The CometD Tutorials

Simone Bordet

CometD 2.9.1

29 May 2014

Table of Contents

List of Examples

Chapter 1. Introduction

Welcome to the CometD tutorials, a place where you will learn to write your CometD applications step by step.

The CometD tutorials are the companion of **The CometD Reference Book**. Please refer also to **The CometD Reference Book** for further details about the CometD API usage.

1.1. Requirements

The CometD tutorials require you to install the following tools:

- The **Java™ Development Kit (JDK)**, version 1.5 or greater
- **Apache Maven**, version 3 or greater
- **Git**, the distributed version control system

Even if you do not use Maven and/or Git in your organization, using Maven and Git in the tutorials will provide you with complete projects that you can use as the basis for your projects: just copy the tutorial project built with Maven, eventually accommodating it to your project structure.

First, install the JDK into the directory of your choice. To verify that the Java installation completed correctly, issue this command:

```
$ java -version
```

You should obtain an output similar to the following:

```
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)
```

Next, install Maven into the directory of your choice. To verify that the installation of Maven completed correctly, issue this command:

```
$ mvn -version
```

You should obtain an output similar to the following:

```
Apache Maven 3.0.4 (r1232337; 2012-01-17 09:44:56+0100)
Maven home: /home/simon/programs/maven
Java version: 1.7.0_04, vendor: Oracle Corporation
Java home: /home/cometd/jdk1.7.0_04/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.2.0-24-generic", arch: "amd64", family: "unix"
```

Make sure that the "Java home" found by Maven (fourth line) points to the JDK directory (or a subdirectory).

Next, install Git into the directory of your choice. To verify that the installation of Git completed correctly, issue this command:

```
$ git --version
```

You should obtain an output similar to the following:

```
git version 1.7.9.5
```

1.2. Additional Tools

CometD projects require server-side development in Java. We recommend that you use an IDE such as **IntelliJ IDEA**, **NetBeans**, or **Eclipse**.

CometD projects require also client-side development, usually in JavaScript, running on a browser. You must be able to verify, in the browser, if your code is running as expected, and we recommend that you use tools such as **Firebug** (if you're using Firefox for development), or the equivalent for Internet Explorer (version 8 or greater), called **Developer Tools**. Google's Chromium/Chrome browser has a built-in tool that is accessible from the "Tools" menu of the browser.

Windows Users

If you are working in the Windows OS, avoid at all costs using a path that contains spaces, such as "C:\Document And Settings\", as your base path. Use a base path such as "C:\jdk1.7.0_04\" or "C:\apache-maven-3.0.4\", etc. instead.

1.3. Tutorials Pre-Made Artifacts

CometD applications are standard Java Enterprise (JEE™) web applications. A CometD tutorial produces a CometD application and, in order to see it in action, you need to package the resulting JEE web application and deploy it to a JEE servlet container.

In order to create and run a CometD tutorial web application, you need to figure out its library dependencies, package it and deploy it. Each of these tasks may be complicated and time consuming.

In order to simplify these tasks, the CometD tutorials will make use of tools and pre-made artifacts, so that you can focus on the tutorial details rather than spending time to figure out the library dependencies needed to properly package a CometD web application. In particular:

- Maven, to resolve library dependencies
- A skeleton web application, that will be used as the basis for the CometD tutorials; this skeleton web application contains a pre-configured `web.xml`, the logging configuration, a skeleton JSP page, the libraries needed to run the CometD server and the **Dojo Toolkit** libraries needed by the CometD client.

  The tutorial skeleton web application is not a CometD web application, in that it does not establish a connection between the CometD client and the CometD server. This step is explained and performed in the tutorials.
- The Jetty Maven plugin, that allows you to start Jetty and deploy your CometD web application with just one command.

The tutorials will provide detailed information on how to use the tools and how to retrieve the pre-made artifacts needed.

Chapter 2. Tutorial: Client-Side Hello World

2.1. Introduction

In this CometD tutorial you will learn how to send a "Hello, World" message from the CometD client to the CometD server. In doing this, you will primarily learn how to use the CometD JavaScript API, although you will see also how to setup a server-side service that listens to the "Hello, World" message.

2.2. Requirements

See **Section 1.1, "Requirements"**.

2.3. Step 1: Setting up the Project

Start by cloning the skeleton tutorial project into the `client-hello` directory. Open a terminal window and type the following commands:

```
$ git clone https://github.com/cometd/cometd-tutorials-skeleton.git client-hello
$ cd client-hello
```

The tutorial will modify the skeleton project, enriching it with the code and configuration needed by this tutorial.

Modify the `project/artifactId` element in the `pom.xml` file from:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <artifactId>cometd-tutorials-skeleton</artifactId>
    ...
</project>
```

to:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <artifactId>client-hello</artifactId>
    ...
</project>
```

In this way, the project will be correctly named `client-hello`.

2.4. Step 2: Initializing the CometD Client

The skeleton project provides an almost empty JSP page and a default CometD server. In order to be able to send a message to the CometD server, you need to write a JavaScript file (your CometD client application) where you setup the CometD client to connect to the CometD server. Once connected, you will send a "Hello, World" message to the server.

Open the project (now in the `client-hello` directory) in your favorite IDE, then open the `src/main/webapp/index.jsp` file (provided by the skeleton webapp). This file is the first file that will be downloaded to the browser when you run the tutorial, and it is from this file that you need to trigger the loading of the JavaScript file(s) that constitute your CometD client application.

Modify the following line from:

Example 2.1. index.jsp

```
<html>
...
<script data-dojo-config="async: true, tlmSiblingOfDojo: true"
        src="${pageContext.request.contextPath}/dojo/dojo.js.uncompressed.js"></script>
...
</html>
```

to:

Example 2.2. index.jsp

```
<html>
...
<script data-dojo-config="async: true, tlmSiblingOfDojo: true, deps: ['application.js']"
        src="${pageContext.request.contextPath}/dojo/dojo.js.uncompressed.js"></script>
...
</html>
```

Note how the `deps` field was added in the `data-dojo-config` attribute. This tells Dojo to load the dependencies listed by the `deps` field, in this case the file named `application.js`. This file is still missing, but you will be writing it next.

In the IDE, create the `src/main/webapp/application.js` file with the following content:

Example 2.3. application.js

```
require(['dojox/cometd', 'dojo/domReady!'], function(cometd)
{
    cometd.configure({
        url: location.protocol + '//' + location.host + config.contextPath + '/cometd',
        logLevel: 'info'
    });

    cometd.handshake();
});
```

Make use of Dojo's `require()` function to declare the dependencies you need; in this case you need the `cometd` object provided by the CometD Dojo binding via the `dojox/cometd` dependency, that is passed as argument to the anonymous function.

You also need the code to be executed when the DOM document of the HTML page is ready, and therefore you need to use the `dojo/domReady!` dependency.

In the anonymous function, configure the CometD object, specifying the CometD server URL (by appending the right pieces to avoid to hard-code the URL), and by specifying the log level of the CometD JavaScript library. The log level specified is `info`, but you can specify `debug` to obtain a lot of detailed logging information, although sometimes the debug information is too much - try and find the best setting for you.

After having configured the CometD object, call `handshake()` in order to start the attempt to establish the connection with the CometD server. Remember that the `handshake()` call is asynchronous and as such returns immediately without telling whether the handshake was successful or not.

In order to detect whether the handshake was successful, you need to register a listener on the `/meta/handshake` channel. This listener will receive handshake responses indicating whether they have been successful or not. Modify `application.js` accordingly:

Example 2.4. application.js

```
require(['dojox/cometd', 'dojo/dom', 'dojo/domReady!'], function(cometd, dom)
{
    cometd.configure({
        url: location.protocol + '//' + location.host + config.contextPath + '/cometd',
        logLevel: 'info'
    });

    cometd.addListener('/meta/handshake', function(message)
    {
        if (message.successful)
        {
            dom.byId('status').innerHTML += '<div>CometD handshake successful</div>';
        }
        else
        {
            dom.byId('status').innerHTML += '<div>CometD handshake failed</div>';
        }
    });

    cometd.handshake();
});
```

Note how for this step the `require(...)` statement changed and now declares one more dependencies, to `dojo/dom`, that is used to obtain the Dojo object that allows us to manipulate the DOM of the HTML page.

2.5. Step 3: Sending CometD Messages

In order to send CometD messages to the server, you need a way to bind a user event, such as a button click, to the sending of the message using the CometD API.

First, add a button to the `index.jsp` page, giving it the id of "greeter":

Example 2.5. index.jsp

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <script data-dojo-config="async: true, tlmSiblingOfDojo: true, deps: ['application.js']"
            src="${pageContext.request.contextPath}/dojo/dojo.js.uncompressed.js"></script>
    <%--
    The reason to use a JSP is that it is very easy to obtain server-side configuration
    information (such as the contextPath) and pass it to the JavaScript environment on the client.
    --%>
    <script type="text/javascript">
        var config = {
            contextPath: '${pageContext.request.contextPath}'
        };
    </script>
    <title>CometD Tutorial</title>
</head>
<body>

    <h2>CometD Tutorial</h2>

    <div id="status"></div>

    <button id="greeter">
        Send Hello to Server
    </button>

</body>
</html>
```

Then, bind the greeter button to a function in `application.js`:

Example 2.6. application.js

```
require(['dojox/cometd', 'dojo/dom', 'dojo/domReady!'], function(cometd, dom)
{
    cometd.configure({
        url: location.protocol + '//' + location.host + config.contextPath + '/cometd',
        logLevel: 'debug'
    });

    cometd.addListener('/meta/handshake', function(message)
    {
        if (message.successful)
        {
            dom.byId('status').innerHTML += '<div>CometD handshake successful</div>';
        }
        else
        {
            dom.byId('status').innerHTML += '<div>CometD handshake failed</div>';
        }
    });

    dom.byId('greeter').onclick = function()
    {
        cometd.publish('/service/hello', 'Hello, World');
    };

    cometd.handshake();
});
```

You have bound the `onclick` event of the greeter button to an anonymous function that calls `cometd.publish(...)`.

The first argument of `cometd.publish(...)` is the channel name; the second argument is the data that you want to send to the server. In this case it is a string, but it could well be a JavaScript object, for example:

```
cometd.publish('/service/hello', {
    text: 'Hello, World',
    sender: 'tutorial'
});
```

The channel name is `/service/hello`, and therefore the channel is a service channel (for further information about channels, see **The CometD Reference Book**). Remember that messages received by the server on a service channel are not broadcasted to other clients. The reason to use a service channel and not, for example, a broadcast channel such as `/hello` is that we want to say hello to the server, not to all the clients that may be listening on the `/hello` channel.

2.6. Step 4: Receiving CometD Messages

On the server side, you would like to receive greeting messages from clients and be able to perform some action when such messages are received. In CometD, you do so via services (for more information about services, see **The CometD Reference Book**), and in particular, you will use annotated services.

A annotated service is a Java class annotated with CometD annotations that is declared in `web.xml` as a parameter of CometD's `org.cometd.annotation.AnnotationCometdServlet`.

Modify `web.xml` by adding the "services" init parameter to the `AnnotationCometdServlet`, specifying the full qualified name of the service class:

Example 2.7. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <servlet>
        <servlet-name>cometd</servlet-name>
        <servlet-class>org.cometd.annotation.AnnotationCometdServlet</servlet-class>
        ...
        <init-param>
            <param-name>services</param-name>
            <param-value>org.cometd.tutorials.ClientHelloService</param-value>
        </init-param>
        ...
    </servlet>
    ...
</web-app>
```

Then create the service class as `src/main/java/org/cometd/tutorials/ClientHelloService.java`:

Example 2.8. ClientHelloService.java

```
package org.cometd.tutorials;

import org.cometd.annotation.Listener;
import org.cometd.annotation.Service;
import org.cometd.bayeux.server.ServerMessage;
import org.cometd.bayeux.server.ServerSession;

@Service
public class ClientHelloService
{
    @Listener("/service/hello")
    public void processClientHello(ServerSession session, ServerMessage message)
    {
        System.out.printf("Received greeting '%s' from remote client %s%n", message.getData(), session.getId());
    }
}
```

The `ClientHelloService` annotates method `processClientHello(...)` to be invoked when a message is received by the server on channel `/service/hello`. Method `processClientHello(...)` simply prints on the console the greeting received.

2.7. Step 5: Running the Tutorial

You can now deploy the tutorial web application to a servlet container and try it out. You will use Maven to build the tutorial web application, and the Jetty Maven plugin to deploy and run it.

Open a terminal window and type the following commands to build the tutorial web application:

```
$ cd client-hello
$ mvn install
```

The output should look like this:

```
[INFO] Scanning for projects...
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
...
```

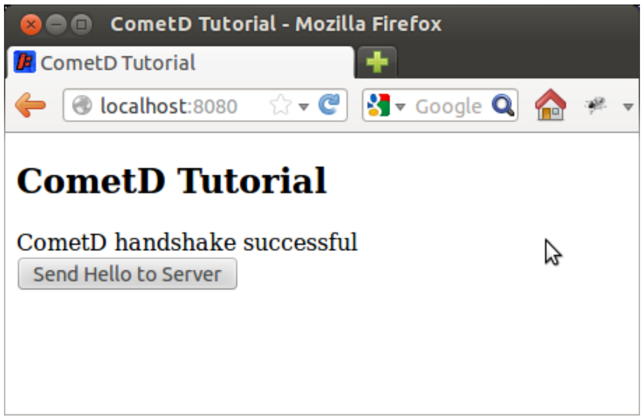
Now type this command to deploy and run the tutorial web application:

```
$ mvn jetty:run
```

The output should look like this:

```
[INFO] Scanning for projects...
...
yyyy-MM-dd HH:mm:ss.SSS:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Now the server is up and running, with the tutorial web application deployed on the root context. Start your favorite browser and open the URL http://localhost:8080, you should see the following page:



CometD Tutorial

CometD handshake successful

You can now click on the button to send a message to the server, and if you look at the terminal window you should see the log message printed by the `ClientHelloService` (the client ID may be different):

```
...
Received greeting 'Hello, World' from remote client 2qydk1xpao8t55fpukh0a8vc
...
```

2.8. Conclusions

In this tutorial, you have seen:

- How to initialize the CometD JavaScript client in a JSP page using Dojo
- How to handshake and publish a message to a channel using the CometD JavaScript API
- How to write a simple CometD service that receives the message and prints it on the terminal window

Chapter 3. Tutorial: Server-Side Stock Price Notification

3.1. Introduction

In this CometD tutorial you will learn how to send simulated stock price updates from the CometD server to CometD clients. In doing this, you will primarily learn how to use the CometD Server Java API, although you will see also how to setup the client-side that listens to the stock price update messages.

3.2. Requirements

See **Section 1.1, "Requirements"**.

3.3. Step 1: Setting up the Project

Start by cloning the skeleton tutorial project into the `server-stock-price` directory. Open a terminal window and type the following commands:

```
$ git clone https://github.com/cometd/cometd-tutorials-skeleton.git server-stock-price
$ cd server-stock-price
```

The tutorial will modify the skeleton project, enriching it with the code and configuration needed by this tutorial.

Modify the `project/artifactId` element in the `pom.xml` file from:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <artifactId>cometd-tutorials-skeleton</artifactId>
    ...
</project>
```

to:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <artifactId>server-stock-price</artifactId>
    ...
</project>
```

In this way, the project will be correctly named `server-stock-price`, then open the project (now in the `server-stock-price` directory) in your favorite IDE.

3.4. Step 2: Writing the Server-Side Services

In order to notify clients of stock price updates, you need two server-side components:

- a stock price emitter that emits stock price updates
- a CometD service that receives stock price updates from the stock price emitter and forwards them to clients

The reason to keep these two components separated is to emphasize the fact that the stock price emitter belongs to the "domain" of your application, while the CometD service is the practical implementation that you use to notify clients of stock price updates.

Create the `StockPriceEmitter` in `src/main/java/org/cometd/tutorials/StockPriceEmitter.java` as follows:

Example 3.1. StockPriceEmitter.java

```
package org.cometd.tutorials;
```

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.EventListener;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class StockPriceEmitter implements Runnable
{
    private final ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();
    private final List<String> symbols = new ArrayList<String>();
    private final Map<String, Float> values = new HashMap<String, Float>();
    private final List<Listener> listeners = new CopyOnWriteArrayList<Listener>();

    public StockPriceEmitter()
    {
        symbols.addAll(Arrays.asList("ORCL", "MSFT", "GOOG", "YHOO", "FB"));
        values.put("ORCL", 29.94f);
        values.put("MSFT", 27.10f);
        values.put("GOOG", 655.37f);
        values.put("YHOO", 17.82f);
        values.put("FB", 21.33f);
    }

    public List<Listener> getListeners()
    {
        return listeners;
    }

    public void start()
    {
        run();
    }

    public void stop()
    {
        scheduler.shutdownNow();
    }

    public void run()
    {
        Random random = new Random();

        List<Update> updates = new ArrayList<Update>();

        // Randomly choose how many stocks to update
        int howMany = random.nextInt(symbols.size()) + 1;
        for (int i = 0; i < howMany; ++i)
        {
            // Randomly choose which one to update
            int which = random.nextInt(symbols.size());
            String symbol = symbols.get(which);
            float oldValue = values.get(symbol);

            // Randomly choose how much to update
            boolean sign = random.nextBoolean();
            float howMuch = random.nextFloat();
            float newValue = oldValue + (sign ? howMuch : -howMuch);

            // Store the new value
            values.put(symbol, newValue);

            updates.add(new Update(symbol, oldValue, newValue));
        }

        // Notify the listeners
        for (Listener listener : listeners)
        {
            listener.onUpdates(updates);
        }

        // Randomly choose how long for the next update
        // We use a max delay of 1 second to simulate a high rate of updates
        long howLong = random.nextInt(1000);
        scheduler.schedule(this, howLong, TimeUnit.MILLISECONDS);
    }

    public static class Update
    {
        private final String symbol;
        private final float oldValue;
        private final float newValue;

        public Update(String symbol, float oldValue, float newValue)
        {
            this.symbol = symbol;
            this.oldValue = oldValue;
            this.newValue = newValue;
        }

        public String getSymbol()
        {
            return symbol;
        }

        public float getOldValue()
        {
            return oldValue;
        }

        public float getNewValue()
        {
            return newValue;
        }
    }

    public interface Listener extends EventListener
    {
        void onUpdates(List<Update> updates);
    }
}
```

The `StockPriceEmitter` is a simple class that allows listeners interested in stock price changes to register to be notified of updates. It uses a scheduler to simulate random delays between stock price changes, that are encapsulated into `StockPriceEmitter.Update` objects notified to `StockPriceEmitter.Listener` instances.

The stock updates are emitted at a relatively high rate, with a delay between updates chosen randomly between 0 and 1000 milliseconds.

Think of the `StockPriceEmitter` as your business class that knows nothing about CometD, and that just emits stock price updates.

Next, the CometD service. The CometD service is interested in knowing stock price updates because it needs to notify stock price changes to clients, and therefore will implement `StockPriceEmitter.Listener`.

CometD services are the common way to implement server-side activity in CometD, and annotated CometD services are very simple to use.

Create the annotated CometD service `StockPriceService` in `src/main/java/org/cometd/tutorials/StockPriceService.java` as follows:

Example 3.2. StockPriceService.java

```
package org.cometd.tutorials;

import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import javax.inject.Inject;

import org.cometd.annotation.Service;
import org.cometd.annotation.Session;
import org.cometd.bayeux.server.BayeuxServer;
import org.cometd.bayeux.server.ConfigurableServerChannel;
import org.cometd.bayeux.server.LocalSession;
import org.cometd.bayeux.server.ServerChannel;

@Service
public class StockPriceService implements StockPriceEmitter.Listener
{
    @Inject
    private BayeuxServer bayeuxServer;
    @Session
    private LocalSession sender;

    public void onUpdates(List<StockPriceEmitter.Update> updates)
    {
        for (StockPriceEmitter.Update update : updates)
        {
            // Create the channel name using the stock symbol
            String channelName = "/stock/" + update.getSymbol().toLowerCase(Locale.ENGLISH);

            // Initialize the channel, making it persistent and lazy
            bayeuxServer.createIfAbsent(channelName, new ConfigurableServerChannel.Initializer()
            {
                public void configureChannel(ConfigurableServerChannel channel)
                {
                    channel.setPersistent(true);
                    channel.setLazy(true);
                }
            });

            // Convert the Update business object to a CometD-friendly format
            Map<String, Object> data = new HashMap<String, Object>(4);
            data.put("symbol", update.getSymbol());
            data.put("oldValue", update.getOldValue());
            data.put("newValue", update.getNewValue());

            // Publish to all subscribers
            ServerChannel channel = bayeuxServer.getChannel(channelName);
            channel.publish(sender, data, null);
        }
    }
}
```

The `StockPriceService` iterates over the stock price updates, and for each update:

1. Creates the channel name dynamically based on the stock symbol
2. Initializes a `ServerChannel` with the channel name just created
3. Converts the domain specific `StockPriceEmitter.Update` objects to a format that is suitable for CometD
4. Broadcasts the update to all subscribers of that channel

Channel names have the form, for example, of `/stock/GOOG` or `/stock/MSFT`. This allows clients to subscribe and be notified for changes of a particular stock symbol (and not all of them). If the client wants to be notified of stock price updates for all symbols, it just have to subscribe to `/stock/*`.

For this particular tutorial, the number of channels is not a problem (only few), but deciding the channel names is an important design step of a CometD application. In the stock domain, it is easy to have thousands of symbols (for example, there are roughly 7800 Nasdaq stock symbols). While CometD scales well with the number of channels, and a few thousands of channels is not a problem, having too many channels has an effect on memory occupation and increases the work of the garbage collector. Most often, applications can go by using only a few channels, say in the orders of tens or hundreds. Even for applications in the stock domain, it is probably true that the most active symbols are in the order of hundreds, not thousands.

`ServerChannel`s need to be initialized before being used. In this tutorial you need to initialize them to be *persistent* to make sure that the channel exists even if it has no subscribers. This is just an optimization to avoid null checks everywhere in your code.

The `ServerChannel`s are also configured to be *lazy*. Because the stock updates are emitted at a relatively high rate, you want to use the optimization of making the `ServerChannel`s lazy to avoid that updates flow to clients too often for the user to even notice. For more information about lazy channels, see **The CometD Reference Book**.

The conversion from the domain object `StockPriceEmitter.Update` into a `Map` is needed because CometD messages must be in JSON format. A `Map` has a natural conversion to JSON, while domain objects may not have this natural conversion.

However, CometD is very flexible in allowing domain objects to be converted into JSON automatically. CometD supports pluggable JSON libraries and the most common are Jetty's own JSON library and **Jackson**. In both these libraries you can choose "invasive" solutions like having the domain class implement some JSON library specific interface (or be annotated with JSON library specific annotations), or "non-invasive" solutions like implementing serializers/converters for your domain classes. For simplicity, in this tutorial you performed a manual conversion.

3.5. Step 3: Configuring the Server-Side Services

Now that you have written the server-side services, in this step you will configure and wire them up. In particular, you need to instantiate the `StockPriceEmitter` service and the `StockPriceService`, and register the latter as a listener of the former.

Because you need the extra configuration step of registering the `StockPriceService` as a listener of `StockPriceEmitter`, you cannot just rely on `AnnotationCometdServlet` to instantiate the CometD services for us. In CometD web applications it is typical to write an *initializer* that performs the instantiation and the wiring of services. The most common form for the initializer is that of a servlet.

Create the initializer servlet as `src/main/java/org/cometd/tutorials/Initializer.java`

Example 3.3. Initializer.java

```java
package org.cometd.tutorials;

import java.io.IOException;
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.UnavailableException;

public class Initializer extends GenericServlet
{
    private StockPriceEmitter emitter;

    @Override
    public void init() throws ServletException
    {
        // Create the emitter
        emitter = new StockPriceEmitter();

        // Retrieve the CometD service instantiated by AnnotationCometdServlet
        StockPriceService service = (StockPriceService)getServletContext().getAttribute(StockPriceService.class.getName());

        // Register the service as a listener of the emitter
        emitter.getListeners().add(service);

        // Start the emitter
        emitter.start();
    }

    @Override
    public void destroy()
    {
        // Stop the emitter
        emitter.stop();
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException
    {
        throw new UnavailableException("Initializer");
    }
}
```

The `Initializer` creates the `StockPriceEmitter` (assigning it to a field to prevent its garbage collection), then retrieves the `StockPriceService` (created by the `AnnotationCometdServlet`) and registers it as a listener of `StockPriceEmitter`.

CometD services created by `AnnotationCometdServlet` are registered as servlet context attributes under their fully qualified name.

Create the web application descriptor as `src/main/webapp/WEB-INF/web.xml`:

Example 3.4. web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         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>cometd</servlet-name>
        <servlet-class>org.cometd.annotation.AnnotationCometdServlet</servlet-class>
        <init-param>
            <param-name>transports</param-name>
            <param-value>org.cometd.websocket.server.WebSocketTransport</param-value>
        </init-param>
        <init-param>
            <param-name>services</param-name>
            <param-value>org.cometd.tutorials.StockPriceService</param-value>
        </init-param>
        <init-param>
            <param-name>maxLazyTimeout</param-name>
            <param-value>2000</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>initializer</servlet-name>
        <servlet-class>org.cometd.tutorials.Initializer</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <filter>
        <filter-name>cross-origin</filter-name>
        <filter-class>org.eclipse.jetty.servlets.CrossOriginFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>cross-origin</filter-name>
        <url-pattern>/cometd/*</url-pattern>
    </filter-mapping>

</web-app>
```

Note how you specified `load-on-startup` to be `1` for the `AnnotationCometdServlet` and to be `2` for the `Initializer`. This startup order guarantees that the CometD framework and the `StockPriceService` are already initialized and configured when the `Initializer` runs. Failing to specify a startup order may result in the `Initializer` to not find the `StockPriceService` as servlet context attribute with the result that the system will not be wired up correctly.

You also specified the `maxLazyTimeout` parameter. This parameter controls the timeout of lazy channels, so that stock price updates are not broadcasted too frequently. The reason for doing this is that too frequent updates may not be noticed by users and cause unnecessary increased load on the server.

3.6. Step 4: Initializing the CometD Client

In order to be able to receive stock price updates from the CometD server, you need to write a JavaScript file (your CometD client application) where you setup the CometD client to connect to the CometD server and subscribe to the stock symbols that you are interested in.

Open the project (now in the `server-stock-price` directory) in your favorite IDE, then open the `src/main/webapp/index.jsp` file (provided by the skeleton webapp). This file is the first file that will be downloaded to the browser when you run the tutorial, and it is from this file that you need to trigger the loading of the JavaScript file(s) that constitute your CometD client application.

Modify the `index.jsp` file to add the `deps` field that specify the application's JavaScript file - `application.js`, like so:

Example 3.5. index.jsp

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <script data-dojo-config="async: true, tlmSiblingOfDojo: true, deps: ['application.js']"
            src="${pageContext.request.contextPath}/dojo/dojo.js.uncompressed.js"></script>
    <%--
    The reason to use a JSP is that it is very easy to obtain server-side configuration
    information (such as the contextPath) and pass it to the JavaScript environment on the client.
    --%>
    <script type="text/javascript">
        var config = {
            contextPath: '${pageContext.request.contextPath}'
        };
    </script>
    <title>CometD Tutorial: Stock Price Notification</title>
</head>
<body>

    <h2>CometD Tutorial<br />Stock Price Notification</h2>

    <div id="status"></div>

    <div id="stocks"></div>

</body>
</html>
```

Next, in the IDE, create the `src/main/webapp/application.js` file with the following content:

Example 3.6. application.js

```
require(['dojox/cometd', 'dojo/dom', 'dojo/dom-construct', 'dojo/domReady!'],
function(cometd, dom, doc)
{
    cometd.configure({
        url: location.protocol + '//' + location.host + config.contextPath + '/cometd',
        logLevel: 'info'
    });

    cometd.addListener('/meta/handshake', function(message)
    {
        if (message.successful)
        {
            dom.byId('status').innerHTML += '<div>CometD handshake successful</div>';
            cometd.subscribe('/stock/*', function(message)
            {
                var data = message.data;
                var symbol = data.symbol;
                var value = data.newValue;

                // Find the div for the given stock symbol
                var id = 'stock_' + symbol;
                var symbolDiv = dom.byId(id);
                if (!symbolDiv)
                {
                    symbolDiv = doc.place('<div id="' + id + '"></div>', dom.byId('stocks'));
                }
                symbolDiv.innerHTML = '<span>' + symbol + ': ' + value + '</span>';
            });
        }
        else
        {
            dom.byId('status').innerHTML += '<div>CometD handshake failed</div>';
        }
    });

    cometd.handshake();
});
```

Make use of Dojo's `require()` function to declare the dependencies you need; in this case you need the `cometd` object provided by the CometD Dojo binding via the `dojox/cometd` dependency, the `dom` object provided by Dojo to read the DOM and the `doc` object provided by Dojo to update the DOM. These objects are passed as arguments to the anonymous function.

You also need the code to be executed when the DOM document of the HTML page is ready, and therefore you need to use the `dojo/domReady!` dependency.

In the anonymous function, configure the CometD object, specifying the CometD server URL (by appending the right pieces to avoid to hard-code the URL), and by specifying the log level of the CometD JavaScript library. The log level specified is `info`, but you can specify `debug` to obtain a lot of detailed logging information, although sometimes the debug information is too much - try and find the best setting for you.

In this tutorial, you want browsers to receive stock price updates, and for this you need to subscribe to the `/stock/*` channel. The right place to perform subscriptions to channels is from a `/meta/handshake` listener, when the handshake is successful.

Therefore you subscribe to the `/stock/*` channel, and provide a subscription function to be called when a message arrives on such channels. In the subscription function, we just lookup the `div` element for the given stock symbol (or create it if it's missing), and update the price it is showing.

3.7. Step 5: Running the Tutorial

You can now deploy the tutorial web application to a servlet container and try it out. You will use Maven to build the tutorial web application, and the Jetty Maven plugin to deploy and run it.

Open a terminal window and type the following commands to build the tutorial web application:

```
$ cd server-stock-price
$ mvn install
```

The output should look like this:

```
[INFO] Scanning for projects...
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
...
```

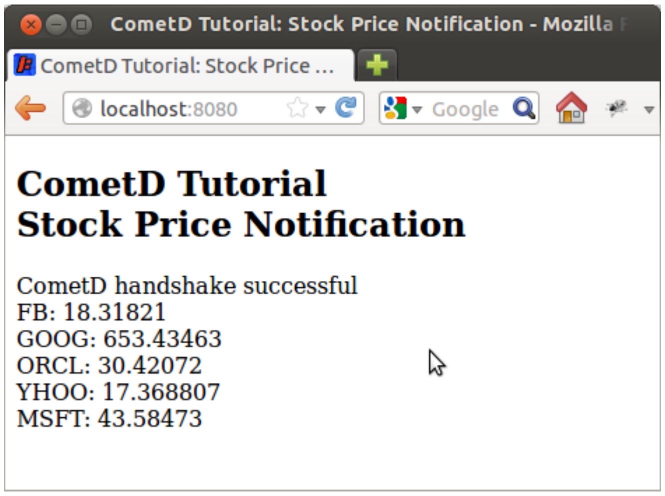
Now type this command to deploy and run the tutorial web application:

```
$ mvn jetty:run
```

The output should look like this:

```
[INFO] Scanning for projects...
...
yyyy-MM-dd HH:mm:ss.SSS:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Now the server is up and running, with the tutorial web application deployed on the root context. Start your favorite browser and open the URL http://localhost:8080, you should see the following page:


CometD Tutorial: Stock Price Notification - Mozilla F
CometD Tutorial: Stock Price ...
localhost:8080
Google

CometD Tutorial
Stock Price Notification

CometD handshake successful
FB: 18.31821
GOOG: 653.43463
ORCL: 30.42072
YHOO: 17.368807
MSFT: 43.58473

You can now watch the stock prices change at a regular intervals. The rate of updates of the stock prices is determined by the `maxLazyInterval`, by the rules of lazy channel updates (see **The CometD Reference Book**), but also by internal CometD optimizations. Do not expect the interval between updates to be exactly equal to `maxLazyInterval`.

3.8. Conclusions

In this tutorial, you have seen:

- How to simulate a server-side service that emits stock price updates
- How to write a CometD service that listen for stock price updates and broadcasts them to browsers
- How to write a simple CometD client that shows stock price updates in the browser