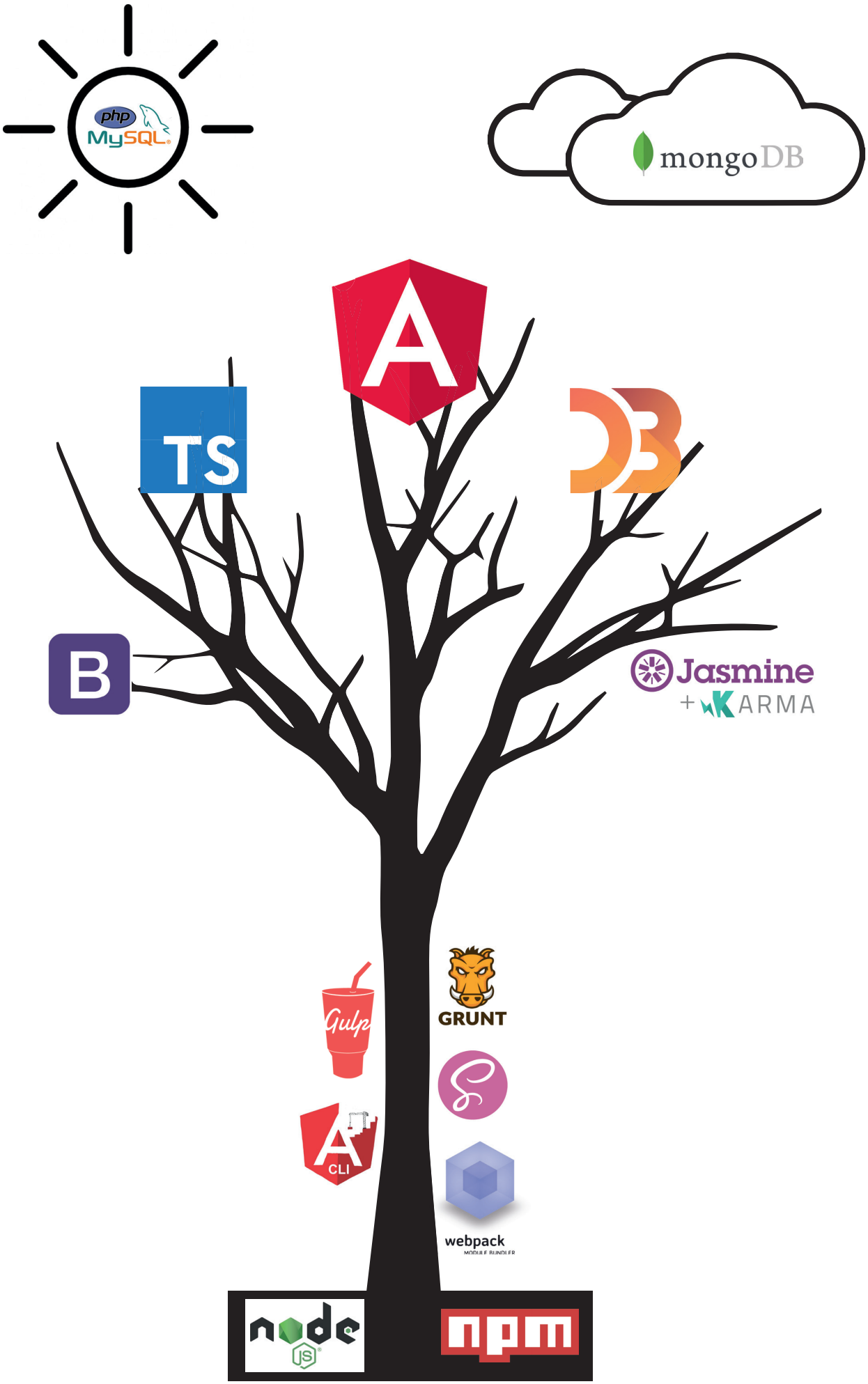
php
MySQL
mongoDB
TS
D3
B
Jasmine
+KARMA
Gulp
GRUNT
CLI
webpack
MODULE BUNDLER
node
JS
npm

Contents

1 Node.js, NPM, NVM

Before we dive further into web technologies we should investigate the following questions:

- What is node.js?
- What is npm?
- What is nvm?
- What are they used for?

In the following sections we are going to answer these questions and investigate them in further detail. Please note that a total insight can't be given in the short time and therefore some informations might be left out.

Note: Further insight to the topics can be gained on their official sites.
`https://nodejs.org/en/`
`https://www.npmjs.com/`
`https://github.com/creationix/nvm`

1.1 Node.js

Question: What is node.js?

JavasScript evolved over the past years from a not so well known language to a new standard in web technologies and web development. For a long time, it was seen as "a little something that would occasionally come in handy to add effects to a webpage". The following figure 1.1 shows the three main stages of the JavaScript evolution.

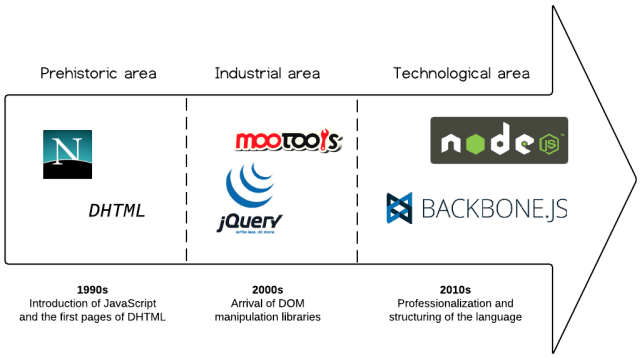

Figure 1.1. Three stages of JavaScript eveolution over the past years.

So the question is: What can we do actually with Node.js in this context? Basically Node.js lets us use **JavaScript language on the server**. So it allows us to write JavaScript outside the browser or a server-side JavaScript.

Until now, JavaScript has always been used on the client's side, i.e. the side seen by visitors browsing our websites. The visitor's browser (Firefox, Chrome, IE, etc.) runs the JavaScript code and performs the actions on the webpage (see figure 1.2). However, Node.js offers a server-side

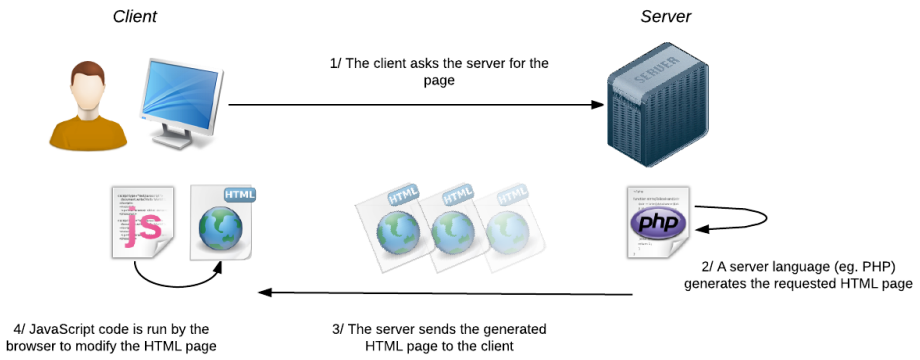

Figure 1.2. Classic flow with PHP on server and JavaScript for the client.

environment, which allows us to also use JavaScript language to generate web pages. Basically, it replaces server languages such as PHP, Java EE, etc. (see figure 1.3). Node.Js is different

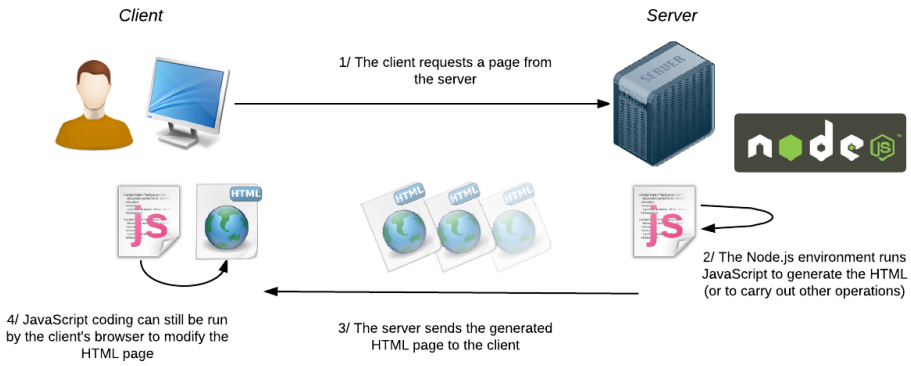

Figure 1.3. With Node.js it's possible to use JavaScript on the server as well.

beacause JavaScript is a language based on events, therefore Node.js is in itself based on events. So, it's the entire way of writing applications that has changed. This is where Node.js draws all its power and speed from.

Note: Node.js is not a framework. Node.js is a very low level environment. In some ways it is more like C than PHP, Ruby on Rails, or Django. This is why it is not really recommended for novices. Please note, however, that there are web-based frameworks, such as Express, which are based on Node.js. They allow us to avoid repetitive tasks, which are imposed by the low level nature of Node.js, but they are still more complex to use than languages such as PHP.

The speed of Node.js is resulting of it's V8 enginge (Google Chrome execution engine) and it's non-blocking feature. It is highly optimized and carries out what we call JIT (Just In Time) compilation. It quickly transforms JavaScript code into machine language and even optimizes. As JavaScript is a language built around the idea of events, Node.js allows an entirely non-blocking code algorithm to be put into place.

1.1.1 Difference between blocking and non-blocking

Blocking model:
Imagine a program which is created to upload a file and then display it. The code could look like this:

```
{
        Upload the file ...
        ... Display the file ...
        ... Do something other
}
```

The actions defined above are carried out in order which will execute them sequentially.

Non-Blocking model:
Now we can write the same code using a non-blocking approach, which could be something like this:

```
{
        Upload the file ...
        ... (As soon as finished) Display the file ...
        ... (then) Do something other
}
```

Now the programm no longer carries out the tasks in the order they are wirtten. It launches the upload of the file to the internet and meanwhile does other things (it follows it's course). As soon as the upload is finished, the pgoram carries out the actions that we asked it to do, e.g. displaying the file or processing it further.

This is exactly how Node.js works. As soon as "File uploaded" appears, a function known as a callback function is called and carries out actions (here, the callback function displays the file).

With real code, using Node.js it would look somehting like this. The following code snippet uploads a file to the internet and displays the `String` "File uploaded!" when it's finished:

```
request('http://www.site.com/file.zip', function (error,
response , body) {
console.log("File uploaded!");
});
console.log("I do other things while I'm waiting...");
```

Here the program launches the request first and then does other things, e.g. the `console.log()` statement. As soon as it has finsihed the program displays the "File uploaded!" message. The function that is called or declared inside the `request` function is an anonymous function or **callback**, which runs after the upload finished.

Note: Since ES5 we use **fat arrow =>** functions as callback mostly.

So one may ask now: "Whats the purpose of this and how does it make the program run faster?" Well the answer can be given by viewing another code example, which will ask to upload 2 files instead of one:

```
const callback = function (error, response, body) {
console.log ("File uploaded!");
});

request ('http://www.site.com/file.zip', callback);
request ('http://www.site.com/otherfile.zip', callback);
```

Imagine if the program would be blocking, the program would have launched the upload of the 1st file and waited for it's finish. Then it would have launched the upload of the 2nd file and waited for it to finish.
However, with Node.js, the two uploads are launched at the same time and the program doesnt wait for the end of the first upload – therefore, the upload of both files goes a lot quicker because the program does them both at the same time.

1.1.2 Conclusion

- Node.js is a development environment, which allows for server-side coding in JavaScript. In this sense, it can be compared to PHP, Python/Django, Ruby on rails, etc.
- Writing an application using Node.js requires a particular frame of mind because everything is based on events.
- Node.js is recognized and appreciated for its speed: a Node.js program never hangs around doing nothing.

Node.js uses a module architecture to simplify the creation of complex applications. Modules are akin to libraries in C, or units in Pascal. Each module contains a set of functions related to the "subject" of the module. For example, the http module contains functions specific to HTTP. Node.js provides a few core modules out of the box to help you access files on the file system, create HTTP and TCP/UDP servers, and perform other useful functions. The `require()` function returns the reference to the specified module.

Sources:
`https://goo.gl/ZPQfZi`
`https://goo.gl/DFcvwV`

1.2 NPM

Question: What is npm?

First and formost the tool is most often refered as **Node Package Manger**, which is tho not the official name directly. If you visit the official page `https://www.npmjs.com/` you will see a funny random generation of names out of the acronym NPM.
Node.js makes it possible to write applications in JavaScript on the server. It's built on the V8 JavaScript runtime and written in C++ — therefore fast. Originally, it was intended as a server environment for applications, but developer's used it to create tools that aid them in local task automation. Since then, a whole new ecosystem of Node-based tools (such as Grunt, Gulp and Webpack) has evolved to transform the face of front-end development, which we will cover in the next section of this document.

To make use of these tools (or packages) in Node.js we need to be able to install and manage them in a useful way. This is where npm, the Node package manager, comes in. It installs the packages you want to use and provides a useful interface to work with them. After you successfully installed Node.js on your machine, you will have also a new global command avaialbe, called `npm`. You can check if you have npm installed by using the following command, which will also show the version of npm `npm --version`

1.2.1 Node Packaged Modules

To install any package from the repository on the web, you just need to type the package name and the install command. However this will install it only for the current project. npm can install packages in local or global mode. In local mode it installs the package in a `node_modules` folder in your parent working directory. This location is owned by the current user. Global packages are installed in `refixprefix/lib/node_modules/` which is owned by root.
Of course it's possible to change the default path of the global installation process, but for this this document refers to various online tutorials. Nevertheless, global packages are the most common way to use npm.

1.2.2 Installing Packages locally

If you want to install packages locally you normally neeed a `package.json` file, if it's not already included in your project. In order to create a fresh new one you go to the command line and type the following, which will require you to insert some variables:

```
$ npm init
package name: (project)
version: (1.0.0)
description: Demo of package.json
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

If you accept the defaults, this will create a `package.json` file with the following look:

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

So afterwards you can begin to install some packages. In order to save them automatically to the `package.json` file, you should use the `--save` or `-s` option. Then anyone who is using your project can install all dependencies by just typing `npm install`.
In order to install for example **Underscore.js**, you type the following command: `npm install unsersoce --save`, which will add the library to the `dependencies` field of your `package.json`:

```
{
  ...
  "dependencies": {
    "underscore": "^1.8.3"
  }
}
```

By far and away the biggest reason for using `package.json` to specify a project's dependencies is portability. For example, when you clone someone else's code, all you have to do is run `npm i` in the project root and npm will resolve and fetch all of the necessary packages for you to run the app. The caret `(^)` at the front of the version number indicates that when installing, npm will pull in the highest version of the package it can find where the only the major version has to match.

1.2.3 Uninstalling Packages locally & Install specific

As npm is a package manager it must be also able to remove a package. For this purpose there is of course an uninstalling command available. Let's assume that the current Underscore package is

causing problems and we want to remove it. If we type the following command: `npm uninstall underscore` This will remove the package from the file and also from the project. However you can reinstall it or a specific version of it by using a specific addition to the normal installation command. For example if you type `npm install underscore@1.8.2` the version 1.8.2 of the package will be installed.

1.2.4 Important Infos

For further instructions and functions please refert to official documentations of npm and further guidelines. Here only a short list of useful commands is given:

- `npm i <package>` - install local package
- `npm i -g <package>` - install global package
- `npm un <package>` - uninstall local package
- `npm up` - update packages
- `npm t` - run tests
- `npm ls` - list installed modules
- `npm ll` or `npm la` - print additional package information while listing modules

Sources:
`https://goo.gl/ddw9QE`

1.3 NVM

Question: What is nvm?

Most node versions are on the gitHub project site or in the readme of the respective project.

As many versions of node are available and various programs run with different versions of node.js the framework needs to be updated either continuously as well as the projects involved or it should be easy to switch between various versions of node. This allows the selection of the perfect node version for each project.
The installation varies between the differnt operateing systems, but there are numerious helpful tutorials or installation guides avaialbe for this purpose.

After the installation is finished, the command `nvm` is available globally on the system. This means, that every command line should know this command. There are numerous features of nvm which aren't described further here now. Only the most important ones are listed as well as their purpose:

- To install a new verison of node e.g. **verison 0.12.7** for a 64-bit system you can use the following command: `nvm install 0.12.7 64`
- In order to see which node versions are installed already, you can use the following command, which will list all node versions on you system: `nvm list`
- After you have multiple versions installed you can use a specific version for your current project. In order to do so, you navigate to the project path and type the following command: `node use 4.2.1 64`, which will use the node **version 4.2.1** for a 64-bist system.

2 Project Setup

This chapter will mostly deal with the right project setup and compilation of all necessary files. The aim is to have a bundle of languages which compiles and runs very easily and let's us view the results fast and quick without trouble. We will also explore the possiblities of various rapid prototyping modules or technologies that help us to set up our project really quick and start to program.

Back in the day, it was enough to concatenate scripts together. Times have changed, though, and now distributing JavaScript code can be a complicated endeavor. This problem has escalated with the rise of single page applications (SPAs). They tend to rely on many different libraries (e.g. Angular, Bootstrap, JQuery, etc.).

For this reason, there are multiple strategies on how to load them. You could load them all at once or consider loading libraries as you need them. Webpack supports many of these sorts of strategies.

The popularity of Node and npm, its package manager, provide more context. Before npm became popular, it was hard to consume dependencies. There was a period when people developed front-tend specific package managers, but npm won in the end. Now dependency management is easier than before, although there are still challenges to overcome.

With the fast growing community and more and more libraries, also more build tools or task runners have emerged. They help us in our daily routine as developer.

Note: The main idea behind a build tool is: **To compile, bundle and minfiy (even uglify) all scripts and stylesheets**.

2.1 Rough criteria

There are some basic criterias that a build tool or task runner should fullfill. There are numerous available out there but only a few are the most popular ones. The following criteria are important for all task runners:

- Transcompiling JavaScript: CoffeeScript, Dart, Babel, Traceur etc.
- JavaScript Transforms: wrapping in modules or ng-annotate etc.
- Bundling/Concatenation: combining of scripts and styles into multiple files
- Minification: scripts, styles and html
- Source Maps: for both scripts and styles
- CSS Preprocessor: Less, Sass, Stylus etc.
- Style Transforms: Autoprefixer, PostCSS etc.

- Cache Busting: renaming files with a hash to prevent incorrect caching
- Image Optimization
- Compiling Templates: Mustache or HandlebarsJS, Underscore, EJS, Jade etc.
- Copying Assets: html, fav icon etc.
- Watching for Changes
- Incremental Rebuild
- Clean Build: deleting all files at start or preferably cleaning up files as needed
- Injecting References: generating script and style tags that reference the bundled files
- Build Configurations: separate Dev, Test and Prod configuration, for example to not minify html in dev build
- Serve: running a development web server
- Running Unit Tests
- Running JavaScript and CSS Linters: jshint, csslint etc.
- Dependencies: handle dependencies on npm and Bower packages, Browserfy etc.

Generally speaking, there are a bunch of "build systems" out there. Most of the build systems in the JavaScript world are actually task runners, which "just" execute several tasks and manage the dependencies between these tasks (i.e. which has to run after which other task). The actual tasks are usually just calls to other libraries (e.g. compiler, minificator, etc.) which do the actual work.

2.2 Differentiation

The terms taks runner or bundle system can be confusing on first sight as they slightly differ from each other. We won't go into much detail but only will have a look at the general overview for each type.

2.2.1 Build systems

The two most popular task runners are Gulp and Grunt. Grunt uses a more configuration like approach whereas Gulp use the code over configuration approach. Some other tools, but not quite as widespread, in this category are Broccoli.js or Brunch. All of these are bringing their own configuration file: gulpfile.js, Gruntfile.js, Brocfile.js and brunch-config.js. In these we configure which resources need to be processed by which plugins and in what kind of files output it bundled. Usually the tools behind these plugins (e.g. the CSS preprocessor LESS) are plain node modules and every build system has its own wrapper around it, making the tool usable within the build system.

2.2.2 JavaScript compilers

Usually one part inside our buildsystem is a compiler of some kind. If we us some language beside JavaScript (like TypeScript or CoffeeScript) we require their compilers to compile that language down to JavaScript, which than can be executed in a browser. There are two popular compilers to compile modern JavaScript to "old" JavaScript (aka compile ECMAScript2015 upwards to ECMAScript5): Babel and Traceur, with the former being a bit more popular und used from my observations. Babel is also used to compile JSX (a special dialect used in the famous React framework) to regular JavaScript.

2.2.3 Other JavaScript Tools

Besides compiling our JavaScript (or other language) to commonly understood JavaScript, there are some other tools, that are widely used in building web applications. **UglifyJS** is a commonly used postprocessor for JavaScript, that will minify our code in size, remove unused and dead code — kind of what ProGuard does in the Java world. **ESLint**, **TSLint** and **JSHint** are linting tools, that will check our code for possible bad (or even buggy) code. They also offer options to enforce coding styleguides in our code.

2.2.4 JavaScript Bundlers

Using modules is a common technique to structure the source code. There are libraries, that can load these modules in the browser, like RequireJS. But often we would like to bundle all modules into a few or a single bundled JavaScript files, so the browser doesn't need to make hundreds of requests to fetch all required modules.

To bundle all those files together there are mainly three famous tools: webpack, SystemJS and Browserify. SystemJS and RequireJS are both hybrid solutions, that can bundle code during building or run in the browser and requesting module files on the fly. From the projects seen in the last years and from GitHub stats, webpack kind of outruns the others.

JavaScript does know several different syntax to import modules, e.g. require('module') called CommonJS or import 'module' from the ECMASCript2015 specification. Those bundlers usually understand more than one syntax how modules can imported. They will get one or multiple input files and from there on include the modules/files, that are imported, into one (or multiple) output bundles.

Since the bundler is traversing the source code while bundeling and knows which files are actually in the output, the compiling will now be done as part of this traversing. That's why e.g. webpack has several so called loaders to use Babel, TypeScript or CoffeeScript for compiling.

Besides compiling the bundlers also have several plugins for pre- and postprocessing the files. The range of plugins bundlers support are growing and so we can also use our CSS preprocessor s part of the bundler, and require the CSS files.

2.2.5 CSS Build Tools

Since the CSS as understood by browser out there is still kind of limited (e.g. older browsers don't have a support for variables, no support for functions, etc.), we will often see LESS or SASS as a richer CSS variant.

Those languages are enhanced CSS variants, which compile down to CSS. There are also tools like autoprefixer which help with better browser support by automatically prefixing our CSS.

Building these files can either be part of our build system, which reads all .less or .scss files, compile them and do the requested postprocessing or as mentioned above. They could also be part of the bundling process, if we decide to import CSS the same way as modules in our JavaScript files.

2.2.6 Generators

Since we have to link up a lot of this tools ourself to create a proper build process, there are generators and yeoman is the de facto standard. Yeoman is a tool, that several people have written templates for, so that setting up a new project means just calling one command on the commandline which will then setup the project structure and create appropriated build files for us.

2.3 Conclusion

To build an up-to-date web project we will need several tools working together. There are numerous tutorials available that help us with the setup and the process. Sometimes less is needed than described here but often enough the build tools and task runners help us in the creation of new tools [1].

Special: lite-server

A special tool shall be mentioned here that helps with rapid prototyping and if we need a server locally. The tool or npm package is called **lite-server**. It's a lightweight development only node server that serves a web app, opens it in the browser, refreshes when html or javascript changes, injects CSS changes using sockets, and has a fallback page when a route is not found.
Source: `https://github.com/johnpapa/lite-server`

[1] Source: https://www.timroes.de/2016/10/03/javascript-tooling-explained/

3 Angular Version 2&4

This chapter will introduce you to Angular 2 or 4 now, which was a major update to Angular [1]. Anyway the functionality is still the same just improved. The chapter will only introduce you to the most relevant information about Angular 4 and will not work as full guide or tutorial. For detailed documentation or tutorials, visit the following links:

- `https://angular.io/`
- `https://www.youtube.com/`
- `https://www.udemy.com/`
- `https://www.lynda.com/`

Of course there are numerous other great websites and tutorials available that help you diving into Angular further more. Angular is a fantastic framework that lets us create powerful desktop or mobile applications. Even cross hybrid development is possible. There are numerous reasons why Angular is great and especially Angular 2 or 4, but here are the most important ones (21 reasons):

1. The tooling support is as good as on Java or .Net platforms.
2. TypeScipt code analyzer warns you about the errors as you type.
3. Using TypeScript classes and interfaces makes the code more concise and easy to read and write.
4. Clean separation between the code that renders UI and the code that implements application logic.
5. The UI doesn't have to be rendered in HTML, and there are already products supporting native UI rendering for iOS and Android.
6. Angular 2/4 offers a simple mechanism for modularizing application with support of lazy loading of modules.
7. The TypeScript compiler generates JavaScript that a human can read.
8. The TypeScript code can be compiled into ES3, ES5, or ES6 versions of JavaScript.
9. The router supports complex navigation scenarios in single-page applications.
10. Dependency injection give you a clean way to implement loose coupling between components and services.
11. Binding and events allows you to create reusable and loosely coupled components.

[1] Sources:
https://yakovfain.com/2016/09/05/twenty-reasons-to-use-angular-2-and-typescript/
https://jaxenter.com/angular-4-top-features-133165.html
http://www.dotnetcurry.com/angular/1385/angular-4-cheat-sheet

12. Each component goes through a well-defined lifecycle, and hooks for intercepting important component events are available for application developers.
13. Automatic (and fast) change detection mechanism spares you from the need to manually force UI updates while giving you a way to fine-tune this process.
14. Angular 2/4 comes with the Rx.js library, which allows you to arrange a subscription-based processing of asynchronous data and eliminates the callback hell (Observables).
15. Support of forms and custom validation is well designed.
16. Unit and integration testing are well supported and you can integrate tests into your building process.
17. The bundling and optimization of the code with Webpack (and its multiple plugins) makes the size of deployed application small.
18. An ability to pre-compile the code eliminates the need to package Angular compiler (not to be confused with TypeScript compiler) with your app, which further minimizes the overhead of the framework.
19. Angular Universal turns your app into HTML in an offline build step, that can be used for server-side rendering, which in turn greatly improves indexing by search engines and SEO.
20. The library of the modern-looking UI components Angular Material 2 offers a number of modern looking components.
21. The scaffolding and deployment tool (Angular CLI) spares developers from writing the boilerplate code and configuration scripts.

There are plenty more reasons to use Angular and especially the newer Angular versions of 2 and 4. For simplicity the further document refers Angular as Angular version 2 and 4, where the similarities between thoose 2 versions are very high. For a detailed comparison and feature list between 2 and 4, please refer to the following.

Note: For a short description of new features for Angular 4, visit the following link `https://jaxenter.com/angular-4-top-features-133165.html`

3.1 Introduction & History

Angular is a framework for building client applications in HTML and JavaScript, or by using a language like TypeScript, which ultimately compiles to JavaScript.

3.1.1 History

So far, three major Angular versions have been released – Angular v1.x (a.k.a AngularJS), Angular 2 and the newly released Angular 4 (also known as Angular). AngularJS has had a few major releases of its own with v1.1, v1.2 and so on, but let us stick with v1.x for all purposes of discussion. Angular v1.x and v2 are quite different in terms of architecture. While Angular v1.x (also known as AngularJS) was based on an MVC architecture, Angular 2 is based on a component/services architecture. Considering Angular was moving from MV* (Model View Whatever) pattern to components focused approach, the framework features were very different from each other and caused

many application developers to rewrite a major portion of their code base. However, Angular v2 to v4 is a very different story. It is a rather progressive enhancement. A majority of changes are non-breaking.

Why not Angular 3?

MonoRepo: Angular 2 has been a single repository, with individual packages downloadable through npm with the @angular/package-name convention. For example @angular/core, @angular/http, @angular/router so on.

Considering this approach, it was important to have a consistent version numbering among various packages. Hence, the Angular team skipped a major version 3. It was to keep up the framework with Angular Router's version. Doing so will help avoid confusions with certain parts of the framework on version 4, while the others on version 3.

3.1.2 Angular 4 Enhancements

Consider the following enhancements in Angular v4:

- This release has considerable improvements in bundle size. Some have reported up to 60% reduction in Angular bundles' file size
- The ngc, AOT compiler for Angular and TypeScript is much faster
- Angular 4 is compatible with TypeScript's newer versions 2.1 and 2.2. TypeScript release helps with better type checking and enhanced IDE features for Visual Studio Code. The changes helped the IDE detect missing imports, removing unused declarations, unintentionally missing "this" operator etc.

There are some other features which we will investigate shortly. They are note worthy.

Router ParamMap:

Starting from version 4, it is possible to query a so-called ParamMap in the router, meaning a request for the route- and queryparameter assigned to a route.

Up until now, route parameters were stored in a simple key-value object structure, therefore being accessible by using the standard JavaScript syntax (parameterObjekt['parameter-name']).

Source Code 3.1. Old Way

```
class MyComponent {
 sessionId: Observable<string>;

 constructor(private route: ActivatedRoute) {}

 ngOnInit() {
   this.sessionId = this.route
     .queryParams
     .map(params => params['session_id'] || 'None');
 }
}
```

Now, the parameters are also available as a map, so you can run them as simple method calls.

Source Code 3.2. New Way

```
class MyComponent {
 sessionId: Observable<string>;

 constructor(private route: ActivatedRoute) {}

 ngOnInit() {
   this.sessionId = this.route
     .queryParamMap
     .map(paramMap => paramMap.get('session_id') || 'None');
 }
}
```

The utilization as a map also brings advantages in terms of type security. The old key-value structure had an unsafe type (type Params = [key : string]: any), whereby the parameter value could take all possible types.

Animations:
Functions necessary for animations up until now were provided as part of @angular/core module, implying that these parts of the code were always included in applications, even if they did not get used in apps without animations. To avoid creating bundles with unnecessary large sizes, this function has been put into its own package. (This isn't just a new feature but also a change making modifications to existing applications necessary if they contain animations.)

Animations are to be provided in the module BrowserAnimationsModule from @angular/platform-browser/animations.

ngIf with "else":
It's quite a frequent thing to use "conditional rendering" in templates to display information depending on some condition. This is done by using *ngIf. If a condition isn't met, the corresponding element and all child elements are not added to the DOM-tree. Many times there was also a need for the opposing case, making it necessary to formulate the same condition just the other way around and add another *ngIf. In Angular 4, this use case can be solved with a newly added else. Maybe unexpected for some, Angular uses a separately referenced template fragment, which in the else-case will be used in place of the element marked with *ngIf.

Source Code 3.3. Old ngIf syntax

```
<div *ngIf="condition">
    <!-- markup here -->
</div>
<div *ngIf="!condition">
     <!-- additional markup here -->
</div>
```

The functionality of *ngIf in the field of reactive programming got improved when interacting with the

async-pipe. It was already possible to “subscribe” (and “unsubscribe”) to asynchronous objects like observables and such from within a template by using the async-Pipe. The new *ngIf-Syntax now makes it possible to also add a local template variable to the result of the if-clause. In the example below the observable, placed inside the variable auth, is resolved by the async-pipe. The result can be used within the template by means of the user variable.

Source Code 3.4. New ngIf-else syntax

```
<div *ngIf="condition; else elseBlock">
    <!-- markup here -->
</div>
<ng-template #elseBlock>
     <div>
         <!-- additional markup here -->
     </div>
</ng-template>
```

Using the new syntax it’s possible to add a component block or template for the else condition, which can be refered. The evaluation of the condition has to be only performed once and not twice like previously, by using the "old way".

Dynamic Components with NgComponentOutlet:
The new *ngComponentOutlet-Directive makes it possible to build dynamic components in a declarative way. Up until now, it has been quite a lot of work to build and produce components dynamically at runtime. It’s not enough to just write some HTML code. Angular needs to be notified about the component and add it to the lifecycle, take care of the data binding and change detection. The old way of using ComponentFactory therefore involved relatively much programming work.

Source Code 3.5. Using the new *ngComponentOutlet

```
@Component({
  selector: ‘app-first-time-visitor’,
  template: ‘
<h1>Welcome to our Page!</h1>

’,
})
export class FirstTimeVisitorComponent {}
@Component({
  selector: ‘app-frequent-visitor’,
  template: ‘
<h1>Welcome Back!</h1>

’,
})
export class FrequentVisitorComponent {}
@Component({
```

```
  selector: ‘app-root',
  template: `

<h1>Hello Angular v4!</h1>

        <ng-container *ngComponentOutlet="welcome"></ng-container>
  `,
})
export class App implements OnInit{
  welcome = FirstTimeVisitorComponent;

  ngOnInit() {
    if(!this.user.isfirstVisit){
      this.alert = FrequentVisitorComponent;
    }
  }
}
```

In this example, either FirstTimeVisitorComponent or FrequentVisitorComponent will be displayed as a greeting to visitors, depending on whether they are accessing the page for the first time or have visited it before. The check for prior visits is conducted in the OnInit-Lifecycle-Hook; based on the result, different components are handed over to the template for display.

TypeScript 2.1/2.2:
Type security of Angular applications and the speed of ngc-Compiler have been improved – thanks to the official support for the most recent TypeScript versions.

StrictNullChecks:
Sadly, some parts of the improved Type Checks could not be included in Angular 4 [for now] because some incompatibilities were found in the RC phase. There are plans to include this feature in v4.1, though.

Angular Universal:
With Angular Universal, it’s possible to render Angular applications outside of the browser, for instance directly on the web server. With that, JavaScript is no longer necessary for initially rendering the page content, so websites can be optimized better for search engines. Another use case is the utilization of WebWorker Threads to render content outside the GUI Thread. This rendered content can simply be added to the DOM Tree for display later.

Smaller changes:

- New Pipe: Title Case - Changes first letter of each word to upper case.
- Forms get assigned "novalidate" automatically.
- Sourcemaps for templates.
- New View Engine for phenomenal speed.

3.2 Angular setup and Start

There are many methods that can be used to stand up a new Angular project, along with an even larger number of technologies to pair with it [2]. The following section will show 2 approaches, where the first is the better way, by using the Angular CLI (Command Line Interface) and the second is the "manual" approach using webpack.

3.2.1 Angular CLI

Angular CLI is a preferred way to get started with an Angular project (v2.x and above). It not only saves time, but also makes it easy to maintain the code base during the course of the project, with features to add additional components, services, routing etc. Refer to the appendix at the end of this article for an introduction to Angular CLI.

We can download the Angular CLI at `https://cli.angular.io/`. Since the CLI project is available as an NPM package, we can install angular-cli globally on our system by running the following command:

```
npm install -g angular-cli@latest
```

However if you already have Angular CLI installed and want to upgrade to the latest version of Angular CLI, run the following commands:

```
npm uninstall -g angular-cli
npm cache clean
npm install -g angular-cli@latest
```

To create a new project with Angular CLI, run the following command:

```
ng new TestProject
```

TestProject is the name of the new project. We may optionally use `--routing` parameter to add routing to the Angular project. The latest version of Angular CLI (v1.0) makes scaffolding possible with Angular 4, which lets us directly point at a database and make simple functions possible like "create, read, update, remove".

Note: caffolding makes it possible to add new components, routes or services etc. from the command line.

Adding Bootstrap

We can install Bootstrap by using the following command for our "TestProject" (please note we are using here an early alpha version, which may have been updated already):

```
npm install --save bootstrap@4.0.0-alpha.6
```

Once the package is downloaded, we need to add Bootstrap references in .angular-cli.json. We modify the styles configuration to add Bootstrap CSS:

[2] Sources:
http://1904labs.com/2017/05/06/1399/

```
styles": [
    "../node_modules/bootstrap/dist/css/bootstrap.min.css",
    "styles.css"
],
```

We can also modify the scripts configuration to add jQuery, Bootstrap and Tether JS files for example:

```
"scripts": [
    "../node_modules/jquery/dist/jquery.min.js",
    "../node_modules/tether/dist/js/tether.min.js",
    "../node_modules/bootstrap/dist/js/bootstrap.min.js"
],
```

Pros & Cons

The easiest, and quickest by far, way to setup a new Angular project is to simply use the Angular CLI (developed by the Angular team itself). With the CLI you can have a new Webpack-based project set up and running within minutes with almost zero manual configuration required.

Pros:

- Extremely quick and easy to setup, ideal for beginners
- Webpack, Protractor e2e testing, Karma/Jasmine testing setup automatically
- Dependencies pulled in automatically
- Centralized configuration via an Angular CLI config file
- Quick and easy CLI commands to run the app, create new Components, etc...

Cons:

- Less advanced configuration (cannot change advanced Webpackconfig, etc...)
- More rigid configuration structure (harder to move config files to desired locations, they're more strewn about inside of various different directories)
- Less or no support for various addons (HTML template engines like PugJS, etc...)

According to the Angular CLI team, Node 6.9.0 or higher, together with NPM 3 or higher, are required to run both the CLI and the project generated by the CLI.

Running the app

Running the app locally is equally simple. We can either use `npm start`, or `ng serve` in our command window to serve up the local development build. Then the application will be available on the specified port and localhost.

3.2.2 Manual Setup

For a manual setup, the pros and cons are effectively the exact opposite of the pros and cons for using the Angular CLI, so we'll just go over a few of the main benefits:

- Advanced Webpack configuration

- Ability to use HTML template engines (due to advanced Webpackconfig)
- Cleaner config file and directory structure
- Greater flexibility (also due to advanced config)

o start, checkout the seed project on Github and fork/clone the repository: `https://github.com/Plum-Crazy/angular-seed`
Once that's done, open up your preferred command window and run an npm install on the project's root directory (same level as the package.json file). NPM should go through and install all of the required packages to use the project.

3.3 Visual Studio Code IDE

Visual Studio Code, an open source IDE from Microsoft, has good integration with TypeScript and Angular. With the new features in TypeScript 2.2, this IDE can detect missing and unused imports, missing this operator and also allows for quick refactoring.

The Visual Studio Code extensions' eco system provides many useful features.

Angular 4 TypeScript Snippets is one such extension. It's a collection of snippets readily available for creating a component, service, pipe, module etc. Start typing "a-" in VS Code, and it shows list of available snippets.

Imagine we chose to create a component. Tab through to provide component selector, template file name and component class name etc. The Component skeleton file is ready to use. This is an alternative to using Angular CLI for creating a new component.

Here you can find a productivity guide for VSCode which helps you with the most usful commands: `http://www.dotnetcurry.com/visualstudio/1340/visual-studio-code-tutorial`

3.4 Conclusion

Angular went through a good amount of transition from Angular 1.x, MV* model, to the framework we know today. The purpose of the transition is to effectively support new features in JavaScript, as well as sync up with the latest web development standards. When used with TypeScript, it is on steroids with support for types, integration with IDE like Visual Studio Code and many other useful features. In the process, Angular 1 to 2 upgrade included many breaking changes. However, upgrading to future versions of Angular are expected to be smooth with minimal or no breaking changes. The Angular framework will continue to evolve to support more features and make the developer's job easy. [3]

[3] Sources:
http://www.dotnetcurry.com/angularjs/1366/angular-4-app-typescript-bootstrap

4 Angular Cheat Sheet

This chapter is a consumption of various things that are most common in Angular and can easily be cheat sheeted. Of course there are design patterns and further things to note, but this chapter is dedicated only as pure simple cheat sheet, that can be printed out easily.

4.1 Components

Components are building blocks of an Angular application. A component is a combination of HTML template and a TypeScript (or a JavaScript) class. To create a component in Angular with TypeScript, we create a class and decorate it with the Component decorator.

Consider the following example:

```
import { Component } from '@angular/core';
@Component({
  selector: 'hello-ng-world',
  template: `<h1>Hello Angular world</h1>`
})
export class HelloWorld {
}
```

The Component is imported from the core angular package. The component decorator allows specifying metadata for the given component. While there are many metadata fields that we can use, here are some important ones:

- **selector:** is the name given to identify a component. In the sample above, hello-ng-world is used to refer to the component in another template or HTML code.
- **template:** is the markup for the given component. While the component is utilized, bindings with variables in the component class, styling and presentation logic are applied.
- **templateUrl:** is the url to an external file containing a template for the view. For better code organization, template could be moved to a separate file and the path could be specified as a value for templateUrl.
- **styles:** is used to specific styles for the given component. They are scoped to the component.
- **styleUrls:** defines the CSS files containing the style for the component. We can specify one or more CSS files in an array. From these CSS files, classes and other styles could be applied in the template for the component.

4.2 Data-binding

String interpolation is an easy way to show data in an Angular application. Consider the following example in an Angular template to show the value of a variable:

```
import { Component } from '@angular/core';
@Component({
  selector: 'hello-ng-world',
  template: `<h1>Hello {{title}} world</h1>`
})
export class HelloWorld {
title = 'Angular 4';
}
```

Where title is the variable name. Notice single quotes (backtick ‘) around the string which is the ES6/ES2015 syntax for mixing variables in a string, called **Template Strings**.
To show the value in a text field, we can use the DOM property “value” wrapped in square brackets. As it uses a DOM attribute, it’s natural and easy to learn for anyone who is new to Angular. “title” is the name of the variable in the component.

```
<input type="text" [value]="title">
```

This will bind the title attribute to the input. Anyway if we want to update our model and view directly we can use ngModle, which we will see later.

4.3 Events

To bind a DOM event with a function in a component, we use the following syntax. Wrap the DOM event in circular parenthesis, which invokes a function in the component.

```
<button (click)="updateTime()">Update Time</button>
```

4.3.1 Accepting user input

As the user keys-in values in text fields or makes selections on a dropdown or a radio button, values need to be updated on component/class variables. We may use events to achieve this. The following snippet updates value in a text field, to a variable on the component:

```
<!- Change event triggers function updateValue on the component -->
<input type="text" (change)="updateValue($event)">

  updateValue(event: Event){
    // event.target.value has the value on the text field.
    // It is set to the label.
    this.label = event.target.value;
  }
```

Note: Considering it's a basic example, the event parameter to the function above is of type any. For better type checking, it's advisable to declare it of type KeyboardEvent.

4.3.2 Accepting user input - Better

In the Accepting user input sample, `$event` is exposing a lot of information to the function/component. It encapsulates the complete DOM event triggered originally. However, all the function needs is the value of the text field. Consider the following piece of code, which is a refined implementation of the same. Use template reference variable on the text field:

```
<input type="text" #label1 (change)="updateValue(label1.value)">
```

Notice updateValue function on change, which accepts value field on label1 (template reference variable). The update function can now set the value to a class variable.

```
updateValue(value: any){
// It is set to the label.
this.label = value;
}
```

4.3.3 Banana in a box

From Angular 2 onwards, two way data binding is not implicit. Consider the following sample. As the user changes value in the text field, it's doesn't instantly update the title in h1:

```
<h1> {{title}} </h1>
<input type="text" [value]="title">
```

se the following syntax for two-way data binding. It combines value binding and event binding with the short form – `[()]`. On a lighter note, it's called banana in a box.

```
<h1> {{title}} </h1>
<input type="text" [(ngModel)]="title" name="title">
```

4.4 ngModel & form fields

The ngModel is not only useful for two-way data binding, but also with certain additional CSS classes indicating state of the form field. Consider the following form fields - first name and last name. ngModel is set to fname and lname fields on the view model (the variable vm defined in the component):

```
<input type="text" [(ngModel)]="vm.fname" name="firstName" #fname required />
{{fname.className}} <br />
<input type="text" [(ngModel)]="vm.lname" name="lastName" #lname /> {{lname.className}}
```

Note: If ngModel is used within a form, it's required to add a name attribute. The control is registered with the form (parent using the name attribute value. Not providing a name attribute will result in the following error.

It adds the following CSS classes as and when the user starts to use the form input elements:

- **ng-untouched** – The CSS class will be set on the form field as the page loads. User hasn't used the field and didn't set keyboard focus on it yet.
- **ng-pristine** – This class is set as long as value on the field hasn't changed.
- **ng-dirty** – This class is set as user modifies the value.
- **ng-valid** – This class is set when all form field validations are satisfied, none failing. For example a required field has a value.
- **ng-invalid** – This class is set when the form field validations are failing. For example a required field doesn't have a value.

This feature allows customizing CSS class based on a scenario.

4.5

ngModule We can create an Angular module by using the ngModule decorator function on a class. A module helps package Angular artifacts like components, directives, pipes etc. It helps with ahead-of-time (AoT) compilation. A module exposes the components, directives and pipes to other Angular modules. It also allows specifying dependency on other Angular modules. Consider the following code example:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MyAppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ MyAppComponent ],
  bootstrap:    [ MyAppComponent ]
})
export class MyAppModule { }
```

We have to import ngModule from Angular core package. It is a decorator function that can be applied on a class. Following is the metadata we provide to create a module. The following options are the most common used in a module:

- **imports:** Dependency on BrowserModule is specified with imports. Notice it's an array. Multiple modules' dependency could be specified with imports.
- **declarations:** All components associated with the current module could be specified as values in declarations array. Basically these are the view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.
- **bootstrap:** Each Angular application needs to have a root module which will have a root component that is rendered on index.html. For the current module, MyAppComponent (imported from a relative path) is the first component to load.
- **providers:** Is an array to make services and values known to components. They are added

to the root scope and injected to other services or directives that have them as dependency.

- **exports:** makes the components, directives, and pipes available in modules that add this module to imports.

4.6 Service

We create a service in an Angular application for any reusable piece of code. It could be accessing a server side service API, providing configuration information etc. To create a service, we import and decorate a class with @injectable (TypeScript) from @angular/core module. This allows angular to create an instance and inject the given service in a component or another service. Let's consider the following example. It creates a TimeService for providing date time values in a consistent format across the application. The `getTime()` function could be called in a component within the application:

```
import { Injectable } from '@angular/core';

@Injectable()
export class TimeService {

  constructor() { }

  getTime(){
    return `${new Date().getHours()} : ${new Date().getMinutes()} :
    ${new Date().getSeconds()}`;
  }
}
```

4.6.1 Dependency Injection (DI)

In the service section, we have seen creating a service by decorating it with Injectable() function. It helps angular injector create an object of the service. It is required only if the service has other dependencies injected. However, it's a good practice to add Injectable() decorator on all services (even the ones without dependencies) to keep the code consistent.

With DI, Angular creates an instance of the service, which offloads object creation, allows implementing singleton easily, and makes the code conducive for unit testing.

4.6.2 Providing a service

To inject the service in a component, specify the service in a list of a providers. It could be done at the module level (to be accessible largely) or at component level (to limit the scope). The service is instantiated by Angular at this point.

Here's a component sample where the injected service instance is available to the component and all its child components:

```
import { Component } from '@angular/core';
import { TimeService } from './time.service';
@Component({
  selector: 'app-root',
  providers: [TimeService],
  template: `<div>Date: {{timeValue}}</div>,
})
export class SampleComponent {
  // Component definition
}
```

To complete the DI process, we need to specify the service as a parameter property in the constructor:

```
export class SampleComponent {
  timeValue: string = this.time.getTime(); // Template shows

  constructor(private time: TimeService){
  }
}
```

While the service implementatino looks like this:

```
import { Injectable } from '@angular/core';

@Injectable()
export class TimeService {

  constructor() { }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}`;
  }
}
```

4.6.3 Provide a service at module level

When the TimeService is provided at module level (instead of the component level), the service instance is available across the module (and application). It works like singleton across the application:

```
@NgModule({
  declarations: [
    AppComponent,
    SampleComponent
  ],
  imports: [
```

```
    BrowserModule
  ],
  providers: [TimeService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

4.6.4 Alternate Syntax

In the above sample, the provider statement is a short cut to the following:

```
// instead of providers: [TimeService] you may use the following,
  providers: [{provide: TimeService, useClass: TimeService}]
```

It allows using a specialized or alternative implementation of the class (TimeService in this example). The new class could be a derived class or the one with similar function signatures to the original class:

```
providers: [{provide: TimeService, useClass: AlternateTimeService}]
```

For the sake of an example, the AlternateTimeService provides date and time value. Original TimeService provided just the date value:

```
@Injectable()
export class AlternateTimeService extends TimeService {

  constructor() {
    super();
   }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}
                  ${dateObj.getHours()}:${dateObj.getMinutes()}`;
  }
}
```

Note: When the alternate method is used to provide a service, the Angular injector creates a new instance of the service.

To rather use an existing instance of the service, use useExisting instead of useClass:

```
providers: [AlternateTimeService, {provide: TimeService, useExisting: AlternateTimeService}]
```

4.6.5 Provide an Interface & it's Implementation

It might be a good idea to provide an interface and implementation as a class or a value. Let's consider the following interface:

```
interface Time{
    getTime(): string
```

```
}
export default Time;
```

This interface could be implemented for example by a TimeService:

```
@Injectable()
export class TimeService implements Time {
  constructor() { }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}`;
  }
}
```

At this point, we can't implement Time interface and TimeService class like the above sample. The following piece of code does not work, because a TypeScript interface doesn't compile to any equivalent in JavaScript:

```
providers: [{provide: Time, useClass: TimeService}] // WRONG
```

To make this work, we need to import Inject (Decorator) and InjectionToken from @angular/core:

```
// create an object of InjectionToken that confines to interface Time
let Time_Service = new InjectionToken<Time>('Time_Service');

// Provide the injector token with interface implementation.
providers: [{provide: Time_Service, useClass: TimeService}]

// inject the token with @inject decorator
constructor(@Inject(Time_Service) ts,) {
      this.timeService = ts;
}

// We can now use this.timeService.getTime().
```

4.6.6 Provide values

We may not always need a class to be provided as a service. The following syntax allows us to provide a JSON object. Notice the JSON object has the function getTime(), which could be used in components:

```
providers: [{provide: TimeService, useValue: {
getTime: () => `${dateObj.getDay()} - ${dateObj.getMonth()} - ${dateObj.getFullYear()}`
  }}]
```

Note: For an implementation similar to the section "Provide an Interface and its implementation", provide with useValue instead of useClass. The rest of the implementation stays the same.

```
providers: [provide: Time_Service, useValue: {
    getTime: () => 'A date value' }}]
```

4.7 Directives

From Angular 2 onwards, directives are broadly categorized as following:

- **Components** - Includes a template. They are the primary building blocks of an Angular application.
- **Structural directives** – A directive for managing layout. It is added as an attribute on the element and controls flow in DOM. Example NgFor, NgIf etc.
- **Attribute directives** – Adds dynamic behavior and styling to an element in the template. Example NgStyle.

4.7.1 ng-if ... else

A structural directive to show a template conditionally. Consider the following sample. The ngIf directive guards against showing half a string Time: [with no value] when the title is empty. The else condition shows a template when no value is set for the title:

```
<div *ngIf="title; else noTitle">
  Time: {{title}}
</div>

<ng-template #noTitle> Click on the button to see time. </ng-template>
```

As and when the title value is available, Time: [value] is shown. The #noTitle template hides, as it doesn't run else.

4.7.2 ng-template

Ng-Template is a structural directive that doesn't show by default. It helps group content under an element. By default, the template renders as a HTML comment:

```
<div *ngIf="isTrue; then tmplWhenTrue else tmplWhenFalse"></div>
<ng-template #tmplWhenTrue >I show-up when isTrue is true. </ng-template>
<ng-template #tmplWhenFalse > I show-up when isTrue is false </ng-template>
```

Note: Instead of ng-if...else (as in ng-if...else section), for better code readability, we can use if...then...else. Here, the element is shown when the condition is true and also moved in a template.

4.7.3 ng-container

Ng-container is another directive/component to group HTML elements. We may group elements with a tag like div or span. However, in many applications there could be default styles applied on these elements. To be more predictable, Ng-Container is preferred. It groups elements, but doesn't render itself as a HTML tag. Consider following example:

```
// Consider value of title is Angular
Welcome <div *ngIf="title">to <i>the</i> {{title}} world.</div>
```

It will not render in one line due to the div. We may change the behavior with CSS. However, we may have a styling applied for div by default. That might result in unexpected styling to the string within the div.

Now instead, consider using ng-container:

```
Welcome <ng-container *ngIf="title">to <i>the</i> {{title}} world.</ng-container>
```

It will be now rendered in one line and you can notice ng-container didn't show up as an element.

4.7.4 ngSwitch and ngSwitchCase

We can use switch case statement in Angular templates. It's similar to a switch..case statement in JavaScript. Let's consider the following snippet. `isMetric` is a variable on the component. If its value is true, it will show Degree Celsicus as the label, else it will show Fahrenheit.

Note: ngSwitch is an attribute directive and ngSwitchCase is a structural directive.

```
<div [ngSwitch]="isMetric">
  <div *ngSwitchCase="true">Degree Celsius</div>
  <div *ngSwitchCase="false">Fahrenheit</div>
</div>
```

Please note, we may use ngSwitchDefault directive to show a default element when none of the values in the switch. . . case are true:

```
<div [ngSwitch]="isMetric">
  <div *ngSwitchCase="true">Degree Celsius</div>
  <div *ngSwitchDefault>Fahrenheit</div>
</div>
```

4.7.5 Input decorator function

A class variable could be configured as an input to the directive. The value will be provided by component using the directive. Consider the following code snippet. It is a component that shows login fields. (A component is a type of directive). The Component invoking login component could set `showRegister` to true, resulting in showing a register button. To make a class variable input, annotate it with the Input decorator function.

```
Import Input()
import { Component, OnInit, Input } from '@angular/core';
...
@Input() showRegister: boolean;
```

Now we can use the input value in a component for example:

```
<div>
  <input type="text" placeholder="User Id" />
```

```
  <input type="password" placeholder="Password" />
  <span *ngIf="showRegister"><button>Register</button></span>
  <button>Go</button>
</div>
```

We could provide a different name to the attribute than that of the variable. Consider the following snippet:

```
@Input("should-show-register") showRegister: boolean;
```

Now, use the attribute should-show-register instead of the variable name showRegister:

```
<app-login should-show-register="true"></app-login>
```

4.7.6 Output decorator function

Events emitted by a directive are output to the component (or a directive) using the given directive. In the login example, on clicking login, an event could be emitted with user id and password.

Consider the following snippet. We have to import the Output decorator function and EventEmitter and declare an Output event of type EventEmitter:

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
...
@Output() onLogin: EventEmitter<{userId: string, password: string}>;
```

Notice the generic type (anonymous) on EventEmitter for onLogin. It is expected to emit an object with user id and password. Then we initialize the object and declare the event handler:

```
constructor() {
    this.onLogin = new EventEmitter();
}

loginClicked(userId, password){
        this.onLogin.next({userId, password});
}
```

The component emits the event. In the sample, when user clicks on login, it emits the event with a next function. Here's the template:

```
<button (click)="loginClicked(userId.value, password.value)">Go</button>
```

While we are using the component, we specify a login handler for the onLogin output event:

```
<app-login (onLogin)="loginHandler($event)"></app-login>
```

The Login handler receives user id and password from the login component:

```
loginHandler(event){
    console.log(event);
    // Perform login action.
  }

// Output: Object {userId: "sampleUser", password: "samplePassword"}
```

Similar to the input decorator, output decorator could specify an event name to be exposed for consuming component/directive:

```
@Output("login") onLogin: EventEmitter<{userId: string, password: string}>;
```

4.8 Change Detection Strategy

Angular propagates changes top-down, from parent components to child components. Each component in Angular has an equivalent change detection class. As the component model is updated, it compares previous and new values. Then changes to the model are updated in the DOM. For component inputs like number and string, the variables are immutable. As the value changes and the change detection class flags the change, the DOM is updated. For object types, the comparison could be one of the following:

Shallow Comparison: Compare object references. If one or more fields in the object are updated, the object reference doesn't change. Shallow check will not notice the change. However, this approach works best for immutable object types. That means, objects cannot be modified. Changes need to be handled by creating a new instance of the object.

Deep Comparison: Iterate through each field on the object and compare it with the previous value. This identifies change in the mutable object as well. That means, if one of the fields on the object is updated, the change is noticed.

4.8.1 Change detection on a component

On a component, we can annotate with one of the two change detection strategies.

ChangeDetectionStrategy.Default: As the name indicates, it's the default strategy when nothing is explicitly annotated on a component. With this strategy, if the component accepts an object type as an input, it performs deep comparison every time there is a change. That is, if one of the fields have changed, it will iterate through all the fields and identify the change. It will then update the DOM. Consider the following example:

```
import { Component, OnInit, Input, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-child',
  template: '<h2>{{values.title}}</h2> <h2>{{values.description}}</h2>',
  styleUrls: ['./child.component.css'],
  changeDetection: ChangeDetectionStrategy.Default
})
export class ChildComponent implements OnInit {

// Input is an object type.
  @Input() values: {
    title: string;
    description: string;
```

```
  }

  constructor() { }

  ngOnInit() {}
}
```

Notice, the change detection strategy is mentioned as ChangeDetectionStrategy.Default, it is the value set by default in any component. The input to the component is an object type with two fields title and description. We are updating the values object. The Object reference doesn't change. However the child component updates the DOM as the strategy performs deep comparison.

ChangeDetectionStrategy.OnPush: The default strategy is effective for identifying changes. However, it's not performant as it has to loop through a complete object to identify change. For better performance with change detection, consider using OnPush strategy. It performs a shallow comparison of the input object with the previous object. That means, it compares only the object reference. The code snippet before will not work with the OnPush strategy. In the sample, the reference doesn't change as we modify values on the object. The Child component will not update the DOM.

When we change the strategy to OnPush on a component, the input object needs to be immutable. We should create a new instance of input object, every time there is a change. This changes the object reference and hence the comparison identifies the change. Consider following snippet for handling change event in the parent:

```
ON PUSH:
updateValues(param1, param2){
  this.values = { // create a new object for each change.
    title: param1,
    description: param2
  }
}

ON DEFAULT:
updateValues(param1, param2){
    this.values.title = param1;
    this.values.description = param2;
}
```

Note: Even though the above solution for updateValues event handler might work okay with the OnPush strategy, it's advisable to use immutable.js implementation for enforcing immutability with objects or observable objects using RxJS or any other observable library.

4.9 Transclusion in Angular

In the above sections (input decorator and output decorator), we have seen how to use component attributes for input and output. How about accessing content within the element, between begin and end tags?

AngularJS 1.x had transclusion in directives that allowed content within the element to render as part of the directive. For example, an element <blue></blue> can have elements, expressions and text within the element. The component will apply blue background color and show the content:

```
<blue>sample text</blue>
```

In the blue component's template, we use ng-content to access content within the element. Consider the following. The CSS class blue applies styling:

```
<div class="blue">
  <ng-content></ng-content>
</div>
```

The `ng-content` shows "sample text" from the above example. It may contain more elements, data binding expressions etc.

4.10 Observables in Angular templates

Observable data by definition, may not be available while rendering the template. The `*ngIf` directive could be used to conditionally render a section. Consider the following example:

```
<div *ngIf="asyncData | async; else loading; let title">
  Title: {{title}}
</div>
<ng-template #loading> Loading... </ng-template>
```

Notice the async pipe. The sample above checks for asyncData observable to return with data. When observable doesn't have data, it renders the template loading. When the observable returns data, the value is set on a variable title. The async pipe works the same way with promises as well.

NgFor We can use the `*ngFor` directive to iterate through an array or an observable. The following code iterates through a colors array and each item in the array is referred to as a new variable color:

```
<ul *ngFor="let color of colors">
  <li>{{color}}</li>
</ul>

/* component declares array as  colors= ["red", "blue", "green", "yellow", "violet"]; */
```

Note: Use async pipe if colors is an observable or a promise.

4.11 Strict Null Check

TypeScript 4 introduced strict null check for better type checking. TypeScript (& JavaScript) have special types namely `null` and `undefined`. With strict type check enabled for an Angular application, `null` or `undefined` cannot be assigned to a variable unless they are of that type. Let's consider the following example:

```
let firstName: string, lastName: string;
//returns an error, Type 'null' is not assignable to type 'string'.
firstName = null;
// returns an error, Type 'undefined' is not assignable to type 'string'.
lastName = undefined;
```

Now explicitly specify null and undefined types. Then revisit the variable declaration as following:

```
let firstName: string | null, lastName: string | undefined;
// This will work
firstName = null;
lastName = undefined;
```

By default, strictNullCheck is disabled. To enable strictNullCheck edit `tsconfig.json`. Add `"strictNullChecks": true` in compilerOptions.

4.12 HTTP API calls and Observables

We use the built-in Http service from '@angular/http' module to make server side API calls. Inject the Http service into a component or another service. Consider this snippet which makes a GET call to a Wikipedia URL. Here http is an object of Http in @angular/http:

```
this.http
      .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json");
```

The `get()` function returns an observable. It helps return data asynchronously to a callback function. Unlike promise, an observable supports a stream of data. It's not closed as soon as data is returned. Furthermore it can be returned till the observable is explicitly closed. Observables support multiple operators or chaining of operators. Consider the map operator below which extracts a JSON out of the http response.

```
this.http
      .get("http://localhost:3000/dino")
      .map((response) => response.json());
```

Note: We have to import the operations from the rxjs library before we can use it. We normally do this in a rxjs-operators.ts file.

An observable has a subscribe function, which accepts three callback:

- Success callback that has data input
- Error callback that has error input
- Complete callback, which is called while finishing with the observable

Let's consider der following example:

```
his.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json())
  .subscribe((data) => console.log(data), // success
    (error) => console.error(error), // failure
```

```
    () => console.info("done")); // done
```

We may set the observable result to a class/component variable and display it in the template. A better option would be to use async pipe in the template:

```
dino: Observable<any>; // class level variable declaration

this.dino = this.http // Set returned observable to a class variable
      .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
      .map((response) => response.json());
```

In the template we now use async pipe. When the dino object is available, we use fields on the object:

```
<div>{{ (dino |async)?.bruhathkayosaurus.appeared }}</div>
```

4.13 Promises

We may use an RxJS operator to convert the observable to a promise. If an API currently returns a promise, for backward compatibility, it's a useful operator. Import `toPromise` operator and use it on an observable:

```
import 'rxjs/add/operator/toPromise';
...
getData(): Promise<any>{
  return this.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json())
  .toPromise();
}
```

Then we can use the data returned on the success callback. We set it to a class variable to be used in the template.

```
this.getData()
  .then((data) =>  this.dino = data)
  .catch((error) => console.error(error));
```

Then we can use it inside a template:

```
<div *ngIf="dino">
        <div>Bruhathkayosaurus appeared
    {{ dino.bruhathkayosaurus.appeared }} years ago</div>
</div>
```

4.14 Router

Routing makes it possible to build an SPA (Single Page Application). Using the router configuration, components are mapped to a URL. To get started with an Angular application that supports routing

using Angular CLI, we run the following command:

```
ng new sample-application --routing
```

To add a module with routing to an existing application using Angular CLI, run the following command:

```
ng g module my-router-module --routing
```

4.14.1 Router outlet

Components at the route (URL) are rendered below router outlet. In general, RouterOutlet is placed in the root component for routing to work:

```
<router-outlet></router-outlet>
```

4.14.2 Route configuration

We have to create a routes object of type Routes (in@angular/router package) which is an array of routes JSON objects. Each object may have one or more of the following fields:

- **path:** Specifies path to match
- **component:** At the given path, the given component will load below router-outlet
- **redirectTo:** Redirects to the given path, when path matches. For example, it could redirect to home, when no path is provided.
- **pathMatch:** It allows configuring path match strategy. When the given value is full, the complete path needs to match. Whereas prefix allows matching initial string. Prefix is the default value.

Consider the following route configuration for example:

```
const routes: Routes = [
  {
    path: 'home',
    component: Sample1Component,
  },
  {
    path: 'second2',
    component: Sample2Component
  },
  {
    path: '',
    redirectTo: '/home',
    pathMatch: 'full'
  }
];
```

4.14.3 Child Routes

For configuring child routes, we use children within the route object. The child component shows-up at the router-outlet in the given component. The following sample renders Sample1Component. It has a router-outlet in the template. Sample2Component renders below it:

```
const routes: Routes = [
  {
    path: 'home',
    component: Sample1Component,
    children: [
      {
        path: 'second',
        component: Sample2Component
      }
    ]
  }... // rest of the configuration.

// Template for Sample1Component

<div>
  sample-1 works!
  <router-outlet></router-outlet> <!--Sample2Component renders below it -->
</div>
```

4.14.4 Params

Data can be exchanged between routes using URL parameters.
Configure variable for route: In the sample, the details route expects an id as a parameter in the URL. Example http://sample.com/details/10:

```
{
    path: details/:id,
    component: DetailsComponent
}
```

Read value from the URL: Import ActivatedRoute from @angular/router. Inject ActivatedRoute and access params. It's an observable to read value from the URL as seen in the example:

```
// inject activatedRoute
constructor(private activeRoute: ActivatedRoute) { }

// Read value from the observable
this.activeRoute
      .params
      .subscribe((data) => console.log(data[id]));
```

5 Tipps & Tricks

The following chapter is mainly an unorganized consumption of various Angular, D3 and general JavaScript tipps. The are categorized by the languages but within for example the Angular section, there is no specific order.

5.1 Angular

5.1.1 Tipp - Basic Setup

A basic boilerplate or setup for an angular project should contain the following structure. It helps creating a new angular application and understand what angular basically is. It's taken from a file called "app.module.ts" which is mostly the starting point of the application.

```
* Every angular application needs a root module. This file represents the route module
* of the application where the whole program starts.
* -) NgModule -- is used to define the module and is part of the angular core library.
* -) BrowserModule -- is required by any applciation that will run in the browser. Contains
* necessary directives such as ngFor or ngIf.

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
imports: [BrowserModule],            //Here the module declares all parts it needs.
declarations: [AppComponent],     //Here components, directives and pipes are declared
bootstrap: [AppComponent]       //Here it's defined which is the starting point of the app
})

export class AppModule { }
```

Bootstrapping: In computer technology the term (usually shortened to booting) usually refers to the process of loading the basic software into the memory of a computer after power-on or general reset, especially the operating system which will then take care of loading other software as needed. For angular this means basically "Start the application".

5.1.2 Tipp - Components

Components are nearly the most important part of every angular 2 application. They will be explained in the following very briefly in order to give a good introduction and also highlight their features:

- Imports
- Metadata
- Component class

Every angular component needs to import "Component" from the angular core. The Component decorator "@Component" is used in order to define the components metadata. An example declartion can be seen here:

```
@Component({
moduleId: module.id, //Important to identify or use relative path to component.
selector: 'd3-app', //The name of the html tag, css class, or attribute - for render.
templateUrl: 'componentsHTML/app.component.html',
styleUrls: [], //Reference here any stylesheets that should be applied.
providers: [] //Add here all services that are used by the component.
})
```

The meta data tells angular how to process the component, while the logic is located in the class of the component. Best practice is to use the "templateUrl" property to declare the relative path to an external html file with the components html. There are also other options like stlyeUrls or the providers (also child components of current can access) array which adds the services that are used by the component. The service still has to be imported by the component though.

Note: Further information is available under `https://angular.io/guide/cheatsheet`

The class of the component contains the logic. It contains variables, a constructor, functions, etc. An example declartion of a class can be seen in the following code snippet:

```
export class AppComponent {
  @Input() name: String;        // Component accepts input.

  constructor() { } // Constructor function

  ngOnInit() { } // Angular events. There are more custom event callbacks.

  changeName() { // Custom functions
      this.name = 'Bob';
  }
}
```

Important: The "this" refers to the component and is equally or nearly equal to the $scope of angular 1.

5.1.3 Tipp - Services

Define services within the "ngInit()" lifecycle method and not in the "constructor()" as it imporves the testability and unit testing.

5.1.4 Tipp - Observables in Angular

Observables are powerful and helpful in Angular 2. Here is a quote that describes them acutally really well.

You can think of an observable as an array whose items arrive asynchronously over time. Observables help you manage asynchronous data, such as data coming from a backend service. Observables are used within Angular itself, including Angular's event system and its http client service. To use observables, Angular uses a third-party library called Reactive Extensions (RxJS). Observables are a proposed feature for ES 2016, the next version of JavaScript.

The main purpose of using Observables is to observe the behaviour of a variable. In an imperative way, a variable is only changed when its state is mutated by assigning a new or updated value. The reactive approach allows variables to develop a history of evolving change, asynchronously being updated to the values of other points of data. Observables follow the Observer pattern, where a single piece of state (the Observable) is watched by one or more Observers which can react as it changes over time. At a low level, the Observable acts as an event emitter, sending a stream of events to any subjects that have subscribed to it. The Observable object doesn't contain any references to the subject classes itself: instead, an event bus is used to decouple publishing and subscribing concerns. What we get is an asynchronous stream of values that can be operated on using common iteration patterns.

5.1.5 Tipp - Angular CLI & Setup

The Angular CLI makes it easy to create angular applications with a default set of already built in functionality and a professional project setup. It has an build system included already, which is webpack, infrastructure for unit tests and much more. To install it gloabally and make it avaiable on the whole system use the command:

```
npm install -g @angular/cli
```

In order to create a new angular app, just type the following command:

```
ng new <nameOfApp>
```

After the files are created, a default setup is made, which has various files and already karma and jasmin tests included. In order to start the application or the example just type:

```
ng serve
```

A brief introduction to the folder structure is given in the following, while a more precise look needs research. The structure consists of the following main parts:

- e2e - Folder: Contains configuration for end to end testing

- node_modules - Folder: Contains necessary libraries and project dependencies
- src - Folder: Contains all of the angular code
- Several configuration files: package.json: As usual includes all dependencies and various script commands. Or Readme.md: Important information and further links for details.

To finally build the application, the following command can be used:
`ng build`

5.1.6 Tipp - Angular CLI Creation

The Angular CLI can also be used to create classes or components. For example in order to create a new class along all necessary files, type the following command:
`ng generate class <ClassName>`

Same goes for any service that should be used in the application. Type the following command to create a service:
`ng g service <ServcieName>`

5.1.7 Tipp - Passing Arguments to Components

In Angular there is a difference at creating components, especially by using the notation for binding. The converts the javascript statement inside to a string. In order to give a component a javascript object the property needs to be wrapped inside of [] and the binding can be removed. The following example shows it:

```
<my-card title='{{ card.title }}'></my-card>
<!--This will pass the title property as string to the custom component.-->
```

In Contrast:

```
<my-card [title]='card.title'></my-card>
<!--This will pass the title property as javascript object to the custom component.-->
```

Furthermore it's possible to give a component a event or handle events with components by using the ( ) notation for the attribute or event. For example:

```
<my-card (finishCardLoad)='onFinishCardLoad()'></my-card>
<!--This would pass the event finishCardLoad to the custom component.-->
```

This will subscribe to the event "finishCardLoad", which has to be implemented inside the component or emitted.

5.1.8 Tipp - EventEmitter

With event emitters it's possible to subscribe to them or specific events as they are working as Observables. So for example if an event emitter is created, more subscriber can take part in the event emitter:

```
let e = new EventEmitter();
e.subscribe(function() {
     console.log('First subscriber.');
});
e.subscribe(function() {
     console.log('Second subscriber.');
});

e.emit() // This tirggers an event and all subscribers that listen.
```

Over the emit function it's also possible to give parameters, that can be catched iniside the subscribers.

5.1.9 Tipp - Double Bound variable

In order to make a double bound variable and save a lot of code angular provides a module "FormsModule" which offers the functionality of `[(ngModel)]` and this allows a fast two way data binding. So if the variable is changing in the view also the variable in the model is changed. This is an awesome feature of Angular which removes unnecessary code and greatly enhances the application.

5.1.10 Tipp - Dynamic Classes

In order to dynamically render a class for an angular component or assign a class based on a condition it's possible to use the bracket notation []. The following example shows the assignment of a bootstrap class based on a condition:

```
...
<li class='list-group-item'
     [class.list-group-item-success]='listItem.myNumber >= 5'>
     <!--This will make any listitem green which meets the condition.-->
</li>
```

Another but more elegant way is to use the directive "ngClass" which either accepts an array of classes, that are assigned to the list element or any element or an object which allows the definition of property and values with conditions. The following example shows it:

```
<li *ngFor='let todo of todos' class='list-group-item'
     [ngClass]='{"list-group-item": true, // Default this class
     "list-group-item-success": todo.urgency == "low", // If condition is met assign this
     "list-group-item-danger": todo.urgency == "high"}' // If other condition is met assign thi
>
```

It's even possible to outsource the object generation into a function which further enhances the readability.

5.1.11 Tipp - Dynamic Styles

The same procedure for classes in angular is also possible for styles to apply them dynamically and with conditions. Either use bracket notation [] or also ngStyle which offers the same possiblities like ngClass.

5.1.12 Tipp - Render HTML inside a component

When using custom components it's often necessary to add also html inside of them. So for example the component <my-card> could include further html inside. In order to display this, the component needs to have inside it's definition the `<ng-content></ng-content>` tag. Wherever this tag is placed the content will be displayed that is passed to the component. If more content should be passed to the component it's possible to use the "select" attribute for the ng-content tag. Then it's easy to differ where each content part should be displayed.

5.1.13 Tipp - Style only Components

An important feature of angular is the use of the "styles" key in the component declaration (or "stylesUrl"). This allows the definition of a specific style for the current component. All style rules are only applied to html elements of the current component.

5.1.14 Tipp - Angular and Databases

Angular Js works at client side that runs in browser and calls API's (whether they be in php, node or any other server side language). The philosophy of angular application is to segregate front end functionality with back-end API. So the only thing we have to do with PHP now is simply send JSON responses.

The Angular CLI tool allows us to simple create our application client side and communicate with the serve. Then we "ng build" and the app gets compiled. Afterwards we can upload the folder it generates up to any apache server and it works. The Angular CLI makes a folder called "dist" that contains all the stuff our front end will need.

Note: PHP will be sitting on a server doing the job of an API, which is answering with JSON/XML to request, our angular app will then use the JSON to build the web interface.

The following example shows just a fronted call by using "observables" in angular and a php code in background:

```
...
ngOnInit() {

    let body=Path+'single.php'+'?id=' + this.productid;
    console.log(body);
    this._postservice.postregister(body)
                .subscribe( data => {
                            this.outputs=data;
```

```
                                    console.log(this.outputs);

                    },
                error => console.log("Error HTTP Post Service"),
                () => console.log("Job Done Post !") );
}
...
```

And here the respective php backend with a MySQL database:

```
$faillogin=array("error"=>1,"data"=>"no data found");
$successreturn[]=array(
        "productid"=>"any",
       "productname"=>"any",
      "productprice"=>"any",
    "productdescription"=>"any",
    "productprimaryimg"=>"any",
    "otherimage"=>"any",
    "rating"=>"any");
// Create connection  id,name,price,description,primary_image,other_image,rating

    $productid = $_GET["id"];
    $sql="SELECT * FROM product_list where id='$productid'";
    $result = mysqli_query($conn,$sql);
    $count = mysqli_num_rows($result);
    $value=0;
    while($line = mysqli_fetch_assoc($result))
       {

         $successreturn[$value]['productid']=$line['id'];
         $successreturn[$value]['productname']=$line['name'];
         $successreturn[$value]['productprice']=$line['price'];
         $successreturn[$value]['productdescription']=$line['description'];
         $successreturn[$value]['productprimaryimg']=$line['primary_image'];
         $successreturn[$value]['otherimage']=$line['other_image'];
         $successreturn[$value]['rating']=$line['rating'];
         $value++;
        }
     echo json_encode($successreturn);

 mysqli_close($conn);
```