Workshop - Angular 2

Welcome to this lab created by **Angular Labs**. It's a pleasure having you here. Hope you enjoy it!

Author: **Gerard Sans** (@gerardsans)
Duration: 6 hours

Come back to this page when you need assistance as most information will be here.

If you like this lab we'd love to know about it. Tell us on @angularjs_labs. Thanks!

Join our Slack!

Hack | Learn | Share | Socialise

Slides

Follow slides synced with speaker bit.ly/qcon-workshop-slides-live or as standalone bit.ly/qcon-workshop-slides.

Levels (orientative)

🚀	Easy
🚀🚀	Intermediate
🚀🚀🚀	Advanced

QCon LONDON

Introduction

We are going to use a seed project to do the lab: angular2-webpack-starter by @AngularClass. A top of the shelf Angular 2 starter kit. This is a solid base that will stand time so we can focus on learning Angular 2.

The final solution is available for reference at github. Angular 2 is very exciting but can be frustrating sometimes. Don't forget to have fun!

Setup

Follow these steps to setup your environment:

1) Dependencies. Follow the instructions on the following link to get the basic environment working in your laptop before the session (getting-started). If you find any bug or need support please report it here.
   Other resources: Setting up your environment (OSX). Install node and npm (Windows).

2) Installing:

```
# navigate to a parent folder where you want the project
$ git clone https://github.com/angularclass/angular2-webpack-starter.git
$ cd angular2-webpack-starter
$ npm install
$ typings install
$ npm run server
# open your browser http://localhost:3000
```

Questions/Support

If you have any question that can benefit the group tell us using **#qconng2** on Twitter. For general troubleshooting you can use the **angular-labs** slack. Ultimately reach for me or my assistant (Todd Motto, @toddmotto) but be patient. Thanks!

🚀 10-15 min	**Angular 2 basics. Bootstrap**
	We are going to explore Angular 2 bootstrap. Uses: *bootstrap*, import, export, class *Author/s: @gerardsans*

1) Open "src/index.html". Notice the *<app>* element. This is where our Angular 2 application will be rendered. Try adding HTML elements within the *<app>* element.

```
// src/index.html
43 <app>
44  Loading...
45 </app>
```

2) Open "src/main.ts". This is the entry point for our application as it was setup in webpack.prod.config.js (line 44). *bootstrap* takes the root component (App) and an Array of dependencies (here you will add angular 2, application and vendors dependencies). We need to import the ***App*** component in order to pass it to bootstrap. Get familiarised with ES6 modules import. It allows us to cherry-pick components instead of loading the full source into the browser.

```
// src/main.ts
5 import {bootstrap, ...} from 'angular2/platform/browser';
...
21 import {App} from './app/app';
...
28 bootstrap(App, [ <dependecies> ]);
```

3) Open "src/app/app.ts". ***App*** is our root component. Bootstrap will use the CSS selector '**app**' to locate the DOM element and instantiate it. This selector must match the one used in "src/index.html" (line 43). Take a first look at annotations (@Component). Notice also how we used export in front the component (ES6 class) to make it available to the import in "src/main.ts" (line 21).

```
// src/app/app.ts
15 @Component({
16  selector: 'app'
64 })
...
72 export class App {...}
```

Angular 2 bootstrap will vary depending on your setup (ES5/ES6/TypeScript). In all cases, our Application starts with the call to ***boostrap*** passing in the ***root component*** and its dependencies.

🚀 🚀 15-20 min	**Angular 2 basics. Components**
	We are going to explore the structure of a simple Angular 2 Component. Uses: @Component, import, export, class, DatePipe *Author/s: @gerardsans*

1) Open "src/app/home/home.ts". An Angular 2 component will always follow this structure: imports section (angular, vendors or application imports), component annotations (@Component) and the component definition (ES6 class). Check out links to see all options. See pseudo-code below

```
import {<Class>} from '<sourcefile-without-extension>';
@Component({
  <option>: <value>
})
export class <ComponentName> {
  constructor() { }
}
```

2) Annotations help us to describe the details about our components. ***@Component.selector*** binds our class to a DOM element. Let's see how we can configure our component's template and styles. We used ***require*** to load the template and styles.

```
// src/app/home/home.ts
7 @Component({
11   selector: 'home',  // <home></home>
25   styles: [ require('./home.css') ],
27   template: require('./home.html')
28 })
```

3) Sometimes we would like to use inline strings (ES6 template strings). Replace current code to use them instead of using require for ***styles*** and ***template***. You can use simple replacements like these below but do try of your own.

```
// src/app/home/home.ts
   styles: [`h1 { color: red }`],
   template: `<h1>Home</h1>`
```

4) Extend the current ***Home*** component so it renders the current date/time every second. You can change the component state without having to worry about the digest cycle. Angular 2 change detection will pick up on changes (See zones). You can use the snippet below to demonstrate it. We are using an arrow function together with setInterval.

```
// src/app/home/home.ts
constructor() {
  setInterval(() => this.date = new Date(), 1000);
}
```

5) Add the date to the template so it displays the date. We can use the DatePipe to format the date output.

```
// src/app/home/home.html
<p>{{ date | date:'medium' }}</p>
```

🚀 🚀 15-20 min	**Angular 2 basics. Creating a Service**
	We are going to create a simple Service to provide a list of users and roles. Uses: import, export, class, Http *Author/s: @gerardsans*

1) Create a new file “src/app/services/usersService.ts”. This will be a simple ES6 class with a ***get*** method returning an array of users. You can start with the code below. Notice how we used export in front our ***App*** component to make it available to import. Add a ***get*** method to allow consumers of this class to retrieve the user list. Class properties and methods are public by default in TypeScript.

```
// src/app/services/usersService.ts
export class UsersService {
  private _users; //class property
  constructor(){
    this._users = [{
      id: 34,
      username: 'batman',
      roles: ['admin', 'user']
    }, {
      id: 67,
      username: 'spiderman',
      roles: ['user']
    }];
  }
  get() {
    return this._users;
  }
}
```

2) That was easy but is not a realistic scenario. Let’s read the user data using the new Http module. Create a json file ”src/assets/users.json” and use the previous data following an object structure like:

```
//src/assets/users.json
{ “users”: [
    { “id”: 34, “username”: “batman”, … }
  ]
}
```

3) In order to use the Http module we have to import it as we did before using:

```
// src/app/services/usersService.ts
import {Http} from 'angular2/http';
```

4) Angular 2 relies on RxJS for asynchronous operations like ajax calls. We need to instantiate Http in the constructor in order to use it later on the ***get*** method. Once we get the response we parse it into an object with json() and return it.

```
// src/app/services/usersService.ts
constructor(http: Http) {
  this.http = http;
}
get() {
  return this.http.get('/assets/users.json')
    .map(response => response.json());
}
```

Angular 2 Services could not be any easier. Let's use the ***usersService*** we just created to display some data.

🚀 🚀 15-20 min	**Users template. Rendering a list**
	We are going to render a simple list using the UsersService and apply some styling. Uses: *ngFor, *ngIf, ngClass, import, export *Author/s: @gerardsans*

1) We are going to render a list using the data from ***UsersService***. Import ***UsersService*** as we did before. Make sure the path is right.

```
// src/app/users/users.ts
import {UsersService} from '../services/usersService';
```

2) Add the service to providers array to make it available to Dependency Injection.

```
// src/app/users/users.ts
@Component({
  selector: 'users',
  providers: [
    UsersService
  ]
})
```

3) Retrieve the user list on the constructor of the ***Users*** component. Use parameter injection to make the service available to the class. Create the property ***userList*** to hold the list.

```
// src/app/users/users.ts
export class Users {
  private userslist;
  constructor(users: UsersService) {
    this.userslist = users.get();
  }
}
```

4) The code before would work just fine but we replaced the initial array with a call to Http.get returning an Observable<Response>. For this code to work we need to subscribe and set ***userList*** with the result.

```
// src/app/users/users.ts
users.get().subscribe(data => this.userslist = data.users);
```

5) Now we can use ***userList*** on our template. You can use the template here.

6) In order to display a list of the users we can use *ngFor (similar to previous ng-repeat). See the explanation for using asterisk. *ngFor will iterate over the ***userList*** Array. For each user we are setting #user a local template variable only available within the current element.

```
// src/app/users/users.ts
  <tr *ngFor="#user of userslist">
    <td>{{user.id}}</td> ...
  </tr>
```

7) Let's extend the current code to customise the style for superusers. Add a ***superuser*** flag to all your users in "src/assets/users.json".

```
//src/assets/users.json
{ "id": 34, "username": "batman", ... , "superuser": true }
```

8) On the template we can add a text after the user name. We can use *ngIf and the ***user.superuser*** field to do that.

```
// src/app/users/users.ts
<td>{{user.username}} <span *ngIf="user.superuser">(superuser)</span></td>
```

9) Maybe that last change was not enough. Let's change the row background. We will use a "superuser" CSS class in order to do that. Check out the syntax replacing ngClass directive below.

```
// src/app/users/users.ts
styles: [`
  .superuser {
    background-color: #eee;
  }
`],
template: `
  <tr [class.superuser]="user.superuser">...</tr>
`
```

🚀 🚀 15-20 min	**Angular 2. Setting up routing**
	We are going to learn how to setup the routes for our application. Uses: @RouteConfig, router-outlet, routerLink *Author/s: @gerardsans*

1) In order to use Angular 2 router we can include it's dependencies globally during bootstrap in "src/main.ts". We are going to set the location strategy to use hashes in the URL. You can remove it to try the default.

```
// src/main.ts
6 import {ROUTER_PROVIDERS, ...} from 'angular2/router';
...
28  bootstrap(App, [
31    ...ROUTER_PROVIDERS,
32    provide(LocationStrategy, { useClass: HashLocationStrategy })
33  ])
```

2) Routes are defined in ***App***, the root component. Open "src/app/app.ts". We need to import @RouteConfig annotation and the router directives (Eg: router-outlet and routerLink) so we can setup the routes and use the directives in our template.

```
// src/app/app.ts
5 import {RouteConfig, ROUTER_DIRECTIVES} from 'angular2/router';
15 @Component({
16  selector: 'app',
18  directives: [ ...ROUTER_DIRECTIVES, RouterActive ],
64 )
65 @RouteConfig([<routes>])
72 export class App {}
```

3) We are defining few routes for components ***Home*** and ***About***. The last one is to redirect all urls not matching any of the routes to the Home route.

```
// src/app/app.ts
65 @RouteConfig([
66  { path: '/', component: Home, name: 'Index' },
67  { path: '/home', component: Home, name: 'Home' },
  ...
70  { path: '/**', redirectTo: ['Index'] }
71 ])
72 export class App {}
```

4) We need to add a placeholder to render the content for the different routes on our template. We can use router-outlet so components can render its content depending on the current route.

```
// src/app/app.ts
35 template: `
53    <main>
54      <router-outlet></router-outlet>
55    </main>
```

5) To navigate between pages we use routerLink directive taking an Array including the route name.

```
// src/app/app.ts
44 <a [routerLink]=" ['Home'] ">Home</a>
```

6) Add a new ***User*** section to the application. You will have to add a new entry in the Router configuration and a link to navigate to it.

```
// src/app/app.ts
template: `
  <li router-active>
    <a [routerLink]=" ['Users'] ">Users</a>
  </li>
  `
})
@RouteConfig([
  { path: '/users', component: Users, name: 'Users' },
```

🚀 🚀 15-20 min	**Angular 2. Creating a subscription form**
	We are going to create a simple form to submit a subscription email. Uses: ngForm, ngControl, ngModel *Author/s: @gerardsans*

1) Create a new file “src/app/contact/contact.ts”. This will be a new route component. Add the new route to the @RouteConfig

```
// src/app/app.ts
@RouteConfig([
  { path: '/contact', component: Contact, name: 'Contact' }
])
```

2) Also remember to add the navigation link at the top.

```
// src/app/app.ts
<a [routerLink]=" ['Contact'] ">Contact</a>
```

3) Add the skeleton for our form to “src/app/contact/contact.ts” so it includes a header and a form like below. Add **#f=”ngForm”** to the form. ngForm directive will hold any child controls defined by ngControl and track their validity.

```
// src/app/contact/contact.ts
import {Component} from 'angular2/core';
@Component({
  selector: 'contact',
  template: `
    <h1>Contact</h1>
    <form #f="ngForm">
      <button type="submit">Submit</button>
    </form>`
})
export class Contact { }
```

4) Let’s add some code to handle the submit. Now we can reference various form states using the local template variable f. See how we disable the submit button and pass down the form content on submit.

```
// src/app/contact/contact.ts
@Component({
  template: `
    <form #f="ngForm" (ngSubmit)="onSubmit(f.form.value)">
      <p><button type="submit" [disabled]="!f.form.valid">Submit</button></p>
    </form>`
```

```
})
export class Contact {
  onSubmit(value) {
    console.log(`Submitted: ${JSON.stringify(value)}`);
  }
}
```

5) Let's add the email input field. First we will create the model to hold the email data. It will be useful when creating more complex forms too. Add ***[(ngModel)]="model.email"*** to set the two-way binding (ngModel). Then add ***ngControl="email"*** to register the input field within the ngForm. We will mark this field as required.

```
// src/app/contact/contact.ts
@Component({
  template: `
       <label>Email:</label>
       <input type="email"
         [(ngModel)]="model.email"
         ngControl="email"
         required>`
})
export class Contact {
  model = {};
}
```

6) Let's add a message when the field is invalid. We can define a local template variable email for that purpose using ngForm to monitor the input.

```
// src/app/contact/contact.ts
<input type="email"
     #email="ngForm"
     required>
<div [hidden]="email.valid" class="alert alert-danger">Email is required</div>
```

7) Last change enabled the input monitoring. Check how Angular automatically adds different CSS styles depending on the state. Let's add some styles.

```
  styles: [`
    .ng-valid[required] { border: 2px solid #42A948; /* green */ }
    .ng-invalid { border: 2px solid #a94442; /* red */ }
    .alert { color: #a94442; /* red */ }
  `]
```

🚀 🚀 15-20 min	**Angular 2. Testing a Service**
	We are going to create a simple service and create some tests. Uses: describe, it, beforeEachProviders, inject *Author/s: @gerardsans*

8) Create a new file “src/app/services/languagesService.ts”. This will be a simple ES6 class with a ***get*** method returning an array of languages. You can start with the code below. Add a ***get*** method to allow consumers of this class to retrieve the language list.

```
// src/app/services/languagesService.ts
export class LanguagesService {
  get() {
    return ['en', 'es', 'fr'];
  }
}
```

9) Create a new file “src/app/services/languagesService.spec.ts”. We will add our tests in this file. Let’s add the imports required for testing our new service.

```
// src/app/services/languagesService.spec.ts
import {describe, it, expect, inject, beforeEach, beforeEachProviders} from 'angular2/testing';
import {LanguagesService} from './languagesService';
```

10) In order to create our tests we need at least a suite and a spec like the one below:

```
// src/app/services/languagesService.spec.ts
describe('Service: LanguagesService', () => {
  it('should return available languages', () => {
    expect(true).toBe(false); //will fail
  });

  it('should return available languages', () => {
    expect(true).toBe(true); //will pass
  });
})
```

11) We can use modifiers to selectively disable specs (xit) or only execute them (fit). Try to disable the first test (spec).

```
// src/app/services/languagesService.spec.ts
describe('Service: LanguagesService', () => {
  xit('should return available languages', () => {
    expect(true).toBe(false); //will fail
  });
  ...
})
```

12) We can use modifiers to selectively disable specs (xit) or focus execution on them (fit). Try to disable the first test (spec).

```
// src/app/services/languagesService.spec.ts
xit('should return available languages', () => {
  expect(true).toBe(false); //will fail
});
```

13) First step of any test is doing the setup. This will usually involve beforeEachProviders and inject from Angular 2 testing wrapper. These will help the DI engine to resolve all dependencies as we do during bootstrap. beforeEachProviders expects an array with all dependencies. Let's add our service so it can be then injected later on our specs.

```
// src/app/services/languagesService.spec.ts
beforeEachProviders(() => [
  LanguagesService
]);
```

14) We can go ahead and improve our inital tests. Let's inject the service using inject and add some expectations.

```
// src/app/services/languagesService.spec.ts
it('...', inject([LanguagesService], (service) => {
  let languages = service.get();
  expect(languages).toContain('en');
  expect(languages).toContain('es');
  expect(languages).toContain('fr');
  expect(languages.length).toEqual(3);
});
```

15) That works just fine but if we were to create more tests we would be repeating inject for each of them. We can refactor the previous code with beforeEach. We can share the code to create the instance for the service and share the inject code with all specs.

```
// src/app/services/languagesService.spec.ts
describe('Service: LanguagesService', () => {
  let service;

  beforeEachProviders(...);

  beforeEach(inject([LanguagesService], s => {
    service = s;
  }));

  it('should return available languages', () => {
    let languages = service.get();
    expect(languages).toContain('en');
    ...
  });
})
```

Additional Resources

- JavaScript — Just another introduction to ES6
- Angular 2 Cheat Sheet
- Angular 2 Official Documentation