf(symbol);
stocks.remove(removedIndex);
stocksFlexTable.removeRow(removedIndex+1);
}
});
stocksFlexTable.setWidget(row, 3, removeStock);
```

Новое дополнение должно быть относительно простым. Во-первых, мы добавить новую строку в наш виджет FlexTable, у нас настроено, что первая колонка отображает название акции (помните, что метод setText(int, int, String) будет автоматически создавать новые клетки по мере необходимости, поэтому нам не нужно конкретно указывать размер таблицы). Далее, мы создаем кнопку для удаления акции и помещаем ее в последнюю колонку таблицы используя метод setWidget (int, int, Widget) (но не перед установкой слушателя ClickListener, который удаляет акцию из таблицы и массива акций (ArrayList)).

А теперь момент истины: запустите (или обновите) хост браузер. Та-да! Вы должны иметь возможность добавлять и удалять символы акций сами. Осталось не реализованная функциональность изменения цен.

Обновление цен акций

Заключительный и наиболее важной особенностью StockWatcher является это достаточно поразительное обновление наблюдаемых цен акций. Если бы мы писали приложение, использующие традиционные методы веб-разработок мы бы рассчитывали на полную

перезагрузку страницы каждый раз, когда хотели бы обновить цены. Это может быть достигнуто либо вручную (нажатием пользователей в своем браузере кнопки "Обновить") или автоматически (например, с использованием <META http-equiv="refresh" content="5"> тегов HTML в заголовке нашей страницы). Но в эту эпоху Web 2.0, это просто не прилично. В наше время торговцам (брокерам и т.д.) нужно автоматическое обновление цен... без всяких обновлений страницы.

К счастью для нас GWT позволяет легко обновить содержание "на лету" нашего приложения. Давайте добавим автоматическое обновление цен акций приложения StockWatcher с помощью класса таймера.

Автоматическое обновление с помощью таймера

Таймер является одно-поточным, браузерно-безопасным классом. Это позволяет нам планировать запуск кода в какой либо момент времени, либо один раз используя schedule(int), или неоднократно с scheduleRepeating(int). Мы будем использовать последний метод для автоматического обновления наших фондовых цен каждые пять секунд.

Для использования таймера, мы создаем новый экземпляр Timer в нашем методе onModuleLoad () и переопределиям метод Run() в таймере. Метод Run () будет вызываться, когда таймер сработает. В нашем случае, мы вызовем метод refreshWatchList(), который будет фактически выполнять обновление. Добавим инструкцию импорта Timer, а затем изменим код onModuleLoad()Ю добавим в его конец следующие строки:

```
import com.google.gwt.user.client.Timer;
public void onModuleLoad() {
...
// add the main panel to the HTML element with the id "stockList"
RootPanel.get("stockList").add(mainPanel);
// setup timer to refresh list automatically
Timer refreshTimer = new Timer() {
public void run() {
refreshWatchList();
}
};
refreshTimer.scheduleRepeating(REFRESH_INTERVAL);
// move cursor focus to the text box
newSymbolTextBox.setFocus(true);
}
```

В дополнение к изменениям кода onModuleLoad(), вам также необходимо определить константу, которая определяет частоту обновления. Добавим, что в верхней части класса StockWatcher.

```
private static final int REFRESH_INTERVAL = 5000; // ms
```

Прежде чем мы продолжим, нам необходимо добавить еще один вызов метода refreshWatchList(). Мы вызовем его в addStock () сразу после добавления новой акции в FlexTable (Так при появлении новой акции, сразу будут видны изменения в ценах):

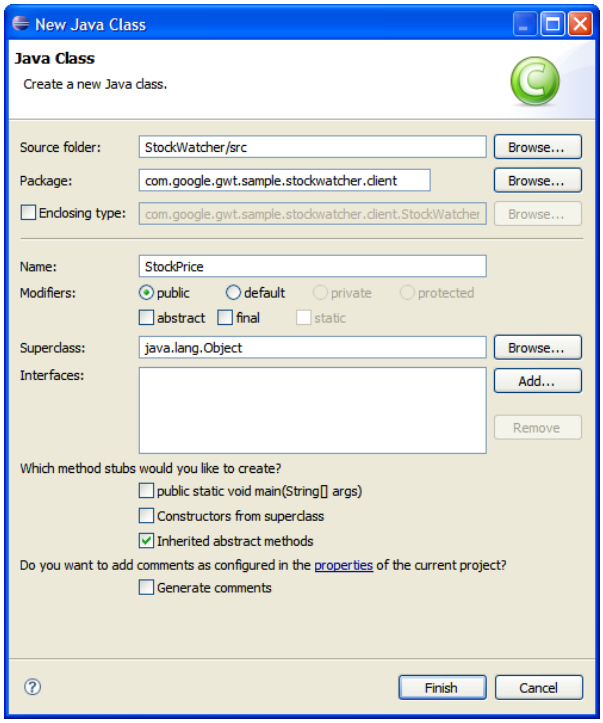
```
private void addStock() {
...
Button removeStock = new Button("x");
removeStock.addClickListener(new ClickListener(){
public void onClick(Widget sender) {
int removedIndex = stocks.indexOf(symbol);
stocks.remove(removedIndex);
stocksFlexTable.removeRow(removedIndex+1);
}
});
stocksFlexTable.setWidget(row, 3, removeStock);
// get stock price
```

```
refreshWatchList();
}
private void refreshWatchList() {
// Code will be added in the next section
}
```

StockPrice класс

Одним из основных путей развития GWT AJAX это возможность нам писать наши приложения на языке Java. Из-за этого, мы можем воспользоваться статичной проверкой компилятора и проверкой во времени, задействовать шаблоны объектно-ориентированного программирования, которые в сочетании с современными функциями IDE, такие как завершения кода и автоматизированный рефакторинг, все это дает ее проще, чем когда-либо писать надежные AJAX приложения с хорошо организованной базой кода.

Мы будем использовать этот потенциал для StockWatcher. Для данных о ценах акция и их изменении мы создадим собственный класс. Создадим новый Java класс с именем StockPrice в пакете com.google.gwt.sample.stockwatcher.client (в Eclipse, File -> New -> Class).



Вот полная реализация нашего нового класса:

```
package com.google.gwt.sample.stockwatcher.client;
public class StockPrice {
private String symbol;
private double price;
private double change;
public StockPrice() {
}
public StockPrice(String symbol, double price, double change) {
this.symbol = symbol;
this.price = price;
this.change = change;
}
public String getSymbol() {
return this.symbol;
}
public double getPrice() {
return this.price;
```

```
}
public double getChange() {
return this.change;
}
public double getChangePercent() {
return 10.0 * this.change / this.price;
}
public void setSymbol(String symbol) {
this.symbol = symbol;
}
public void setPrice(double price) {
this.price = price;
}
public void setChange(double change) {
this.change = change;
}
}
```

Генерирование цен акций

Теперь, когда мы имеем класс StockPrice для акций включая данные о ценах, давайте использовать его для обновления таблицы акций. Во-первых, мы должны создать реальные данные. Вместо фактически получения в реальном времени цен на акции от источника данных в сети, давайте просто использовать встроенный GWT генератор псевдо-случайных чисел для цен их изменений. Мы будем заполнять массив объектов StockPrice с этими значениями, а затем передавать их в другую функцию для обновления списка FlexTable. Добавим инструкцию импорта функции импорта и код в класс StockWatcher:

```
import com.google.gwt.user.client.Random;
private void refreshWatchList() {
final double MAX_PRICE = 100.0; // $100.00
final double MAX_PRICE_CHANGE = 0.02; // +/- 2%
StockPrice[] prices = new StockPrice[stocks.size()];
for (int i=0; i<stocks.size(); i++) {
double price = Random.nextDouble() * MAX_PRICE;
double change = price * MAX_PRICE_CHANGE * (Random.nextDouble() * 2.0 - 1.0);
prices[i] = new StockPrice((String)stocks.get(i), price, change);
}
updateTable(prices);
}
```

Обновление списка

Последние две новых функций будут обновлять дисплей новыми ценами. В updateTable(StockPrice) вы можете увидеть пример использования класса NumberFormat для форматирования числовых значений в строку. В нашем случае мы используем его для вывода числа в формате цены (разделитель с двумя знаками после запятой), и для отображения индикатора перемены цены. Кроме того, обратите внимание на использование класса DateTimeFormat в методе updateTable(StockPrice[]) на формат текущей даты и времени. Идем дальше и включим обе эти функции в StockWatcher:

```
private void updateTable(StockPrice[] prices) {
for (int i=0; i<prices.length; i++) {
updateTable(prices[i]);
}
// change the last update timestamp
lastUpdatedLabel.setText("Last update : " +
DateTimeFormat.getMediumDateTimeFormat().format(new Date()));
}
private void updateTable(StockPrice price) {
// make sure the stock is still in our watch list
if (!stocks.contains(price.getSymbol())) {
```

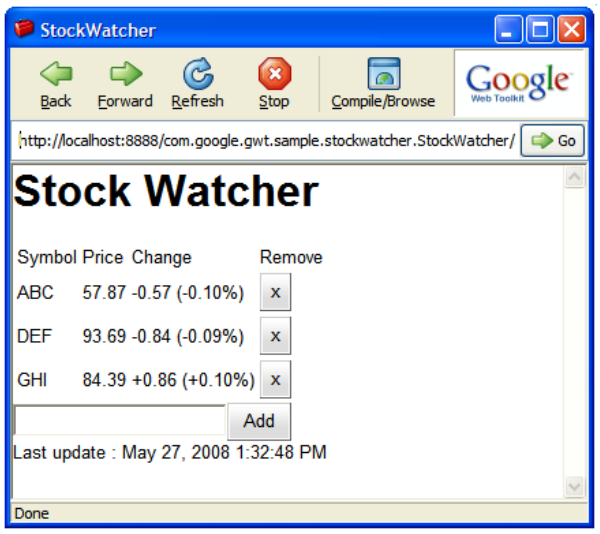
```
return;
}
int row = stocks.indexOf(price.getSymbol()) + 1;
// apply nice formatting to price and change
String priceText = NumberFormat.getFormat("#,##0.00").format(price.getPrice());
NumberFormat changeFormat = NumberFormat.getFormat("+#,##0.00;-#,##0.00");
String changeText = changeFormat.format(price.getChange());
String changePercentText = changeFormat.format(price.getChangePercent());
// update the watch list with the new values
stocksFlexTable.setText(row, 1, priceText);
stocksFlexTable.setText(row, 2, changeText + " (" + changePercentText + "%)");
}
```

DateTimeFormat и NumberFormat классы находятся в отдельном пакете следовательно вам необходимо добавить еще инструкции импорта, а также одну инструкцию для стандартного класса времени (java.util.Date) Java стандартные:

```
import com.google.gwt.i18n.client.DateTimeFormat;
import com.google.gwt.i18n.client.NumberFormat;
import java.util.Date;
```

Возможно, вы заметили, что DateTimeFormat и NumberFormat находяться в пакете из com.google.gwt.i18, которые свидетельствуют о том, что они имеют дело с интернационализацией в той или иной мере. И действительно они ... оба класса будут автоматически использовать настройки локали для форматирования чисел и дат. Мы будем говорить о локализации и переводе ваших приложений GWT позднее.

Настало время для тестирования наших изменений. Запустите или обновите приложение в режиме хоста и попробуйте добавить некоторые акции в список. Теперь мы можем видеть текущий список акций и их изменений. Если вы посмотрите за работой приложения, то вы уведите как меняются цены акций и дата обновления внизу.



Так, полюбовавшись нашим приложением StockWatcher — оно работает прекрасно. Или это не так? Как выясняется, есть одна тонкая ошибка. Вы можете заметить ее (подсказка: это заметно в скриншоте выше). В следующем разделе мы будем использовать Java для приобретения навыков для отладки.

Добавляем новые возможности

Теперь, когда вы разобрались с основами GWT, можно заняться изучением более специфичных возможностей, которые вам обязательно понадобятся как только вы начнёте создавать более сложные приложения.

- Использование удаленные вызов процедур (remote procedure calls)

  Вызов сервера с использованием GWT-RPC.
- Интернационализация

  Перевод интерфейса StockWatcher на другой язык.
- Поддержка JSON

  Получение данных формата JSON через HTTP сервер
- Тестирование с помощью Junit

  Добавление unit-тестов.

Использование удаленного вызова процедур

Все GWT приложения запускаются как код JavaScript в браузере пользователя. Тем не менее, вам достаточно часто понадобится создавать нечто большее чем просто приложение на стороне клиента. Вашему приложению понадобится связаться с сервером для отправки запросов и получения обновленных данных. Обычные web-приложения при обращении к web-серверу каждый раз загружают новую HTML страницу. С другой стороны, приложения использующие AJAX, разгружают логику пользовательского интерфейса (UI - user interface) делая асинхронные удаленные вызовы процедур, посылая и оправляете только необходимые данные. Это делает ваш пользовательский интерфейс более гибким и быстрым, уменьшая при этом требования к пропускной способности и нагрузку на сервер.

Фреймоворк GWT RPC позволяет вашему приложению легко обмениваться JAVA объектами между клиентом и сервером по HTTP. Код со стороны сервера запрошенный клиентским приложением часто называют сервисом. реализация сервисов GWT RPC основана на хорошо всем знакомой архитектуре Java сервлетов. На стороне клиента для вызова сервиса вы используете автоматически созданный прокси-класс. GWT получит сериализованные аргументы вызванного метода и вернет значение.

Важно отметить, что GWT RPC сервисы не то же самое что и web-сервисы основанные на SOAP или REST.

Они были разработаны как легковесный метод для передачи данных между сервером и GWT приложением на стороне клиента. Руководство разработчика содержит более подробную информацию о различных архитектурных настройках, которые вам понадобятся при создании RPC сервисов для вашего приложения.

StockPriceService (Сервис цен акциий)

Чтобы приобрести опыт работы с GWT RPC, мы добавим RPC сервис к StockWatcher. Назовем его "StockPriceService". Как вы возможно поняли по имени, этот сервис будет отправлять данные о ценах клиентскому приложению. Для простоты, мы используем уже имеющуюся логику для случайной генерации значений. Однако сейчас этот код будет запущен на сервере. Вы легко можете модифицировать сервис для работы с базой данных или какого-либо другого веб-сервиса.

Интерфейс сервиса

Первый этап в создании сервиса - интерфейс. В GWT RPC сервис определяется как интерфейс, наследуемый от интерфейса RemoteService.

В нашем интерфейс StockPriceService будет всего один метод, который будет принимать массив символов и возвращать массив объектов StockPrice. В пакете com.google.gwt.sample.stockwatcher.clientсоздаем новый файл с именем StockPriceService.java:

```
package com.google.gwt.sample.stockwatcher.client;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

@RemoteServiceRelativePath("stockPrices")
public interface StockPriceService extends RemoteService
{
    StockPrice[] getPrices(String[] symbols);
}
```

Единственное интересное место - аннотация @RemoteServiceRelativePath. Она ассоциирует сервис с относительным путем по умолчанию с URL модуля.

Так как StockPriceService всего лишь интерфейс, он, конечно, ничего не делает. Таким образом нам нужна реализация сервиса.

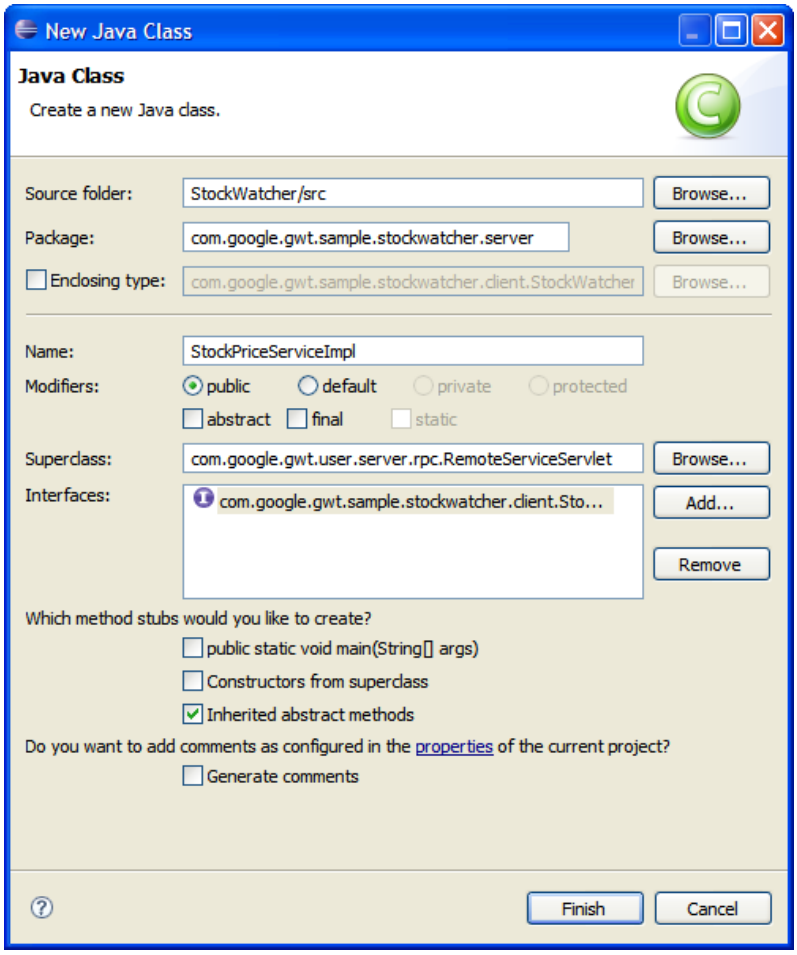
Реализация сервиса

Реализация сервиса содержит код, который будет запущен в момент вызова сервиса. В GWT есть одно важное различие между сервисом и клиентским кодом. Так как реализация сервиса находится на сервере, она не будте транслироваться в JavaScript, как клиентский код. Сервис будет запущен как байткод Java, что означает, что сервис будет иметь полный доступ к библиотекам Java Platform и любым другим компонентам который вы захотите добавить.

Реализующий класс

Чтобы реализовать RPC сервис, нам необходимо создать класс, реализующий интерфейс сервиса. Так же он должен наследоваться от RemoteServiceServlet, который содержит необходимый функционал RPC.

Давайте создадим реализацию StockPriceService. В Eclipse, откройте диалог New Java Class (File → New → Class).



Условились называть реализующий класс так же как интерфейс, но с суффиксом Impl, таким образом мы назовем новый класс StockPriceServiceImpl. Как упоминалось раньше необходимо реализовать интерфейс (StockPriceService) и унаследовать класс RemoteServiceServlet. Так же необходимо разместить класс в отдельном пакете (com.google.gwt.sample.stockwatcher.server). Это говорит GWT о том, что это серверный код и его не нужно транслировать в JavaScript. Как только вы нажмете Finish, новый класс будет добавлен:

```
import com.google.gwt.sample.stockwatcher.client.StockPrice;
import com.google.gwt.sample.stockwatcher.client.StockPriceService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

public class StockPriceServiceImpl extends RemoteServiceServlet implements
StockPriceService
{
@Override
public StockPrice[] getPrices(String[] symbols)
{
// TODO Auto-generated method stub
return null;
}
}
```

Все что осталось это реализовать в вашем единственном интерфейсе метод getPrices(String[]). Для этого мы позаимствуем код из нашего метода refreshWatchList() в StockWatcher.java.

Вот так теперь выглядит StockPriceServiceImpl.java:

```
package com.google.gwt.sample.stockwatcher.server;

import com.google.gwt.sample.stockwatcher.client.StockPrice;
import com.google.gwt.sample.stockwatcher.client.StockPriceService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

import java.util.Random;

public class StockPriceServiceImpl extends RemoteServiceServlet implements
StockPriceService {

    private static final double MAX_PRICE = 100.0; // $100.00
    private static final double MAX_PRICE_CHANGE = 0.02; // +/- 2%

    public StockPrice[] getPrices(String[] symbols) {
        Random rnd = new Random();

        StockPrice[] prices = new StockPrice[symbols.length];
        for (int i=0; i<symbols.length; i++) {
            double price = rnd.nextDouble() * MAX_PRICE;
            double change = price * MAX_PRICE_CHANGE *
                                    (rnd.nextDouble() * 2f - 1f);

            prices[i] = new StockPrice(symbols[i], price, change);
        }

        return prices;
     }
}
```

Наша реализация сервиса выглядит привычно, но есть одно неуловимое отличие. Вы заметили import of java.util.Random? Отлично! Хотите бонус? Зачем мы сделали это изменение?

Если наш вопрос поставил вас в тупик, попробуйте вспомнить, что этот код теперь выполняется на сервере. Это значит что мы используем "настоящий" класс java.util.Random из Java runtime library, а не его эмуляцию com.google.gwt.user.client.Random. Когда в будущем будете писать свои RPC сервисы, помните: реализация сервиса запускается как байткод Java, таким образом вы можете использовать любой Java класс или библиотеку не заботясь о том, как она будет транслирована в JavaScript.

Тестируем реализацию

Так как режим хоста дает возможность легко протестировать и сделать debug клиентской части кода. То возможно сделать тоже самое и для серверной части кода. Встроенный Tomcat сервер может хостить сервелеты содержащие реализации сервисов. Все что вам нужно сделать это добавить <servlet> в XML файл модуля указывающий на реализующий класс. Вам также нужно указать URL для реализующего класса.

URL-адрес должен быть в форме абсолютного пути к каталогу (например, /spellcheck или /common/login). Если вы указали умолчанию путь аннотацией @RemoteServiceRelativePath на интерфейс сервиса (как мы это делали с StockPriceService), вы захотите убедиться, что атрибута пути совпадает с значением (annotation value).

Давайте добавим тег <servlet> в файл StockWatcher.gwt.xml

```
<module>
      <!-- Inherit the core Web Toolkit stuff. -->
      <inherits name='com.google.gwt.user.User'/>
      <!-- Inherit the default GWT style sheet. You can change -->
      <!-- the theme of your GWT application by uncommenting -->
      <!-- any one of the following lines. →
      <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"/> -->
      <!-- Other module inherits -->
      <!-- Specify the app entry point class. -->
      <entry-point
      class='com.google.gwt.sample.stockwatcher.client.StockWatcher'/>
      <servlet path="/stockPrices"
      class="com.google.gwt.sample.stockwatcher.server.StockPriceServiceImpl" />
      <!-- Specify the application specific style sheet. -->
      <stylesheet src='StockWatcher.css' />
</module>
```

Как только добавлен мэппинг StockPriceService на /stockPrices, полный URL будет следующим:

Хорошо, серверная часть готова, теперь поработаем над клиентским кодом. Нам нужны так называемые асинхронные вызовы.

Асинхронный вызовы

Все RPC вызовы которые вы делаете из GWT асинхронны, это значит они не заставляют вас ждать пока придет ответ. По сравнению с обычными синхронными (блокирующими) вызовами в этом есть ряд преимуществ:

- Пользовательский интерфейс продолжает реагировать на действия пользователя. Обработчики JavaScript в браузерах как правило однопоточны, таким образом использование синхронных вызовов приведет к «зависанию» страницы до того момента, пока ответ не будет получен. Если сеть медленная или сервер не отвечает это может привести к краху фронт энда. Синхронные вызовы сервера, так же нарушают принципы AJAX (Asynchronous JavaScript And XML — Асинхронный JavaScript и XML).
- Вы можете посылать несколько запросов на сервер в одно и то же время. Однако, браузеры как правило ограничивают число исходящих соединений до двух, ограничивая таким образом количество параллельно используемых асинхронных вызовов.
- Вы можете совершать любые другие действия пока ждете ответа от сервера. Например, вы можете наращивать пользовательский интерфейс, походу того как получаете очередные данные. Это сокращает время проходящее между тем как пользователь запросил и увидел данные на странице.

Таким образом асинхронные вызовы очень полезны. Так как асинхронный вызов не блокирует программу, код следующий за вызовом будет выполнен немедленно. Когда вызов будет завершен, он запустит код с помощью callback-метода, который вы укажете при вызове.

Чтобы указать callback-метод, необходимо передать объект AsyncCallback проксирующему сервисному классу при вызове одного из его методов. Callback-объект должен содержать два метода: onFailure(Throwable) и onSuccess(T). Когда запрос к серверу будет завершен, один из этих методов будет вызван в зависимости от того был вызов успешным или нет.

Чтобы добавить параметр AsyncCallback ко всем сервисным методам мы создадим новый интерфейс с объявлениями методов. Эта асинхронная версия будет очень похожа на оригинальный интерфейс с небольшими отличиями:

- Интерфейс должен называться так же как и исходный с Async на конце.
- Он должен находиться в том же пакете что и интерфейс сервиса.

- Каждый метод должен иметь ту же сигнатуру, но не иметь возвращаемого значения и передавать объект AsyncCallback в качестве последнего параметра.

Следуя этим правилам создадим асинхронный интерфейс для StockPriceService. Добавим новый файл с именем StockPriceServiceAsync.java в проект:

```
package com.google.gwt.sample.stockwatcher.client;
import com.google.gwt.user.client.rpc.AsyncCallback;
public interface StockPriceServiceAsync {
 void getPrices(String[] symbols, AsyncCallback<StockPrice[]> callback);
}
```

Вас может заинтересовать отсутствие возвращаемого значения. Если вы задумаетесь об этот, то поймете, что у нас просто нет выбора: так как асинхронный вызов приходит сразу же, мы просто не можем вернуть результат вызова. Итак, откуда же он придет? Ответом является параметр callback, который мы добавили только что. Если вызов будет завершен успешно возвращаемое значение будет передано в наш onSuccess(T) метод. Вы скоро сами все увидите.

Сторона клиента

Все теперь готово чтобы вызвать RPC сервис со стороны клиента. Вот что вам нужно сделать (не пытайтесь сразу набрать все, просто прочитайте сначала для ознакомления).

Создайте прокси сервисного класса

Вызовите GWT.create(Class) чтобы создать экземпляр прокси сервисного класса:

```
StockPriceServiceAsync stockPriceSvc = GWT.create(StockPriceService.class);
```

Создайте callback

Создайте экземпляр AsyncCallback .

Когда RPC вызов будет завершен, если все прошло хорошо будет вызван наш onSuccess(T) и мы получим возвращаемое значение в аргументе result.

Если что-то пойдет не так, GWT вызовет метод onFailure(Throwable), передающий нам исключение:

```
AsyncCallback<StockPrice[]> callback = new AsyncCallback<StockPrice[]>() {
 public void onFailure(Throwable caught) {
  // do something with errors
 }
 public void onSuccess(StockPrice[] result) {
  // update the watch list FlexTable
 }
};
```

Сделайте вызов

Все что осталось, это собственно вызов:

```
stockPriceSvc.getPrices(symbols, callback);
```

Изменения в StockWatcher

Теперь когда вы знаете что делать, давайте добавим RPC-вызове в StockWatcher. Откройте

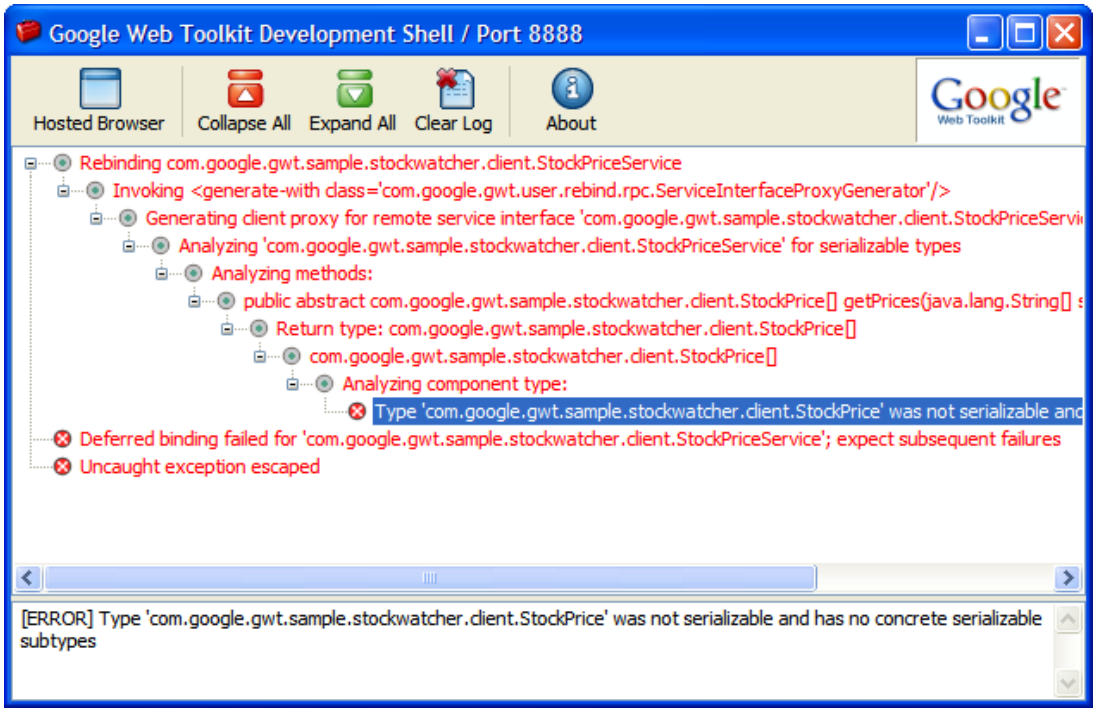
файл StockWatcher.java .

Первым делом нужно добавить новые выражения import чтобы подключить необходимые классы:

```
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
Далее создать поле для ссылки на прокси сервисаЖ
private StockPriceServiceAsync stockPriceSvc;
И, наконец, заменить старый метод refreshWatchList() переделанной версией:
private void refreshWatchList() {
 // lazy initialization of service proxy
 if (stockPriceSvc == null) {
  stockPriceSvc = GWT.create(StockPriceService.class);
 }

 AsyncCallback<StockPrice[]> callback = new AsyncCallback<StockPrice[]>() {
  public void onFailure(Throwable caught) {
   // do something with errors
  }
  public void onSuccess(StockPrice[] result) {
   updateTable(result);
  }
 };

 // make the call to the stock price service
 stockPriceSvc.getPrices(stocks.toArray(new String[0]), callback);
}
```

Добавить новый код не так сложно. Учтите, что можно кэшировать прокси сервиса для последующих вызовов сервиса. Когда вы вызываете удаленный метод getPrices(String[]), FlexTable содержащая ваш список будет обновлена или, в случае неудачи, ничего не произойдет (не беспокойтесь, позже мы добавим обработку ошибок).

Вы готовы увидеть GWT RPC в действии? Тогда вперед, запускайте новый StockWatcher и попытайтесь добавить новые акции в список. Так как мы поменяли XML файл модуля, вам не придется перезапускать режим хоста, достаточно просто обновить его. Приложение должно выглядеть также за исключением...



Ой.. . Этого не должно было произойти. Что же пошло не так? Глядя в лог среды

разработки, мы увидим в чем проблема: наш класс StockPrice не может быть сериализован.

Сериализация

Каждый раз когда вы передаете объект по сети посредством GWT RPC, он нуждается в сериализации. Сериализация это процесс упаковки содержания объекта, так чтобы он мог быть передан из одного приложения в другое или сохранен для последующего использования. GWT RPC требует чтобы все параметры сервисных методов и возвращаемые значения были сериализуемы.

Отметим также, что в GWT сериализация и сериализация основанная на Java Serializable interface это не одно и то же.

Требования к сериализации

Что делает класс сериализуемым? Для начала, все примитивные типы (int, char, boolean, etc.) и их объекты оболочки сериализуемы по умолчанию. Массивы сериализуемых типов также сериализуемы. Классы сериализуемы если они соответствуют нескольким основным требованиям:

1. Класс реализует IsSerializable или Serializable напрямую или наследуется от суперкласса реализующего эти интерфейсы.
2. Он не является final или transient, объектные поля также сериализуемы и, наконец
3. Он содержит конструктор по умолчанию (без аргументов) с любым модификатором доступа (например private Foo(){} вполне достаточно)

GWT уважает ключевое слово transient , так что значения в этих полях не сериализуемы, и не посылаются посредством RPC-вызовов.

Для любопытных, Руководство Разработчка содержит более подробную информацию о сериализации в RPC.

Исправление StockPrice

Теперь, когда мы вооруженны новыми знаниями о сериализации в GWT, давайте исправим класс StockPrice. Так как все наши поля - примитивные типы, все что нам нужно сделать это реализовать IsSerializable:

```
package com.google.gwt.sample.stockwatcher.client;
import com.google.gwt.user.client.rpc.IsSerializable;
public class StockPrice implements IsSerializable {

 private String symbol;
 private double price;
 private double change;

 ...
```

Запустим StockWatcher снова. Успех! На первый взгляд ничего не изменилось, но внутри, StockWatcher получает обновленные цены со стороны сервера, от сервлета StockPriceService вместо того, чтобы генерировать их на стороне клиента. Теперь когда основа RPC работает, давайте займемся обработкой ошибок.

Обработка ошибок

В нашей обновленной реализации WatchList(), метод объекта AsyncCallback onFailure(Throwable) пока совершенно бесполезен. Он пуст, таким образом любые исключения возникшие в процессе работы RPC будут проигнорированы. Это значит, что если у пользователя отключится сетевое соединение, то цены не будут обновляться! До тех пор пока он не обратит внимание на дату последнего обновления внизу, он может и не подозревать, что видит устаревшие данные (Хм.. рынки сегодня какие-то медленные... сегодня праздник или что?). Мы не можем допустить чтобы наш пользователь торговал по неверным ценам в StockWatcher, так что давайте оповестим его если что-то пойдет не так.

Вызов GWT RPC может не сработать по одной из двух причин: по причине не ожидаемых исключений или по причине проверяемых исключений.

Неожидаемые исключения

Существует довольно много причин которые могут препятствовать вызову RPC. Сеть может не работать, HTTP сервер может не слушать запросы, наш DNS сервер может быть неисправен, и так далее. Когда у GWT возникает проблема с вызовом сервиса, выбрасывается InvocationException в метод onFailure(Throwable).

Другой вид не ожидаемых исключений может возникать если GWT может вызвать метод, но реализация сервиса выбросила не объявленное исключение. Например, может возникнуть NullPointerException. Когда возникает не ожидаемое исключение в реализации сервиса, вы можете получить полный стек в логе режима хоста:

На стороне клиента, ваш onFailure(Throwable) метод получит InvocationException с сообщением: The call failed on the server; see server log for details.

Проверяемые исключения

Если сервисный метод может выбрасывать определенный тип исключений и мы хотим иметь возможность обработать их на стороне клиента, мы будем использовать проверяемые исключения. GWT поддерживает ключевое слово throws так что вы можете добавлять его в методы интерфейса сервиса в случае необходимости. Когда проверяемые исключения возникает в RPC сервисном методе, GWT сериализует исключение и пошлет его обратно к вызвавшему на сторону клиента для обработки.

Выбрасывание исключения в StockPriceServiceImpl

Ок, давайте попробуем все это на примере. Для иллюстрации, предположим, мы хотим выбросить исключение когда пользователь пытается вернуть данные по ценам акций, которые были исключены. Первый шаг - создаем собственные класс исключения, который

мы назовем DelistedException:

```
package com.google.gwt.sample.stockwatcher.client;
import com.google.gwt.user.client.rpc.IsSerializable;
public class DelistedException extends Exception implements IsSerializable {

 private String symbol;

 public DelistedException() {
 }

 public DelistedException(String symbol) {
  this.symbol = symbol;
 }

 public String getSymbol() {
  return this.symbol;
 }
}
```

Обратите внимание, что так как это исключение может быть отправлено посредством RPC мы должны отметить класс как сериализуемый с помощью IsSerializable.

Далее нам нужно добавить объявление throws в метод интерфейса сервиса в StockPriceService.java, вместе с соответствующим выраженим import:

```
import com.google.gwt.sample.stockwatcher.client.DelistedException;
StockPrice[] getPrices(String[] symbols) throws DelistedException;
```

Нам также нужно реализацию сервиса в StockPriceServiceImpl.java. Первое изменение это throws в методе getPrices(String[]). Другое изменение - это код который собственно выбрасывает DelistedException. Мы для простоты будем выбрасывать исключение каждый раз когда символ акции ERR добавлен в ползовательский список.

```
import com.google.gwt.sample.stockwatcher.client.DelistedException;
public StockPrice[] getPrices(String[] symbols) throws DelistedException {
 Random rnd = new Random();

 StockPrice[] prices = new StockPrice[symbols.length];
 for (int i=0; i<symbols.length; i++) {
  if (symbols[i].equals("ERR")) {
   throw new DelistedException("ERR");
  }
  double price = rnd.nextDouble() * MAX_PRICE;
  double change = price * MAX_PRICE_CHANGE * (rnd.nextDouble() * 2f - 1f);

  prices[i] = new StockPrice(symbols[i], price, change);
 }

 return prices;
}
```

Вот и все. Нет необходимости добавлять объявление throws в сервисный метод в StockPriceServiceAsync.java. Этот метод всегда возвращается мгновенно (помните, что он асинхронный), таким образом мы вместо этого получим любое выброшенное исключение когда GWT вызывает наш onFailure(Throwable).

Теперь код на стороен пользователя. Нам необходимо решить что делать когда мы ловим исключение в RPC вызове. Мы можем показать окно с собщением используя Window.alert(String), но это слишком просто. Кроме того если мы полчим много исключений пользователя может буквально закидать сообщениями об ошибках, если ошибки возникают пока он не за компьютером. Это не очень красиво. Давайте пойдем дальше и добавим новый виджет Label для вывода ошибок без лишнего беспокойства для пользователя.

Мы хотим, чтобы ошибки выделялись, так что давайте начнем с добавления нового класса в CSS в файле StockWatcher.css:

```
.errorMessage {
 color: Red;
}
```

Далее, добавим виджет для вывода ошибок на экран. В StockWatcher.java добавим следующее поле:

```
private Label errorMsgLabel = new Label();
Инициализируем его в  onModuleLoad():
...
// assemble Add Stock panel
addPanel.add(newSymbolTextBox);
addPanel.add(addButton);
addPanel.addStyleName("addPanel");
// assemble main panel
errorMsgLabel.setStyleName("errorMessage");
errorMsgLabel.setVisible(false);
mainPanel.add(errorMsgLabel);
mainPanel.add(stocksFlexTable);
mainPanel.add(addPanel);
mainPanel.add(lastUpdatedLabel);
And now, the final step: handling the error in refreshWatchList(). Fill in the
onFailure(Throwable) callback method as follows:
public void onFailure(Throwable caught) {
 // display the error message above the watch list
 String details = caught.getMessage();
 if (caught instanceof DelistedException) {
  details = "Company '" + ((DelistedException)caught).getSymbol() + "' was
delisted";
 }

 errorMsgLabel.setText("Error: " + details);
 errorMsgLabel.setVisible(true);
}
```

Чуть не забыл: последний, последний шаг - спрятать виджет с сообщением об ошибке в конце updateTable() в случае, если ошибка была исправлена (сервер вернулся в онлайн, символ ERR был удален из таблицы и т.д.), ошбика пропадает:

```
private void updateTable(StockPrice[] prices) {
 for (int i=0; i<prices.length; i++) {
  updateTable(prices[i]);
 }

 // change the last update timestamp
 lastUpdatedLabel.setText("Last update : " +
   DateTimeFormat.getMediumDateTimeFormat().format(new Date()));
 // clear any errors
 errorMsgLabel.setVisible(false);
}
```

Настало время проверить наши изменения. Запустите StockWatcher (изменения встепят в силу если мы обновим окно браузера в режиме хоста, после того как модифицировали сервис) и добавим символ ERR в список просмотра:

Если вы переключитесь на окно оболочки разработчика (Development Shell window), вы не увидите сообщений об исключениях в логе, так как исключения были проверены (и, соответственно, ожидаемы).

Развертывание сервисов на рабочем сервере

В процессе тестирования, вы можете пользоваться встроенным Tomcat сервером в режиме хоста (hosted mode) для тестирования кода на стороне сервера. Когда вы будуте деплоить ваше GWT приложение, вы можете использовать любой контейнер сервлетов для ваших сервисов. Просто убедитесь что ваш клиентский код вызывает сервис используя URL на который прицеплен сервлет в web.xml конфигурациооном файле. За более подробной информацией о деплое RPC сервлетов смотрите эту статью.

Дополнительная информация

Чтобы узнать больше читайте секцию Remote Procedure Calls в руководстве разработчика.