SERVER-SIDE RENDERING, TESTING AND DISTRIBUTION

LESSON 07

SWAFE-01

AGENDA

- Testing Angular applications
  - Jasmine
  - Angular TestBed
  - Karma test runner
- Server-side rendering
  - Rendering on the web
  - Angular Universal
- Deploying Angular applications
  - Building production code

TESTING

OVERVIEW

- Testing helps to check that it is working as expected
- Testing is set up by default when generating new applications with `ng new`
  - Run `ng new|g application --skip-tests|-S` to create a workspace/application without preconfigured testing
- The Angular CLI uses Jasmine and Karma
  - Jasmine is a behavior-driven development framework for testing JavaScript code
  - Karma is a test runner that spawns a web server that executes source code against test code for each of the browsers connected

TESTING SERVICES

- Services are (for the most part) straightforward to test
  - Services without dependencies can be tested without Angular testing utilities
- Services with dependencies
  - Services often depend on other services
    - They can be injected by hand while calling the service's constructor
    - Injecting real services rarely works well as most dependant services are difficult to create and control
  - Instead, mock the dependency
    - Use dummy values
    - Create a spies for relevant methods
- You almost always inject services into classes using DI
  - Tests should reflect that usage pattern

ANGULAR `TestBed`

- The most important Angular testing utility
- Creates a dynamically-constructed Angular module that emulates an `@NgModule`
- Inject components and services into the `TestBed` with `TestBed.inject()`

ANGULAR TestBed

```
describe('AuthService', () => {
  let service: AuthService;

  const loginResponse: LoginResponse = {
    access_token: 'access_token',
    refresh_token: 'refresh_token'
  }

  beforeEach(() => {
    const spy = jasmine.createSpyObj('RealFakeApiService', {
      login: of(loginResponse),
      logout: EMPTY,
    })

    TestBed.configureTestingModule({
      providers: [
        { provide: RealFakeApiService, useValue: spy }
      ]
    });
```

examples/lesson07-testing/projects/auth/src/app/auth.service.spec.ts

TESTING `auth.service.ts`

```
describe('#signUp', () => {
  let expectedPassword = 'correct-horse-stable-battery'
  let anotherPassword = 'another_password'

  it('should return an Observable<Error> when passwords do not match',
    (done: DoneFn) => {
      service.signUp('user@example.com', expectedPassword, anotherPassword)
        .subscribe(value => {
          expect(value).toBeInstanceOf(Error)
          done()
        }
      )
    }
  )

  it('should Observable<[]> when successful',
    (done: DoneFn) => {
      service.signUp('user@example.com', expectedPassword, expectedPassword)
        .subscribe(value => {
```

examples/lesson07-testing/projects/auth/src/app/auth.service.spec.ts

TESTING COMPONENTS

- A component combines an HTML template and a TypeScript class
  - In most cases, the component can be validated by testing only the TypeScript class
  - To adequately test a component, you should test that they work together as intended
- The Angular `TestBed` facilitates testing the template and the class
  - Component with `@Input()` and `@Output()`
  - Components with dependencies
  - Components with nested components

TESTING DEPENDENCIES

```
beforeEach(async () => {
  const spy = jasmine.createSpyObj('AccessLogService', {
    getAccessLogEntries: of(expectedEntries)
  })

  await TestBed.configureTestingModule({
    declarations: [
      AccessLogListComponent,
      AccessLogListItemComponentStub,
    ],
    providers: [
      { provide: AccessLogService, useValue: spy }
    ]
  }).compileComponents().then(() => {
    fixture = TestBed.createComponent(AccessLogListComponent);
    fixture.detectChanges();
    component = fixture.componentInstance;
    return fixture.whenStable().then(() => {
      page = new AccessLogEntryListPage();
```

examples/lesson07-testing/projects/auth/src/app/access-log/access-log-list/access-log-list.component.spec.ts

TESTING `@Input` AND `@Output`

```
1  beforeEach(async () => {
2    await TestBed.configureTestingModule({
3      declarations: [
4        RouterLinkDirectiveStub,
5        AccessLogListItemComponent,
6      ]
7    })
8    .compileComponents();
9  });
10
11 beforeEach(() => {
12   fixture = TestBed.createComponent(AccessLogListItemComponent);
13   component = fixture.componentInstance;
14   component.entry = expectedAccessLogEntry
15
16   fixture.detectChanges();
17 });
18
19 ...
```

examples/lesson07-testing/projects/auth/src/app/access-log/access-log-list-item/access-log-list-item.component.spec.ts

TESTING ROUTING

```
describe('#routing', () => {
  let stubs: RouterLinkDirectiveStub[] = []
  let debugElements: DebugElement[] = []

  beforeEach(() => {
    fixture.detectChanges();
    debugElements = fixture.debugElement.queryAll(By.directive(RouterLinkDirec
    stubs = debugElements.map(element => element.injector.get(RouterLinkDirect
  })

  it('should have links to all pages', () => {
    expect(stubs.length).toBe(2)
  })

  it('should have "" for home' , () => {
    expect(stubs[0].params).toBe('')
  })

  it('should have "access-log" for access-log', () => {
```

examples/lesson07-testing/projects/auth/src/app/navigation/navigation.component.spec.ts

CODE COVERAGE

- Code coverage pecentages estimate how much of a codebase is tested
- Angular projects (generated with the CLI) can generate coverage reports with `ng test --no-watch --code-coverage`
- Can be generated automatically every time tests run if configured in `angular.json`

SERVER-SIDE RENDERING

OVERVIEW

- A normal Angular application executes in the browser
- Angular Universal executes on the server
  - Generates static pages
  - Pages render more quickly, making them visible earlier to the client
- Angular Universal are compiled with Ahead-of-Time (AOT) compilation
  - Faster rendering—the browser downloads a pre-compiled version of the application
  - Fewer asynchronous requests—the compiler inlines external HTML and CSS within the application JavaScript
  - Smaller framework download size—No need to download the compiler

WHY USE SERVER-SIDE RENDERING

- Facilitate web crawlers through search engine optimization (SEO)
  - Search engines rely on web crawlers to index pages and their content searchable on the web
  - Web crawlers only read static content
  - SSR returns a static version of pages that makes them readable for web crawlers
- Improve performance on mobile and low-powered devices
  - Some devices might not support JavaScript
- Show the first page quickly
  - Displaying the first page quickly can be critical for user engagement
  - Serve a static version of the landing page to hold the user's attention
    - While loading the full application in the background

TERMINOLOGY

RENDERING

- SSR—Server-side rendering. Rendering a client-side application or universal app to HTML on the server
- CSR—Client-side rendering. Rendering an app in a browser, generally using the DOM
- Rehydration—“Booting up” JavaScript views on the client such that they reuse the server-rendered HTML's DOM tree and data
- Prerendering—Running a client-side application at build time to capture its initial state as static HTML

TERMINOLOGY

PERFORMANCE

- TTFB—Time to First Byte. Seen as the time between clicking a link and the first bit of content coming in
- FP—First Paint. The first time any pixel gets becomes visible to the user
- FCP—First Contentful Paint. The time when requested content (article body, etc) becomes visible
- TTI—Time To Interactive. The time at which a page becomes interactive (events wired up, etc)

SERVER RENDERING VS. CLIENT-SIDE RENDERING

- Server rendering
  - 👍—Fast FP, FCP, and TTI
  - 👎—Slow TTFB
- Static rendering
  - 👍—Fast FP, FCP, TTI, and TTFB
  - 👎—All HTML must be rendered, infeasable if unable to predict
- Client-side rendering
  - 👍—Flexible, fast TTFB
  - 👎—Slow TTI and FCP

UNIVERSAL WEB SERVERS

- A Universal web server responds with static HTML
  - Rendered with the Universal template engine
  - The server receives and responds to HTTP requests
  - Serve static HTML, JavaScript and CSS
- There are different rendering engines:
  - `@nguniversal/express-engine` –the application is using the Express.js engine
  - `@nguniversal/aspnetcore-engine` —the application is using the ASP.NET Core engine

WORKING AROUND BROWSER APIS

- Since the application is not running in the browser, some browser APIs and capabilities might not be available
  - Server-side application cannot reference global browser-only global objects, such as: `window`, `document`, `navigator`, or `location`
  - Angular offers injectable abstractions for some of these objects: `Location`, `DOCUMENT`
- If Angular does not provide it, it is possible to write new abstractions that delegate to the browser APIs while in the browser, and to an alternative implementation while on the server (also known as shimming)

SCRIPTS

- `npm run dev:ssr` — similar to `ng serve`, but uses server-side rendering
- `ng build && ng run <APP_NAME>:server` — builds the application and server code in production mode
- `npm run serve:ssr` — starts the server script serving the application locally. Remember to run `ng run build:ssr` to build the application before running it
- `npm run prerender` — used to prerender application pages

index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Server-side rendering</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
<link rel="stylesheet" href="styles.css"></head>
<body>
  <app-root></app-root>
  <script src="runtime.js" defer></script>
  <script src="polyfills.js" defer></script>
  <script src="vendor.js" defer></script>
  <script src="main.js" defer></script>
</body>
</html>
```

http://localhost:4200/about (`ng serve`)

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>Server-side rendering</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="styles.css" media="print" onload="this.media='a
  <link rel="stylesheet" href="styles.css">
  </noscript>
  <style ng-transition="serverApp">
    /*# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJzb3Vy
  </style>
  <style ng-transition="serverApp">
    /*# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJzb3Vy
  </style>
</head>
```

http://localhost:4200/about (`npm run dev:ssr`)

DEPLOYMENT

OVERVIEW

- Optimized build for production
- Environment files
- Browser support

PRODUCTION BUILDS

- When building `production` applications, the following build optimization features are used
  - AOT compilation—precompiles Angular templates
  - Bundling—Concatenates application and library files into large chunks
  - Minification—removes excess whitespace, optional tokens and comments
  - Uglification—rewrites code to use short, cryptic variables and function names
  - Dead code elimination—removes unreferenced modules and unused code
- Angular has a runtime production mode, that improves performance
  - Disables development-only safety checks
  - Disables debugging utilties

BUNDLE SIZES

`ng build --configuration=production`

Initial Chunk Files	Names	Size
main.7fed410212f761978c28.js	main	269.60 kB
polyfills.53fa429e6c79f04ad0e8.js	polyfills	36.21 kB
runtime.7f2a8ad29c32b55fb45e.js	runtime	2.72 kB
styles.31d6cfe0d16ae931b73c.css	styles	0 bytes
	Initial Total	**308.53 kB**

Lazy Chunk Files	Names	Size
308.77dc008d3591ddc9da63.js	—	270.60 kB

`ng build --configuration=development`

Initial Chunk Files	Names	Size
vendor.js	vendor	2.42 MB
polyfills.js	polyfills	128.55 kB
main.js	main	16.12 kB
runtime.js	runtime	12.21 kB
styles.css	styles	1.22 kB
	Initial Total	**2.58 MB**

Lazy Chunk Files	Names	Size
projects_auth_src_app_access-log_access-log_module_ts.js	—	317.01 kB

APPLICATION ENVIRONMENTS

- It is possible to define different named build configurations
  - Often, there will be different configurations for *staging*, *development*, and *production*
- A project's `src/environment` folder contains the base configuration file, `environment.ts`
  - Can be override by adding target-specific configuration files, eg. `environment.prod.ts`, `environment.staging.ts`
- To use a environment configuration, you must import the base configuration file (`environment.ts`)

ENVIRONMENT FILES

```ts
export const environment = {
  production: true,
  app_title: 'Auth Production'
};
```

```ts
export const environment = {
  production: false,
  app_title: 'Auth Debug'
};
```

examples/lesson07-testing/projects/auth/src/environments

```ts
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';
import { environment } from '../../environments/environment';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent {
  environmentTitle: string = environment.app_title;
  ...
  constructor(private formBuilder: FormBuilder) { }
  ...
}
```

examples/lesson07-testing/projects/auth/src/app/home/home.component.ts

FILE REPLACEMENT

- The main CLI configuration file, `angular.json`, contains a `fileReplacements` section in the configuration for each build target
  - Replace any file in the TypeScript program with a target-specific version of that file
- Select configuration with `--configuration` when serving/building the application

```
...
"configurations": {
  "production": {
    ...,
    "fileReplacements": [
      {
        "replace": "projects/auth/src/environments/environment.ts",
        "with": "projects/auth/src/environments/environment.prod.ts"
      }
    ],
    "outputHashing": "all"
  },
  "development": {
    "buildOptimizer": false,
    "optimization": false,
    "vendorChunk": true,
    "extractLicenses": false,
    "sourceMap": true,
    "namedChunks": true
```

examples/lesson07-testing/angular.json

BROWSER SUPPORT

- Angular supports most browsers
  - Chrome (latest)
  - Firefox (latest and extended support release (ESR))
  - iOS, Edge, Safari (2 most recent major versions)
  - Android (Q (10.0), Pie (9.0), Oreo (8.0), Nougat (7.0))
- Angular is built on the latest standards of the web platform
  - Targeting such a wide range of browsers is challenging, because they do not support all features of modern browsers
  - This is compensated by loading polyfills
  - A polyfill is a piece of code used to provide functionality in a browser that it do not support natively
- Differential loading is a strategy that allows your web application to support multiple browsers, but only load the necessary code that the browser needs

WRAP-UP

- Testing
  - Use `TestBed` for component testing
  - Jasmine and Karma
- Server-side rendering
  - Angular Universal
  - Pros & cons
- Deployment
  - Environment configuration
  - Browser support