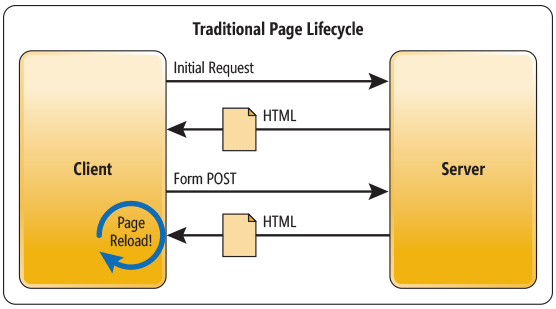
# Single page application (SPA).

## What is SPA?

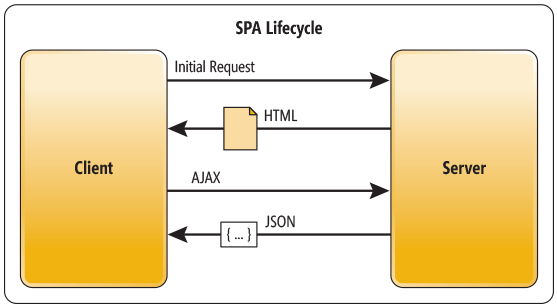
A single-page application (SPA) is a web application or web site that fits on a single web page with the goal of providing a user experience similar to that of a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[1] or the appropriate resources are dynamically loaded and added to the page as necessary

Multi-page Website (MPW). In a traditional Web app, every time the app calls the server, the server renders a new HTML page. This triggers a page refresh in the browser.



|  |  |
| --- | --- |
|  | JSP |
| Template engines | Thymeleaf + ***thymeleaf-layout-dialect*** |
| Freemarker |
| Velocity |
| Layout engines | Apache Tiles |
| SiteMesh |

In an SPA, after the first page loads, all interaction with the server happens through AJAX calls. These AJAX calls return data—not markup—usually in JSON format. The app uses the JSON data to update the page dynamically, without reloading the page.



One benefit of SPAs is obvious: Applications are more fluid and responsive, without the jarring effect of reloading and re-rendering the page. Another benefit might be less obvious and it concerns how you architect a Web app. Sending the app data as JSON creates a separation between the presentation (HTML markup) and application logic (AJAX requests plus JSON responses).

This separation makes it easier to design and evolve each layer. In a well-architected SPA, you can change the HTML markup without touching the code that implements the application logic (at least, that’s the ideal). You’ll see this in action when I discuss data binding later.

In a pure SPA, all UI interaction occurs on the client side, through JavaScript and CSS. After the initial page load, the server acts purely as a service layer. The client just needs to know what HTTP requests to send. It doesn’t care how the server implements things on the back end.

With this architecture, the client and the service are independent. You could replace the entire back end that runs the service, and as long as you don’t change the API, you won’t break the client. The reverse is also true—you can replace the entire client app without changing the service layer. For example, you might write a native mobile client that consumes the service.

We already used at least two SPA frameworks such as:

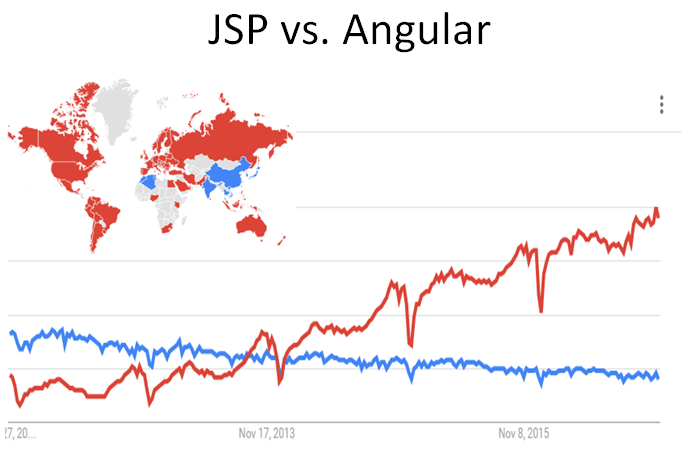
* IsomorphicJs
* ExtJs

The problem is that in most cases we were completely ignored SPA paradigm for which those frameworks were introduced. Instead we render new page on almost every request, moreover every page reload javascript framework. Even most of browsers supports static resource caching it doesn’t make such approach optimal**.**

We didn’t use

* AngularJs
* Angular II

but I’ve experiment with AngularJs. Find it very complicated and tricky. Finally this September google release first version of Angular 2.

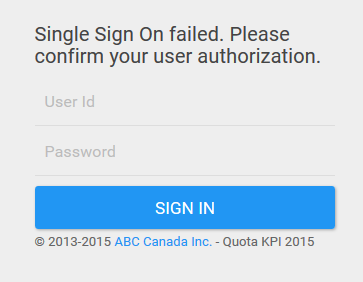


## Mock project overview

Mock project developed as template for new projects. It demonstrates two UI front-ends. One is for site administrators and business analysts based on ExtJs framework. Other one for clients and visitors based on Angular 2. Both UI use SPA paradigm.

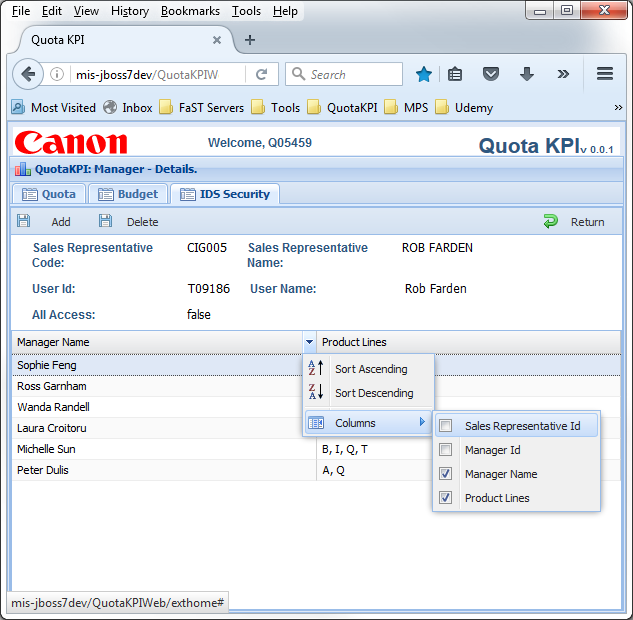
### Workflow Login page

From Login page depending of user role login module will forward user to either admin or client view.



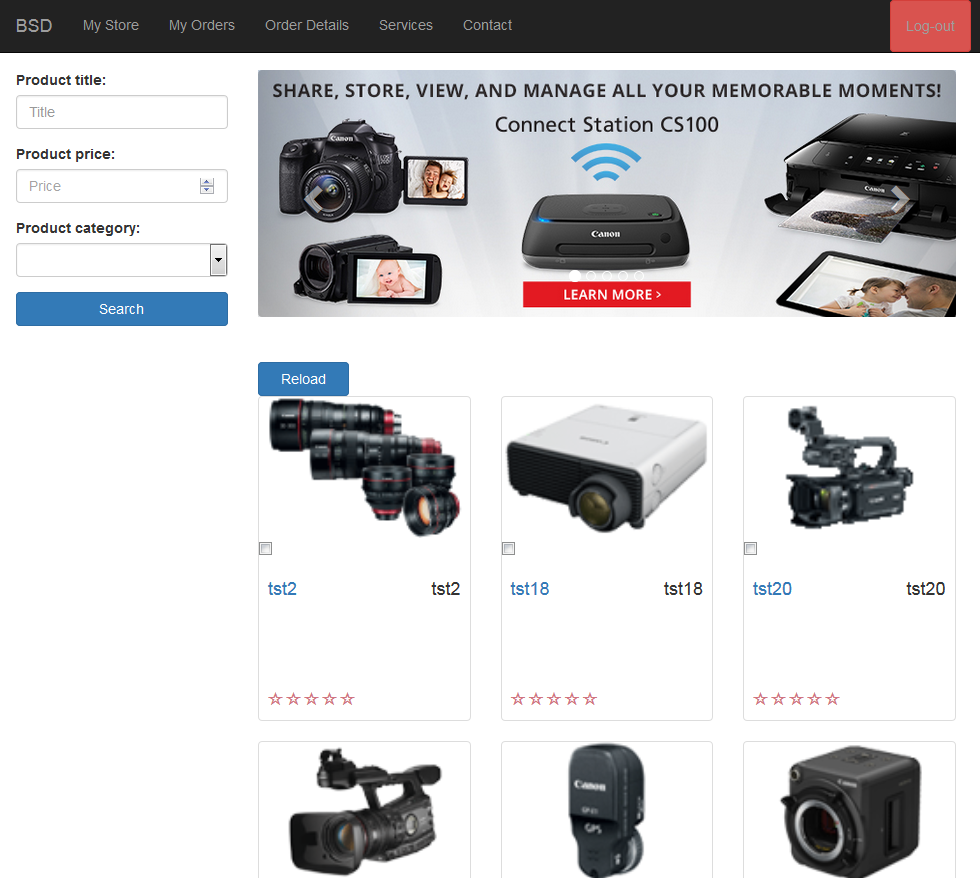
### Admin view

Admin view allow site admin or business analysts work as they would in desktop application



### Clent view

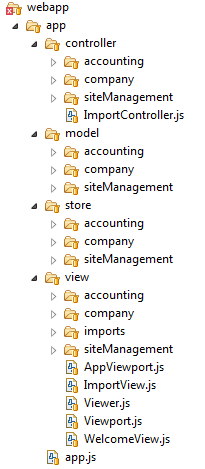
Client view has almost traditional web page look and feel, but in our project we use modern Angular 2 framework which is as close to SPA as ExtJs site.



## SPA with ExtJs

Both frameworks suggest to use application directory layout. (Very similar to maven project structure. Yes, you may completely customize it but all you’ll get is a big problem)

Regular project structure for ExtJs app with default folder names

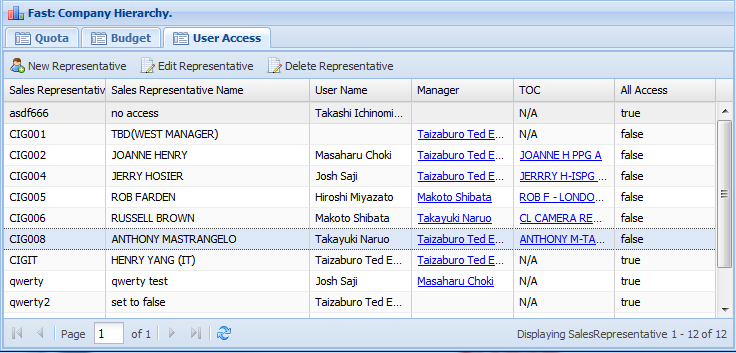


As you can see it’s quite intuitive

* Root application folder is what you expect ‘app’
* ‘view’ folder for all visual components
* ‘model’ folder contains all entities. These models/ entities are twins siblings to their server side counterparts.
* ‘controller’ folder contain all application controllers. Because SPApplication may provide rich behavior it require controllers which handle full components life cycle.
* And last folder ‘store’ which contains all datasources for application components

Here is very basic picture how those parts works together:

* SalesRepresentativesGrid represents table on screen
* SalesRepresentativesGrid takes data from SalesRepresentative store
* SalesRepresentative store use SalesRepresentativeModel format to retrieve data from json
* All user interactions orchestrated by MaintenanceController



appview/company/SalesRepresentativesGrid.js

app/store/company/SalesRepresentative.js

app/model/company/SalesRepresentativeModel.js

app/controller/company/MaintenanceController.js

## SPA with Angular 2

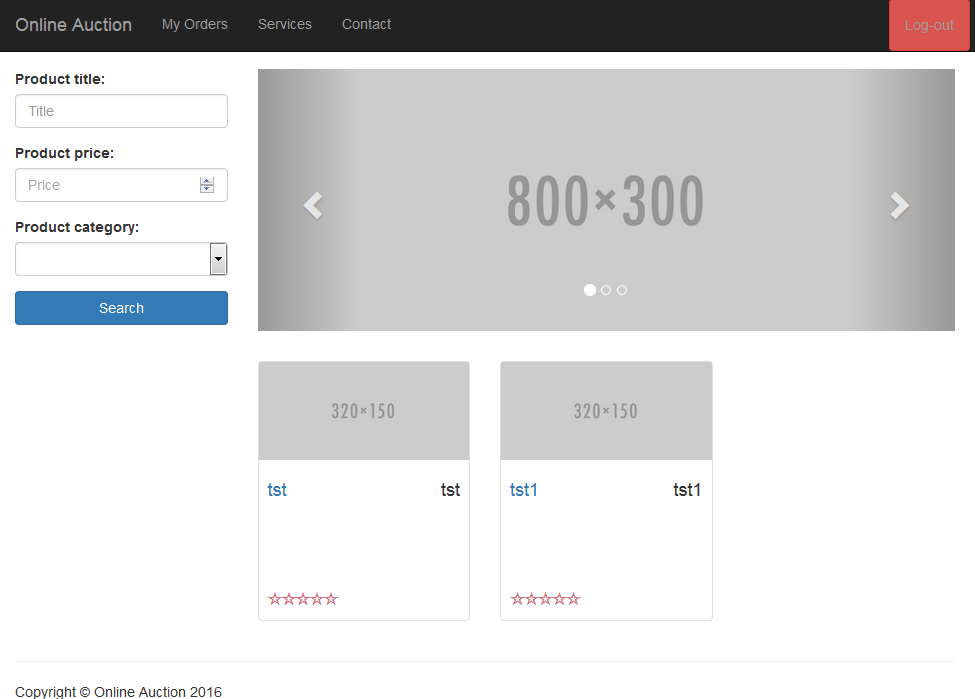
Here is list of few new features which make Angular 2 outstanding:

* Mobile - Angular focuses on the development of mobile apps
* Modern - Angular 2 target ES6/typescript next generation of javascript and “evergreen” modern browsers (those automatically updated to the latest version)
* Modular – No need to load Angular 2 framework as one piece. Load only required modules.
* Components Over Controllers - Unlike AngularJs and ExtJs Angular's main building block is a Component (custom element with its own html tag, controller, view declared inside)
* Improved Data Binding
* Support for Functional Reactive Programming paradigm.
* Routing – improved page flow

All listed features are very important and should not be overlooked when choose UI framework for new projects but my favorite is “Components Over Controllers”.

“*In Accelerating Through Angular 2, we start by teaching components instead of controllers. Learning a framework should start with one of the most important building blocks, which, in the case of Angular 2 apps, is components. It makes the framework easier to learn and use.*”

Look at the snapshot below. Everything you see on page is components, and components as composition of components. This feature is really feature which distinct Angular 2 from ExtJs and AngularJs. Now we can say that Component literally is autonomous building block of UI layer.



product-item/product-item.html

**<auction-search>**

CarouselComponenCarouselComponentt **>**

Home.ts **>**

<auction-footer>

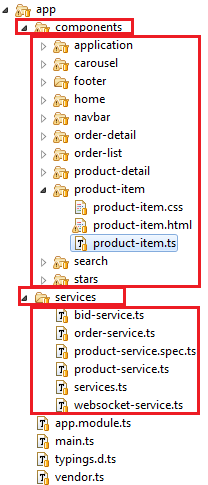
**<auction-navbar>**

Let’s see what is inside

Let’s see what is inside product-item component

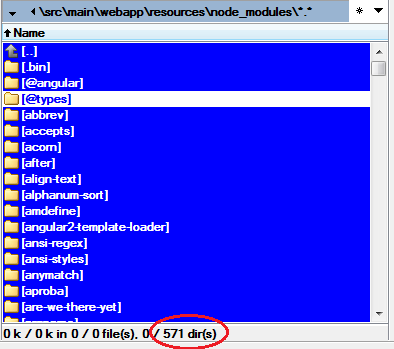
|  |
| --- |
| 001: import { Component, Input } from '@angular/core'; **<- Import what we need from core**  002: import { Product } from '../../services/product-service'; **<- Import what we need from service**  003:  004: @Component({  005: selector: 'auction-product-item', **<- Define HTML tag <auction-product-item>**  006: styleUrls: [ 'resources/app/components/product-item/product-item.css'], **<- Link to CSS**  007: templateUrl: 'resources/app/components/product-item/product-item.html' **<- Link to HTML view**  008: })  009: export default class ProductItemComponent { **<- Controller class for this component**  010: @Input() product: Product;  011:  012: constructor() {  013: }  014: } |

Regular project structure for Angular 2 app doesn’t have to be as strict as for ExtJs application (actually there no conventions at all) but still most tutorials follow convention to separate components from services and each other:



## Deployment of Angular 2 application

If you look at node\_modules\ folder where all angular’s framework dependencies are, you would be surprised how many packages are there



No surprise that at the beginning I had to wait almost a minute until page open. Web traffic was terrible showing ~5MB of downloads and over 500 requests to server.

Nowadays almost all javascript UI applications should be packed for deployment. Browserify, Webpack, Broccoli and Rollup are all popular bundlers. Each of them can create code bundles to be consumed by the browsers. The simplest is Webpack, which allows you to convert and combine all your application assets into bundles with minimal configuration.

In this project I use npm (NodeJs package manager) to manage dependencies and build project

To run ***webpack*** against our project I include build script in package.json

|  |
| --- |
| "scripts": {  "clean": *"rimraf dist"*,  "prebuild": *"npm run clean"*,  **"build": *"webpack --progress --profile --colors --display-error-details --display-cached"*,**  "buildProd": *"webpack --config webpack.prod.config.js --progress --profile --colors --display-error-details --display-cached"*,  "startWebpackDevServer": *"webpack-dev-server --inline --colors --progress --display-error-details --disaply-modules --display-cached --port 8080"*  } |

By default ***webpack*** reads its own configuration from webpack.config.js where we tell him to package application into two bundles:

|  |  |
| --- | --- |
| dist/bundle.js | Application code packed in this bundle |
| dist/vendor.bundle.js | Angular with all dependencies packed in this vendor bundle |

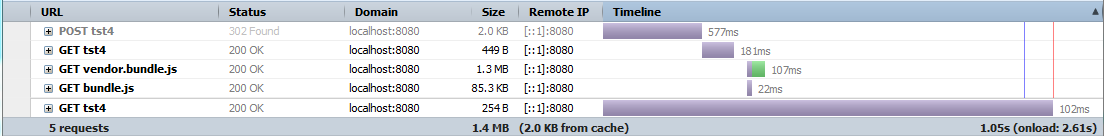
By executing:

MockWebExtJs6\_bsd\src\main\webapp\resources>***npm run build***

We’ll get our application in two small bundles

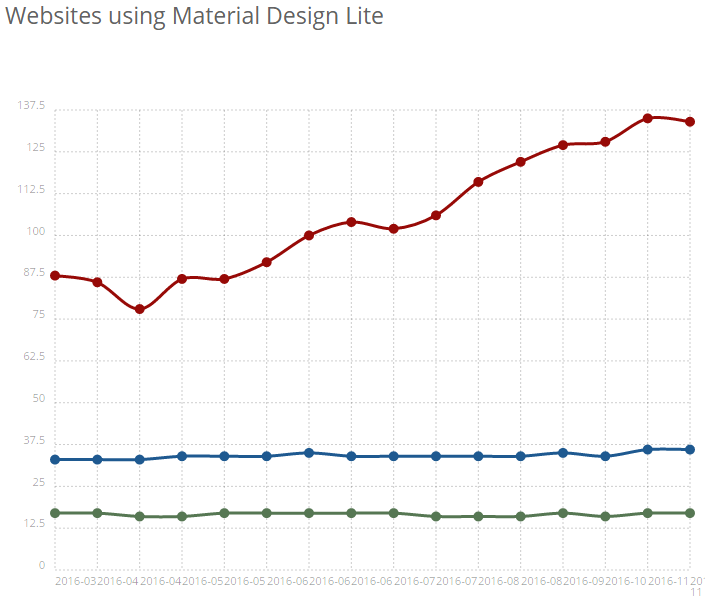
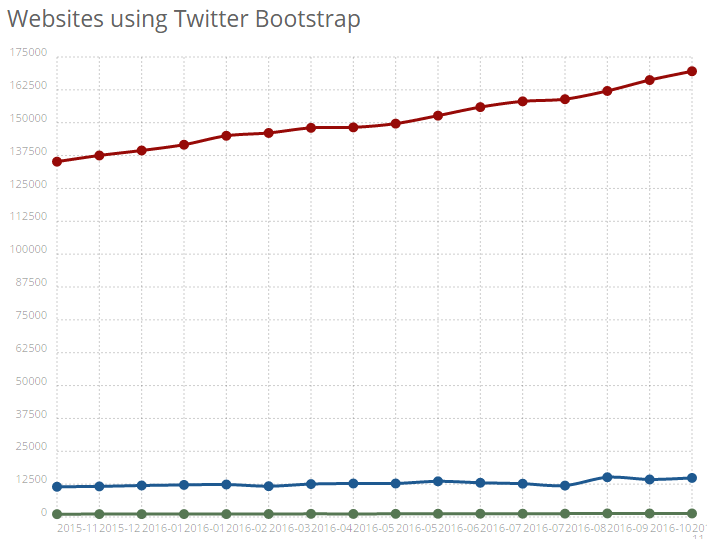


Page load traffic and time looks much better after that:



## Styling Application

* In the beginning of July 2015, Google publicly released a new framework that is easy to use and has no additional dependencies. It’s called **Material Design Lite** (MDL for short).
* We’ll see how developers accept it.
* Google ‘bootstrap vs. mdl’



|  |  |
| --- | --- |
| * **Bootstrap Coverage Totals**   **Top 10k** 1,342 of 10,000 **13.4%**  **Top 100k** 14,862 of 100,000 **14.9%**  **Top Million** 169,571 of 942,811 **18%**  **Entire Internet** 10,066,875 of 365,379,928 **2.8%** | **MDL Coverage Totals**  **Top 10k** 17 of 10,000 **0.2%**  **Top 100k** 36 of 100,000 **<0.1%**  **Top Million** 134 of 942,811 **<0.1%**  **Entire Internet** 3,588 of 365,379,928 **<0.1%** |

## Styling Application with Bootstrap

Google release Material Design Lite. We’ll see how developers accept it. But for now I use Bootstrap.

There many ways to customize bootstrap, but we can narrow them to just three.

* Easiest one is to customize bootstrap distributive online (plenty of resources to choose)
* Override styles in static CSS included after bootstrap core
* Override through Less files

### Customizing Bootstrap Using CSS

This type of customization is generally known as **overriding the default CSS**. We effectively overwrite some of Bootstrap's CSS properties with different values of our own.

In \index.html

|  |
| --- |
| <link rel="stylesheet" type="text/css" href="css/bootstrap.css">  <link rel="stylesheet" type="text/css" href="css/app.css"> |

In \css\app.css we override main CSS styles

|  |
| --- |
| .btn{  border-radius: 0;  padding: 5px 10px;  font-size: 16px;  }  .btn-primary{  background: #63AEF0;  } |

That’s make our button more square and change background color.

### Override through Less files

To customize Bootstrap through less files you need to:

* install bootstrap with sources
* Install Grunt (same as maven open source building tool)
* add your own theme files

|  |  |
| --- | --- |
|  | \less\canon-theme\canon-theme.less  .**content** {  .**make-md-column**(9);  **article** {  **margin-bottom**: 40**px**;  } } .**sidebar** {  .**make-md-column**(3); } .**sidebar-avatar** {  **display**: **block**;  **margin-bottom**: 20**px**;  **max-width**: 100%; } .**sidebar-bio** {  **color**: **@gray**; } |
| \less\canon-theme\canon-variables.less  *//@brand-color: #bada55;* **@navbar-default-color**: **#ffffff**; **@navbar-default-bg**: **#ff0000**; **@navbar-default-border**: **darken**(**@navbar-default-bg**, 96.5%);  *// Navbar links* **@navbar-default-link-color**: **#ececec**; **@navbar-default-link-hover-color**: **#ffffff**; **@navbar-default-link-hover-bg**: **transparent**; **@navbar-default-link-active-color**: **#aaaaaa**; **@navbar-default-link-active-bg**: **lighten**(**@navbar-default-bg**, 50%); **@navbar-default-link-disabled-color**: **#999999**; **@navbar-default-link-disabled-bg**: **transparent**;  **@grid-gutter-width**: 50**px**; |

* build bootstrap distributive ***\bootstrap>grunt dist***

Separate project to make Canon custom bootstrap experimental page. TBD.

|  |  |
| --- | --- |
|  |  |

## Project CVS repository

You may find source code for project I’ve referred in this document in CVS repository:

***mis-cvs-v:/CCI\_SOurceCodes\_Repo/MockExtJs6\_bsd***

## References

1. Single-Page Applications Paradigm. <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>
2. Angular2 home page <http://www.angular2.com/>
3. Components Over Controllers <https://www.codeschool.com/blog/2016/06/29/whats-new-in-angular-2/>
4. Google Material Design Lite <https://getmdl.io/index.html>