Dev in Test - Recruitment Task

For the hereby "*Movie List*" project I am using classical, three layered architecture including Angular for the **Client (Front End) layer**, Node for the **Business (Middle) layer** and I am using a Data.json file as a mock for the **Data (Back) End layer**.

Business (Middle) layer

Creating services with Node and Express is simple and fun. What I like to do is to use promise libraries such as Q. A promise is an abstraction for asychronous programming. I also prefer to detach my logic into separate, middleware files. I do this on purpose. I create different modules of logic. One or combination of several modules might implement the endpoint logic.

Implementation:

- The server is serving server/data.json when a request is made to /api/movies

- Clients are being able to request a subset of data

    - Allowed limiting the number of items returned – list parameter

    - Allowed pagination of data – page parameter

Testing:

I really would like not to just test my endpoints, but to test each of the logic modules. Once I am sure my modules are doing fine I am attaching them to my endpoint and microservices. The Behavior Driven Development (which is extension of the Test Driven Development) is the core motivation.

For testing my Q promises nicely I am using the extension for Chai – Chai-As-Promised and Mocha. In the tests you can find various scenarious where I am showing the very Best Practices for testing promises. The comments are all around the implementation and the testing logic.

Client (Front End) layer

Structure:

Resuable modules. I believe that what works together should live together. All necessary files of each single module are in one being stored in folder, Example: 'public/js/app/modules/…'. This helps to easily store and find all the build files of each module (Controllers, Directive, Views). This work great for average or huge Web Applications where the logic of hundreds of Controllers, Directives and Views are being declared in the code.

Organazing the code 'The Domain Style'

With a complex domain model and hundreds of components, an enterprise application can easily become a mess if certain concerns are overlooked. One of the best ways to organize the code in this situation is by distributing each component in a domain-named folder structure.

Example:

app/ -> files of the application

dist/ -> the concatenated js and css files

app.min.css -> main application stylesheet, consists of concatenated and minified css files

app.min.js -> main application java script, consists of concatenated and minified js files

ReusableModules/

login/ -> login module directory

login.css -> login stylesheet

loginCtrl.js -> login controller

login.html -> login view

listMovies/ -> listMovies module directory

listMovies.css -> listMovies stylesheet

listMovies.js -> listMovies controller

listMovies.html -> listMovies view

movie/ -> movie module directory

movie.css -> movie stylesheet

carCtrl.js -> movie controller

movie.html -> movie view

lib/ -> javascript libraries

angular.js -> AngularJS script

index.html -> main html file

Implementation

For the front end I prefer to break down the logic for gathering data from backend:

- I am having an ***'API Service'*** layer, which is only responsible for doing the CRUD operations and to handle the communication with the servers. API service provider. Responsible for declaring and offering services to the backend using $http. It handles and saves (logs) any errors related to backend oeprations. It offers Layer of abstraction for dealing with backend manipulations. ***'APIservices'*** are being used together with the ***'modelServices'***. '***Model services'*** is another layer of abstraction for saving and updating any data with the backend. It offers to the controllers in the application reusable logic for saving and sharing temporary data models saved in the model services.

randomController => modelService => APIService => server operation request.

- I use add another layer of abstraction the '***Model Service'*** layer which delegates the CRUD operations to the an ***'API Service'*** layer and stores the received models (might be a simple or more complex object, or collection of objects). The received model might be updated and changed in the front end. When we are ready with the changes we are pushing the updated model to the ***'API Service'*** layer for further manipulations with the '***Business and Data Layers'***.

All the requests between all the different layers of abstraction are being done with ***$q,*** which is the promise library for working with Angular.

Testing

I am testing my '***Front End Layer'*** with Jasmine and Karma. Protractor is nice tool as well. In protractor test I am showing how to isolate the test logic (***movieListPage.js***) and reuse it in different tests (***movieList.spec.js***). Creating reusable, easy maintanable code for the test services is essential for boosting the development of the test.

Acceptance Criteria Covered

- On page load, you I display first 20 movies, ordered alphabetically by title.

- For each movie list: title, year, rating, duration in minutes and all actors.

- If the search box contains less than three characters, but more than 0, the following message is being displayed: 'Enter at least three characters to begin search'.

- When the user enters a search phrase of three characters or more, a list of movies is displayed below the search box, filtered down to the ones whose title field contains the phrase entered.

    - The match is case-insensitive.

    - The search is being performed as the user types.

- When there is at least one matching movie:

    - The following text is displayed: 'Matched X of Y movies total'

    - Where X is the number of movies found and Y is the total number of movies

    - For each movie, both Title and Year are displayed

- Backend

    - The server serves server/data.json when a request is made to /api/movies

    - Clients is being able to request a subset of data

        - Allowed limiting the number of items returned

        - Allowed pagination of data, e.g. get 10 items from the 4th 'page'

- When there is at least one matching movie:

    - The following text is displayed: 'Matched X of Y movies total'

    - Where X is the number of movies found and Y is the total number of movies

    - For each movie, both Title and Year are displayed

- When the search phrase does not match any movies, the following message is being displayed: 'No matching items'

Automation of the tasks and processes

Automation of the processes is the best approach to save developers time from the boring, repetitive tasks so that he can spend it in other more important parts of the development. It is a must and I like using Grunt. In this project the main commands to use are:

grunt test – runs all the tests [e2e, unit, server-unit]

we can run any of the test separately by running the following commands:

grunt e2e

grunt unit

grunt server-unit

grunt rebuild – it cleans the dist directory, runs jshint to confirm that the js is error free, concatenates all js into one bundle, then we check for any errors after concatination again using jshing, then we minify the js and the css.

grunt start – it rebuilds the project and runs the Express Server

The commands are alternative to the one used in the base project:

Commands to run:

npm run start - starts local server

npm run unit - run the karma unit tests

npm run e2e - run the protractor tests

npm run server-unit - run the backend mocha tests

Using Bower

Bower is a package manager for the web that I like to use. I showed some example of how to use it together with grunt and how to automatically update my libraries for the project.

Example: **grunt update-frontendlibs** will update the front end libraries used within the project. Simplify builds of a dependency for dev vs. prod. No need to commit dependencies to version control.