EventsPlugIn Developer Guide

Reference Documentation

Version 1.0

Author: Rolando Santamaría Masó

Control of versions

Fecha	Versión	Descripción	Autor
04/05/2012	1.0	Document creation	Rolando Santamaría Masó

Table of Contents

Introducing EventsPlugIn

EventsPlugIn is an extension for the jWebSocket framework to provide a server-side event-driven programming model. The extension increase the Oriented-Object(OO) Programming usage and provide support for Aspect Oriented Programming (AOP) in the client to server communication process.

EventsPlugIn also promote the integration with the Spring framework to re-use it excellents mechanisms oriented to the server side application development.

Event-driven programming in a WebSocket network

"*In computer programming, event-driven programming or event-based programming is a programming paradigm in which the flow of the program is determined by events—i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads. Event-driven programming can also be defined as an application architecture technique in which the application has a main loop which is clearly divided down to two sections: the first is event selection (or event detection), and the second is event handling...*"[1]

The WebSocket communication is by default an asynchronous process, the protocol does not specify the term “response” and the communication in fact is about in/out messages from both directions, the client and the server.
But in the traditional application work-flow the user makes “questions” that the server application must “answer”, this is request/response mechanism. For example: “Do I have new mails? What is the content of this mail? ”
The server-side applications that pretends to support the communication with the clients by using the WebSocket protocol, require to implement a simulation of the request/response mechanism to support the traditional “client to server”(C2S from now) dialogs, but also a new kind of dialogs, the “server to client”(S2C from now) ones.
The best technique to implement a simulation of request/response mechanism in both directions is by using callbacks to be executed when the client or the server receives a response for a custom request, the rest is just identify what message is

1 http://en.wikipedia.org/wiki/Event-driven_programming

the request and what message is the response, wish can be done by using unique message identifiers.
The solution scenario becomes in consequence in an event-driven scenario.

The Observable Pattern

The Observable pattern defines one-to-many dependency between objects, so when the object changes its state, all its listeners are notified automatically. Solve the need to maintain consistency between related objects without making classes tightly coupled. Generally guarantee clean implementations, high flexibility and less dependency between objects.
If there is one disadvantage, it is the memory and processor consumption in systems with a lot of listeners.

The Observable pattern implementation for EventsPlugIn has been designed re-using the concepts of the Symfony and ExtJs frameworks, about events.
Two interfaces model the subjects and listeners relationship.

The IObservable interface for subjects:

```
void on(Collection<Class<? extends Event>> aEventClassCollection, IListener
aListener) throws Exception;

void on(Class<? extends Event> aEventClass, IListener aListener) throws
Exception;

void addEvents(Class<? extends Event> aEventClass);

void addEvents(Collection<Class<? extends Event>> aEventClassCollection);

void removeEvents(Class<? extends Event> aEventClass);

void removeEvents(Collection<Class<? extends Event>> aEventClassCollection);

void un(Class<? extends Event> aEventClass, IListener aListener);

void un(Collection<Class<? extends Event>> aEventClassCollection, IListener
aListener);

ResponseEvent notify(Event aEvent) throws Exception;

ResponseEvent notify(Event aEvent, ResponseEvent aResponseEvent, boolean
aUseThreads) throws Exception;

ResponseEvent notifyUntil(Event aEvent) throws Exception;

ResponseEvent notifyUntil(Event aEvent, ResponseEvent aResponseEvent) throws
Exception;
```

```
boolean hasListeners(Class<? extends Event> aEventClass) throws Exception;

boolean hasListener(Class<? extends Event> aEventClass, IListener aListener)
throws Exception;

void purgeListeners();

void purgeEvents();

boolean hasEvent(Class<? extends Event> aEventClass);
```

The Ilistener interface for listeners:

```
void processEvent(Event aEvent, ResponseEvent aResponseEvent);
```

Subject and listener basic example:

```
//The DoorOpened event class
public class DoorOpened extends Event {}

//The observable Home class
public class Home extends ObservableObject {
  public Home(){
    addEvents(DoorOpened.class);
  }
}

//The HomeListener class
public class HomeListener implements IListener {
  @Override
  public void processEvent(Event aEvent, ResponseEvent aResponseEvent){
    //Raw interface implementation, is called only when
    //a non-specialized processEvent method is available to
    //process custom events. In this case, is not executed.
  }
  public void processEvent(DoorOpened aEvent, ResponseEvent
      aResponseEvent){
  //Do something when the door is opened
  }
}

//Creating the subject instance
Home lHome = new Home();
//Listener registration
lHome.on(DoorOpened.class, new HomeListener);
//Executing the DoorOpened event notification
```

```
ResponseEvent response = lHome.notify(new DoorOpened(), null, false);
```

The EventsPlugIn model

The EventsPlugIn is about event-driven programming, in the server and in the client, its intention is support bi-directional events notification as transparent and simple as possible.

The EventsPlugIn model start by extending from a TokenPlugIn wish receives and sends tokens. When the EventsPlugIn receive a token from the client, a new event instance of C2SEvent class is generated according to the token type, for example a token with type “auth.logon” is mapped to “org.jwebsocket.eventmodel .event.auth.Logon” event class (previously defined), then the incoming token is wrapped in a new Logon event object, wish is passed through the filter chain for the “before call” filtering process, next interested listeners on the Logon event are notified, once the listeners has been finished the filter chain is executed again for the “after call” filtering process and then the event work-flow finish.

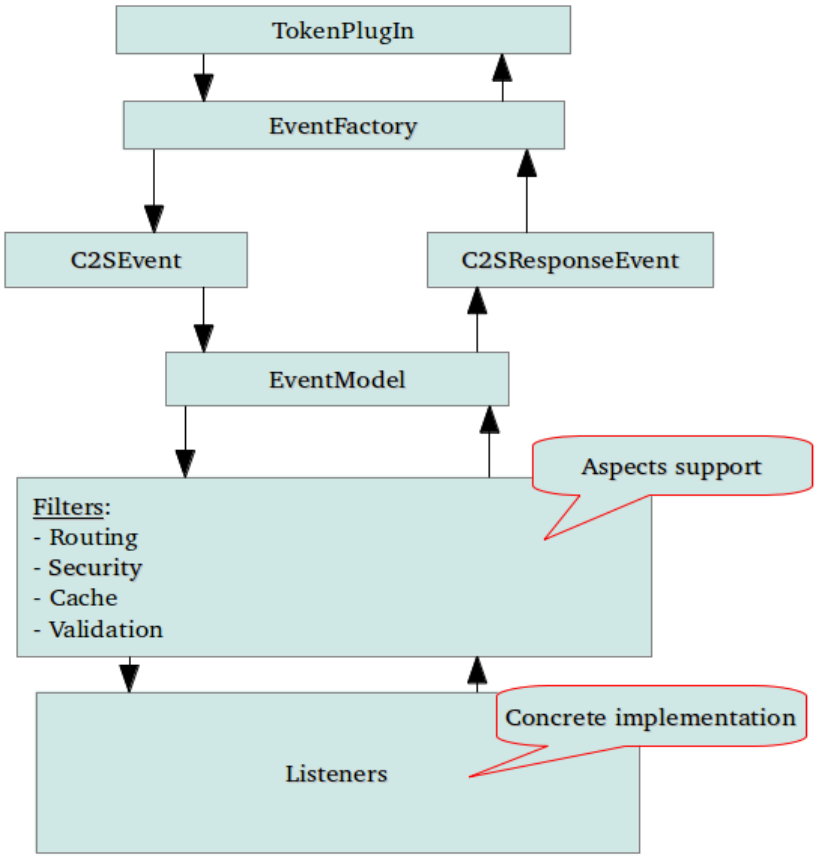

Illustration 1: C2S communication diagram

Events vs Tokens vs Packets

In the low level of the jWebSocket communication layer we have a data structure called “WebSocketPacket” to define the message received/sent from/to the client, the data is represented using a byte array. In the next level we have another structure more abstracted and developer friendly called “Token”, were the data is represented with a Map like structure. The EventsPlugIn propose new data structures wish are classes that extends from “C2SEvent, C2SResponseEvent and S2CEvent”, the objective is offer to developers a high abstracted and API friendly mechanism to deal with requests and responses.

The events are the next oriented-object definitions used for handling incoming/outgoing messages from/to the client, because tokens are good but in a really oriented-object system a token is just a Map and developers misses the

oriented-object benefits. What the EventsPlugIn propose is: “messages from and to the client are all events, so lets give a definition using a class for each one...”

Let see what we are talking using an example:

In a secure application X, suppose that we have the requirements to “logon” and “logoff” users. We creates a simple web form with the classic “username” and “password” text-boxes and the “login” button, we expects to remove the web form with the “Logoff” button after a user logon successfully in the application.

The first step here is detect what are the client events (C2SEvent) present in the requirement. In this case there are two ones: “Logon” and “Logoff” events.

Detailing a bit we have:

```
C2S event 1:
   –  identifier: logon
   –  namespace: myxapp.events.Logon
   –  arguments:
         –  0
               –  name: username
               –  type: string
         –  1
               –  name: password
               –  type: string
```

```
C2S event 2:
   –  identifier: logoff
   –  namespace: myxapp.events.Logoff
```

The next step is to represent the events into Java classes to work with events as classes instead of the Token structure.

Then we have:

```
//Java class 1:
package myxapp.events.c2s;
...
public class Logon extends C2SEvent {
  public String getUsername() {
    return getArgs().getString("username");
  }
  public String getPassword() {
    return getArgs().getString("password");
  }
}
```

```
//Java class 2:
package myxapp.events.c2s;
...
public class Logoff extends
    C2SEvent {
}
```

Now we require to describe the aspects related to both events by using the

“C2SEventDefinition” class. Example using XML and Spring:

```
//Logon event definition:
<bean parent="AbstractEventDefinition">
  <property name="id" value="logon" />
  <property name="ns" value="myxapp.events.Logon" />
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
  <property name="incomingArgsValidation">
    <set>
      <bean class="org.jwebsocket.eventmodel.filter.validator.Argument" >
        <property name="name" value="username" />
        <property name="type" value="string" />
        <property name="optional" value="false" />
      </bean>
      <bean class="org.jwebsocket.eventmodel.filter.validator.Argument" >
        <property name="name" value="password" />
        <property name="type" value="string" />
        <property name="optional" value="false" />
      </bean>
    </set>
  </property >
</bean>
```

```
//Logoff event definition:
<bean parent="AbstractEventDefinition">
  <property name="id" value="auth.logoff" />
  <property name="ns" value="myxapp.events.Logoff" />
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
  <property name="securityEnabled" value="true" />
  <property name="roles">
    <set>
      <value>USER</value>
    </set>
  </property>
</bean>
```

Already we have declared and ready to use both events, so in a event plug-in or event listener we can do:

```
...
  public void processEvent(Logon aEvent, C2SResponseEvent aResponseEvent) {

  }

  public void processEvent(Logoff aEvent, C2SResponseEvent aResponseEvent) {

  }
...
```

This is all that we require to do, for support C2S calls as events instead of tokens, giving us the real power of the oriented-object programming.

But a new requirement appear in our application and now we require to send a notification when a user “logoff” to the rest of the connected users. Is the time to define a “S2CEvent”.

A "S2CEvent" is the abstract class used to wrap the events that the server sends to the client(s) in the EventsPlugIn.

So describing the new requirement we get:

```
S2C event 1:
   -  identifier: user.logoff
   -  arguments:
         -  0
               -  name: username
               -  type: string
```

Representing the event as a Java class:

```
package myxapp.events.s2c;
...
public class UserLogoff extends S2CEvent {
  private String username;
  ...
  public UserLogoff(String username){
    super();
    setId("userLogoff");
    this.username = username;
  }

  @Override
  public void writeToToken(Token token){
    token.setString("username", username);
  }
}
```

Already we have declared and ready to use the S2C event "userLogoff"; now inside an events plug-in we can do:

```
  ...
  for (WebSocketConnector c : authenticatedUsers){
    notifyS2CEvent(new UserLogoff(username)).to(c, null);
  }
  ...
```

Work with events allow us to benefits from Aspect Object Programming (AOP) and even of more oriented-object programming in the in/out messages treatment.

Spring framework integration

"The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications. However, Spring is modular, allowing

you to use only those parts that you need, without having to bring in the rest. You can use the IoC container, with Struts on top, but you can also use only the Hibernate integration code or the JDBC abstraction layer. The Spring Framework supports declarative transaction management, remote access to your logic through RMI or web services, and various options for persisting your data. It offers a full-featured MVC framework, and enables you to integrate AOP transparently into your software.

Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself. In your integration layer (such as the data access layer), some dependencies on the data access technology and the Spring libraries will exist. However, it should be easy to isolate these dependencies from the rest of your code base."[2]

Integrated components from Spring framework in the EventsPlugIn are:

IoC Container*[3]: "Inversion of Control (IOC) is a style of software construction where reusable generic code controls the execution of problem-specific code. It carries the strong connotation that the reusable code and the problem-specific code are developed independently, which often results in a single integrated application. Inversion of Control as a design guideline serves the following purposes:*

- *There is a decoupling of the execution of a certain task from implementation.*
- *Every system can focus on what it is designed for.*
- *The systems make no assumptions about what other systems do or should do.*
- *Replacing systems will have no side effect on other systems."*

Security*: "Spring Security provides comprehensive security services for J2EE-based enterprise software applications. There is a particular emphasis on supporting projects built using The Spring Framework, which is the leading J2EE solution for enterprise software development."*

2 http://static.springsource.org/spring/docs/3.0.x/reference/spring-introduction.html
3 http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html

***Validation**: "There are pros and cons for considering validation as business logic, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Specifically validation should not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator available. Considering the above, Spring has come up with a Validator interface that is both basic and eminently usable in every layer of an application."*

The EventsPlugIn version 1.0 uses the last stable version of Spring, 3.1 version.

The core

Extending from a TokenPlugIn

The selected mechanism to integrate the EventsPlugIn with the jWebSocket framework is by using a “TokenPlugIn”. The EventsPlugIn re-use from a TokenPlugIn the following features:

- New client connected notification (connectorStarted).
- Receive tokens from the client through the process token method (processToken).
- Send tokens to connected clients (sendToken).
- Client stopped notification (connectorStopped)

Required configuration to link an EventsPlugIn application to the jWebSocket application server.
Configuration file: $JWEBSOCKET_HOME/conf/jWebSocket.xml

The EventModel

The “EventModel” (EM from now) class is an observable object that represents the EventsPlugIn core, it contains the application listeners, filters, plug-ins, the event factory, the S2C event notification handler and the global exception handler.
An EventsPlugIn application is in fact an EM object instance.

The EventModel class:

```
class EventModel extends ObservableObject implements IInitializable,
IListener {
  private String mEnv = EventModel.DEV_ENV;
  public final static String DEV_ENV = "dev";
  public final static String PROD_ENV = "prod";
  private Set<IEventModelFilter> mFilterChain;
  private Set<IEventModelPlugIn> mPlugIns;
  private EventsPlugIn mParent;
  private EventFactory mEventFactory;
  private static Logger mLog = Logging.getLogger(EventModel.class);
  private IExceptionHandler mExceptionHandler;
  private S2CEventNotificationHandler mS2CEventNotificationHandler;
  private String mNamespace;
  ...
```

In the other hand, the EM instance is responsible for the notification of all the C2S events to the server-side listeners, this means that the EM is the observable object that handle all the client events notifications.

The EM also notify by default the following internal events to the server-side interested listeners:

- ConnectorStarted: A new client gets connected.
- ConnectorStopped: A client gets disconnected.
- BeforeProcessEvent: A C2S event is being to be processed.
- AfterProcessEvent: A C2S event has been processed.
- EngineStarted: The engine has started.
- EngineStopped: The engine has stopped.
- BeforeRouteResponseToken: A Token is being to be route to the client.
- S2CResponse: A S2C event notification response has arrived.
- S2CEventNotSupportedOnClient: A S2C event is not supported in the client.

EM listener registration example from inside a EventModelPlugIn (see “Event plug-ins” section). The listener logs the C2S events that the server is going to process:

```
//Getting the EM instance of the application
EventModel lEm = getEm();
//Registering a listener for the “BeforeProcessEvent” event
lEm.on(BeforeProcessEvent.class, new IListener() {

  @Override
  public void processEvent(Event aEvent, ResponseEvent aResponseEvent) {
    //Casting the event to it super class
    BeforeProcessEvent lEvent = (BeforeProcessEvent)aEvent;
    //Genereting the log message
    String lMessage = "The event " +
        lEvent.getEvent().getClass().getSimpleName() +
       " is going to be processed...";
    //Logging the message
    if (mLogger.isDebugEnabled()){
      mLogger.debug(lMessage);
    }
  }
});
```

The EventFactory

Because a TokenPlugIn receive tokens, the EventsPlugIn require a mapping strategy to convert incoming tokens to C2S events and C2S response events to tokens, this is done by a component named EventFactory.
The link between tokens and events is possible because of the token type value wish is also the C2SEvent identifier. The token type is a unique value for each TokenPlugIn, so all the C2S event identifiers are unique across the entire EventsPlugIn application.

The EventFactory methods:

- `Token` **`eventToToken`**`(Event aEvent)`**:**
  Returns the Token representation of a given Event instance.
- `C2SEvent` **`tokenToEvent`**`(Token aToken) throws Exception`**:**
  Returns the C2SEvent representation of a given Token instance.
- `C2SEvent idToEvent(String aEventId) throws Exception`**:**
  Returns a new instance of the C2SEvent that matches the given event identifier.
- `String` **`eventToId`**`(C2SEvent aEvent)`**:**
  Returns the event identifier of the given C2SEvent instance.
- `String` **`eventClassToEventId`**`(Class<? extends Event> aEventClass) throws Exception`**:**
  Returns the identifier of the event that matches the given event class.
- `C2SResponseEvent` **`createResponseEvent`**`(C2SEvent aEvent)`**:**
  Create a response event for a given C2SEvent instance.
- `boolean` **`hasEventDefinition`**`(String aEventId)`**:**
  Returns TRUE if exists a C2SEventDefinition that matches the given identifier, FALSE otherwise.

The EventFactory uses an “IC2SEventDefinitionManager” instance to access the events definitions.

- Event definitions and aspects

An event definition, represented by the C2SEventDefinition class, is the mechanism used in the EventsPlugIn to describe attributes, aspects and restrictions information related C2S events. The events definitions are used to get information in runtime and control the events notification work-flow.
The filters use the events definitions to handle the aspects related to the events notifications, like cache, security, validation, etc...

Each C2SEvent event object requires a C2SEventDefinition instance associated. Instances of C2SEventDefinition are singletons that are consulted like a manual of how to proceed with every C2SEvent notification.

C2SEventDefinition class attributes:

#	Attribute	Description
1	`String id`	The token type. Example: “auth.logon”.
2	`String ns`	The mapping class used to create the C2S event instance, example: “org.jwebsocket.eventmodel.event.auth.Logon”.
3	`Set<Argument> incomingArgsValidation`	Describe the C2SEvent arguments.
4	`Set<Argument> outgoingArgsValidation`	Describe the C2SResponseEvent arguments.
5	`Boolean responseRequired`	Indicates if an event requires response to the client. Default value is FALSE.
6	`Boolean responseToOwnerConnector`	Indicates if the same client that notify the C2S event is waiting for the event response. Default value is TRUE.
7	`Boolean responseAsync`	Indicates if the response delivery must to be asynchronous or not. Default value is FALSE.
8	`Boolean notificationConcurrent`	Indicates if the listeners can be notified concurrently or using an iteration in the same thread. Default value is FALSE.
9	`Boolean cacheEnabled`	Indicates if the cache aspect is enabled for an event. Default value is FALSE.
10	`Integer cacheTime`	Time used to store in cache the response event. Default value is 0
11	`Boolean securityEnabled`	Indicates if the security aspect is enabled for an event. Default value is FALSE.
12	`Set<String> roles`	Roles restrictions to allow an event notification. Example: “!USER, ADMIN”.

13	Set<String> ipAddresses	IP addresses restrictions to allow an event notification. Example: “192.168.0.1/254, !66.66.66.66”.
14	Set<String> users	Users restrictions to allow an event notification. Example: “!hacker, admin”.
15	Validator validator	The main Validator instance to validate the event.
16	Integer timeout	The time limit associated to the server response. If the server takes more than this time (ms) a timeout callback should be fired on the client. Default value is 1000 (1 second).

Adding more attributes to support new aspects is possible by extending the C2SEventDefinition class and adding new filters to the filter chain (see Event filters).

C2SEventDefinition configuration example by using the Spring IOC:

```
<bean parent="AbstractEventDefinition">
  <property name="id" value="auth.logoff" />
  <property name="ns" value="org.jwebsocket.eventmodel.event.auth.Logoff" />
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
  <property name="securityEnabled" value="true" />
    <property name="roles">
      <set>
        <value>USER</value>
      </set>
    </property>
</bean>
```

Getting the C2SEventDefinition from inside an event model filter:

```
public class MyCustomFilter extends EventModelFilter {
  @Override
  public void beforeCall(WebSocketConnector aConnector, C2SEvent aEvent)
      throws  Exception {
    C2SEventDefinition lDef = getEm().getEventFactory().
        getEventDefinitions().getDefinition(aEvent.getId());
    ...
  }
  @Override
  public void afterCall(WebSocketConnector aConnector, C2SResponseEvent
      aEvent) throws  Exception{
    C2SEventDefinition lDef = getEm().getEventFactory().
        getEventDefinitions().getDefinition(aEvent.getId());
    ...
  }
}
```

Event listeners

An event listener in the EventsPlugIn is an object with a class that implements the IListener interface. A listener must be registered to a target observable object in order to be notified when interested events are fired.

The listeners in the EventsPlugIn can also distinguish between event classes:

```
//The HomeListener class
public class HomeListener implements IListener {
  @Override
  public void processEvent(Event aEvent, ResponseEvent aResponseEvent){
    //Raw interface implementation, is called only when
    //a non-specialized processEvent method is available to
    //process custom events. In this case, is not executed.
  }
  public void processEvent(DoorOpened aEvent, ResponseEvent
      aResponseEvent){
    //Do something when the door is opened
  }
  public void processEvent(DoorClosed aEvent, ResponseEvent
      aResponseEvent){
    //Do something when the door is closed
  }
}
```

EventModel plug-ins

The EM plug-ins are extensions to support C2S/S2C event notifications in the server side, basically are listeners listening or notifying related events. For example the authentication plug-in listen “Logon” and “Logoff” events and notify the “UserLogoff” event.

The EM plug-ins abstract definition:

```
public abstract class EventModelPlugIn extends ObservableObject implements
IEventModelPlugIn { … }
public interface IEventModelPlugIn extends IListener, IInitializable { … }
```

The plug-ins has the capacity to define its client API, this is used by the SystemPlugIn to export to clients the server side plug-ins API. API exporting for

plug-ins is like WSDL in web services, in fact the JavaScript client use it to create a client side representation of the server side EM plug-ins.

Plug-in API exported information example (authentication plug-in: AuthPlugIn):


```
"logon":{
	"users":[],
	"outgoingArgsValidation":[{
		"optional":false,
		"name":"username",
		"type":"string"
	},{
		"optional":false,
		"name":"roles",
		"type":"array"
	}],
	"isSecurityEnabled":false,
	"roles":[],
	"cacheTime":0,
	"ip_addresses":[],
	"incomingArgsValidation":[{
		"optional":false,
		"name":"username",
		"type":"string"
	},{
		"optional":false,
		"name":"password",
		"type":"string"
	}],
	"type":"auth.logon",
	"timeout":1000,
	"isCacheEnabled":false
},
"logoff":{
	"users":[],
	"outgoingArgsValidation":[],
	"isSecurityEnabled":true,
	"roles":["USER"],
	"cacheTime":0,
	"ip_addresses":[],
	"incomingArgsValidation":[],
	"type":"auth.logoff",
	"timeout":1000,
	"isCacheEnabled":false
}
```


- Creating a new EM plug-in.

1. Define the events that the plug-in will support. (see “Events vs Tokens vs Packets” section)
2. Create the plug-in class.

```
public class AuthPlugIn extends EventModelPlugIn {
  private AuthenticationManager am;
  …
  public void processEvent(Logon aEvent, C2SResponseEvent
aResponseEvent) { … }
  public void processEvent(Logoff aEvent, C2SResponseEvent
aResponseEvent) { … }
  private void notifyUserLogoff(String username){ … }
}
```

3. Configure the plug-in by using the Spring IOC.

```
<bean id="AuthPlugIn"
        class="org.jwebsocket.eventmodel.plugin.auth.AuthPlugIn"
        parent="AbstractPlugIn">
  <property name="id" value="auth" />
  <property name="authenticationManager"
      ref="org.springframework.security.authenticationManager" />
  <property name="emEventClassesAndClientAPI">
    <map>
      <entry key="logon"
      value="org.jwebsocket.eventmodel.event.auth.Logon"/>
      <entry key="logoff"
      value="org.jwebsocket.eventmodel.event.auth.Logoff"/>
    </map>
  </property>
  <property name="eventsDefinitions">
    <set>
      <bean parent="AbstractEventDefinition">
        <property name="id" value="auth.logon" />
        <property name="ns"
         value="org.jwebsocket.eventmodel.event.auth.Logon" />
        <property name="responseRequired" value="true" />
        <property name="responseToOwnerConnector" value="true" />
        <property name="incomingArgsValidation">
          <set>
            <bean
         class="org.jwebsocket.eventmodel.filter.validator.Argument">
              <property name="name" value="username" />
              <property name="type" value="string" />
              <property name="optional" value="false" />
            </bean>
            <bean
            class="org.jwebsocket.eventmodel.filter.validator.Argument">
              <property name="name" value="password" />
              <property name="type" value="string" />
              <property name="optional" value="false" />
            </bean>
          </set>
        </property>
        <property name="outgoingArgsValidation">
          <set>
            <bean
            class="org.jwebsocket.eventmodel.filter.validator.Argument">
              <property name="name" value="username" />
              <property name="type" value="string" />
              <property name="optional" value="false" />
            </bean>
            <bean
            class="org.jwebsocket.eventmodel.filter.validator.Argument">
              <property name="name" value="uuid" />
              <property name="type" value="string" />
              <property name="optional" value="false" />
            </bean>
```

```
            <bean
            class="org.jwebsocket.eventmodel.filter.validator.Argument">
              <property name="name" value="roles" />
              <property name="type" value="array" />
              <property name="optional" value="false" />
            </bean>
          </set>
        </property >
      </bean>
      <bean parent="AbstractEventDefinition">
        <property name="id" value="auth.logoff" />
        <property name="ns"
            value="org.jwebsocket.eventmodel.event.auth.Logoff" />
        <property name="responseRequired" value="true" />
        <property name="responseToOwnerConnector" value="true" />
        <property name="securityEnabled" value="true" />
        <property name="roles">
          <set>
            <value>USER</value>
          </set>
        </property>
      </bean>
    </set>
  </property>
</bean>
```

4. Register the new plug-in to be loaded in the system start (bootstrap.xml).

```
<bean id="EventModel" class="org.jwebsocket.eventmodel.core.EventModel"
    init-method="initialize" destroy-method="shutdown"
    scope="singleton">
  <constructor-arg index="0" value="alarm"/>
  <constructor-arg index="1" ref="EventFactory"/>
  <constructor-arg index="2" ref="S2CEventNotificationHandler"/>
  <constructor-arg index="3" ref="ExceptionHandler"/>
  <property name="env" value="dev" />
  <property name="maxExecutionTime" value="3" />
  <property name="plugIns">
    <set>
      <ref bean="SystemPlugIn" />
      <ref bean="AuthPlugIn" />
      ...
    </set>
  ...
```

EventModel filters

A filter is a piece of code executed before/after every C2S event notification. In the EventsPlugIn many filters are used to handle the aspects related to the C2S events notifications.

The EventModelFilter interface:

```
void beforeCall(WebSocketConnector aConnector, C2SEvent aEvent) throws
Exception {};

void afterCall(WebSocketConnector aConnector, C2SResponseEvent aEvent) throws
Exception {};
```

Current aspects supported by filters are:

1. **Routing**: In the first call this filter checks if the incoming event has listeners in the server-side, if not, the event work-flow finish in that point. In the second call this filter sends the response to targeted connectors.
2. **Security**: Checks if the client is authenticated and has the required permissions to notify an event. If not, the event notification is not authorized. The security checks include IP addresses, user roles and usernames and the checks are executed in the same order.
3. **Cache**: In the first call this filter checks is a non-expired response event is cached for the incoming event, if exists, the cached response is sent to the client. In the second call the response event is stored in cache if required to.
4. **Validation**: This filter checks if every defined incoming/outgoing arguments math the validation rules associated to them. The feature to check incoming and outgoing arguments without wire code in the listeners keeps clean the implementation. The validation process ensures that the server listeners only receive events with valid information and the client-side always receives the response that is waiting for, however the response validation is active only for development scenarios.

The S2C event notifications

The EventsPlugIn allows S2C events notifications through a component named S2CEventNotificationHandler. This is:

1. The server application notify an event to the client.
2. The client process the event (if possible).
3. The client sends a response event to the server application (if required).
4. The server application process the client response event through server callbacks.

The S2CEventNotificationHandler methods:

- `void send(S2CEvent aEvent, String aConnectorId, OnResponse`

`aOnResponse) throws InvalidConnectorIdentifier`

Sends a S2C event notification to a given client by it identifier.

- `void send(S2CEvent aEvent, WebSocketConnector aTo, OnResponse aOnResponse) throws InvalidConnectorIdentifier`

  Sends a S2C event notification to a given client by it connector instance.

The OnResponse callback class:

Developers can use a S2C event notifications callbacks to process the client responses. The S2C event notifications callbacks that can be attached to a S2C event notification are:

- `void success(Object aResponse, String aFrom);`

  The S2C event notification has been processed successfully. The callback receives the client response and the client connector identifier parameters.

- `void failure(FailureReason aReason, String aFrom);`

  The S2C event notification has been processed with errors. The callback receive the failure reason and the client connector identifier parameters. The supported failure reasons are:

  - CONNECTOR_STOPPED: The connector has stopped in the middle the event notification.
  - EVENT_NOT_SUPPORTED_BY_CLIENT: The S2C event notification is not supported in the client-side.
  - INVALID_RESPONSE: The response from the client does not pass the validation process.
  - TIMEOUT: The client exceeds the allowed time interval to process the S2C event.

The validation rules for S2C event responses:

A S2C event response is valid if the response type match the expected type and if the response pass the custom validation:

```
class S2CPlusXYEvent extends S2CEvent
  ...
  public S2CPlusXYEvent(int aX, int aY) {
    super();
    setId("plusXY");
```

```
    //The response value require to be type of “integer”
    setResponseType("integer");
    setTimeout(5000);

    ...
  }
  ...
```

```
new OnResponse() {

  //Supporting custom validations
  @Override
  public boolean isValid(Object aResponse, String aFrom) {
    return aResponse.equals(10);
  }

  @Override
  public void success(Object aResponse, String aFrom) {
    //Do something with the response value
  }

  @Override
  public void failure(FailureReason aReason, String aFrom) {
    //Do something with the failure
  }
}
```

The S2C event notification timeout argument:

The S2C event notification failure callback is called with the TIMEOUT failure reason if the client take more time than allowed to process the S2C event notification.

Setting the timeout argument value:

```
class S2CPlusXYEvent extends S2CEvent
  ...
  public S2CPlusXYEvent(int aX, int aY) {
    super();
    setId("plusXY");
    setResponseType("integer");
    //Timeout value in milliseconds
    setTimeout(5000);

    ...
  }
  ...
```

The default timeout value is 1000 milliseconds, 0 (cero) as value means without timeout limits.

Sending S2C events inside EM plug-ins methods:

The classes that extends the EventModelPlugIn class, can use a short-cut method to send S2C events.

```
notifyS2CEvent(new S2CPlusXYEvent(5, 5)).to(aEvent.getConnector(),
    new OnResponse() {
      @Override
      public boolean isValid(Object aResponse, String aFrom) {
        return aResponse.equals(10);
       }

      @Override
      public void success(Object aResponse, String aFrom) {...}

      @Override
      public void failure(FailureReason aReason, String aFrom) {...}
});
```

Processing the S2C events in the client (JavaScript):

```
//Generating the client-side plug-in representation
//(see “Events for JavaScript clients” section)
mathPlugIn = generator.generate("math", notifier, function(p){

  //Supporting the S2C plusXY event
  //p is a reference to mathPlugIn object
  p.plusXY = function(e){
    return e.x + e.y;
  }
}
```

The EventsPlugIn application concept

In the EventsPlugIn an application is just an EM instance, it suppose a collection of listeners, plug-ins and filters isolated under the same namespace, with the function of work together in combination to solve a custom application requirements.

By default the EventsPlugIn uses the Spring IoC Container to build the EM instance, the configuration of the applications are located in the “$JWEBSOCKET_HOME/conf/EventsPlugIn/yourappname-application” directory.

Events for JavaScript clients

The JavaScript client for the EventsPlugIn is an extension for the jWebSocket JavaScript client to support the events work-flow. The source code is available for download in the jWebSocket 1.0 web client package:

http://jwebsocket.org/downloads/downloads.htm

The EventsPlugIn JavaScript library is located at: web/res/jwsEventsPlugIn.js

The components that compose the JavaScript extension are:

- EventsNotifier: This component is the backbone, it wraps the jWebSocket client to support events notifications instead of tokens, contains the filters and plug-ins collections.

```
class EventsNotifier{
  public String ID;
  public jWebSocketTokenClient jwsClient;
  public String NS;
  public EventsBaseFilter[] filterChain;
  public EventsPlugIn[] plugIns;
  public void initialize();
  public void notify(String aEvenName, JSONObject aOptions);
  public processToken(Token aToken);
}
```

- EventsPlugInGenerator: The component use the server side API exporting feature to generate dynamically JavaScript plug-ins, wish are the JavaScript representations of the server-side EM plug-ins, allowing direct events notifications via methods calls.

```
class EventsPlugInGenerator{
  public void generate(String aPlugInId, EventsNotifier
      aNotifier, Function onReady);
}
```

```
//Callback that is executed when the plug-in instance has
//been generated successfully
onReadyCallback = function(aPlugIn){...}

//Example using the plug-in generator to generate
//the authentication plug-in
auth = generator.generate("auth", notifier, onReadyCallback);
```

- EventsBaseFilter: Filters on the client are designed to support aspects in the client using the server-side configuration totally transparent for developers. The main features are:
  - Validation of parameters on the client before execute requests, according to the server API metadata. Client validations avoid unnecessary responses from the server with the message “Invalid arguments exception” ;)
  - More off-line oriented applications by using cache, this reduce the server calls and reduce data redundancy on the network because of the data expiration capability.

- Security permissions checks per request. Avoid unnecessary "NotAuthorizedException" exceptions from the server.

In general with client the applications are benefits with more speed, less bandwidth consumption and desktop-like development style.

```
class EventsBaseFilter{
  public String id;
  public void initialize();
  public void beforeCall(Token aToken, OnResponseObject
      aOnResponseObject);
  public void afterCall(JSONObject aRequest, JSONObject
      aResponseEvent);
}
```

Current implemented filters:

- SecurityFilter: Support the security aspect.
- CacheFilter: Support the cache aspect, allowing RAM memory and HTML5 session and local storages. Uses the third-party library JsCache.
- ValidationFilter: Support the validation aspect.

- EventsPlugIn: Represents the base class of the generated plug-ins in the JavaScript side.

```
class EventsPlugIn{
  public String id;
  public EventsNotifier notifier;
  public JSONObject plugInAPI;

  //The methods are generated in runtime
}
```

How to setup the JavaScript environment?

Below are described the required steps to setup the JavaScript environment and how to configure all the required components.

```
//Creating the filter chain
securityFilter = new jws.SecurityFilter();
//Global "on not authorized" callback
securityFilter.OnNotAuthorized = function(aEvent){ ... }

cacheFilter = new jws.CacheFilter();
cacheFilter.cache = new Cache();

validatorFilter = new jws.ValidatorFilter();
```

```
//Creating the event notifier
notifier = new jws.EventsNotifier();
notifier.ID = "notifier0";
notifier.NS = "test";
notifier.jwsClient = jws.myConn;
notifier.filterChain = [securityFilter, cacheFilter, validatorFilter];

//Initializing the event notifier
notifier.initialize();

//Creating a plugin generator
generator = new jws.EventsPlugInGenerator();

//Generating the auth plug-in.
auth = generator.generate("auth", notifier, function(){});
```

Calling method on generated plug-ins:

Generated plug-ins contains methods that are shortcuts to notify the corresponding server-side plug-in C2S events.

```
auth.logon({
  args: {
    username: “aschulze”,
    password: “jws+rocks”
  },
  OnSuccess: function(aResponse){
    //Authenticated
  },
  OnFailure: function(aResponse){
    //Do something here
  }
});
```

Allowed callbacks to handle the response of the C2S event notifications:

- OnResponse: Receives the response of the C2S events notifications, without distinguish if it is success or failure.
- OnSuccess: Receives the success response of the C2S events notifications.
- OnFailure: Receives the failure response of the C2S events notifications.
- OnTimeout: Is called when the server exceeds the time limit to process the C2S event.

Developing with EventsPlugIn

The EventsPlugIn extension has been created to offer high productivity and efficiency levels during software development. The EventsPlugIn deals also with the aspects (AOP) related to the C2S communication, i.e cache, access permissions and validations, simplifying to the minimum the complexity of the development process.

An EventsPlugIn developer commonly needs to focus on:

- ✓ Specify the C2S and S2C events.
- ✓ Specify the services that will support the application requirements (including the Model layer if required).
- ✓ Specify the EM plug-ins that will process the events. The controller layer.
- ✓ Program and program in Java ;)
- ✓ Create the application configuration by using the Spring IoC Container (see “Configuring the application without Spring” section)
- ✓ Test
- ✓ Deploy

The environment

For the following tutorial will not use Maven, instead will use the jWebSocket 1.0 jars: “jWebSocketServer-Bundle-1.0.jar” and “jWebSocketEventsPlugIn.jar”.

The jWebSocket server package can be downloaded from at:

http://jwebsocket.org/downloads/downloads.htm

Will use OpenJDK 1.7 as JRE and JDK, NetBeans 7.1.1 as IDE and Google Chrome v18 as web browser.

The Operating System environment variable “JWEBSOCKET_HOME” is set to the jWebSocket directory path value (where the downloaded package was uncompressed).

The sample alarm application

“Online Alarm” is a very basic alarm-like application where users can define messages that will be displayed by the application in a given future time.

The objective instead to provide a huge set of features, is about to show all the EventsPlugIn features in a simple and comprehensive application.

Enjoy it ;)

The application requirements

1. Create new alarms: Application users should be able to create new alarms as they want, passing the alarm message and the future time were the message will be displayed.
2. List user alarms: A user should be able to show the list of previously defined alarms.
3. Alarm notifications: The system requires to notify the user alarms. The users only get alarm notifications if they are connected to the application on the alarms targeted time.
4. User authentication and authorization: All the clients requires to be authenticated and authorized to use the application features.

The server side

The server-side Java application will be organized under the namespace “org.jwebsocket.eventsplugin.alarm”.

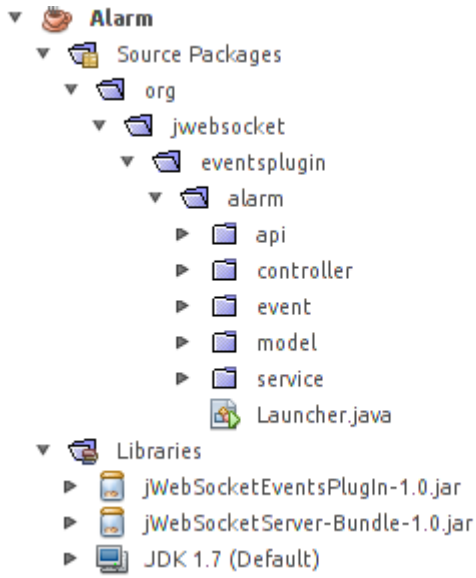

Illustration 2: NetBeans project view

- The application events

Is the time to define the events that will be associated to our application, these are: C2S events and S2C events.

- C2S events

The first C2S event correspond to our first application requirement wish is “Create new alarms”, so our event can class can be defined:

```
package org.jwebsocket.eventsplugin.alarm.event;

import org.jwebsocket.eventmodel.annotation.ImportFromToken;
import org.jwebsocket.eventmodel.event.C2SEvent;

public class CreateNewAlarm extends C2SEvent {

  private String mMessage;
  private Long mTime;

  public Long getTime() {
    return mTime;
  }

  @ImportFromToken
  public void setTime(String aTime) {
    this.mTime = Long.parseLong(aTime);
  }

  public String getMessage() {
    return mMessage;
  }

  @ImportFromToken
```

```
  public void setMessage(String aMessage) {
    this.mMessage = aMessage;
  }
}
```

The second and last C2S event correspond to our second application requirement, “List user alarms”.

```
package org.jwebsocket.eventsplugin.alarm.event;

import org.jwebsocket.eventmodel.event.C2SEvent;

public class ListAlarms extends C2SEvent {}
```

- S2C events

To support the requirement “Alarms notification”, our application require to send a S2C events to connected users. The S2C event class required:

```
package org.jwebsocket.eventsplugin.alarm.event;

import org.jwebsocket.eventmodel.event.S2CEvent;
import org.jwebsocket.token.Token;

public class AlarmActiveNotification extends S2CEvent {

  private String mMessage;

  public AlarmNotification(String aMessage) {
    this.mMessage = aMessage;
    this.setId(“alarmActive”)
  }

  @Override
  public void writeToToken(Token aToken) {
    aToken.setString("message", mMessage);
  }
}
```

- The Service and Model layers

Now is time to define the services and models that will support the application requirements. For this sample application will use simply memory maps, so the model layer has not to much complexity.

The alarm service interface:

```
package org.jwebsocket.eventsplugin.alarm.api;

import java.util.List;
import org.jwebsocket.eventsplugin.alarm.model.Alarm;

public interface IAlarmService {

  /**
```

```
   * Create a new alarm
   *
   * @param aAlarm The alarm to be created
   */
  void create(Alarm aAlarm);

  /**
   * Get all the created alarms of a user given it username
   *
   * @param aUsername The username value
   * @return
   */
  List<Alarm> list(String aUsername);
}
```

The Alarm entity class:

```
package org.jwebsocket.eventsplugin.alarm.model;

public class Alarm {

  private long mTime;
  private String mMessage;
  private String mUsername;

  public Alarm(long aTime, String aUsername, String aMessage) {
    mTime = aTime;
    mUsername = aUsername;
    mMessage = aMessage;
  }

  public String getMessage() {
    return mMessage;
  }

  public long getTime() {
    return mTime;
  }

  public String getUsername() {
    return mUsername;
  }

  @Override
  public String toString() {
    return "Alarm{" + "time=" + mTime + ", message=" + mMessage + ",
        username=" + mUsername + '}';
  }
}
```

The IAlarmService implementation:

```
package org.jwebsocket.eventsplugin.alarm.service;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.TimerTask;
import javolution.util.FastList;
import org.jwebsocket.eventmodel.observable.ObservableObject;
import org.jwebsocket.eventsplugin.alarm.api.IAlarmService;
import org.jwebsocket.eventsplugin.alarm.event.AlarmActive;
```

```
import org.jwebsocket.eventsplugin.alarm.model.Alarm;
import org.jwebsocket.util.Tools;

public class AlarmService extends ObservableObject implements IAlarmService {

  private Map<String, List<Alarm>> mAlarms = new HashMap();

  public AlarmService() {
    //Registering the service events
    addEvents(AlarmActive.class);
  }

  private List<Alarm> getUserAlarms(String aUsername) {
    if (!mAlarms.containsKey(aUsername)) {
      mAlarms.put(aUsername, new FastList());
    }

    return mAlarms.get(aUsername);
  }

  private class TTask extends TimerTask {

    private ObservableObject mSubject;
    private Alarm mAlarm;

    public TTask(ObservableObject aSubject, Alarm aAlarm) {
      mSubject = aSubject;
      mAlarm = aAlarm;
    }

    @Override
    public void run() {
      try {
        mSubject.notify(new AlarmActive(mAlarm));
      } catch (Exception ex) {
        //catch me...
      }
    }
  }

  @Override
  public void create(Alarm aAlarm) {
    getUserAlarms(aAlarm.getUsername()).add(aAlarm);

    long lDelay = aAlarm.getTime() - System.currentTimeMillis();
    //Using the jWebSocket utility Timer
    Tools.getTimer().schedule(new TTask(this, aAlarm), lDelay);
  }

  @Override
  public List<Alarm> list(String aUsername) {
    return getUserAlarms(aUsername);
  }
}
```

- The Controller layer (EM plug-ins)

The EventsPlugIn propose to build the controller layer by using the EM plug-ins (see “EventModel plug-ins” section).

Let's create our first controller to support the following alarm applications requirements (see “The application requirements” section):

- Create user alarms
- List user alarms
- Alarm notifications

```
package org.jwebsocket.eventsplugin.alarm.controller;

import java.util.Iterator;
import org.apache.log4j.Logger;
import org.jwebsocket.api.WebSocketConnector;
import org.jwebsocket.eventmodel.event.C2SResponseEvent;
import org.jwebsocket.eventmodel.observable.ResponseEvent;
import org.jwebsocket.eventmodel.plugin.EventModelPlugIn;
import org.jwebsocket.eventsplugin.alarm.event.AlarmActive;
import org.jwebsocket.eventsplugin.alarm.event.AlarmActiveNotification;
import org.jwebsocket.eventsplugin.alarm.event.CreateNewAlarm;
import org.jwebsocket.eventsplugin.alarm.event.ListAlarms;
import org.jwebsocket.eventsplugin.alarm.model.Alarm;
import org.jwebsocket.eventsplugin.alarm.service.AlarmService;
import org.jwebsocket.logging.Logging;

public class AlarmController extends EventModelPlugIn {

  private AlarmService mService;
  private static Logger mLog = Logging.getLogger();

  @Override
  public void initialize() throws Exception {
    //Listening the AlarmActive eventon the AlarmService subject...
    mService.on(AlarmActive.class, this);
  }

  public AlarmService getService() {
    return mService;
  }

  public void setService(AlarmService aService) {
    mService = aService;
  }

  public void processEvent(CreateNewAlarm aEvent, C2SResponseEvent
      aResponseEvent) throws Exception {
    if (mLog.isDebugEnabled()) {
       mLog.debug("Processing 'CreateNewAlarm' event: " +
           aEvent.getTime() + " - "     + aEvent.getMessage() + "...");
    }
    mService.create(new Alarm(
      aEvent.getTime(),
      aEvent.getConnector().getUsername(),
      aEvent.getMessage()));
  }

  public void processEvent(ListAlarms aEvent, C2SResponseEvent
      aResponseEvent) throws Exception {
    if (mLog.isDebugEnabled()) {
      mLog.debug("Processing 'ListAlarms' event: ...");
    }
    aResponseEvent.getArgs().setList("data",
        mService.list(aEvent.getConnector().getUsername()));
  }

  public void processEvent(AlarmActive aEvent, ResponseEvent
      aResponseEvent) throws Exception {
    if (mLog.isDebugEnabled()) {
      mLog.debug("Processing 'AlarmActive' event: " +
```

```
            aEvent.getAlarm().toString());
    }

    //Just online users are able to receive actives alarms
    //notifications
    for (Iterator<WebSocketConnector> it = getServerAllConnectors().
        values().iterator(); it.hasNext();) {
      WebSocketConnector lConnector = it.next();
      if (lConnector.getUsername().equals(
          aEvent.getAlarm().getUsername())) {
        notifyS2CEvent(new AlarmActiveNotification(
            aEvent.getAlarm().getMessage())).
               to(lConnector, null);
      }
    }
  }
}
```

Validating arguments

The EventsPlugIn ValidatorFilter, uses the arguments definitions (part of the events definitions, see “Event definitions and aspects” topic) type parameter to restrict the arguments value type. The ValidatorFilter also uses the validator parameter of the arguments definitions to improve the validation process by using the “org.springframework.validation.Validator” interface.

The CreateNewAlarm event definition:

The event receives two incoming arguments (time, message), the “time” argument is a long parseable string value, indicating a future time and the “message” argument is just a string value.

```
<bean parent="AbstractEventDefinition">
  <property name="id" value="alarm.create" />
  <property name="ns"
    value="org.jwebsocket.eventsplugin.alarm.event.CreateNewAlarm" />
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
  <property name="incomingArgsValidation">
    <set>
      <!-- The “time” argument is required, must be type of “string”
      and will be also validated by the “TimeValidator” -->
      <bean
        class="org.jwebsocket.eventmodel.filter.validator.Argument" >
         <property name="name" value="time" />
         <property name="type" value="string" />
        <property name="optional" value="false" />
        <property name="validator" ref="TimeValidator" />
       </bean>
      <!-- The “message” argument is required and
      must be type of “string” -->
       <bean
        class="org.jwebsocket.eventmodel.filter.validator.Argument" >
         <property name="name" value="message" />
         <property name="type" value="string" />
         <property name="optional" value="false" />
```

```
      </bean>
    </set>
  </property>
</bean>
```

The TimeValidator class:

```
package org.jwebsocket.eventsplugin.alarm.controller.validator;

import java.util.Date;
import org.jwebsocket.eventmodel.filter.validator.Argument;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class TimeValidator implements Validator {

  @Override
  public boolean supports(Class<?> aType) {
    if (aType.equals(String.class)) {
      return true;
    }
    return false;
  }

  @Override
  public void validate(Object aArg, Errors aErrors) {
    Argument lArg = (Argument) aArg;
    try {
      long lTime = Long.parseLong(lArg.getValue().toString());
      if (lTime <= new Date().getTime()) {
        aErrors.rejectValue(lArg.getName(), "The alarm time value is
          not valid. Please enter a future time value!");
      }
    } catch (Exception lEx) {
      aErrors.rejectValue(lArg.getName(), "The alarm time value
        cannot be parsed to Long!");
    }
  }
}
```

The ListAlarms event definition:

The event does not need incoming arguments, because the “username” value can be obtained from the WebSocketConnector instance. The correspondent response event have an argument named “data” wish contains the “List” of user alarms.

```
<bean parent="AbstractEventDefinition">
  <property name="id" value="alarm.list" />
  <property name="ns"
    value="org.jwebsocket.eventsplugin.alarm.event.ListAlarms" />
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
  <property name="outgoingArgsValidation">
    <set>
      <bean class="org.jwebsocket.eventmodel.filter.validator.Argument" >
        <property name="name" value="data" />
        <property name="type" value="array" />
        <property name="optional" value="false" />
       </bean>
    </set>
  </property>
```

```
</bean>
```

- Securing the application

Now it is time to secure our application, in the EventsPlugIn language it means: “set execution permissions for the C2S events notifications”.

In our Alarm application according to our requirements, will define one single role, it is the USER that interacts with the application.

The EventsPlugIn provide the support for Authentication and Authorization.

- Authentication

The EventsPlugIn supports Authentication through a core plug-in named “AuthPlugIn” (authenticator plug-in). The plug-in process two C2S events, Logon and Logoff.

When a user tries to get authenticated, it require send it “username” and “password”, in the server-side the AuthPlugIn authenticates the user against the defined Spring AuthenticationManager or AuthenticationProvider.

By default the security configuration of our application is located in the file $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/security.xml wish uses by default a static UserDetails service (see Spring Security

Reference Documentation).

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/security.xml -->
<authentication-manager>
  <authentication-provider>
    <user-service>
      <!-- Setting five users with the required application role -->
      <user name="aschulze" password="123" authorities="USER" />
      <user name="psingh" password="123" authorities="USER" />
      <user name="quentin" password="123" authorities="USER" />
      <user name="jan" password="123" authorities="USER" />
      <user name="kyberneees" password="123" authorities="USER”/>
    </user-service>
  </authentication-provider>
</authentication-manager>
```

The AuthenticationManager bean reference is assigned to the AuthPlugIn in the configuration file:

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/plugins.xml -->
<bean id="AuthPlugIn"
    class="org.jwebsocket.eventmodel.plugin.auth.AuthPlugIn"
    parent="AbstractPlugIn">
  <property name="id" value="auth" />
  <property name="authenticationManager"
      ref="org.springframework.security.authenticationManager" />
  <property name="emEventClassesAndClientAPI">
    <map>
      <entry key="logon" value="org.jwebsocket.eventmodel.event.auth.Logon"/>
      <entry key="logoff"
          value="org.jwebsocket.eventmodel.event.auth.Logoff"/>
    </map>
  </property>
</bean>
```

Authorization

Once the AuthPlugIn is configured and we know the role that the users require to execute events notifications, then we proceed to specify it in each C2S event definition:

```
<bean parent="AbstractEventDefinition">
  <property name="id" value="alarm.create" />
  <property name="ns"
    value="org.jwebsocket.eventsplugin.alarm.event.CreateNewAlarm"/>
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
  <property name="securityEnabled" value="true" />
    <property name="roles">
      <set>
        <value>USER</value>
      </set>
    </property>
  </property>
  <property name="incomingArgsValidation">
    <set>
      <bean
        class="org.jwebsocket.eventmodel.filter.validator.Argument">
         <property name="name" value="time" />
         <property name="type" value="string" />
        <property name="optional" value="false" />
        <property name="validator" ref="TimeValidator" />
       </bean>
       <bean
        class="org.jwebsocket.eventmodel.filter.validator.Argument">
         <property name="name" value="message" />
         <property name="type" value="string" />
         <property name="optional" value="false" />
      </bean>
    </set>
  </property>
</bean>
```

```
<bean parent="AbstractEventDefinition">
  <property name="id" value="alarm.list" />
  <property name="ns"
    value="org.jwebsocket.eventsplugin.alarm.event.ListAlarms" />
  <property name="responseRequired" value="true" />
  <property name="responseToOwnerConnector" value="true" />
```

```
  <property name="securityEnabled" value="true" />
    <property name="roles">
      <set>
        <value>USER</value>
      </set>
    </property>
  </property>
  <property name="outgoingArgsValidation">
    <set>
      <bean
        class="org.jwebsocket.eventmodel.filter.validator.Argument" >
        <property name="name" value="data" />
        <property name="type" value="array" />
        <property name="optional" value="false" />
       </bean>
    </set>
  </property>
</bean>
```

From now, the C2S events can be only notified by users that are authenticated and has the USER role.

- Configuring the application with Spring

The EventsPlugIn propose to use the Spring IoC Container to configure and bootstrap the applications (see “Spring framework integration” topic).

The application configuration (Spring IoC Container) is located in the directory: $JWEBSOCKET_HOME/conf/EventsPlugIn/**alarm**-application. Where “alarm” is the name (unique identifier) of our application.

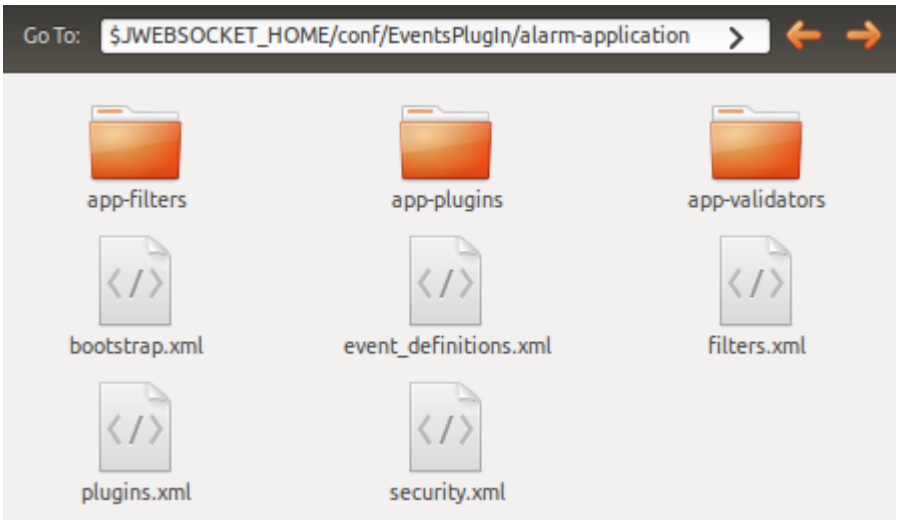

Illustration 3: Alarm application configuration directory

Files description:

bootstrap.xml: Main configuration file. Contains the EM instance's construction information. The file imports the configuration files: “events_definitions.xml”, “filters.xml”, “plugins.xml” and “security.xml”.

events_definitions.xml: Contains the application default events definitions. Specifically the events definitions for EM, System and Authentication plug-ins.

filters.xml: Contains the application default filters configuration. The file imports additional application filters configuration (app_filters/importer.xml).

plugins.xml: Contains de application core plug-ins (controllers) configuration (System and Authenticator). The file imports additional application plug-ins configuration (app_plugins/importer.xml), also imports the application validators configuration (app_validators/importer.xml).

The Alarm plug-in configuration:

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/app-
plugins/alarm.xml -->
<bean id="AlarmPlugIn"
class="org.jwebsocket.eventsplugin.alarm.controller.AlarmController"
  parent="AbstractPlugIn">
  <property name="id" value="alarm" />
  <property name="service">
    <bean id="AlarmService"
      class="org.jwebsocket.eventsplugin.alarm.service.AlarmService">
    </bean>
  </property>
  <property name="emEventClassesAndClientAPI">
    <map>
      <entry key="create" value="org.jwebsocket.eventsplugin.
          alarm.event.CreateNewAlarm"/>
      <entry key="list" value="org.jwebsocket.eventsplugin.
          alarm.event.ListAlarms"/>
    </map>
  </property>
  <property name="eventsDefinitions">
    <set>
      <!--  CreateNewAlarm and ListAlarms event definitions -->
    </set>
  </property>
</bean>
```

When related C2S events are going to be process by only one EM plug-in, the events definitions should be defined in the plug-in definition “eventsDefinitions” property.

security.xml: Contains the default application AuthenticationManager configuration. Suppose to be the file were the developers configure the AuthenticationManager or AuthenticationProvider components.

- Running the application

To run the alarm application, we will use the jWebSocket developer configuration template, wish is supposed to used during the development process: $JWEBSOCKET_HOME/conf/jWebSocketDevTemplate.xml.

Registering our Alarm application as a plug-in to be loaded:

```
<!-- $JWEBSOCKET_HOME/conf/jWebSocketDevTemplate.xml -->
  ...
  <plugins>
    <plugin>
      <name>org.jwebsocket.plugins.events.EventsPlugIn</name>
        <ns>alarm</ns>
        <id>jws.eventsplugin.alarm</id>
        <jar>jWebSocketEventsPlugIn-1.0.jar</jar>
        <server-assignments>
          <server-assignment>ts0</server-assignment>
        </server-assignments>
    </plugin>
  </plugins>
  ...
```

The application launcher class:

```
package org.jwebsocket.eventsplugin.alarm;

import org.jwebsocket.config.JWebSocketConfig;
import org.jwebsocket.factory.JWebSocketFactory;

public class Launcher {

  public static void main(String[] aArgs) {
    // the following line must not be removed due to GNU LGPL 3.0 license!
    JWebSocketFactory.printCopyrightToConsole();
    // check if home, config or bootstrap path are passed by command line
    JWebSocketConfig.initForConsoleApp(aArgs);
    try {
      // start the jWebSocket Server
      JwebSocketFactory.start("$
          {JWEBSOCKET_HOME}conf/jWebSocketDevTemplate.xml", null);
      // run server until shut down request
      JWebSocketFactory.run();
    } catch (Exception lEx) {
      System.out.println(
        lEx.getClass().getSimpleName()
        + " on starting jWebSocket server: "
        + lEx.getMessage());
    } finally {
      JWebSocketFactory.stop();
    }
  }
}
```

Now you can run the server-side application by running the Launcher.java file... jWebSocket server logs:

```
...
...
...
2012-05-20 15:05:58,129 DEBUG - EventsPlugIn: Creating EventsPlugIn instance
for 'alarm' application...
2012-05-20 15:05:58,130 DEBUG - SpringEventModelBuilder: Building EventModel
instance from: '/home/svn/ijwssvn/rte/jWebSocket-1.0/conf/EventsPlugIn/alarm-
application/bootstrap.xml'...
2012-05-20 15:05:59,309 DEBUG - SpringEventModelBuilder: EventModel instance
for 'alarm' application build successful!
...
...
...
2012-05-20 15:05:59,742 INFO  - JWebSocketFactory: jWebSocket server startup
complete
```

The client side

To create the Alarm web client, we require to import the following JavaScript files (download first the jWebSocket 1.0 web client package from http://jwebsocket.org/downloads/downloads.htm):

web/res/js/jWebSocket.js: Contains the jWebSocket JavaScript client.
web/res/js/jwsEventsPlugIn.js: Contains the EventsPlugIn JavaScript client layer.
web/res/js/jwsCache.js: Contains the JsCache (see “JsCache” section) third-party library.

- Setting up the connection with the server application

```
jws.serverConn = new jws.jWebSocketJSONClient();
jws.serverConn.open(jws.JWS_SERVER_URL, {
  OnWelcome: function (){
    //Creating the filter chain
    var securityFilter = new jws.SecurityFilter();
    securityFilter.OnNotAuthorized = function(aEvent){
      //”Not Authorized” application global callback!
    }

    var cacheFilter = new jws.CacheFilter();
    cacheFilter.cache = new Cache();
    var validatorFiler = new jws.ValidatorFilter();

    //Creating the event notifier
    var notifier = new jws.EventsNotifier();
    notifier.ID = "alarmNotifier";
    notifier.NS = "alarm";
    notifier.jwsClient = jws.serverConn;
    notifier.filterChain = [securityFilter, cacheFilter,
```

```
        validatorFiler];
    notifier.initialize();

    //Creating a plugin generator
    var generator = new jws.EventsPlugInGenerator();

    //Setting the application namespace
    jws.alarmApp = {};

    //Generating the auth & alarm client plug-ins
    jws.alarmApp.authPlugIn = generator.generate("auth", notifier,
    function(){});
    jws.alarmApp.alarmPlugIn = generator.generate("alarm",
    notifier, function(){});
  },
  OnClose: function(){
    console.log("You are not connected to the server!")
  }
});
```

Testing

Testing an application is a complex process, so for our Alarm application we just will execute functional tests from the client by using the Jasmine test framework (see "Jasmine" section).

Testing the authenticator plug-in "Logon" event notification:

```
testLogon: function() {
  var lSpec = this.NS + ": logon";
  it( lSpec, function () {
    var lResponse = null;
    var lUsername = "kyberneees";
    jws.alarmApp.authPlugIn.logon({
      args: {
        username: lUsername,
        password: "123"
      },
      OnResponse: function(aResponse){
        lResponse = aResponse;
      }
    });

    waitsFor(
      function() {
        return( lResponse != null );
      }, lSpec, 3000);

    runs( function() {
      expect( lResponse.code ).toEqual( 0 );
      expect( lResponse.username ).toEqual( lUsername );
      expect( lResponse.uuid ).toEqual( lUsername );
      expect( lResponse.roles instanceof Array ).toEqual( true );
    });
  });
}
```

Testing the alarm plug-in "CreateNewAlarm" event notification:

```
testCreate: function() {
  var lSpec = this.NS + ": create alarm";
  it( lSpec, function () {
    var lResponse = null;
    jws.alarmApp.alarmPlugIn.create({
      args: {
        //Active alarm on the next second
        time: (new Date().getTime() + 1000).toString(),
        message: "Alarm message!"
      },
      OnResponse: function(aResponse){
        lResponse = aResponse;
      }
    });
    waitsFor(
      function() {
        return( lResponse != null );
      }, lSpec, 3000);
    runs( function() {
      expect( lResponse.code ).toEqual( 0 );
    });
  });
}
```

Testing the alarm plug-in “AlarmActive” S2C event notification:

```
testAlarmActiveNotification: function() {
  var lSpec = this.NS + ": alarm active notification";
  it( lSpec, function () {
    var lAlarmMessage = null;
    //Setting the S2C event listener
    jws.alarmApp.alarmPlugIn.alarmActive = function(aEvent) {
      lAlarmMessage = aEvent.message;
    }
    waitsFor(
      function() {
        return( lAlarmMessage != null );
    }, lSpec, 3000);
    runs( function() {
      //Matching the arrived alarm message with the expected one
      expect( lAlarmMessage ).toEqual( "Alarm message!" );
    });
  });
}
```

Testing the alarm plug-in “ListAlarms” event notification:

```
testList: function() {
  var lSpec = this.NS + ": list alarms";
  it( lSpec, function () {
    var lResponse = null;
    jws.alarmApp.alarmPlugIn.list({
      OnResponse: function(aResponse){
         lResponse = aResponse;
      }
    });
    waitsFor(
      function() {
        return( lResponse != null );
      }, lSpec, 3000);

    runs( function() {
      expect( lResponse.code ).toEqual( 0 );
```

```
      var lLastPosition = lResponse.data.length - 1;
       expect( lResponse.data[lLastPosition].message )
         .toEqual( "Alarm message!" );
    });
  });
}
```

Testing the authenticator plug-in “Logoff” event notification:

```
testLogoff: function() {
  var lSpec = this.NS + ": logoff";
  it( lSpec, function () {
    var lResponse = null;
    jws.alarmApp.authPlugIn.logoff({
      OnResponse: function(aResponse){
        lResponse = aResponse;
      }
    });
    waitsFor(
      function() {
        return( lResponse != null );
    }, lSpec, 3000);
    runs( function() {
      expect( lResponse.code ).toEqual( 0 );
    });
  });
}
```

Illustration 4: Jasmine tests execution result

Deploying

In the EventsPlugIn language “deploying” means:

1. Generate the application jar
2. Copy the generated application jar (let's assume it is Alarm.jar) to the $JWEBSOCKET_HOME/libs/ directory.

3. Set the application configuration in the jWebSocket.xml plug-in section:

```
<!-- $JWEBSOCKET_HOME/conf/jWebSocket.xml -->
...
  <plugins>
    <plugin>
      <name>org.jwebsocket.plugins.events.EventsPlugIn</name>
        <ns>alarm</ns>
        <id>jws.eventsplugin.alarm</id>
        <jar>jWebSocketEventsPlugIn-1.0.jar</jar>
        <settings>
          <setting key="jars">Alarm.jar</setting>
        </settings>
        <server-assignments>
          <server-assignment>ts0</server-assignment>
        </server-assignments>
    </plugin>
  </plugins>
...
```

4. Start (or restart) the jWebSocket server ;)

Advanced topics

Supporting new aspects

The EventsPlugIn 1.0 support almost all common enterprise applications required aspects, it is done by using AOP through the combination of the C2S events definitions and the EM filters.

However, support new aspects for custom application requirements is a very simple task, just three steps:

1. Extend the C2SEventDefinition (see “Events definitions and aspects” section) class to provide new meta information about the target requirement.
2. Create the EM filter that will support the requirement.
3. Configure ;)

Let's explain it with an example for our Alarm application:

- New requirement: The users can't notify the “CreateNewAlarm” C2S event in time intervals less than one minute.

Extending the C2SEventDefinition class:

```
package org.jwebsocket.eventsplugin.alarm.filter;

import org.jwebsocket.eventmodel.event.C2SEventDefinition;

public class C2SEventDefinitionExt extends C2SEventDefinition {

  private Integer mMinimumNotificationTimeInterval;

  public Integer getMinimumNotificationTimeInterval() {
    return mMinimumNotificationTimeInterval;
  }

  public void setMinimumNotificationTimeInterval(
      Integer aMinimumNotificationTimeInterval) {
    mMinimumNotificationTimeInterval = aMinimumNotificationTimeInterval;
  }
}
```

The new filter:

```
package org.jwebsocket.eventsplugin.alarm.filter;
```

```
import java.util.Date;
import java.util.Map;
import javolution.util.FastMap;
import org.jwebsocket.api.WebSocketConnector;
import org.jwebsocket.eventmodel.event.C2SEvent;
import org.jwebsocket.eventmodel.event.C2SEventDefinition;
import org.jwebsocket.eventmodel.filter.EventModelFilter;

public class MinimumNotificationTimeIntervalFilter extends EventModelFilter {

  private Map<String, Long> mNotifications = new FastMap();

  @Override
  public void beforeCall(WebSocketConnector aConnector, C2SEvent aEvent)
      throws Exception {
    C2SEventDefinition lEventDef = getEm().getEventFactory().
        getEventDefinitions().getDefinition(aEvent.getId());

    //Processing only if the event definition is intance of
    //C2SEventDefinitionExt
    if (lEventDef instanceof C2SEventDefinitionExt) {
      String lKey = aConnector.getUsername() + aEvent.getId();
      Long lLastAccess = null;
      Boolean lUpdate = false;
      if (mNotifications.containsKey(lKey)) {
        //Getting the last notification time
        lLastAccess = mNotifications.get(lKey);
      } else {
        lUpdate = true;
      }

      if (null != lLastAccess) {
        //Checking if the notification interval is less than allowed
        if (lLastAccess + ((C2SEventDefinitionExt) lEventDef).
            getMinimumNotificationTimeInterval() * 60000 >
            new Date().getTime()) {
          throw new Exception("The C2S event '" +
              aEvent.getClass().getSimpleName()
              + "' can't be notified due to the minimum notification"
              + " time interval restrictions!");
        } else {
          lUpdate = true;
        }
      }

      if (lUpdate) {
        //Updating the last event notification time
        mNotifications.put(lKey, new Date().getTime());
      }
    }
  }
}
```

Updating the “CreateNewAlarm” C2S event definition:

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/app-
plugins/alarm.xml-->
...
<bean parent="AbstractEventDefinition"
    class="org.jwebsocket.eventsplugin.alarm.filter.C2SEventDefinitionExt">
  <property name="id" value="alarm.create" />
  <property name="ns"
      value="org.jwebsocket.eventsplugin.alarm.event.CreateNewAlarm" />
  <property name="responseRequired" value="true" />
```

```
  <property name="responseToOwnerConnector" value="true" />
  <!-- Time value is given in minutes -->
  <property name="minimumNotificationTimeInterval" value="1" />
  ...
```

Setting the new filter configuration:

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/app-
filters/filters.xml -->
...
<bean parent="AbstractFilter" id="MinimumNotificationTimeIntervalFilter"
    class="org.jwebsocket.eventsplugin.alarm.filter.
    MinimumNotificationTimeIntervalFilter">

</bean>
...
```

Registering the new filter in the EM filter chain:

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/alarm-application/bootstrap.xml -->
...
<property name="filterChain">
  <set>
    <ref bean="RouterFilter" />
    <ref bean="SecurityFilter" />
    <ref bean="MinimumNotificationTimeIntervalFilter" />
    <ref bean="CacheFilter" />
    <ref bean="ValidatorFilter" />
    <ref bean="AnnotationFilter" />
  </set>
</property>
...
```

Now run the Alarm application again ;)

The global exception handler

The EventsPlugIn provide a mechanism to process the controller layer (EM plug-ins) uncaught exceptions. It objective is to give always a response to the client(if required) even if the server interrupt the normal C2S event work-flow due to an uncaught exception.

The IExceptionHandler interface:

```
package org.jwebsocket.eventmodel.api;

import org.jwebsocket.api.IInitializable;

public interface IExceptionHandler extends IInitializable {

  void process(Exception ex);
}
```

The EventsPlugIn proposed exception handler implementation also provide a mechanism to notify interested listeners about the processed exceptions. It is useful to offer clean exceptions treatments or notify system's administrators about runtime exceptions using various sources like mail, chat, phone, etc...

The IExceptionNotifier interface:

```
package org.jwebsocket.eventmodel.api;

public interface IExceptionNotifier {

  void notify(Exception ex);
}
```

Setting the application exception handler:

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/appname-application/bootstrap.xml -->
<bean id="ExceptionHandler"
    class="org.jwebsocket.eventmodel.exception.ExceptionHandler"
    scope="singleton" init-method="initialize" destroy-method="shutdown">
  <property name="notifiers">
    <set>
      <!-- The notifiers set -->
    </set>
  </property>
</bean>
```

```
<!-- $JWEBSOCKET_HOME/conf/EventsPlugIn/appname-application/bootstrap.xml -->
<bean id="EventModel" class="org.jwebsocket.eventmodel.core.EventModel"
    init-method="initialize" destroy-method="shutdown" scope="singleton">
  <constructor-arg index="0" value="alarm"/>
  <constructor-arg index="1" ref="EventFactory"/>
  <constructor-arg index="2" ref="S2CEventNotificationHandler"/>
  <constructor-arg index="3" ref="ExceptionHandler"/>
  <property name="env" value="dev" />
  ...
```

Configuring the application without Spring

As we mention before, an EventsPlugIn application is in fact an EM instance. By default the EventsPlugIn uses the Spring Framework IoC Container to build the EM applications instances. However if don't want to use Spring, EventsPlugIn still being your friend ;)

The EventsPlugIn uses an abstract EM builder to bootstrap the applications, by default it uses “org.jwebsocket.plugins.events.SpringEventModelBuilder”, but you can create yours.

The IEventModelBuilder interface:

```
package org.jwebsocket.eventmodel.api;

import org.jwebsocket.eventmodel.core.EventModel;
import org.jwebsocket.plugins.events.EventsPlugIn;

public interface IEventModelBuilder {

  /**
   * Abstract builder for the EventModel instance
   *
   * @param aPlugIn The EventsPlugIn application instance
   * @return The EventModel instance to be used by the EventsPlugIn
   * @throws Exception
   */
  EventModel build(EventsPlugIn aPlugIn) throws Exception;
}
```

Configuration to setup a new EM builder:

```
<!-- $JWEBSOCKET_HOME/conf/jWebSocket.xml -->
...
<plugins>
  <plugin>
    <name>org.jwebsocket.plugins.events.EventsPlugIn</name>
      <ns>appname</ns>
      <id>jws.eventsplugin.appname</id>
      <jar>jWebSocketEventsPlugIn-1.0.jar</jar>
      <server-assignments>
        <server-assignment>ts0</server-assignment>
      </server-assignments>
      <settings>
        <setting key="em_builder">appname.bootstrap.MyEMBuilder</setting>
      </settings>
  </plugin>
</plugins>
...
```

Third-party libraries

○ Spring

Spring is the most popular application development framework for enterprise Java™. Millions of developers use Spring to create high performing, easily testable, reusable code without any lock-in.
WebSite: http://www.springsource.org/
License: Apache License 2.0

○ JsCache

Just a simple LRU cache written in javascript. It is loosely based on ASP.NET's Cache, and includes many caching options such as absolute expiration, sliding expiration, cache priority, and a callback function. It can be used to cache data locally in the user's browser, saving a server roundtrip in AJAX heavy applications.
WebSite: https://github.com/monsur/jscache/
License: MIT License

○ Jasmine

Jasmine is an open source testing framework for JavaScript. It aims to run on any JavaScript-enabled platform, to not intrude on the application nor the IDE, and to have easy-to-read syntax. It is heavily influenced by other unit testing frameworks, such as ScrewUnit, JSSpec, JSpec, and RSpec.
WebSite: http://pivotal.github.com/jasmine/
License: MIT License

Getting support

The EventsPlugIn extension is part of the official jWebSocket release, has been designed and developed by Rolando Santamaría Masó<kyberneees>, member of the jWebSocket development team.

Contact with kyberneees at:

- jWebSocket Forum: http://jwebsocket.org
- Twitter: https://twitter.com/#!/kyberneees

Report issues at: http://jwebsocket.org/issue_report.htm

Reference

Ben-Ari, M. 1990. "*Principles of Concurrent and Distributed Programming*". 1990. ISBN 0-13-711821-X.

Ferg, Stephen. 2006. Event-Driven Programming. [En línea] Enero de 2006. http://eventdrivenpgm.sourceforge.net/.

Framework Approach for WebSockets. **Schulze, Alexander. 2011.** Web Technologies & Internet Applications (WebTech 2011) : http://dl.globalstf.org/index.php?page=shop.product_details&flypage=flypage_images.tpl&product_id=528&category_id=42&option=com_virtuemart&Itemid=4&vmcchk=1&Itemid=4, 2011.

Garrett, Jesse James. 2005. Denken Über. *AJAX un nuevo acercamiento a Aplicaciones Web.* [En línea] 18 de febrero de 2005. http://www.uberbin.net/archivos/internet/ajax-un-nuevo-acercamiento-a-aplicaciones-web.php.

Gosling, James, y otros. 2005. *The Java language specification.* s.l. : Addison Wesley, 2005. 3ra Edition.

Hybi. 2011. The WebSocket protocol. *The WebSocket protocol.* [Online] septiembre 30, 2011. [Cited: diciembre 1, 2011.] http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17.

Kaazing Corporation. 2012. Kaazing, WebSocket Gateway. [En línea] 2012. [Citado el: 22 de 02 de 2012.] http://kaazing.com/products/kaazing-websocket-gateway.html.

Kirkpatrick, Marshall. 2009. Read Write Web. [En línea] 22 de 9 de 2009. [Citado el: 21 de 02 de 2012.] http://www.readwriteweb.com/archives/explaining_the_real-time_web_in_100_words_or_less.php.

Pressman, Roger S. 2005. *Ingeniería de Software. Un enfoque práctico.* 2005. ISBN: 9701054733.

SpringSource. 2011. Spring Reference Documentation. [Online] SpringSource,

2011. http://static.springsource.org/spring/docs/3.0.6.RELEASE/spring-framework-reference/html/.

—. 2012. SpringSource. [En línea] 2012. [Citado el: 22 de 02 de 2012.] http://www.springsource.org/about.

Symfony. 2011. A Gentle Introduction to symfony. [En línea] 2011. http://www.symfony-project.org/gentle-introduction/1_4/en/.

—. 2011. The symfony Reference Book. [Online] 2011. http://www.symfony-project.org/reference/1_4/en/.

w3schools.com. 2010. Introduction to Web Services. [En línea] 2010. http://www.w3schools.com/webservices/ws_intro.asp.

WebSocket für alle. **Alexander Schulze, Rolando Santamaría Masó. 2011.** 2, Germany: Mobile Developer Android, 2011.

Wikipedia. 2011. Aplicación web. *Aplicación web.* [En línea] 2011. [Citado el: 1 de diciembre de 2011.] http://es.wikipedia.org/wiki/Aplicaci%C3%B3n_web.

—. 2011. Software Framework. *Software Framework.* [En línea] 2011. [Citado el: 1 de diciembre de 2011.] http://en.wikipedia.org/wiki/Software_framework.

—. 2011. Web Application Framework. *Web Application Framework.* [En línea] 2011. [Citado el: 1 de diciembre de 2011.] http://en.wikipedia.org/wiki/Web_application_framework.

Wolff, Phil. 2009. Read Write Web. [En línea] Septiembre de 2009. [Citado el: 21 de 02 de 2012.] http://www.readwriteweb.com/archives/explaining_the_real-time_web_in_100_words_or_less.php.