Getting-started-report

# Purpose

We chose to work with the entire MEAN-stack. This decision was made because only one person in our group had experience with web development. The fact that MEAN is completely based on JavaScript, minimizes the number of languages we had to learn. Furthermore, Joel already had some experience with programming in JavaScript. In the end, we chose to keep MEAN-stack entirely and not substitute any of the technologies because we thought it would integrate best this way.

The versions we used are:

* MongoDB: v3.4.9
* Express.js: 5.5.1
* AngularJS: v1.6.6
* Node.js: v8.7.0

# Technology

## 2.1 MongoDB

The architecture used by MongoDB is “The Nexus Architecture”. It is based on combining the strengths of RDBs which have been developed over time, with the strengths provided by the innovative NoSQL databases. The properties based on the relational databases are: the *expressive query language and secondary indexes*, so the data can be accessed efficiently in the database rather than implementing access in the application code. *Strong consistency* and the ability for *enterprise management and integration*. Properties coming from the more recent NoSQL side are: a *flexible data model* which allows, for example to store data of any structure. *A scalable database*, horizontally scalable through, what they call ‘sharding’. This is a form of partitioning. Lastly the database is *designed to run worldwide* across many different servers and data-centres, it was made for the Cloud.

In MongoDB the data are stored as documents. More specifically the use BSON (binary JSON), which is based on JSON. Only it allows to also use different types of data types such as date, int, long and floating point. The documents consist of one or more fields which each contain a certain data aspect in a certain data type. Documents with a comparable structure can be organized in collections. These collections can be thought of as analogous to tables in a RDB.

## Express.js

Express is a HTTP framework used in the server back-end together with Node.js. The API for Express is based on five different concepts. Four objects and one set of special functions. To start with, the ‘**express()**’ functions. Express() creates an Express application. There are methods for this function to deal with JSON-files, static files, routers and encoded URL. The ‘express()’-function is the top-level function for the Express module. Then the four objects.

**Application**. The ‘app’-object embodies the Express instance. It has methods to route the HTTP requests, to configure middleware, to render HTML views and to register a template engine. The app also has settings to tweak how the application behaves.

**Request**. The ‘req’-object embodies the HTTP request. It has properties for the requests parameters, HTTP headers, body, query string… The request object is an improved version of the request object in Node.js itself. It supports all built-in fields and methods.

**Response**. The ‘res’-object embodies the HTTP response given by an Express application after receiving a HTTP request. The response object is an improved version of the response object available in Node.js itself. It supports all built-in fields and methods.

**Route**r. The router object can be thought of as a mini-application, which is solely able to carry out middleware and routing functions. Each Express instance has a built-in application router. The router also acts as a middleware, it can be used as the argument of an ‘app.use()’-function or as the argument of the ‘use()’-function of another router. The router can be created by the ‘router()’-function of the top-level Express object.

## AngularJS

AngularJS deals with the frontend of your web application, it works mainly with a Model View Controller model. It makes it possible to create dynamic webpages. Extending HTML, which focusses on static pages. AngularJS can be divided into nine modules, of which ‘**ng**’ is the core module. The ‘ng’ module contains the most important components and is included by default. It has the core directives, services/factories, filters and the core global API functions linked to the *angular* object. The other modules need to be added manually.

The ‘**ngRoute**’ module can be used to enable URL routing in your application. URL management is then supported in hashbang and HTML 5 pushState.

The ‘**ngAnimate**’ module enables animation elements in your application. When ‘ngAnimate’ is included numerous core will implement animation hooks into your app. Animations are defined as CSS transitions/animations or JavaScript call backs.

‘**ngAria**’is used to inject common accessibility attributes into directives and improve the experience for users with disabilities. ‘**ngResource**’ is used to query and post data to a REST API. With ‘**ngCookie**’ you can handle cookie management in your application. ‘**ngTouch**’ is used to develop for mobile browsers/devices. ‘**ngSanitize**’ enables you to securely parse and manipulate HTML data in your app. ‘**ngMock**’ can be used to inject and mock modules, factories, services and providers in your tests.

## Node.js

Node.js is the runtime environment where the MEAN-stack is built around. It can be seen as the breakpoint for the usage of JavaScript as a server-side scripting language. Node.js is a single threaded technology, however concurrent processes are supported by the asynchronous event-loop. The event-loop is used to manage blocking processes, such as I/O. Blocking processes are taken out of the event-queue and put in the thread-pool. They are brought back into the event-queue once they are ready and the event-queue is empty, with a call-back function.

# Software and hardware requirements

# Simple examples

--add a few basic mongo shell commands--

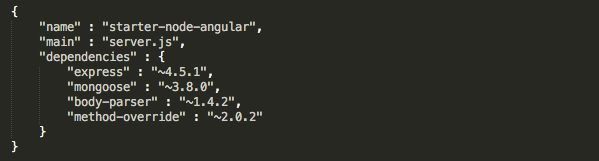
Instead of making a simple example for every technology separately, I will make a simple example of how to make a one-page MEAN-stack application. This seems more interesting, because this way we also discuss the interaction between the technologies.

In this introduction, we pay specific attention to the structure of the application. This will allow the application to scale more easily later on. This is the structure we will use:

* app
  + models/
    - courses.js
  + routes.js
* config
  + db.js
* node\_modules
* public
  + css
  + js
    - controllers
    - services
    - app.js
    - appRoutes.js
  + img
  + libs
  + views
    - home.html
    - courses.html
  + index.html
* .bowerrc
* bower.json
* package.json
* server.js

All backend work is spread across ‘*server.js*’, ‘*app*’ and ‘*config*’. While all frontend work happens in the ‘*public*’ folder.

A node application starts with the package.json file, in this case we will fill it with:

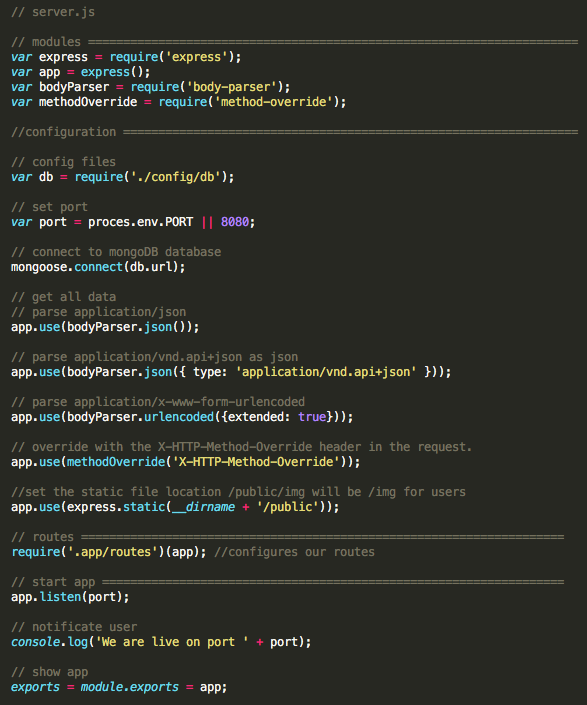


To install the technologies we introduced in the *package.json* file, we need to run:

***$ npm install***

This will create the *node\_modules* file, mentioned in the file structure above.

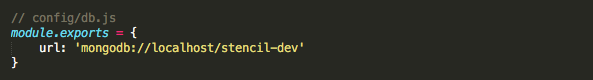
*server.js* is the heart of the application. This is the code we will put into it:



Here we pulled in the modules, configured the application, set some Express settings set the routes and then, started our server. We will pull in *mongoose* later in our model.

## Config

For the time being, all we need in this file is *db.js*. Later more config files can be added and called in the *server.js* file. This is what we want in the db.js file:



You can now call any items in it using *db.url* –check this—

## Installing MongoDB

This intermezzo explains how to download and install MongoDB. There are a lot of different ways to do this. Using Homebrew on mac I just ran:



This installs MongoDB here:



MongoDB is programmed to store its data in the file ***/data/db***. What you need to do next is create this file and give yourself the rights to read and write this file. On mac, this can be done with:



Next you need to add ***mongodb/bin*** to ***$PATH***



Now you are ready to run. Try running: ***$ mongod***

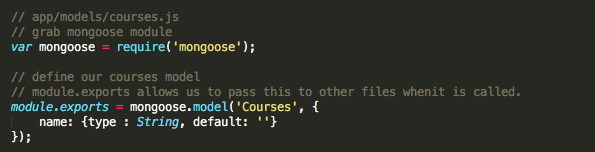
If at the end of the output you see: “*waiting for connections on port 27017*”, this means MongoDB is up and running.

You now open another window of Terminal and run ***$ mongo***.

If everything went well you should now be able to use MongoDB with its mongo shell.

## Courses model app/models/courses.js

This file is required to create records in our database. When the Mongoose model is created it will allow us to create, read, update and delete our sports courses. This is the code we write in this file:



With the Mongoose modules, we create this model for the courses. –structure used for courses--

## Node Routes app/routes.js

The app directory can be used in the future to add anything to do with the backend which is specific to your app, such as models, routes, controllers etc.

You would want to separate the frontend and backend duties as much as possible. This is the pace where you can deal with your API routes. –everywhere where a (\*) is used we send the user to the frontend, where Angular can deal with the further routes. --

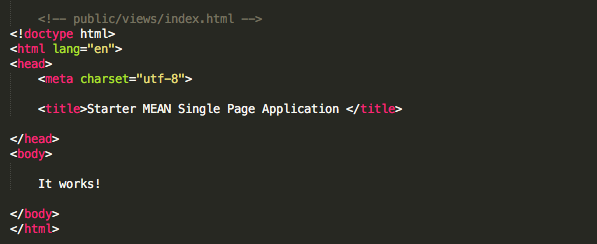


--backend done—

We now have everything to get up our server. Now we can start up our server, send a user the Angular app (index.html) and handle one API route to retrieve the sports courses.

## A quick index view file to test our server public/views/index.html

This is what we put into it:



Running ***$node server*** in the root directory of your application should start running the webpage on port 8080 now.

-- Frontend AngularJS –

The Node backend we created will send everybody who visits our app to index.html. We defined it that way with the “***app.get(\*)***”. This is called a catch-all route. –where—

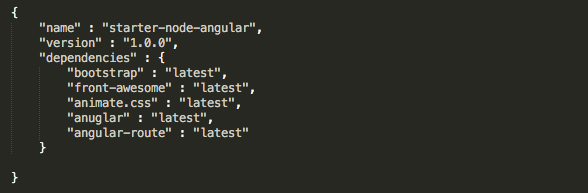
## Installing Bower and pulling in the components

We need components such as bootstrap, bower and angular to complete our frontend. We will use bower to get all these components. Bower is a frontend tool manager.

To install bower on your machine run: ***$ npm install –g bower***. This gives you access to bower globally. We will need two files in the root directory of our application to get started: ***.bowerrc*** and ***bower.json***.

***.bowerrc*** tells Bower where to install the files:



***bower.json*** is analogous to ***package.json*** it tells Bower which packages we need:

If you run ***$bower install*** in the directory with ***bower.json*** now, it will install all the required technologies in ***public/libs***.

What we implement:

* Two pages: Home and Courses
* An Angular controller for each.
* An Angular service for Courses
* No refreshing when navigating between pages.

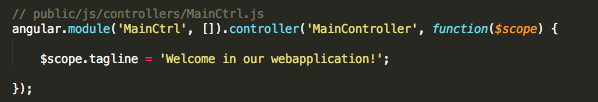
As mentioned before everything will be done in the ***public*** directory. This is the structure we want for this directory:

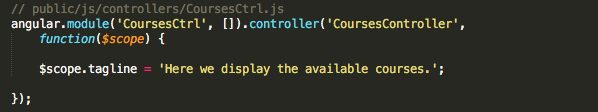
* Public
  + Js
    - Controllers
      * MainCtrl.js
      * CoursesCtrl.js
    - Services
      * CoursesService.js
    - App.js
    - appRoutes.js

When all controllers, services and routes have been created, we inject them into ***app.js***. Then everything will be able to work together.

## Angular controllers

For this one page app, the controllers are very basic. In practice, they can become a lot more complicated. This is the code we put into these controllers:





## Angular services

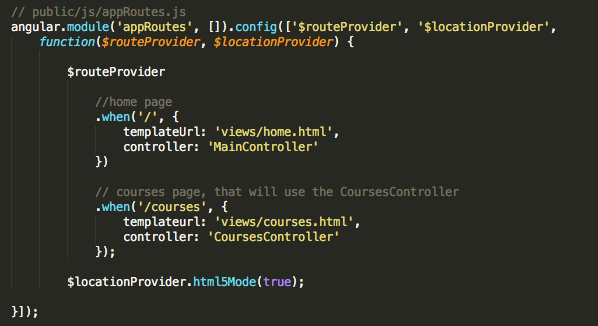
The services are where you would do API calls to your Node backend from your Angular frontend, using ***$http*** and ***$resource***



You could also add services for e.g. creating or deleting a course. But as in the ***routes.js*** file, we will not include them here.

## Angular routes

We define the Angular routes inside the ***public/js/appRoutes.js*** file.



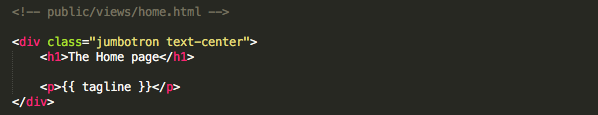
The Angular frontend will use the template file and inject it into the ***<div ng-view></div>*** in the ***index.html*** file. This will happen without a page refresh, which is exactly what we wanted in this one-page application.

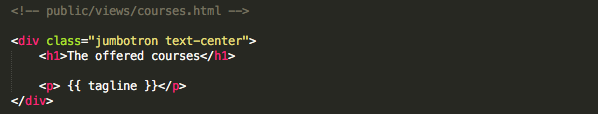
## Rework the view files

Now all the Angular routes are in place, we will create some view files as the finishing touch. ***home.html*** and ***courses.html*** will be injected into our ***index.html*** file inside of the ***<div ng-view></div>***.

In this ***index.html*** we are calling the components we pulled in with Bower.







Now that everything else is done, all that is left to do is make sure that everything works together. We take care of this in the ***app.js*** file. Using dependency injection, we will set up our application to use all the components.



# Summary and conclusions

The largest advantages of using the MEAN-stack are of course:

* Your code is isomorphic because JavaScript is used everywhere. This makes programming very **flexible**. Isomorphic code means: if you implement a certain part of your project in the backend, but you decide later that it would be better in the front end, you can just copy most of your code into the frontend and it would still work as desired.
* The technology has been thoroughly tested. MEAN-stack is being used by a lot of big companies for development. You can use these technologies without needing to worry about the technologies failing you.

Disadvantages of using the MEAN-stack:

* MEAN is probably a bit overhyped at the moment. This is not a disadvantage of the technology, but it might cause people to use it, without properly considering the alternatives. More people will end up using MEAN, while other technologies might have been more apt.
* Because of its runtime architecture MEAN would not be the best choice if you have a lot of heavy computations and data processing on the server side. Micro services would be more fit for such tasks than the monolithic architecture.
* It is difficult to realise joins on a MongoDB database, this can make querying a pain. For large datasets MongoDB can contain a lot of redundancies.

Conclusion: MEAN-stack is cited as one of the simplest technologies for web development to learn. As we are a relatively inexperienced team, this is a good choice for that part. Furthermore, our application will be rather small and it won’t demand any complex calculations. The database needed for storing all university sport activities will certainly not need to be huge. However not being able to join in queries might prove to be a challenge.

# Glossary

# References

* The MongoDB Architecture Guide: <https://webassets.mongodb.com/_com_assets/collateral/MongoDB_Architecture_Guide.pdf>
* The API provided on the Express.js website: <http://expressjs.com/en/4x/api.html>
* The API provided by AngularJS: <https://docs.angularjs.org/api>
* The AngularJS website: <https://angularjs.org/>
* The official Node.js website: <https://nodejs.org/en/>
* The API provided by Node.js: <https://nodejs.org/dist/latest-v8.x/docs/api/index.html>
* A tutorial on the MEAN-stack: <https://scotch.io/tutorials/setting-up-a-mean-stack-single-page-application>
* <https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-javascript-full-stack-development/>
* <https://www.brainmobi.com/blog/advantages-mean-stack/>
* <http://digi117.com/blog/what-are-the-advantages-of-developing-apps-with-mean-stack.html>