COMP-308 Winter 2018

Lesson 7 Review

- **Authenticating Express apps**
  - **passport module**
  - **Authentication strategies**
- **Install configure passport**
  - Configure passport module in config folder
  - Register passport module in express.js file
  - Install authentication strategies modules
  - Configure **authentication strategies** in separate files in *strategies* subfolder of *config* folder
- **Passport Strategies**
  - Local strategies
  - Oauth strategies
- **Install passport-local**
- **Configure the local strategy in local.js**
- **Check against a mongoose model.**
- **Use user's instance method authenticate**
- **Configure the local authentication in passport.js**
  - include the local strategy config file
  - **serialize** the authenticated user (convert into stream of bytes)
  - **deserialize** when request are made (convert back to user object
- Add appropriate authentication fields to User model

Lesson 7 Review

- **Create pre-save middleware to handle the hashing of users‘ passwords**
  - create an autogenerated pseudo-random **hashing salt**
  - replace the current user password with a **hashed password** (more secure)
  - **authenticate** the password
- **Create authentication views**
  - **Signup.ejs**
  - **Signin.ejs**
- Connect the model and views using Users controller
- **Implement error handling**
  - Install and register Connect-Flash module
  - Create getErrorMessage method
  - Use req.flash to store error info and retrieve it to present it to views
- Create user session when user creation is successful
  - Use req.login method to create the session
  - req.user will hold the user object
  - Use req.logout to end the session

Lesson 7 Review

- **Wire user's routes**

  Post request to /signin route is handled by calling passport.authenticate('local',
  {
  successRedirect: '/',
  failureRedirect: '/signin',
  failureFlash: true
  }));

- **passport.authenticate()** method will try to **authenticate the user request using the strategy defined by its first argument**, local in this case.

- **Passport OAuth strategies**
  - allows users to register with your web application using an external provider
  - used by social platforms, such as Facebook, Twitter, and Google, etc.
  - Install passport-facebook
  - Configure Facebook strategy
  - Obtain Facebook info
  - Use passport.authenticate method with facebook strategy

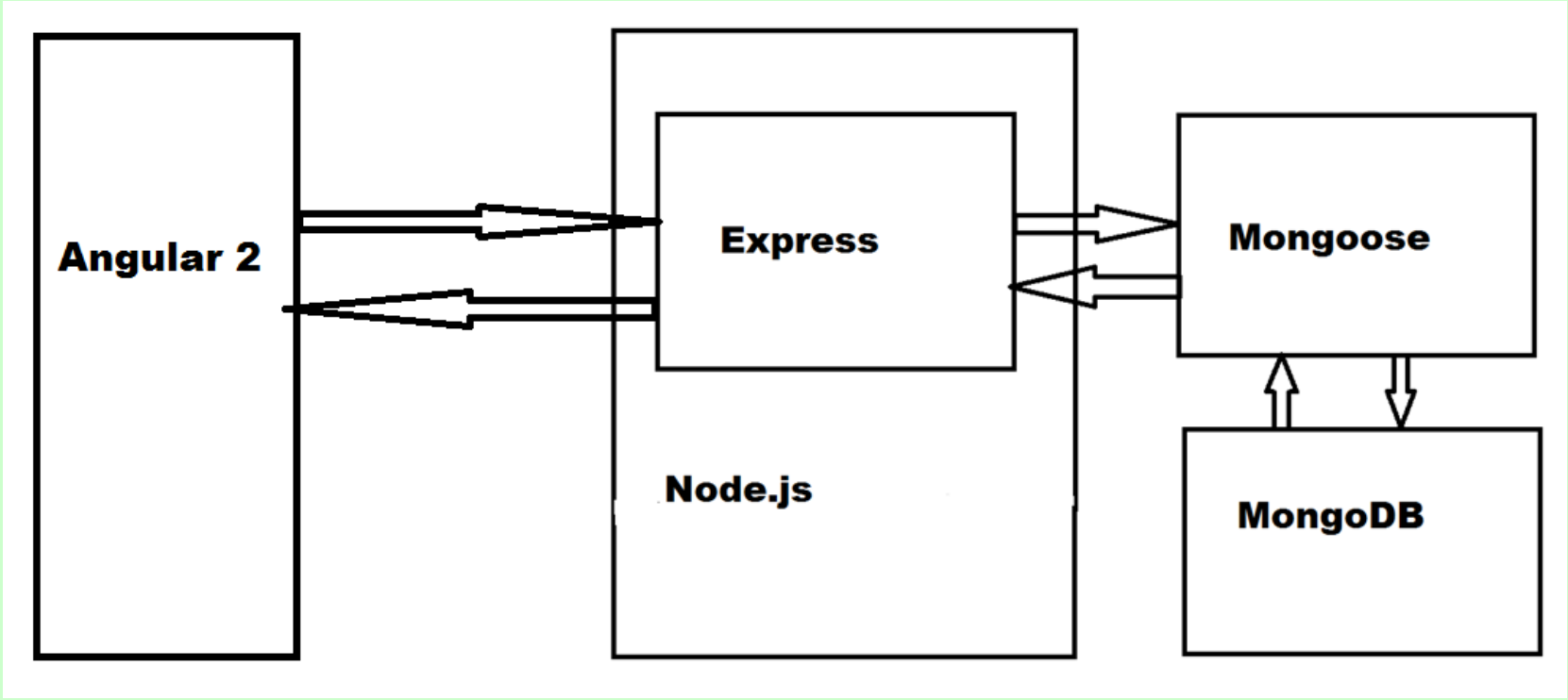
Introduction to Angular

Objectives:

- Introduce TypeScript
- Introduce Angular
- Understand the building blocks of Angular
  - components
  - services
  - templates
- Install and configure TypeScript and Angular
- Create and organize the Angular application

MEAN 2.0 Architecture

- Using Angular for building the front end:



Introduction to Angular

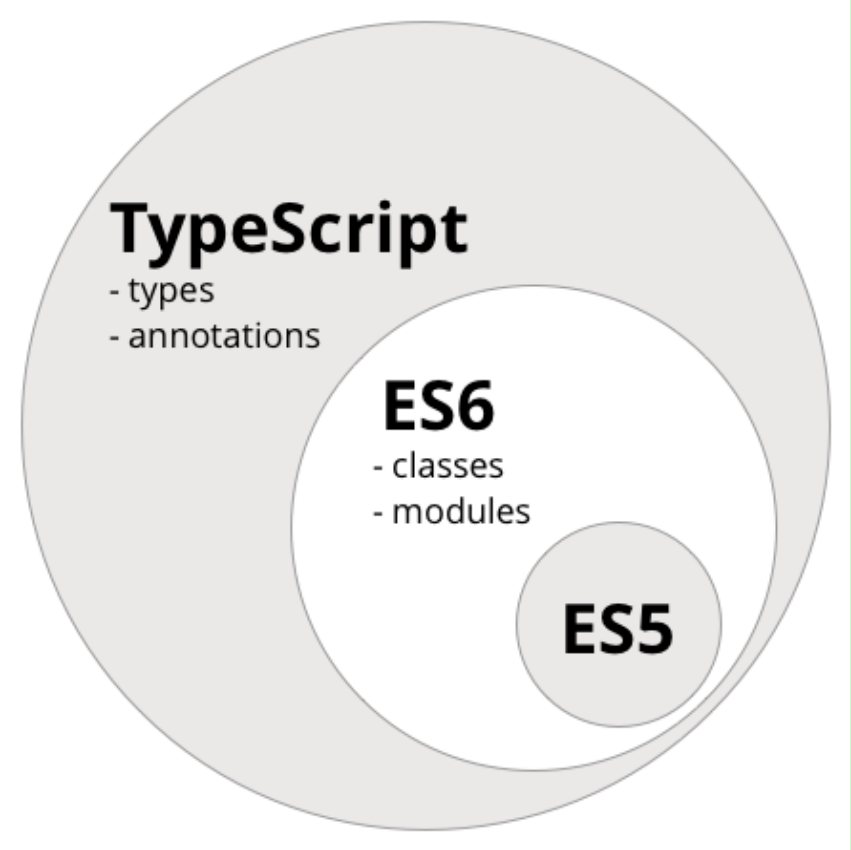
- **Angular** (previously **AngularJS**) project started in 2009 by Miško Hevery and Adam Abrons
- Angular is a **frontend JavaScript framework designed to build single-page applications using the MVC architecture**.
- The AngularJS approach **extends the functionality of HTML using special attributes** that binds JavaScript business logic with HTML elements.
  - Allows cleaner DOM manipulation through **client-side templating** and **two-way data binding** that seamlessly **synchronizes between models and views**.
  - Improved the application's code structure and testability **using MVC** and **dependency injection**.

Introduction to Angular 2

- **The current version is Angular 5**
- **Syntax**: Angular **relies on ES2015** (ES6) and because browser support is still lacking the Angular team decided to use TypeScript.
- **TypeScript is a typed programming language created by Microsoft**, which uses the object-oriented foundations of C#, Java, and now ES2015.
- Code written in TypeScript is **transpiled** into JavaScript code either in ES3, ES5 or ES2015 and can be run on any of the modern web browsers.
  - **Transpiling** takes source **code** written in TypeScript and translates that to JavaScript.

TypeScript

- **TypeScript is a superset of ES2015**, which means that it **allows you to write strongly typed ES2015 code**, which will later be compiled into the ES5 or ES2015 source depending on your needs and platform support.



Introduction to TypeScript

- **Types** - TypeScript support the **basic JavaScript types** and also allows developers to create and use their own types:
  - Types can be JavaScript **primitive types**, as shown in the following code:

```
let firstName: string = "John";
let lastName = 'Smith';
let height: number = 6;
let isDone: boolean = false;
```

- TypeScript also allows you to work with **arrays**:

```
var numbers:number[] = [1, 2, 3];
var names:Array<string> = ['Alice', 'Helen', 'Claire'];
```

Introduction to TypeScript

- **The any type**
  - represents any freeform JavaScript value.
  - the value of any will go through a minimal static type checking by the transpiler and will support all operations as a JavaScript value.
  - All properties on an any value can be accessed, and an any value can also be called as a function with an argument list.
  - Actually, **any is a supertype of all types**, and whenever TypeScript cannot infer a type, the any type will be used.
  - You'll be able to use the any type either explicitly or not:

```
var x: any;
var y;
```

Introduction to TypeScript

- **TypeScript classes can implement interfaces**
  - they have to conform to the properties or methods declared in the interface:

```
interface IVehicle {
    wheels: number;
    engine: string;
    drive();
}
```

```
class Car implements IVehicle {
    wheels: number;
    engine: string;
    constructor(wheels: number, engine: string) {
    this.wheels = wheels;
    this.engine = engine;
    }
    drive() {
        console.log('Driving...');
    }
}
```

Introduction to TypeScript

- **Decorators (introduced in ES7)**
- Decorators provide developers with a reusable way **to annotate and modify classes and members (methods and properties)**.
- A decorator uses the @decoratorName form, where the decoratorName parameter must be **a function that will be called at runtime with the decorated entity**.
- A simple decorator example:

```
function Decorator(target: any) {
}
@Decorator
class MyClass {
}
```

Introduction to TypeScript

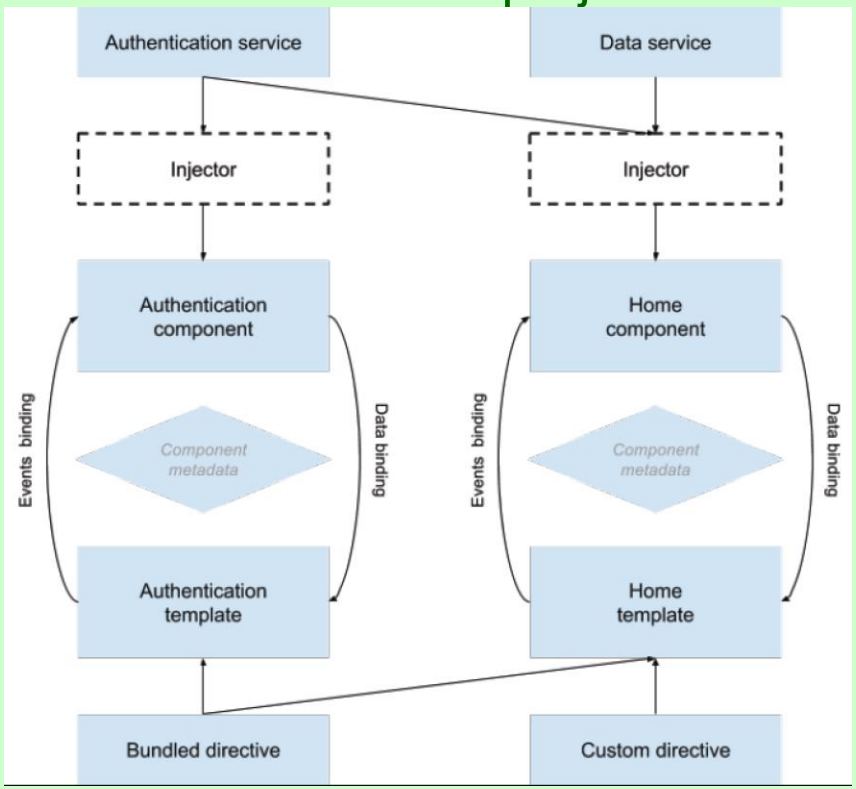
- At runtime, the decorator will be executed with the target parameter populated with the MyClass constructor.
- The decorator can also **have arguments** as follows:

```
function DecoratorWithArgs(options: Object) {
    return (target: Object) => {
    }
}
@DecoratorWithArgs({ type: 'SomeType' })
class MyClass {
}
```

- This pattern is also know as a **decorator factory**

Angular Architecture

- Angular uses a **component-based approach** with supporting entities, such as **services** and **directives** being injected into the components at runtime.
- It allows us to keep a **clear separation of concerns** and generally maintain a clearer project structure.



- Authentication **component example**:
  - uses authentication **service.**
  - performs **event binding** and **data binding** with Authentication **template**.
  - Authentication template uses **directives** to render the template.

Angular Architecture

- Consider an Angular application consisting of two components: Authentication and Home:
  - The center entities are the **components**.
    - Each component performs **data binding** and **event handling** with its template in order **to present the user with an interactive UI**.
  - **Services** are created for any other task, such as **loading data**, performing **calculations**, etc..
    - The services are then consumed by the components that delegate these tasks.
  - **Directives** are the **instructions for the rendering of the component's templates**.

Angular modules

- An Angular application consists of multiple modules, and each one is a piece of code usually dedicated to a single task.
- In fact, the entire framework is built in a **modular way** that allows developer to **import only the features they need**.
- Angular uses the ES2015 module syntax we covered earlier.
- An **NgModule** is a class decorated with @NgModule metadata (imports, declarations, etc)
- Our application will be built of custom modules as well, and a **sample application module** would look as follows:

Angular modules

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';
import { AppComponent } from './app.component';
import { AppRoutes } from './app.routes';
@NgModule({
imports: [
CommonModule,
RouterModule.forRoot(AppRoutes),
],
declarations: [
AppComponent
],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular Components

- A component is the essential building block of an Angular application.
- Its job is to **control a dedicated part of a user interface** usually referred to as a view.
- Most applications will consist of **at least one root application component** and, usually, **multiple components that control different views**.
- Components are usually defined as a **regular ES2015 class with a @Component decorator** that defines it as a component and includes the component metadata.
- The component class is then **exported as a module** that can be imported and used in other parts of your application.

Angular Components

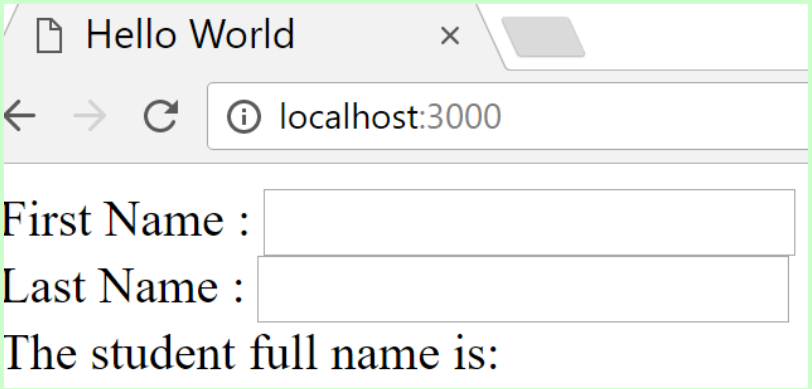
- A simple application component will be as follows:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<h1>I am an application component!</h1>'
})
export class AppComponent { }
```

Angular Templates

- Templates are used by the components to **render a component view**.
  - **a mix of basic HTML combined with Angular-dedicated annotations**, which tells the component how to render the final view.
- In the previous example, a simple template is passed directly to the AppComponent class.
- You can also save your template in an external template file and change your component as follows:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
templateUrl: 'app.template.html'
})
export class AppComponent { }
```



Angular data binding

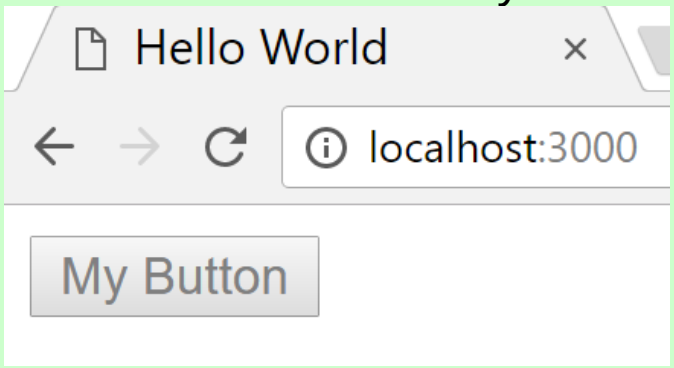
- Angular's data binding provides you with a straightforward way of managing the **binding between your component class and the rendered view**.
- **Interpolation binding**
  - An interpolation **binds a value of the class property with your template** using the **double curly brackets syntax**.
  - A simple example of this mechanism will be as follows:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<h1>{{title}}</h1>'
})
export class AppComponent {
title = 'Angular 2 Data Binding - interpolation binding';
}
```

Property binding

- Allows you to **bind an HTML element property value with a component property value** or any other template expression.
- This is done using square brackets, as follows:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<button [disabled]="isButtonDisabled">My
Button</button>'
})
export class AppComponent {
isButtonDisabled = true;
}
```

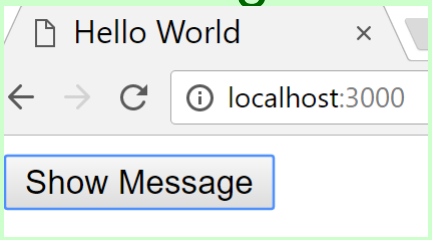


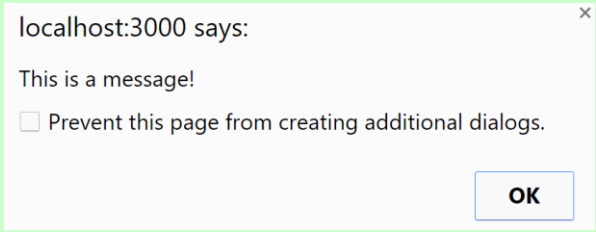
- The button will be rendered as disabled since isButtonDisabled property is set to true.

Event binding

- The mechanism of event binding allows to **bind a DOM event to a component method**
  - all you have to do is **set the event name inside round brackets**, as shown in the following example:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<button (click)="showMessage()">Show Message</button>'
})
export class AppComponent {
showMessage() { alert('This is a message!')
}
}
```



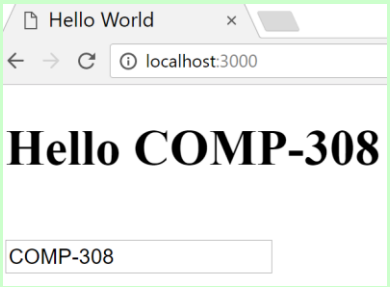


- In this example, a click event of the view button will call the showMessage() method inside our AppComponent class.

Two-way data binding

- When dealing with user inputs, we'll need to be able to **do two-way data binding** in a seamless way.
- This can be done by adding the **ngModel** property to your input HTML element and **binding it to a component property**.
  - use a combination syntax of round and square brackets, as shown in the following example:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<h1>Hello {{name}}</h1><br><input [(ngModel)]="name">'
})
export class AppComponent {
name = 'COMP-308 '
}
```



- The input binds the **name** property both ways, so every change to the input value will be updated in the AppComponent class and rendered into the view

Two-way data binding

- **FormsModule required for ngModel to work in HTML**
- **Import it in your app.module.ts file:**

```
import { FormsModule } from '@angular/forms';
//required for ngModel to work in HTML
@NgModule({
    imports: [
        BrowserModule, FormsModule
    ],
……..
…….
```

Angular Directives

- An Angular directive is a **set of instructions to transform dynamic templates into views**.
- There are several types of directives, but the most basic and surprising one is the **component**.
- The @Component decorator actually extends the @Directive decorator by adding a template to it.
- There are three types of directives:
  - **Attribute directives**
  - **Structural directives**
  - **Component directives**

Attribute directives

- Attribute directives **change the behavior or appearance of a DOM element**.
  - Used as HTML attributes on the given DOM element that we want to change.
- Angular comes with several **prebuilt** attribute directives, such as the following:
  - **ngClass**: to **bind** singular or multiple **classes** to an element
  - **ngStyle**: to **bind** singular or multiple inline **styles** to an element
  - **ngModel**: creates a **two-way data binding** over form elements
- You can and should write your own **custom directives**.

Structural directives

- Structural directives **change our application's DOM layout** by removing and adding DOM elements.
- Angular 2 contains three major structural directives you should know about:
  - **ngIf**: to add or remove elements according to the **condition**
  - **ngFor**: to create copies of an element based on a list of objects
  - **ngSwitch**: to display a single element out of a list of elements based on a property value
- All structural directives use a mechanism called the **HTML5 template**, which allows our DOM to hold an HTML template without rendering using the template tag.
  - This has a consequence that we'll discuss when we use these directives.

Component directives

- Every component is basically a directive.
- For instance, let's say we have component called SampleComponent:

```
import { Component } from '@angular/core';
@Component({
selector: 'sample-component',
template: '<h1>I am a sample component</h1>'
})
export class SampleComponent {
}
```

Component directives

- We can use it as a directive in our AppComponent class, as follows:

```
import { Component } from '@angular/core';
import { SampleComponent } from 'sample.component';
@Component({
selector: 'mean-app',
template: '<sample-component></sample-component>'
})
export class AppComponent {
}
```

- Notice how we use the **sample-component** tag as template
- You should declare all components using *declarations* property in @NgModule section of *app.module.ts* file.

Component directives

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

import { FormsModule } from '@angular/forms'; //required for ngModel to work in HTML
@NgModule({
    imports: [
        BrowserModule, FormsModule
    ],
    //declare all components here
    declarations: [
        AppComponent, SampleComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular Services

- Services are an essential part of Angular.
- They are just **plain classes** with a defined functionality.
- While components are focused on the user experience, services handle:
  - **data management**, **logging**
  - **application configuration**
  - any other functionality that does not belong in a component will be implemented as a service.
- We can **make these services available for components** using a mechanism called **Dependency Injection**.

Dependency injection (DI)

- Is a **software design pattern** popularized by a software engineer named Martin Fowler.
  - A **dependency** is an object that can be used (a service)
  - An **injection** is the passing of a **dependency** to a dependent object (a client) that would use it.
- In the following example the wheels and doors objects are created in the creator of the Car instance, called **injector**:

```
class Car { //dependency creation is out of the constructor
  constructor(wheels, doors) {
        this.wheels = wheels;
        this.doors = doors;
  }
}
//dependency injection as constructor injection
var car = new Car(new Wheels(), new Doors()); //better for testing
```

- In Angular, the DI system is **responsible for creating the dependencies and assembling them**.

Using Dependency Injection in Angular 2

- Dependency Injection is used to inject services into components.
- Here is a simple service:

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

@Injectable()
export class ExampleService {

    // this is a simple method of the service
    simpleMethod() {
        return 'Hi, I am a simple service!';
    }

}
```

Using Dependency Injection in Angular

- A service is used by components – **injected as an argument to component's constructor**.
- When Angular creates an instance of a component class, it will first **request an injector that will resolve the needed services** to call the constructor function.
  - If an injector contains a previous instance of the service, it will provide it; otherwise, the injector will create a new instance.
  - To do that, you'll need to provide the component injector with the service provider by adding the **providers** property to the @Component decorator.
- We can register providers at any level of our component tree, and a common pattern is to **register providers at the root level when the application is being bootstrapped**, so the same instance of the service will be available throughout the application component tree.

Using a Service

```
import { Component } from '@angular/core';
import { SampleComponent } from './sample/sample.component';
import { ExampleService } from './example.service';

@Component({
    selector: 'first-angular-application',
    //using a service
    template: '<h1>{{ title }}</h1>',
    providers: [ExampleService]
})
export class AppComponent {
    //service related code
    title: string;
    //include the service through dependency injection
    constructor(private _exampleService: ExampleService) {
    }
    ngOnInit() {
        this.title = this._exampleService.simpleMethod();
    }
}
```

Angular Routing

- Using web applications, users expect a certain type of URL routing.
- For this purpose, the Angular team created a module called the **component router**.
  - The component router **interprets the browser URL and then looks up in its definition to find and load a component view**.
  - Supporting the modern browser's history API, the router will respond to any URL change whether it's coming from the browser URL bar or a user interaction.
- You'll need to **load the router file separately** - either from a local file or using a CDN.

Routes

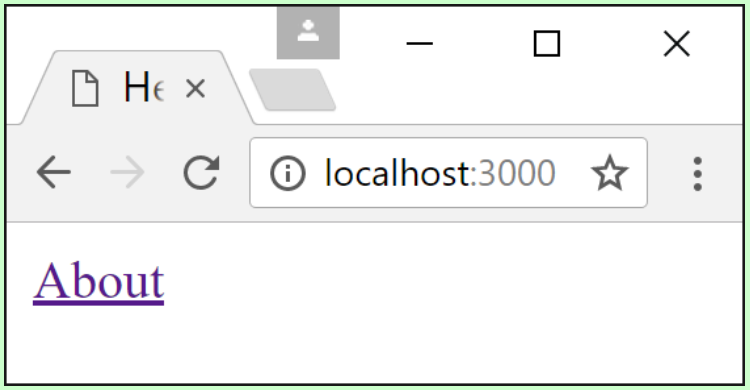
- Every application will have **one router**
  - when a URL navigation occurs, the router will look for the routing configuration made inside the application in order to determine which component to load.
- A special **array class** called **Routes is use to configure the application routing**
  - it includes a list mapping between URLs and components.
- This example configures routes for the HomeComponent:

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home.component';
export const HomeRoutes: Routes = [{
path: ' ', component: HomeComponent,}];
```

Router outlet

- The component router uses a **hierarchical component structure**
  - the root component is loaded, and it renders its view in the main application tag.
  - To **render your child components, include the RouterOutlet directive inside your parent component's template**.
  - An example component is as follows:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<h1>Application Title</h1>
<br><router-outlet></router-outlet>'
})
export class AppComponent { ... }
```

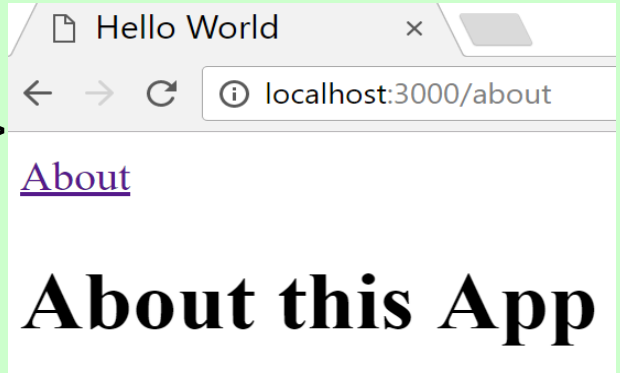


- Note that the **router-outlet tag will be replaced with your child component's view**.

Router links

- After we configure our application routes, we'll be able to navigate through our application either by changing the browser URL or using the **RouterLink** directive to **generate anchor tags pointing to a link** inside our app.
- The RouterLink directive uses an array of link parameters, which the router will later resolve into a URL matching a component mapping.
- An example anchor with the RouterLink directive will look like this:

```
<a [routerLink]="['/about']">About</a>
<div class="outer-outlet">
    <router-outlet></router-outlet>
</div>
```



Setting Up Angular Projects

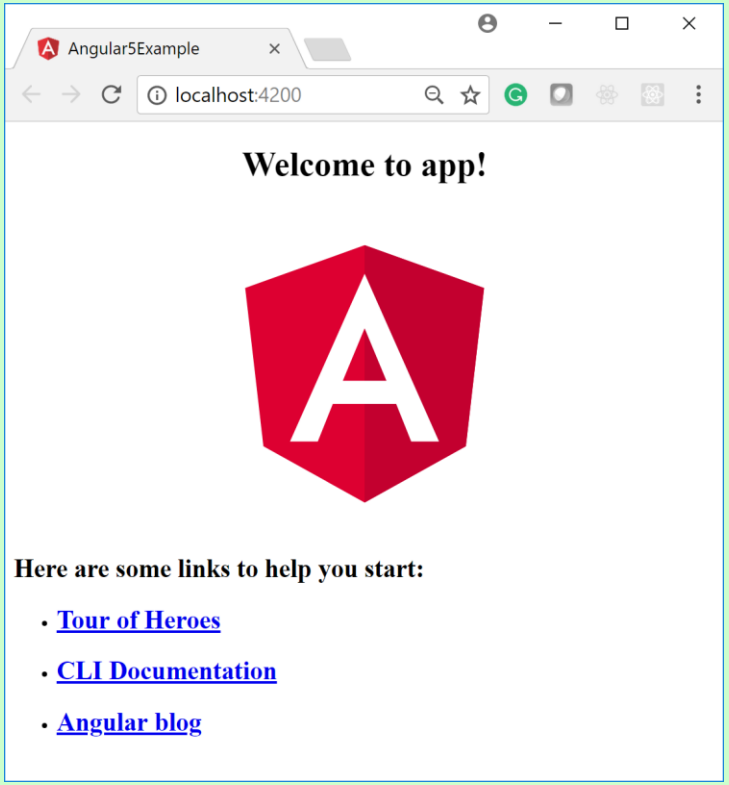
- You can generate an angular 5 app using **angular/cli**:

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

  ng new angular5-example

  cd angular5-example

  ng serve



- Navigate to http://localhost:4200/

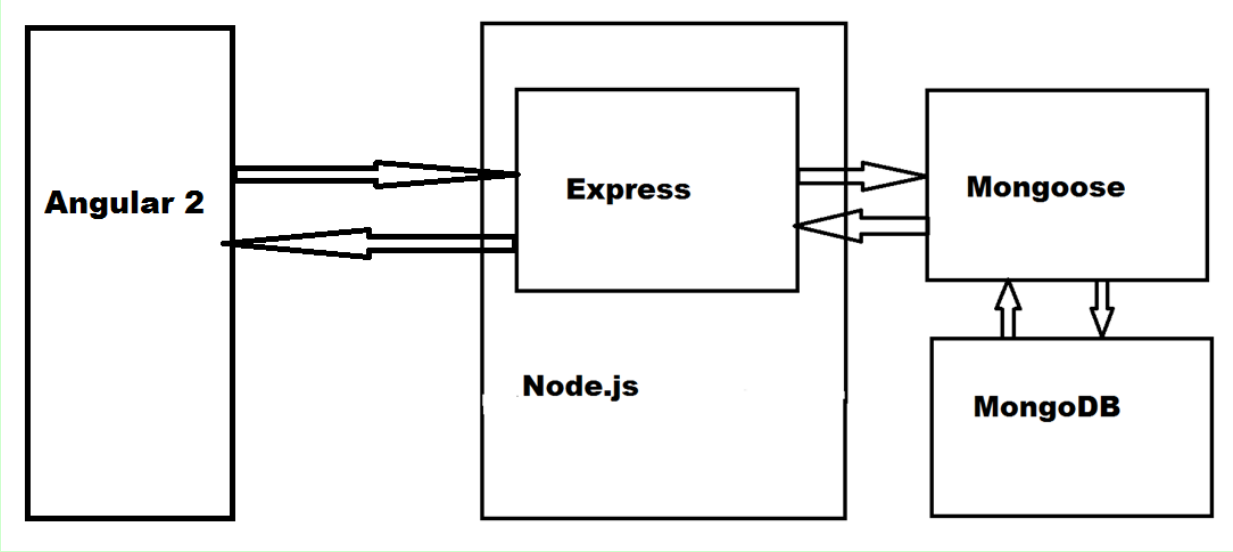
- You can continue to generate components:

ng generate component login

ng generate component user

Setting Up Angular Projects

- You can also follow the following steps:
  - Configure TypeScript
  - Configure Express
  - Restructure the application
  - Create the application **modules**
  - Create the application **components**
  - **Bootstrap** the application module
  - Start Angular application



Configure TypeScript and Angular

- In order to use Angular in our project, we'll need to install both **TypeScript** and **Angular**.
  - We'll need to use the TypeScript transpiler to convert our TypeScript files into valid ES5 or ES6 JavaScript files.
  - Since Angular is a frontend framework, installing it **requires the inclusion of JavaScript files in the main page of your application**.
- Use NPM to install all of our dependencies and run the TypeScript transpiler while we develop our application.
- In order to do that, you'll need to change your package.json file, as follows:

Configure TypeScript and Angular

```
{
  "name": "angular5-http-client",
  "version": "0.0.0",
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "app": "node server",
    "start": "concurrently \"npm run tsc:w\" \"npm run app\" ",
    "postinstall": "typings install"
  },
  "description": "Angular5HttpClient",
  "main": "server.js",
  "author": {
    "name": "inika"
  },
  "dependencies": {
    "@angular/common": "^5.2.7",
    "@angular/compiler": "^5.2.7",
    "@angular/core": "^5.2.7",
    "@angular/forms": "^5.2.7",
    "@angular/platform-browser": "^5.2.7",
    "@angular/platform-browser-dynamic": "^5.2.7",
    "@angular/router": "^5.2.7",
    "body-parser": "^1.18.2",
    "core-js": "^2.5.3",
    "compression": "^1.7.2",
    "connect-flash": "^0.1.1",
    "ejs": "2.5.2",
    "express": "^4.16.2",
    "express-session": "^1.15.6",
    "method-override": "^2.3.10",
    "mongoose": "^5.0.8",
    "morgan": "^1.9.0",
    "passport": "^0.4.0",
    "passport-facebook": "^2.1.1",
    "passport-google-oauth": "^1.0.0",
    "passport-local": "^1.0.0",
    "passport-twitter": "^1.0.4",
    "reflect-metadata": "^0.1.12",
    "rxjs": "^5.5.6",
    "systemjs": "0.21.0",
    "zone.js": "^0.8.20"
  },
  "devDependencies": {
    "concurrently": "^3.5.1",
    "traceur": "0.0.111",
    "typescript": "^2.7.2",
    "typings": "^2.1.1"
  }
}
```

Supportive libraries

- **CoreJS**: This will provide us with some ES6 polyfills
- **ReflectMetadata**: This will provide us with some a metadata reflection polyfill
- **Rx.JS**: This is a **Reactive framework** that we'll use later
- **SystemJS**: This will help with **loading our application modules**
- **Zone.js**: This allows the creation of different execution context zones and is used by the Angular library
- **Concurrently**: This will allow us to run both the TypeScript transpiler and our server concurrently
- **Typings**: This will help us with downloading predefined TypeScript definitions for our external libraries

Configuring TypeScript

- In order **to configure the way TypeScript works**, we'll need to add a new file called **tsconfig.json** to our application's root folder. In your new file, paste the following JSON:

```
{
"compilerOptions": {
"target": "es5",
"module": "system",
"moduleResolution": "node",
"sourceMap": true,
"emitDecoratorMetadata": true,
"experimentalDecorators": true,
"removeComments": false,
"noImplicitAny": false
},
"exclude": [
"node_modules",
"typings/main",
"typings/main.d.ts"
]
}
```

Configuring TypeScript

- Add a new file called **typings.json** to your application's root folder - extends the JavaScript environment with features and syntax that the TypeScript compiler doesn't recognize natively.
- In your new file, paste the following JSON:

```
{
"globalDependencies": {
"core-js": "registry:dt/core-js#0.0.0+20160914114559",
"jasmine": "registry:dt/jasmine#2.5.0+20161025102649",
"socket.io-client":
"registry:dt/socket.io-client#1.4.4+20160317120654",
"node": "registry:dt/node#6.0.0+20161102143327"
}
}
```

- Install new dependencies: **npm install**

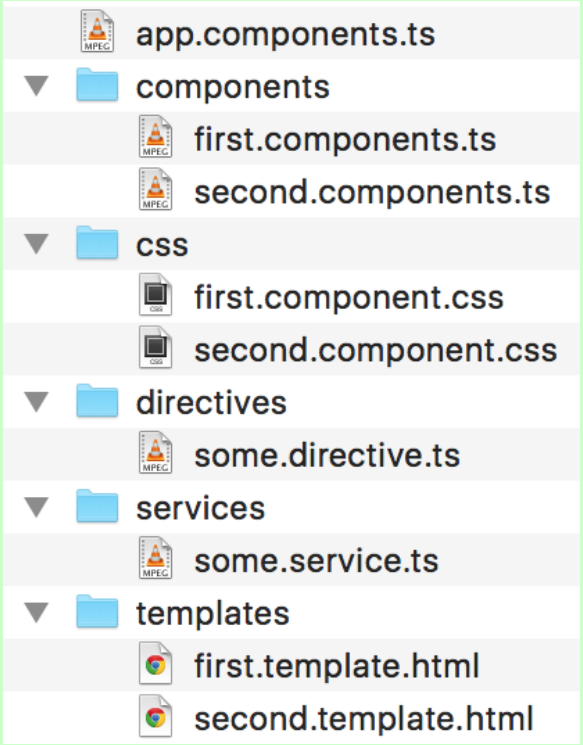
Configuring Express

- To start using Angular, you will need to **include the new JavaScript library files in our main EJS view**, app/views/index.ejs file, as the main application page.
- However, NPM installed all of our dependencies in the node_module folder, which is not accessible to our client side.
- To solve this issue, we'll have to change our config/express.js file as follows:

```
…..
app.use('/', express.static(path.resolve('./public')));
app.use('/lib', express.static(
path.resolve('./node_modules')));
….
```

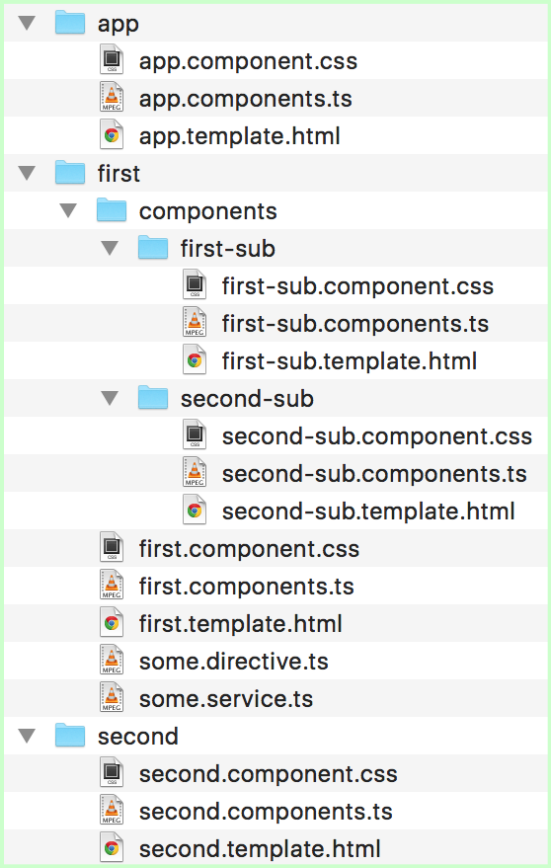
Restructuring the application

- A simple application can have a **horizontal structure** where **entities are arranged in folders according to their type**, and a main application file is placed at the root folder of the application:



Restructuring the application

- A **vertical structure** positions every file according to its functional context, so different types of entities can be sorted together according to their role in a feature or a section:



Creating the application module

- Clear the contents of the public folder and create the folder named **app** inside it. Inside your new folder, create a file named **app.module.ts** and add the following code:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
@NgModule({
imports: [
BrowserModule
],
declarations: [
AppComponent
],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Creating the application component

- Inside your public/app folder, create a new file named *app.component.ts* and add the following code:

```
import { Component } from '@angular/core';
@Component({
selector: 'mean-app',
template: '<h1>Hello World</h1>',
})
export class AppComponent {}
```

Bootstrapping the application module

- To bootstrap your application module, go to your app folder and create a new file named *bootstrap.ts*. In your file, add the following code:

```
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

- Basically, this code is using the browser platform module to bootstrap the application module for browsers.

Starting your Angular application

- Load our bootstrap code using the **SystemJS** module loader
  - create a new file named **systemjs.config.js** inside our public folder:

```
(function (global) {
    var packages = {
        app: {
            main: './bootstrap.js',
            defaultExtension: 'js'
        }
    };
    var map = {
        '@angular': 'lib/@angular',
        'rxjs': 'lib/rxjs'
    };
……..
```

app/views/index.ejs file

```
<!DOCTYPE html>
<html>
    <head>
        <title><%= title %></title>
        <base href="/">
</head>
    <body>
        <first-angular-application>
            <h1>Loading...</h1>
        </first-angular-application>

        <script src="lib/core-js/client/shim.min.js"></script>
        <script src="lib/zone.js/dist/zone.js"></script>
        <script src="lib/reflect-metadata/Reflect.js"></script>
        <script src="lib/systemjs/dist/system.js"></script>
        <script src="systemjs.config.js"></script>
        <script>
            System.import ('app').catch(function(err) { console.error(err); });
        </script>
    </body>
</html>
```

References

- Textbook
- https://github.com/angular/angular-cli
- https://angular.io/guide/npm-packages
- https://www.tutorialspoint.com/angular2/angular2_architecture.htm
- https://angular.io/docs/ts/latest/guide/typescript-configuration.html
- https://github.com/angular/quickstart/blob/master/src/systemjs.config.js
- https://coryrylan.com/blog/introduction-to-angular-routing
- http://plnkr.co/edit/RPvgcUdiLFP4Mtig9Q7n?p=preview
- https://angular.io/guide/setup-systemjs-anatomy