# Inclass 3

March 25, 1159PM

Please upload this document with answers in it to the drop box.

Contents

[Inclass 3 1](#_Toc504243408)

[Angular CLI 1](#_Toc504243409)

[Creating and Running an Application 2](#_Toc504243410)

[Templating 2](#_Toc504243411)

[Creating Components 3](#_Toc504243412)

[Creating Services 4](#_Toc504243413)

[Deployment 6](#_Toc504243414)

[End to End Testing 7](#_Toc504243415)

[Protractor 7](#_Toc504243416)

[Jasmine 7](#_Toc504243417)

[Testing Web Applications on Any Platform 14](#_Toc504243418)

[Waiting for Resources 14](#_Toc504243419)

[Setting Up a Promise 15](#_Toc504243420)

[Unit Testing 16](#_Toc504243421)

[Linting Your Code 19](#_Toc504243422)

[Implementing SASS 19](#_Toc504243423)

[Asynchronous Calls Using Observables 21](#_Toc504243424)

# Angular CLI

You may be wondering now how to manage Angular projects since Angular relies on such a huge code base. Angular CLI is here to help. Angular CLI is endorsed by the Angular team to help automate Angular development and to manage the many different configurations and pieces needed.

<https://cli.angular.io/>

The latest version with documentation can be found here:

<https://github.com/angular/angular-cli>

Install angular-cli with the following command:

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

## Creating and Running an Application

Example 1: Creating and Running an Application with Angular CLI

This example shows the steps needed to create a new Angular application with Angular CLI. In a folder where you want your application, create the application with the following command. Then wait about 5minutes:

**>ng new myapp**

After the installation, cd into the **myapp** directory where index.html is and run the following command to build and launch your application:

>**cd myapp**

**myapp>ng serve**

To view your application, navigate to the following address in your browser:

<http://localhost:4200/>

## Templating

To help automate generation of components you may use the following commands:

ng generate component my-new-component

ng g component my-new-component # using the alias

# components support relative path generation

# if in the directory src/app/feature/ and you run

ng g component new-cmp

# your component will be generated in src/app/feature/new-cmp

# but if you were to run

ng g component ../newer-cmp

# your component will be generated in src/app/newer-cmp

# if in the directory src/app you can also run

ng g component feature/new-cmp

# and your component will be generated in src/app/feature/new-cmp

Of course, you can also use the following generation commands depending on the item being generated:

|  |  |
| --- | --- |
| [Component](https://github.com/angular/angular-cli/wiki/generate-component) | ng g component my-new-component |
| [Directive](https://github.com/angular/angular-cli/wiki/generate-directive) | ng g directive my-new-directive |
| [Pipe](https://github.com/angular/angular-cli/wiki/generate-pipe) | ng g pipe my-new-pipe |
| [Service](https://github.com/angular/angular-cli/wiki/generate-service) | ng g service my-new-service |
| [Class](https://github.com/angular/angular-cli/wiki/generate-class) | ng g class my-new-class |
| [Guard](https://github.com/angular/angular-cli/wiki/generate-guard) | ng g guard my-new-guard |
| [Interface](https://github.com/angular/angular-cli/wiki/generate-interface) | ng g interface my-new-interface |
| [Enum](https://github.com/angular/angular-cli/wiki/generate-enum) | ng g enum my-new-enum |
| [Module](https://github.com/angular/angular-cli/wiki/generate-module) | ng g module my-module |

### Creating Components

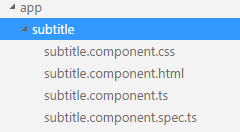
Angular CLI automates the process of creating classes, components and services.

Example 2: Building Directives

To build a directive with Angular CLI, run the command:

**c:\deljul4\myapp>ng g component subtitle**

Once you do, notice that a sub-title folder is created with css, html a component file and even a unit test.



The subtitle model in the subtitle.component.ts file is already built for you.

**myapp\src\app\subtitle\** **subtitle.component.ts**

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  @Component({  selector: 'app-subtitle',  templateUrl: './subtitle.component.html',  styleUrls: ['./subtitle.component.css']  })  export class SubtitleComponent implements OnInit {  subtitle: string;  constructor() {  this.subtitle = 'This is my new subtitle.';  }  ngOnInit() {  }  } |

Now, navigate to the subtitle-component.html file and customize the html template.

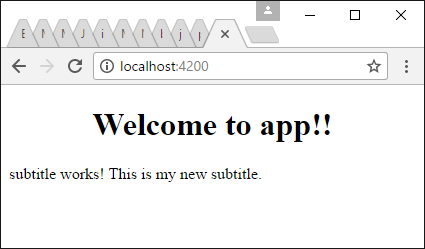
**myapp\src\app\subtitle\subtitle.component.html**

|  |
| --- |
| <p>  sub - title works! **{{**subtitle **}}**  </p> |

Next, replace the contents of app.component.html with the following tags to reference your new component directive:

|  |
| --- |
| <!--The whole content below can be removed with the new code.-->  <div style="text-align:center">  <h1>  Welcome to **{{**title**}}**!!  </h1>  </div>  <app-subtitle></app-subtitle> |

The output should now show the application with the new directive nested inside.



Exercise 1

Explain how the AppComponent class can access the SubTitleComponent class. In other words, where does it get imported and is this done automatically?

|  |
| --- |
|  |

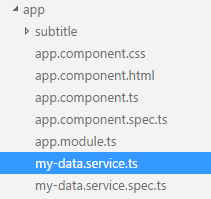
### Creating Services

Example 3: Building Services

This example shows how to use Angular CLI to build a service. First start with Example 2 and run the following command:

**myapp>ng g service myData**

Once you do, the my-data.service.ts and my-data.service.spec.ts files appear in the app directory.



Next, add the following code that is highlighted in yellow.

**myapp\src\app\my-data.service.ts**

|  |
| --- |
| import { Injectable } from '@angular/core';  @Injectable()  export class MyDataService {  names: Array<any>;  constructor() {  this.names = ['John', 'Mary', 'Joan'];  }  getNames() {  return this.names;  }  } |

Then, modify your app.component.ts file so it includes the necessary references to your service.

**myapp\src\app\app.component.ts**

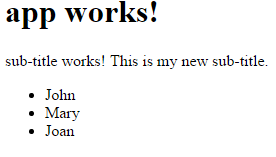
|  |
| --- |
| import { Component } from '@angular/core';  import { MyDataService } from './my-data.service'  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css'],  // 'providers' allows you to create and pass an instance  // of the class to the constructor header.  providers: [MyDataService]  })  export class AppComponent {  title = 'app works!';  firstnames: Array<any>;  // Store instance of 'MyDataService' at class level.  \_myDataService: MyDataService;  // Create instance of 'MyDataService' right in the constructor  // header.  constructor(mydataDervice: MyDataService) {  // Set class level reference.  this.\_myDataService = mydataDervice;  // Use service to call getNames() method.  this.firstnames = this.\_myDataService.getNames();  }  } |

Finally, modify app.component.html so it displays all of the names that are returned by the service.

**myapp\src\app\app.component.html**

|  |
| --- |
| <!--The whole content below can be removed with the new code.-->  <div style="text-align:center">  <h1>  Welcome to **{{**title**}}**!!  </h1>  </div>  <ul>  <li \*ngFor="let firstname of firstnames">  **{{**firstname**}}**  </li>  </ul>  <app-subtitle></app-subtitle> |

This is the output that appears.



## Deployment

It is possible to drastically reduce the number of files needed for deployment with Angular CLI. Since TypeScript does not run in the browser the TypeScript files do not actually get uploaded to your server. We need a way to automatically generate JavaScript for deployment. Angular CLI does a great job of bundling up a package for us.

Example 4: Deploying Application

To bundle all dependencies into a single file start with Example 1 and run the following command which places your production build in the myapp\dist folder:

**c:\deljul4\myapp>ng build -- prod**

To deploy your application, upload all files that are located in the myapp\dist folder after running the command above. Or, try hosting them on a local http server and running them from there. You could even run the app in a program like brackets. No typescript or node\_modules package is required.

## End to End Testing

End to end tests are automated tests of the application interface. This automation can be a significant help in testing out the application behavior as the application size grows. With Angular’s e2e test engine, Protractor, you can launch the application, inspect pages for content, send key strokes to your application and click on desired elements. Angular’s test engine is actually based on Selenium WebDriver which is a popular framework for testing web applications. The Selenium WebDriver API is can be used for testing with any web development framework.

### Protractor

Protractor is a Node.js program that the AngularJS team built to perform end-to-end tests. Protractor uses WebDriver to run test instance of web applications in most main stream browsers and to simulate user actions in the browser.

### Jasmine

Protractor uses Jasmine syntax to set up comparisons for actual and expected behavior. Jasmine enables assertions with **expectation functions** and **matchers**. The expectation function receives the actual value. The expectation is chained with a matcher to perform a Boolean comparison between the actual value and an expected value. Violations of expected behavior raise assertions. Among many other functions, matchers may include:

.not().toBe(null/true/false)

.not().toEqual(value)

.not.().toMatch(regex/string)

toBeDefined()

toBeUndefined()

toBeNull()

toBeTruthy()

toBeFalsy()

|  |
| --- |
| expect(nameService.getName()).not.toBeUndefined();  expect(nameService.getName()).toBe('Mary'); |

#### describe Blocks

*describe* blocks group a **suite** of tests together while also stating the purpose of the tests.

|  |
| --- |
| describe('This is a test for my AppCtrl controller: ', function () {  }); |

#### it Blocks (specs)

*it* blocks define each test. The test may contain one or more assertions.

|  |
| --- |
| describe('This test inspects the name model: ', function () {  ...  it('Should show Mary.', function () {  expect(nameService.getName()).toBe('Mary');  });  }); |

#### Document Locator Methods

To find objects (typically HTML elements) in a document, these locator methods (in addition to many others) can be applied to locate items:

by.binding('{{status}}')

by.className('redBtn')

by.css('.redBtn')

by.id('loginButton')

by.input("user")

by.linkText('Go Home')

by.partialLinktext('Home')

by.model('message')

by.name('email')

by.repeater("cat in pets")

by.repeater("cat in pets").row(1).column("{{cat.name}}"))

by.select("user")

by.selectedOption("red")

by.tagName('h2')

by.xpath('')

For example:

|  |
| --- |
| expect(element(by.id('fname')).getText()).toEqual('Jane'); |

##### Searching Lists of Elements

The .all() extension method can be added to the *element* object to implement searches on multiple HTML elements:

|  |
| --- |
| // Get list of elements.  let elements = element.all(by.tagName("li"));  expect(elements.count()).toEqual(3);  // Check text of first element.  expect(elements.get(0).getText()).toEqual("John"); |

#### Web Element Functions

To assist with testing, several additional functions can be used to extract data from elements and to perform actions on elements.

|  |  |
| --- | --- |
| **clear()** | If this element is a text entry element, this will clear the value. |
| **click()** | Click this element. |
| **getAttribute(name)** | Get the value of a given attribute of the element. |
| **getCssValue(propertyName)** | Get the value of a given CSS property. |
| **getLocation()** | Where on the page is the top left-hand corner of the rendered element? |
| **getSize()** | What is the width and height of the rendered element? |
| **getTagName()** | Get the tag name of this element. |
| **getText()** | Get the visible (i.e. not hidden by CSS) innerText of this element, including sub-elements, without any leading or trailing whitespace.  This does not work with the input element. Instead use the following:  var firstNameElement = element(by.model('firstName'));  var inputContents = firstNameElement.getAttribute('value'); |
| **isDisplayed()** | Is this element displayed or not? This method avoids the problem of having to parse an element's "style" attribute. |
| **isEnabled()** | Is the element currently enabled or not? This will generally return true for everything but disabled input elements. |
| **isSelected()** | Determine whether or not this element is selected or not. |
| **sendKeys(keysToSend)** | Use this method to simulate typing into an element, which may set its value. |

#### Browser Functions

To run a test in a browser, the following functions can be used to direct a browser to a site or get information about the page title.

|  |  |
| --- | --- |
| **get()** - Gets url. | browser.get('http://ssdprogram.ca/protractor/index.html'); |
| **getTitle()** - Gets page title. | browser.getTitle() |

Example 5: Setting Up an E2E Test with Angular CLI

Note: To get this application to run in BCIT’s labs you will need to update the version of Chrome. So please uninstall the current version of Chrome and then download and install the latest version.

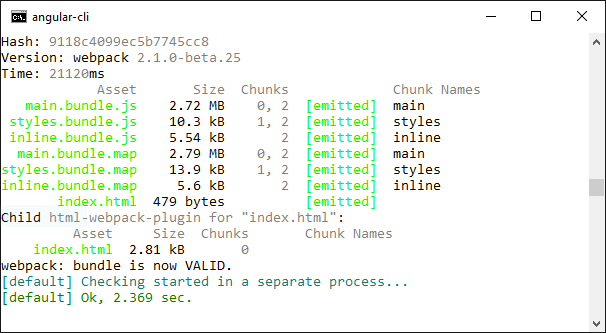
When you generate an application with Angular CLI you will notice that an end-to-end test is created for you. Remember, a new Angular application can then be created with this command:

**>ng new myapp**

Once the set-up is complete, cd into the src directory where index.html is and run the following command to build and launch your application:

**myapp>ng serve**

When you see the following output you can view the application.



To view your application, navigate to the following address in your browser:

<http://localhost:4200/>

The default end-to-end test is located at ‘myapp\e2e\app.po.ts’. First, notice the app.po.ts code loads the application in the browser and it retrieves the text that is shown within h1 tags. We can re-use this structures in this class from any end-to-end test page to load our application in the browser.

**myapp\e2e\app.po.ts**

|  |
| --- |
| import { browser, by, element } from 'protractor';  export class AppPage {  navigateTo() {  return browser.get('/');  }  getParagraphText() {  return element(by.css('app-root h1')).getText();  }  } |

This page runs the actual end-to-end test to ensure the “Welcome to app!!” text is present.

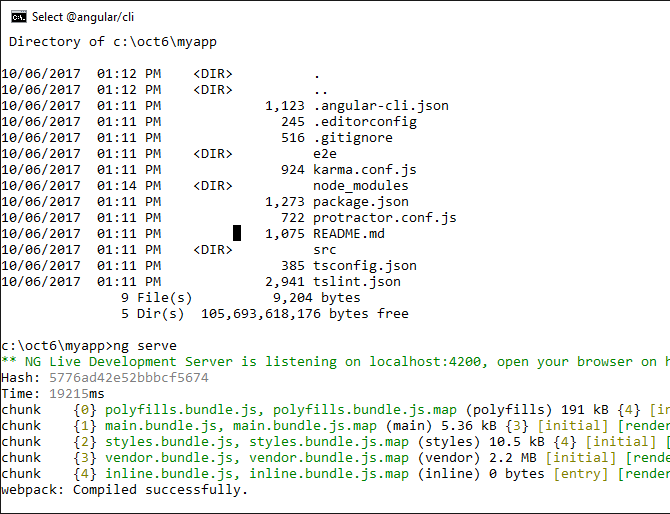
**myapp\e2e\app.e2e-spec.ts**

|  |
| --- |
| import { AppPage } from './app.po';  describe('myapp App', () => {  let page: AppPage;  beforeEach(() => {  page = new AppPage();  });  it('should display welcome message', () => {  page.navigateTo();  expect(page.getParagraphText()).toEqual('Welcome to app!');  });  }); |

You can run the e2e script in a second terminal or command prompt with the following:

**myapp>ng e2e**

After running the test, a browser will launch so the test can run. Then the test results are displayed in the console.



Example 6: E2E Testing Angular

This demonstration takes e2e testing a little further to show how you can navigate through an Angular document, send keys to its inputs and click on specific elements. Starting with an Angular CLI generated applications, replace the code in **app.component.html** as follows:

|  |
| --- |
| <h1>  **{{**title**}}**  </h1>  <p id='fname'>**{{**firstName**}}**</p>  <input type="text" id='cityInput' [(ngModel)]="city">  <input type="submit" id="showMsgButton" (click)="getMessage(city)">  <!-- Show is selectedCard is defined. -->  <div \*ngIf="message">  <label id="messageLabel">**{{**message**}}**</label>  </div> |

Then, replace the contents of **app.component.ts** with the following:

|  |
| --- |
| import { Component } from '@angular/core';  import { FormControl, FormGroup } from '@angular/forms';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app works!';  firstName = 'Jane';  message: string;  getMessage(\_city: string) {  this.message = 'It is always sunny in ' + \_city;  }  } |

To enable access to the form functions add the following highlighted lines to **app.module.ts**:

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

Next, to enable access to more test methods, add this import statement to the top of **app.e2e-spec.ts**:

|  |
| --- |
| import { browser, element, by } from 'protractor'; |

Then, replace the it block in **app.e2e-spec.ts** with the following code:

|  |
| --- |
| it('Get more practice working with elements in page', () => {  page.navigateTo();  // Ensure element with id='fname' has text equalling Jane.  expect(element(by.id('fname')).getText()).toEqual('Jane');  // Get input element and enter the text 'Boston'.  var cityInput = element(by.id('cityInput'));  cityInput.sendKeys('Boston');  // Get the button and click it.  var msgButton = element(by.id('showMsgButton'));  msgButton.click();  // Test the message that is output after the button is clicked.  var messageOutput = element(by.id('messageLabel'));  expect(messageOutput.getText()).toEqual('It is always sunny in Boston');  }) |

When you run the e2e test with the command:

**>ng e2e**

You will notice that the application launches, then the text ‘Boston’ is entered in the input box. After a button is clicked and the message that is output is tested.

Exercise 2

Add a *lastName* property (in addition to the firstName property) to the *AppComponent* class in Example 6. Assign a value to this property. Display the last name in a separate label or div tag and give it a unique ‘id’ attribute. Then, write an end-to-end test to determine if the element containing the last name equals the value that you gave it.

Show your revised app.component.html file here:

|  |
| --- |
|  |

Show your revised app.component.ts file here:

|  |
| --- |
|  |

Show your revised app.e2e-spec.ts file here:

|  |
| --- |
|  |

## Testing Web Applications on Any Platform

Selenium WebDriver can be used to test web applications that are built on any platform. Testing a Live Application. You could modify the test set-up generated by Angular CLI to point to any application whether it is development or production by changing the browser location:

|  |
| --- |
| import { browser, element, by } from 'protractor';  export class AppPage {  navigateTo() {  browser.waitForAngularEnabled(false);  return browser.get('http://cnn.com');  }  getParagraphText() {  return element(by.css('app-root h1')).getText();  }  } |

### Waiting for Resources

When testing for remote resources such as the transfer of JSON from one location to another, the Selenium WebDriver API does not have a precise way to wait for the resource to return. However, Selenium WebDriver does offer a simple method to implement the wait. You can hard code in a wait of 5 seconds (or other time) with the *sleep()* function:

browser.sleep(5000);

Example 7: Testing a List of Items

This example shows how to inspect a list of items on a web page that consists of HTML elements which really have nothing to do with Angular. To begin, start with Example 6. Then, replace the contents of **app.component.html** with the following source code:

|  |
| --- |
| <h1>Morning Show</h1>  <ul>  <li>Johnny</li>  <li>Holly</li>  <li>Nira</li>  </ul> |

Next, check replace the contents of app.e2e-spec.ts with this version.

**app.e2e-spec.ts**

|  |
| --- |
| import { AppPage } from './app.po';  import { browser, element, by } from 'protractor';  describe('myapp App', function() {  let page: AppPage;  beforeEach(() => {  page = new AppPage();  });  it('Testing one element by tag name.', () => {  page.navigateTo();  let headerElement = element(by.tagName("h1"));  expect(headerElement.getText()).toEqual("Morning Show");  });  it('Testing a list of items.', () => {  // Get list of elements.  let elements = element.all(by.tagName("li"));  expect(elements.count()).toEqual(3);  // Check text of first element.  expect(elements.get(0).getText()).toEqual("Johnny");  })  }); |

### Setting Up a Promise

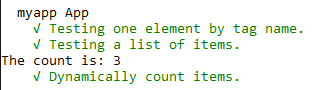
e2e tests execute on a separate thread other than the main thread so a promise must be used to allow processing of things like counts and for logging values while testing. Writing tests can get complicated so sometimes it helps to log data as you write the tests.

Example 8: E2E Testing with a Promise

This example demonstrates how to use a promise to process and log results of an e2e tests. In this case, the count is dynamically obtained, logged and used to display first and last elements in a list. To begin, start with Example 7 and add the following functions to the e2e test:

|  |
| --- |
| // Do more testing once the count has been determined.  function ProcessPromiseResults(elements, originalCount) {  expect(elements.count()).toEqual(originalCount);  console.log("The count is: " + originalCount);  expect(elements.get(0).getText()).toEqual("Johnny");  expect(elements.get(originalCount - 1).getText()).toEqual("Nira")  }  it('Dynamically count items.', function () {  let elements = element.all(by.tagName("li"));  // Set up a promise to retrieve the actual count value.  // Wait for the result and then process it.  elements.count().then(function (originalCount) {  ProcessPromiseResults(elements, originalCount)  });  }); |

Notice when running the e2e test, the dynamic content obtained with the promise is logged in the command prompt:



## Unit Testing

Whenever you use Angular CLI to generate a component, you will notice that it creates a unit test for it too. Like with E2E tests, Jasmine syntax is also used to enable unit tests.

If you look at the app folder you will notice the following two files have been created. First this is the component which will be unit-tested:

**myapp\src\app\app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app';  } |

Here is a subset of the HTML that is generated:

**myapp\src\app\app.component.html**

|  |
| --- |
| <!--The whole content below can be removed with the new code.-->  <div style="text-align:center">  <h1>  Welcome to **{{**title**}}**!!  </h1>  </div> |

Next, here is the unit test that is created for AppComponent. Three separate tests (specs) are run. The first test ensures that the app exists. The second test checks to ensure the title property equals ‘app works!’. The third test checks the title value in the html.

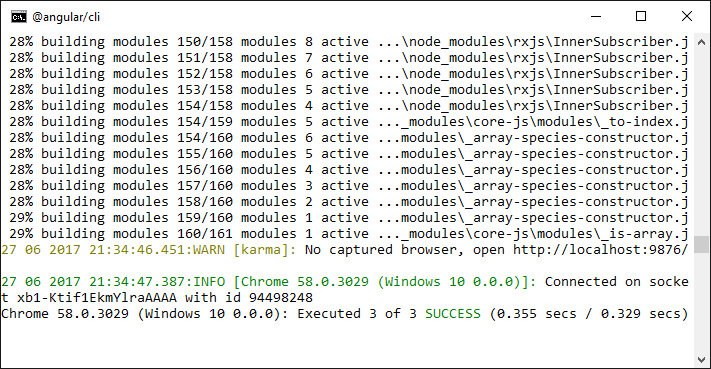
**myapp\src\app\app.component.spec.ts**

|  |
| --- |
| import { TestBed, async } from '@angular/core/testing';  import { AppComponent } from './app.component';  describe('AppComponent', () => {  beforeEach(async(() => {  TestBed.configureTestingModule({  declarations: [  AppComponent  ],  }).compileComponents();  }));  it('should create the app', async(() => {  const fixture = TestBed.createComponent(AppComponent);  const app = fixture.debugElement.componentInstance;  expect(app).toBeTruthy();  }));  it(`should have as title 'app'`, async(() => {  const fixture = TestBed.createComponent(AppComponent);  const app = fixture.debugElement.componentInstance;  expect(app.title).toEqual('app');  }));  }); |

To test your application, make sure the application is already being served and then run the following command:

**c:\deljul4\myapp>ng test**

After running this command Karma will launch and the results of the test will display.



Exercise 3

Add a property called *myName* to the *AppComponent* class. Assign your name to it. Add a test that ensures the actual value matches the expected value. Show your modified app\app.component.ts here.

HINT: Look at the highlighted code in **app.component.spec.ts** above:

|  |
| --- |
|  |

Show your modified app\app.component.spec.ts file here:

|  |
| --- |
|  |

Exercise 4

Given the following statements, indicate whether e2e or unit testing applies.

1. This type of testing allows you to simulate how a user would interact with the application.
2. Unit testing
3. e2e testing
4. This type of test offers precise information on where the error occurs.
5. Unit testing
6. e2e testing
7. This type of test takes longer to run.
8. Unit testing
9. e2e testing
10. This type of test examines one component at a time.
11. Unit testing
12. e2e testing

## Linting Your Code

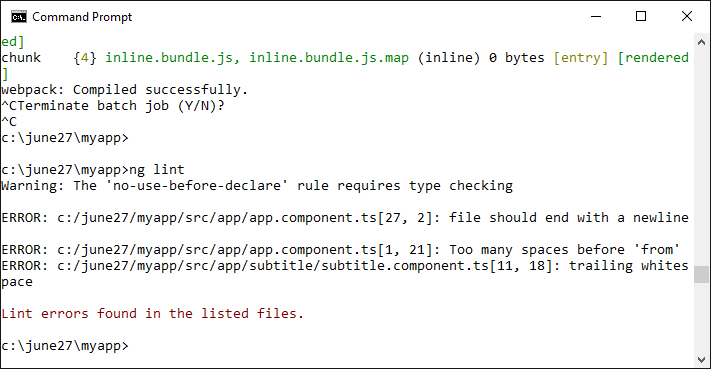
To check for syntax errors or to ensure your code complies with accepted standards, Angular CLI allows you to run lint.

Example 9: Linting Your Code

Starting with Example 3, run the following command:

**myapp>ng lint**

After, warnings will appear to help you tidy up your code.



## Implementing SASS

You can use Angular CLI to enable SASS to manage your CSS implementation. A really great summary of some SASS basics is here:

<http://sass-lang.com/guide>

Example 10: Implementing SASS with Angular CLI

This example shows how to implement SASS with Angular CLI to manage SASS variables. To begin start with Example 9.

**c:\deljul4\myapp>npm install node-sass**

Next, rename app.component.css to app.component.scss. The compiler will automatically convert this .scss or .sass files to css. Replace all initial code in the file with the following Sassy CSS:

|  |
| --- |
| $font-stack: Helvetica, sans-serif;  $primary-color:#ff0000;  .myclass {  font: 100% $font-stack;  color: $primary-color;  } |

Also, be sure to update the reference to the scss file from the app.component.ts file.

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.scss']  })  export class AppComponent {  title = 'app works!';  constructor() {  }  } |

To ensure try the scss out, replace the app.component.html code with this version.

|  |
| --- |
| <h1>  <div class="myclass">**{{**title**}}**</div>  </h1> |

Production mode is needed to enable Sass compilation so add the highlighted line to main.ts:

|  |
| --- |
| import './polyfills.ts';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { enableProdMode } from '@angular/core';  import { environment } from './environments/environment';  import { AppModule } from './app/';  environment.production = true;  if (environment.production) {  enableProdMode();  }  platformBrowserDynamic().bootstrapModule(AppModule); |

Refresh the browser and admire the new look!



## Asynchronous Calls Using Observables

Often, web applications will need to make requests for remote resources. Waiting for the resources can take several seconds and there is no guarantee the resource will arrive. There are several ways to manage this request so the call can be made on a separate thread and the resource can be received by the main thread in the application when the response arrives. One way to manage this is with observables. You can think of an Observable as a stream of events published by some source. We listen for events in this stream by subscribing to the Observable. In these subscriptions we specify the actions to take when the web request produces a success event or a fail event.

[https://angular.io/docs/ts/latest/guide/server-communication.html#!#http-client](https://angular.io/docs/ts/latest/guide/server-communication.html#!)

Example 11: Asynchronous Calls

To demonstrate how to use an observable, this example shows how to perform an asynchronous call for JSON. To begin, you could start with one of the previous examples in this document and add the following file.

**assets/test.json**

|  |
| --- |
| [  {  "first": "Jane",  "last": "Chan"  },  {  "first": "Bill",  "last": "Good"  }  ] |

Next, the following code can be added to your application. This code imports a third party library, rxjs, which has been endorsed for managing observables by the Angular team. The code in app.mynameservice.ts is basically a template that can easily be modified to enable get, post, put or delete actions with a remote service. For this case, the Observable is set up to retrieve JSON from the test.json file using a get request. Also notice that we are passing a reference through the constructor to the Http class which has access to the get, post, put and delete functions. This reference is passed from the module a little later.

**app/app.mynameservice.ts**

|  |
| --- |
| import { Injectable } from '@angular/core';  import { Http, Response } from '@angular/http';  import { Observable } from 'rxjs/Observable';  import 'rxjs/add/operator/map';  import 'rxjs/add/operator/catch';  import 'rxjs/add/observable/throw';  @Injectable()  export class MyNameService {  private dataUrl = './assets/test.json'; // URL to web API  constructor(private http: Http) { }  getNames(): Observable<string[]> {  return this.http.get(this.dataUrl)  .map(this.extractData)  .catch(this.handleError);  }  private extractData(res: Response) {  let body = res.json();  return body || {};  }  private handleError(error: any) {  // In a real world app, we might use a remote logging infrastructure  // We'd also dig deeper into the error to get a better message  let errMsg = (error.message) ? error.message :  error.status ? `${error.status} - ${error.statusText}` : 'Server error';  console.error(errMsg); // log to console instead  return Observable.throw(errMsg);  }  } |

Next, replace the code inside app.component.ts with the following code. This revised AppComponent creates an instance of the service through it’s constructor. The instance can be created in the constructor header with the help of the providers option. Note too how, since we are using a service that returns an Observable we can set up a subscription with handlers for the following cases:

1. Data is returned.
2. The request experiences an error.
3. A final block is entered regardless of whether 1 or 2 is selected.

**app/app.component.ts**

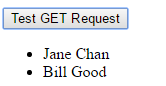
|  |
| --- |
| import { Component } from '@angular/core';  import { MyNameService } from './app.mynameservice';  @Component({  selector: 'app-root',  template: `<button (click)="getSomeData()">Test GET Request</button>  <ul>  <li \*ngFor="let myData of myNames">{{myData.first}}  {{myData.last}}</li>  </ul>`,  providers: [MyNameService]  })  export class AppComponent {  myNames: Array<any>;  \_myDataService: MyNameService;  // Since we are using a provider above we can receive  // an instance through an instructor.  constructor(myDataService: MyNameService) {  // Store local reference to MyDataService.  this.\_myDataService = myDataService;  }  getSomeData() {  this.\_myDataService.getNames()  // Subscribe to changes in the observable object  // that is returned by getRemoteData.  .subscribe(  // You basically get three handlers.  // 1. Handle successful data.  data => {  this.myNames = data  console.log(JSON.stringify(data))  },  // 2. Handle error.  error => {  alert(error)  },  // 3. Execute final instructions when successful.  () => {  console.log("Finished")  });  }  } |

To enable the HTTP get request we have to reference the HttpModule in our main module.

**app/app.module.ts**

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

The output from running the program after these changes is as follows:



Exercise 5

What is the role of the providers option in AppComponent?

|  |
| --- |
|  |

Exercise 6

What event happens if the address to the JSON is invalid? Please explain how this failed event is detected in app.component.ts.

|  |
| --- |
|  |