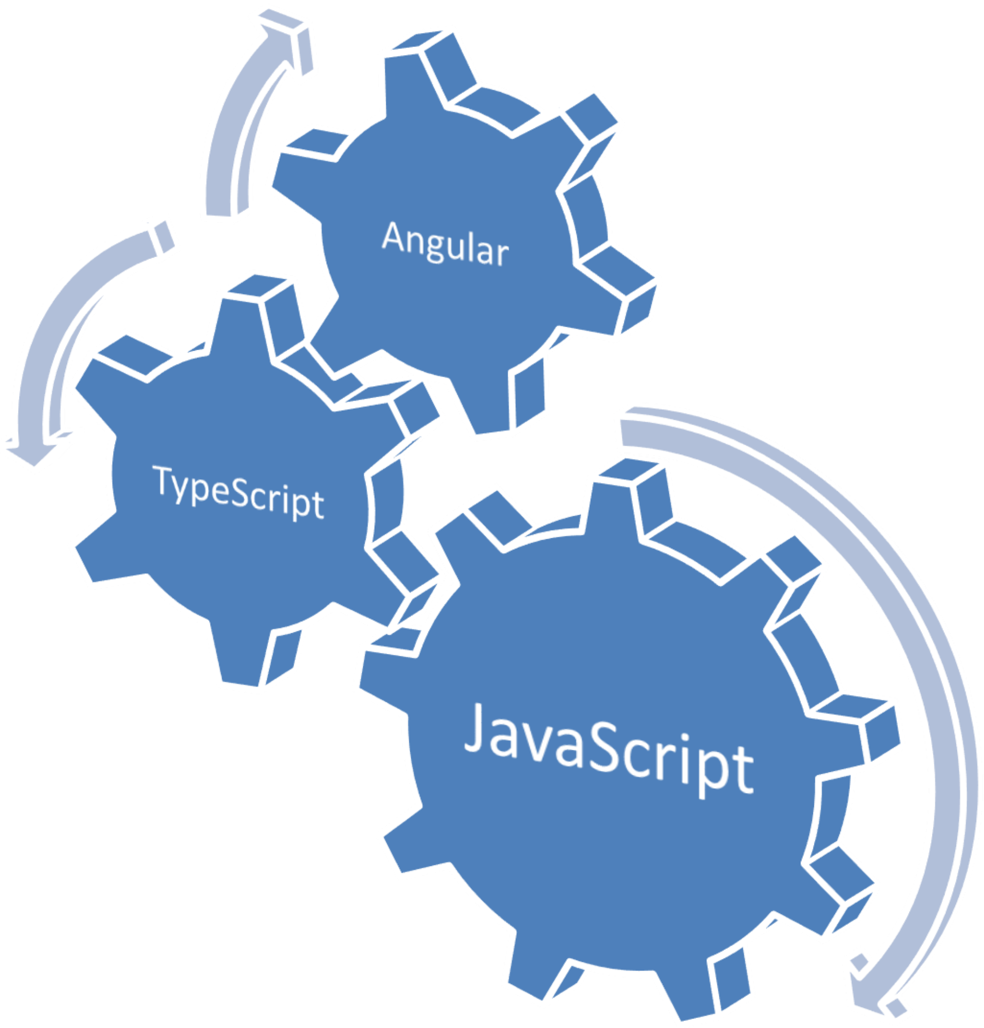
WD-530 Angular for TypeScript exercises 1

Client-side application building

Centriq Training



530 Angular exercises v3.0.docx
last saved March 13, 2020

Table of Contents

Angular for TypeScript

TypeScript Setup

1. Set up basic apps

Create project folder

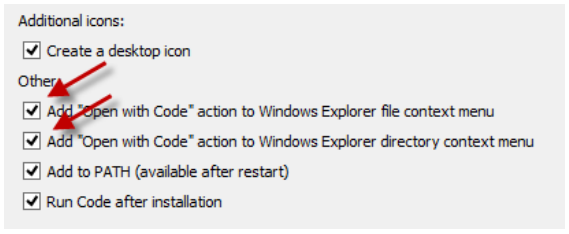
- Create a folder on your computer called **typescript**.
    - The desktop is a good place to put it so you can find it unless you have a better place.

Install node.js to use npm

- Navigate to **https://nodejs.org/** in your browser.
- Download the **Current** version (latest features) by clicking, downloading, and installing the msi download.
    - All the defaults are good.

Install Visual Studio Code

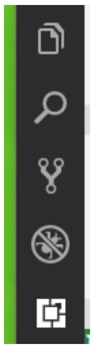
- Navigate to https://code.visualstudio.com/ and download the application.
- Install the application
    - Check the two options for 'Add "Open with Code" action to Windows Explorer...' .
    - All other defaults are good.
- Read the welcome file when Code opens if you want and then close Code.
- Right-click on the **typescript** directory and select **Open with Code**.



Additional icons:
- [✓] Create a desktop icon

Other:
- [✓] Add "Open with Code" action to Windows Explorer file context menu
- [✓] Add "Open with Code" action to Windows Explorer directory context menu
- [✓] Add to PATH (available after restart)
- [✓] Run Code after installation

Install VS Code extensions



- Install the following extensions by clicking the last icon on the left ribbon and searching for the extensions. Then click **install** and subsequently the **reload** button that appears.
- The one thing missing from Code is the ability to format CSS code. To allow you to right-click format your code select either
    - Prettier
    - Beautify
- Optionally you may install whatever you like
    - **Code Runner** – run selected code in any installed coding environment
    - **Node Exec** – execute selected code directly in Node's environment
    - **Debugger for Chrome** – a Microsoft extension for a debugging session

2. Set up command line environment

Open an integrated terminal to install TypeScript

- Select **View / Terminal** from Code's menu
  or
- Type Control+backtick
- To install **TypeScript** globally, At the command prompt type

```
npm install -g typescript
```

- Check that TypeScript has been installed by typing at the command prompt with
  - You will see lots of help text for valid options.

```
tsc
```

- If you need to, set your working directory with the cd command.

Transpile with tsc

- Create a file called HelloWorld.ts in your **typescript** directory with the following code

HelloWorld.ts

```
{
    let name: string = 'world';
    console.log('Hello, ' + name);
}
```

- Then transpile the script from TypeScript to JavaScript in a Terminal / New Terminal window.

```
tsc HelloWorld
```

- Click and view the resulting JavaScript file, HelloWorld.js. It will look a little different.
- Run the resulting JavaScript file with node.js's V8 JavaScript engine in the terminal.

```
node HelloWorld
```

- Kill the terminal window by clicking on the trash can icon.

3. Set up VS Code for TypeScript

Set up TypeScript settings

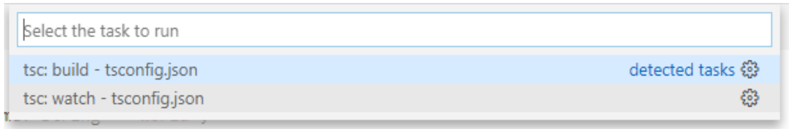
- Delete your **HelloWorld.js** file from the last exercise
  - Don't delete the TypeScript file.
- Open a Integrated Terminal window (Control - `) and type the following to create a tsconfig.json file easily

```
tsc –init
```

- Review the tsconfig file.
- Look at other options for values by deleting a value and asking Code to suggest a value with Control-Space. Try it on the target property (line 4).
- Under compiler options, update the target to ECMAScript 2015 (ECMAScript 6 or ES2016 or ES6).
- Close the file.

Run TypeScript tasks

- When a tsconfig.json file exists, Code will provide default tasks without the need of a tasks.json file.
- Select **Terminal / Run Task…**
  - This is the task of building the file and you can also reach this menu through the **Terminal / Run Build Task…** option.
- Select either of the **tsc** options of watch (continual background task) or build (one-time transpile).
  - The watch mode may transpile more than you wish if you have too many files but keeps you from building after every change in code.



Select the task to run
tsc: build - tsconfig.json detected tasks
tsc: watch - tsconfig.json

- Verify that the HelloWorld.js file has been recreated.
- Make the default build task a Configure Default Build Task… and set to either build or watch. This will allow you to access it easily through a keyboard shortcup of **Control+Shift+B** or the **Tasks / Run Build Task** menu option.

Run the HelloWorld app

- Click on the **HelloWorld.ts** file
- Format the file with a right-click on the editor window and select **Format Document**.
- Build the file (if you did not choose the watch mode) with

    - Control+Shift+B
      or
    - F1 and then type **build,** make sure it says **Run Build Task** or cursor to that option**,** and then hit **Enter** (also saves the file)
- Clicking back and forth between the two allows you to view them. A file will only be added to the Open Editors section when you edit it eliminating the need to close and open files to view them.
- Open another integrated terminal to the project's directory by clicking the large plus sign in the terminal's icon bar, and run the command (not case sensitive)

```
node helloworld
```

- Change the value of the name variable to your name, rebuild (automatic after save with watch), and rerun the app to get familiar with the workflow.

Set up optional Code tasks with a tasks file

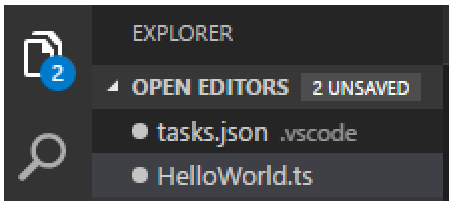
- Create a tasks.json file by selecting **Terminal / Configure Tasks…** then selecting **Create tasks.json file from template** and then selecting the **Others** option
- Read through the tasks.json file.
- Run the example task with **Terminal / Run Task…** then selecting **echo**. Then select "Never scan the task output" for this task.
- Rerun the task quickly with Control-P (Quick Open), and typing "task echo"
- Add a new task to run JavaScript through node with:

```
"problemMatcher": []
},
{
    "label": "node",
    "type": "shell",
    "command": "node ${fileDirname}/${fileBasename} "
}
```

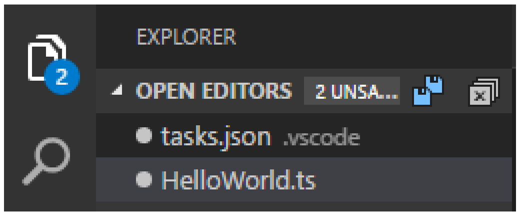
  - Open the HelloWorld.js file and then run the file with the node task by typing **Control-P**, then **task node**.

Noticing Code's visual clues

- Be aware of Code's icons and symbols for files
    - Type a space at the end of any file in an open editor
    - In the **Explorer** you'll notice
        - the **Open Editors** has the file with a large dot prefix,
        - the working folder TypeScript section of files has it highlighted also,



- the large dot next to the file name shows it's unsaved,
- the title bar of the editing window has a large dot to show it's unsaved, and
- the Explorer icon has count of the files that have not been saved in blue.



- Save the file by
  - clicking the Save All icon next to the Open Editors section when you hover there
    or
  - the standard File / Save method.
- Close the file.
  - Click the x in the title bar of the file to close the file editor tab.
- Also, close out all the open editors (window with tabs) by either
  - Clicking on the x next to each file name
    or
  - Clicking on the Close All icon next to the Open Editors section header when you hover there.

Hide non-modified files

- Since you never need to see the .js files when you are working in TypeScript usually, hide them by adding a settings property for the project.
- In VS Code, select **File / Preferences / Settings**. This will open the **User Settings** file in a custom GUI.
- Type the word **exclude** in the braces to set the files to exclude from showing using a glob pattern.
  - Click on files.exclude.
  - Add two new exclude lines of:

```
**/*.map
**/*.js: {"when": "$(basename).ts"}
```

- When you save the file, the .js files will disappear in the Explore panel. You will need to use the Refresh icon in the project if you still see .js files in folders that weren't open when you saved this.
- Other good settings properties to try are:
  - TypeScript > Update Imports On File Move: Enabled: always
  - Editor > Minimap Enabled: unchecked
  - Editor > Word Wrap: on

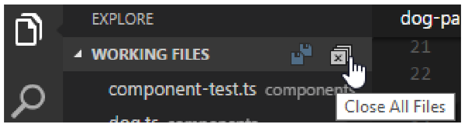
Show file icons (optional)

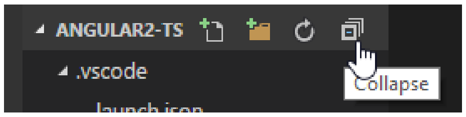
- Show some different icons or install and extension for more variations. The Seti theme is installed by going to File / Preferences / File Icon Theme and selecting Seti (Visual Studio Code).

- Select others after installing an Extension of your preference.

Clean up

- Close All Files in the Open Editors.
- Also, if you have lots of folders open, you can collapse them all at once with the project's Collapse icon.

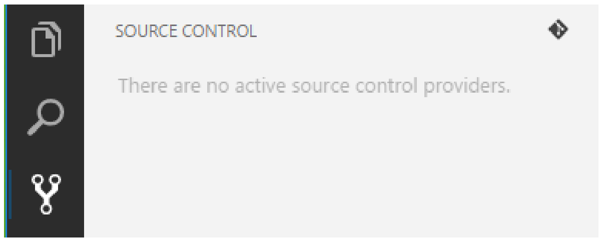




4. Set up VS Code with git (optional)

Create local git repo

- Install git on your machine from https://git-scm.com/ if you don't have it.
- Initialize a local repo now by clicking the git icon in the icon ribbon on the left side then clicking the git diamond in the upper right.



There are no active source control providers.

- Select the folder to put under git control and click Initialize Repository.
- Under the three dot icon where the diamond icon was, Commit All or Stage All Changes and then Commit Staged. The plus sign under the Changes section will stage and the checkmark icon at the top starts a commit.
- Provide a commit message.

Create remote repo

- Set up your empty repo folder at github, BitBucket, or Visual Studio.com.
- Get the required URL from the repo host. Once you have that URL, add it to the Git settings by running a command line action.

```
git remote add origin <remote URL>
```

- Then you can push your files. You will not have an upstream branch so you can accept Code's suggestion to do that or you can run this command before you push:

```
git push -u origin master
```

- To receive any updates you can affirm that you'd like to have Code periodically run 'git fetch' but that's not necessary for any of our projects where no one will be updating the code. If you wanted to keep the class repo up to date with a local repo, you could do that though.

5. Set up a local server (optional)

This will not be necessary for the Angular projects that are using the CLI since it comes with a local server installed. But it's useful when you are doing any other web page viewing without a Code extension.

- In your terminal window, type

```
npm install -g lite-server
```

- You should see a resulting output of the directory used to install the app from your npm root directory.
- Make sure the default path of the command prompt window is at the typescript directory. Then start up the server on the default port 3000 with

```
lite-server
```

- You will get an error message on the browser because of no file to load. That's OK for now.
- Create a file in Code in your **typescript** directory called **index.html** and type the following Emmet shortcut which expands with selecting the options that appears.

```
html:5<tab>
```

- You will be in a template editing mode and you can notice a light grey background. This means that you can tab around the template and fill in the parts that are editable.
- Update the file with
  - a title
  - some text in the body
- To leave this editing mode, press ESC and the add
  - and a script tag for your **HelloWorld.js** file
    Emmet: script[src=HelloWorld.ts]<tab>
- Your file should look similar to the one below.
- Navigate in the browser to

```
http://localhost:3000/typescript/index.html
```

- You should see your page.
- Open the dev tools (F12) and check the console message.

HTML

typescript / index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home Page</title>
    <script src="HelloWorld.js"></script>
```

```
</head>
<body>
    This is my home page!
</body>
</html>
```

Angular setup

6. Setup using CLI project with Code

Shortcut to writing files

- Files for the basic template and some for other exercises when noted can be copied from my Github repo:
    - https://github.com/doughoff/WD-530v9
    - To copy an entire file, use the raw button at the top of the file listing to view only code so you can select all and copy.

Install the Angular CLI

- In a command prompt from the desktop (C:\Users\Student\Desktop) , install the Angular CLI:

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

- Then follow these commands to scaffold a new project inside a new folder.

```
ng new angular
```

- When asked about adding **Angular routing**, go ahead and add it even though it won't be a part of this course. Then choose the **CSS stylesheet format** unless you are familiar with the another (this will take about 3 minutes).
- Then switch to the new folder.

```
cd angular
```

- Update all the packages that are declared in the package.json file with

```
npm install
```

- If you get a message to resolve any vulnerabilities, you might feel like doing what it says about the audit. Don't do it. In the past it has created breaking changes. Currently none are usually found.
- Close Visual Code and reopen it with just the Angular folder. Run the task from Terminal/Run Task… or even better, launch the project from a new terminal so you can control the port:

```
ng serve --port 80
```

- You can load the project by navigating to **localhost** in your browser. It should say **angular app is running!**
- You won't need to refresh the browser since there is a live-reload running which will trigger the build and reload on any change but you can always manually build the project to the /dist directory with

```
ng build
```

- The build uses webpack to find the initial app which isn't declared in the src/index.html so you can't just use the code to do a project without knowing how webpack works. It's the build tool of choice right now.
- The other commands that can be used to add a component or other code to the project look like these:

```
ng generate component components/new-component
ng generate directive directives/new-directive
ng generate route routes/new-route
      --default
      --lazy
ng generate pipe pipes/new-pipe
ng generate service services/new-service
```

- More information is at https://cli.angular.io/

Hide more files in Code (optional)

- After you open the angular project In VS Code, select **File / Preferences / Settings**. Other files to exclude (Files:Exclude) with glob patterns might be:

```
        **/dist
        **/node_modules
        **/tmp
        **/typings
```

- Close all your open editors.

Set up TypeScript typings (optional)

- A file called **typings.json** may be used to map type definitions to the standard packages used in this project with these settings copied from the Github site:

typings.json

```
{
  "name": "angular",
  "dependencies": {},
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160725163759",
    "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
    "node": "registry:dt/node#6.0.0+20160909174046"
  }
}
```

- This would be necessary if you were trying to use the datatypes from these packages to code to, so the editor would know how to check function calls, new variable refs, etc.

Set up TSLint (optional)

- A linter for TypeScript was included in the development dependencies when you installed the Angular files. You can insert this into your VS Code workflow by adding an extension and the config file.
- Search for the extension TSLint and install it.
- Then go back to File / Preferences / Settings and search for TSLint to see the options to configure.
- Open the tslint.json file that is in the project. In the rules secion, create a new line and have Code autosuggest a new rule with Control-Space.
- You can download a TSLint.json file from the Github site for more rules.

Set up style frameworks and the module loader

- Open the **src/index.html** file for our default page of the site with the Angular container (the app-root element) on it. Use the code below or copy it from the Github site.
  - There will be one error on the page. Ignore it.
- The base CSS style will be the second most popular CSS framework, Zurb's Foundation. You can use Twitter's Bootstrap as well as a CSS reset file later if you prefer.
  - You can copy links into the index.html file for Foundation at **https://get.foundation/sites/docs/installation.html** and paste them as below or…
  - You can install Foundation locally via npm and then use the styles and scripts sections of the **angular.json** file to reference them.
  - Font Awesome was referenced via the cdn of **cdnjs**.

/ src / **index.html**

```
   <!DOCTYPE html>
   <html>

   <head>
       <meta charset="utf-8">
       <title>Angular</title>
       <base href="/">

       <meta name="viewport" content="width=device-width, initial-scale=1">
       <link rel="icon" type="image/x-icon" href="favicon.ico">

   <!-- Stylesheets - Foundation + Font Awesome + site -->
   <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.12.1/css/all.min.css" integrity="sha256-
mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />

   <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/foundation/6.6.1/css/foundatio
n.min.css" integrity="sha256-Q2fXBbKhtsyhYrSLa7hXCNV+FdhbQhyrjks8Kic0u/U="
crossorigin="anonymous" />
```

```
        <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/foundation/6.6.1/css/foundatio
n-float.min.css" integrity="sha256-
4ldVyEvC86/kae2IBWw+eJrTiwNEbUUTmN0zkP4luL4=" crossorigin="anonymous" />

   <link rel="stylesheet" type="css" href="./styles.css">

   <!-- JavaScript libraries - jQuery + Zurb Foundation -->
   <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"
integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
crossorigin="anonymous"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/foundation/6.6.1/js/foundation.
min.js" integrity="sha256-tdB5sxJ03S1jbwztV7NCvgqvMlVEvtcoJlgf62X49iM="
crossorigin="anonymous"></script>
   </head>

   <body>
       <div class='callout'>
           <app-root>
               <i class="fa fa-spinner fa-pulse"></i>Loading...
           </app-root>
       </div>
       <p>The rest of the page</p>
   </body>

   </html>
```

- Foundation will clip the spaceship icon and a few of the other cards.
- Create other CSS styles to link to the index page in **styles.css**. Here are the styles that are used on the Angular site.

Create an Angular module

- To properly organize our files, we'll put each app into its own directory. Create a directory called **basic** in **src**. All the references to files will be in the **src** folder from now on unless noted.
- Then update the bootstrapping component called **main.ts** in **src**. Here's the content with the one change:

/ main.ts

```
   import { enableProdMode } from '@angular/core';
   import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';

   import { AppModule } from './basic/app.module';
   import { environment } from './environments/environment';

   if (environment.production) {
```

```
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

- Saving your TypeScript files now will show an error in Code from the lack of the **./app.module**.
- Now create the module configuration file with this content, or copy it from the Github site:

/ basic / app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { BasicComponent }  from './basic.component';

@NgModule({
  imports:      [ BrowserModule  ],
  declarations: [ BasicComponent ],
  bootstrap:    [ BasicComponent ]
})
export class AppModule { }
```

- You probably should follow the style of using the name for the module (**App**Module should be **Basic**Module) also in production code. Not renaming this saves you some dull work during your class exercises though.
- This next new file is the component file which calls to the external CSS and HTML files using this content:

/ basic / basic.component.ts

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

- The stylesheet and template are basic files and you can modify them however you like. The stylesheet for the module is waiting for your custom styles but has a comment in it and looks like this:

basic / basic.component.css

```
/* component specific stylesheet */
```

- And the basic html template for the module looks like this:

basic / basic.component.html

```
<h1>A basic template for an Angular app!</h1>
<h6>Now at version 9!</h6>
```

Watch for TypeScript changes and run a server with Live Update

- Take a look at the scripts section of the **package.json** to see what commands can be run. We will use two. But since the start command runs both we can use one terminal window.
- From Visual Studio Code, select **View / Terminal** to get a command prompt.
    - When switching back to the terminal, the keyboard shortcut Control-backtick is useful.
    - Make sure the previous web server instance of lite-server has been stopped.
- To start up the lite-server via npm's task runner and transpile the TypeScript first, type

```
npm start
```

- Check out the other scripts in the package.json file.
- Now when you make a change to a TypeScript file, it will automatically transpile.
- This should also launch your default browser and render the page. When you make a change and save it, the files are being watched so the browser will reload.
- You should see a headline for **A basic template for an Angular app! Now at version 9!**
- For a little more customization you can create a bs-config.json file in your project before starting it up with the following code:

bs-config.json

```
{
    "injectChanges": false,
    "files": ["./**/*.{html,htm,css,js}"],
    "watchOptions": { "ignored": "node_modules" },
    "port": 3000,
    "server": {
        "baseDir": "./",
        "middleware": []
    }
}
```

- To stop the server, use Control-C in the terminal and terminate the batch job. Keep it running though throughout the exercises and start it up again with **npm start** (or **lite-server** if you don't need to transpile first) if you need to because it might occasionally crash.

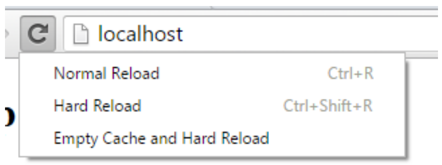
7. Load CLI project with local server extension (optional)

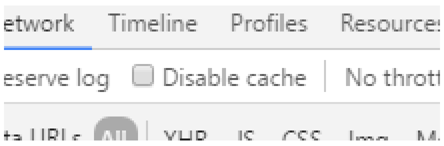
- Make sure the project is built and has a /dist folder.
- Open another instance of VS Code to load the **/dist/angular** folder so that index.html is at the top level.
- Choose the last icon on the icon ribbon on the left side.
- Search the Marketplace for **Live Server** and install it.
    - This will add a Go Live "button" at the lower right of the Code window.

- Click the Go Live button. It will load the project automatically into a new browser window to port 5500.
    - This will not work on the non-distribution code.

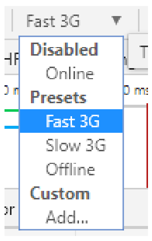
8. Measure template's resource loading times

- Use the Network tab of Chrome's Dev Tools (F12) to answer these questions after you refresh the page to record data about the page. Most answers are found in the bottom line called the Summary View.
  - What is the average initial **load** time (empty cache and hard reload or check the Disable Cache checkbox) of the template (styled in red at bottom of page)?





    - 1st __________ ms
    - 2nd __________ ms
    - 3rd __________ ms
    - __________ ms averaged
  - What is the initial load time of the template when throttled down to **Fast 3G**?



    - __________ ms
  - What was the total file size **transferred** the first time when not cached?
    - __________ MB
  - What is the average normal (cached) **load** time of the template?
    - __________ ms
  - What is the total file size when normal (cached)?
    - __________ KB
  - What is the largest file being transferred? (click on size heading)
    - ____________________________
  - Which file took the longest being transferred? (click on time heading)
    - ____________________________
    - Google this file to find out what it contains.

Components

9. Create a template

- Create a template folder in your src folder.
- The CLI project now contains these:
  - app - the default CLI application
  - basic - our first attempt to create an application and a playground to try out new ideas
  - template - our decision to use this code for future exercises
- Copy the contents of your **basic** directory in VS Code and paste to create copies of all the files. Rename the new **basic copy** directory to **template**. Then rename the files in it as follows allowing Code to update imports.

From	To
basic / **basic**.component.css	template / app.component.css
basic / **basic**.component.html	template / app.component.html
basic / **basic**.component.ts	template / app.component.ts

- Update the references in the new files with the bold code:

template / app.component.html

```
    <h1>Angular app content from template</h1>
```

template / app.component.ts

```
      templateUrl: 'app.component.html'
      styleUrls: ['app.component.css'],
    })
    export class AppComponent {
```

template / app.module.ts

```
    import { AppComponent }  from './app.component';

    @NgModule({
      imports:       [ BrowserModule  ],
      declarations:  [ AppComponent ],
      bootstrap:     [ AppComponent ]
    })
    export class AppModule { }
```

/ main.ts

```
import { AppModule } from './template/app.module';
```

- Here's a little styling from Foundation.

/ index.html

```
  <body>
    <div class='callout'>
      <app-root> <i class="fa fa-spinner fa-pulse"></i>Loading... </app-
root>
```

- Restart the running task of the local server (Terminal / Restart Running Task… - npm: start). Angular caches so much it doesn't recognize just a change in the folder. It will, however, pick up a change to the contents of the folder and rerender then.
- View the page in your browser (localhost:4200). You should see "Angular content from template" and "the rest of the page."
- Make any final changes to your template that you want to reuse in the upcoming exercises.

10. Use different selectors

Create new component from template

- Copy the contents of your **template** directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **selectors**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './selectors/app.module';
```

- Change the heading in the external html of the component.

/ selectors / app.component.html

```
<h1>Selectors</h1>
```

- Restart the running task of the local server (Terminal / Restart Running Task… - npm: start).

Test CSS selectors

- Allow multiple selectors in the root component file by adding a few more to the selector property of the Component decorator

app / app.component.ts

```
    import { Component } from '@angular/core';
    @Component({
      selector: 'app-root, .app-root, [app-root]',
        styleUrls: ['app.component.css'],
        templateUrl : 'app.component.html'
```

- The index file has an element selector for the root template now. Add another option to start a root component after the first with

index.html

```
        </app-root>
      </div>
      <div class='app-root'>Loading from class selector...</div>
    <p>The rest of the page</p>
```

- What happens when you load this file?
- Add another new element for calling the component with an attribute as follows:

index.html

```
    <div class='app-root'>Loading from class selector...</div>
    <div app-root>Loading from attribute selector...</div>
```

- View the file again and notice any changes.

Questions

- Do the selectors and content differ from the source code in the rendered code of the Dev Tools Elements tab in any way?
- Can you define a site level or page level CSS class with the same name as the class selector for the component to modify the style of the element? Add a style element in your head element for a page level style rule for the class component selector.app-root and find out.
- Can you define and use a page level CSS element selector (h1) that will modify the html in the Angular custom component?

11. Add more style

Create new component from template

- Copy the contents of your **selectors** directory in VS Code and paste to create copies of all the files. Rename the new **selectors copy** directory to **selectors2**.
- Update the bootstrap module:

/ main.ts

```
• import { AppModule } from './selectors2/app.module';
```

- Change the heading in the external html of the component.

/ selectors2 / app.component.html

```
<h1>Selectors 2</h1>
```

- Restart the running task of the local server (Terminal / Restart Running Task... - npm: start).

Adding style

- To confirm that we are just updating the style in the root component, remove the extra app-root elements and add some more html as follows:

/ index.html

```
    <div class='callout'>
      <app-root>
        <i class="fa fa-spinner fa-pulse"></i>Loading...
      </app-root>
    </div>
    <div class="success callout">
      <div>The rest of the page</div>
    </div>
```

- Then we'll update the component template with a similar piece of code. We can use the internal template or the external template.
- The first option with the internal template is to take advantage of using backticks for doing template literals with **the dollar sign and single braces** in ES6 and show off a little of the ES6 magic for evaluating expressions. Remember this is NOT Angular!

selectors2 /app.component.ts

```
  styleUrls: [`app.component.css`],
  template:
   `<div class="success callout">
    <h1>An Angular v${4 + 5} App</h1>
    </div>`
  })
```

- Review the rendered code.
- The second option is to use the external template by using the templateUrl property set to like this:

selectors2 /app.component.ts

```
 styleUrls: [`basic.component.css`],
 templateUrl: 'app.component.html'
 })
```

- If you place that same internal html in your external template, you get the same magic through Angular expression evaluation but using its double mustache braces. Here's the template code.

selectors2 /app.component.html

```
<h1>Selectors 2</h1>
<div class="success callout">
    <h1>An Angular v{{4 + 5}} App</h1>
</div>
```

Save the file. You should see a full-width green callout box.

Add style with decorator styles:

- You can also update the component with the following code to add styles and removing the styleUrls.

selectors2 / app.component.ts

```
    selector: 'app-root',
    styles: [
    `.success { background-color: lightcyan; }`,
         ` h1 { color: lightslategray; }`
    ]
,
    templateUrl: 'app.component.html'
})
```

- I prefer the external sheet because it doesn't require parsing the JavaScript, so let's switch. Use the external template to add these styles to your template with the switch back to the styleUrls

selectors2 / app.component.ts

```
    selector: 'app-root, .app-root, [app-root]',
    styleUrls: ['app.component.css'],
    templateUrl: 'app.component.html'
}
```

selectors2 / app.component.css

```
/* component specific stylesheet */
.success {
   background-color: lightcyan;
}
h1 {
      color: goldenrod;
   }
```

- Or try putting them inline in the template itself at the top in a style element somewhat like having it in the head element:

app /app.component.html

```
   <style>
   .success {
      background-color: wheat;
   }
   h1 {
      color: brown;
   }
   </style>

   <h1>Selectors 2</h1>
```

- Save and view in the browser.

Questions

- Does the external CSS file and the “inline” style element in the template work together?
- Where does the style get rendered in the page? At the top of the template or, more intelligently, in the head element where it belongs?
- How does the styles panel in the DevTools show the location of this CSS code in the external template?

Templates

12. Get text from component

Create new component from template

- Close all your open editors.
- Copy the contents of your **template** directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **templates1**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './templates1/app.module';
```

- Change the heading in the external html of the component and add another element.

/ templates1 / app.component.html

```
<h1>Templates 1</h1>
   <div class="success callout">
       <div>{{'The interpolated template in this ' + title}}</div>
   </div>
```

- Restart the running task of the local server (Terminal / Restart Running Task… - npm: start).

Accessing text

- Add the text to access. A type is inferred from the static string already:

/ templates1 / app.component.ts

```
     selector: 'app-root',
     templateUrl: 'app.component.html',
     styleUrls: ['app.component.css']
   })
   export class AppComponent {
     private title = 'app works!';
   }
```

- Save all the files. The browser should have a callout that says “The interpolated template in this app works!”

Make template updates

- Make changes to the template like below. Save and check the results. This adds a pipe to the result string.

/ templates1 / app.component.html

```
<div class="success callout">
       <div>{{'The interpolated template in this ' + title | uppercase}}
       </div>
   </div>
```

- Add another reference to a class property and check the results of this change

/ templates1 / app.component.html

```
   <div class="success callout">
       <div>{{'The text of this field is ' + property1 }}</div>
   </div>
```

- Now give the property a value in the component class

/ templates1 / app.component.ts

```
   export class AppComponent {
       private title = 'app works!';
       private property1 = 'a property';
   }
```

- Save the file to see the new results.
- Set up the constructor of the class to properly initialize the property and refresh the browser again

templates1 / app.component.ts

```
     private property1  = 'a property';
     constructor() {
       this.property1 = 'a property from the constructor';
     }
   }
```

- Save the file to see the new results.
- Now add an accessor for the property but update the other references so they don't conflict. You can also drop the initialization in the field declaration since we do that in the constructor.

templates1 / app.component.ts

```
     private _property1: string;
     constructor() {
       this._property1 = 'a property from the constructor';
     }
     public get property1(): string {
       return this.property1 + ' from the accessor';
     }
```

- Save the file to see the new results.
- Let's add one more problem into the template and fix it this time. When the field was first loaded it had an undefined value. We'll try that again with a new variable but protect it from the undefined message. Here's the updated code to use in the template:

templates1 / app.component.html

```
   <div class="success callout">
       <div>{{'The text of this field is ' + property1 }}</div>
       <div>{{'The text of the field is ' + ( property2 || 'not
there') }}</div>
```

```
    </div>
```

- Save the file to see the new results.

13. Set attributes from component

Create new component from previous exercise

- Copy the contents of your **templates1** directory in VS Code and paste to create copies of all the files. Rename the new **templates1 copy** directory to **templates2**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './templates2/app.module';
```

- Change the heading in the external html of the component and add another element.

/ templates2 / app.component.html

```
<h1>Templates 2</h1>
   <div class="success callout">
   </div>
```

- Restart the running task of the local server (Terminal / Restart Running Task… - npm: start).

Adding properties

- Let's add some properties for an image, initialize them in the constructor, and provide a few accessors for them. Remove the code from the AppComponent class and use the code below. Pick your own URL from a Google Images search if you don't want to copy mine.

/ templates2 / app.component.ts

```
   export class AppComponent {
     private _src: string;
     private _alt: string;
     constructor() {
       this._src = 'http://www.dogbreedplus.com/dog_names/images/funny-dog-
names.jpg';
       this._alt = 'funny dog';
     }
     get src(): string { return this._src; }
     get alt(): string { return this._alt; }
   }
```

- To keep your image from getting too big, add a scoped style to your file:

/ templates2 / app.component.css

```
   img {max-height: 150px;}
```

- Then use the accessors (no parentheses used in accessor methods) to provide values to the DOM property of the image in the template:

/ templates2 / app.component.html

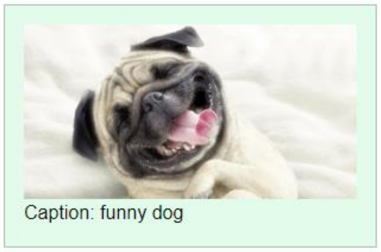
```
    <div class="success callout">
        <img [src]='src' [alt]='alt' [title]='alt'>
    </div>
```

- Save the file to see the new results.
- Now let's reuse the alt attribute's value for a caption to the image. This version uses a local variable of #dogs to refer to the image's alt property but without the dogs variable, the alt variable would refer to the component class' variable and get the same result.

/ templates2 / app.component.html

```
    <div class="success callout">
    <figure>
        <img [src]='src' [alt]='alt' [title]='alt' #dogs >
        <figcaption>{{'Caption: ' + dogs.alt}}</figcaption>
    </figure>
    </div>
```

- Save the file to see the new results.
- Add a caption rule for the component in your external stylesheet file:


Caption: funny dog

/ templates2 / app.component.css

```
    figcaption {
            font-size: .7em;
    }
```

- Check the page out to make sure it looks good.

Use style property binding

- And we should put on a border on the callout which we'll do as a style property binding. Styles are not as reusable as classes but still might be needed at some point. Add a property to your component:

/ templates2 / app.component.ts

```
    private _alt: string;
    needsBorder: boolean;
```

- Initialize the property in the constructor

/ templates2 / app.component.ts

```
    this._alt = 'funny dog';
    this.needsBorder = true;
```

- Then use the property to conditionally add the border in the template

/ templates2 / app.component.html

```
    <div class="success callout"
      [style.border] = "needsBorder ? '5px solid lightblue' : 'none' ">
    <figure>
```

- Save the files and check the page out. You should have a wider blue border.

  For excellent code, the booleans should be paired with accessors (_needsBorder, get needsBorder()) instead of directly accessing the property but it makes for shorter code in the exercises.

Class property binding

The callout currently has a green background because of the success class. But we'd like to use other classes available in Zurb's Foundation CSS for hiding, floating, and background color in case it's not a success. We'll also add the enclosing figure element to make it semantically correct.

- Add a few more boolean properties to the component class which will help us add some classes.

/ templates2 / app.component.ts

```
    private needsBorder: boolean;
    private isSuccess: boolean;
    private isHidden: boolean;
    private isFourColumns: boolean;
```

- Initialize them in the constructor. This is where you can change the values to test out the visual changes on the page when the class is not loaded.

/ templates2 / app.component.ts

```
    this.needsBorder = true;
    this.isSuccess = true;
    this.isHidden = false;
    this.isFourColumns = false;
```

- Then update your template to use those properties to load the special Foundation classes of warning, hide, and small-4 with the ngClass multiple state checker. Remove the class of **success** and watch out for the mixed quotes. Here's the final version:

/ templates2 / app.component.html

```
    <div class ="callout"
     [style.border] = "needsBorder ? '5px solid lightblue' : 'none'"
     [ngClass]="{
  success: isSuccess,
  warning: !isSuccess,
  hide: isHidden,
 'small-4': isFourColumns }"
     >
     <figure>
            <img [src]='src' [alt]='alt' #dogs>
            <figcaption >{{'Caption: ' + dogs.alt}}</figcaption>
     </figure>
     </div>
```

- Save and look at the changes and then change the values of the properties in the constructor to see how it changes the rendered component. Change
    - isHidden to true and back
    - isFourColumns to true

14. Use a dog-panel model class

Create new component from template

- Close all your open editors.
- Copy the contents of your **template** directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **dog-panel**.
- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './dog-panel/app.module';
```

- Change the heading in the external html of the component and add another element.

/ dog-panel / app.component.html

```
<h1>Dog panel</h1>
```

- Restart the running task of the local server (Terminal / Restart Running Task… - npm: start).

Create and test the model class

- We need a model. This is a view model for holding and formatting information necessary for the view. Create a new directory called **model** in your **dog-panel** module directory.
- Copy a file from Github called **dog.ts** and put it in the **dog-panel/model** directory.
- Update and save this information in the file:

dog-panel / model / dog.ts

```
    export class Dog {
        private _balance: number;
        private _name: string;
        private _age: number;
        private _breed: string;
        private _isMale: boolean;
        private _lastOfficeVisit: Date;
        private _imageUrl: string;

        constructor(balance: number, name: string, age: number, breed:
string,
        isMale: boolean, lastOfficeVisit?: Date, imageUrl?: string) {
            this._balance = balance;
            this._name = name;
            this._age = age;
            this._breed = breed;
            this._isMale = isMale;
            this._lastOfficeVisit = lastOfficeVisit || new Date();
            this._imageUrl = imageUrl ||
'http://icons.iconarchive.com/icons/icons8/windows-8/512/Animals-Dog-icon.png';
   console.info('Created', this);
       }

       get balance()       : number {return this._balance;}
       get name() : string {return this._name;}
       get age()     : number {return this._age;}
       get breed(): string {return this._breed;}
       get isMale()        : boolean {return this._isMale;}
       get lastOfficeVisit()     : Date {return this._lastOfficeVisit;}
       get imageUrl()            : string {return this._imageUrl;}

       get gender() : string {return (this._isMale ? 'male' : 'female');

       get oneLine(): string {
           return `${this.name}, a ${this.isMale ? 'male' : 'female'}
${this.breed}, is ${this.age} years old.
   Last office visit: ${this.lastOfficeVisit}
   Balance: $${this.balance}
           `;
   }
   }
   //---------------------------------------------------------- testing
      let dog:Dog = new Dog(1, "Test Dog", 5, "test breed", true);
         console.log(dog.oneLine);
```

- Build and run the model code with the test below in a powershell terminal window with

```
•  tsc --target es5 src/dog-panel/model/dog
•  node src/dog-panel/model/dog
```

- You should get some information about a dog.

You'll see the information again on the browser console when the class is loaded and be marked with an info icon.

Update the dog panel component

- Start using the Dog model class in the component file by importing Dog at the beginning of the file:

dog-panel/app.component.ts

```
   import { Component } from '@angular/core';
   import { Dog }         from './model/dog';
```

- Add a few styles for use with the dog panel component in the external style sheet. You'll see these used in the template.

dog-panel / app.component.css

```
.c-imagePlacement1 {margin: 1em 1em 3em 0; max-height: 5em}
.c-imagePlacement2 {margin: .5em .5em 1.5em 0; max-height: 2.5em}
```

- For now, we'll set up a default dog to use in our component file through the constructor that initializes a property. Feel free to play with this code.

dog-panel /app.component.ts

```
export class AppComponent {
    private _dog: Dog;

    constructor() {
        this._dog = new Dog(1, 'Rover', 5, "mongrel", true);
        console.info('Created', this);
    }

    get dog() { return this._dog; }
}
```

- Run the page again in the browser to look at the console output. Why are you getting three output statements?
- Alter the static text in the console.info call to verify that you are correct.

Update the dog panel component template and view

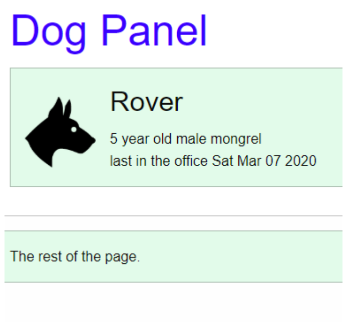
- Update the code in the template to look like

dog-panel / app.component.html

```
<div class="success callout small-4">

<img src="{{dog.imageUrl}}" title="{{dog.name}}" alt="{{dog.name}}"
class='float-left c-imagePlacement1'>

<h3>{{dog.name}}</h3>
<p>
{{dog.age}} year old {{dog.gender}} {{dog.breed || 'unknown breed'}}
   <br/>
last in the office
{{dog.lastOfficeVisit?.toDateString() || 'never'}}
</p>

</div>
```

- You can format it how you like and use the right-click Format Code command to make it look good.

- Now view the dog panel page in the browser and you should get something like the image on the right.


Dog Panel

Rover

5 year old male mongrel
last in the office Sat Mar 07 2020

The rest of the page.

View logic

15. For directive

- We'll use most of the pages from the dog-panel exercise and just add new templates to play with the view logic. We'll need some mocked data to play with so we'll create a class to supply that until we do a service.

Get mocked data to loop over

- Create a file in the **dog-panel/model** directory called **dog-data.ts** and use this code found on the Github site for our mock data:

dog-panel / model / dog-data.ts

```
import {Dog} from './dog';

export class MockDogs {
  static SIX : Dog[] = [
new Dog(245.75, "Bailey", 5, "Labrador", true, new Date(2019,0,1)),
new Dog(21.22, "Max", 6, "German Shepherd", true, new Date(2019,7,1)),
new Dog(0, "Bella", 7, "Golden Retriever", false, new Date(2015,2,1)),
new Dog(0, "Buddy", 8, "Bulldog", true, new Date(2018,5,1)),
new Dog(0, "Lucy ", 9, "Beagle", false, new Date(2017,6,1)),
new Dog(0, "Molly", 10, "Yorkshire Terrier", false, new Date(2016,7,1))
]
}
```

- Update the component class to provide that data to the template with this code:

dog-panel / app.component.ts

```
import {Dog}            from './model/dog';
import {MockDogs}       from './model/dog-data';
```

- and declare a property in the class to hold the data with

dog-panel / app.component.ts

```
    private _dog: Dog;
    mockDogs : Dog[] = MockDogs.SIX;
```

- Run the code in the browser again and check out the console messages.
- Copy the **dog-panel\app.component.html** to a new file called **app.component2.html** and make it look like:

dog-panel / app.component2.html

```
<h1>Dog Panel v2</h1>
<div class='row'>
<div *ngFor="let dog of mockDogs" class="success callout small-4 columns
">
    <img src="{{dog.imageUrl}}" title="{{dog.name}}" alt="{{dog.name}}"
class='float-left c-imagePlacement1'>
```

```
        <h3>{{dog.name}}</h3>
        <p>{{dog.age}} year old {{dog.gender}} {{dog.breed || 'unknown
breed'}}
    last in the office {{dog.lastOfficeVisit?.toDateString() ||
'never'}}</p>
    </div>
    </div>
```

- Update the template in the component to use the template file.

dog-panel / app.component.ts

```
        selector: 'app-root',
        templateUrl: 'app.component2.html',
```

- Save the files and take a look at the dog panels in your browser now.

You are supposed to add a class of **.columns** to the child elements of the row in Foundation but I didn't see any difference in the rendered result. So it's not there.

Use a counter in the for loop

- We want to add a counter to each iteration of the loop. Add an index to the for loop with

dog-panel / app.component2.html

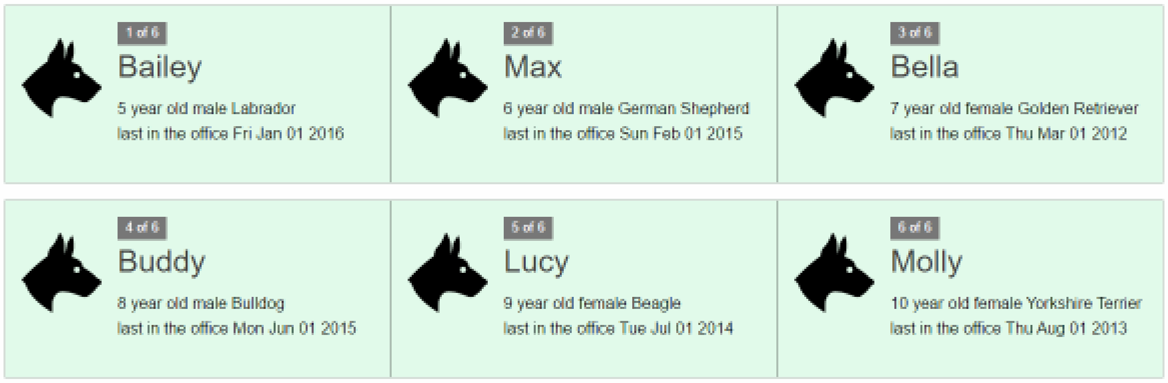
```
    <div class='row'>
    <div *ngFor="let dog of mockDogs; let i = index;" class="success callout
small-4 columns ">
```

- Add an element to show the index as part of the total mock dogs being shown. Indexes are zero-based so we bump up the values by one.

dog-panel / app.component2.html

```
    <div *ngFor="let dog of mockDogs; let i = index" class="success callout
small-4 columns">
        <span class='secondary label'><span id='dogIndex'>{{i+1}}</span> of
{{mockDogs.length}}</span>
```

- Save the file and view the changes.



1 of 6
Bailey
5 year old male Labrador
last in the office Fri Jan 01 2016

2 of 6
Max
6 year old male German Shepherd
last in the office Sun Feb 01 2015

3 of 6
Bella
7 year old female Golden Retriever
last in the office Thu Mar 01 2012

4 of 6
Buddy
8 year old male Bulldog
last in the office Mon Jun 01 2015

5 of 6
Lucy
9 year old female Beagle
last in the office Tue Jul 01 2014

6 of 6
Molly
10 year old female Yorkshire Terrier
last in the office Thu Aug 01 2013

16. If directive

- Even though you could write the business logic directly into the view, it's better to have that logic as part of the entity.

Add a label for senior dogs

- In the template file, make the following update:

dog-panel / app.component2.html

```
       <span class='secondary label'><span id='dogIndex'>{{i+1}}</span> of
{{mockDogs.length}}</span>
       <span *ngIf="dog.isSenior" class='warning label'>Senior</span>
```

- Now add the getter to declare a rule about when a dog becomes a senior dog using the call from the template. In the Dog model class update the code with:

dog-panel / model / dog.ts

```
       get gender(): string { return (this._isMale ? 'male' : 'female'); }
       get isSenior(): boolean {return this.age >= 7;}
```

- Build the files and view the page to see that the dogs have the labels on them that match the rule.
- Let's put a marker on each callout when the balance is high. It's a simple enough rule to put in the template but it really should be managed by the Dog class like we did above. You can write the method to be a good programmer a little later. Add this code

dog-panel / app.component2.html

```
       <span *ngIf ="dog.isSenior" class='warning label'>Senior</span>
       <span *ngIf ="dog.balance > 200" class='alert label'><i class="fas
fa-dollar-sign"></i></span>
```

- Save the file and make sure we have one alert on a dog.

17. Switch directive

- We'll add a type of exam (this is a vet's office) for a dog when they have not been in the office for a certain number of months. In using a switch we need to match against a specific value. We'll get that again from a rule in the Dog class. Update your Dog model class **dog.ts** with

dog-panel / model / dog.ts

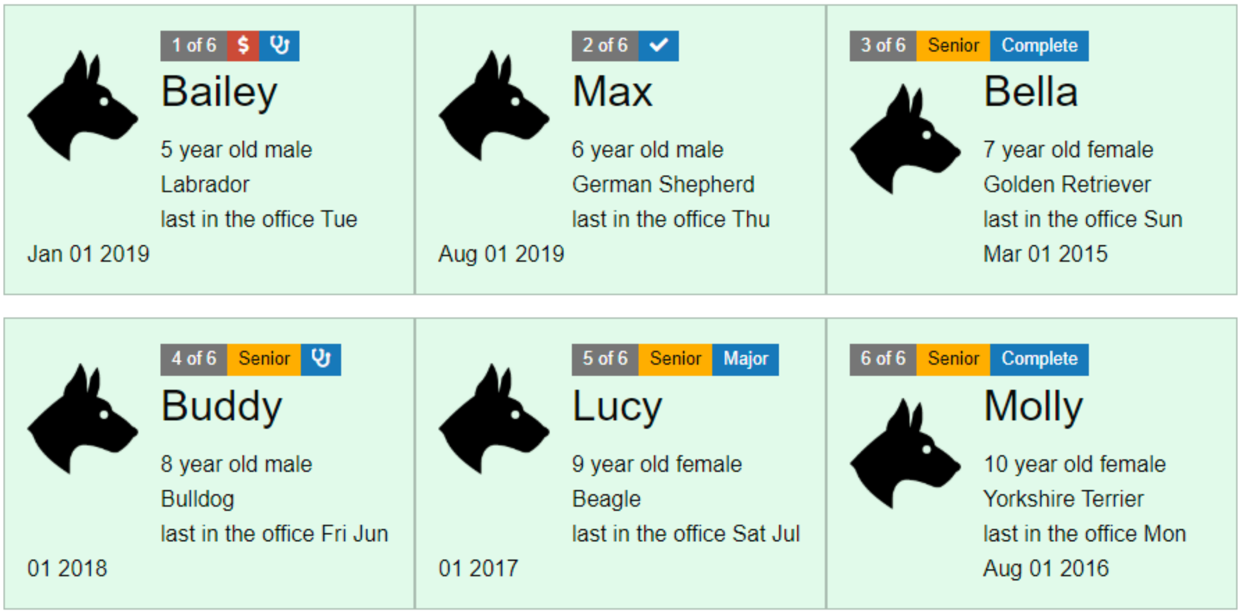
```
        get isSenior(): boolean { return this.age >= 7; }
        get examType(): string {
            let oneMonthInMs: number = 1000 * 60 * 60 * 24 * 30;
            let elapsedMonths: number = ( new Date().getTime() -
this.lastOfficeVisit.getTime() ) / oneMonthInMs;
            if (elapsedMonths <  6) { return ""; }
            else if (elapsedMonths <  12) { return "checkup"; }
            else if (elapsedMonths <  24) { return "routine"; }
            else if (elapsedMonths <  36) { return "major"; }
            else return "complete";
        }
```

- Now we'll use that information to put another label or icon on the callout of each dog with a switch statement in the template

dog-panel / app.component2.html

```
    <span *ngIf ="dog.balance > 200" class='alert label'><i class="fa fa-
dollar"></i></span>
    <span [ngSwitch]="dog.examType">
      <span *ngSwitchCase="'checkup'" class="label"><i class="fas fa-
check"></i></span>
      <span *ngSwitchCase="'routine'"class="label"><i class="fas fa-
stethoscope"></i></span>
      <span *ngSwitchCase="'major'" class="label">Major</span>
      <span *ngSwitchCase="'complete'"class="label">Complete</span>
      <span *ngSwitchDefault> ? </span>
    </span>
```

- Build the files and view in the browser to make sure that the dogs have the proper exam types on them now. The template should look like this:



1 of 6 $
Bailey
5 year old male
Labrador
last in the office Tue Jan 01 2019

2 of 6 ✓
Max
6 year old male
German Shepherd
last in the office Thu Aug 01 2019

3 of 6 Senior Complete
Bella
7 year old female
Golden Retriever
last in the office Sun Mar 01 2015

4 of 6 Senior
Buddy
8 year old male
Bulldog
last in the office Fri Jun 01 2018

5 of 6 Senior Major
Lucy
9 year old female
Beagle
last in the office Sat Jul 01 2017

6 of 6 Senior Complete
Molly
10 year old female
Yorkshire Terrier
last in the office Mon Aug 01 2016

Pipes

18. Common pipes

Date pipes

- In the last exercise, you used a date to show the last office visit, but it didn't look too good. Improve on that by changing the code in the template.
- Remove the clumsy JavaScript toDateString( ) formatting first. For a short date use:

dog-panel / app.component2.html

```
   last in the office {{( dog.lastOfficeVisit | date:'shortDate') ||
'never'}}</p>
```

- Refresh the browser to see the difference.
- Try the long date format with

dog-panel / app.component2.html

```
   last in the office {{( dog.lastOfficeVisit | date:'longDate') ||
'never'}}</p>
```

- You also can try the 'mediumDate' parameter in place of the other parameters or remove the parameters all together like so:

dog-panel / app.component2.html

```
    | date:'mediumDate'
    | date:''
```

Currency pipes

- To put the current balance of the dog on the template, add a line at the bottom of the callout like this:

dog-panel / app.component2.html

```
   last in the office {{(dog.lastOfficeVisit | date:'longDate') ||
'never'}}
   <span *ngIf = "dog.balance > 0">Balance: {{dog.balance |
currency:'USD':true:'1.2-2' }}</span>
   </p>
```

- Switch to euros by changing the currency type from USD to EUR (Euros). Do you get a new symbol?
- Try MXN (Mexico), CAD (Canada Dollars) or others.

19. Async pipe

- So, this is a really cheesy sales web site for a vet app. You get to look at the dog data for four seconds before you get a call to action in your face. Here's some interesting code to show you how to do that using a "secondary template" brought in with a property binding that the pipe will get for us.
- This is something we don't want to use after this exercise, so we'll create another template for it. Copy dog-**panel\app.component2.html** and paste to get **dog-panel\app.component3.html.**
- Update your heading in the template:

dog-panel \ app.component3.html

```
<h1>Dog Panel v3</h1>
```

- Then switch the templateUrl in component over to the new template

dog-panel \ app.component.ts

```
    selector: 'app-root',
    styleUrls: [app.component.css']
    templateUrl: app.component3.html',
```

- The component file needs the extra "template" code that it will show. Add it (buyTemplate) and the other property to delay it (delayedMessage) which will be used in the template like this:

dog-panel \ app.component.ts

```
    mockDogs: Dog[] = MockDogs.SIX;
    buyTemplate: string = `<div class="ad"><h1>Buy now!</h1>
        <p>Vet Pro is the only way to go!</p></div>`;
    delayedMessage: Promise<string> = new Promise((resolve, reject) => {
        setTimeout(() => resolve(this.buyTemplate), 4000);
    });
```

- Change the timeout to any value in milliseconds you want. And you can change what appears in the "template" of the buyTemplate.
- Place the CSS code needed to obscure the dog panels, in the site CSS file **css/styles.css**. (I know, it should be in the dog panel's styleUrls since it's just about that dog panel but you can try it. It broke in v5.)

css / styles.css

```
    /* dog-panel module */
    .ad {
        padding: 2em;
        display: block;
        top: 0px;
        left: 3em;
        position: absolute;
        background-color: hsla(0, 100%, 90%, .90);
    }
```

- Now you put the modal on the real template in the template file at the bottom or almost wherever you want as long as it's in the repeated div.

dog-panel / app.component3.html

```
         <span *ngIf = "dog.balance > 0">Balance: {{dog.balance  |
currency:'USD':true:'1.2-2' }}</span>
   </p>
       <div [innerHTML] ="delayedMessage | async"></div>
```

  - The alternate syntax for that new code is

dog-panel / app.component3.html

```
   <div innerHTML ='{{delayedMessage | async}}'></div>
```

- Save your files and view in your browser to see the change after 4000ms. Either style of syntax works but see which one you like the best. I prefer the first for more simple syntax.
- We will get rid of this template in the next exercise since this is annoying.

20. Custom pipe

Occasionally you need a JavaScript function available as a way to process text and maybe you'd like to not have that JavaScript on the page. You refactor it out as a pipe so it's easier to read and use. We'll make a useful quoting pipe for curly quotes since the Unicode characters are hard to look up.

- Switch the templateUrl in the component over to the previous template

dog-panel \ app.component.ts

```
      selector: 'app-root',
      styleUrls: ['dog-panel.component.css'],
      templateUrl: 'dog-panel.component2.html'
```

- Then we want to properly quote the dog's name in the panel. Let's add the pipe first:

dog-panel \ app.component2.html

```
      <h3>{{dog.name | curlyQuotes }}</h3>
```

- Of course, that won't do anything except cause a bunch of error messages on top of your page and in your browser console, which you might like to get acquainted with. We'll work backwards to add the pipe connections to our module setup file**.** First, we add the import for the class we'll be writing for the pipe:

dog-panel \ app.module.ts

```
   import { AppComponent }        from './app.component';
   import {CurlyQuotesPipe}   from './curlyquotes.pipe';
```

- Then in the @NgModule, we tell it to bring it into scope with the declarations property:

dog-panel \ app.module.ts

```
   @NgModule({
     imports:       [ BrowserModule ],
     declarations: [ AppComponent, CurlyQuotesPipe ],
```

Using the pipes property in the Component class will be deprecated in the future. Watch for old code doing this and move the declaration to the module file.

- Now we write the pipe. Create a new file called **curlyquotes.pipe.ts** in the **dog-panel** directory. Place this code in there:

dog-panel / curlyquotes.pipe.ts

```
   import { Pipe, PipeTransform } from '@angular/core';

   // USAGE for @NgModule
   // import {CurlyQuotesPipe}       from './curlyquotes.pipe';
   // declarations: [CurlyQuotesPipe]
   // USAGE for template
   // {{text | curlyquotes}}
```

```
@Pipe({name: 'curlyQuotes'})
export class CurlyQuotesPipe implements PipeTransform {
  transform(value: string, args?: any[]) {
      return '\u201C' + value.trim() + '\u201D';
  }
}
```

- Save your files and then refresh the browser to see the pipe at work. If you don't find that it works, you may have to restart the start task (Terminal / Start).

Optional Unicode stuff

- If you'd like other dog symbols to play with in your pipe
  - \u1f415 is a sideways dog
  - \u1f436 is a dog face
- But these are higher than what can be escaped in ES5. You have to turn them into their separate parts like this.
  - "\uD83D\uDC15"
  - "\uD83D\uDC36"
- ES6 has Unicode code point escapes would make these characters look like
  - '\u{1f415}'
  - '\u{1f436}'
- And TypeScript recognizes ES6 code points so it's better to use these since online resources, like my favorite AmpWhat (http://www.amp-what.com/unicode/search/dog), shows you the code point syntax.

Child components

21. Content projection

A useful trick is to dump all sorts of semantic elements out into an app and have it sort it out for you, such as articles, nav, asides, header and footers. We'll build a couple of children together into a template to demonstrate the power of how Angular moved away from Angular 1's transclusion into content projection.

Copy and update basic template

- Copy the contents of your template directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **children**.
- First switch the import in the main config file so that the app now will load our new root module.

main.ts

```
import { AppModule } from './children/app.module';
```

- Then update the template

children / app.component.html

```
<h1>Children</h1>
```

- View the page in your browser. You should see a Children headline. If you don't, restart the start task.

New children

- Set up the module references in **children / app.module.ts**

children / app.module.ts

```
import { Children, OrganizingChild, IgnoringChild }  from
'./app.component';

@NgModule({
  imports:       [ BrowserModule],
  declarations: [ AppComponent, OrganizingChild, IgnoringChild],
  bootstrap:     [ AppComponent]
```

- Building the files now will cause an error because of the absence of the two new imported classes that do not exist.
- The code in **children/app.component.ts** can include all the children components so that the file looks like this at the end:

children / app.component.ts

```
import { Component } from '@angular/core';
//--------------------------------------------- optional child component
```

```
@Component({
   selector: 'ignored-stuff',
   templateUrl:'ignoring-child.html'
})
export class IgnoringChild { }
//------------------------------------------ optional child component
@Component({
    selector: 'unorganized-stuff',
    templateUrl:'organizing-child.html'
})
export class OrganizingChild { }
//---------------------------------------------- root (app) component
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css']
})
export class Children {
 }
```

- You will notice that you need two other template files for this exercise, the two child component templates. Create both files
  - **ignoring-child.html** and
  - **organizing-child.html**

  in the correct directory and leave them empty for now.
- Building and viewing the project will show the same result.

- Add code to **children/children.component.html** to look like this:

children / app.component.html

```
<h1>Children</h1>
<p>Before organizing child component</p>
<unorganized-stuff>
</unorganized-stuff>
<p>After organizing child component</p>

<p>Before ignoring child component</p>
<ignored-stuff>
</ignored-stuff>
<p>After ignoring child component</p>
```

- Save all the files and view the page in your browser. It should not have much on it except this text under the heading
  - Before organizing child component
  - After organizing child component
  - Before ignoring child component
  - After ignoring child component

- The rest of the page

Organizing the children

- The easiest child to take care of first is the ignoring-child. Update the template of **ignoring-child.html** with this code.

children / ignoring-child.html

```
<div class='callout secondary'>
    <p>I'm ignoring all of your stuff.</p>
</div>
```

- When the server reloads the page, you'll see at least one more line in a shaded box.
- Now we want to put together a structure of a component that orders the marked-up information in the parent template. In the **organizing-child.html**, place this code:

children / organizing-child.html

```
<div class='callout primary'>
  <p>All details element selector</p>
         <ng-content select="details"></ng-content>
  <p>All class togetherness selector</p>
        <ng-content select=".togetherness"></ng-content>
  <p>All attribute a selector</p>
        <ng-content select="[a]"></ng-content>
  <p>All element+attribute aside of='any' selector</p>
         <ng-content select="aside[of]"></ng-content>
  <p>All the unselected body content</p>
         <ng-content></ng-content>
</div>
```

- Review the selectors that are being used in this template. Make sure you know your CSS selector syntax.
- Now when the browser refreshes, you shouldn't see much since you haven't fed this template any information from the parent template.

Feeding the children

- Now we start to add unstructured information into the parent and it will be selected into the child template.
- There is quite a bit of information to put in here, so you see it working. Try adding a few lines at a time refreshing each time.
- Make sure you use complete elements. Angular doesn't like it when you don't close your open tags.
- You can use your own random pieces of code in the ignored-stuff element. Here's the final template for **app.component.html**:

```
<h1>Children</h1>
<p>Before organizing child component</p>
```

```
    <unorganized-stuff>
        <div>I am the first of all of the unwanted.</div>
        <details>This is detailed stuff 1. </details>
        <div a>1. aaaaaaaaaaaaaattribute.</div>
        <details>This is detailed stuff 2. </details>
        <div>We have leftovers.</div>
        <div class='togetherness'>A class likes to be together.</div>
        <div>Somebody lost this div. Is it yours?</div>
        <details>This is detailed stuff 3. </details>
        Where do you think this goes?
        <div a>2. Another aaaaaaaaaaaaaattribute.</div>
        <div class='togetherness'>We like each other.</div>
        <aside of='fries'>Crunchy hot good fries.</aside>
        <aside of='bacon'>Bacon goes with anything.</aside>
        At the end.
        <aside of='hash browns'>Wait, more potato crunchiness.</aside>
        <p>Unorganized paragraphs. Styled or unstyled? You decide.</p>
    </unorganized-stuff>
    <p>After organizing child component</p>
    <p>Before ignoring child component</p>
    <ignored-stuff>
        <div>bla bla bla</div>
        <img src='https://s-media-cache-
ak0.pinimg.com/236x/cc/cc/95/cccc95d90c482d7ae3618d492f625216.jpg' />
        <div>Like you're really gonna read this.</div>
    </ignored-stuff>
    <p>After ignoring child component</p>
```

- You should see all the lines reordered by the child template.
- You shouldn't see any of your ignored code. See if you can add an element in your ignoring-child.html file that will pull all of the inner HTML into that body just to make sure you can. Yes, this is a test.

Styling the children

- You can add style to the template you are working with a styles property. Let's add style to just the unorganized stuff. In **children.component.ts** update that section like this:

children / app.component.ts

```
    templateUrl:'organizing-child.html',
        styles: ['p {color: cadetblue; font-weight: bold; font-size: 1.3rem}
']
```

- When you view the file, you will have some but not all paragraphs styled with that. Only the viewable paragraphs in the template will be styled. The included content will not inherit that style because of view encapsulation. There's one paragraph that is not styled (Styled or unstyled? You decide.)

- We need to let the style be accessible to the child template. Make this change to the component to let the view style not be so isolated.

children / app.component.ts

```
      styles: ['p {color: cadetblue; font-weight: bold; font-size: 1.3em}
'],
       encapsulation: ViewEncapsulation.None
```

- And then add the import above from the Core module:

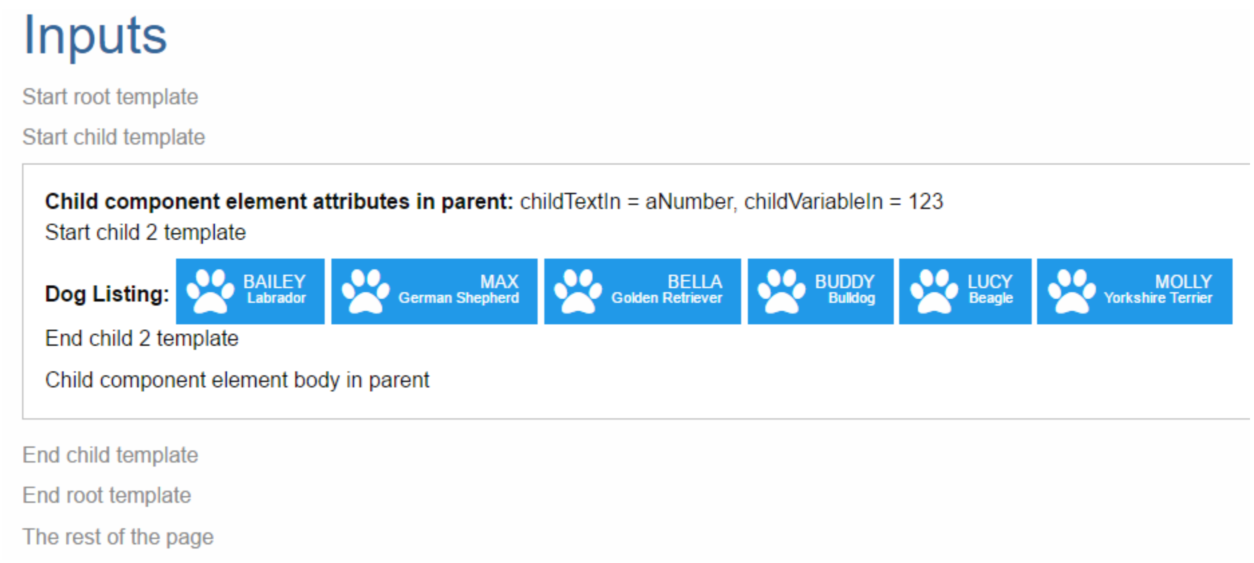
children / app.component.ts

```
   import { Component, ViewEncapsulation } from '@angular/core';
```

- Now save and view in the browser and notice the changes. Our style is now not isolated and applies to our included content.

22. Data input from parents

- This exercise will show how to get data through to multiple nested child components in various ways in this exercise. The final product will eventually look something like:



Inputs

Start root template

Start child template

Child component element attributes in parent: childTextIn = aNumber, childVariableIn = 123

Start child 2 template

Dog Listing: BAILEY Labrador | MAX German Shepherd | BELLA Golden Retriever | BUDDY Bulldog | LUCY Beagle | MOLLY Yorkshire Terrier

End child 2 template

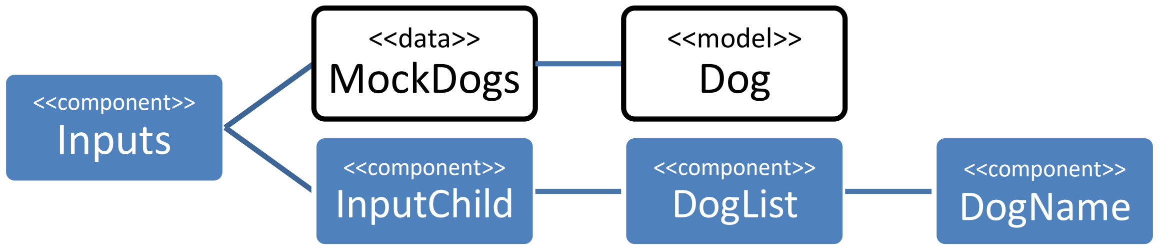
Child component element body in parent

End child template

End root template

The rest of the page

- You can see the nested child templates and there’s one more that makes the labels for the dogs that we pass data from the root component for. The component relationships will look like



Copy and update basic template

- Copy the contents of your template directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **inputs**.
- Then update the references in the template

inputs / app.component.html

```
<h1>Inputs</h1>
```

- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './inputs/app.module';
```

- And then copy the **model** directory from the dog-panel exercise into the Inputs directory so the path matches. You are bounded by the directory for the component the way this is configured for now.
- Save the files, restart the start task if necessary, and view the page in your browser. You should see an Inputs headline.

Prepare the parent to send data to a child

- The code in our component file will declare some variables:

inputs / app.component.ts

```
    export class AppComponent{
        aNumber: number = 123;
        dogs = MockDogs.SIX;
    }
```

- This will output the initialization logs from the model classes to the console if you build and view it once the imports for Dog and MockDogs are in place (see below).
- We'll update our template page with a child component selector that will allow different types of data from its parent:

inputs / app.component.html

```
    <h1>Inputs</h1>
    <p>Start root template</p>
        <input-child childTextIn='aNumber' [childVariableIn]='aNumber'
[dogsToChild]='dogs'>
            <div>Child component element body in parent</div>
        </input-child>
    <p>End root template</p>
```

- That will create some errors for now. Then we'll add that child component in the **app.component.ts** class so it's convenient. This will declare that it will allow data and rename one with an alias.

inputs / app.component.ts

```
    import { Component, Input } from '@angular/core';
    import {Dog}             from './model/dog';
    import {MockDogs}        from './model/dog-data';

    //------------------------------------------  optional child component
    @Component({
        selector: 'input-child',
        templateUrl: 'inputchild.component.html'
    })
    export class InputChild {
        @Input('dogsToChild') dogsFromParent: Dog[];
    @Input( ) childVariableIn: string;
    @Input( ) childTextIn: string;
}
//-------------------------------------------  root component
@Component({
```

- The child template of **inputchild.component.html** will display the passed in data by referencing the names of the variables but passing the dogs to its child. Create that file with the content below.

inputs / inputchild.component.html

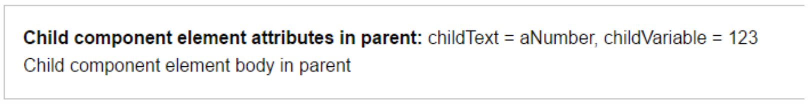
```

<h6>Start child template</h6>
<div class="callout">
    <b>Child component element attributes in parent:</b>
    childTextIn = {{childTextIn}},
    childVariableIn = {{childVariableIn}}
    <!-- <dog-list [dogsToChildAgain]='dogsFromParent'></dog-list> -->
    <ng-content></ng-content>
</div>
<h6>End child template</h6>
```

- The commented-out line will be used in the next step to loop over the dogs and display them. If you copy the code from the repo, it will not be commented out causing error messages.
- The last step is to tell the parent component module that it should include the child component in the module also. In the app.module.ts file you'll update two lines:

inputs / app.module.ts

```
import { AppComponent, InputChild }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [AppComponent, InputChild],
```

- Now when you build and view the file you'll see the child component template embedded in the parent with the child looking like the image here.


Child component element attributes in parent: childText = aNumber, childVariable = 123
Child component element body in parent

Prepare a child to get and receive data

- The dogs that are coming in from the parent will be passed to the child of this child. We'll start with the template and uncomment the line that uses the dog-list element.

inputs / inputchild.component.html

```
childVariableIn = {{childVariableIn}}
    <dog-list [dogsToChildAgain]='dogsFromParent'></dog-list>
    <ng-content></ng-content>
```

- Then we'll add that child component in the **app.component.ts** class again so it's convenient. This will declare that it will allow data but rename it with an alias so you understand where it's going. You may decide to call all the variables the same and not use an alias. It would be easier and less buggy I'm sure.

inputs / app.component.ts

```
//------------------------------------------  optional child component
@Component({
    selector: 'dog-list',
    templateUrl: 'doglist.component.html'
})
export class DogList {
     @Input('dogsToChildAgain') dogsFromParent: Dog[];
}
//------------------------------------------  optional child component
```

- The new template that this component references, **doglist.component.html** does the loop for us and looks like

inputs / doglist.component.html

```
<h6>Start child 2 template</h6>
<div>
    <b>Dog Listing:</b>
    <span *ngFor="let aDog of dogsFromParent">
    <span class='label'><i class='fas fa-paw fa-3x float-left'></i>
    <!-- <dog-name [dogNameIn]='aDog'></dog-name> -->
    </span>
    </span>
</div>
<h6>End child 2 template</h6>
```

- The last step is to tell the parent component module that it should include this next child component in the module also. In the **app.module.ts** file you'll update two lines:

inputs / app.module.ts

```
import { AppComponent, InputChild, DogList }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [AppComponent, InputChild, DogList],
```

- Now when you build (restart the start task) and view, you should see six paw print icons, as there are six dogs in the MockDogs data.

Prepare the final child to show the name

- The dogs that are coming in from the parent will again be passed to the child of this child. We'll start with the template and uncomment the line that uses the dog-name element.

inputs / doglist.component.html

```
        <span class='label'><i class='fa fa-paw fa-3x float-left'></i>
        <dog-name [dogNameIn]='aDog'></dog-name>
        </span>
```

- Then we'll add that child component in the **inputs.component.ts** class again so it's convenient. The template and the styles are short enough and likely not to require updating so we'll include them in the TypeScript file. You might decide to externalize them for easier editing and no recompiles. But then, it's easier to see all the code in one place.

inputs / app.component.ts

```
    //----------------------------------------  optional child component
    @Component({
        selector: 'dog-name',
        styles: [
            '.dog-nameContainer {margin:.4rem; display: inline-block; text-
align: right; }',
            '.dog-breed {font-size: .7rem; }'
        ],
        template: `
        <div class='dog-nameContainer'>
            <div class='dog-name'>{{dogNameIn.name | uppercase }}</div>
            <div class='dog-breed'>{{dogNameIn.breed}}</div>
        </div>
    `
    })
    export class DogName {
        @Input( ) dogNameIn: Dog;
    }
    //----------------------------------------  optional child component
```

- The last step is to tell the parent component module that it should include this next child component in the module also. In the **app.module.ts** file you'll update the same two lines:

inputs / app.module.ts

```
    import { Inputs, InputChild, DogList, DogName}  from
'./inputs.component';

    @NgModule({
      imports:      [ BrowserModule ],
      declarations: [Inputs, InputChild, DogList, DogName],
```

- Now you should be able to see the final version of the file after you save.

- If you have a problem accessing a property in the Dog class, save the Dog class again to rebuild it.

23. Event input from parent

We'll set up a simple parent-child component relationship and have the child change a color with its own button first. Then we'll move the button to the parent and have it talk to the child to change the color of all the children.

Copy and update basic template

- Copy the contents of your template directory in VS Code and paste to create copies of all the files. Rename the new **template copy** directory to **parent-child**.
- Then update the references in the new files:

parent-child / app.component.html

```
    <h1>A parent to child example</h1>
```

- Update the bootstrap module:

/ main.ts

```
import { AppModule } from './parent-child/app.module';
```

- View the page in your browser. You should see **A parent to child example** headline.

Add a data class and a child component

- We will add a class to supply data for valid styles and sizes of callouts in the Zurb Foundation library so we don't have to think about it. It will go first ibut any order of the classes is fine including externalizing them in other files.

parent-child\app.component.ts

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

    //--------------------------------------  reference data class
    export class Zurb {
        static validStylesForCallouts = ['', 'primary', 'secondary',
'success', 'warning', 'alert'];
        static validSizesForCallouts  = ['', 'small', 'large'];
    }
```

- Next we'll add the child component class called ChildComponent. **Note:** this class must be first before the parent-child component. A forward reference can be used if you require a different loading order.

parent-child\parent-child.component.ts

```
    static validSizesForCallouts  = ['', 'small', 'large'];
    }

    //--------------------------------------  child component
```

```
@Component({
    selector: 'child',
    templateUrl : `child.component.html`
})
export class ChildComponent {
    private _styleOfCallout: string;
    private _sizeOfCallout: string;
    constructor() {
        this._styleOfCallout = '';
        this._sizeOfCallout  = '';
    }
    get styleOfCallout(): string{
        return this._styleOfCallout;
    }
    set styleOfCallout(value: string){
        if (Zurb.validStylesForCallouts.indexOf(value) >= 0) {
            this._styleOfCallout = value;
        }
    }
    get sizeOfCallout(): string{
        return this._sizeOfCallout;
    }
    set sizeOfCallout(value: string){
        if (Zurb.validSizesForCallouts.indexOf(value) >= 0) {
            this._sizeOfCallout = value;
        }
    }
    public changeStyle() : void {
        this.styleOfCallout =
        Zurb.validStylesForCallouts[Math.floor(Math.random() *
        Zurb.validStylesForCallouts.length)];
    }
    public changeSize() : void {
        this.sizeOfCallout =
        Zurb.validSizesForCallouts[Math.floor(Math.random() *
        Zurb.validSizesForCallouts.length)];
    }
}

@Component({
```

- The template for the child class looks like this

parent-child\child.component.html

```
<button class='button' (click)='changeStyle()'>Change style</button>
<button class='button' (click)='changeSize()'>Change size</button>
<div class="callout {{styleOfCallout}} {{sizeOfCallout}}">Callout
    {{styleOfCallout}} {{sizeOfCallout}}</div>
```

- Then we associate the parent with the child by telling the parent about the child in the module file

parent-child / app.module.ts:

```
import { AppComponent, ChildComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule  ],
  declarations: [ AppComponent, ChildComponent ],
```

- And finally, put the child where we want it in the parent template

parent-child / app.component.html

```
<h1>A parent to child example</h1>
<child></child>
```

- Now try out the buttons in the browser.

Relocate buttons and delegate to child

- Remove the buttons from the child template, **parent-child / child.component.html,** and put them in the parent app template, so that, that template looks like:

parent-child / app.component.html

```
    <button class='button' (click)='changeStyle()'>Change style</button>
    <button class='button' (click)='changeSize()'>Change size</button>
    <child></child>
```

- The buttons will not show any change in position. But the events don't have anywhere to go you will get a modal covering the screen from Angular saying it doesn't know what function your click should be talking to.
- We'll be using the @ViewChild decorator so you'll need an import for that at the top of your component file**.**

parent-child / app.component.ts

```
import {Component, ViewChild}       from '@angular/core';
```

- Then we'll make a reference to the child component in the parent component so we can talk to it and make a few delegate methods that our buttons need to use to send an event to.

parent-child / app.component.ts

```
export class AppComponent {
    @ViewChild(ChildComponent)

    private childCallout: ChildComponent;

    changeStyle(): void { this.childCallout.changeStyle(); }
    changeSize(): void { this.childCallout.changeSize();}
```

- Test the page by clicking on the buttons again and make sure the events are flowing from the parent to the child via our delegation.

Make more children

- Add several more children to your parent template in the template file.

parent-child / app.component.html

```
        <button class='button' (click)='changeSize()'>Change size</button>
        <child></child>
        <child></child>
        <child></child>
        <child></child>
```

- Refresh your browser and you'll see that your clicks only go to the first child callout.
- Add another few imports to the top of your component file

parent-child / app.component.ts

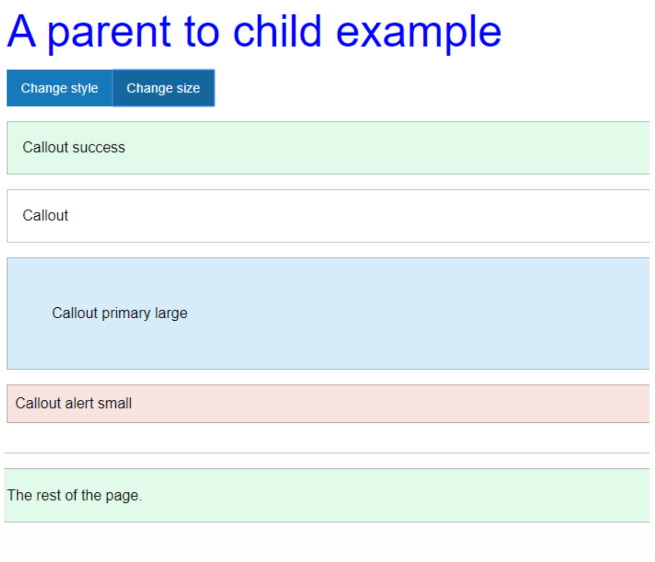
```
    import {Component, ViewChild, ViewChildren, QueryList} from
'@angular/core';
```

- Switch your ViewChild over to a collection of child components with ViewChildren. There are several changes to make and one debugging line for your console so you can see what's in the ViewChildren object.

parent-child / app.component.ts

```
    export class AppComponent {

        @ViewChildren(ChildComponent)
        private viewChildren: QueryList<ChildComponent>;
    ~~    private childCallout: ChildComponent;~~

        public changeStyle(): void {
            console.info('viewChildren object', this.viewChildren);
            let calloutChildren = this.viewChildren;
            calloutChildren.forEach(callout => {callout.changeStyle();});
        }
        public changeSize(): void {
            let calloutChildren = this.viewChildren;
            calloutChildren.forEach(callout => {callout.changeSize();});
        }
    }
```

- Now refresh the browser and you should be able to see all the components randomly pick a style or a size.



A parent to child example

Callout success

Callout

Callout primary large

Callout alert small

The rest of the page.