Writing real time applications using Spring, AngularJS and WebSockets

HTML5 made browsers a powerful alternative to desktop applications. However, network traffic (when using AJAX or by simply loading pages) is always in one direction. The client requests a page from the server, which in turn responds with the proper content. With HTML5 WebSockets that's a different story, websockets allow you to setup a full duplex channel between server and client, making it possible to send messages in both ways. In this tutorial I'm going to setup a small CRUD application that will update across all clients simultaneous.

Application architecture

The first thing I'm going to explain is how I'm going to setup the application. The application can be divided in two parts, the back-end component (using Spring) and the front-end component (using AngularJS). These two components will actually communicate in three different ways:

- **Initial**: Initially the client will make a simple HTTP request to retrieve the view (HTML page)
- **REST**: Changing the model will be persisted by using a small RESTful webservice
- **WebSockets**: WebSockets will be used to notify each client that the datasource has been changed, each client will then make a new call to the RESTful webservice to retrieve the latest model.

This isn't a "pure" WebSockets application, but uses WebSockets to notify each client. You can choose for this hybrid setup, or you can choose to persist model changes through WebSockets as well, however, this is not fully integrated into AngularJS (not on the same level RESTful webservice are), so it means the project setup will take quite longer.

Preparation

Before we start programming, we actually need to make sure we have a proper web container. WebSockets were introduced in Java EE 7 (JSR-356), which means you need to update your JVM if it's not up to date. You also need a servlet container that supports these standards, for example **Tomcat 7** or **Tomcat 8**. If you want to use Tomcat 8 and run it from Eclipse, you will have to download the latest WTP.

Maven setup

Now we can create our project. There are a few dependencies you're going to need, first you need the following Spring dependencies:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-messaging</artifactId>
    <version>4.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-websocket</artifactId>
    <version>4.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>4.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.5.0.RELEASE</version>
</dependency>
```

I'm going to create a simple webapp here that will persist to an embedded in memory HSQL database. So we're also going to need a few extra dependencies, such as hibernate and HSQL:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.3.3.Final</version>
</dependency>
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.3.2</version>
</dependency>
```

Jackson will be used to serve our model data as a JSON response in our RESTful webservice:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.3.0</version>
</dependency>
```

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.3.0</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.3.0</version>
</dependency>
```

Dozer will be used to map between our entities and our model objects:

```
<dependency>
    <groupId>net.sf.dozer</groupId>
    <artifactId>dozer</artifactId>
    <version>5.4.0</version>
</dependency>
```

To enable our websockets on each change, we will use aspects (cross-cutting concerns), include **AspectJ** as well:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.7.4</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.7.4</version>
</dependency>
```

And of course, we also need to include the necessary APIs, though we can put them on **provided** since they’re already there when deploying on a servlet container.

```
<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
```

```
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

Front-end dependencies

To manage our front-end libraries I'm going to use Bower, if you're not into that stuff you can always manually download the libraries. The Bower dependencies are:

```
{
    "name": "spring-live-updates",
    "version": "0.0.1-SNAPSHOT",
    "dependencies": {
        "angular": "latest",
        "angular-resource": "latest",
        "jquery": "latest",
        "semantic-ui": "latest",
        "sockjs": "latest",
        "stomp-websocket": "latest",
        "showdown": "latest"
    }
}
```

Make sure you don't forget to add a `.bowerrc` file as well to change the location that will be used by Bower to install its dependencies.

```
{
    "directory": "src/main/webapp/libs",
    "json": "bower.json"
}
```

We will use **AngularJS** to make our front-end application, **SockJS** and STOMP will be used to establish WebSocket connections, **Showdown** allows us to convert Markdown synta to plain HTML and **Semantic UI** is the UI library I'm going to use.

Web descriptor

The next step is that we're going to define our `web.xml`. I'm going to use **Spring AppConfig** here, so to configure your web descriptor you need the following servlet:

```
<servlet>
    <servlet-name>SpringServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <init-param>
        <param-name>contextClass</param-name>
        <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationConte
xt</param-value>
    </init-param>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            be.g00glen00b.config.AppConfig, be.g00glen00b.config.WebAppConfig,
be.g00glen00b.config.WebSocketAppConfig
        </param-value>
    </init-param>
    <async-supported>true</async-supported>
</servlet>

<servlet-mapping>
    <servlet-name>SpringServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

AppConfig

We're going to specify three different configurations. One is our global configuration (similar to `applicationContext.xml`), the next one is our web application configuration (similar to `**-servlet.xml`) and the third one is our web socket configuration.

The first thing is to annotate our `AppConfig` class with the following annotations:

```
@Configuration
@ComponentScan(basePackages = { "be.g00glen00b" }, excludeFilters = {
@ComponentScan.Filter(value = Controller.class, type = FilterType.ANNOTATION) })
@EnableJpaRepositories(basePackages = { "be.g00glen00b.repository" })
@EnableTransactionManagement
@EnableAspectJAutoProxy
public class AppConfig {
    // Config
}
```

I’m going to use **spring-data-jpa** to manage our data source, I’m also going to use Spring AOP here, so we need the `@EnableAspectJAutoProxy` annotation as well.

To create a datasource, entity manager and transaction manager, we need the following beans configured in `AppConfig`:

```
@Bean
public DataSource dataSource() {
    return new
EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.HSQL).build();
}

@Bean
public JpaVendorAdapter jpaVendorAdapter() {
    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    adapter.setShowSql(true);
    adapter.setGenerateDdl(true);
    adapter.setDatabase(Database.HSQL);
    return adapter;
}

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() throws
ClassNotFoundException {
    LocalContainerEntityManagerFactoryBean factoryBean = new
LocalContainerEntityManagerFactoryBean();
    factoryBean.setDataSource(dataSource());
    factoryBean.setPersistenceUnitName("ideas");
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());

    return factoryBean;
}

@Bean
public JpaTransactionManager transactionManager() throws ClassNotFoundException {
    JpaTransactionManager transactionManager = new JpaTransactionManager();

transactionManager.setEntityManagerFactory(entityManagerFactory().getObject());

    return transactionManager;
}
```

I’m going to use an embedded database here, but you can always replace that by your own data source.

The next step is our Dozer configuration, Dozer allows us to map beans to other types, in this case I will use it to map my entity object (`IdeaDto.class`) to a model object (`Idea.class`). The beans we need for this are:

```
@Bean
public BeanMappingBuilder beanMappingBuilder() {
    BeanMappingBuilder builder = new BeanMappingBuilder() {
        protected void configure() {
            mapping(IdeaDto.class, Idea.class);
        }
    };

    return builder;
}

@Bean
public Mapper mapper() {
    DozerBeanMapper mapper = new DozerBeanMapper();
    mapper.addMapping(beanMappingBuilder());

    return mapper;
}
```

Then the last thing we need is to load our aspect bean, this is done by:

```
@Bean
public NotifyAspect notifyAspect() {
    return new NotifyAspect();
}
```

WebAppConfig

The next one is our web application configuration, the annotations for the `WebAppConfig` class are the following:

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "be.g00glen00b.controller" })
public class WebAppConfig extends WebMvcConfigurerAdapter {
    // Our configuration
}
```

The next thing to do is to register the view resolver that will allow us to map a view name like `"ideas"` to a file called **/WEB-INF/views/ideas.jsp**. The code for this is:

```
@Bean
public InternalResourceViewResolver getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
    return resolver;
}
```

Now we also need to make sure that certain paths like /app/, **/assets/** and **/libs/** are not forwarded to the controllers as they contain our static resources. You can do that by overriding the `addResourceHandlers()` method, in this case:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/libs/**").addResourceLocations("/libs/");
    registry.addResourceHandler("/app/**").addResourceLocations("/app/");
    registry.addResourceHandler("/assets/**").addResourceLocations("/assets/");
}
```

Initially I had this as setup, but I noticed that on **Internet Explorer** there are issues with AngularJS resource because they tend to cache the results of the REST calls. To solve that I added a **No-Cache** header which can be done by adding the following configuration:

```
@Bean
public WebContentInterceptor webContentInterceptor() {
    WebContentInterceptor interceptor = new WebContentInterceptor();
    interceptor.setCacheSeconds(0);
    interceptor.setUseExpiresHeader(true);;
    interceptor.setUseCacheControlHeader(true);
    interceptor.setUseCacheControlNoStore(true);

    return interceptor;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(webContentInterceptor());
}
```

WebSocketAppConfig

The final configuration file is the websocket configuration. Just like the `WebAppConfig` we need to specify some annotations and extend a specific class to make it work, in this case it is:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketAppConfig extends AbstractWebSocketMessageBrokerConfigurer
{
    // Our configuration
}
```

Then we actually have to add an endpoint that can be used by our client to connect on and we also have to define the topic prefix that can be used to publish/subscribe. In this case our configuration is:

```
@Override
public void configureMessageBroker(MessageBrokerRegistry config) {
    config.enableSimpleBroker("/topic");
    config.setApplicationDestinationPrefixes("/app");
}

@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/notify").withSockJS();
}
```

Persistence configuration

Then the final step in configuring our web application is adding a `persistence.xml` file containing our entities, in this case it's quite simple:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="ideas" transaction-type="RESOURCE_LOCAL">
        <class>be.g00glen00b.dto.IdeaDto</class>
    </persistence-unit>
</persistence>
```

With this our first step in the tutorial is finished. In the next tutorial I will complete the back-end code for our small web application including the controllers, services, data access layer and aspects.