Capgemini

Java EE Training Course

Cap Gemini

week21-AngularDart

microRPG

1.1

week21-AngularDart

group size:	1
repository name:	Dart_microRPG
repository rights:	gecko
language:	Dart, Angular2, HTML, CSS

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

Foreword

Up until now, we've seen how to use Java to create the dynamic part of a website, handle requests, generate pages etc... But this is just one of many existing technologies you can use! Dart is a language developed by Google, designed entirely to handle websites, or servers. The syntax is close to Java, so you shouldn't have too much trouble adapting.

The real difference is that the whole Dart standard library is built to be efficient in a browser, or on the server, and bundles everything a web developer could need. For instance, converting Dart objects to JSON is a **standard operation**: the `dart:convert` library includes everything needed to serialize and deserialize objects into different languages, without any annotations or anything complicated; an object is a map, and a map is easy to convert to JSON.

For the front part, we are going to use Angular2, which you should already know. Fear not though, we are dropping JS in favor of Dart! **AngularDart** is the official port of AngularJS to Dart, meaning you don't have anything specific to AngularDart to learn: it works exactly the same, but with Dart running behind instead of JS.

As opposed to Java, Dart is **made** to run in the browser alongside Angular. If the browser doesn't natively support Dart, a **dart2js** script is automatically called so that your Dart code is transpiled to minified JS, executable by any browser! What this means is that most of your webapp will be rendered directly in the user's browser, making the server only serve files (mostly). Several years ago, this would have been a big no-no, since web browsers were not capable of much more than rendering html. That is why technologies like JEE or Spring

worked so well: they allowed for easy HTML generation, with simple intermediate languages (JSP), and a real language (JAVA) behind the scenes to handle everything else like you would do for any application. Web became closer to 'standard' development, making it widely used.
Nowadays, web browsers are capable of a lot of things, and are made to handle it correctly. Google Chrome might be the most advanced one, Firefox is close behind, Safari ... well, works great as long as Apple decides it should work this way, and Internet Explorer is ... the least advanced, by far. All those browsers manage JS correctly though, which means dart2js works perfectly with them. Plus, using Polyfills, a set of HTML / CSS / JS functions and structures used as a compatibility 'framework', you ensure all browsers handle webcomponents the same way (or close enough).

The real advantage of AngularDart is the Component paradigm. Seeing your web page as a collection of interdependant components seems obvious today, but very little languages offer easy component creation without any external libs. Dart does, using Angular (or Polymer) to define what is a component, and exposes functions to manipulate the DOM (Document Object Model, the set of html nodes rendered by your browser) tree directly.
An example is worth more than long texts, so let's dive right in! To illustrate the technology, we are going to implement a simple text based RPG, making heavy usage of webcomponents.

STEP 00 - SETUP

You might find references to WebStorm in the Dart docs: it is just a 'fork' of IntelliJ with default plugins for web development, but IntelliJ does exactly the same, without the default plugins. To acquire those plugins, go to Settings > Plugins > Browse Repositories and search for Dart, install it, and restart IntelliJ.

You'll have to install Dart first. Follow the little tutorial.

Step 01 - Create the project

This step is going to be really faster than with Java and Spring, as the whole language is designed to build dynamic webapps. We will still be using IntelliJ, which has the best tools to develop Dart.
Create a new Dart project, using IntelliJ menus: New > Project > Dart, chose the WebApp archetype, name it Dart_microRPG, and that's it, everything is setup.
The only configuration file you have to care about is the **pubspec.yaml**, which describes your application. It regroups metadata about the developer, the application itslef, like the version number, the dependencies, and the plugins we want to apply during the build process, called **Transformers**.
If you followed the little installation tutorial, you should have a basic todo list in AngularDart!
Let's look at the generated code. First, let's explain the directory structure:
A Dart web application is composed of several necessary files. First, under the **web** folder:

- **index.html**: nothing new here, except those lines:
  `<script defer src="main.dart" type="application/dart"></script>` and
  `<script defer src="packages/browser/dart.js"></script>`
  the first one is the entry point of our application, which contains the root component. The second one loads Dart. The only html 'code' here is the `<my-app>` tag: this is what a component is; just a custom HTML tag, nothing more. The magic happens in `main.dart`.
- **main.dart**: the entry point of the application. No need to modify this, except in very particular cases. It imports AngularDart and our application, and bootstraps our app in Angular (links everything together).
- Other resource files.

The **web** folder **ONLY** contains the entry point of our app, nothing more. In fact, it only contains what is necessary to bootstrap the application. Everything else, all our code, HTML CSS or Dart, is going in the **lib** folder. Let's look at it now:

- **app_component**: Those three files represent the 'main' component of your app; hence, app_component. See it as the main function of your webapp: it contains default configuration and describes the root component.
- **src/*** : This is where you put everything related to your app. We try to create a folder per component: everything is clean and organized this way. For now, there is just the little todo list app.

There is also a **test** folder generated by stagehand (the program that generates projects achetypes), but we won't use it for now.
You might wonder what **root component** means, so let's explain:
A component oriented webapp looks like a tree: you have a root component, that will contain all other components as its children. Each of these components can also have children, thus creating a tree: the **DOM Tree**. The point is to be as close as possible to the way HTML is handled: each tag can contain other tags inside, also creating a tree. The browser (Chrome, Firefox ...) then parses this tree and renders it to the screen. With web components, you can even create a **Shadow DOM**, a dynamic DOM, manipulated by Dart and Angular, that will

be inserted in the ‘natural’ tree just before it is rendered.

That may seem a bit technical, but it just means that with web component technologies, you can both generate HTML **AND** manipulate DOM elements directly **in the browser**, whereas JEE generates everything in the Servlets, which are executed **server-side**!

The project

+ Cleanup

This step is optional, so feel free to skip it.
Just remove the todo_list folder, as we won't need it for the rest of the project

+ Models

As usual, we are going ot need some models:

Hero

The Hero class will hold every information related to your character:

- A name as a String
- A level as an Integer
- An amount of experience gained so far
- An amount of experience needed to level-up
- An amount of health points as an Integer
- An amount of Mana points as an Integer
- A List of Abilities

Ability

An ability is what a Hero can cast to fight. It will hold:

- A name as a String
- A description as a String
- A Type as a SpellType
  - SpellType is an enum containing **Physical**, **Magical** and **Neutral**
- An amount of Mana required to cast it as an Integer
- A number of Health points to remove when it hits an ennemy.

Monster

A monster is a minion of Evil. They are what you are fighting against, and will hold:

- A name as a String
- A level as an Integer
- An amount of Health points as an Integer
- An amount of experience given when defeeated as an Integer

Don't bother with database storage for now, just use Maps for everything, it is enough for today's subject.

Pseudo database

Create a FakeDatabase class that will hold maps for each Model type, simulating a real database.

Using dart packages to bind your data to MongoDB is really easy though.
Feel free to try it if you feel you have the time

+ Views

No surprise there either, we need to display our models. Once again, you are free to design the views however you like, just don't loose too much time on it.
For now, just implement a way to display and modify the models:

- A Hero view showing the info of one Hero
  - With a 'sub-view' to display the list of abilities and their description
- Monster view to display the info of one Monster

Use the Dart tutorial to learn how to bind HTML, Dart, and Angular together. It is really well made, and if you search well enough, you might even find something really similar to our problem... Just adapt it to our exact needs.

Once you have those views, start implementing a Battleground component. It needs:

- A List of Heroes
- A List of Monsters
- A button to use an Ability
  - A way to list the abilities available to the currently playing character and click on them to cast them.

Of course, you will need some Dart, HTML and Angular to create this view. Once again, the documentation has everything you need.

+ SERVICES

As usual, we need something between our views and our models, and once again, we are going to call them Services.
Create a HeroService class that will hold every function you need to access your Hero 'pseudo-database'. Do the same for the abilities and the monsters.

Your Angular components MUST NOT directly call the Database. They MUST use the services to access it.