Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Notes	Weekly
Resources	Graded work
Policies	Standards
Professors & addendum	Code examples

Angular Components Introduction

This document introduces “components”.

What is an Angular component?

The following was summarized from Angular Docs > Fundamentals > Angular Concepts

A *component* controls an area of the user interface, called a *view*.

As code, a component is a class, with a “decorator”.

The class is JavaScript, specifically TypeScript. It includes all the code needed for the the component’s *behaviour* during its lifetime.

A decorator is a function that modifies a class; see the example below. It has one parameter, which is an object composed of configuration information as key-value pairs. This object is *metadata*, and the Angular runtime uses the metadata when initializing the component.

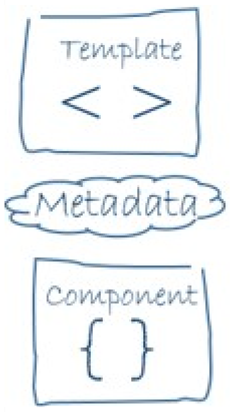
```
@Component({
  selector: 'app-customers',
  templateUrl: './customers.component.html',
  styleUrls: ['./customers.component.css']
})
export class CustomersComponent implements OnInit {
  // etc.
```

One of the decorator's properties is a *template*, defines the *appearance* of the component. The template includes HTML, or the name of an HTML file. By definition, HTML is the language of the Angular template. Almost all HTML elements are valid in a template, except for these: `html`, `body`, `base`, and `script`.

Another decorator property is the *selector*. Its value is the name of the custom HTML element in the *parent* template that becomes the component.

In summary, from the Angular documentation's Fundamentals > Angular Concepts guide:

Together, a component and template define an Angular view.



If you are familiar with the MVC or MVVM design patterns, how do they map to Angular code? Well, in Angular:

- The *component* has the code for the *controller* or the *view model*.
- The *template* has the code for the *view*.

Quick Review - Creating an Angular Project from Scratch

Before we start writing components, we should get a new Angular project going. Recall, from the week 6 notes:

1. Open and use the Terminal app.

2. Navigate to the folder that will hold your new project.

3. Generate a new project (replace "my-dream-app" with the desired name of the project):

```
  ng new my-app-name --routing -S -g
```

> ***Remember:***
> *The `--routing` option adds the code we need for "routing", which is a topic that will be covered in detail soon. Adding routing now (when the new project is created) is a best practice.*
> *The `-S` option does not add "testing" code, and the `-g` option does not add the git repo code.*

1. Go into the new folder, and open your editor (`code .`).

2. Finally, run/start the app:

```
cd my-app-name
ng serve --open
```

Creating a Component from Scratch (Using "ng generate")

Recall, from the Angular "Tour of Heroes" app, we can manually create a component using the following task:

```
# Make sure you are in the project folder
ng generate component foo --flat -S
# You can use abbreviations too...
# ng g c foo --flat -S

# If your ng version is 9 or later, omit -S...
# ng g c foo --flat
```

> ***Note:***
> *The "–flat" flag will NOT create a folder to enclose the component's source code files. We'll use that for the first while, until the number of files in the app folder gets too numerous.*
> *Documentation for the `ng generate component` command is here.*

Using "ng generate" to create the "foo" component for us has saved us time by automating the following (necessary) steps:

- Created the files:
  foo.component.css

foo.component.html
foo.component.ts
(including the correct class, and "@Component" decorator to make use of the .html & .css files / providing a default "app-" selector property)

- Added the correct "import" statement to app.module.ts, ie:

```
import { FooComponent } from './foo.component';
```

- Added "FooComponent" to the `@NgModule` decorator in `app.module.ts`:

```
@NgModule({
  declarations: [
    AppComponent,
    FooComponent // Added!
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Adding the new component to the view

Recall the "selector" property that was automatically added, ie "app-foo". This corresponds to the custom element `<app-foo></app-foo>` that we can use to render our newly created "foo" component.

To see this in action, add the `<app-foo></app-foo>` element to the bottom of your **app.component.html** file and "serve" your application again, if it's not already running.

You should now see the text "foo works" at the bottom of the default "start" page.

How to think about and plan your components

Now that we are comfortable adding new (simple) components, why don't we try creating a view using *multiple* components.

> *We will return to this document soon…*
>
> *Let's do some hands-on coding, and create an app that has multiple components.*
>
> *The content for this discussion can be found in the "Angular Components Example" document. Open that now.*

Using "Templates" in your components

From the example above, we have seen how we can add multiple components as children of a parent component. However, these were simply "static" components (i.e. their content is hardcoded into the .html). If we wish our "template" (.html file) to reference values within its corresponding component, we need to reference them using the following techniques:

- Interpolation
- Template Expressions / Property Binding
- Template Statements
- Attribute, class, and style bindings

Essentially, the above logic is really referring to specific types of "Binding Syntax".

Note: We will delay in-depth coverage of "two-way binding" until we discuss forms in Angular.

Quick "directive" overview

A *directive* is an Angular class. It interacts with HTML elements in the browser DOM. There are three kinds of directives:

1. Components — directives that have an HTML template
2. Structural directives — they change the DOM layout by adding and removing DOM elements
3. Attribute directives — change the appearance or behavior of an element

Components are the most common kind of directive.

Directives overview

Built-in directives

- Built-in attribute directives
- Built-in structural directives e.g. `ngIf`, `ngFor`, etc.)

Building a simple "attribute" directive

A topic for more advanced scenarios:

- Build a simple attribute directive