Modern java webapp

The CodeStory Way

by David Gageot & Jean-Laurent de Morlhon

Abstract

Come participate to this 3 hours Hand's On lab. Our target: teach you to program a modern webapp in Java (yes Java!), quickly, pragmatically and with ease.

With the help and live code demos of David Gageot & Jean-Laurent de Morlhon.

On the menu: Java 8, some AngularJs, a taste of CoffeeScript, Pair Programming, UI tests, hotkeys you didn't knew existed, plugins from outer space and an ultra fast development cycle. Yes we're still talking about Java.

Monday, at work, you won't see your java project the same way.

Install party

To attend this workshop in the best possible conditions:

- A laptop with enough power for 3 hours
- A teammate
- Software:
  - Java 8
  - Maven 3.1
  - An IDE
  - A few graphical assets you'll find in this repo or on a usb key, network drive we will share during the session.

Recruteur.io

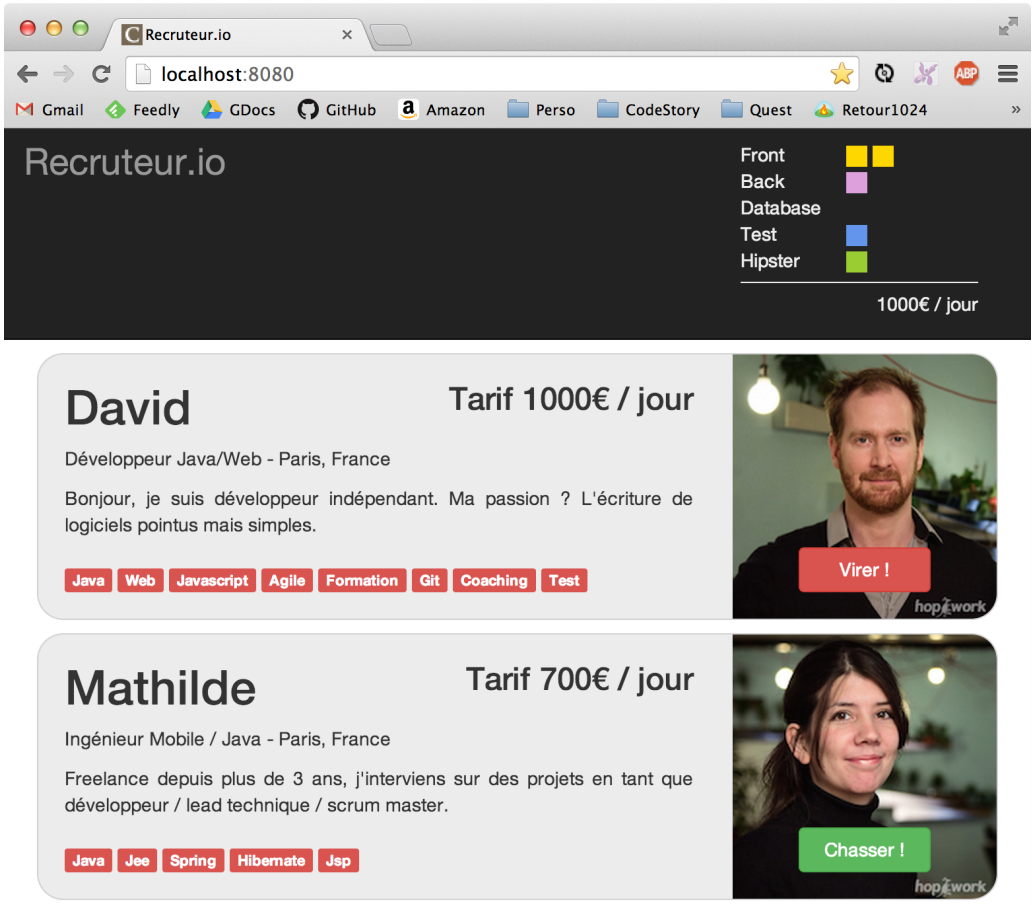
You friend Jean-Claude from a famous second-zone commerce school in the countryside has a tremendous business idea:

We're going to make a website to find programmers, and make a ton of money placing them in businesses across the world to help write web projects.

Jean-Claude has heard that we probably need different skill sets to build a great team. You need to mix these skills.

In 2014, Jean-Claude has decided you need 4 different skills: `Front`, `Back`, `Database`, `Test`. And to sounds more appealing and also because it sells well, he added a fifth skill named `Hipster`.

The website could look like this:



Martine from HR has already bought the domain name on godady, you're free to go. We have installed FrontPage and IIS on your laptop, you've got 2 hours!

Let's write some code

Server startup

1. Create a blank directory, in which you add a pom.xml which looks like:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4
  <modelVersion>4.0.0</modelVersion>

  <groupId>net.code-story</groupId>
  <artifactId>recruteurio</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>net.code-story</groupId>
      <artifactId>http</artifactId>
      <version>2.20</version>
    </dependency>
  </dependencies>
</project>
```

So, yes, we're going to use java 8, we have waited for it for too long, not to use it right away. Fasten your seat belts.

1. Then you create, like a grownup, the sources & tests directories (yeah, we got tests too, I know so modern ...).

```
mkdir -p src/{main,test}/java
```

(btw you can create them with your mouse, but it's less hype and stylish. Modern web remember?)

1. We are here to make a webapp. But we are going to be classical for a change and start with a good old 'Hello World'.

   You should create an `index.html` at the root of an `app` directory beside your `pom.xml` like this:

```
mkdir app
touch app/index.html
```

Then edit your `index.html` and type in:

```
---
layout: default
title: Hello Devoxx
---

<h1>Hello Devoxx</h1>

<p>I can serve a web page in a java app in less than 2 minutes... Yes, I can!</p>
```

Before you ask, the header in between dashes is called Yaml Front Matter. You can enter a bunch of information using Yaml syntax there and everything after the last `---` is going to be plain old HTML. You can do crazy stuff in here, if you're nice you'll see a glimpse of it, but we won't go into more details there, but trust us, it's quite convenient.

In fluent-http, everything you put in the `app` directory is served at the root of your webapp. If you put some html, it will be serve, as-is. Same for js files, images etc...

If you put some Less files, they will be compiled to css and served (with a cache don't worry), the same applies to Coffeescript compiled to Javascript, Markdown to Html and a few others.

1. Ok it's a java workshop or what? When will I write some Java Code?!:

   Just about now : In `src/main/java` create a `Server` class. Like this one:

```
import net.codestory.http.*;

public class Server {
  public static void main(String[] args) {
    new WebServer().start();
  }
}
```

1. Then you execute the `Server` class, open a browser and aim it towards http://localhost:8080 If everything goes according to plan, just about now, you'll feel less inclined to use weblo or tomcat, monday at work. I started a java program that serves content in 1 line of code and 5 minutes...

(If you're on the fancy side of stuff, and that you change your working dir, I know *crazy*, but some of you do it, you'll have to point your working dir to the root of your app. It's usually done in the working dir input field in the run class dialog of your IDE)

Server Side Mustaches with Handlebars

1. Fluent-http provides some kind of server side, logic less, templating.

Change your server to the following, we add a route to '/' which defines a conference variable to be

used in your template in a Java8-lambdaish way.

```
public class Server {
  public static void main(String[] args) {
    new WebServer(routes -> routes.get("/", () -> Model.of("conference", "Devoxx"))).start()
  }
}
```

Change your index.html to :

```
---
layout: default
title: hello mix-it
---

<h1>Hello [[conference]]!</h1>
```

The templating language used here is Handlebars. You can use every handlebar instructions but within `[[` and `]]` instead the usual `{{` and `}}`. As you may know, we are going to use some angular code in a few minutes, so we changed the way handlebars detect it's tag so it doesn't clash with angular. You can then use a mix of server side and client side content.

Woot! Some Handlebars and some java 8 lambda at the same time. Everything is rendered server side. Consider it the jsp of 2014.

Handlebars supports Loops

In the app, Jean-Clause wants us to display a bunch of developers, so we need a way to iterate around a list of developers :

```
public class Server {
  public static void main(String[] args) {
    new WebServer(routes -> routes.get("/", () -> Model.of("developers", Arrays.asList("Dav
  }
}
```

Display the loop content like this:

```
[[#each developers]]
  [[.]]
[[/each]]
```

You can use Java Beans, Pojos, Java Objects, you name it.

But developers aren't define only by their names, (we tend to say the define themselves by the number of bugs they produces but that's another story). Developers needs properties let's start simply with name and price.

```
ublic class Developer {
 String name;
 int price;

 public Developer(String name, int price) {
   this.name = name;
   this.price = price;
 }


ublic class Server {
 public static void main(String[] args) {
   new WebServer(routes -> routes.get("/", () -> Model.of("developers", Arrays.asList(new Dev
 }
```

Then you can display each developer's fields.

```
[[#each developers]]
  [[name]] [[price]]
[[/each]]
```

You can do many more things in Handlebars, but keep in mind it's call logic less for a reason, you can see more at: http://handlebarsjs.com/.

Tests

We are going to extract a `Configuration` object to make it usable for tests. Like this:

```
public class Server {
  public static void main(String[] args) {
    new WebServer(new ServerConfiguration()).start();
  }

  public static class ServerConfiguration implements Configuration {
    @Override
    public void configure(Routes routes) {
      routes.get("/", () -> Model.of("developers", Arrays.asList(new Developer("David", 100
    }
  }
}
```

Let's write an end to end test, also called sometimes acceptance test, UI test or **test-which-brake-too-often-but-are-really-really-life-saver(tm)**.

So take an hour, setup selenium, install all drivers. Just kidding! We do everything for you, with our hand-cooked selenium wrapper called SimpleLenium

```
<dependency>
  <groupId>net.code-story</groupId>
  <artifactId>simplelenium</artifactId>
  <version>1.25</version>
  <scope>test</scope>
</dependency>
```

```
import net.codestory.http.WebServer;
import net.codestory.simplelenium.SeleniumTest;
import org.junit.Test;

import static net.codestory.Server.ServerConfiguration;

public class BasketSeleniumTest extends SeleniumTest {
  WebServer webServer = new WebServer(new ServerConfiguration()).startOnRandomPort();

  @Override
  public String getDefaultBaseUrl() {
    return "http://localhost:" + webServer.port();
  }

  @Test
  public void list_developers() {
    goTo("/");

    find(".developer").should().haveSize(2);
    find(".developer").should().contain("David", "Jean-Laurent");
  }
}
```

Don't need to install Chrome, Selenium, PhantomJS or what. It just works.

To avoid port conflicts (two server asking for the same port) with test running in parallel, fluent-http gives you a `startOnRandomPort()` method which makes sure to avoid conflicts.

Simple REST Service

Routes can be written with Java 8 Lambdas but for more complex routes, it's best to extract the route's code into a `Resource` class. And because fluent-http is built from the ground up to be a web container, every time it sees a Java Bean or Pojo in a resource method signature it exposes it as `json` by default.

For instance if your route needs to server a `Basket`, it could be defined like this :

```
public class Basket {
  long front;
  long back;
  long database;
  long test;
  long hipster;
  long sum;
}
```

You can easily add a resource to you http server like this:

```
public class BasketResource {
  @Get("/basket")
  public Basket basket() {
    return new Basket();
  }
}
```

Then you add it to your routes:

```
public class ServerConfiguration implements Configuration {
    @Override
    public void configure(Routes routes) {
      routes.add(BasketResource.class);
    }
}
```

And when you call http://localhost:8080/basket you get something like:

```
{
  "front":0,
  "back":0,
  ...
  "sum":0,
}
```

Now that we created our first resource, let's extract another resource for the index page.

```
import net.codestory.http.annotations.Get;
import net.codestory.http.templating.Model;

public class IndexResource {
  @Get("/")
  public Model index() {
    return Model.of("developers", Arrays.asList(new Developer("David", 1000), new Develope
  }
}
```

And plug it this way:

```
public class ServerConfiguration implements Configuration {
    @Override
    public void configure(Routes routes) {
      routes
        .add(IndexResource.class)
```

```
        .add(BasketResource.class);
    }
}
```

Integration testing with RestAssured

Integration tests at the resource level are interesting because it's the only way to check that our domain code is properly wrapped into a REST resource. You should concentrate on testing on http input/output. While mocking/stubbing the domain code.

We use the RestAssured library which offers a fluent API to write tests. Testing the http interaction layer is quite tedious to write.

Add to your pom the dependency :

```
<dependency>
  <groupId>com.jayway.restassured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>2.3.4</version>
  <scope>test</scope>
</dependency>
```

RestAssured needs a real http server. This is usually done in the integration testing phase through the failsafe maven plugin. But we are crazy modern guys, we don't want to distinguish those tests since we are able to execute integration test at almost the same speed as unit tests.

To be able to have integration test execute as fast as unit test, you need a lighting fast webserver, that's why we use fluent-http. It's very good at it. Less configuration, lighting speed, you saved yourself at many hours writing xml in your project. You're welcome...

Here's a typical skeleton for a REST test:

```
public class BasketRestTest {
  WebServer webServer = new WebServer(new ServerConfiguration()).startOnRandomPort();

  @Test
  public void query_basket() {
    given().port(webServer.port())
      .when().get("/basket")
      .then().contentType("application/json").statusCode(200);
  }
}

## AngularJs

To add angularjs the java-way you can use Webjars. Webjars are a collection of javascript lib

```xml
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>angularjs</artifactId>
  <version>1.3.0</version>
</dependency>
```

You'll use the `/webjars/angularjs/1.3.0/angular.min.js` path in a `<script>` tag. You can do the same thing with all your front-end dependencies: javascript libraries, css styles, fonts, icons etc...

If you think embedding a javascript library into a zip file, renamed to .jar is kind of completely mad, you can use bower. It's a bit more hype, but you have to move the file by hand from the bower directory to your `app` directory.

Use coffeescript

You can write your angular controllers in coffeescript using a class syntax. This enable to properly
```

and easily isolate the scope variables. For instance :

```
angular.module 'devoxx', []

.controller 'BasketController', class
    constructor: (@$http) ->
      @info = "Hello World"
      @basket = {}

    search: ->
      @$http.get("/basket").success (data) =>
        @basket = data
```

To make it work you can add the ng-app tag by hand, or put it in the YAML front matter. don't forget to add the angular lib script tag.

```
---
title: Hello Devoxx
ng-app: devoxx
---
<div ng-controller="BasketController as controller">
  {{controller.info}}
  <a href="" ng-click="controller.search()">search</a>
</div>

<script src="/webjars/angularjs/1.3.0/angular.min.js"></script>
```

Unit, integration, javascript & ui Testing!

Resource Unit Testing with JUnit

Nothing, *that* modern in here.

We use the usual suspects of the industry here, `AssertJ` (fluent assertions) & `Mockito` (mocking).

You can add those two libraries like this in your pom:

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>1.7.0</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>1.10.8</version>
  <scope>test</scope>
</dependency>
```

Let's try starting to write the resources we would need to do our app for Jean-Claude.

We need some kind of developer domain object:

```
public class Developer {
  public String prenom;
  public String job;
  public String ville;
  public String photo;
  public String description;
  public String email;
  public String[] tags;
  public int price;
```

```
}
```

We need some kind of developers list:

```
[
  {
    "email": "david@devoxx.io",
    "prenom": "David",
    "job": "Java/Web Developer",
    "ville": "Paris, France",
    "description": "Bonjour, je suis développeur indépendant. Ma passion ? L'écriture de logici
    "tags": [
      "Java", "Web", "Javascript", "Agile", "Formation", "Git", "Coaching", "Test"
    ],
    "photo": "david",
    "price": 1000
  },
  {
    "email": "jeanlaurent@devoxx.io",
    "prenom": "Jean-Laurent",
    "job": "Programmer",
    "ville": "Houilles, France",
    "description": "WILL WRITE CODE FOR FOOD",
    "tags": [
      "Java", "Test", "CoffeeScript", "Node", "Javascript"
    ],
    "photo": "jl",
    "price": 1000
  }
]
```

So let's write our own version of an "Oracle Database":

```
public class Developers {
  public Developer find(String email) {
    return Stream.of(findAll()).filter(dev -> email.equals(dev.email)).findFirst().orElse(nu
  }

  Developer[] findAll() {
    try {
      return new ObjectMapper().readValue(Resources.getResource("developers.json"), Develo
    } catch (IOException e) {
      throw new RuntimeException("Unable to load developers list", e);
    }
  }
}
```

Here's a corresponding tests: Yes it's a test based on data, no it's not perfect, yes it's a good example of unit testing.

```
import org.junit.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class DevelopersTest {
  @Test
  public void load_developers() {
    Developer[] developers = new Developers().findAll();

    assertThat(developers).hasSize(2);
  }
}
```

Unit testing angular controller with Karma

We can unit test angular controller. We are going to put a foot in the javascript world, using Karma &

Jasmine.

You need to have the angular files available in the path. If you don't use Webjars, it's a good time to type in a `bower install` in your console.

Use the karma configuration file you'll find on the usb key. If you don't use Chrome on your laptop, you can open a configuration file and replace `chrome` by `safari` , `firefox` ... or `ie` !

Testing are launched using `karma start karma.conf.js` (If karma is not in the path, you can find them in `node_modules/karma/bin/karma`).

We use `angular-mocks` (a default testing library for angular) in conjunction with `chai.js` which gives us nice fluent assertions. The testing library used here is `jasmine` . The syntax is `coffeescript` .

```
should = chai.should()

describe 'Basket tests', ->
  beforeEach ->
    module 'devoxx'
    inject ($controller) ->
      @controller = $controller 'BasketController'

  it 'should start with an empty basket', ->
    @controller.emails.should.eql []
    @controller.basket.should.eql {}
```

this is a `package.json` file, it's the `pom.xml` in the node world. It enables us to define all the libraries and dependencies we need for karma to launch properly.

```
{
    "name": "devoxx-codestory-lab",
    "version": "1.0.0",
    "description": "Node dependencies for devoxx-codestory-lab",
    "main": "index.js",
    "author": "Jean-Laurent de Morlhon && David Gageot",
    "license": "MIT",
    "devDependencies": {
        "chai": "^1.9.1",
        "coffee-script": "^1.7.1",
        "karma": "^0.12.16",
        "karma-coffee-preprocessor": "^0.2.1",
        "karma-jasmine": "^0.2.2",
        "karma-jsmockito-jshamcrest": "0.0.6",
        "karma-phantomjs-launcher": "^0.1.4",
        "jsmockito": "^1.0.5",
        "protractor": "^0.20.1"
    },
    "scripts": {
        "test": "node_modules/mocha/bin/karma start karma.conf.coffee"
    },
    "repository": {
        "type": "git",
        "url": "git://github.com/CodeStory/devoxx-quickstart.git"
    }
}
```

maven frontend plugin

Close your eyes, welcome to the wonderful world of maven xml and plugins : the maven frontend plugin is able to do dirty stuff you don't want to have to do by hand, especially if you haven't done node stuff recently. So the price to pay is the following horribles 40 lines. But trust us, it's a life saver, it automates reliably the process of launching javascript tests trough karma.

```
<profiles>
<profile>
```

```
    <id>karma</id>
    <activation>
      <property>
        <name>!skipTests</name>
      </property>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>1.3.2</version>
          <executions>
            <execution>
              <id>extract webjars</id>
              <goals>
                <goal>java</goal>
              </goals>
              <phase>generate-test-resources</phase>
            </execution>
          </executions>
          <configuration>
            <mainClass>misc.ExtractWebjars</mainClass>
          </configuration>
        </plugin>
        <plugin>
          <groupId>com.github.eirslett</groupId>
          <artifactId>frontend-maven-plugin</artifactId>
          <version>0.0.16</version>
          <configuration>
            <workingDirectory>${project.basedir}</workingDirectory>
          </configuration>
          <executions>
            <execution>
              <id>install node and npm</id>
              <goals>
                <goal>install-node-and-npm</goal>
              </goals>
              <phase>generate-test-resources</phase>
              <configuration>
                <nodeVersion>v0.10.29</nodeVersion>
                <npmVersion>1.4.16</npmVersion>
              </configuration>
            </execution>
            <execution>
              <id>npm install</id>
              <goals>
                <goal>npm</goal>
              </goals>
              <phase>generate-test-resources</phase>
              <configuration>
                <arguments>install</arguments>
              </configuration>
            </execution>
            <execution>
              <id>karma tests</id>
              <goals>
                <goal>karma</goal>
              </goals>
              <phase>test</phase>
              <configuration>
                <karmaConfPath>${project.basedir}/src/test/karma.conf.ci.js</karmaConfPath>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
  </profiles>
```

http service

Here you can write a more complicated test, to handle some tricky situation where your angular controller is making an http call (wich occurs... very often).

```
should = chai.should()

describe 'Basket tests', ->
  beforeEach ->
    module 'devoxx'
    inject ($controller, $httpBackend) ->
      @controller = $controller 'BasketController'
      @http = $httpBackend

  it 'should refresh basket after adding a developer', ->
    @http.expectGET('/basket?emails=foo@bar.com').respond '{"test":0,"back":0,"database":0,"fro
    @controller.add 'foo@bar.com'
    @http.flush()

    @controller.emails.should.eql ['foo@bar.com']
    @controller.basket.should.eql
      test: 0
      back: 0
      database: 0
      front: 0
      hipster: 0
      sum: 0
```

Deuglifying the page.

You can do your own css, but you can also use css webjars like Twitter Bootstrap to ease the pain of spending two hours having 3 divs side by side.

To add bootstrap you can use Webjars. You add to your pom:

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.0</version>
</dependency>
```

If you use the YAML Front Matter you can easily add in the header:

```
---
title: recruteur.io
styles: ['/webjars/bootstrap/3.1.1/css/bootstrap.css']
---
```

Now you should have everything you need to finish the app.

What are you waiting ?? Jean-Claude is not a patient man and as during estimation phase you said the project could take between 3 hours to two days, Jean-Claude thinks you can make it in 2 hours, or your job will be outsourced in a far away countries, where developers are cheap.

-- David & Jean-Laurent