**AngularJS example**

<http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/>

# Real-time Applications with AngularJS and Java – Part 1

Posted by: [Sylvain Cloutier](http://examples.javacodegeeks.com/author/sylvain-cloutier/) in [Core Java](http://examples.javacodegeeks.com/category/core-java/) September 16th, 2015

### Table Of Contents

[1. Introduction](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#introduction)

[2. The Periodic Refresh AJAX pattern](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#periodicRefreshPattern)

[3. RESTful JSON Java Back-end](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#restFullBackEnd)

[3.1. Maven dependencies](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#mavenDependencies)

[3.2. The TaskExecutor](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#taskExecutor)

[3.3. The Task object](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#taskObject)

[3.4 The Web Service](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#webService)

[3.5 The ManagedBean to start the tasks](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#managedBean)

[4. Front-end implementation with AngularJS](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#angularPart)

[5. What’s next](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#whatsNext)

[6. Download the Eclipse project](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/#down)

## 1. Introduction

In 2015, is it still acceptable to develop web applications in which we have to hit F5 to actualize the page content? The answer is simple: Of course yes! But still, we can offer our users a better overall experience. I could ask myself: How comes **Twitter** or **Facebook** are able to notify me when I have new interactions or messages, but the software I use at my job is not? This is where the real-time applications get on stage.

This article is the first of a series of three in which I want to introduce you the basic concepts behind real-time applications in **Java** with simple examples.  In this first article, I will demonstrate how **GMail**, **Facebook**, **Twitter** and many other websites have implemented real-time notifications using the **periodic refresh** design pattern.

In this example, I will be showing you how real-time updates can be useful in a system where the users execute tasks that take time to run (import of CSV files into the system, copy of files from server to server, batch update in the database, etc.). **AngularJS** will be used in the front end to implement the **Periodic Refresh** pattern. **Spring** and **Jackson** will be used together to create a **RESTful JSON Web Service** answering the AJAX request made by Angular. If you do not know about **AngularJS** or **Spring MVC**, I would suggest you read tutorials before. Here is a screenshot of the end result:

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/example.png)

Figure 1. Screenshot of the end result

## 2. The Periodic Refresh AJAX pattern

The **Periodic Refresh** pattern or polling is the simplest way of creating a real-time application. Basically, a JavaScript function periodically creates an XMLHttpRequest that is sent to the server. This request asks the server for updated information, then the view is actualized if necessary.

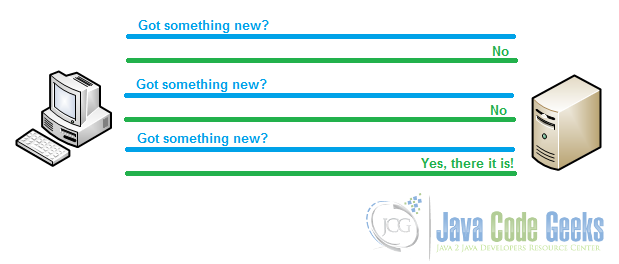
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/ClientServerBlank_PeriodicRefresh.png)

Figure 2. ClientServerBlank\_PeriodicRefresh

You can easily see this pattern in action in your Twitter feed. In any modern browser, hit F12, go to the Network tab and filter the requests so only XHR are displayed. Every 10 seconds or so, a new request is sent to the server asking for an update regarding new tweets. If there is any, a notification is displayed.

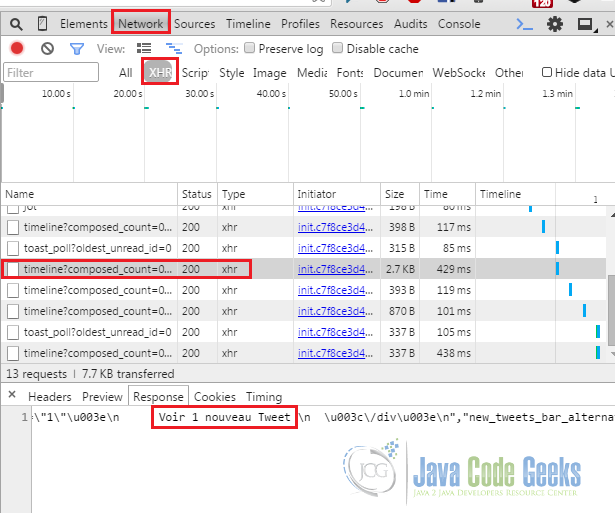
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/TwitterPeriodicRefresh.png)

Figure 3. TwitterPeriodicRefresh

## 3. RESTful JSON Java Back-end

### 3.1. Maven dependencies

In order to create a simple REST web service that will accept and answer JSON objects, you have to include **Spring MVC** and **Jackson**. Your pom.xml file should contain the following dependencies:

pom.xml

|  |  |
| --- | --- |
| 01 | <dependency> |
| 02 | <groupId>org.springframework</groupId> | |

|  |  |  |
| --- | --- | --- |
| 03 | <artifactId>spring-core</artifactId> | |
| 04 | <version>4.2.1.RELEASE</version> |

|  |  |  |
| --- | --- | --- |
| 05 | </dependency> | |
| 06 | <dependency> |

|  |  |
| --- | --- |
| 07 | <groupId>org.springframework</groupId> |
| 08 | <artifactId>spring-webmvc</artifactId> |

|  |  |  |
| --- | --- | --- |
| 09 | <version>4.2.1.RELEASE</version> | |
| 10 | </dependency> |

|  |  |
| --- | --- |
| 11 | <dependency> |
| 12 | <groupId>com.fasterxml.jackson.core</groupId> | |

|  |  |  |
| --- | --- | --- |
| 13 | <artifactId>jackson-databind</artifactId> | |
| 14 | <version>2.6.1</version> |

|  |  |  |
| --- | --- | --- |
| 15 | </dependency> | |
| 16 | <dependency> |

|  |  |
| --- | --- |
| 17 | <groupId>javax.servlet</groupId> |
| 18 | <artifactId>javax.servlet-api</artifactId> | |

|  |  |  |
| --- | --- | --- |
| 19 | <version>3.1.0</version> | |
| 20 | </dependency> |

### 3.2. The Task object

Then, we want to create our Task object that will be used in the web service. This Task object has a duration in milliseconds and a status that can either be IDLE, RUNNING or SUCCESS.

Task.java

|  |  |
| --- | --- |
| 01 | public class Task { |
| 02 | private TaskStatus status = TaskStatus.IDLE; | |

|  |  |  |
| --- | --- | --- |
| 03 | private long duration; | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | // Getters and Setters... | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | public void decrementDuration() { | |
| 08 | this.duration--; |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | public boolean isRunning() { |
| 12 | return this.status.equals(TaskStatus.RUNNING); | |

|  |  |  |
| --- | --- | --- |
| 13 | } | |
| 14 |  |

|  |  |
| --- | --- |
| 15 | public String getName() { |
| 16 | return this.toString(); |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | public void start() { |
| 20 | this.status = TaskStatus.RUNNING; | |

|  |  |  |
| --- | --- | --- |
| 21 | } | |
| 22 | } |

### 3.3. The TaskExecutor

Those Task objects, once instanciated, will be managed by a class called TaskExecutor. This class is a **Spring Singleton** that holds the tasks submitted by all users. Once it’s instanciated, it starts a thread that loops through the running tasks in the tasks pool and decrement their duration. When the duration gets to zero, the status is set to SUCCESS:

TaskExecutor.java

|  |  |  |
| --- | --- | --- |
| 01 | package com.javacodegeeks.examples.realtimeapp.part1.services; | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | import java.util.LinkedList; | |
| 04 | import java.util.List; |

|  |  |
| --- | --- |
| 05 |  |
| 06 | import javax.annotation.PostConstruct; | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | import org.springframework.context.annotation.Scope; | |

|  |  |  |
| --- | --- | --- |
| 09 | import org.springframework.stereotype.Component; | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | import com.javacodegeeks.examples.realtimeapp.part1.domain.Task; |
| 12 | import com.javacodegeeks.examples.realtimeapp.part1.domain.TaskStatus; | |

|  |  |
| --- | --- |
| 13 |  |
| 14 | @Component | |

|  |  |
| --- | --- |
| 15 | @Scope("singleton") |
| 16 | public class TaskExecutor { | |

|  |  |  |
| --- | --- | --- |
| 17 | private List pool = new LinkedList<>(); | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | @PostConstruct |
| 20 | public void initialize() { | |

|  |  |  |
| --- | --- | --- |
| 21 | Runnable taskPoolConsumer = () -> { | |
| 22 | while (true) { |

|  |  |
| --- | --- |
| 23 | try { |
| 24 | this.pool.stream() | |

|  |  |  |
| --- | --- | --- |
| 25 | .filter(task -> task.isRunning() && task.getDuration() > 0) | |
| 26 | .forEach(task -> task.decrementDuration()); |

|  |  |
| --- | --- |
| 27 |  |
| 28 | this.pool.stream() | |

|  |  |  |
| --- | --- | --- |
| 29 | .filter(task -> task.isRunning() && task.getDuration() == 0) | |
| 30 | .forEach(task -> task.setStatus(TaskStatus.SUCCESS)); |

|  |  |
| --- | --- |
| 31 |  |
| 32 | Thread.sleep(1000); | |

|  |  |  |
| --- | --- | --- |
| 33 | } catch (Exception e) { | |
| 34 | e.printStackTrace(); |

|  |  |  |
| --- | --- | --- |
| 35 | } | |
| 36 | } |

|  |  |  |
| --- | --- | --- |
| 37 | }; | |
| 38 |  |

|  |  |  |
| --- | --- | --- |
| 39 | new Thread(taskPoolConsumer).start(); | |
| 40 | } |

|  |  |
| --- | --- |
| 41 |  |
| 42 | public void startAllTasks() throws InterruptedException { | |

|  |  |  |
| --- | --- | --- |
| 43 | this.pool.stream().forEach(task -> task.start()); | |
| 44 | } |

|  |  |
| --- | --- |
| 45 |  |
| 46 | public List getPool() { | |

|  |  |  |
| --- | --- | --- |
| 47 | return this.pool; | |
| 48 | } |

|  |  |
| --- | --- |
| 49 |  |
| 50 | public void addTask(Task taskToAdd) { | |

|  |  |  |
| --- | --- | --- |
| 51 | this.pool.add(taskToAdd); | |
| 52 | } |

|  |  |
| --- | --- |
| 53 |  |
| 54 | } | |

### 3.4 The Web Service

To create the web service, we will be using the @RestController annotation from **Spring**. This web service will be mapped to "/api/task" and will answer POST and GET requests.

TaskService.java

|  |  |
| --- | --- |
| 01 | @RestController |
| 02 | @RequestMapping("/api/task") | |

|  |  |  |
| --- | --- | --- |
| 03 | public class TaskService { | |
| 04 | @Autowired |

|  |  |  |
| --- | --- | --- |
| 05 | private TaskExecutor taskExecutor; | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | @RequestMapping(method = RequestMethod.GET) | |
| 08 | public List getTasks() { |

|  |  |  |
| --- | --- | --- |
| 09 | return this.taskExecutor.getPool(); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 |  |
| 12 | @RequestMapping(method = RequestMethod.POST) | |

|  |  |  |
| --- | --- | --- |
| 13 | public void addTask(@RequestBody Task taskToAdd) { | |
| 14 | this.taskExecutor.addTask(taskToAdd); |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | public void startIdleTasks() throws InterruptedException { | |
| 18 | this.taskExecutor.startAllTasks(); |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 |  |

|  |  |
| --- | --- |
| 21 | } |

### 3.5 The ManagedBean to start the tasks

Finally, we have a managed bean of **JSF** to execute the code behind the last button.

TaskController.java

|  |  |  |
| --- | --- | --- |
| 01 | @ManagedBean(name = "taskController", eager=true) | |
| 02 | @Component |

|  |  |
| --- | --- |
| 03 | @RequestScoped |
| 04 | public class TaskController { | |

|  |  |
| --- | --- |
| 05 | @Autowired |
| 06 | private TaskService taskService; | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | public void startTasks(ActionEvent event) throws InterruptedException { | |

|  |  |  |
| --- | --- | --- |
| 09 | this.taskService.startIdleTasks(); | |
| 10 | } |

|  |  |  |
| --- | --- | --- |
| 11 |  | |
| 12 | } |

## 4. Front-end implementation with AngularJS

First, you want to create your module, then your controller. In the end result screenshot above, our controller will manage the first three buttons (Add Task, Refresh Tasks and Activate Auto Refresh). The last button is a **JSF** button managed by a backing bean.

index.xhtml

|  |  |
| --- | --- |
| 01 | var part1 = angular.module("part1", []); |
| 02 | part1.controller("RealtimeCtrl", function($scope, $http, $timeout) { | |

|  |  |  |
| --- | --- | --- |
| 03 | $scope.tasks = []; | |
| 04 |  |

|  |  |
| --- | --- |
| 05 | $scope.addTask = function() { |
| 06 | $http.post("api/task", $scope.task); | |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | $scope.getTasks = function() { | |
| 10 | $http.get("api/task") |

|  |  |  |
| --- | --- | --- |
| 11 | .success(function(data) { | |
| 12 | $scope.tasks = data; |

|  |  |  |
| --- | --- | --- |
| 13 | }); | |
| 14 | } |

|  |  |
| --- | --- |
| 15 |  |
| 16 | $scope.activateRealtime = function() { | |

|  |  |
| --- | --- |
| 17 | $scope.getTasks(); |
| 18 | $timeout($scope.activateRealtime, 1000); | |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 |  |

|  |  |
| --- | --- |
| 21 | }); |

I used Angular’s dependency injection to get the $scope, $http and $timeout services. In the Angular scope, I initially set the list of tasks to an empty array which will be replaced by the tasks array returned by the web service. Now, we have our three functions. The first one is addTask() which simply creates an Ajax POST request to the server with the task object from Angular’s scope in the request data. The second function above is getTasks() which creates an Ajax GET request to the server. The server will return an array of JSON objects corresponding to the tasks registered on the server. The last but not the least function is the implementation of the real-time feel, that is a recursive function with a pause of a second that simply retrieves the tasks using the controller’s function getTask(). This is the complete index.xhtml code:

index.xhtml

|  |  |
| --- | --- |
| 01 | <!DOCTYPE html> |
| 02 | <html xmlns="<http://www.w3.org/1999/xhtml>" | |

|  |  |
| --- | --- |
| 03 | xmlns:h="<http://java.sun.com/jsf/html>" |
| 04 | xmlns:f="<http://java.sun.com/jsf/core>"> | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | <h:head> | |

|  |  |  |
| --- | --- | --- |
| 07 | | <title>Real-time applications - Part 1 - Java Code Geeks</title> |
| 08 | <link rel="stylesheet" href="<https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css>"/> | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 09 | <script src="<https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.5/angular.min.js>"></script> | | |
| 10 | |  |

|  |  |
| --- | --- |
| 11 | <script> |
| 12 | var part1 = angular.module("part1", []); | |

|  |  |  |
| --- | --- | --- |
| 13 | part1.controller("RealtimeCtrl", function($scope, $http, $timeout) { | |
| 14 | $scope.tasks = []; |

|  |  |
| --- | --- |
| 15 |  |
| 16 | $scope.addTask = function() { | |

|  |  |  |
| --- | --- | --- |
| 17 | $http.post("api/task", $scope.task); | |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |
| 20 | $scope.getTasks = function() { | |

|  |  |
| --- | --- |
| 21 | $http.get("api/task") |
| 22 | .success(function(data) { | |

|  |  |  |
| --- | --- | --- |
| 23 | $scope.tasks = data; | |
| 24 | }); |

|  |  |  |
| --- | --- | --- |
| 25 | } | |
| 26 |  |

|  |  |  |
| --- | --- | --- |
| 27 | $scope.activateRealtime = function() { | |
| 28 | $scope.getTasks(); |

|  |  |  |
| --- | --- | --- |
| 29 | $timeout($scope.activateRealtime, 1000); | |
| 30 | } |

|  |  |
| --- | --- |
| 31 |  |
| 32 | }); | |

|  |  |
| --- | --- |
| 33 |  |
| 34 | </script> | |

|  |  |  |
| --- | --- | --- |
| 35 | </h:head> | |
| 36 |  |

|  |  |
| --- | --- |
| 37 | <h:body> |
| 38 | <div ng-app="part1" ng-controller="RealtimeCtrl" class="container"> | |

|  |  |  |
| --- | --- | --- |
| 39 | <h1>Real-time application <SMALL>part 1</SMALL></h1> | |
| 40 | <h2>Add task</h2> |

|  |  |
| --- | --- |
| 41 | <h:form> |
| 42 | <label for="durationField">Duration (in seconds):</label> | |

|  |  |
| --- | --- |
| 43 | <input type="number" id="durationField" class="form-control" ng-model="task.duration"/> |
| 44 | <button type="button" ng-click="addTask()" class="btn btn-success">Add task</button> |

|  |  |
| --- | --- |
| 45 | <button type="button" ng-click="getTasks()" class="btn btn-default">Refresh Tasks</button> |
| 46 | <button type="button" ng-click="activateRealtime()" class="btn btn-default">Activate Auto Refresh</button> |

|  |  |  |
| --- | --- | --- |
| 47 | <h:commandButton actionListener="#{taskController.startTasks}" | |
| 48 | styleClass="btn btn-default" |

|  |  |  |
| --- | --- | --- |
| 49 | value="Start Idle Tasks"> | |
| 50 | <f:ajax execute="@form"/> |

|  |  |  |
| --- | --- | --- |
| 51 | </h:commandButton> | |
| 52 | </h:form> |

|  |  |
| --- | --- |
| 53 |  |
| 54 | <h2>Listing</h2> | |

|  |  |
| --- | --- |
| 55 | <ul class="list-group"> |
| 56 | <li ng-repeat="curTask in tasks" class="list-group-item {{curTask.running ? 'active' : ''}}"> | |

|  |  |  |
| --- | --- | --- |
| 57 | {{curTask.name}} ({{curTask.status}})<span class="badge">{{curTask.duration}}</span> | |
| 58 | </li> |

|  |  |  |
| --- | --- | --- |
| 59 | </ul> | |
| 60 | </div> |

|  |  |  |
| --- | --- | --- |
| 61 | </h:body> | |
| 62 | </html> |

## 5. What’s next?

Obviously, there are tons of different ways of implementing a real-time application with the **Periodic Refresh** pattern. I picked **AngularJS** along **RESTful JSON Web service** because **AngularJS** really simplifies the update of the UI without having to refresh the page.

In the next article, I will reuse the same application but I will show you how have a better feel of the real-time application. The [**part 2**](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-2/) will be on **Long Polling** and the [**part 3**](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-3/) on the spanking new **HTML 5 WebSocket**.

# Real-time Applications with AngularJS and Java – Part 2

Posted by: [Sylvain Cloutier](http://examples.javacodegeeks.com/author/sylvain-cloutier/) in [Core Java](http://examples.javacodegeeks.com/category/core-java/) September 29th, 2015

## 1. Introduction

As the title of this article implies, this is the second part of how to create a **real-time application** using **AngularJS** and **Java**. The [first part](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/) showed how to automatically refresh a page content using the **periodic refresh** AJAX design pattern. Here, I will show and explain the concept behind **long** **polling**.

If you have not, I would suggest that you read and try the example of [part 1](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/). I will use the same skeleton application and modify it, it’s important that you know the basic concepts of the part 1 as I will not explain them again here.

Moreover, a basic knowledge of **AngularJS** and **Spring** is important as I will not explain how to set your workspace up, nor will I explain how they interact with each other.

## 2. Long Polling

**Long polling** is a concept used to emulate the **server push** (CometD, Bayeux, Atmosphere, WebSocket, etc.). Basically, the client starts an XMLHttpRequest with the server using Ajax. The server then accepts the request and check for updated information to send to the client. If the server does not find any new data, it loops until it finds or until a fixed amount of time to avoid infinite loops or client connection timeout.

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/longpolling.png)

Figure 1. Long polling in action

At the time of writing this article, **Facebook** uses **Long Polling** to update the UI with new information. Using **Google Chrome** or any new browser’s network analyzer, you can see it in action. Go to your **Facebook** home page and hit F12. Go to the network tab and filter to show only XHR. You’ll see that a request is sent to the server through a specific **pull channel** and stays in the Pending state for a little while, then the request is completed, a new one is started and so on.

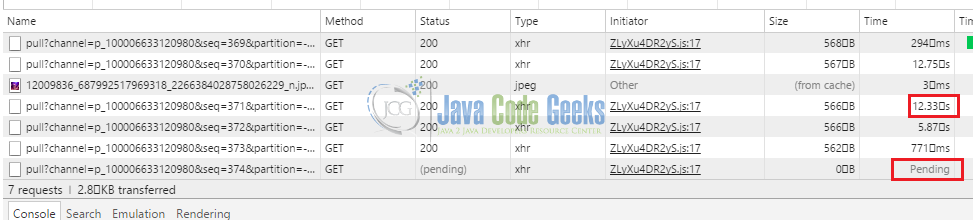
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/fb_example.png)

Figure 2. Facebook’s long polling

The main advantage of this method vs the **periodic refresh** pattern is that we reduce quite a lot the number of requests sent to the server. On the other hand, this uses and holds a thread from the server’s thread pool which could potentially run out of free threads. That means a user would get locked out of the system until a thread is freed, but this is not a show stopper if the server is properly configured or if you have load balancing on different instances.

## 3. The RESTful JSON Java Back-end

### 3.1. The new Task object status

As I said in the introduction, I will modify the example of part 1 in which the Task object had a duration that was decremented by a thread every second or so. That meant the data was actually changing quite often, so the **periodic refresh** was a good solution to display those changes to the client. We simply set the refresh rate at 1 second and it appeared to be **real-time**. Regarding the **Long Polling**, it would not make much sense to have the data updated that often. What we want to emulate is the server telling the client: “Hold on, I will send you data once I got something new for you.“. The data has to be unpredictably updated to see the long polling in action. To implement that, I will add a new TaskStatus that a Task can be in that is CREATED.

TaskStatus.java

|  |  |  |
| --- | --- | --- |
| 1 | public enum TaskStatus { | |
| 2 | CREATED, |

|  |  |
| --- | --- |
| 3 | IDLE, |
| 4 | RUNNING, | |

|  |  |  |
| --- | --- | --- |
| 5 | SUCCESS; | |
| 6 | } |

### 3.2. The Task object

The new version of the Task object need to be instantiated with the new status by default, meaning that all new Tasks will be created with the CREATED status.

Task.java

|  |  |
| --- | --- |
| 01 | public class Task { |
| 02 | private TaskStatus status = TaskStatus.CREATED; | |

|  |  |  |
| --- | --- | --- |
| 03 | private long duration; | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | public TaskStatus getStatus() { | |
| 06 | return status; |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | public void setStatus(TaskStatus status) { | |
| 10 | this.status = status; |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | public long getDuration() { | |
| 14 | return duration; |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | public void setDuration(long duration) { | |
| 18 | this.duration = duration; |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 |  |

|  |  |  |
| --- | --- | --- |
| 21 | public void decrementDuration() { | |
| 22 | this.duration--; |

|  |  |  |
| --- | --- | --- |
| 23 | } | |
| 24 |  |

|  |  |
| --- | --- |
| 25 | public boolean isRunning() { |
| 26 | return this.status.equals(TaskStatus.RUNNING); | |

|  |  |  |
| --- | --- | --- |
| 27 | } | |
| 28 |  |

|  |  |
| --- | --- |
| 29 | public String getName() { |
| 30 | return this.toString(); |

|  |  |  |
| --- | --- | --- |
| 31 | } | |
| 32 |  |

|  |  |
| --- | --- |
| 33 | public void start() { |
| 34 | this.status = TaskStatus.RUNNING; | |

|  |  |  |
| --- | --- | --- |
| 35 | } | |
| 36 | } |

### 3.3. The TaskCreator

To emulate users creating new Tasks, I created a TaskCreator object that randomly creates a new Task with the status CREATED. The point is that, unlike the previous example of part 1, I will query only for new information instead of the whole thing. That obviously will reduce the amount of data transferred over the network.

TaskCreator.java

|  |  |
| --- | --- |
| 01 | @Component |
| 02 | @Scope("singleton") | |

|  |  |
| --- | --- |
| 03 | public class TaskCreator { |
| 04 | private static final int MAX\_TASK\_DURATION = 5000; | |

|  |  |  |
| --- | --- | --- |
| 05 | private static final int MAX\_TASK\_CREATION\_INTERVAL = 10000; | |
| 06 | private static final Random RANDOMIZER = new Random(); |

|  |  |
| --- | --- |
| 07 |  |
| 08 | @Autowired | |

|  |  |  |
| --- | --- | --- |
| 09 | private TaskExecutor executor; | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | public void start() { | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | Runnable taskPoolConsumer = () -> { | |
| 14 | synchronized (executor) { |

|  |  |  |
| --- | --- | --- |
| 15 | while (true) { | |
| 16 | try { |

|  |  |  |
| --- | --- | --- |
| 17 | Task newTask = new Task(); | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | newTask.setStatus(TaskStatus.CREATED); |
| 20 | newTask.setDuration(RANDOMIZER.nextInt(MAX\_TASK\_DURATION)); | |

|  |  |  |
| --- | --- | --- |
| 21 | this.executor.addTask(newTask); | |
| 22 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 | this.executor.wait(RANDOMIZER.nextInt(MAX\_TASK\_CREATION\_INTERVAL)); | | |
| 24 | | } catch (Exception e) { |

|  |  |  |
| --- | --- | --- |
| 25 | e.printStackTrace(); | |
| 26 | } |

|  |  |  |
| --- | --- | --- |
| 27 | } | |
| 28 | } |

|  |  |  |
| --- | --- | --- |
| 29 | }; | |
| 30 |  |

|  |  |  |
| --- | --- | --- |
| 31 | new Thread(taskPoolConsumer).start(); | |
| 32 | } |

|  |  |
| --- | --- |
| 33 | } |

### 3.4. The TaskExecutor

As I said above, we want to improve the application so it only returns Task objects that have changed. A Task will be considered as changed if it’s either new or if its status has changed since the last time it was queried. For simplicity’s sake, this example will work only for one user. You could, like **Facebook** does, have a **channel** opened for each user and compute the delta between what is in the UI and what is in the back-end. To compute the delta in this example, I will simply keep a second list of Task in which will be added Task that were started or completed. This deals pretty badly with concurrency, but again, for simplicity’s sake, I decided that this was enough to show the concept.

TaskExecutor.java

|  |  |
| --- | --- |
| 01 | @Component |
| 02 | @Scope("singleton") | |

|  |  |
| --- | --- |
| 03 | public class TaskExecutor { |
| 04 | private List pool = new LinkedList<>(); | |

|  |  |  |
| --- | --- | --- |
| 05 | private Set updatedTaskPool = new HashSet<>(); | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | @PostConstruct |
| 08 | public void initialize() { | |

|  |  |  |
| --- | --- | --- |
| 09 | Runnable taskPoolConsumer = () -> { | |
| 10 | synchronized(this) { |

|  |  |  |
| --- | --- | --- |
| 11 | while (true) { | |
| 12 | try { |

|  |  |
| --- | --- |
| 13 | this.pool.stream() |
| 14 | .filter(task -> task.isRunning() && task.getDuration() > 0) | |

|  |  |
| --- | --- |
| 15 | .forEach(task -> { |
| 16 | task.decrementDuration(); | |

|  |  |  |
| --- | --- | --- |
| 17 | }); | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | this.pool.stream() |
| 20 | .filter(task -> task.isRunning() && task.getDuration() == 0) | |

|  |  |
| --- | --- |
| 21 | .forEach(task -> { |
| 22 | task.setStatus(TaskStatus.SUCCESS); | |

|  |  |  |
| --- | --- | --- |
| 23 | this.updatedTaskPool.add(task); | |
| 24 | }); |

|  |  |
| --- | --- |
| 25 |  |
| 26 | this.wait(1000); | |

|  |  |  |
| --- | --- | --- |
| 27 | } catch (Exception e) { | |
| 28 | e.printStackTrace(); |

|  |  |  |
| --- | --- | --- |
| 29 | } | |
| 30 | } |

|  |  |  |
| --- | --- | --- |
| 31 | } | |
| 32 | }; |

|  |  |
| --- | --- |
| 33 |  |
| 34 | new Thread(taskPoolConsumer).start(); | |

|  |  |  |
| --- | --- | --- |
| 35 |  | |
| 36 | } |

|  |  |
| --- | --- |
| 37 |  |
| 38 | public synchronized List getUpdatedTasks() { | |

|  |  |  |
| --- | --- | --- |
| 39 | List updatedTasks = new LinkedList<>(); | |
| 40 |  |

|  |  |
| --- | --- |
| 41 | updatedTasks.addAll(this.pool.stream() |
| 42 | .filter(task -> task.getStatus().equals(TaskStatus.CREATED)) | |

|  |  |
| --- | --- |
| 43 | .collect(Collectors.toList())); |
| 44 | updatedTasks.addAll(this.updatedTaskPool); | |

|  |  |
| --- | --- |
| 45 |  |
| 46 | this.changeCreatedStatusToIdle(); | |

|  |  |  |
| --- | --- | --- |
| 47 | this.updatedTaskPool.clear(); | |
| 48 |  |

|  |  |  |
| --- | --- | --- |
| 49 | return updatedTasks; | |
| 50 | } |

|  |  |
| --- | --- |
| 51 |  |
| 52 | private void changeCreatedStatusToIdle() { | |

|  |  |
| --- | --- |
| 53 | this.pool.stream() |
| 54 | .filter(task -> task.getStatus().equals(TaskStatus.CREATED)) | |

|  |  |  |
| --- | --- | --- |
| 55 | .forEach(task -> task.setStatus(TaskStatus.IDLE)); | |
| 56 | } |

|  |  |
| --- | --- |
| 57 |  |
| 58 |  | |

|  |  |  |
| --- | --- | --- |
| 59 | public synchronized void startAllTasks() throws InterruptedException { | |
| 60 | this.pool.stream() |

|  |  |  |
| --- | --- | --- |
| 61 | .filter(task -> task.getStatus().equals(TaskStatus.IDLE)) | |
| 62 | .forEach(task -> { |

|  |  |
| --- | --- |
| 63 | task.start(); |
| 64 | this.updatedTaskPool.add(task); | |

|  |  |  |
| --- | --- | --- |
| 65 | }); | |
| 66 | } |

|  |  |
| --- | --- |
| 67 |  |
| 68 | public List getPool() { | |

|  |  |  |
| --- | --- | --- |
| 69 | this.changeCreatedStatusToIdle(); | |
| 70 | return this.pool; |

|  |  |  |
| --- | --- | --- |
| 71 | } | |
| 72 |  |

|  |  |  |
| --- | --- | --- |
| 73 | public void addTask(Task taskToAdd) { | |
| 74 | this.pool.add(taskToAdd); |

|  |  |  |
| --- | --- | --- |
| 75 | } | |
| 76 | } |

### 3.5. TaskService

In our TaskService, we want to **inject** the new TaskCreator **singleton** and start it at the initialization. Then, we want to create a new mapping for our RestController that is to make the distinction between the function that returns all Task and the one that returns only updated information. That last one will implement the loop necessary for **long polling**.

TaskService.java

|  |  |
| --- | --- |
| 01 | @RestController |
| 02 | @RequestMapping("/api/task") | |

|  |  |  |
| --- | --- | --- |
| 03 | public class TaskService { | |
| 04 | @Autowired |

|  |  |  |
| --- | --- | --- |
| 05 | private TaskExecutor taskExecutor; | |
| 06 | @Autowired |

|  |  |  |
| --- | --- | --- |
| 07 | private TaskCreator taskCreator; | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | @PostConstruct |
| 10 | public void initialize() { | |

|  |  |  |
| --- | --- | --- |
| 11 | this.taskCreator.start(); | |
| 12 | } |

|  |  |
| --- | --- |
| 13 |  |
| 14 | @RequestMapping(path = "/all", method = RequestMethod.GET) | |

|  |  |
| --- | --- |
| 15 | public List getTasks() { |
| 16 | return this.taskExecutor.getPool(); | |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | @RequestMapping(method = RequestMethod.GET) | |
| 20 | public List getUpdatedTasks() { |

|  |  |  |
| --- | --- | --- |
| 21 | List updatedTasks = null; | |
| 22 |  |

|  |  |  |
| --- | --- | --- |
| 23 | // Fetch updated task until there is one or more | |
| 24 | do { |

|  |  |  |
| --- | --- | --- |
| 25 | updatedTasks = this.taskExecutor.getUpdatedTasks(); | |
| 26 | } while (updatedTasks.size() == 0); |

|  |  |
| --- | --- |
| 27 |  |
| 28 | return updatedTasks; | |

|  |  |  |
| --- | --- | --- |
| 29 | } | |
| 30 |  |

|  |  |
| --- | --- |
| 31 | @RequestMapping(method = RequestMethod.POST) |
| 32 | public void addTask(@RequestBody Task taskToAdd) { | |

|  |  |  |
| --- | --- | --- |
| 33 | this.taskExecutor.addTask(taskToAdd); | |
| 34 | } |

|  |  |
| --- | --- |
| 35 |  |
| 36 | public void startIdleTasks() throws InterruptedException { | |

|  |  |  |
| --- | --- | --- |
| 37 | this.taskExecutor.startAllTasks(); | |
| 38 | } |

|  |  |
| --- | --- |
| 39 | } |

As you can see, I did not implement the loop break condition on a maximum waiting time. You could also add a Thread.sleep() to reduce the number of calls to getUpdatedTasks() of the TaskExecutor if necessary.

## 4. Front-end implementation with AngularJS

The front-end part also changes a little bit. First, we want to separate the function that returns all Tasks and the function that returns only the updated Tasks. That last one will be a recursive function calling itself when data has arrived through the **channel** or if the server replies with an error message. Then we either push the Task received in the Array of Tasks if the status is IDLE as the TaskExecutor changes status from CREATED to IDLE before sending them to the client or we try and find the existing Task to update its status if the status is different from IDLE (either RUNNING or SUCCESS).

index.xhtml

|  |  |
| --- | --- |
| 01 | <!DOCTYPE html> |
| 02 | <html xmlns="<http://www.w3.org/1999/xhtml>" | |

|  |  |
| --- | --- |
| 03 | xmlns:h="<http://java.sun.com/jsf/html>" |
| 04 | xmlns:f="<http://java.sun.com/jsf/core>"> | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | <h:head> | |

|  |  |  |
| --- | --- | --- |
| 07 | | <title>Real-time applications - Part 1 - Java Code Geeks</title> |
| 08 | <link rel="stylesheet" href="<https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css>"/> | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 09 | <script src="<https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.5/angular.min.js>"></script> | | |
| 10 | |  |

|  |  |
| --- | --- |
| 11 | <script> |
| 12 | var part1 = angular.module("part1", []); | |

|  |  |  |
| --- | --- | --- |
| 13 | part1.controller("RealtimeCtrl", function($scope, $http, $timeout) { | |
| 14 |  |

|  |  |
| --- | --- |
| 15 | $scope.addTask = function() { |
| 16 | $http.post("api/task", $scope.task); | |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | $scope.getTasks = function() { | |
| 20 | $http.get("api/task/all") |

|  |  |  |
| --- | --- | --- |
| 21 | .success(function(data) { | |
| 22 | $scope.tasks = data; |

|  |  |  |
| --- | --- | --- |
| 23 | }); | |
| 24 | } |

|  |  |
| --- | --- |
| 25 |  |
| 26 | $scope.getUpdatedTasks = function() { | |

|  |  |
| --- | --- |
| 27 | $http.get("api/task") |
| 28 | .success(function(data) { | |

|  |  |
| --- | --- |
| 29 | data.forEach(function(currentTask) { |
| 30 | if (currentTask.status === 'IDLE') { | |

|  |  |  |
| --- | --- | --- |
| 31 | $scope.tasks.push(currentTask); | |
| 32 | } else { |

|  |  |
| --- | --- |
| 33 | $scope.tasks.forEach(function(taskToBeUpdated) { |
| 34 | if (taskToBeUpdated.name === currentTask.name) { | |

|  |  |
| --- | --- |
| 35 | taskToBeUpdated.status = currentTask.status; |
| 36 | taskToBeUpdated.running = currentTask.status === 'RUNNING'; | |

|  |  |
| --- | --- |
| 37 | } |
| 38 | }); |

|  |  |
| --- | --- |
| 39 | } |
| 40 | }); |

|  |  |
| --- | --- |
| 41 |  |
| 42 | // Recursive of Long Polling on success. | |

|  |  |  |
| --- | --- | --- |
| 43 | $scope.getUpdatedTasks(); | |
| 44 | }).error(function() { |

|  |  |  |
| --- | --- | --- |
| 45 | // Recursive of Long Polling on error. | |
| 46 | $scope.getUpdatedTasks(); |

|  |  |  |
| --- | --- | --- |
| 47 | }); | |
| 48 | } |

|  |  |
| --- | --- |
| 49 |  |
| 50 | $scope.activateRealtime = function() { | |

|  |  |  |
| --- | --- | --- |
| 51 | $scope.getUpdatedTasks(); | |
| 52 | } |

|  |  |
| --- | --- |
| 53 |  |
| 54 | $scope.getTasks(); | |

|  |  |  |
| --- | --- | --- |
| 55 | }); | |
| 56 |  |

|  |  |  |
| --- | --- | --- |
| 57 | </script> | |
| 58 | </h:head> |

|  |  |
| --- | --- |
| 59 |  |
| 60 | <h:body> | |

|  |  |  |
| --- | --- | --- |
| 61 | <div ng-app="part1" ng-controller="RealtimeCtrl" class="container"> | |
| 62 | <h1>Real-time application <SMALL>part 2</SMALL></h1> |

|  |  |  |
| --- | --- | --- |
| 63 | <h2>Add task</h2> | |
| 64 | <h:form> |

|  |  |
| --- | --- |
| 65 | <label for="durationField">Duration (in seconds):</label> |
| 66 | <input type="number" id="durationField" class="form-control" ng-model="task.duration"/> | |

|  |  |
| --- | --- |
| 67 | <button type="button" ng-click="addTask()" class="btn btn-success">Add task</button> |
| 68 | <button type="button" ng-click="getTasks()" class="btn btn-default">Refresh Tasks</button> |

|  |  |  |
| --- | --- | --- |
| 69 | <button type="button" ng-click="activateRealtime()" class="btn btn-default">Activate Auto Refresh</button> | |
| 70 | <h:commandButton actionListener="#{taskController.startTasks}" |

|  |  |  |
| --- | --- | --- |
| 71 | styleClass="btn btn-default" | |
| 72 | value="Start Idle Tasks"> |

|  |  |  |
| --- | --- | --- |
| 73 | <f:ajax execute="@form"/> | |
| 74 | </h:commandButton> |

|  |  |  |
| --- | --- | --- |
| 75 | </h:form> | |
| 76 |  |

|  |  |
| --- | --- |
| 77 | <h2>Listing</h2> |
| 78 | <ul class="list-group"> | |

|  |  |
| --- | --- |
| 79 | <li ng-repeat="curTask in tasks" class="list-group-item {{curTask.running ? 'active' : ''}}"> |
| 80 | {{curTask.name}} ({{curTask.status}})<span class="badge">{{curTask.duration}}</span> |

|  |  |  |
| --- | --- | --- |
| 81 | </li> | |
| 82 | </ul> |

|  |  |
| --- | --- |
| 83 | </div> |
| 84 | </h:body> | |

|  |  |
| --- | --- |
| 85 | </html> |

## 5. What’s next?

As you can see, it’s a little bit more complex to implement in comparison with the **periodic refresh** AJAX pattern, but we get a better feel of **real-time**. As the back-end loops and hangs the thread for a couple of seconds until it has found new data, the notification of the update seems to come from the server in **real-time**. Now, the above example is not the perfect implementation of **long polling**. It has many flaws compared to **Facebook’s** implementation, but for demonstration purposes, I think it does the job.

In the [next part of this article](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-3/), I will show you the new HTML 5 **WebSocket** and how this same application here can be improved to get the Task through a **socket** opened with the server.

# Real-time Applications with AngularJS and Java – Part 3

Posted by: [Sylvain Cloutier](http://examples.javacodegeeks.com/author/sylvain-cloutier/) in [Core Java](http://examples.javacodegeeks.com/category/core-java/) October 16th, 2015

## 1. Introduction

This article is the last one of a series of three on **real-time applications**. In [part 1](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-1/), I have created a simple Task management application and implemented the **Periodic Refresh AJAX pattern** to make it feel as if the UI was updated in real-time when tasks were created. In [part 2](http://examples.javacodegeeks.com/core-java/real-time-applications-angularjs-java-part-2/), I modified the project so only updated (delta) information was sent to the browser and I implemented **Long Polling** to get a better real-time feel. The plan now in part 3 is to go ahead and try the **HTML5 WebSocket**.

This example was built using Maven, Spring, Jackson, Gson and Tomcat 8 (JDK8) for the back-end and AngularJS, Bootstrap, HTML5 and Bower for the front-end. You should have a basic knowledge of those technologies.

## 2. WebSocket

**WebSocket** is a protocol allowing to have a **bi-directional** communication channel of a TCP connection. Most of the modern browsers and web servers have implemented the **WebSocket** protocol. After the initial handshake is done, the socket is opened and stays opened. Both parties can send data through the said socket over a channel (URL).

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/ClientServerWebSocket.png)

Figure 1: WebSocket connection between the client and the server.

In this example, we will use the previous part design consisting of a **RESTful JSON Web Service** using **Spring** @RestController and **Jackson-Mapping** from data going from the client to the server through an AJAX call using the $http service of Angular.js, and we will use WebSocket for server updates sent to the clients.

A real life example of this is on the Q&A site [StackOverflow](http://stackoverflow.com/). When you look at a specific question, a **WebSocket** is opened to notify the client in case a new answer is posted on the question. To see that in action, click on any question and hit F12. If you go to the network tab and filter on WebSocket (WS), you will see that a connection of the type WebSocket is opened and pending.

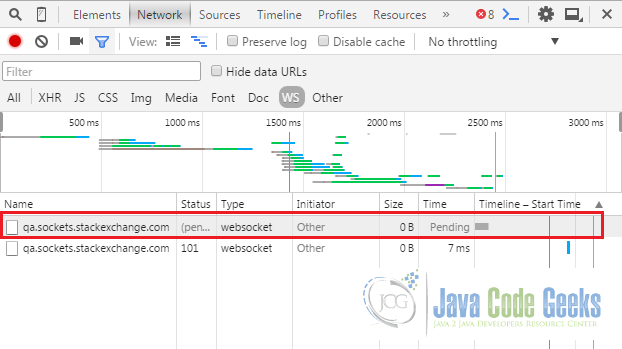
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/WebSocket_StackOverflow.png)

Figure 2: WebSocket on StackOverflow

## 3. High-level architecture

As part of this example, I decided to go to the drawing board and try to have a small and simple design to implement the bi-directional communication. My goal was to send the task objects to all clients as soon as one was added. Since there is a kind of event/reaction process, I implemented the [Observable/Observer design pattern](https://en.wikipedia.org/wiki/Observer_pattern) on the TaskRepository (observable).

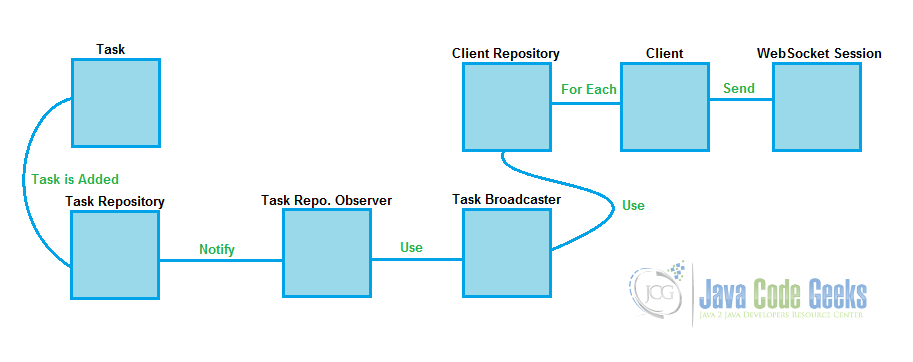
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/WebSocketDiagram.png)

Figure 3: Observable Pattern

## 4. The Front-End

### 4.1. Bower dependencies

Since I used **Bower** for dependency management, I created a bower.json file in my WebContent folder and added jQuery, Bootstrap and Angular.js to it. Here is the result:

bower.json

|  |  |
| --- | --- |
| 01 | { |
| 02 | "name": "WebContent", | |

|  |  |
| --- | --- |
| 03 | "version": "1.0.0", |
| 04 | "description": "For JCG RT App part 3", | |

|  |  |  |
| --- | --- | --- |
| 05 | "main": "index.html", | |
| 06 | "authors": [ |

|  |  |  |
| --- | --- | --- |
| 07 | "Sylvain Cloutier" | |
| 08 | ], |

|  |  |
| --- | --- |
| 09 | "license": "MIT", |
| 10 | "homepage": "<http://examples.javacodegeeks.com>", | |

|  |  |  |
| --- | --- | --- |
| 11 | "ignore": [ | |
| 12 | "\*\*/.\*", |

|  |  |
| --- | --- |
| 13 | "node\_modules", |
| 14 | "bower\_components", | |

|  |  |
| --- | --- |
| 15 | "test", |
| 16 | "tests" |

|  |  |
| --- | --- |
| 17 | ], |
| 18 | "dependencies": { | |

|  |  |
| --- | --- |
| 19 | "jQuery": "~2.1.4", |
| 20 | "bootstrap": "~3.3.5", | |

|  |  |  |
| --- | --- | --- |
| 21 | "angular": "~1.4.7" | |
| 22 | } |

|  |  |
| --- | --- |
| 23 | } |

### 4.2. Angular module and controller

First, we create the TaskApp module as follow:

taskApp.js

|  |  |
| --- | --- |
| 1 | taskApp = angular.module("TaskApp", []); |

Then the controller in a separated file. I use Angular.js **Dependency Injection** to get an instance of the $http service:

taskCtrl.js

|  |  |  |
| --- | --- | --- |
| 01 | taskApp.controller("TaskController", function($http, $scope) { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | $scope.getTasks = function() { |
| 04 | $http.get("<http://localhost:8080/WebSocket/tasks>") | |

|  |  |  |
| --- | --- | --- |
| 05 | .success(function(data) { | |
| 06 | $scope.tasks = data; |

|  |  |  |
| --- | --- | --- |
| 07 | }); | |
| 08 | }; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | $scope.addTask = function() { | |

|  |  |  |
| --- | --- | --- |
| 11 | $http.post("<http://localhost:8080/WebSocket/tasks>", $scope.task) | |
| 12 | .success(function() { |

|  |  |  |
| --- | --- | --- |
| 13 | $scope.resetTask(); | |
| 14 | }); |

|  |  |  |
| --- | --- | --- |
| 15 | }; | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | $scope.resetTask = function() { | |
| 18 | $scope.task = { |

|  |  |
| --- | --- |
| 19 | title: "", |
| 20 | description: "", | |

|  |  |
| --- | --- |
| 21 | duration: "", |
| 22 | universal: true | |

|  |  |  |
| --- | --- | --- |
| 23 | }; | |
| 24 | }; |

|  |  |
| --- | --- |
| 25 |  |
| 26 | $scope.resetTask(); | |

|  |  |  |
| --- | --- | --- |
| 27 | $scope.getTasks(); | |
| 28 |  |

|  |  |
| --- | --- |
| 29 | // WebSocket Initialization |
| 30 | var taskSocket = new WebSocket("<ws://localhost:8080/WebSocket/channel/task>"); | |

|  |  |
| --- | --- |
| 31 |  |
| 32 | taskSocket.onmessage = function(message) { | |

|  |  |  |
| --- | --- | --- |
| 33 | $scope.tasks = JSON.parse(message.data); | |
| 34 | $scope.$apply(); |

|  |  |  |
| --- | --- | --- |
| 35 | }; | |
| 36 |  |

|  |  |  |
| --- | --- | --- |
| 37 | taskSocket.onclose = function() { | |
| 38 | $scope.message = { |

|  |  |
| --- | --- |
| 39 | type: "danger", |
| 40 | short: "Socket error", | |

|  |  |  |
| --- | --- | --- |
| 41 | long: "An error occured with the WebSocket." | |
| 42 | }; |

|  |  |  |
| --- | --- | --- |
| 43 | $scope.$apply(); | |
| 44 | } |

|  |  |  |
| --- | --- | --- |
| 45 |  | |
| 46 | }); |

Again, I’ve not been following Angular.js best practices, because I wanted to centralize the code for this example. Note that the part where we establish the **WebSocket** connection and where we define the methods could have been extracted from the controller and included in a TaskService or such.

At **line 30** of the controller file above, we initialize the **WebSocket** connection. Notice the protocol identifier ws://. This URL will be defined in the back-end as a **server endpoint**.

### 4.3. The view

Unlike part 1 and 2, I decided not to include JSF in the project. I wanted to have a pure HTML and JavaScript front-end so Angular is the only thing interacting with the back-end. This shows the real benefit of Angular.js that is to make view code more like HTML is supposed to look like, that is **declarative**.

That said, the view for this example contains 2 blocs. The first one is the form allowing to create new Task objects through an XHR request by using Angular $http service, and the second bloc is the displaying of the Task objects. At load time, the controller will fetch the list of Task objects from the server and Angular will populate the list. Then a **WebSocket** is initialized so when a new Task object is created in the TaskRepository, it will be added in the list.

index.html

|  |  |  |
| --- | --- | --- |
| 001 | <!DOCTYPE html> | |
| 002 | <html> |

|  |  |
| --- | --- |
| 003 | <head> |
| 004 | <title></title> | |

|  |  |
| --- | --- |
| 005 | <script src="js/angular/angular.min.js"></script> |
| 006 | <script src="js/jQuery/dist/jquery.min.js"></script> | |

|  |  |  |
| --- | --- | --- |
| 007 | <script src="js/bootstrap/dist/js/bootstrap.min.js"></script> | |
| 008 |  |

|  |  |
| --- | --- |
| 009 | <link rel="stylesheet" type="text/css" |
| 010 | href="js/bootstrap/dist/css/bootstrap.min.css" /> | |

|  |  |
| --- | --- |
| 011 | <script src="js/taskApp/taskApp.js"></script> |
| 012 | <script src="js/taskApp/taskCtrl.js"></script> | |

|  |  |
| --- | --- |
| 013 |  |
| 014 | </head> | |

|  |  |
| --- | --- |
| 015 | <body> |
| 016 | <div class="container" ng-app="TaskApp" ng-controller="TaskController"> | |

|  |  |  |
| --- | --- | --- |
| 017 | <h1>Real-time application <small>part 3</small></h1> | |
| 018 |  |

|  |  |  |
| --- | --- | --- |
| 019 | <div ng-show="message" class="alert alert-{{message.type}}"> | |
| 020 | <strong> |

|  |  |  |
| --- | --- | --- |
| 021 | {{message.short}}: {{message.long}} | |
| 022 | </strong> |

|  |  |
| --- | --- |
| 023 | </div> |
| 024 | <div class="container-fluid"> | |

|  |  |  |
| --- | --- | --- |
| 025 | <div class="col-md-6 col-sm-12"> | |
| 026 | <h2>Add task</h2> |

|  |  |
| --- | --- |
| 027 | <form> |
| 028 | <div class="row"> | |

|  |  |
| --- | --- |
| 029 | <div class="form-group"> |
| 030 | <label for="taskTitleFieldId">Title: </label> | |

|  |  |  |
| --- | --- | --- |
| 031 | <input type="text" id="taskTitleFieldId" | |
| 032 | ng-model="task.title" |

|  |  |  |
| --- | --- | --- |
| 033 | class="form-control"/> | |
| 034 | </div> |

|  |  |  |
| --- | --- | --- |
| 035 | </div> | |
| 036 |  |

|  |  |
| --- | --- |
| 037 | <div class="row"> |
| 038 | <div class="form-group"> | |

|  |  |  |
| --- | --- | --- |
| 039 | <label for="taskDescriptionFieldId"> | |
| 040 | Description: |

|  |  |
| --- | --- |
| 041 | </label> |
| 042 | <textarea id="taskDescriptionFieldId" | |

|  |  |  |
| --- | --- | --- |
| 043 | ng-model="task.description" | |
| 044 | class="form-control"> |

|  |  |  |
| --- | --- | --- |
| 045 | </textarea> | |
| 046 | </div> |

|  |  |  |
| --- | --- | --- |
| 047 | </div> | |
| 048 |  |

|  |  |
| --- | --- |
| 049 | <div class="row"> |
| 050 | <div class="form-group"> | |

|  |  |  |
| --- | --- | --- |
| 051 | <label for="durationFieldId"> | |
| 052 | Duration (in seconds): |

|  |  |
| --- | --- |
| 053 | </label> |
| 054 | <input type="number" id="durationFieldId" | |

|  |  |
| --- | --- |
| 055 | class="form-control" |
| 056 | ng-model="task.duration"/> | |

|  |  |  |
| --- | --- | --- |
| 057 | </div> | |
| 058 | </div> |

|  |  |
| --- | --- |
| 059 |  |
| 060 | <div class="row"> | |

|  |  |
| --- | --- |
| 061 | <div class="col-md-6"> |
| 062 | <div class="checkbox"> | |

|  |  |
| --- | --- |
| 063 | <label> |
| 064 | <input type="checkbox" | |

|  |  |
| --- | --- |
| 065 | id="taskUniversalCheckId" |
| 066 | ng-model="task.universal"/> | |

|  |  |  |
| --- | --- | --- |
| 067 | Public task | |
| 068 | </label> |

|  |  |  |
| --- | --- | --- |
| 069 | </div> | |
| 070 | </div> |

|  |  |
| --- | --- |
| 071 |  |
| 072 | <div class="col-md-6"> | |

|  |  |
| --- | --- |
| 073 | <button type="button" |
| 074 | class="btn btn-success" | |

|  |  |  |
| --- | --- | --- |
| 075 | ng-click="addTask()"> | |
| 076 | Add task |

|  |  |
| --- | --- |
| 077 | </button> |
| 078 | <button type="button" | |

|  |  |
| --- | --- |
| 079 | class="btn btn-default" |
| 080 | ng-click="resetTask()"> |

|  |  |  |
| --- | --- | --- |
| 081 | Reset form | |
| 082 | </button> |

|  |  |  |
| --- | --- | --- |
| 083 | </div> | |
| 084 | </div> |

|  |  |  |
| --- | --- | --- |
| 085 | </form> | |
| 086 | </div> |

|  |  |
| --- | --- |
| 087 |  |
| 088 | <div class="col-md-6 col-sm-12"> | |

|  |  |
| --- | --- |
| 089 | <h2>Listing</h2> |
| 090 | <ul class="list-group" ng-hide="tasks.length == 0"> | |

|  |  |  |
| --- | --- | --- |
| 091 | <li ng-repeat="curTask in tasks track by $index" | |
| 092 | class="list-group-item"> |

|  |  |
| --- | --- |
| 093 | <strong>{{curTask.title}}</strong> - {{curTask.description}} |
| 094 | <span class="badge">{{curTask.duration}}</span> |

|  |  |  |
| --- | --- | --- |
| 095 | </li> | |
| 096 | </ul> |

|  |  |  |
| --- | --- | --- |
| 097 | <p ng-show="tasks.length == 0" class="text-info"> | |
| 098 | No tasks to display. |

|  |  |  |
| --- | --- | --- |
| 099 | </p> | |
| 100 | </div> |

|  |  |  |
| --- | --- | --- |
| 101 | </div> | |
| 102 | </div> |

|  |  |  |
| --- | --- | --- |
| 103 | </body> | |
| 104 | </html> |

## 5. The Java Back-end

### 5.1. Maven dependencies

As part of this example, we need Spring-WebMVC, Spring-WebSocket, Google Gson and Jackson-DataBind. Here is the result:

pom.xml

|  |  |
| --- | --- |
| 01 | <dependencies> |
| 02 | <dependency> |

|  |  |
| --- | --- |
| 03 | <groupId>org.springframework</groupId> |
| 04 | <artifactId>spring-webmvc</artifactId> |

|  |  |  |
| --- | --- | --- |
| 05 | <version>4.2.1.RELEASE</version> | |
| 06 | </dependency> |

|  |  |
| --- | --- |
| 07 | <dependency> |
| 08 | <groupId>org.springframework</groupId> | |

|  |  |  |
| --- | --- | --- |
| 09 | <artifactId>spring-websocket</artifactId> | |
| 10 | <version>4.2.1.RELEASE</version> |

|  |  |  |
| --- | --- | --- |
| 11 | </dependency> | |
| 12 | <dependency> |

|  |  |  |
| --- | --- | --- |
| 13 | <groupId>com.google.code.gson</groupId> | |
| 14 | <artifactId>gson</artifactId> |

|  |  |  |
| --- | --- | --- |
| 15 | <version>2.3.1</version> | |
| 16 | </dependency> |

|  |  |
| --- | --- |
| 17 | <dependency> |
| 18 | <groupId>com.fasterxml.jackson.core</groupId> | |

|  |  |  |
| --- | --- | --- |
| 19 | <artifactId>jackson-databind</artifactId> | |
| 20 | <version>2.6.2</version> |

|  |  |
| --- | --- |
| 21 | </dependency> |
| 22 | </dependencies> |

### 5.2. The TaskWebSocketHandler

This class is the **Server Endpoint**. The implementation of the **WebSocket** API is provided by Tomcat since version 7. The way to deploy the **WebSocket** was made really simple. All you need to do is to add the @ServerEndpoint annotation to your class and define the URL on which the **WebSocket** will be listening.

TaskWebSocketHandler.java

|  |  |  |
| --- | --- | --- |
| 01 | @ServerEndpoint(value = "/channel/task", configurator = SpringConfigurator.class) | |
| 02 | public class TaskWebSocketHandler implements WebSocketHandler { |

|  |  |
| --- | --- |
| 03 | @Autowired |
| 04 | private ClientRepository clientRepository; | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | @OnOpen | |

|  |  |
| --- | --- |
| 07 | public void onOpen(Session session) { |
| 08 | this.clientRepository.add(new Client(session)); | |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | @OnClose |
| 12 | public void onClose(CloseReason reason, Session session) { | |

|  |  |  |
| --- | --- | --- |
| 13 | this.clientRepository.remove(new Client(session)); | |
| 14 | } |

|  |  |
| --- | --- |
| 15 | } |

I would like to bring your attention to the configurator property of the annotation. Since the handler is managed by the container, it was not instantiated by Spring so I was not able to inject the ClientRepository dependency. By adding the SpringConfigurator, **Spring** takes control of the initialization and performs the injection. That way, I can use the @Autowired annotation to inject the ClientRepository.

In order to keep track of the active sessions, I added two event functions that are triggered when the connection is established or closed. Those are annotated with @OnOpen and @OnClose.

### 5.3. The ClientRepository

Like I said above, in order to send the Task objects to the client, we need to keep track of the active sessions. This was implemented in the ClientRepository. I decided to wrap the session in a Client object just in case we need to add information regarding the client (name, avatar, etc.).

Client.java

|  |  |
| --- | --- |
| 01 | public class Client { |
| 02 | private final String id; | |

|  |  |  |
| --- | --- | --- |
| 03 | private final Session session; | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | public Client(Session session) { | |
| 06 | this.id = this.toString(); |

|  |  |  |
| --- | --- | --- |
| 07 | this.session = session; | |
| 08 | } |

|  |  |
| --- | --- |
| 09 |  |
| 10 | public void sendText(String text) throws IOException { | |

|  |  |  |
| --- | --- | --- |
| 11 | this.session.getBasicRemote().sendText(text); | |
| 12 | } |

|  |  |
| --- | --- |
| 13 |  |
| 14 | public String getId() { | |

|  |  |  |
| --- | --- | --- |
| 15 | return id; | |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |
| 18 | // hashCode() and equals() | |

|  |  |  |
| --- | --- | --- |
| 19 |  | |
| 20 | } |

The sendText() method here simply wraps the call to the **WebSocket** session sendText() method. That helps customizing the way we communicate through the **WebSocket** if necessary. Regarding the repository itself, I simply created a class containing a List of Client. Notice that I added synchronization on the List to avoid ConcurrentModificationException.

ClientRepositoryImpl.java

|  |  |
| --- | --- |
| 01 | @Repository |
| 02 | @Scope("singleton") | |

|  |  |  |
| --- | --- | --- |
| 03 | public class ClientRepositoryImpl extends ClientRepository { | |
| 04 | private List<Client> clients = new LinkedList<>(); |

|  |  |
| --- | --- |
| 05 |  |
| 06 | @Override | |

|  |  |  |
| --- | --- | --- |
| 07 | public void add(Client session) { | |
| 08 | synchronized (this.clients) { |

|  |  |  |
| --- | --- | --- |
| 09 | this.clients.add(session); | |
| 10 | } |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 |  |

|  |  |
| --- | --- |
| 13 | @Override |
| 14 | public void remove(Client session) { | |

|  |  |
| --- | --- |
| 15 | synchronized (this.clients) { |
| 16 | this.clients.remove(session); | |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |
| 20 | @Override | |

|  |  |  |
| --- | --- | --- |
| 21 | public void forEach(Consumer<Client> clientConsume) { | |
| 22 | synchronized (this.clients) { |

|  |  |  |
| --- | --- | --- |
| 23 | this.clients.forEach(clientConsume); | |
| 24 | } |

|  |  |  |
| --- | --- | --- |
| 25 | } | |
| 26 |  |

|  |  |
| --- | --- |
| 27 | @Override |
| 28 | public List<Client> getAll() { | |

|  |  |  |
| --- | --- | --- |
| 29 | return new LinkedList<>(this.clients); | |
| 30 | } |

|  |  |
| --- | --- |
| 31 |  |
| 32 | } | |

### 5.4. The TaskRepository

The Task object is a little bit different from the last two parts of this example. In part 2, I pointed that Task where not owned by a single client, they were shared. I added here the Client that is the owner of the Task and a flag called universal that would make the Task visible to all clients. For simplicity’s sake, I did not implement that, but I wanted to show you a way to achieve that in case you want to modify the example to make it a bit better and experiment with it.

Task.java

|  |  |
| --- | --- |
| 01 | public class Task { |
| 02 | private String id; | |

|  |  |
| --- | --- |
| 03 | private Client owner; |
| 04 | private String title; |

|  |  |  |
| --- | --- | --- |
| 05 | private String description; | |
| 06 | private long duration; |

|  |  |  |
| --- | --- | --- |
| 07 | private boolean universal; | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | public Task() { |
| 10 | this.id = this.toString(); | |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | // Getters and setters | |
| 14 |  |

|  |  |
| --- | --- |
| 15 | } |

Regarding the repositories, I created an **abstract** and **generic** version of them that extends the Observable class from the Java API. The TaskRepository extends this abstract class so we can connect any repositories with an Observer.

TaskAppRepository.java

|  |  |  |
| --- | --- | --- |
| 01 | public abstract class TaskAppRepository<K, T> extends Observable { | |
| 02 | public abstract void add(T type); |

|  |  |
| --- | --- |
| 03 | public abstract void remove(T type); |
| 04 | public abstract void forEach(Consumer<T> typeConsumer); | |

|  |  |  |
| --- | --- | --- |
| 05 | public abstract List<T> getAll(); | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | protected void publish() { | |
| 08 | this.setChanged(); |

|  |  |  |
| --- | --- | --- |
| 09 | this.notifyObservers(); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 | } |

The main feature here is the publish() method that calls the setChanged() and notifyObservers() methods of the Observable class. This calls the update() method of the registered Observer.

Regarding the TaskRepository itself, I decided that the simplest way to register the Observer was to inject it and add it to the observers in the initialization method. The rest is pretty much like the ClientRepository.

TaskRepositoryImpl.java

|  |  |
| --- | --- |
| 01 | @Repository |
| 02 | @Scope("singleton") | |

|  |  |  |
| --- | --- | --- |
| 03 | public class TaskRepositoryImpl extends TaskRepository { | |
| 04 | @Autowired |

|  |  |  |
| --- | --- | --- |
| 05 | private TaskRepositoryObserver observer; | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | private List<Task> tasks = new LinkedList<>(); | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | @PostConstruct |
| 10 | public void init() { | |

|  |  |  |
| --- | --- | --- |
| 11 | this.addObserver(observer); | |
| 12 | } |

|  |  |
| --- | --- |
| 13 |  |
| 14 | @Override | |

|  |  |  |
| --- | --- | --- |
| 15 | public void add(Task task) { | |
| 16 | synchronized (tasks) { |

|  |  |  |
| --- | --- | --- |
| 17 | this.tasks.add(task); | |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |
| 20 | this.publish(); | |

|  |  |  |
| --- | --- | --- |
| 21 | } | |
| 22 |  |

|  |  |
| --- | --- |
| 23 | @Override |
| 24 | public void remove(Task task) { | |

|  |  |
| --- | --- |
| 25 | synchronized (tasks) { |
| 26 | this.tasks.remove(task); | |

|  |  |  |
| --- | --- | --- |
| 27 | } | |
| 28 |  |

|  |  |  |
| --- | --- | --- |
| 29 | this.publish(); | |
| 30 | } |

|  |  |
| --- | --- |
| 31 |  |
| 32 | @Override | |

|  |  |  |
| --- | --- | --- |
| 33 | public void forEach(Consumer<Task> typeConsumer) { | |
| 34 | synchronized (tasks) { |

|  |  |  |
| --- | --- | --- |
| 35 | this.tasks.forEach(typeConsumer); | |
| 36 | } |

|  |  |  |
| --- | --- | --- |
| 37 | } | |
| 38 |  |

|  |  |
| --- | --- |
| 39 | public List<Task> getAll() { |
| 40 | return new LinkedList<>(this.tasks); | |

|  |  |  |
| --- | --- | --- |
| 41 | } | |
| 42 | } |

You see now that everything is linked up. When the TaskRepository is initialized, the Observer is registered in the init() method. When a Task is added or removed from the List, the publish() method is called which sets the state of the TaskRepository as changed and it notifies the Observer that will retrieve the list of Client from the ClientRepository and send the Task objects through the **WebSocket** **session** directly to the browser. The onmessage() method of the Angular.js TaskController will be fired and the view will be updated!

TaskRepositoryObserver.java

|  |  |
| --- | --- |
| 01 | @Component |
| 02 | public class TaskRepositoryObserver implements Observer { | |

|  |  |
| --- | --- |
| 03 | @Autowired |
| 04 | private TaskBroadcaster broadcaster; | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | @Override | |

|  |  |  |
| --- | --- | --- |
| 07 | public void update(Observable repository, Object param) { | |
| 08 | TaskRepository repo = (TaskRepository) repository; |

|  |  |  |
| --- | --- | --- |
| 09 | this.broadcaster.broadcast(repo.getAll()); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 |  |
| 12 | } | |

TaskBroadcaster.java

|  |  |
| --- | --- |
| 01 | @Component |
| 02 | @Component |

|  |  |  |
| --- | --- | --- |
| 03 | public class TaskBroadcaster implements Broadcaster<Task> { | |
| 04 | @Autowired |

|  |  |  |
| --- | --- | --- |
| 05 | private ClientRepository clients; | |
| 06 | private Gson gson; |

|  |  |
| --- | --- |
| 07 |  |
| 08 | @PostConstruct | |

|  |  |
| --- | --- |
| 09 | public void init() { |
| 10 | this.gson = new Gson(); | |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 |  |

|  |  |
| --- | --- |
| 13 | @Override |
| 14 | public void broadcast(List<Task> task) { | |

|  |  |  |
| --- | --- | --- |
| 15 | this.clients.forEach(client -> { | |
| 16 | try { |

|  |  |  |
| --- | --- | --- |
| 17 | client.sendText(this.gson.toJson(task)); | |
| 18 | } catch (Exception e) { |

|  |  |  |
| --- | --- | --- |
| 19 | e.printStackTrace(); | |
| 20 | } |

|  |  |  |
| --- | --- | --- |
| 21 | }); | |
| 22 | } |

|  |  |
| --- | --- |
| 23 | } |

Here, the best thing would have been to implement a **converter** (Task to String) before calling the sendText() method. That way, we could have more that one converters and change it at will, but for this example, it suites the needs.

Regarding the **web service**, it’s basically the same as in the previous parts. We use **Spring** annotation @RestController.

TaskRestController.java

|  |  |
| --- | --- |
| 01 | @RestController |
| 02 | public class TaskRestController { | |

|  |  |
| --- | --- |
| 03 | @Autowired |
| 04 | private TaskRepository taskRepository; | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | @RequestMapping(path = "/tasks", method = RequestMethod.GET) | |

|  |  |
| --- | --- |
| 07 | public @ResponseBody List getTasks() { |
| 08 | return this.taskRepository.getAll(); | |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | @RequestMapping(path = "/tasks", method = RequestMethod.POST) | |
| 12 | public void addTask(@RequestBody Task task) { |

|  |  |  |
| --- | --- | --- |
| 13 | this.taskRepository.add(task); | |
| 14 | } |

|  |  |
| --- | --- |
| 15 | } |

## 6. Conclusion

This concludes the final part on **real-time applications**. You must know that I made personal choices regarding the implementation and the technologies I used. Those choices were made based on my personal experience and on my desire to **challenge** myself by integrating new technologies all together. There are infinite ways of doing this, some might be better, but I think you get the point and understand how to achieve this real-time feel in an application that makes it less static.

An example of other tools you could use is a module for **WebSocket** called [Angular-WebSocket](https://www.npmjs.com/package/angular-websocket) that can be used instead of the native **WebSocket**. [PrimeFaces](http://www.primefaces.org/showcase/push/chat.xhtml) also offers an implementation that is good (PrimeFaces Push) and that is based on the **Atmosphere Framework**. This one would be a good alternative for production applications in case your users don’t have a browser that supports **WebSocket**. It has a fallback to **Long Polling**.

Now you tell me! What do you think of this series of articles? Send me a tweet with your comments, your suggestions or even your questions. My Twitter handle is [@syl20TOS](https://twitter.com/syl20TOS). I will try to answer every Tweet.

Thanks for reading!