Course 2324

Building Web Applications With Angular

by

Mike Way

LEARNING TREE INTERNATIONAL

Copyright

© LEARNING TREE INTERNATIONAL, INC.
All rights reserved.

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written permission of the publisher.

Copying software used in this course is prohibited without the express permission of Learning Tree International, Inc. Making unauthorized copies of such software violates federal copyright law, which includes both civil and criminal penalties.

Introduction and Overview

LEARNING TREE™
INTERNATIONAL

Course Objectives

- Create well-structured Angular applications
- Develop Components using TypeScript, Templates, and Decorators
- Modularize applications with Feature Modules and the Component Router
- Add interactivity with event handlers and bindings
- Capture and validate input with template-driven and reactive forms
- Consume REST services using Observables

REST = Representational State Transfer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Contents

	Introduction and Overview
Chapter 1	Introducing Angular
Chapter 2	Creating Components
Chapter 3	Dependency Injection
Chapter 4	Testing Angular
Chapter 5	Component Interaction
Chapter 6	Routes, Modules, and Pipes
Chapter 7	Forms
Chapter 8	Managing Asynchronous Behavior
Chapter 9	Extending Angular With Directives and Animation
Chapter 10	Publishing your Application
Chapter 11	Course Summary
	Next Steps
Appendix A	Migrating From Angular JS 1

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Structure

- **General**
  - Course materials presented in an interactive lecture format
  - Course designed in a step-by-step, logical approach
- **Do Now exercises**
  - Short exercises in Course Notes

  - Immediately practice a technique or process discussed during the lecture
- **Hands-On Exercises**
  - Practice key concepts or techniques learned
- **Hints**
  - Are provided within the Exercise Manual
  - Are collapsed in the AdaptaLearn version
- **The following fonts have special meaning in the Course Notes**
  - Text in *Arial italic* introduces an important concept or term
  - `Consolas` is used to indicate JavaScript/TypeScript language keywords or examples

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

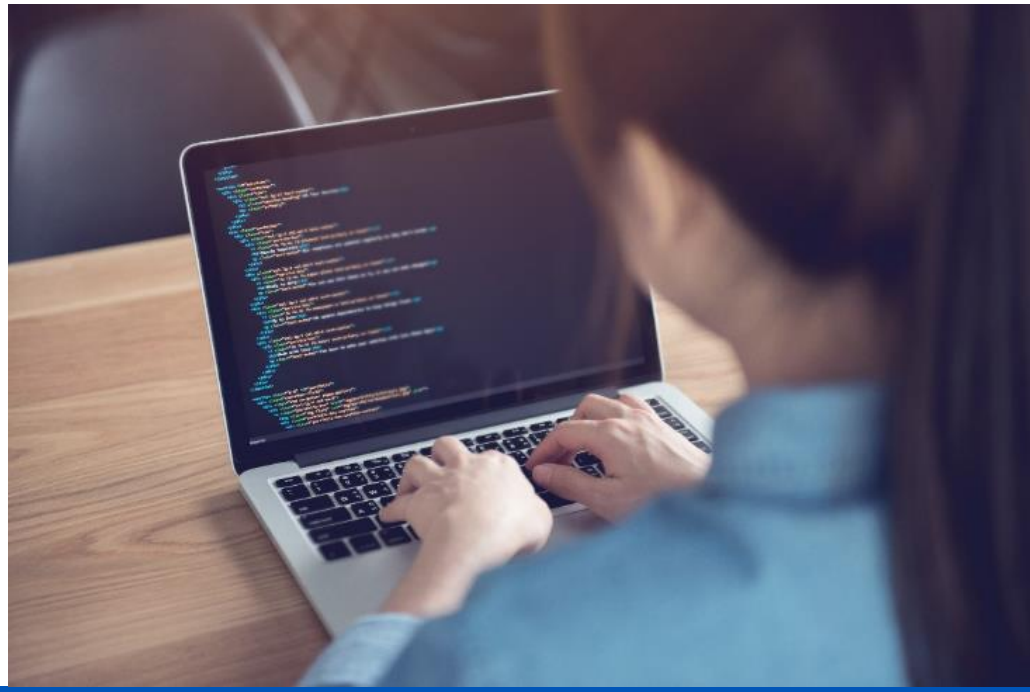
Exercise Structure

- **Exercises are intended to challenge you**
  - Early exercises require you to create all of the code
  - Later exercises provide a framework
    - Within which you will complete the most significant parts of the code
- **Exercise solutions**
  - Provided for all exercises
  - Located in `C:\course2324\Solutions`
    - Feel free to peek if you get stuck
- **All exercises, solutions, and course examples are available online**
  - Can be downloaded from your My Learning Tree account

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Expected Background

- **Experience with JavaScript or TypeScript is required**
  - To the level of Learning Tree Course 2319—JavaScript Essentials for the Full Stack Developer Using ECMAScript
- **Knowledge of HTML and CSS is beneficial**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Our Classroom Environment

- **Each machine runs**
  - Windows 10
  - WebStorm
  - Visual Studio Code
  - Notepad++
  - Chrome
  - Mozilla Firefox
  - Apache Tomcat
  - Node JS
  - Angular-cli

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

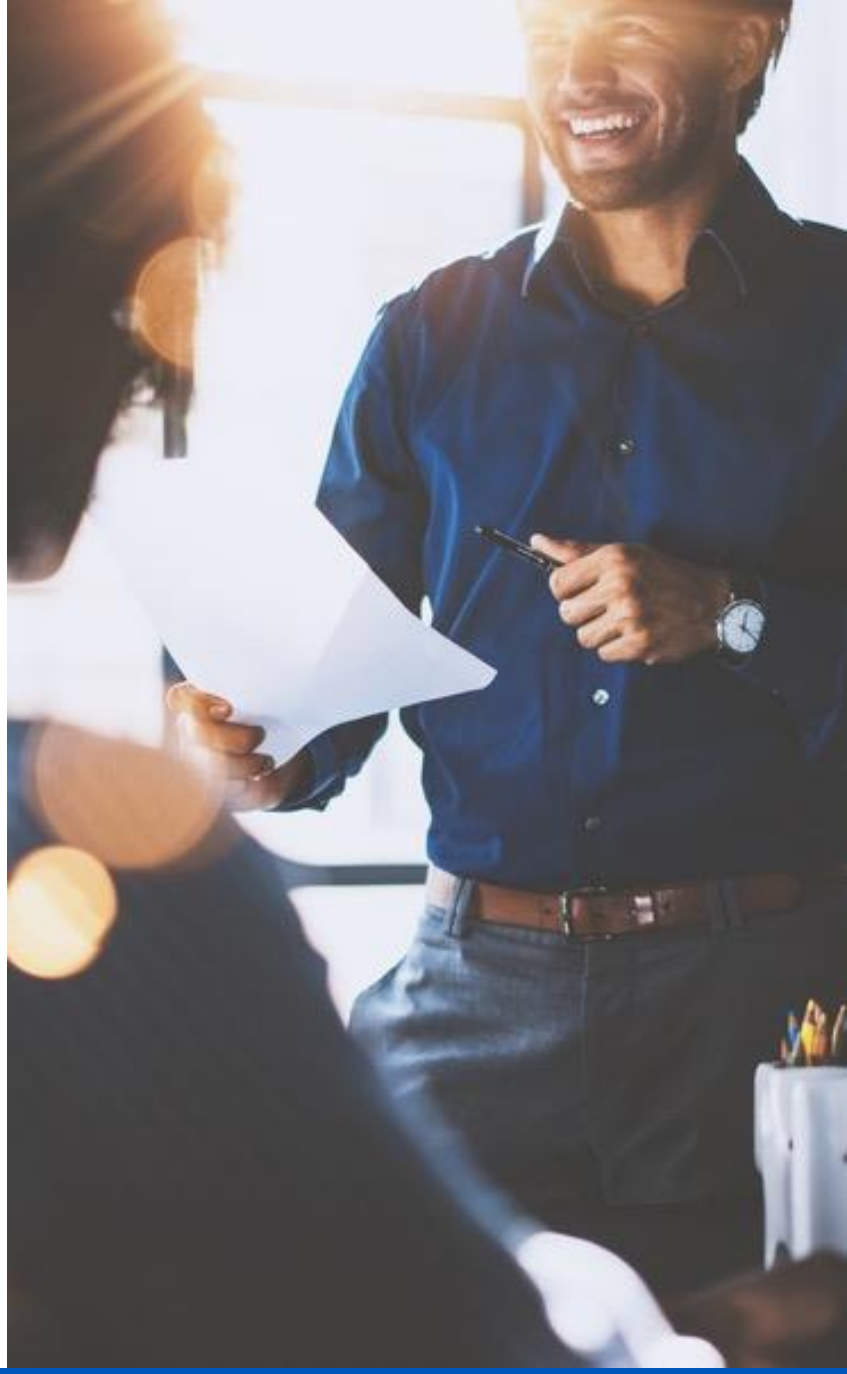
Chapter 1

Introducing Angular

LEARNING TREE™ INTERNATIONAL

Objectives

- Learn how Angular eases the development of modern GUI applications
- Learn how to configure an Angular environment
- Learn how to navigate the architecture of Angular
- Learn essential TypeScript programming skills



GUI = graphical user interface

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Benefits of Angular

- Architectural Overview
- Configuring an Environment
- Components of an Angular Application
- TypeScript Primer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

In the Beginning

- **Early web applications were simple**
  - Server delivered an HTML page to the browser
    - Containing a form if input was required
  - Browser displays the page
    - User submits the form
  - Server sends another page for the browser
- **Simple to develop**
- **The user interface experience was poor**

HTML = hypertext markup language

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Request Processing: Traditional vs. Ajax

- **Traditional web browser—web server interaction:**

- **Ajax interaction:**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Modern Web Applications

- **Modern web applications provide an excellent user experience**
- **Combine three key technologies sometimes known as Ajax**
  - JavaScript
    - The `XMLHttpRequest` (XHR) object
  - REST web services
  - CSS
- **Provide an excellent user experience**
- **Results in larger applications**
  - More difficult to develop, test, and maintain

Ajax = Asynchronous JavaScript and XML
CSS = Cascading Style Sheets

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular—Yet Another Framework?

- **Cross-platform development framework for**
  - Modern web applications
  - Native mobile applications
- **Component-based declarative programming model**
  - Developer defines appearance and behavior
  - Framework maps components onto the HTML DOM
    - Decouples user interface from direct DOM manipulation
  - Simplifies development and testing
- **Dependency injection**
  - Developer specifies component relationships
  - Angular wires them together
- **Easy integration with REST APIs**
- **Modular design**
  - Reduced load times
  - Expandable

API= application programming interface
DOM = document object model

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular—What's in a Name?

- **Angular**
  - Originally known as Angular 2
  - Is a redevelopment of Angular JS
    - AKA Angular 1
  - Same basic objectives
  - Radically different engineering solution
- **Migration is supported**
  - Options
    - Migrate entire Angular JS application to Angular
    - Use Angular Components in an Angular JS application
    - Use Angular JS directives in an Angular application

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular Versions

- **Initial version of what is now called Angular was 2.0.0**
- **Angular uses Semantic Versioning**
  - Changes to the last digit indicate a bug fix
  - Changes to the middle digit indicate a new feature
  - Changes to the first digit indicate a major change
    - Which may not be backward compatible

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

1.1: Investigating the Case Study Application

Do Now

- **From the Desktop, run the Course Command Prompt**
  - On Windows, Angular CLI runs much more quickly as administrator
- **At the command prompt, type `cd FlySharpSolution` and press `<Enter>`**
- **Type `ng serve`**
- **Wait until you see the message "`webpack: Compiled successfully`"**
- **Start Chrome and open the URL `http://localhost:4200/`**
- **You should see the case study application running**
- **Click each of the tabs in the application**
- **Click the "Buy" button on the Buy Flights page**

URL = uniform resource locator

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

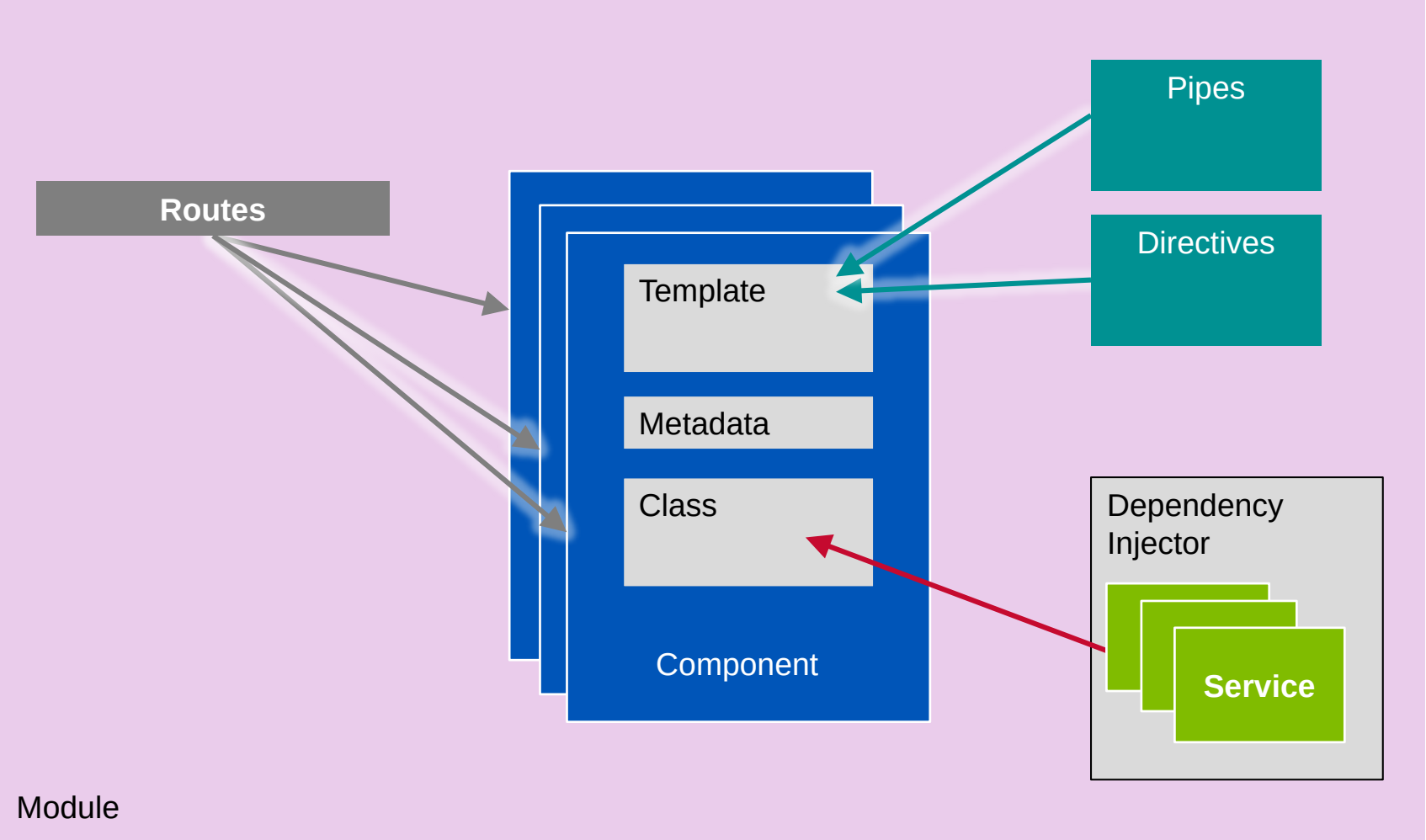
- Benefits of Angular

Architectural Overview

- Configuring an Environment
- Components of an Angular Application
- TypeScript Primer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

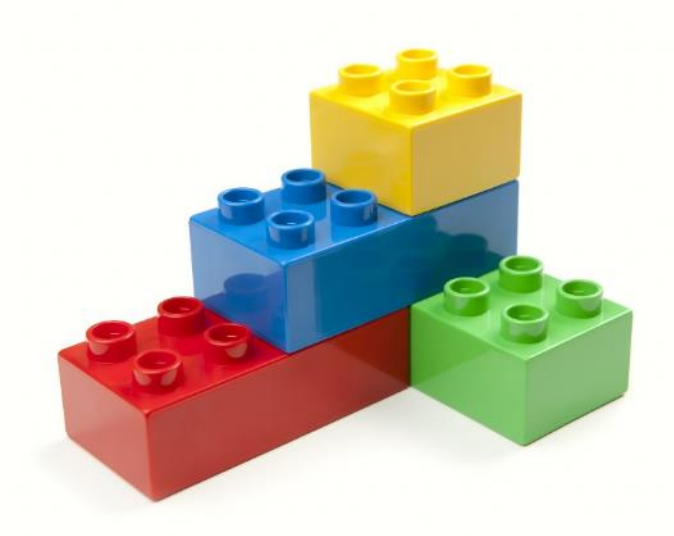
Architecture Diagram



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular Building Blocks

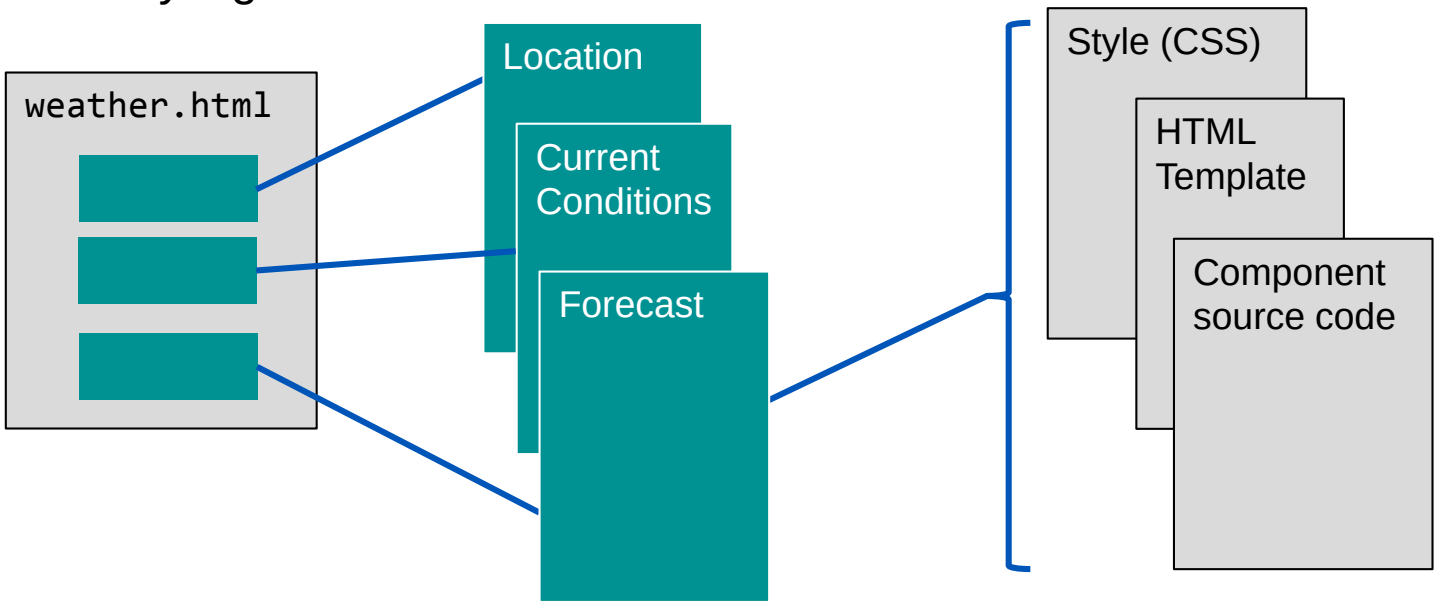
- **Component**
  - Primary building blocks of visual interface
  - A template for the view with supporting code
- **Directive**
  - Allows Angular code to manipulate the DOM
- **Pipe**
  - Data transformation
- **Service**
  - Encapsulation of application logic
  - A mechanism to share functionality across multiple Components
- **Modules**
  - A collection of the previous items
- **Route**
  - A URL path/Component ID combination
    - Used to control navigation/flow through an application



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component-Based Architecture

- **User interfaces are built from one or more Components**
- **Each Component consists of**
  - Code—written or *transpiled* into JavaScript
  - HTML—visual structure of the Component
  - CSS—styling



***Transpiling.** *The process of converting the source code from one programming language into the source code for another*

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

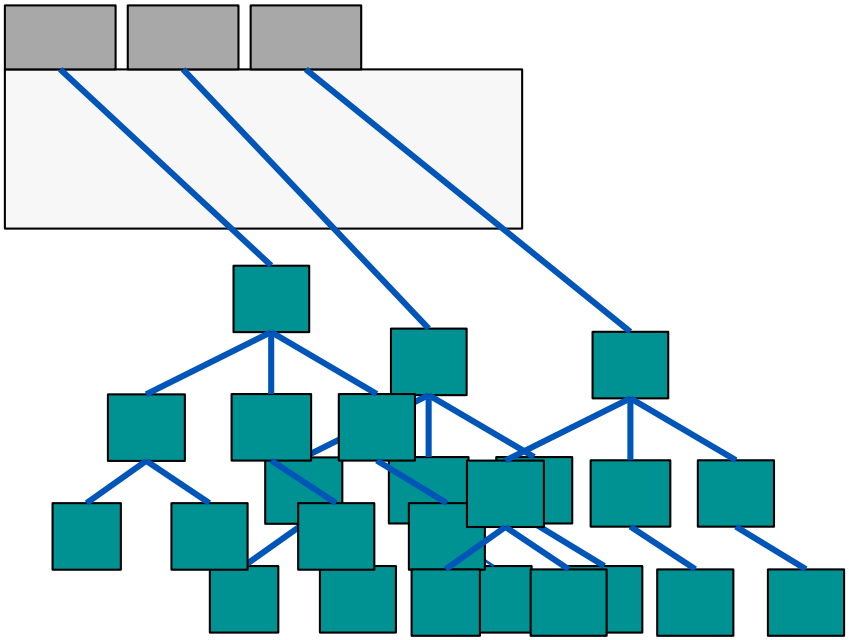
Data-Bound Components

- **The visual representation of a Component is defined by its template**
- **Templates look like regular HTML with additional Angular syntax**
  - Angular expressions and directives
- **Expressions define data bindings**
  - Bindings populate the template from the model at runtime
- **Data binding is also used with forms**
  - To populate the model with user data

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Routing

- **A mechanism to define user paths (routes) through the application**
  - Supports back-button navigation
- **Creates logical divisions within an application**
  - Reduces load time and application complexity
  - Breaks the application into multiple component hierarchies
    - Loaded on demand
- **For example, a tabbed application**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Programming Language for Angular

- **Angular code is written in TypeScript**
- **But browsers do not execute TypeScript!**
  - They execute JavaScript
- **JavaScript is more accurately known as ECMAScript**
  - Versions
    - ES5—released 2009
    - ES6, aka ECMAScript 2015—released 2015
- **Angular 8 generates code for both ES5 and ES6+**
  - ES5 for legacy browsers
  - ES6+ offers native support for many TypeScript features
    - Smaller, more efficient code
- **TypeScript primer coming soon**

Remember: Transpiling is the process of converting the source code from one programming language into the source code for another

---

ECMAScript: European Computer Manufactures Association Script

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Benefits of Angular
- Architectural Overview

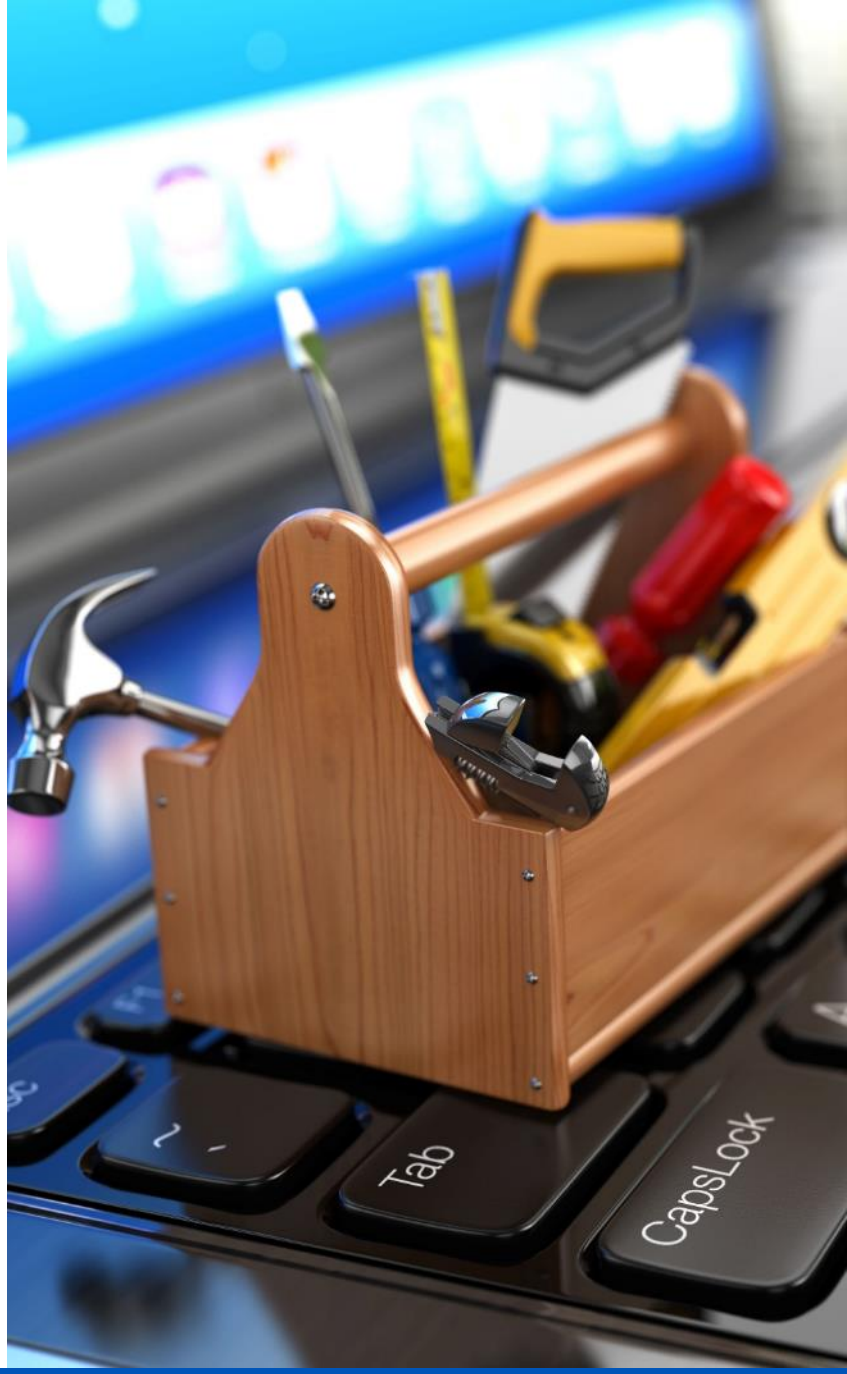
Configuring an Environment

- Components of an Angular Application
- TypeScript Primer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular Development Environment

- **Tools to build an Angular application**
- **An editor**
  - WebStorm, Visual Studio Code, Eclipse
- **Angular CLI**
  - Angular command line interface
  - Supports all development activity except editing
- **Angular CLI is a `Node.js` module**



HTTPD = Apache's Web server
IIS = Internet Information Services

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing `Node.js`

- **`Node.js` is a JavaScript runtime**
  - Based on the Chrome V8 JavaScript engine
  - Lightweight, efficient environment for running JavaScript code
    - Can be used to create and run applications written in JavaScript
    - Outside of a browser
  - Vast number of prebuilt modules available
  - Managed via `npm`
    - The `Node.js` default package manager

- **`Node.js` is required to develop Angular applications**
  - Angular applications should be built and tested using the Angular CLI `Node.js` module
  - Once built, the Angular application can be deployed without requiring `Node.js`
  - Can be deployed into virtually any web server

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular CLI

- **Angular CLI provides a set of utilities to simplify Angular development**
  - Angular CLI is run using the `ng` command
    - Create a new project `ng new`
    - Create a new Component `ng generate`
    - Build and execute `ng serve`
    - Run tests `ng test`
    - Create a production build `ng build --prod`
- **Prerequisites**
  - `Node.js`—version 10+
  - NPM
    - Installed as part of `Node.js`

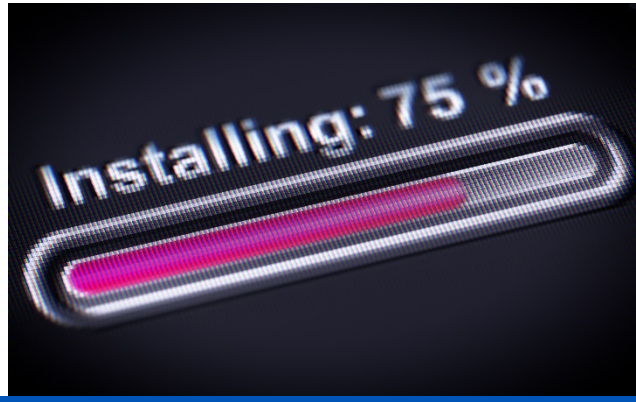
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Installing Angular CLI

- **Install prerequisites**
  1. Download and install `Node.js` from https://nodejs.org/en/
     - Installers for Windows and OS/X
     - RPM and Debian packages available for Linux installations
- **Use `npm` to install Angular CLI**

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

- **Your workstation has already been configured**
- **`@angular/cli` has a huge number of options**
  - Run `ng help` to see them



RPM = RPM Package Manager

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating an Angular Application

- **Creating a new Angular application is easy with the Angular CLI**
  - Type the command:

```
ng new <App_Name>
```

  - This creates a new project
    - Within a new folder named `<App_Name>`
  - And configures the structure of the project with all required artifacts except the Node.js modules
  - There are many additional options
    - You will use `--routing` to add a routing module to the project

- **Creating a project this way is easy and reliable**
  - Installing the `Node.js` modules can be slow!
    - Dependent on your network connection

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

AdaptaLearn™ Enabled

- **Electronic, interactive exercise manual**
- **Offers an enhanced learning experience**
  - Some courses provide folded steps that adapt to your skill level
  - Code is easily copied from the manual
  - After class, the manual can be accessed remotely for continued reference and practice
- **Printed and downloaded copies show all detail levels (hints and answers are unfolded)**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using AdaptaLearn™

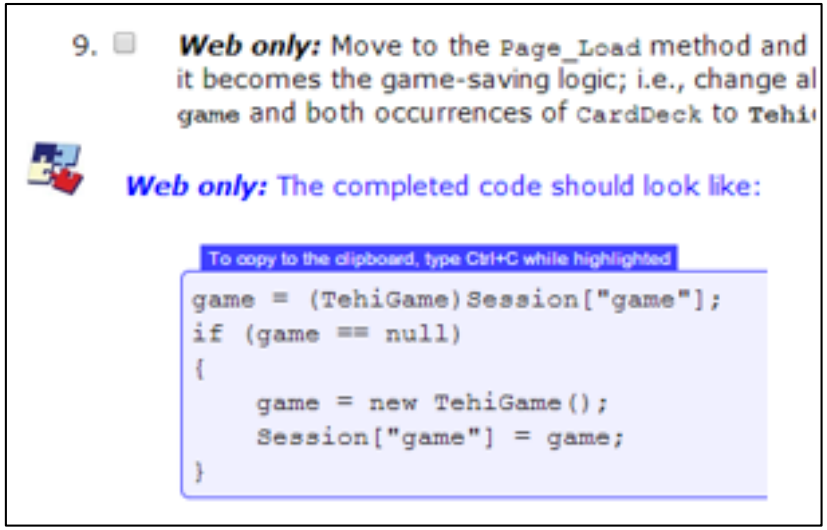
Do Now

1. **Launch AdaptaLearn by double-clicking its icon on the desktop**
   - Move the AdaptaLearn window to the side of your screen or shrink it to leave room for a work area for your development tools

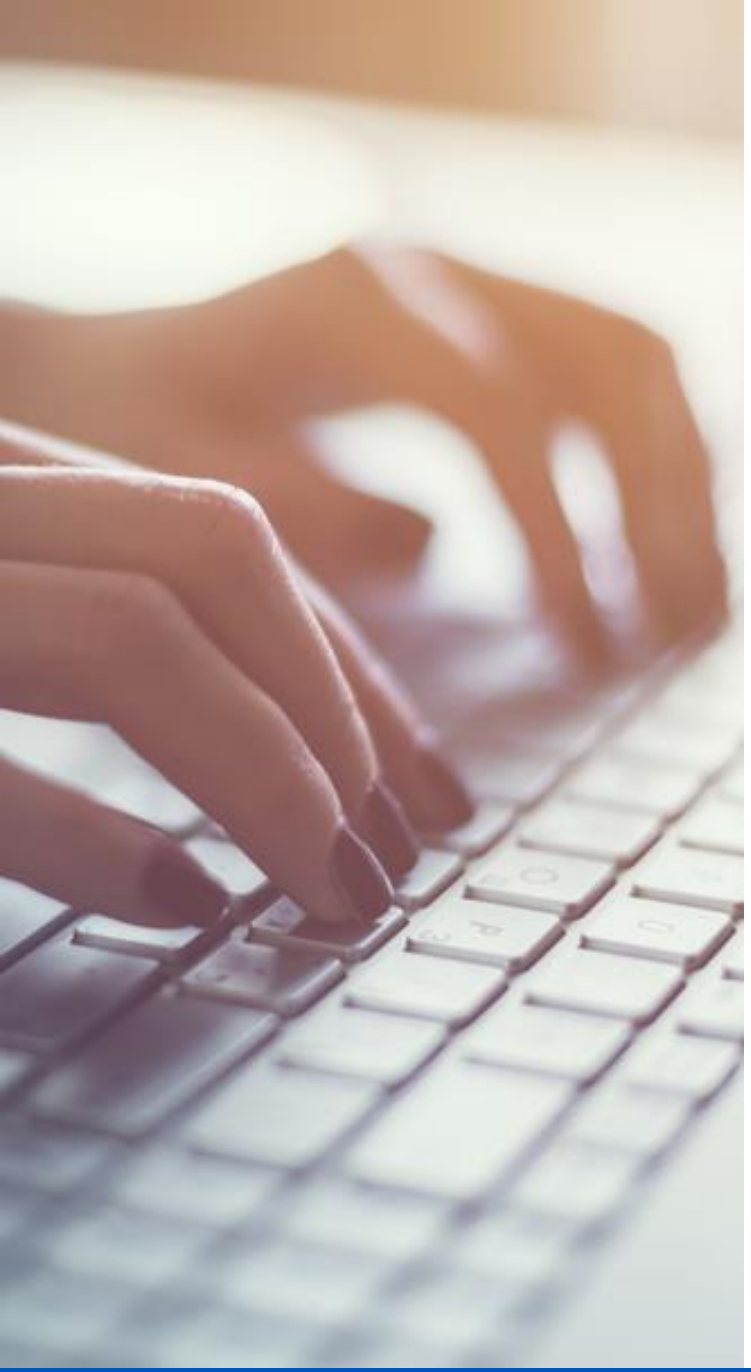
2. **Select an exercise from the exercise menu**
   - Zoom in and out of the AdaptaLearn window
   - Toggle between the AdaptaLearn window and your other windows

3. **Look for a folded area introduced with blue text (not available in all courses)**
   - Click the text to see how folds work

4. **Try to copy and paste text from the manual**
   - Some courses have code boxes that make it easy to copy areas of text while highlighted (as shown)



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



1.2: Creating an Angular Application

Do Now

- In your Exercise Manual, please refer to Do Now 1.2: Creating an Angular Application

IMPORTANT: you must complete this exercise before attempting any others

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Benefits of Angular
- Architectural Overview
- Configuring an Environment

Components of an Angular Application

- TypeScript Primer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Application Component

- **Angular CLI creates an initial Component**
  - In the app directory
- **The Component is named `AppComponent`**
  - In a set of files `app.component.*`
- **Four files created as standard***
  - `.css`—CSS styling (empty placeholder)
  - `.html`—HTML template
  - `.spec.ts`—unit test code (TypeScript)
  - `.ts`—the Component source code and metadata
- **Convention is that all visual Components' names should end with `Component`**

*Our app directory has an additional file `app-routing.module.ts`. This was created as a consequence of using the `--routing` flag.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

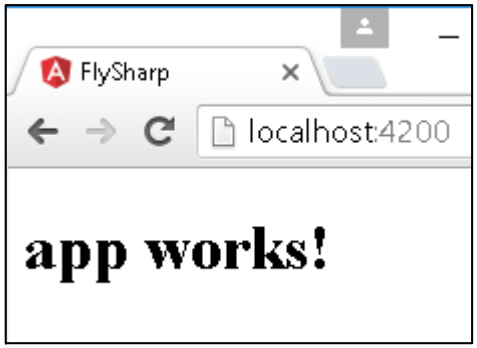
The HTML Template

- **Template file uses HTML 5 plus Angular binding syntax to define visual appearance of Components**

app.component.html

```
<h1>
  {{title}}
</h1>
```

Binding to `title` property



Note: this is a greatly simplified version of the default `app.component.html` page

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Component TypeScript File

- **The code that defines the `Component`**

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css']
})
export class AppComponent {
  title = 'FlySharp';
}
```

Metadata describing the component

Location of template

This is all the executable code!

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Bootstrapping the Component

- **One problem**
  - The Component won't run!
  - It does not actually contain executable code
- **The solution**
  - Bootstrap Angular and load the Component
  - Angular converts the Component definition into executable code

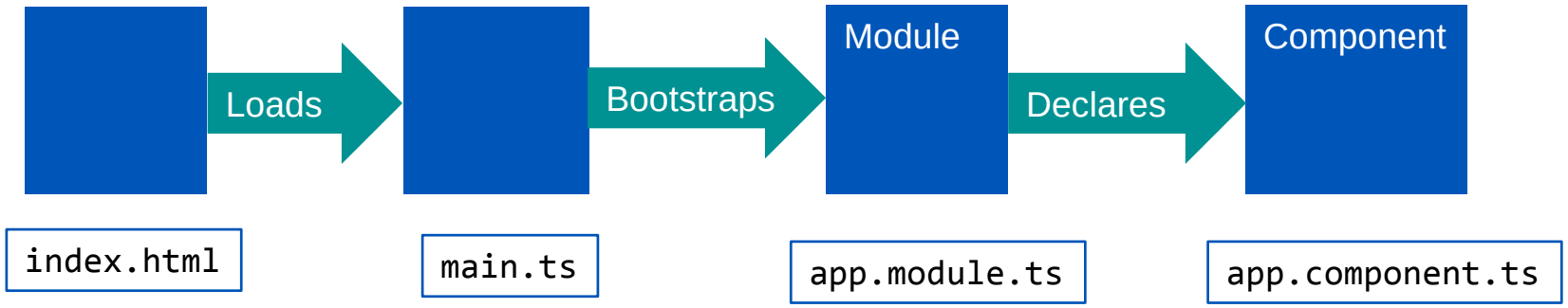
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Bootstrap Process

- **Browser loads `index.html`**
  - Includes an HTML tag identifying where to display the Component
  - Loads the transpiled version of `main.ts` (`main.js`)
    - As part of a bundle of JavaScript
- **`main.ts` bootstraps the application module (`AppModule`)**

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

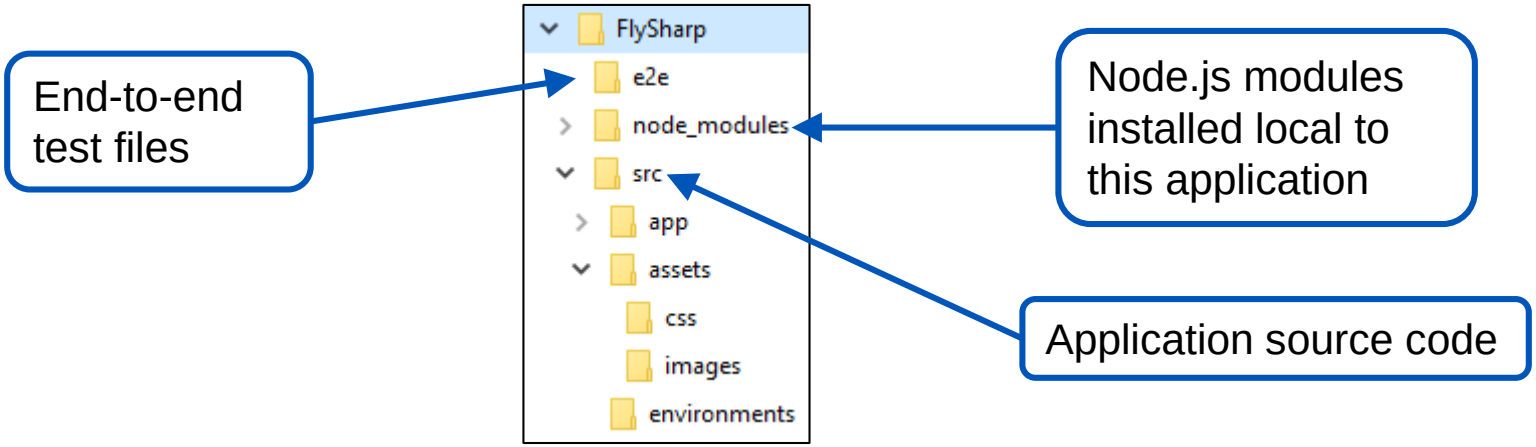
- **`AppModule` makes `AppComponent` available**
  - The main application Component
  - Loaded into the location specified in `index.html`



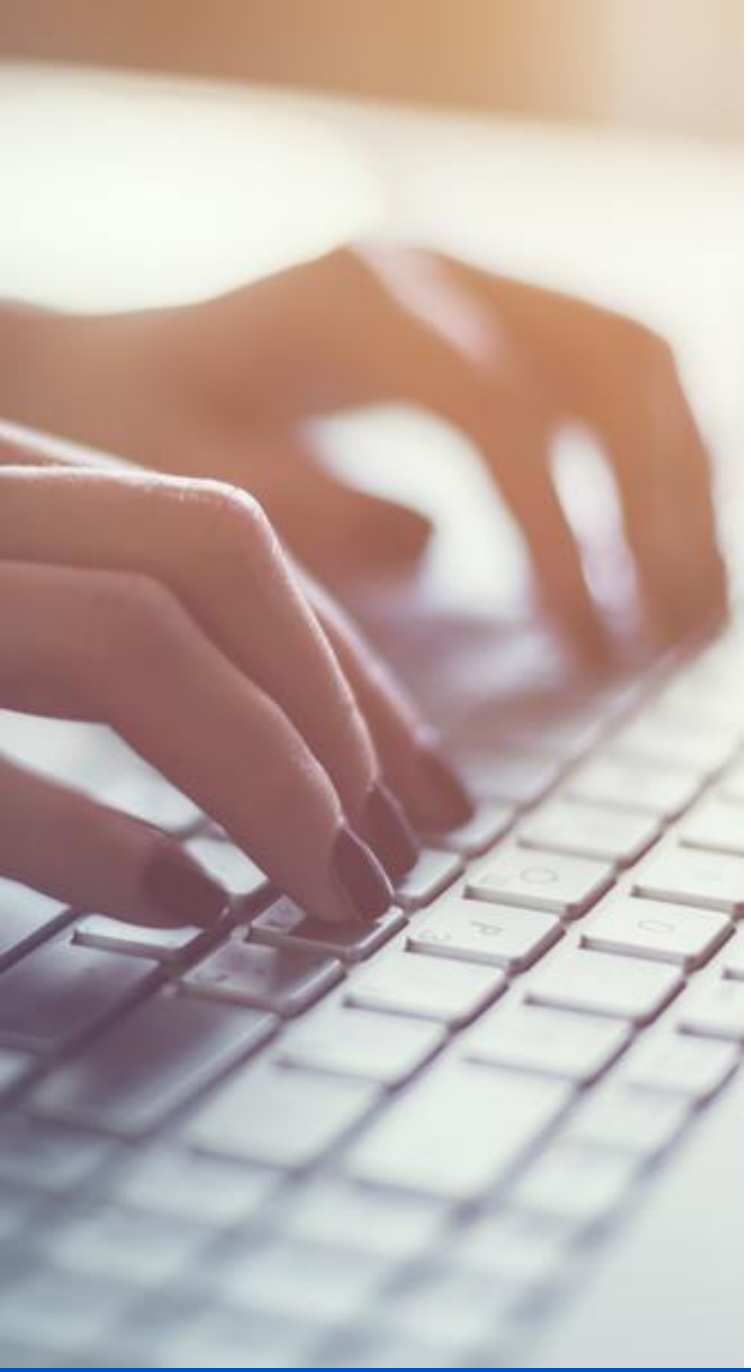
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Directory Structure

- **Angular CLI creates a standard directory structure**
  - As created in the previous Do Now exercise



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Development Environment Configuration

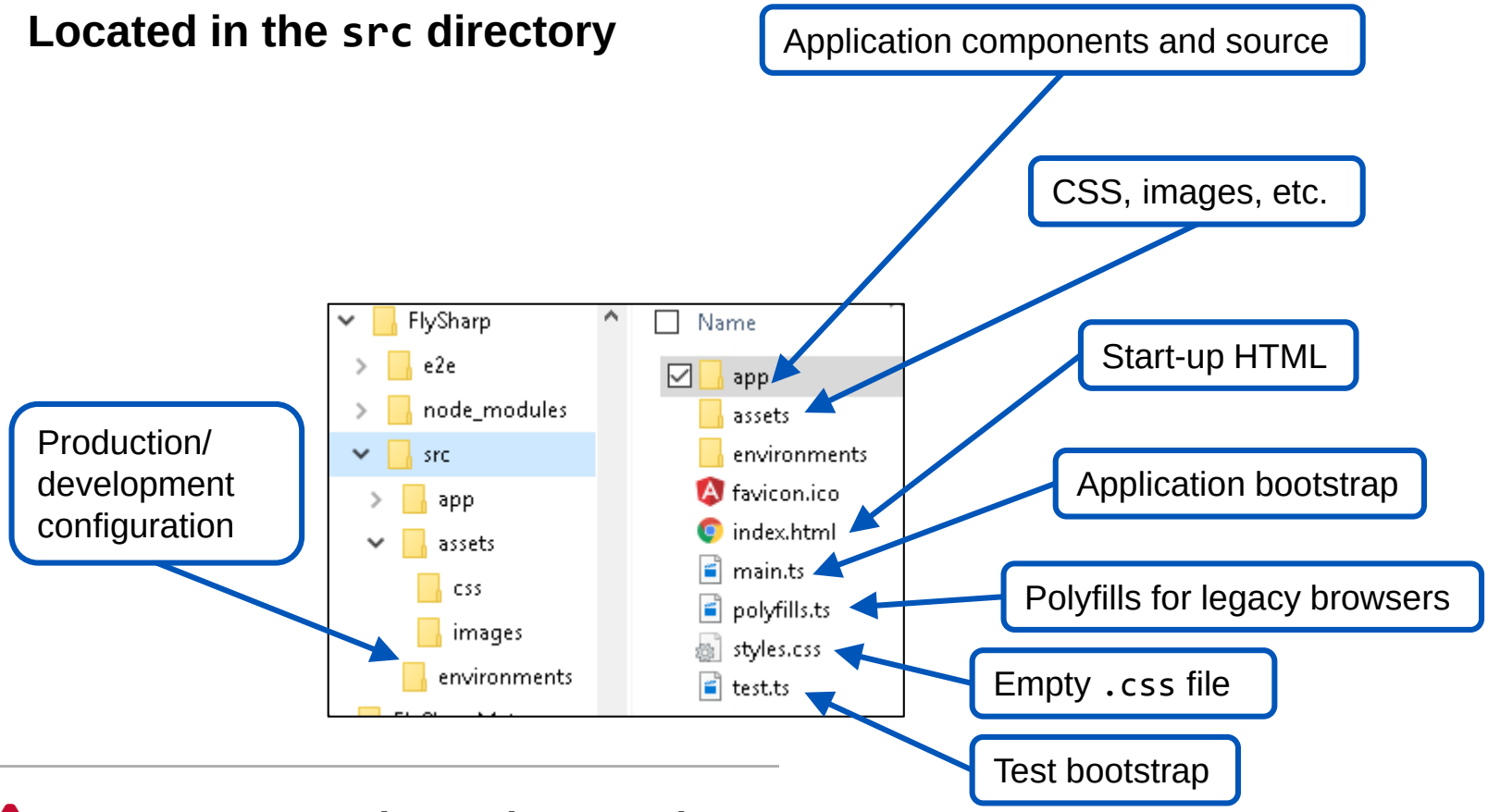
Reference

- **Multiple configuration files created in the project**
  - `angular.json`—overall configuration for Angular CLI
  - `karma-conf.js`—configuration for the Karma test runner
  - `package.json`—npm configuration
  - `E2e/protractor.conf`—configuration for the Protractor E2E test runner
  - `tsconfig.json, tsconfig.app.json`—configures the typescript compiler
  - `tslint.json`—configures the lint source code checking tool

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Application Source Structure

- **Located in the `src` directory**



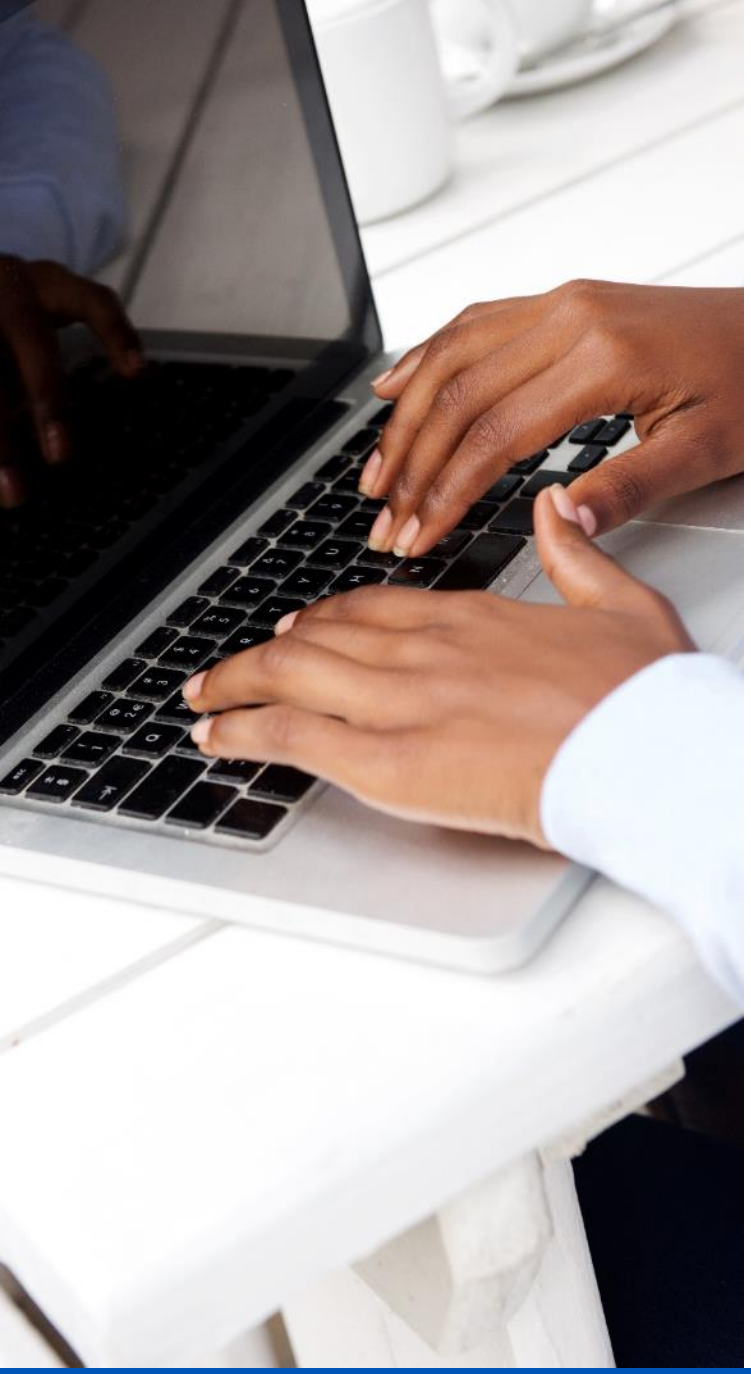
IMPORTANT: when using Angular CLI, all additional web assets (css, images, etc.) must be placed in the assets folder (or sub-folder)

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Choice of Editors

- **The instructions for the exercises specify WebStorm as the editor**
- **Feel free to use an alternative editor**
  - You will need to figure out any extra steps
    - Your instructor will help, but may not know the specifics of your chosen editor
- **The course load has Visual Studio Code and Visual Studio Community Edition installed**
  - Popular alternatives to WebStorm
  - Feel free to install your preferred editor
    - As long as you do not violate any license agreements!

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 1.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 1.1: Developing an Angular Application

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Benefits of Angular
- Architectural Overview
- Configuring an Environment
- Components of an Angular Application

TypeScript Primer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

TypeScript Is a Superset of JavaScript

- **If you know JavaScript, then you know the core of TypeScript**
  - Just change the file extension to `.ts`
- **TypeScript is transpiled to JavaScript**
  - Transpilation is part of the development process
  - Browsers are served JavaScript, not TypeScript

```
> tsc file.ts
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Simple TypeScript Example

- **TypeScript transpiles to JavaScript**
  - Can transpile to ES5 or ES6

starthere.ts

```
function getInfo(){
  return "TypeScript is really just JavaScript with a few
  bits added";
}

document.body.innerHTML = getInfo();
```

starthere.js

```
function getInfo(){
  return "TypeScript is really just JavaScript with a few
  bits added";
}
document.body.innerHTML = getInfo();
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Declaring Variables

- **Variables may be declared with `var`, `let`, or no keyword**
  - `var` makes a variable local to a function
  - `let` makes a variable local to a block
  - All other variables are global
- **Prefer `let` to declare variables**
  - Limits their scope to inside the block they are declared

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Static Types

- **Code is generally more robust if static type checking is used**
  - Typing restricts the value of a variable
- **TypeScript encourages variables and function returns to be typed**
  - Not available in JavaScript
- **Syntax is `variable : type`**
  - TypeScript supports types of:
    - `boolean`, `number`, `string`, `array`, `tuple`, `enum`, `object`, `void`
  - Special type of `any` – where the type is unknown

```
function someFunction() : string {
  let id : number;
  let flightNumber : string;
  return flightNumber;
}
```

- **Type declarations are not mandatory**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Interfaces

- **Interfaces constrain the "shape" of objects**

```
interface Flight{
  id : number;
  flightNumber : string;
  origin : string;
  destination : string;
  price : number;
}
```

: type used to declare a type

```
let flight1298 : Flight =
    {"id": 11, "flightNumber" : "FS1298", "origin":
  "LAX", "destination" : "LHR", "price" : 99.99};
```

- **If the id was "Flight 11" what would happen?** _____

- **JavaScript will still be generated by the transpiler even when there are errors**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Classes

- **Classes are templates for objects**
  - Describing both data and behavior
- **Data fields become properties on the generated JavaScript objects**
- **Methods look like function declarations**
  - Without the `function` keyword
  - Converted to JavaScript and added to object prototypes
- **Object constructor must be named `constructor()`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Classes

```
class AirlineFlight implements Flight {
  id : number;
  flightNumber : string;
  origin : string;
  destination : string;
  price : number;

  constructor(  flight : Flight ){
        this.id = flight.id;
        this.flightNumber = flight.flightNumber;
        this.origin = flight.origin;
        this.destination = flight.destination;
        this.price = flight.price;
  }

  showFlightInfo() : string {
        return this.flightNumber + " : " + " "  +
        this.origin + " " +  this.destination;
  }
}
```

Implements `Flight`, so must have these variables

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Objects From Classes

- **Objects are instantiated with the `new` keyword**
  - By calling an object constructor that has the same name as the class
  - This calls a method named `constructor` in the class
- **Methods and properties accessed using `'.'`**

```
let aFlight = new AirlineFlight(flight);
document.body.innerHTML = aFlight.showFlightInfo();
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Modules

- **Each TypeScript file is a module**
- **Variables, functions, and classes are not visible outside the module**
  - Unless exported

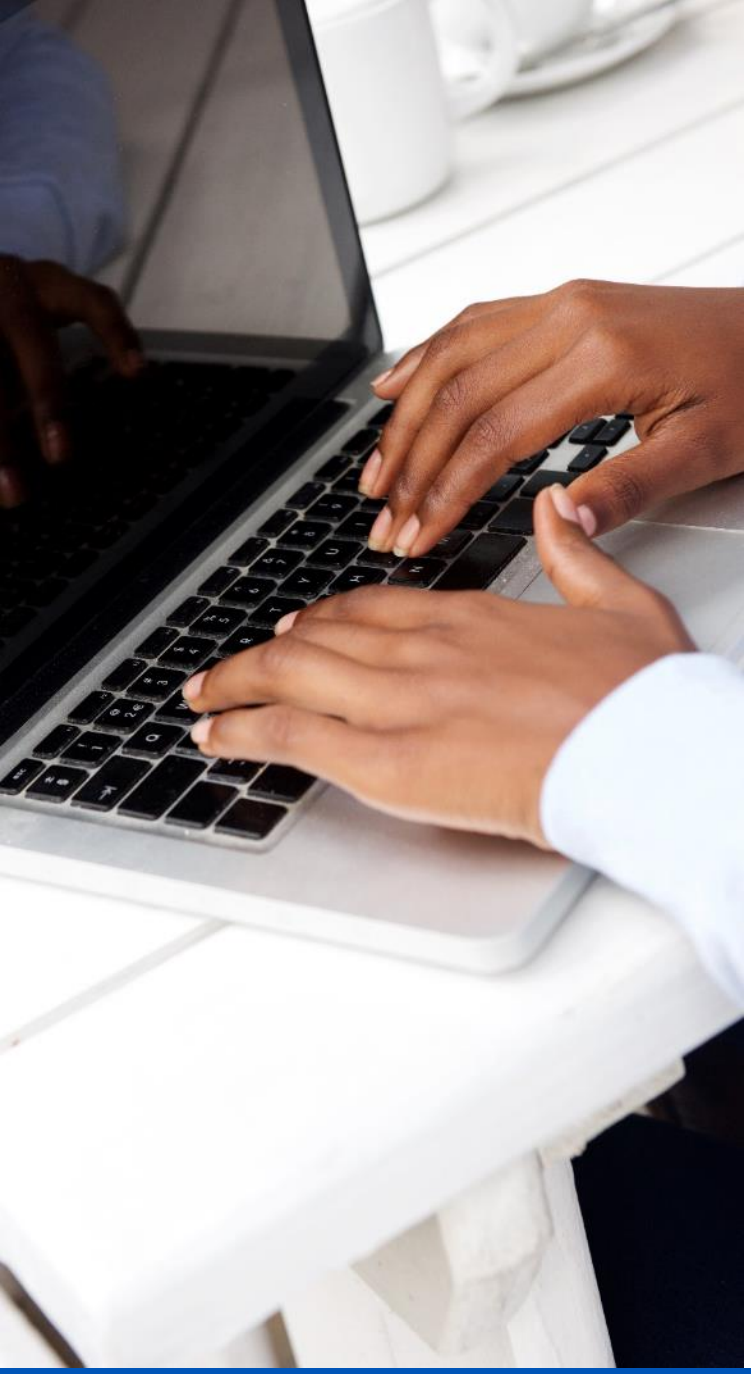
```
export interface Flight{ … }
```

- **Consuming code must import declarations before use**

```
import { Flight } from "./pathToFlightModule";
```

No `.ts` on the filename

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 1.2

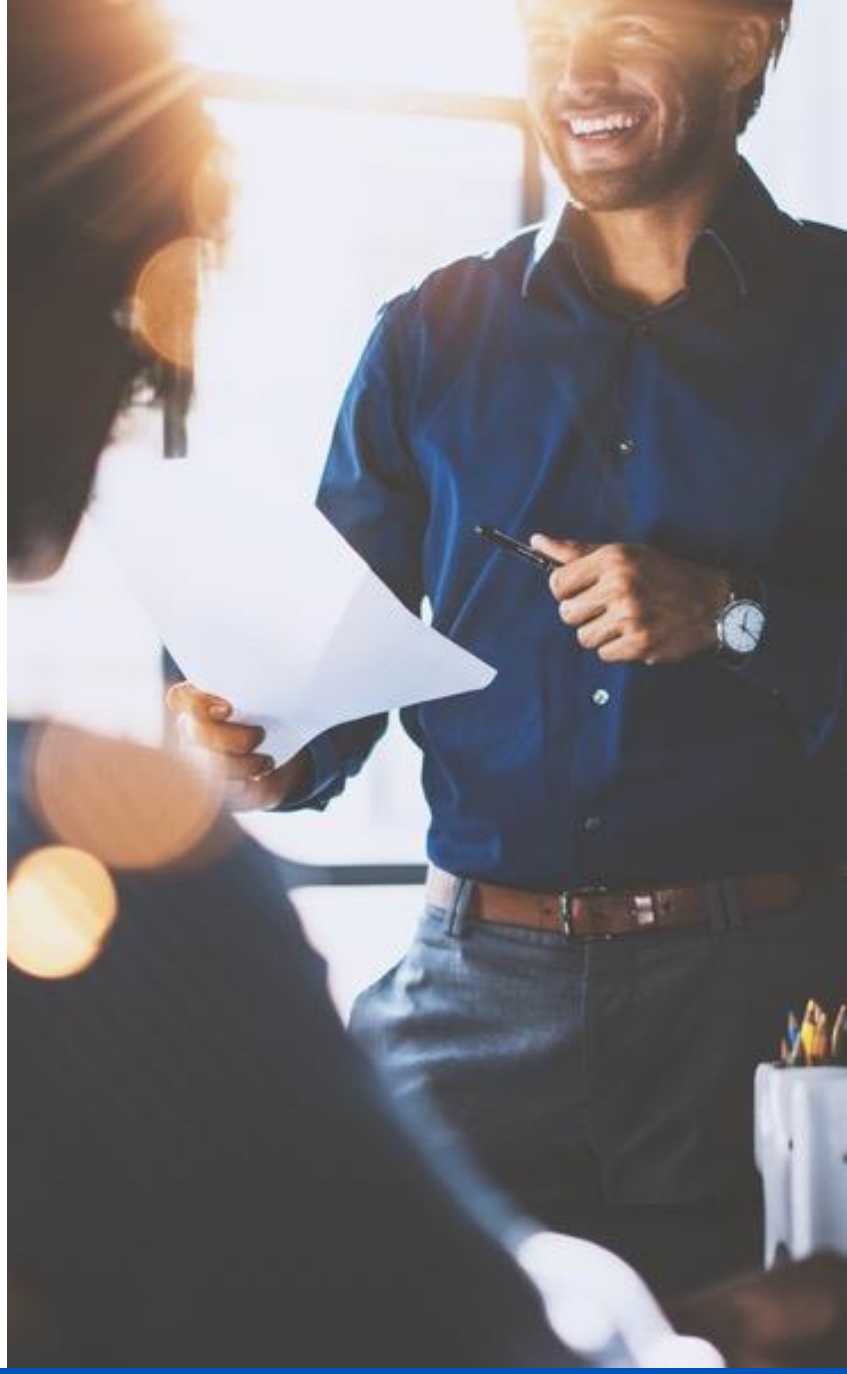
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 1.2: Introducing TypeScript

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

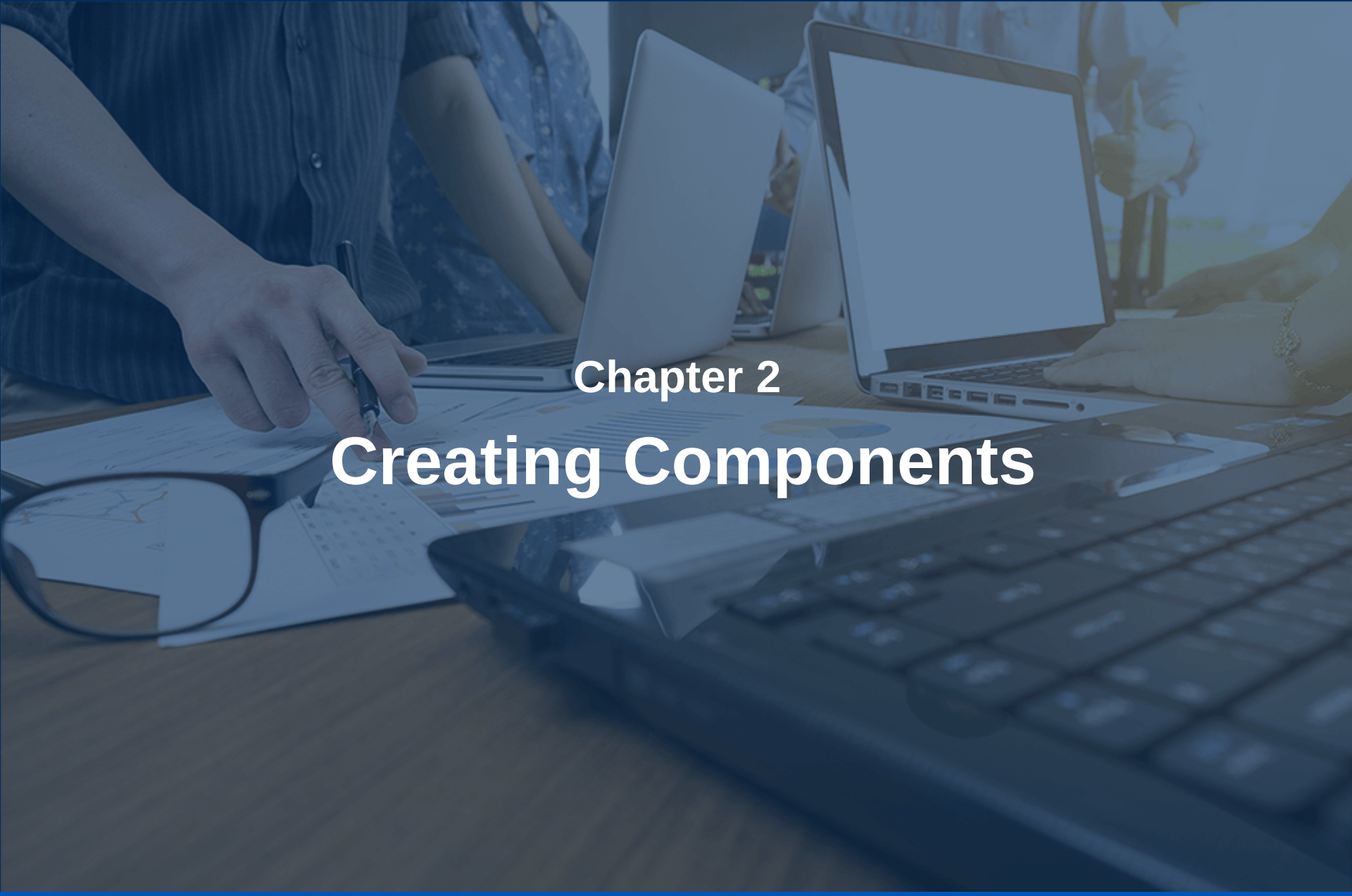
Objectives

- Learn how Angular eases the development of modern GUI applications
- Learn how to configure an Angular environment
- Learn how to navigate the architecture of Angular
- Learn essential TypeScript programming skills



GUI = graphical user interface

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



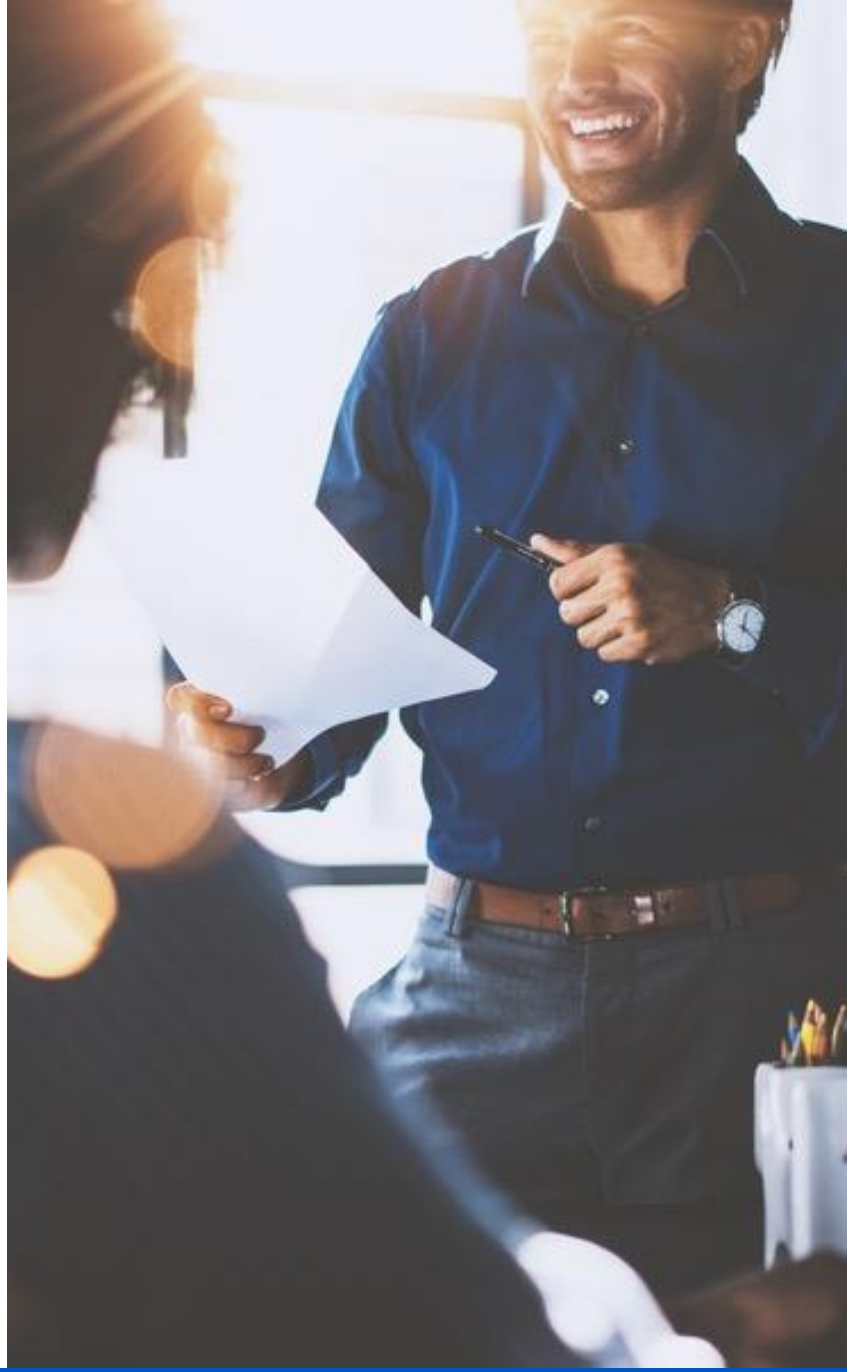
Chapter 2

Creating Components

LEARNING TREE INTERNATIONAL

Objectives

- Create Component definitions
- Specify behavior and relationships with metadata
- Control visual appearance with templates
- Bind data to templates
- Generate repeating and conditional templates structures



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

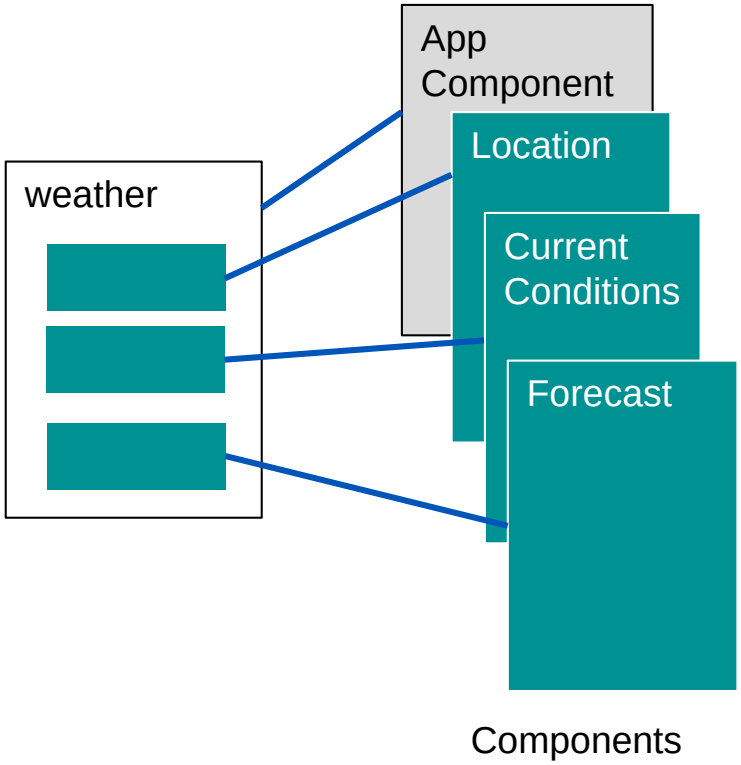
Role and Structure of Components

- Angular Modules
- Component Metadata
- Component Templates
- Debugging Techniques
- Controlling Appearance
- Repeating Data

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Angular Components

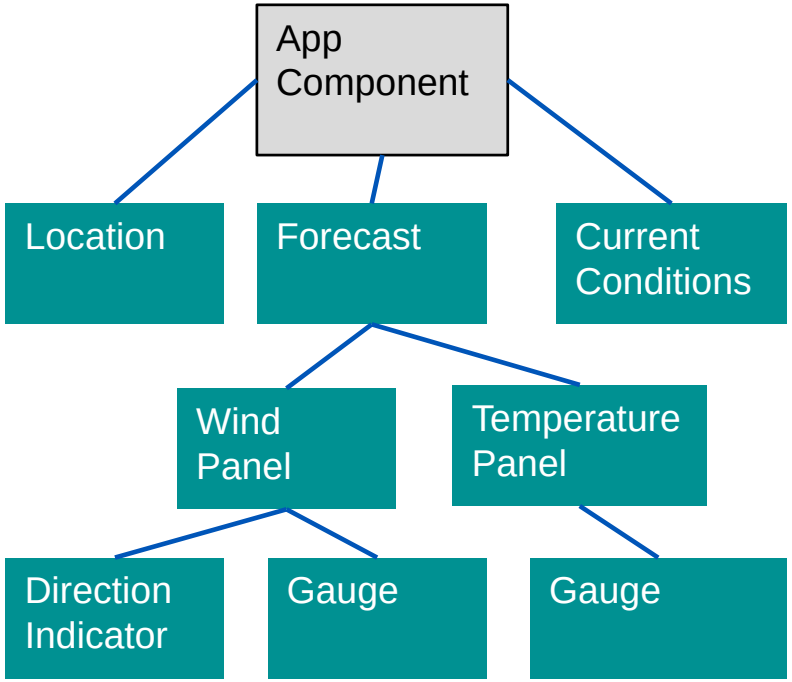
- **The building blocks of Angular**
- **A Component controls part of the screen**
  - A view
- **Defines the visual appearance**
  - Through a template
- **Controls the behavior**
  - Through a TypeScript class



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Hierarchies

- **Components may be composed of other Components**
  - They form a hierarchy
- **Components may be reused as children of multiple Components**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Structure

- **Four elements to a Component**
  - Metadata
  - TypeScript Class
  - Template
  - Styling
- **Metadata and the class declaration defined in the main `*.component.ts` file**
- **Template and styling normally placed in external files**
- **Template and styling can be placed inside the metadata section**
  - Avoid for all but the simplest Components

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Example Component TypeScript File

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css']
})

export class AppComponent {
  title = 'app works!';
}
```

Metadata

Class

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

TypeScript Imports

- **Angular is composed of multiple TypeScript modules**
  - Which must be imported before being used in code
- **Two styles of import**
  - System imports
    - Make functionality defined in system modules available to our code

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

Components to import

Module identifier

  - Local imports
    - Importing code from other source files in our project

```
import { GaugeComponent} from './gauge/gauge.component';
```

Relative path

Note: The `@` symbol in `@angular` is part of the module name—it has no other significance

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

2.1: Resolving `Imports`

Do Now

- **Angular code requires `imports` for multiple modules**
- **The WebStorm editor usually adds `import` statements automatically**
- **We have added the TypeScript Importer extension to Visual Studio Code**
  - Provides assistance with `imports`
- **If all else fails**
  - Use the documentation

Example: find the correct `import` for `Component`

1. **Open `angular.io/api` in a browser**
2. **You should see three text boxes. In the Filter box type `Component`**
3. **Click on `Component` in the list of matching types**
4. **Near the top of the page you should see `npm package: @angular/core`**
5. **Which tells us the `import` will be:**

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

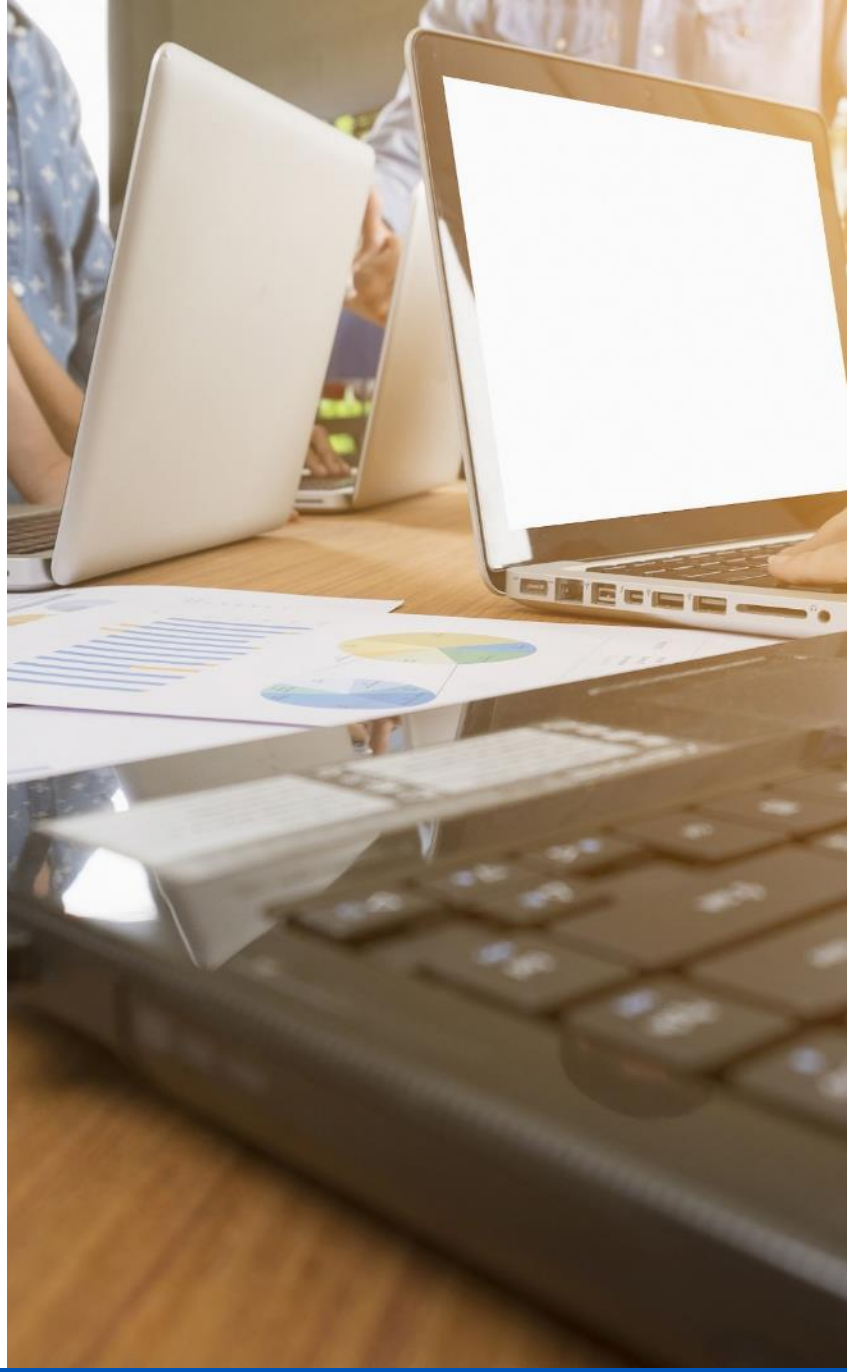
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Role and Structure of Components

Angular Modules

- Component Metadata
- Component Templates
- Debugging Techniques
- Controlling Appearance
- Repeating Data



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Modules

- **Angular application Components must be defined within modules**
  - Allows grouping of related functionality
- **Every Angular application must have at least one module**
  - The *root* module
  - Additional modules are known as feature modules
- **Angular modules contain one or more Components**
- **Angular modules are *not* the same as TypeScript modules**
  - TypeScript modules are a single source file
    - Made visible to another source file through an `import` statement
  - Angular modules provide metadata instructing Angular how to process Components
    - Loaded at runtime as part of the bootstrap process
    - Or loaded by the Component Router

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structure of an Angular Module

- **Angular modules are classes**
  - Decorated with `@NgModule`
  - Imported from `@angular/core`
  - Provides metadata about the module
  - Metadata is defined as an object literal
    - Passed as a parameter to the `@NgModule` decorator
    - `@NgModule({…})`

- **Metadata properties include**
  - `declarations`—identifies the Components, Directives, and pipes contained in the module
  - `imports`—references to Angular modules needed by this module
  - `providers`—used for dependency injection
  - `bootstrap`—the Component to load when the module is bootstrapped

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Example Module

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
…
import { AppComponent } from './app.component';

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

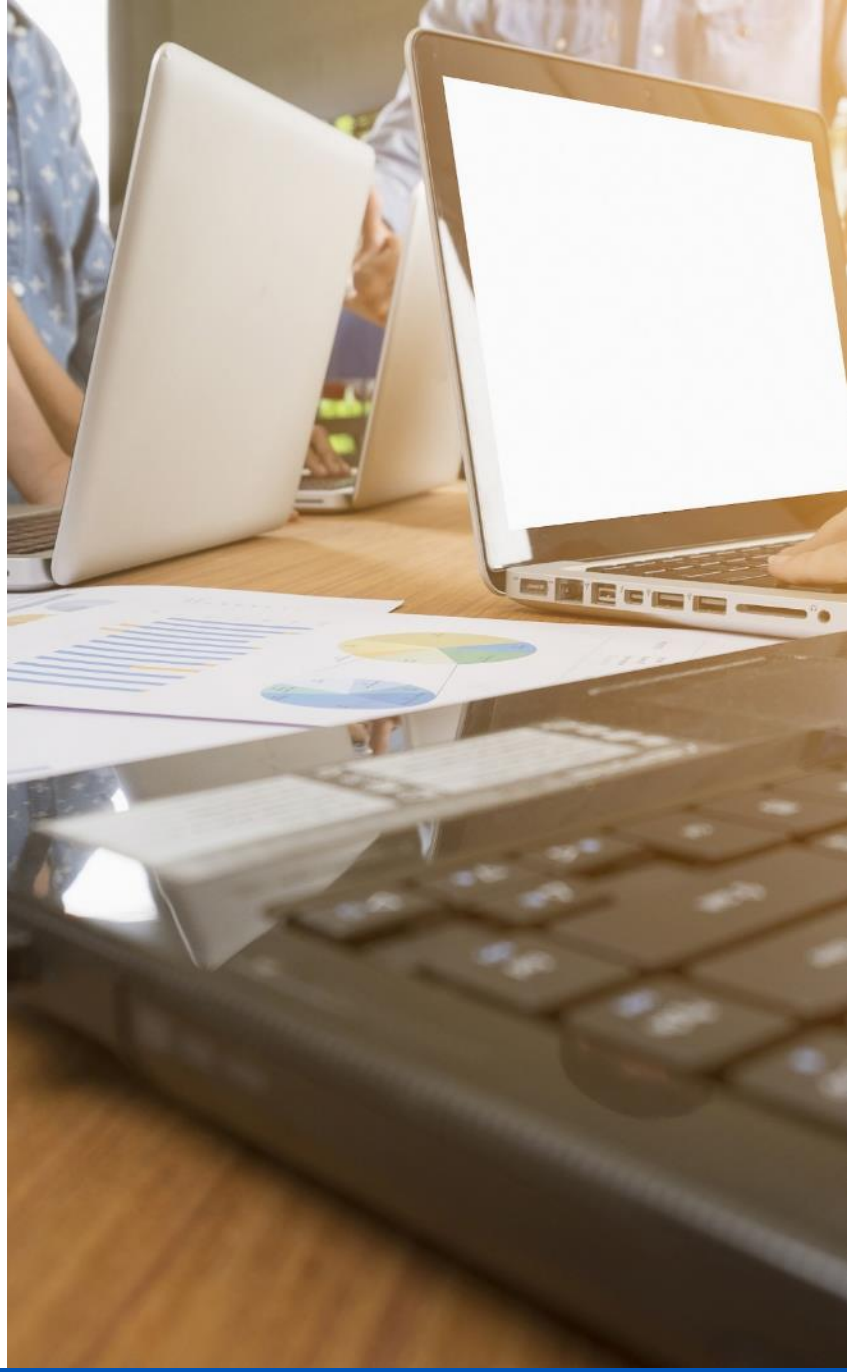
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Role and Structure of Components
- Angular Modules

Component Metadata

- Component Templates
- Debugging Techniques
- Controlling Appearance
- Repeating Data



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Metadata

- **Defined by the `@Component` decorator**
  - Imported from `@angular/core`

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css']
})
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Metadata Breakdown

- **`selector`—CSS selector**
  - Matches the element in the parent Component or HTML page that identifies where this Component is to be placed

index.html

```
<body>
  <app-root>Loading...</app-root>
</body>
```

There are other metadata properties; they will be introduced as needed.

- **`templateUrl`—relative path to the HTML template file**
- **`styleUrls`—an array of relative URLs to CSS files for this Component**
  - CSS can also be applied globally if included in the containing HTML page

There are other metadata properties; they will be introduced as needed.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Class Declaration

- **Component classes must be exported**
- **Properties and methods may be marked `private`**
  - Stops them being used from other components
- **Private fields and methods are not visible in the template**
  - Though, unfortunately, they may work with older (pre Angular 8) development builds!

```
export class AppComponent {
  private title = 'app works!';

  onClick(){
    this.title='new title';
  }
}
```

Property

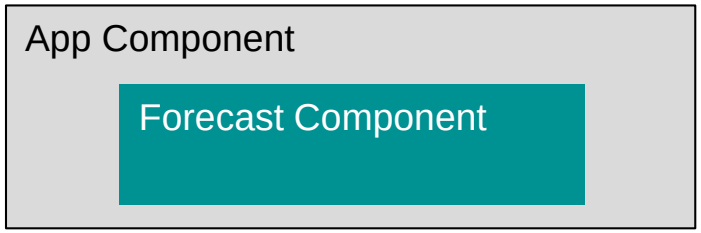
Method

`private` would mean the `title` field would not be visible in the template!

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Components

- **Components can be composed of Components**
  - As we already know…
- **Position of child Component is defined by the template of the parent**
- **Child Components must be declared in a Module**
- **To add a child component**
  1. Import the Component into the `.ts` file for the module
  2. Add the child Component class name to the `declarations` property of the `@NgModule` metadata
  3. Add an element matching the CSS selector of the child to the HTML template of the parent



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Components Example—Module

app.module.ts

```
import { ForecastComponent } from
  './forecast/forecast.component';

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

1

2

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Components Example—Parent Template

app.component.html

```
<h1 (click)="onClick()">
  {{title}}
</h1>
<app-forecast></app-forecast>
```

3

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Components Example—Child Component

forecast.component.ts

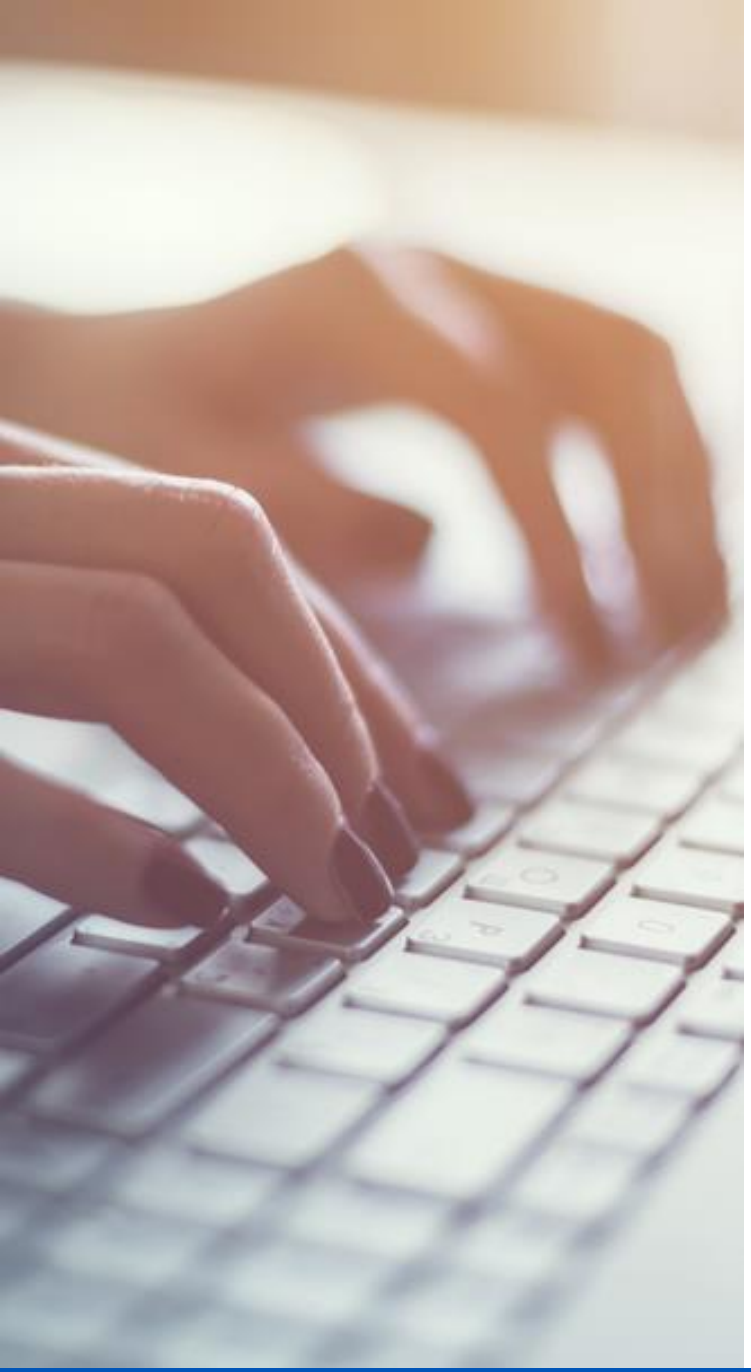
```
@Component({ …
  selector: 'app-forecast', …
})
export class ForecastComponent {…}
```

forecast.component.html

```
<p>
  Weather is expected
</p>
```

- **If a Component is declared in the `declarations` property of the `@NgModule` metadata**
  - Angular will look for templates with CSS selectors matching those Components
- **If HTML matching a CSS selector is found**
  - The declaring Component is displayed at that point in the parent template

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



2.2: Multiple Components

Do Now

- In your Exercise Manual, please refer to Do Now 2.2: Multiple Components

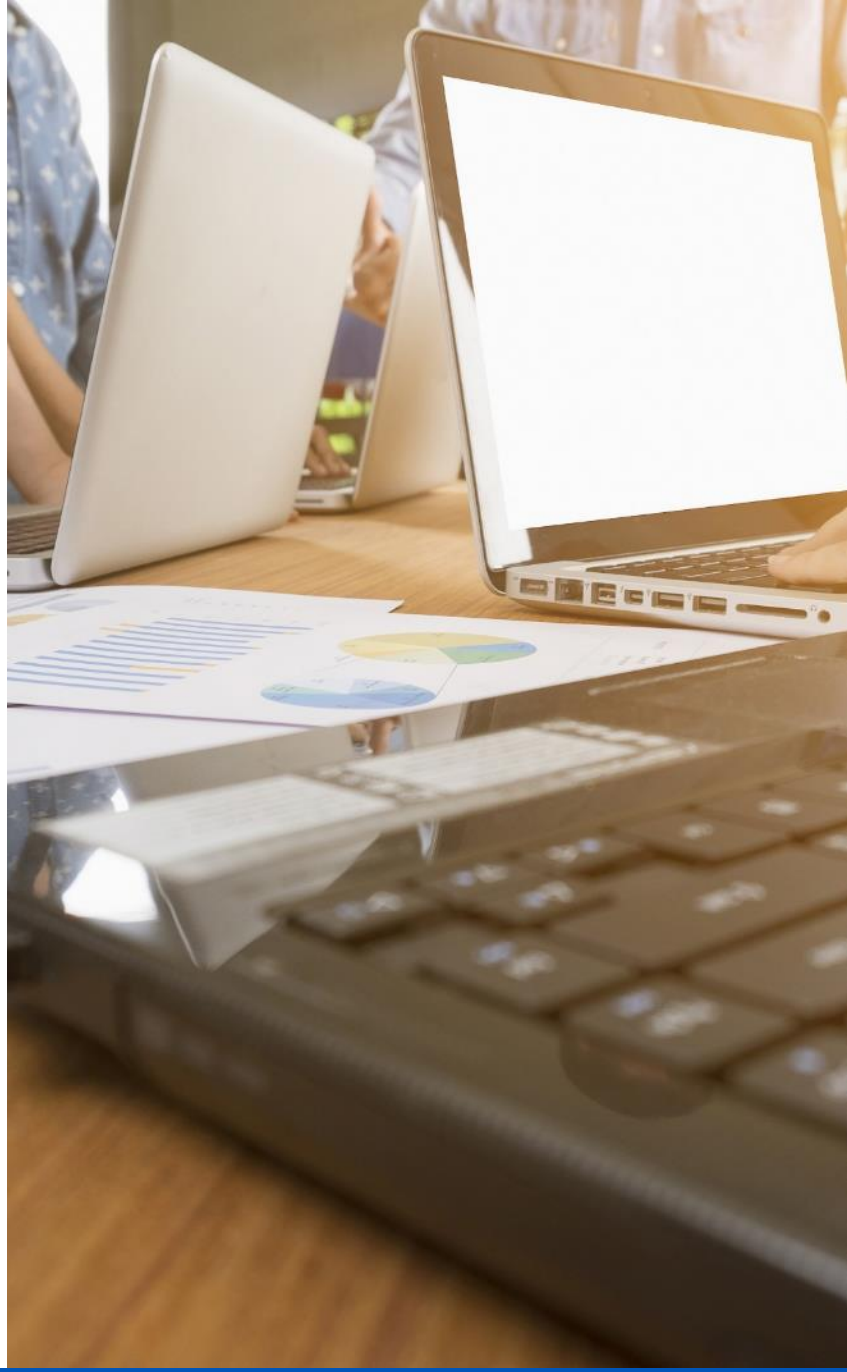
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Role and Structure of Components
- Angular Modules
- Component Metadata

Component Templates

- Debugging Techniques
- Controlling Appearance
- Repeating Data



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

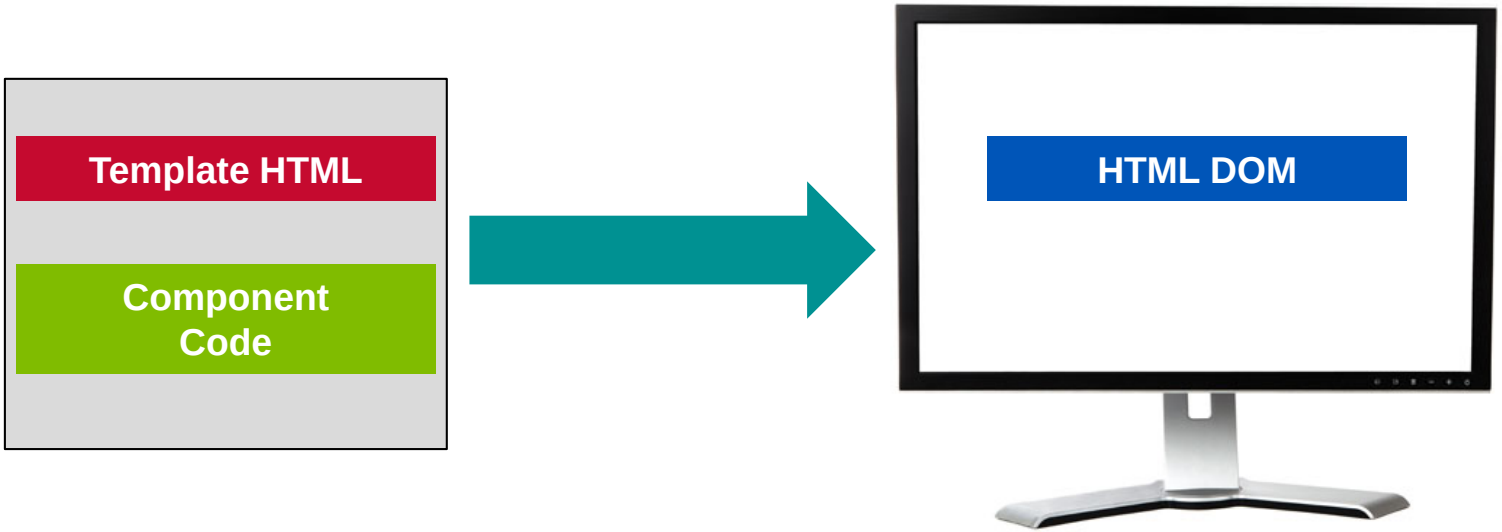
Angular Template

- **Templates define the view for a Component**
- **Using HTML 5 syntax**
  - Plus some additions—bindings
  - Minus some restrictions
- **Binding Types**
  - One-way—data source to view
    - Interpolation `{{expression}}`
    - Property `[target]="expression"`
  - One-way from view to data source
    - Event ()
  - Two-way
    - `Banana in a box [()]`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Rendering

- **Templates are the source of the HTML output to the browser**
  - Angular generates the browser HTML DOM
  - By combining interpolated values from the Component with the template



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Interpolation

- **Calculated values inserted into the template output**
- **Syntax: `{{expression}}`**
- **`expression` is a template expression**
  - Often the name of a property of the Component
- **Interpolation may be used**
  - As the body of an HTML element
  - As the value of an HTML attribute

```
<h1 (click)="onClick()">
  {{title}}
  <img src="{{weatherImgUrl}}">
</h1>
<app-forecast></app-forecast>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Expressions

- **Statements written in a template expression language**
  - An Angular-specific variant of JavaScript
- **Angular executes the expression and assigns the value to a target**
  - HTML element or attribute
  - Other binding targets
- **Prohibited expressions**
  - In interpolation statements: anything that has a side effect
    - For example, modifies something
      - `=`, `+=`, `-=`, `*=`, `++`, `--`, …
      - `new`
  - In all expressions
    - Chaining expressions with `;` or `,`
    - Bitwise operators `|` and `&`
- **Additional operators**
  - Pipe `|`
  - Safe navigation operator `?.`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Expression Context

- **Expression context for `<script>` in a web page is global namespace**
  - Can access objects like `window` and `document`
- **Expression context for Angular expressions is the current Component**
  - `{{title}}` accesses the title property on the current Component
  - `"onClick()"` is the `onClick()` method of the current Component
- **Template expressions cannot access anything in the global namespace**
  - `window`, `document`, `console`, `Math.*`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Interpolation Expressions

```
export class WindComponent {

  windSpeedKnots : number = 20;
  instabilityIndex : number = 1.8;
}
```

```
 <li>Wind Speed (Knots) - {{windSpeedKnots}}</li>
  <li>Wind Speed (MPH) - {{windSpeedKnots * 1.1}}</li>
  <li>Max Speed (Knots) –
              {{windSpeedKnots * instabilityIndex}}</li>
```

- Wind Speed (Knots) - 20
- Wind Speed (MPH) - 22
- Max Speed (Knots) - 36

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Property Binding

- **Sets the value of a property to the value of an expression**
  - `[target]="expression"`
  - Data goes one way—from expression to target property
- **Target property may be**
  - An HTML DOM property

```
<button [disabled]="buttonDisabled">Update Weather</button>
```

  - A Component property

```
<app-current-weather [preferedUnits]="'Knots'"></app-current-
  weather>
```

Important: Square brackets for property bindings

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Square Brackets?

- **The normal form of a property binding is**
  - `[target]="expression"`
- **Cumbersome if the expression is a string**
  - `[target]="'string expression'"`
- **An alternative form is to omit the [ ]**
- **Expression is then assumed to be a string**
  - `target="string expression"`
  - i.e., not evaluated as an expression

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Event Binding

- **Bind event handlers to events**
  - `(event)="eventHandler()"`
- **The event may be**
  - A DOM event—`click`, `mouseover`, `mouseout`, …
  - A custom event from a Component
- **The `eventHandler()` is a method on the Component**
  - Or an expression (with side-effects) in the template

```
<button (click)="myClickHandler()">Update Weather</button>
```

Important: Round brackets for event bindings

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Canonical Form Bindings

- **Angular offers an alternative syntax for bindings**
- **Property bindings**
  - Either `[target]="expression"`
  - Or `bind-target="expression"`

```
<button bind-disabled="buttonDisabled">Update
  Weather</button>
```

- **Event bindings**
  - **`(event)="eventHandler()"`**
  - **`on-event="eventHandler()"`**

```
<button on-click="myClickHandler()">Update Weather</button>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Two-Way Bindings

- **Used to both display a property and update it when the user changes it**
  - For form data processing
  - Works with a special directive: `ngModel`
- **`[(ngModel)]="property"`**

```
<input [(ngModel)]="city">
```

- **Canonical syntax**

```
<input bindon-ngModel="city">
```

Tip: Looks like a banana in a box! `[()]`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Single-File Components

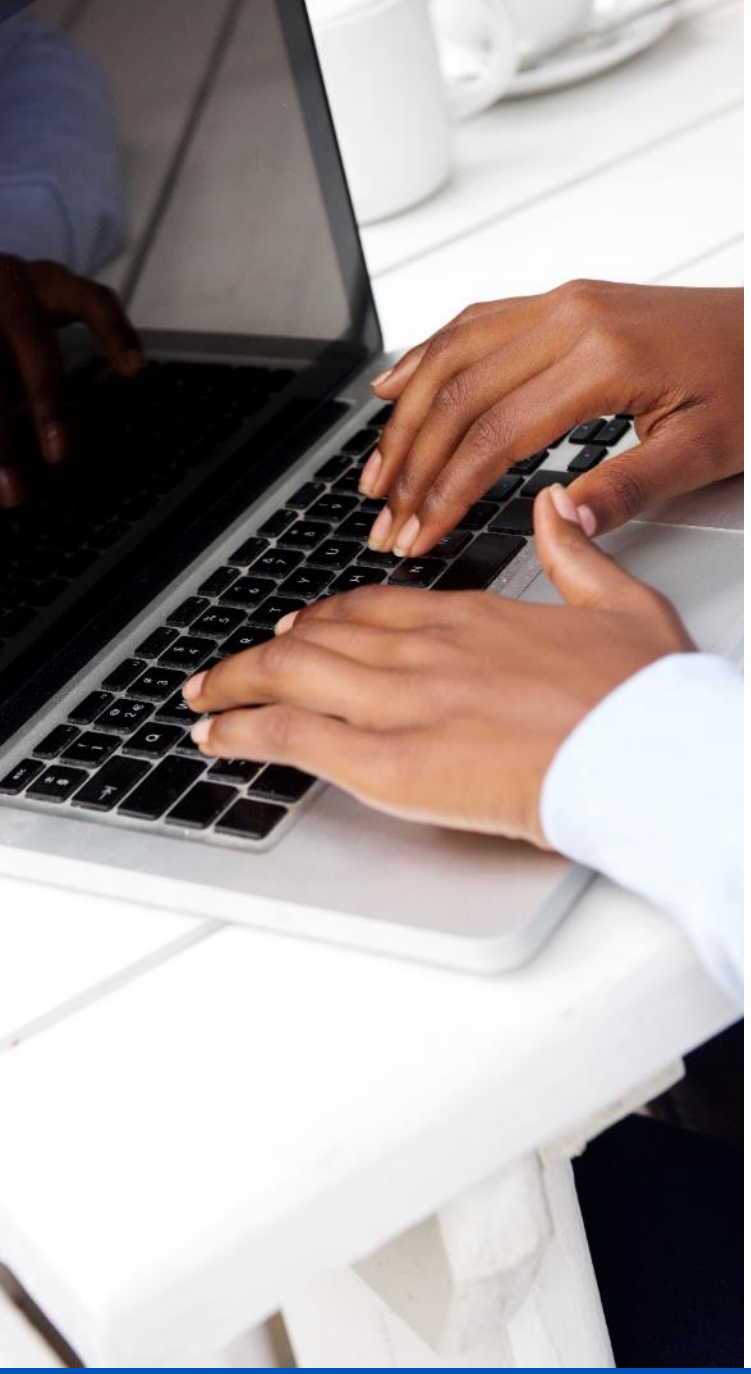
- **Splitting a Component into multiple files is good practice**
  - May be overkill for small Components
- **Template HTML and CSS may be embedded in the Component metadata**
  - Use backtick (`) character for multi-line entries

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

@Component({
  selector: 'app-forecast',
  template: `<p>
    Weather is expected
  </p>`,
  styles: ['p {color:blue;}']
})
export class ForecastComponent  {
}
```

`template` replaces `templateUrl`

`styles` replaces `styleUrls`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 2.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 2.1: Creating an Angular Component

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise Debrief

Debrief

- **You have seen how to create a Component**
- **Applications have lots of them**
- **The good news**
  A. Angular CLI provides a mechanism to do the repetitive stuff for you
     - Creates all the files and skeletons for you
  B. In future exercises, we will use Angular CLI wherever possible

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Role and Structure of Components
- Angular Modules
- Component Metadata
- Component Templates

Debugging Techniques

- Controlling Appearance
- Repeating Data

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Debugging and Troubleshooting

- **Frameworks are great**
  - Saves vast amounts of effort
    - When it all works!
- **Fault-finding can be challenging**
  - Until you know how

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Debugging Tools

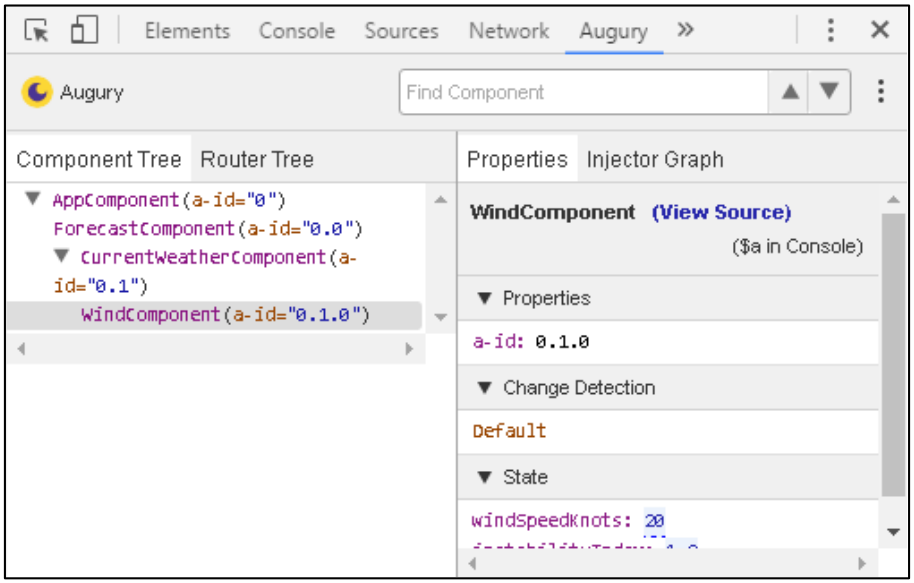
- **TypeScript can be debugged with**
  - Chrome Web Developer tools
  - Visual Studio
- **Augury plug-in for Chrome**
  - Angular explorer

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Augury

Demo

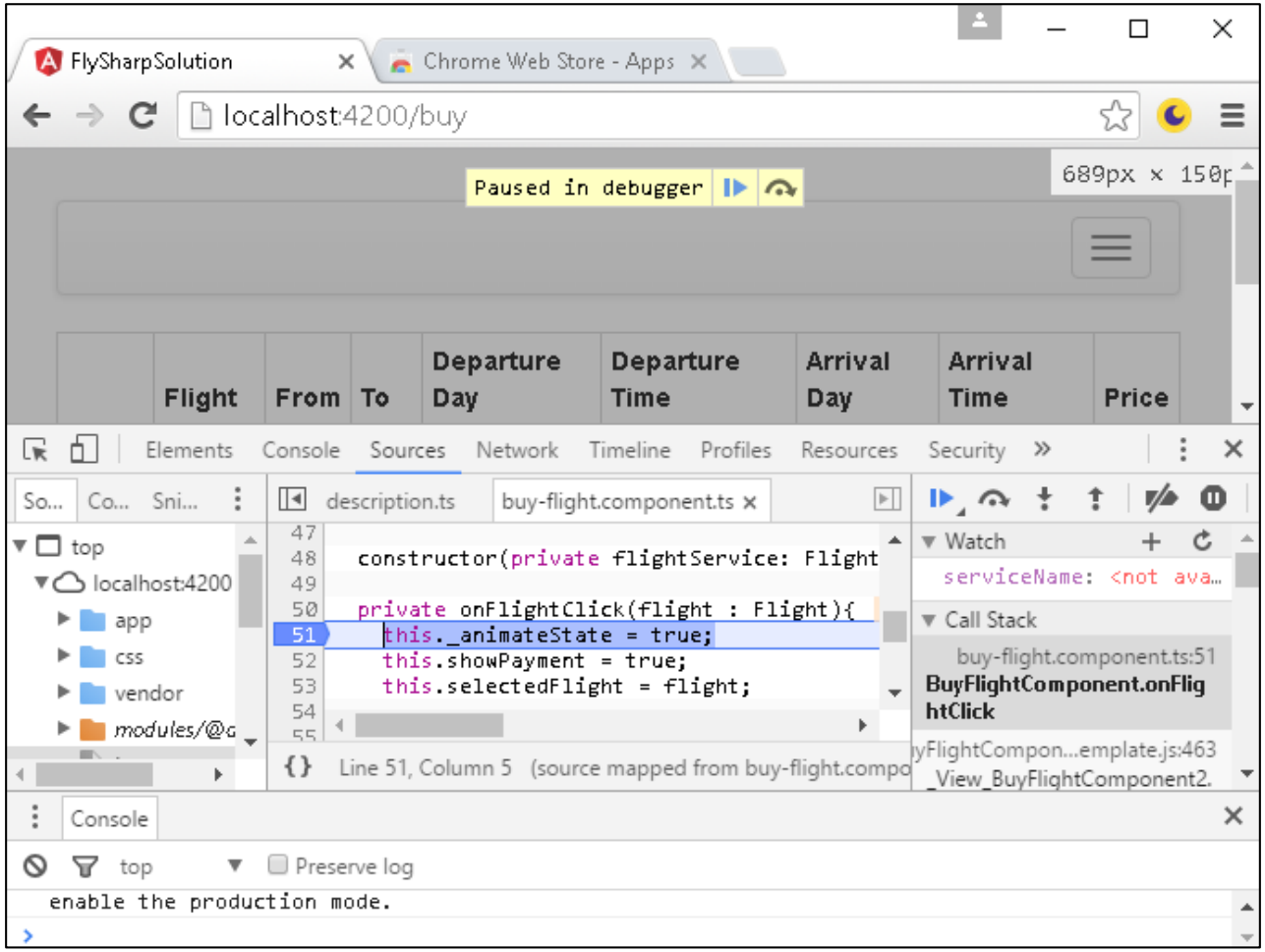
- **A Chrome extension**
  - Add to Chrome from **Settings | Extensions | Get more extensions**
- **Use to explore the Angular Component tree**
  - View Source button takes you to the source for the selected Component



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Debugging With Chrome Web Developer Tools

- **Chrome Web Developer tools allows debugging of TypeScript**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

What to Do When Nothing Is Displayed

- **Occasionally you will have a situation where**
  - No errors are reported by the TypeScript compiler
  - Nothing is displayed
- **Don’t panic!**
  - Open the developer console on your browser (`F12`)
  - Check whether the console output is visible
  - If there are lots of errors looking like

    `Error Uncaught (in promise)…`
    - It’s most likely a problem with the template HTML
      - Missing " on the end of attributes is very common problem
      - Check for unclosed element tags
- **If you are lucky**
  - The console may tell you the exact error
  - Read the message
  - Try copying the console message into an editor
    - You may see more information

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Template Parsing Error

Demo

- **Can you see the error?**

```
EXCEPTION: Template parse errors:
Parser Error: Unexpected token ')' at column 27 in [showPayment =
  !showPayment)] in BuyFlightComponent@3:49 ("
<tbody>
  <tr *ngFor="let flight of flights">
      <td><button class="btn btn-primary btn-sm" [ERROR -
  >](click)="showPayment = !showPayment)">Buy</button></td>
      <td>{{flight.flightNumber}}</td>
      "): BuyFlightComponent@3:49
```

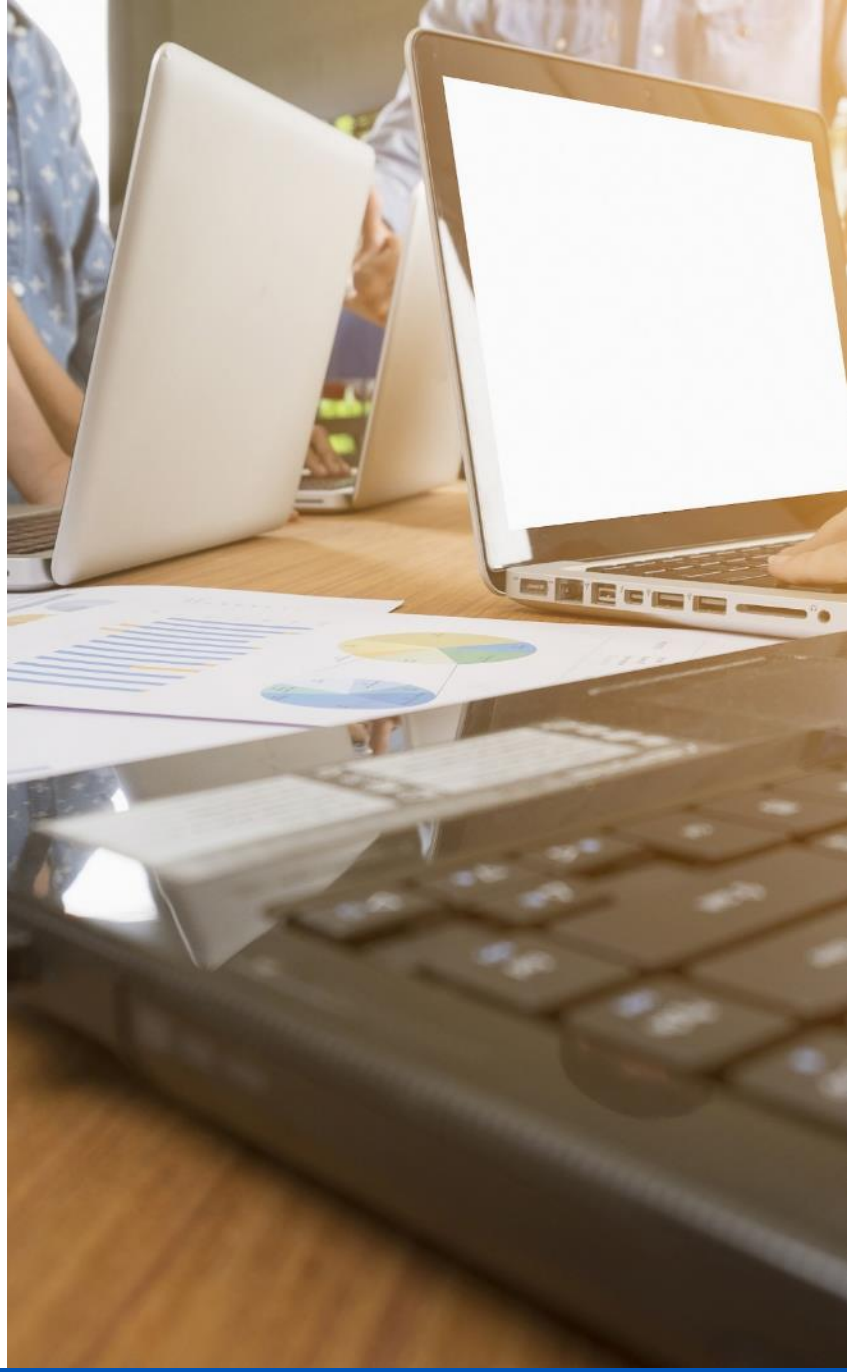
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Role and Structure of Components
- Angular Modules
- Component Metadata
- Component Templates
- Debugging Techniques

Controlling Appearance

- Repeating Data



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

NgClass Binding

- A built-in directive to simplify CSS class control
- Allows direction manipulation of CSS classes associated with an element

```
<li [class.highlight]="highlightSpeed">Wind Speed (Knots) -
  {{windSpeedKnots}}</li>
```

If `highlightSpeed` is truthy

```
<li class="highlight">Wind Speed (Knots) -
  {{windSpeedKnots}}</li>
```

If `highlightSpeed` is falsey

```
<li>Wind Speed (Knots) - {{windSpeedKnots}}</li>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

NgClass Directive

- Use ngClass directive to associate multiple classes with an element

Template

```
<li [ngClass]="classes">Wind Speed (MPH) - {{windSpeedKnots
  * 1.1}}</li>
```

TypeScript

```
classes : string[] = ["listStyle",
                      "highlight",
                      "indent"];
```

HTML

```
<li class="listStyle highlight indent">Wind Speed (Knots) -
  {{windSpeedKnots}}</li>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

NgStyle Binding

- **NgStyle binding: `[style.propertyName] = "expression"`**
  - Where expression evaluates to a CSS property value

Template

```
<li
  [style.color]="highlightSpeed ? 'red' : 'green'">
  Wind Speed (Knots) - {{windSpeedKnots}}</li>
```

TypeScript

```
highlightSpeed : boolean;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

NgStyle Binding and Directive

- **NgStyle directive: [ngStyle] = "expression"**
  - Where expression returns an object that is a set of styles

Template

```
<li [ngStyle]="getStyles()">Wind Speed (MPH) -
  {{windSpeedKnots * 1.1}}</li>
```

TypeScript

```
getStyles() {
    return {
      'font-family' : 'Arial, Helvetica, sans-serif',
      'color' : 'cyan',
      'font-style' : 'italic'
    }
  }
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Content Hiding

- **Common to have HTML that is conditionally displayed**

Template

```
<div [style.display]="raining ? 'block' : none" >
  Bring an umbrella!
</div>
<div [style.display]="sunny ? 'block' : none">
  Don't forget the sun screen!
</div>
```

- **`style.display = 'none'` prevents the content from being displayed**
  - But it is still in the DOM
- **Hiding content with CSS is very common**
- **Is it a good idea?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Conditional Generation of Content With `*ngIf`

- **CSS content hiding can slow both rendering and change detection**
  - What if the conditional Component had nested Angular Components?
- **`*ngIf` allows conditional content generation**
  - If the condition is false then no DOM content is created

Template

```
<div *ngIf="raining" >
  Bring an umbrella!
</div>
<div *ngIf="sunny">
  Don't forget the sun screen!
</div>
```

Tip: Huge potential performance benefits

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Choices With ngSwitch

- `[ngSwitch]="expression"` specifies the condition to be tested
- The `ngSwitchCase` matching the result of the expression will be generated
- Two version of `ngSwitchCase`
  - `[ngSwitchCase]="expression"`
  - `ngSwitchCase="literal"`
- If nothing else matches: `ngSwitchDefault` case is generated

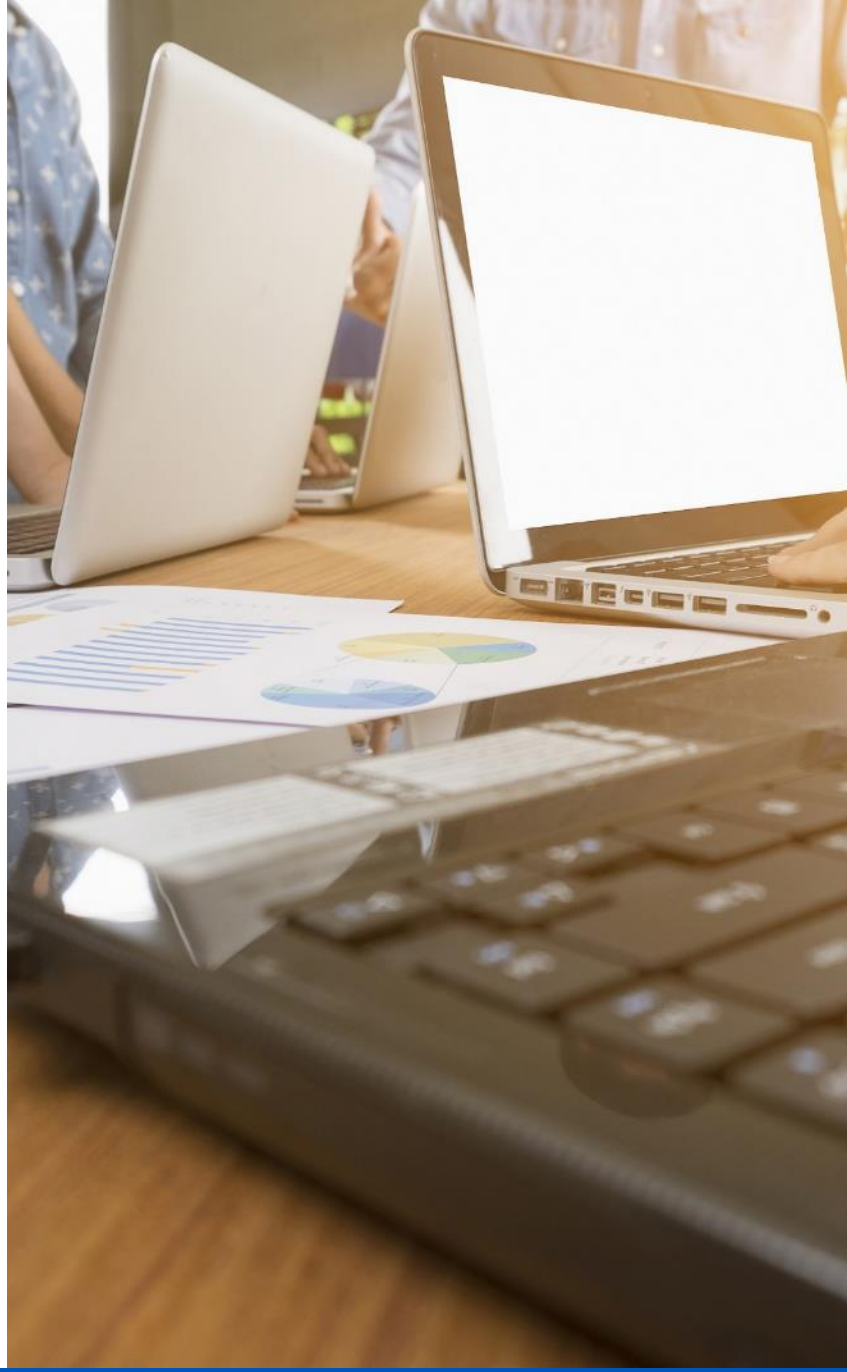
Template

```
<span [ngSwitch]="conditions">
  <template [ngSwitchCase]="'raining'">Bring an
  umbrella!</template>
  <template ngSwitchCase="sunny">Don't forget the sun
  screen!</template>
 <template ngSwitchDefault>If you are in the UK, it could do
  anything!</template>
</span>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Role and Structure of Components
- Angular Modules
- Component Metadata
- Component Templates
- Debugging Techniques
- Controlling Appearance
- **Repeating Data**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Displaying Repeating Data

- **`*ngFor` supports the display of repeating data**
  - Specify an array or other iterable collection to display
- **Define a loop variable name**

```
<tr *ngFor="let tempVar of collection">
```

- **Define a block of HTML populated by the loop variable**

```
<td>{{tempVar}}</td>
```

- **`*ngFor` will repeat the populated HTML once for each item in the collection**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Repeating Data With *NgFor

Template

```
<ul>
  <li *ngFor="let stat of stats">{{stat.date}} -
       {{stat.temperature}}        {{stat.tempUnits}}</li>
</ul>
```

TypeScript

```
export class WeatherStatsComponent {

  stats: Stats[] = [{
    'date': "July 14",
    'temperature': 23,
    'tempUnits': 'C'
  }, {
    'date': "July 15",
    'temperature': 13,
    'tempUnits': 'C'
  }…];
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

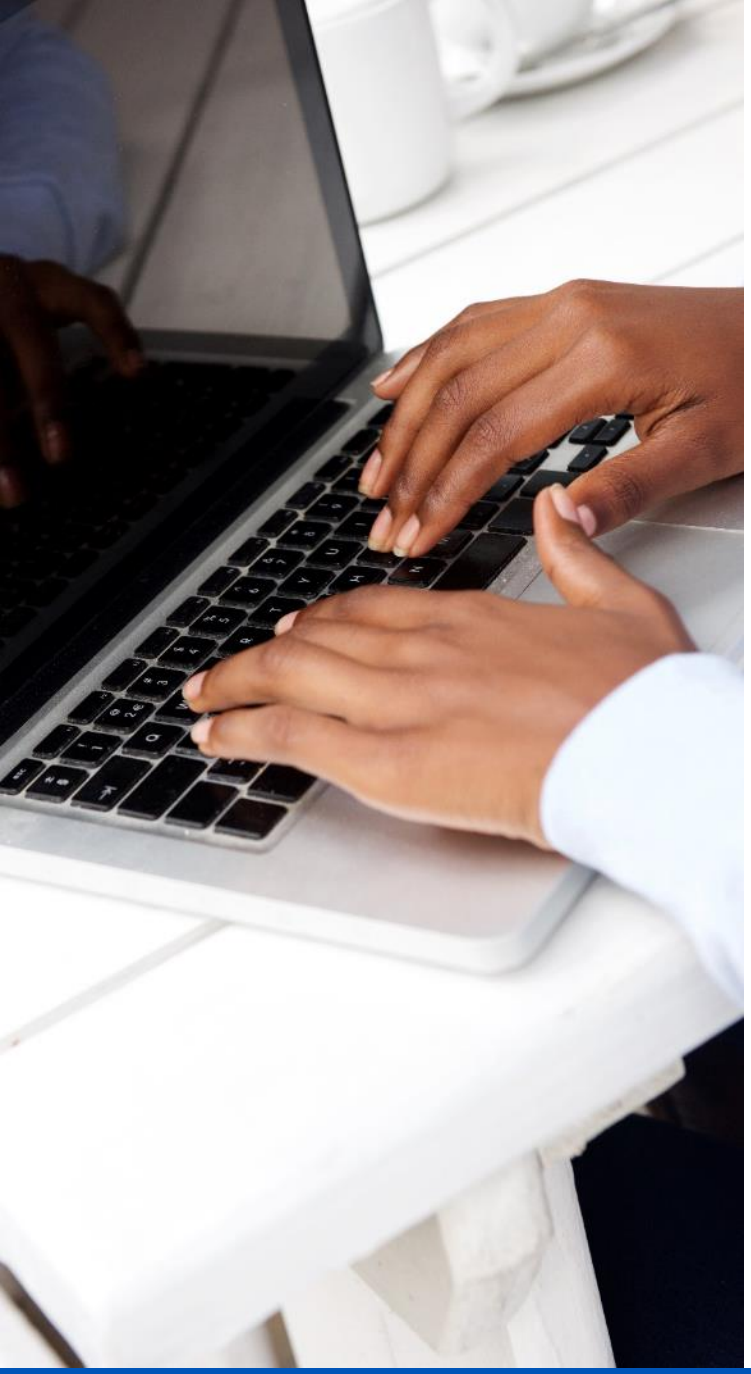
Loop Index Values

- **The keyword `index` can be used to populate a variable**
  - Providing a count of iterations

```
<li *ngFor="let tempVar of collection; let i = index">
       Item {{i}} – {{tempVar}}
</li>
```

- **Other `ngFor` keywords**
  - All are Boolean values
  - `first` true if the item is the first one
  - `last` true if the item is the last
  - `even` true if the index is even
  - `odd` true if the index is odd

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 2.2

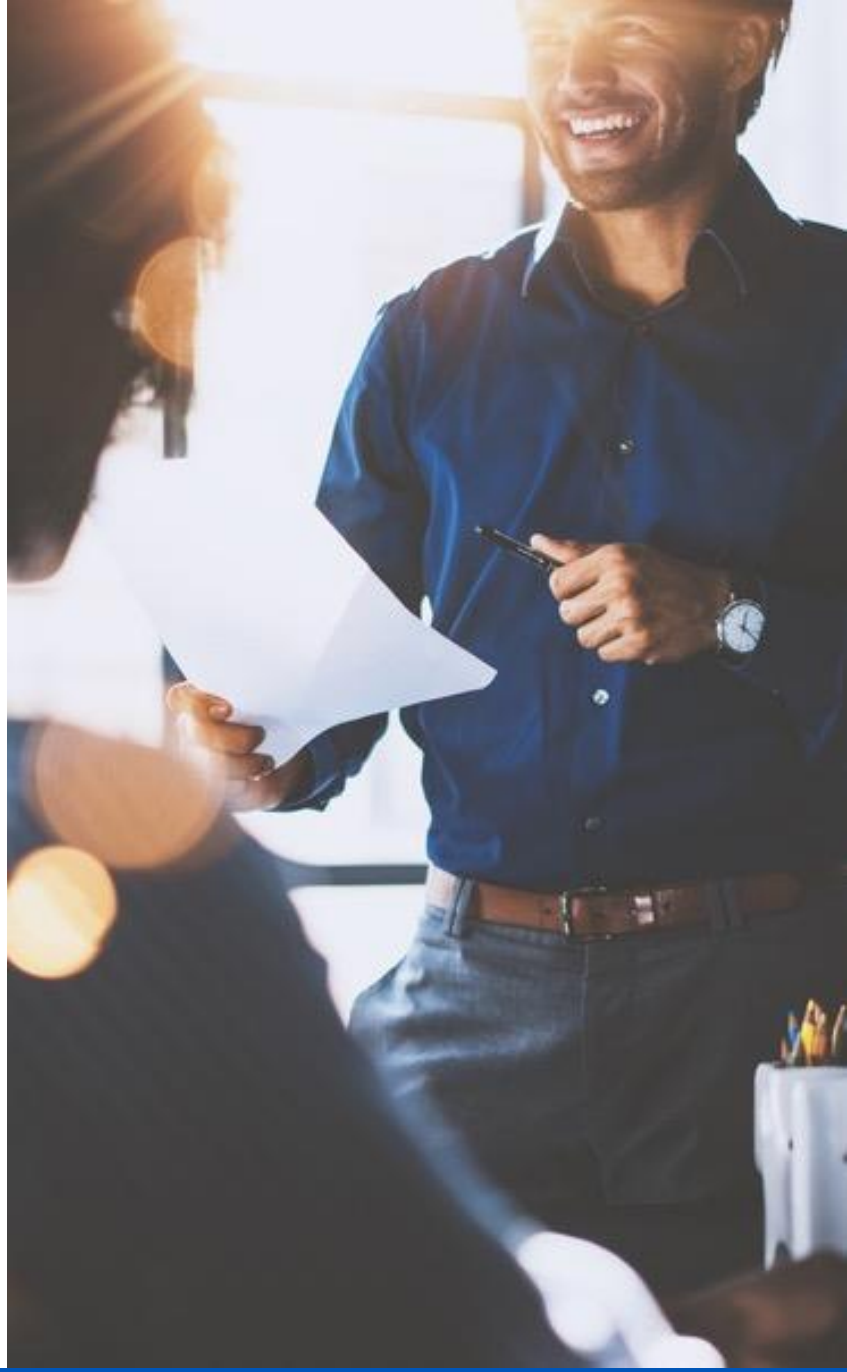
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 2.2: Displaying Repeating Data

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Create Component definitions
- Specify behavior and relationships with metadata
- Control visual appearance with templates
- Bind data to templates
- Generate repeating and conditional templates structures



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

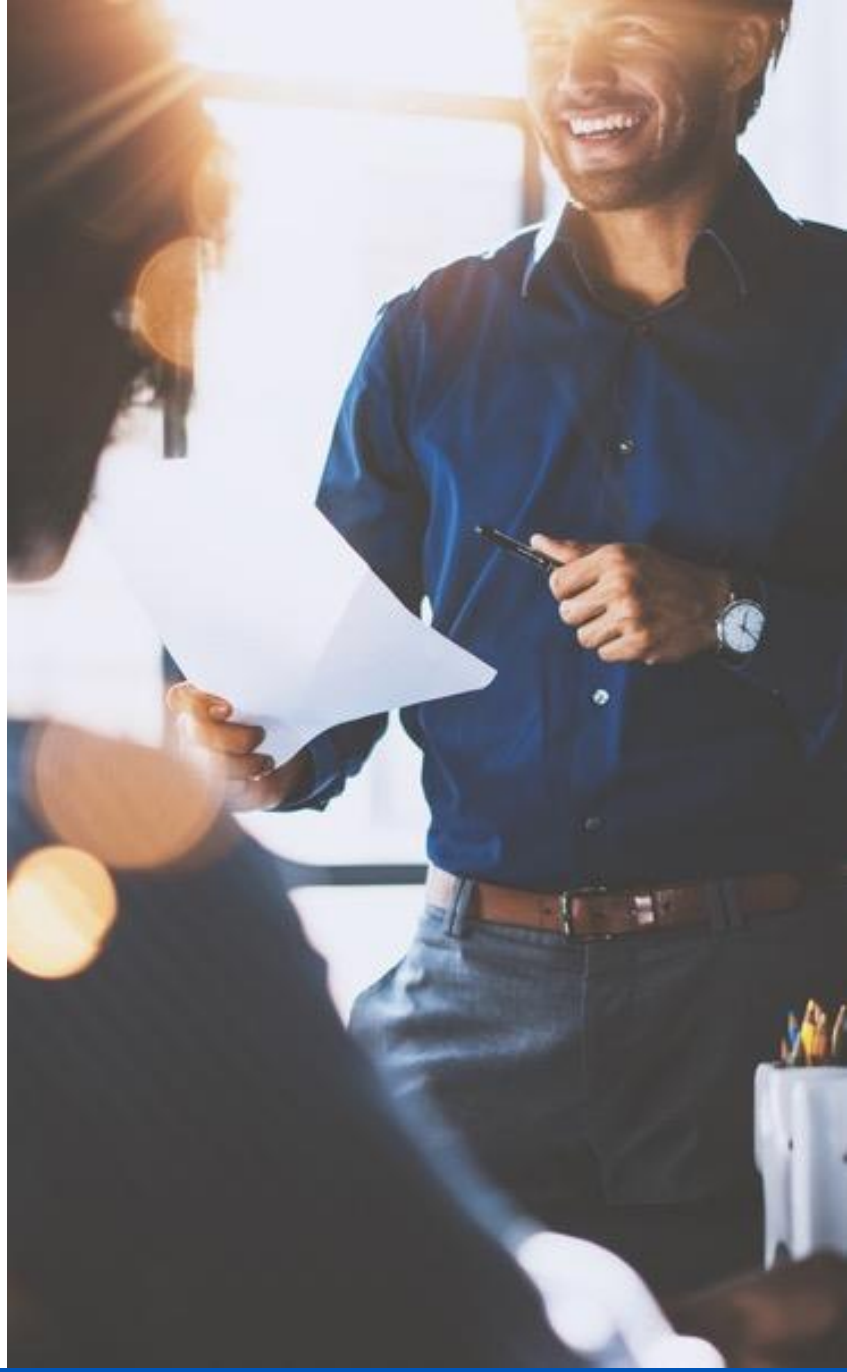
Chapter 3

Dependency Injection

LEARNING TREE INTERNATIONAL

Objectives

- Learn how Angular implements dependency injection
- Learn how to interact with the Angular Component lifecycle



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Dependency Injection

- Lifecycle Events

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

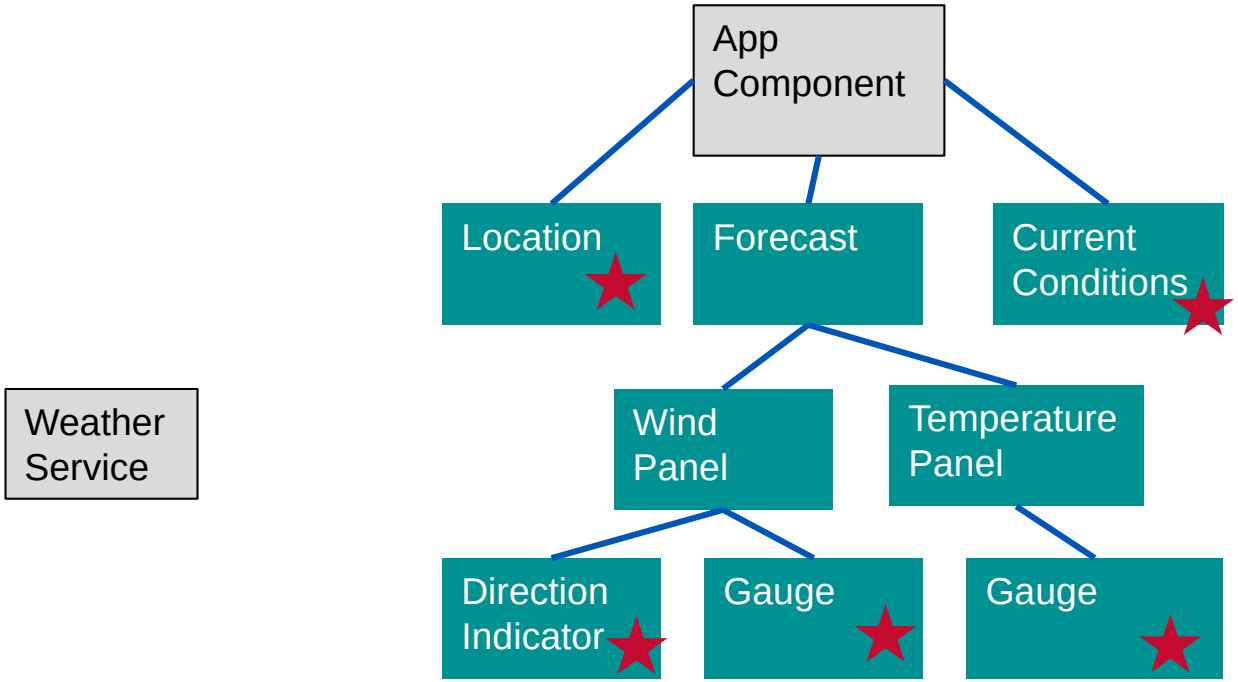
Dependency Injection

- **The need for injection**
  - In Angular, the view is composed of Components
    - In turn composed of Components
    - Angular handles the wiring of these Components for us
  - Angular Components often depend on other data
    - HTTP access
    - Service objects—for sharing model information
- **Angular provides a mechanism to provided resources to Components**
  - Dependency injection

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Resource Sharing With Dependency Injection

- Consider our Component hierarchy for the weather app
- Data is provided by the WeatherService
- Consumed by all the Components with the ★
- How to share the WeatherService?



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

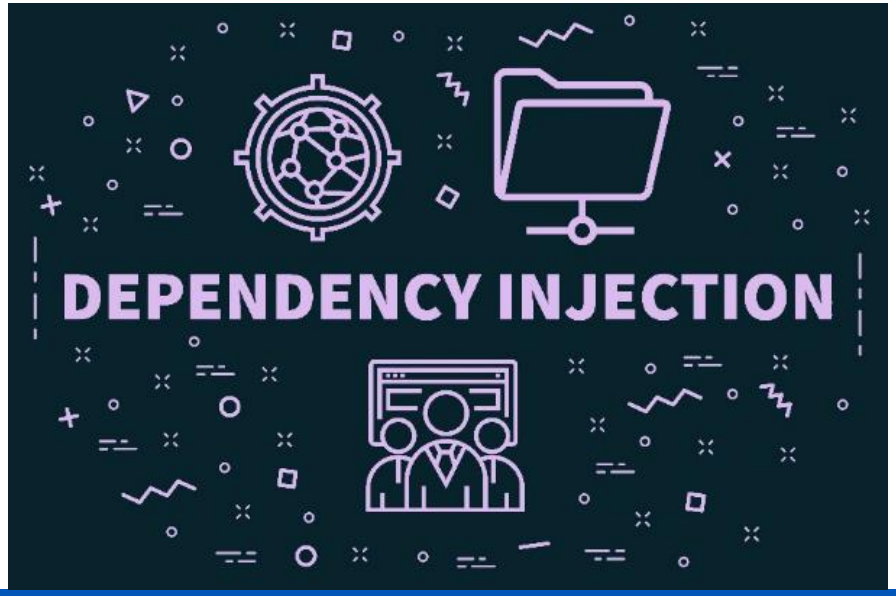
Sharing Resources

- **Option 1: Each consuming Component creates a separate instance of the service**
  - Hard to share data this way
    - Each instance is completely separate
    - May have serious performance impacts
- **Option 2: Create the service at the top level and pass it to each Component as we create it**
  - Problem: In Angular, we do not directly create the Components
    - Angular does!
- **Option 3: Provide a mechanism to tell Angular that the Components need to access the service**
  - Angular then injects references to the service
    - All Components share a single instance of the service
  - Dependency injection

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Implementing Dependency Injection

1. Define an `Injectable` class
2. Declare `providers`
3. Specify injection points on consuming Components



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

1. Define an `Injectable` Class

- **Creating classes for injection is simple**
  - Classes must be decorated with `@Injectable()`

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

@Injectable()
export class WeatherService {

  constructor() {}

}
```

Don't forget the ()

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

2. Declare `Providers` in a Module

- **Dependency injection requires a provider of objects to inject**
  - Often described as a factory
  - Creates instances of the classes, and make them available to inject
- **Providers may be declared at the Module level**
  - Makes Components available to any Component within the module
- **Add a `providers` property to the `@NgModule` declaration**

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent, HomeComponent, BuyFlightComponent
  ],
  imports: [
    BrowserModule,
  ],
  providers: [WeatherService],
  bootstrap: [AppComponent]
})
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Provider` Declaration via Decorators

- **The `@Injectable()`decorator may also be used to specify a provider**

```
@Injectable({
    providedIn: 'root'
})
```

- **The `providedIn` property specifies which module the injector should use as the container of the provided injectable**
- **Possible values:**
  - 'root'—the root module
  - a type—the class of a module
  - null—default indicates it is provided explicitly

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

3. Declare `Providers` in a Component

- **Components may also act as a providers**
  - May even be a provider for themselves
- **Add a `providers` property to the `@Component` declaration**
  - An array of `Class` names

app.component.ts

```
@Component({
  …
  providers: [WeatherService],
  …
})
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

4. Specify Injection Points on Consuming Components

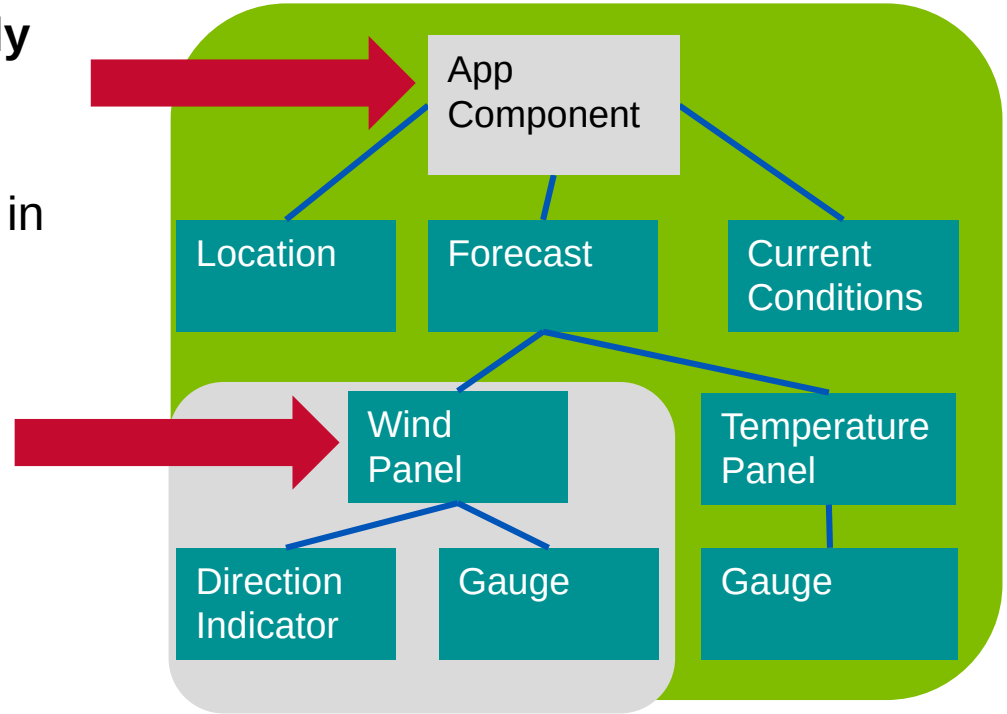
- **Constructor declarations define injection points**
  - Angular will try to satisfy any arguments specified in the constructor
    - From the available `providers`
- **If a provider matching the argument cannot be found**
  - The class will not be constructed
    - The code will fail!
  - Unless the argument is marked `@Optional()`
- **`private` and `public` arguments to constructors become fields of the class**
  - Default arguments are only available within the constructor

```
export class WindComponent {
 …
 constructor(private weatherService : WeatherService) {}
 …
 ngOnInit() {
    this.windSpeed = this.weatherService.getWindSpeed();
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Injection Hierarchy

- **Every Component inherits all injectables from its parent hierarchy**
  - Typically, the module
- **Injectables created as providers in a `Component`**
  - May be injected into the `Component` that declares them
  - Or into any child of that `Component`
- **`A provider` class should normally only be specified once within a hierarchy**
  - Multiple declarations may result in multiple instances
    - Not normally the desired behavior



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Dependency Injection

Lifecycle Events

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Lifecycle

- **Angular Components and Directives have a clearly defined lifecycle**
  - Creation
  - Rendering
  - Creating and rendering children
  - Checking for changes to data-bound properties
  - Removing the Component from the DOM
- **Lifecycle hooks are methods added to Components and Directives**
  - Invoked when lifecycle events occur

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lifecycle Example—`OnInit`

- **Put initialization code in the `ngOnInit()` method**
  - Databound properties are all initialized before this method is called
- **`ngOnInit()`is defined by the `OnInit` interface**
- **Declaring that the class `implements OnInit` is optional**
  - Good practice—reduces the risk of code errors not being detected

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

export class WindComponent implements OnInit {
  windSpeed : number;
  constructor(private weatherService : WeatherService) {}

  ngOnInit() {
    this.windSpeed = this.weatherService.getWindSpeed();
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Common Lifecycle Hooks

- **Hooks available to both Components and Directives**
  - `ngOnInit()` defined by `OnInit`
    - Called after Angular has initialized data-bound properties
    - Commonly used for Component initialization
  - `ngOnChanges()` defined by `OnChanges`
    - Called after Angular updates a data-bound property
    - Receives details of the modified property as a `SimpleChange` object
      - Contains details of current and previous values
  - `ngDoCheck()` defined by `DoCheck`
    - Called on every change detection run
      - Very often
    - Allows custom code to check for changes that Angular does not detect
  - `ngOnDestroy()` defined by `OnDestroy`
    - Cleanup before destruction of Component or Directive
      - Used to detach observables and event handlers

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

View Lifecycle Hooks

Reference

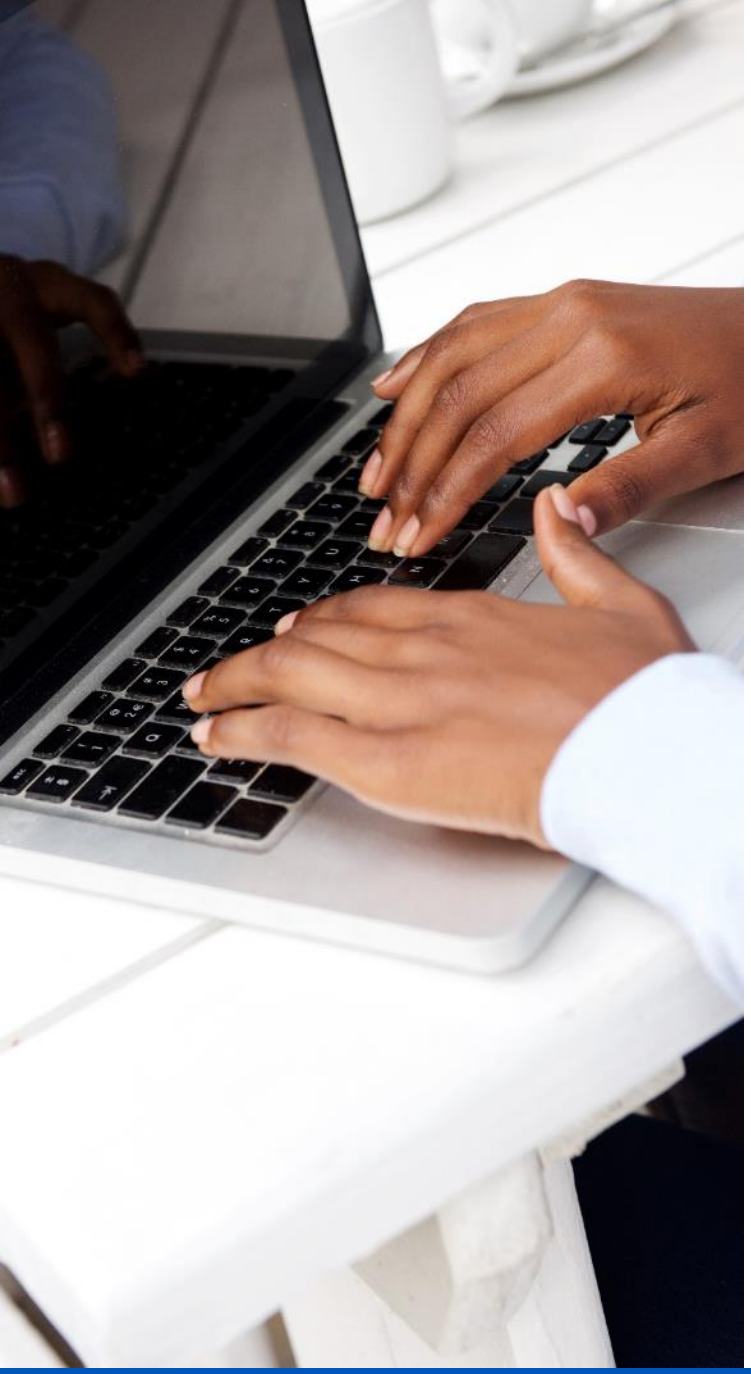
- **For Components only**
  - `ngAfterViewInit` defined by `AfterViewInit`
    - After Angular creates the Component's child view(s) and `ngOnInit()` completes on those views
  - `ngAfterViewChecked` defined by `AfterViewChecked`
    - After Angular checks the bindings of the Component's view(s)
- **Content Projection—external Component output is injected into the current Component at a designated location**
  - Has additional lifecycle methods
  - `ngAfterContentInit` defined by `AfterContentInit`
    - After Angular projects external content into its view
  - `ngAfterContentChecked` defined by `AfterContentChecked`
    - After Angular checks the bindings of the external content that it projected into its view

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lifecycle Event Sequence

- `ngOnChanges`
  - before `ngOnInit` and when a data-bound input property value changes
- `ngOnInit`
  - after the first `ngOnChanges`
- `ngDoCheck`
  - during every Angular change detection cycle
- `ngAfterContentInit`
  - after projecting content into the Component
- `ngAfterContentChecked`
  - after every check of projected Component content
- `ngAfterViewInit`
  - after initializing the Component's views and child views
- `ngAfterViewChecked`
  - after every check of the Component's views and child views
- `ngOnDestroy`
  - just before Angular destroys the Directive/Component

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 3.1

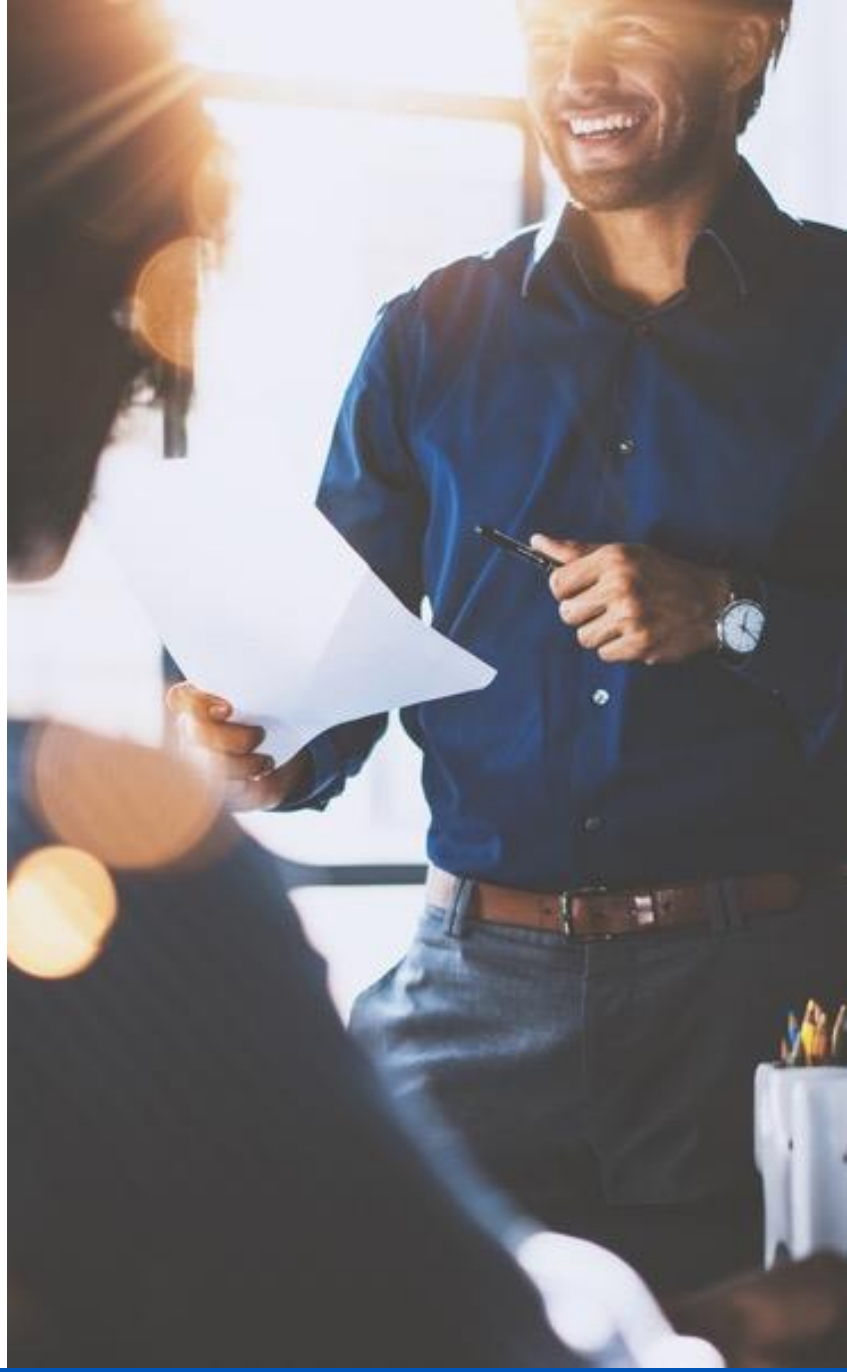
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 3.1: Injecting a Service

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Learn how Angular implements dependency injection
- Learn how to interact with the Angular Component lifecycle



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

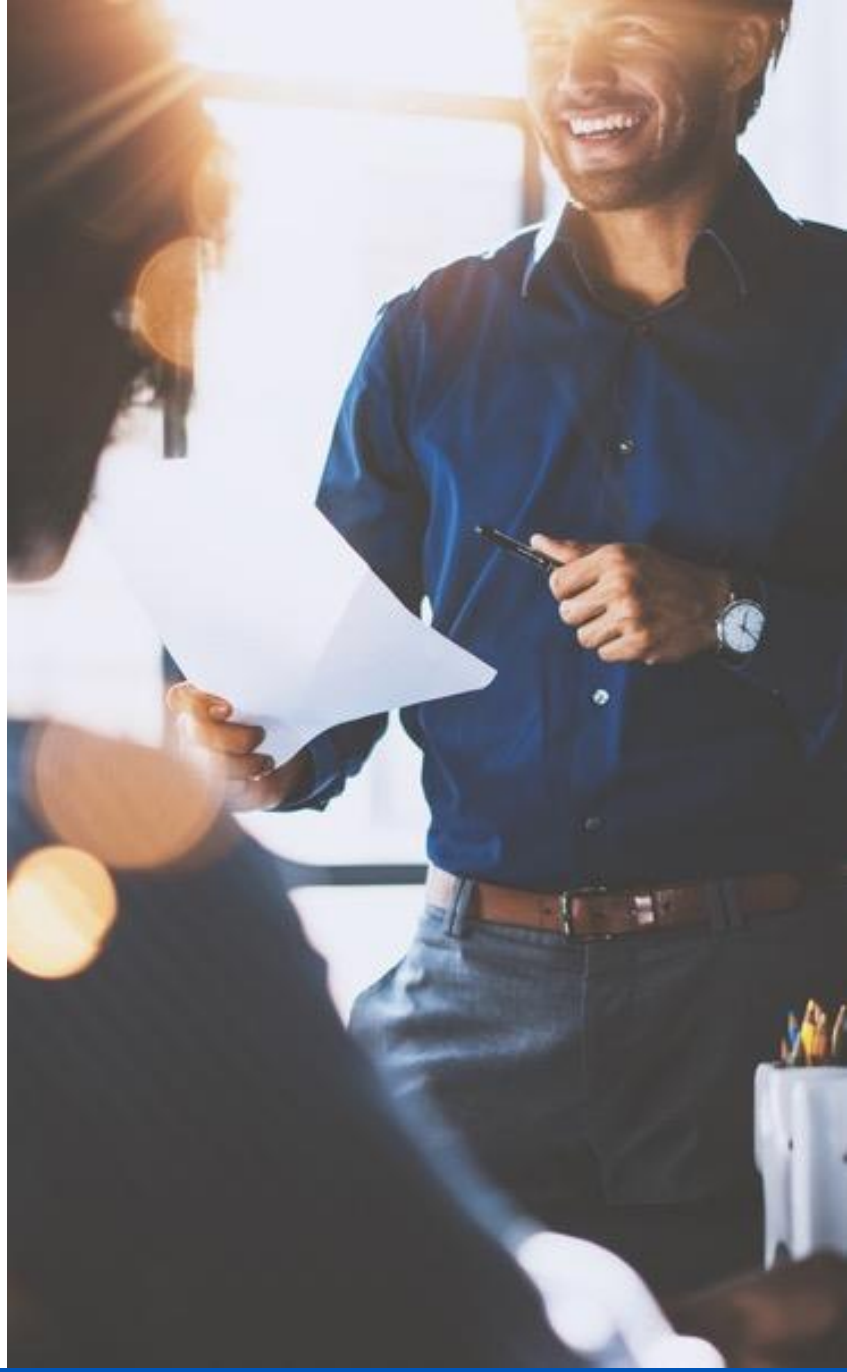
Chapter 4

Testing Angular

LEARNING TREE™ INTERNATIONAL

Objectives

- Test Angular Components
- Create isolated unit tests
- Perform end-to-end (E2E) testing



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Testing Strategies

- Arrow Functions
- Unit Testing
- E2E Testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing Granularity

- **Testing is essential**
- **A good philosophy is that if you have not tested it, then it does not work!**
- **User interface code is challenging to test**
  - Users do not always behave as we expect
  - Small changes to appearance can easily break tests
  - Checking DOM structures can be tedious
- **Testing options**
  1. Manual testing of complete application
  2. Automated testing of complete application
  3. Automated testing of each component in isolation
- **Option 1 may be acceptable for very small applications**
- **Generally need a combination of all three**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Unit Testing

- **Test functionality of each component in isolation**
- **Process**
  - Create an instance of the object under test
  - Apply external conditions to it
  - Check the results
- **Validate that it behaves as expected with normal input**
- **Validate that it handles error conditions correctly**
- **Unit tests should**
  - Test a single component
  - Run quickly
  - Provide good coverage of the component under test
- **Well-written unit tests improve developer productivity**
  - Much quicker to run a test than to simulate user input manually

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Integration and End-to-End Testing

- **Integration testing**
  - Verifies that components interact correctly
- **End-to-end testing validates the entire system**
  - Apply use cases to the application
    - Verify it behaves correctly
- **Integration and E2E tests, while important,**
  - Should be regarded as secondary to unit testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Testing Strategies

Arrow Functions

- Unit Testing
- E2E Testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Arrow Functions

- **Angular makes extensive use of asynchronous functionality**
  - Do something and call a function on completion
- **Both test code and code under test need to specify call-back functions**
- **Simplified by TypeScript (and ES2015) arrow functions**

```
(arg1, arg2) => { // arrow function body }
```

- **Arrow functions can be used anywhere an anonymous function can be used**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Example Arrow Functions

- **Arrow functions are an alternative syntax for anonymous function declarations**

TypeScript

```
this.http.get(url)
     .map().do((res) => console.log("Result" + res));
```

Older JavaScript

```
this.http.get(url)
     .map().do(function(res){
               console.log("Result" + res);
  }
);
```

- **Shorthand syntax if there is only one argument**

```
res => console.log("Result" + res);
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Testing Strategies
- Arrow Functions

Unit Testing

- E2E Testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Unit Testing With Jasmine

- **Jasmine is "a behavior-driven development framework for testing JavaScript code"***
- **Defines**
  - Suite of tests wrapped in a call to the Jasmine `describe()` function
    - Takes a string describing the suite and an arrow function
  - Individual *specs* (tests) defined by calling the Jasmine `it()` function
    - Takes a string describing the test and an arrow function which is the test
- **Angular CLI automatically creates Jasmine test specs for each component generated**
  - With a file extension of `spec.ts`
- **The generated tests are a starting point**

*Jasmine: Behavior-Driven JavaScript, http://jasmine.github.io

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Anatomy of a Jasmine Test Suite

Suite

CurrentWeatherComponent.spec.ts

```
import {CurrentWeatherComponent} from
        './forecast.component';

describe('Component: CurrentWeather', () => {

  it('should create an instance', () => {
    let component = new CurrentWeatherComponent ();
    expect(component).toBeTruthy();
  });

});
```

Name of Component under test

Spec—a test

This auto-generated spec passes if the `CurrentWeatherComponent` can be created

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Anatomy of an Individual Jasmine Spec

Simple description of the test

CurrentWeatherComponent.spec.ts

```
it('should create an instance', () => {
   let component = new CurrentWeatherComponent();
   expect(component).toBeTruthy();
 });
```

The expected behavior

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Test Dependencies

- **Problem: What if the component under test has dependencies?**
  - The constructor would need the dependencies supplied

```
let component = new CurrentWeatherComponent( dependency );
```

- **Dependencies must be provided by the test framework**
- **A `beforeEach( fn() )` function is used to initialize tests**
  - The provided function is executed before each spec is executed
- **`TestBed` is used to provide and initialize the dependencies**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating a Test Environment

- **The simple test on the previous slides has another flaw**
  - It only allows direct testing of the Component class
    - Not DOM interactions
- **The Angular class `TestBed` provides**
  - A configurable test module
    - To satisfy dependencies
  - A DOM framework
    - So that the Components HTML DOM output may be verified

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

TestBed

- **`TestBed` is used to create an environment for the component under test**
- **Created in two phases**
  - Each performed in a `beforeEach()`
- **Phase 1—Create the testing module**
  - `TestBed.configureTestingModule(module_metadata)`
  - `module_metadata` is a module definition based on `@NgModule`
  - The last phase of the module creation - `compileComponents()`
    - Must be performed asynchronously
      - Due to Jasmine constraints
- **Phase 2—Create a test *fixture***
  - Used to perform the tests
  - Typical tests use both the fixture and the component under test
  - Only executed when the first phase has completed

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

TestBed Configuration—Part 1

- **Compile a test module**

CurrentWeatherComponent.spec.ts

```
describe('CurrentWeatherComponent', () => {
  let component: CurrentWeatherComponent;
  let fixture: ComponentFixture<CurrentWeatherComponent>;


  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ CurrentWeatherComponent ],
       providers: [WeatherService]
    })
    .compileComponents();
  }));

  // Phase 2 goes here

});
```

Perform asynchronously

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

TestBed Configuration—Part 2

CurrentWeatherComponent.spec.ts

```
describe('CurrentWeatherComponent', () => {

  // Previous slide content here

  beforeEach(() => {
    fixture =  TestBed.createComponent(CurrentWeatherComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

});
```

`component` and `fixture` now ready to use in subsequent test specs

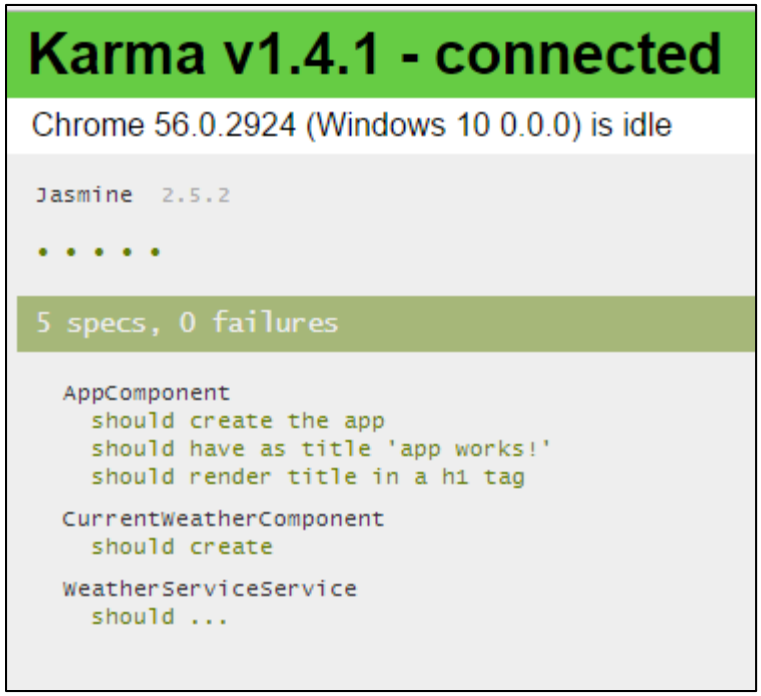
- **`fixture.detectChanges` runs a change detection cycle**
  - Syncs DOM with component properties

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Working Test With All Dependencies Satisfied

- With the `TestBed` configured, we can now test the Component

```
it('should create', () => {
    expect(component).toBeTruthy();
});
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Mocking

- **One small problem!**
  - If we inject a component, then our test is testing both the component under test and the injected component
- **Solution: Mocking**
  - Create a minimal object that satisfies the injection requirement
- **Mock object(s) should be defined with the test code**
  - Provide known values to support test expectations
- **The Mock class must be defined before the test**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Mock Weather Service

- **Mock weather service is the minimum code to satisfy the test**
  - Structure must match the object being mocked
- **A minimal mock for the `WeatherService`**

```
let mockWeatherService = {};
```

- **Mock code can evolve as the tests develop…**

```
class MockWeatherService {
  getWeather() : Observable<Weather[]>{
    return new Observable<Weather[]>();
  }
  savePreferences(preferences: Preferences ){}
  updatePreferences(preferences: Preferences ){}
  getWindSpeed(){}
  getForecast( location : string ){}
}
let mockWeatherService = new MockWeatherService
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Providing the Mock Object

- **The mock object must be added to the available providers, replacing the real object**
  - Real provider

```
providers:    [WeatherService]
```

  - Mock provider

Class being mocked

```
providers: [{provide: WeatherService,
             useValue: mockWeatherService }]
```

Mock to use

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Mocking Components

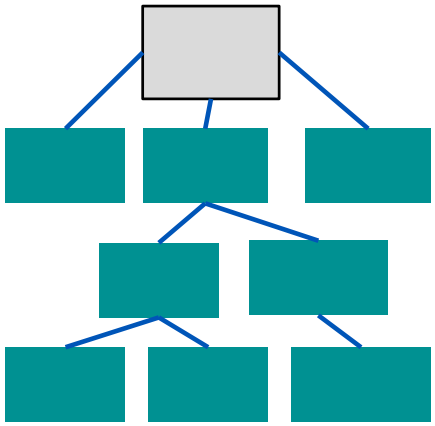
- **Components with child components**
  - Have elements in their templates referencing the child component

app-component.html

```
<app-buy-flight></app-buy-flight>
```

- **If the `TestBed` does not find components matching these selectors**
  - It will cause the test to fail

- **Solution: mock components**
  - Define a mock class
  - Decorate with `@Component` and a selector matching the component to be mocked
  - Use an empty template
    - Unless you *really* need to interact with the child template



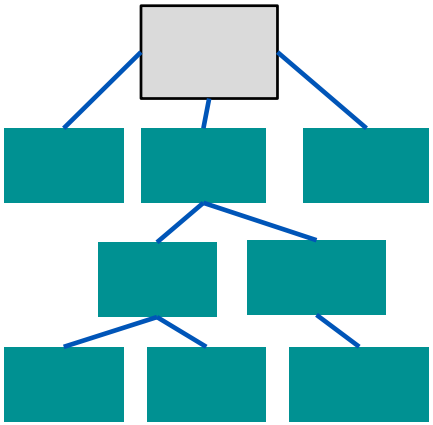
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Mocking Components—Example

- **Solution: mock components**

app-component.html

```
<app-buy-flight></app-buy-flight>
```



Empty template

Selector

```
@Component({
template: '',
selector: 'app-buy-flight'
})
class MockBuyFlightComponent {}

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
    imports: […],
    declarations:AppComponent, MockBuyFlightComponent]});
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

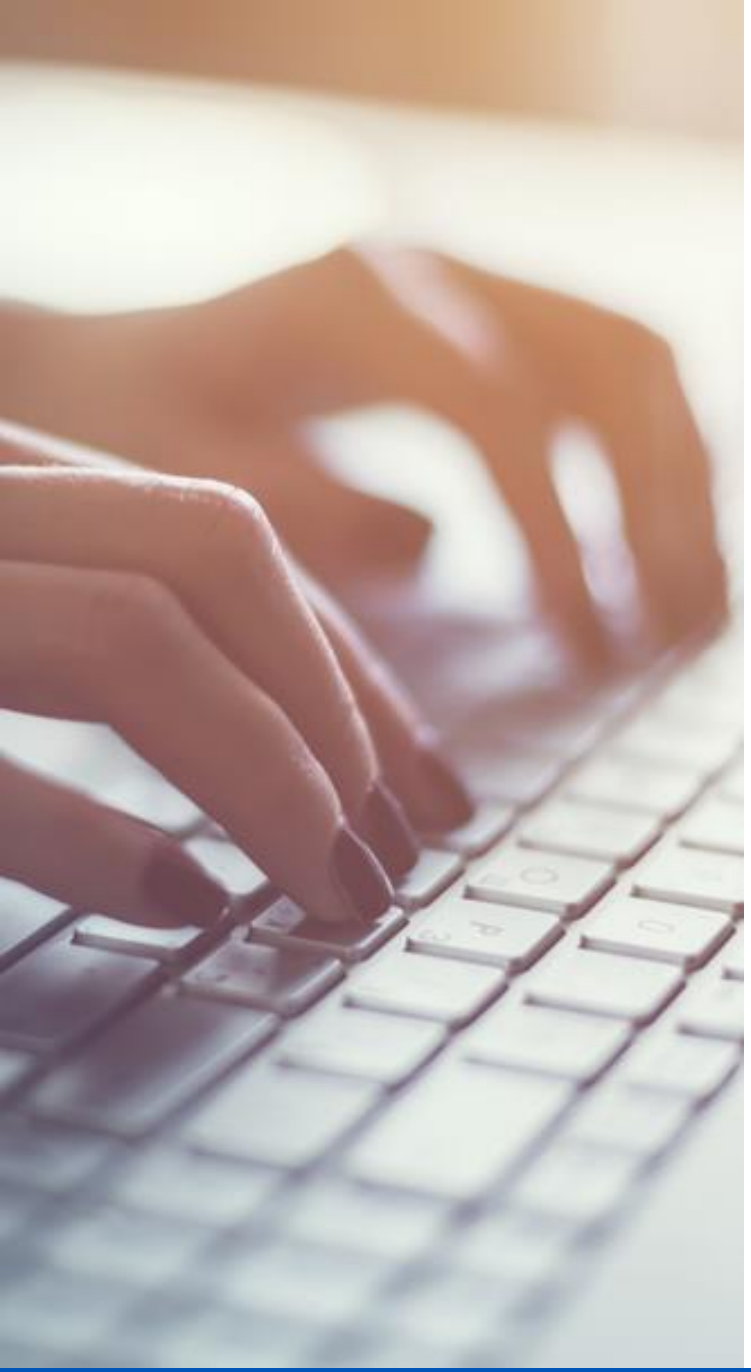
Running Jasmine Tests With Angular CLI

- **Angular CLI will automatically run all the Jasmine unit tests using Karma**
  - Simply run `ng test` from the command line
    - Automatically rebuilds and tests each time a file is saved
  - A browser running Karma is launched to host the tests

```
Karma v1.4.1 - connected
Chrome 56.0.2924 (Windows 10 0.0.0) is idle

Jasmine 2.5.2

• x x • •

5 specs, 2 failures

Spec List | Failures

AppComponent should have as title 'app works!'

Expected 'app does not work!' to equal 'app works!'.

Error: Expected 'app does not work!' to equal 'app works!'.
```

```
Karma v1.4.1 - connected
Chrome 56.0.2924 (Windows 10 0.0.0) is idle

Jasmine 2.5.2

• • • • •

5 specs, 0 failures

AppComponent
  should create the app
  should have as title 'app works!'
  should render title in a h1 tag

CurrentWeatherComponent
  should create

WeatherServiceService
  should ...
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



4.1: Execute a Test

Do Now

- In your Exercise Manual, please refer to Do Now 4.1: Execute a Test

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing a Component—Code Under Test

- **Example Component has a button which, when clicked, is disabled until the other button is clicked**

Update Weather | Enable Other Button

```
<button [disabled]="buttonDisabled"
       (click)="onClick()">Update Weather
</button>
<button [disabled]="!buttonDisabled"
               (click)="onClickOther()">Enable Other Button
</button>
```

```
buttonDisabled : boolean = false;

onClick(){ this.buttonDisabled = true;}

onClickOther(){ this.buttonDisabled = false;}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing a `Component` Directly—Test Spec

- **The spec**
  - Calls the click method
  - Checks the state of the `buttonDisabled` property

```
it('the button should be enabled after onClickOther()',
   ( ) => {

     component.buttonDisabled = true;
     component.onClickOther();
     expect(component.buttonDisabled).toBeFalsy();

   });
```

`component` was initialized in `beforeEach()`

Direct access to methods and properties of the `component` under test

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing a `Component` via DOM Interaction

- **The spec**
  1. Locates the button to click via a CSS lookup
  2. Triggers a click event
  3. Locates the newly disabled button
  4. Checks the state of the `disabled` property

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Locating Controls

- **Interaction via the HTML DOM requires location of HTML controls**
  - Typically via CSS selectors
- **Consider using `id` attribute to create unique control locators**
  - Makes CSS selectors simpler and more robust

```
<button [disabled]="buttonDisabled"  id="updateButton"
       (click)="onClick()">Update Weather
</button>
<button [disabled]="!buttonDisabled" id="otherButton"
       (click)="onClickOther()">Enable Other      Button
</button>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing a Component via DOM Interaction

```
it('the update button should be enabled after clicking the other
  button',( ) => {

  let buttonEle =
    fixture.debugElement.query(By.css('#otherButton'));

  buttonEle.triggerEventHandler('click', null);

  let updateButtonEle =
    fixture.debugElement.query(By.css('#updateButton'));

    expect(updateButtonEle.nativeElement.disabled)
          .toBeFalsy();
  });
```

1

2

3

4

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing Services via Dependency Injection

- **Services can be tested very simply using Jasmine**
  - Simply create the object and verify behavior
- **Angular CLI creates more flexible service tests by default**
  - Use the `TestBed` to make dependencies available

```
describe(WeatherService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
    providers: ['WeatherService]
  });
});

it('should be created',
  inject([WeatherService], (service: WeatherService) => {

  expect(service).toBeTruthy();
}));
```

Configure `TestBed` with service to test as a provider

`inject` function makes the service available to the arrow function

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Test Recommendations

- **Create one test suite per component**
- **Each spec within the suite should**
  - Have a clear description
  - Be fine-grained
    - Test only one thing
- **Specs should build on each other**
  - Test simple things first
  - Gradually test more complex issues

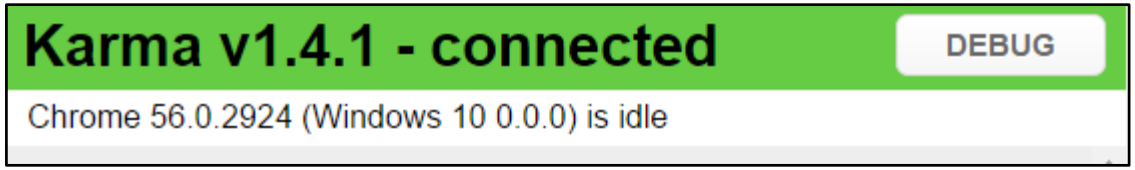
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Debugging With Karma

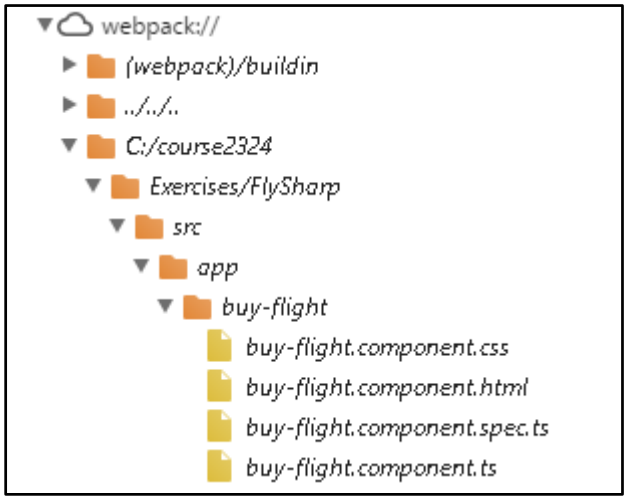
Demo

To debug a test running under Karma:

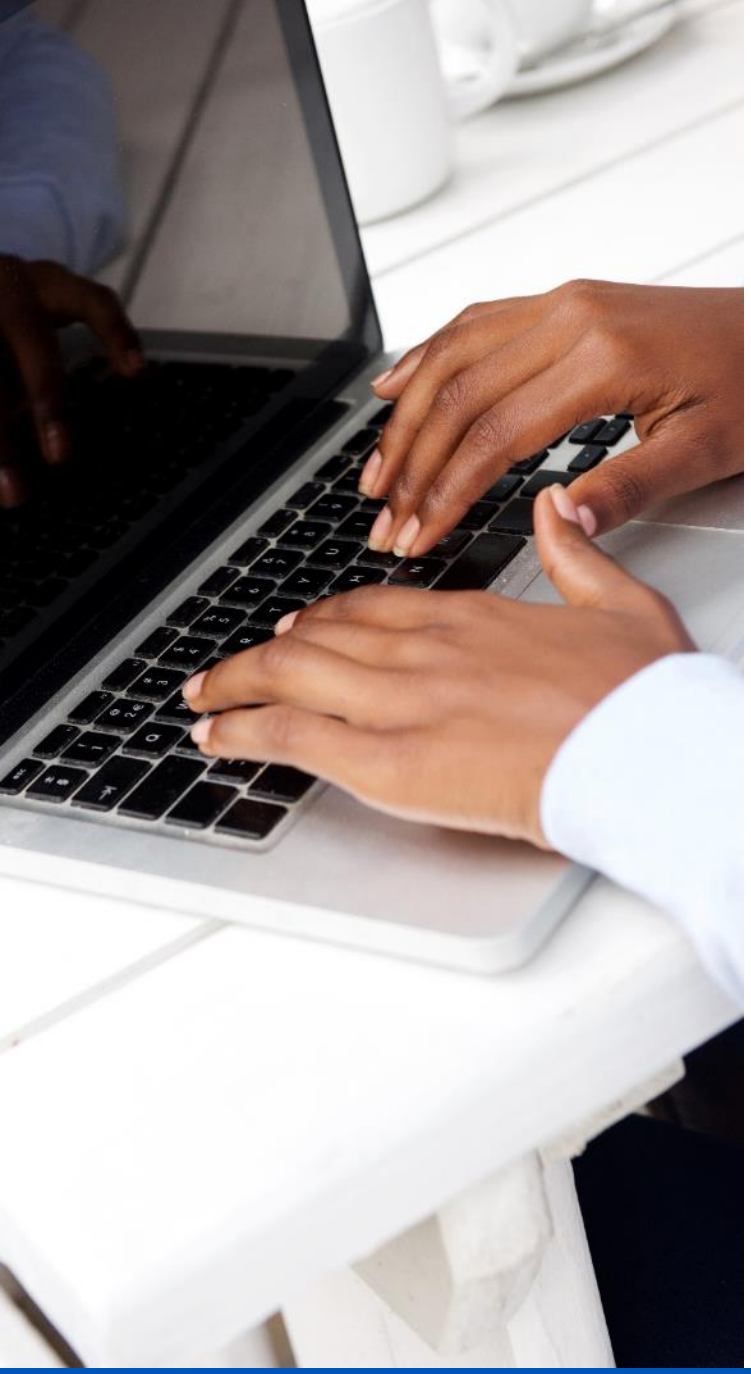
1. **Click the DEBUG button in Chrome**



2. **In the new page, open Web Developer Tools**
3. **Select the Sources tab**
4. **Browse to your test code**
5. **Set a break-point in your code**
6. **Refresh the browser to run your code**
7. **Close all Chrome instances when you are done**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 4.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 4.1: Unit Testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Testing Strategies
- Arrow Functions
- Unit Testing

E2E Testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

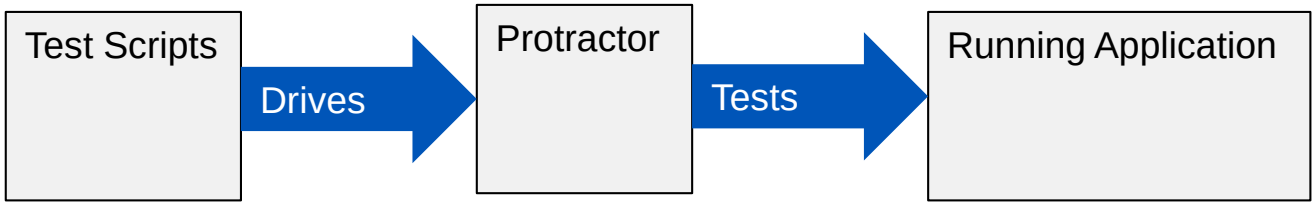
End-to-End (E2E) Testing

- **Testing the complete application**
  - Apply use cases and check results
- **Angular can be tested with Protractor**
  - Automates Jasmine tests on the application running in the browser
  - Caution: Protractor was created to test Angular JS 1 applications
    - Has many Angular JS–aware functions
      - Which may not work with Angular
    - Expect an Angular version soon
- **E2E testing is a powerful mechanism for checking that previously tested components work together**
- **Angular CLI generates a framework for E2E testing when a new project is created**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing With Protractor

- **Protractor**
  - Driven by test scripts
    - `*.e2e-spec.ts`—the overall test script
      - Uses Jasmine syntax
    - `*.po.ts`—page objects
      - Code to interact with the HTML DOM
- **Instructions in the scripts simulate user input**
  - Protractor applies those instructions to the DOM
    - Fetches modified state from the DOM
    - Validates resultant state



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Protractor E2E Test

- **Jasmine driver script**
  - Uses `describe`, `beforeEach`, and `it` as we have seen

app.e2e-spec.ts

```
import { MyAppPage } from './app.po';

describe('fly-sharp App', function() {
  let page: FlySharpPage;

  beforeEach(() => {
    page = new FlySharpPage();
  });

  it('should display message saying app works', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('my-app works!');
  });
});
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Protractor E2E Test

- **Page accessor class provides user input simulation and results accessor**
- **The `by.css()` selector uses CSS selectors to specify data**
  - Works well with Angular

app.po.ts

```
export class FlySharpPage{
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    return element(by.css('my-app-app h1')).getText();
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing DOM Behavior

- **Selecting multiple items**

```
return (element.all(by.css('ul li')));
```

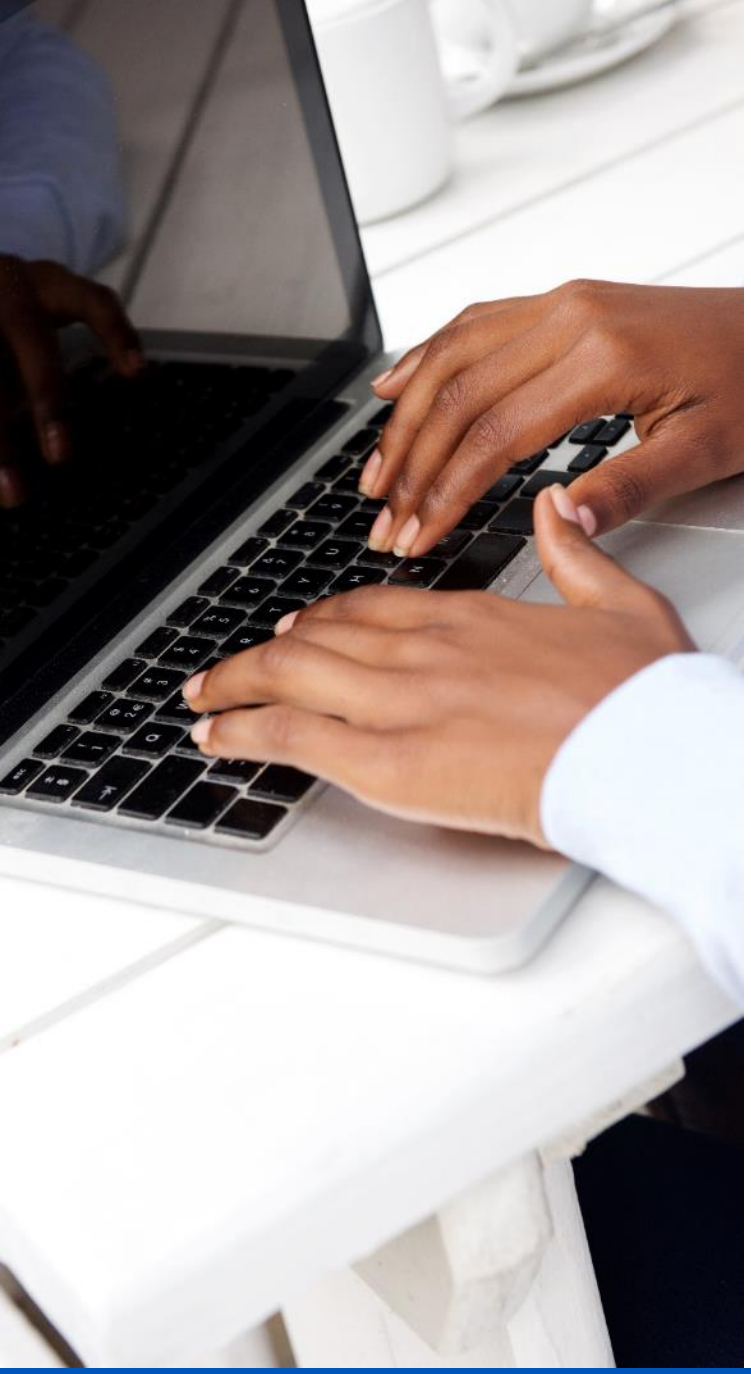
- **Counting items**

```
return (element.all(by.css('ul li'))).count();
```

- **Clicking elements**

```
clickForecastLink() {
   element(by.css('#forecast')).click();
 }
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 4.2

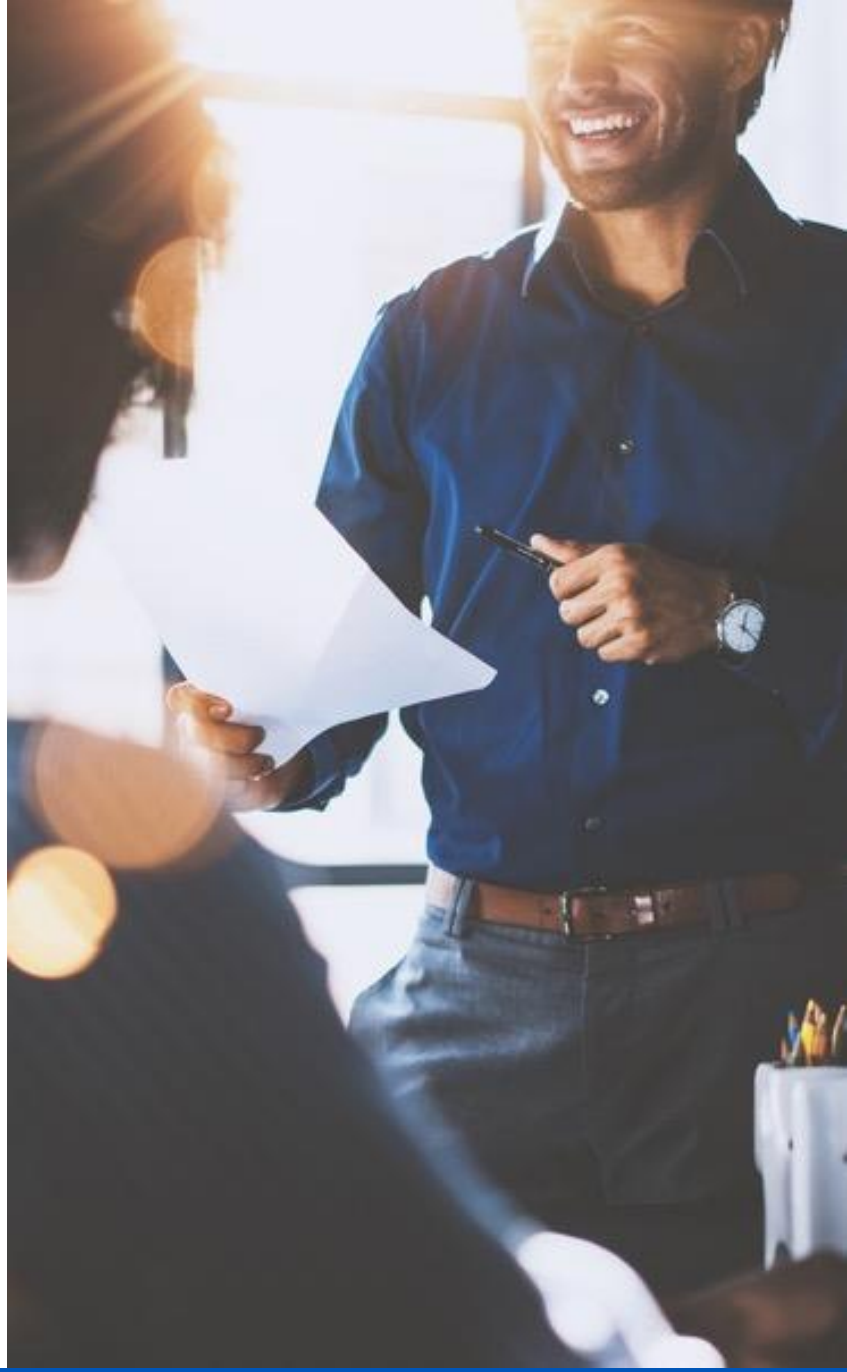
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 4.2: End-to-End Testing

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Test Angular Components
- Create isolated unit tests
- Perform end-to-end (E2E) testing



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

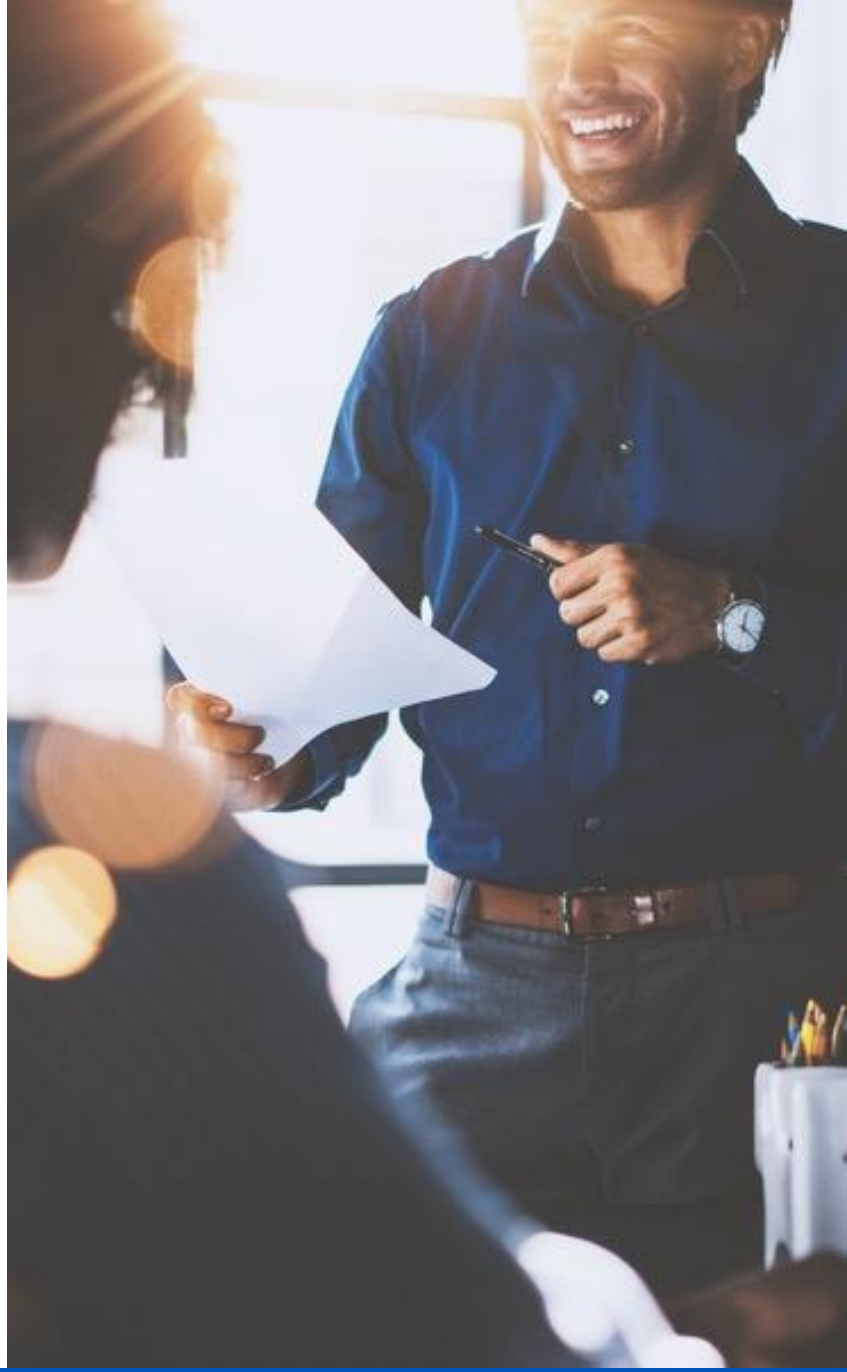
Chapter 5

Component Interaction

LEARNING TREE INTERNATIONAL

Objectives

- Exchange information between Components
- Display data from parent Components
- Define event emitters and handlers
- Share information with service Components



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

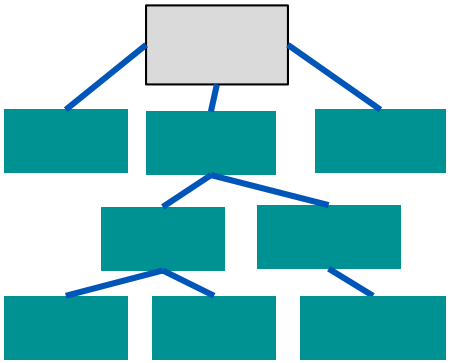
Component Interactions

- Property Bindings
- Events
- Direct Communication

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Interacting Components

- **Angular Components each have**
  - Their own screen area
  - Their own data
- **They may have**
  - Parents
  - Children
- **For the application to function**
  - Information must be communicated between Components



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Component Interactions

- **Example interactions**
  - Buttons on parent Component enabled only when fields are filled on child
  - Data in child Component modified by selection in a parent
  - Events broadcast by a Component
  - Events triggered by user actions
- **Angular techniques for handling interactions**
  - Input bindings
    - Intercepting input changes with a setter
    - Intercepting input changes with a lifecycle event
  - Local template variables
  - Event handling
  - Event generation
  - `ViewChild`
  - Shared Components

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Component Interactions

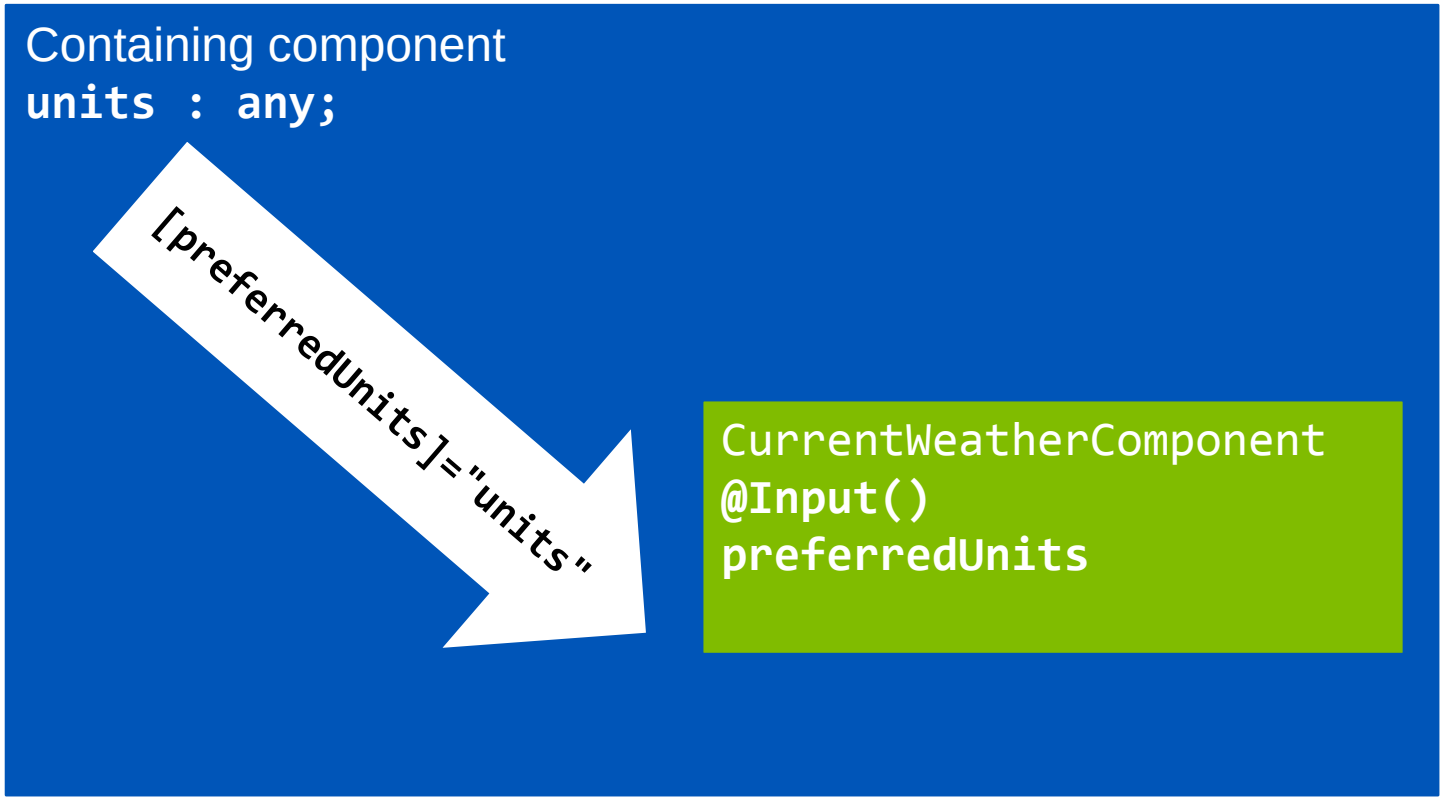
Property Bindings

- Events
- Direct Communication

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Input via a Property Binding

- **In our example**
  - Changes to `units` on the containing (parent) Component
    - Will automatically be copied to `preferredUnits` on the child Component



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Input via a Property Binding

- **Property bindings—`[target]="expression"`**
- **Can set properties on both the DOM and Components**
  - Example sets the `preferredUnits` property of the `app-current-weather` Component to the value of the units property on its containing parent

```
<app-current-weather [preferredUnits]="units"></app-current-
  weather>
```

- **The field on the class of the Component must be decorated with `@Input()`**

```
export class CurrentWeatherComponent implements OnInit {

    @Input()
    preferredUnits : string;
…
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Intercepting Changes to Bound Properties

- **Logic in a Component may need to be notified of a change to a bound property**
  - For example, to trigger loading of resources or change button state
- **Two mechanisms available**
  - Providing a property setter
    - One method per property to be checked
  - Overriding the `ngOnChanges()` lifecycle method
    - Single method for all property changes

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Property Setter

- **TypeScript setter syntax**
  - `set propertyName( prop : type)`
  - NB: Variable used to store property must have a different name to the method

```
set preferedUnits(preferredUnits : string) {
    this._preferredUnits = preferredUnits;
    console.log("New value is: " + preferredUnits );
    doStuff();
}
```

- **Getter syntax**
  - Can have getter or setter or both

```
get preferredUnits() : string {
    return this._preferredUnits;
}
```

- **The `@Input()` decorator must be placed immediately before the setter**

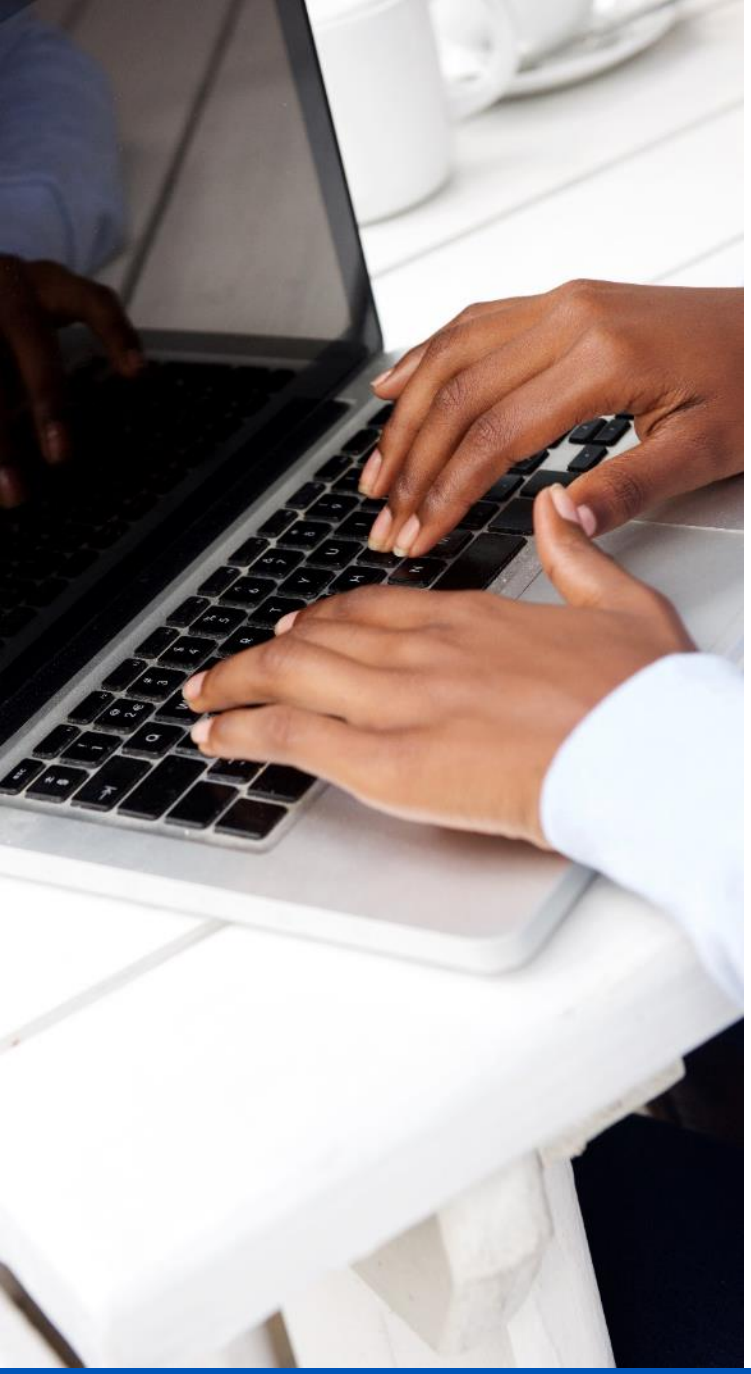
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Override ngOnChanges()

- **`ngOnChanges()` is called every time a bound property is modified**
- **Input parameter is an array of `SimpleChange` objects**
  - Each providing the `currentValue` and the `previousValue`

```
ngOnChanges(
        changes:{[propertyName: string]: SimpleChange }) {

  if (changes['preferredUnits'] !== undefined) {
        let value =
                changes['preferredUnits'].currentValue;
        console.log("New value is: " + value );
       doStuff();
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 5.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 5.1: Implementing Input Bindings

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Component Interactions
- Property Bindings

Events

- Direct Communication

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Emitting Events

- **Components may emit events**
- **Events are created by an `EventEmitter` field**
  - Must be decorated with `@Output()`
  - The emitted event will have the same name as the variable

```
export class CurrentWeatherComponent {

 @Output()
 weatherUpdated : EventEmitter<string>;
  …
   constructor() {
    this.weatherUpdated = new EventEmitter<string>();
  }
  …
  onClick(){
     this.weatherUpdated.emit(this.currentCity);
  }
```

The data type of the event

`emit()`triggers the event

The payload of the event

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Consuming Events

- **To consume an event from a Component**
  - Declare an event binding in the template
    - Binds the event emitter to a handler method
  - Use the `$event` keyword to pass the event to the handler

app.component.html

```
<app-current-weather
 (weatherUpdated)="onWeatherUpdated($event)">
</app-current-weather>
```

Name of event emitter

Method to invoke when event occurs

`$event` keyword passes event to the handler

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Consuming Events—Event Handler

- **The event handler is a method on the Component class**
  - Accepts a single parameter
    - The event data

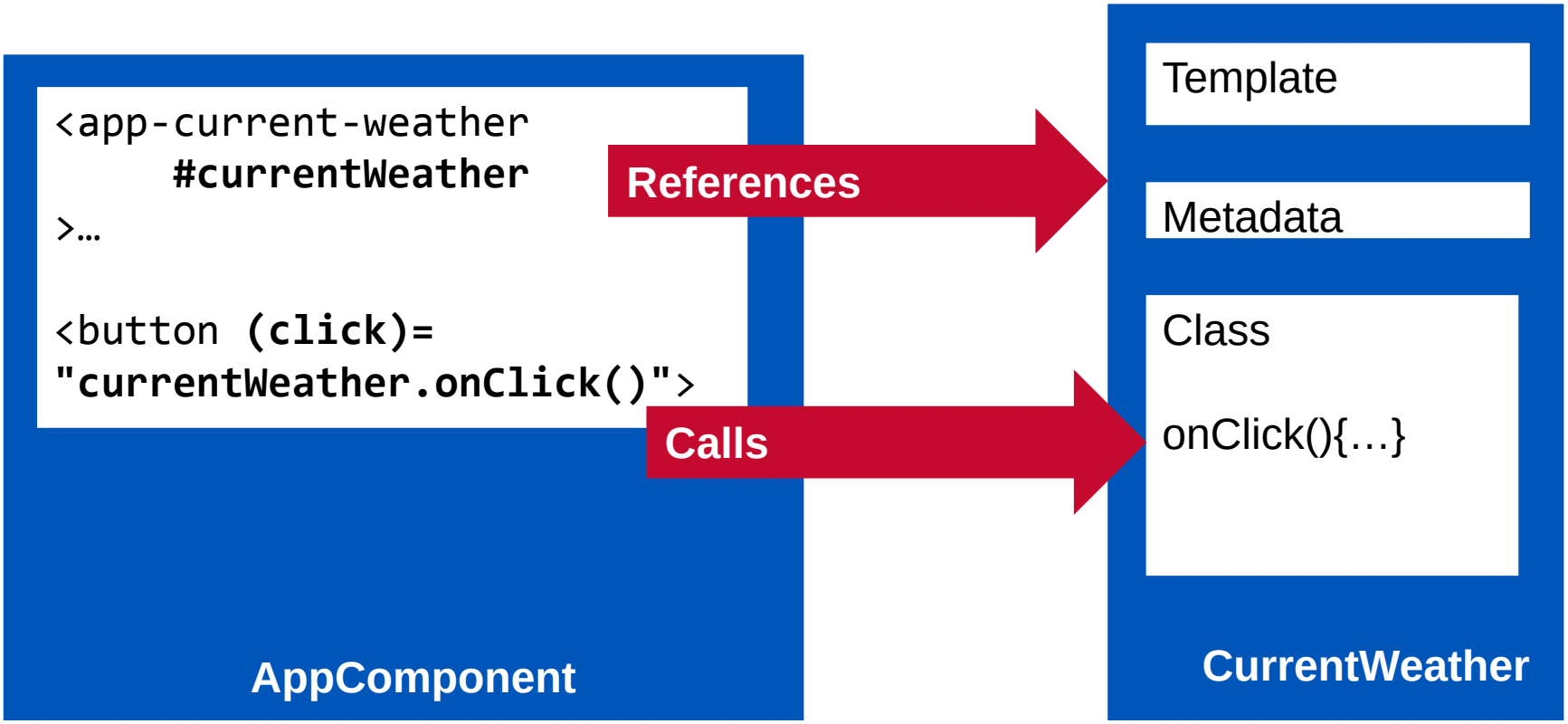
app.component.ts

```
onWeatherUpdated( city : string ){
   console.log("City is : " + city);
 }
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Reference Variables

- **Allow access from the template of one Component**
  - To the properties and methods of another



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Reference Variables

- **A mechanism to allow a parent Component to access methods and properties on a child Component**
  - Only works within templates
    - Not the Components class

1. **Define a reference variable for a Component**
   - The **#** character indicates that a reference variable is being declared
   - Variable becomes a reference to the Component in whose start tag it is declared

2. **Use that variable to access the Component elsewhere in the template**

```
<app-current-weather #currentWeather …></app-current-weather>

<app-weather-stats></app-weather-stats>

<button (click)="currentWeather.onClick()">Update Child
  Weather</button>
```

1

2

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Pseudo Key Events

- **DOM key events may all be bound**
  - `keydown`, `keypress`, `keyup`
- **Pseudo events allow event handlers to be bound to a single key**
- **Example: Update the user field of a Component with the value of the `<input>` control**
  - When the `<Enter>` key is pressed

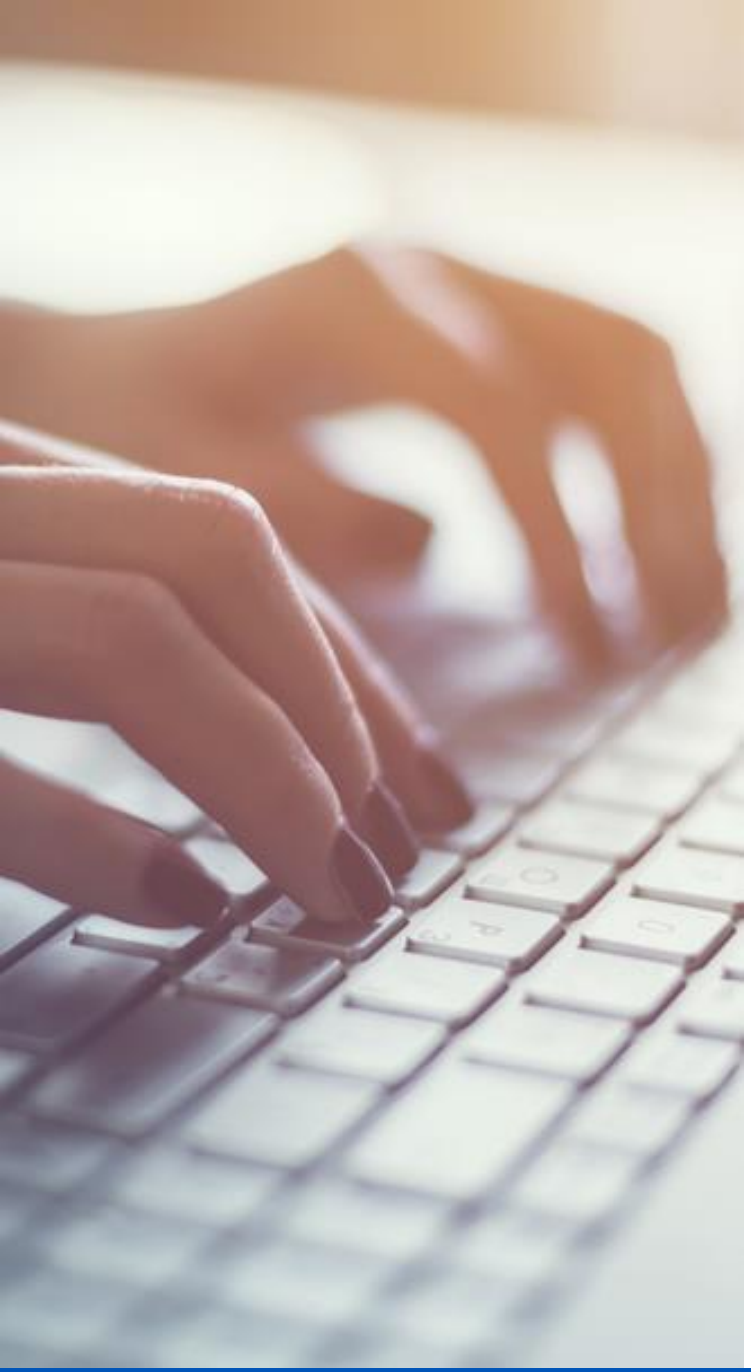
```
<input #userInput (keyup.enter)="user=userInput.value">
```

Reference variable

Pseudo event binding

- **Examples of pseudo key events**
  - `keydown.a`, `keypress.2` ..
- **Special values**
  - `keyup.space`, `keyup.dot`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



5.1: Pseudo Key Events in Action

Do Now

- In your Exercise Manual, please refer to Do Now 5.1: Pseudo Key Events in Action

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Component Interactions
- Property Bindings
- Events

Direct Communication

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

@ViewChild

- **`@ViewChild` allows direct access to all non-private members of the child Component**
  - Template reference variable is limited
    - Allows access to the child only from the template
- **Angular uses DI to populate the `@ViewChild` variable**

```
export class AppComponent {

  @ViewChild(CurrentWeatherComponent)
  currentWeatherComponent : CurrentWeatherComponent;

 private onClick(){
    this.title='new title';
    this.currentWeatherComponent.currentCity = "Tokyo";
  }
```

Class of `@ViewChild` variable

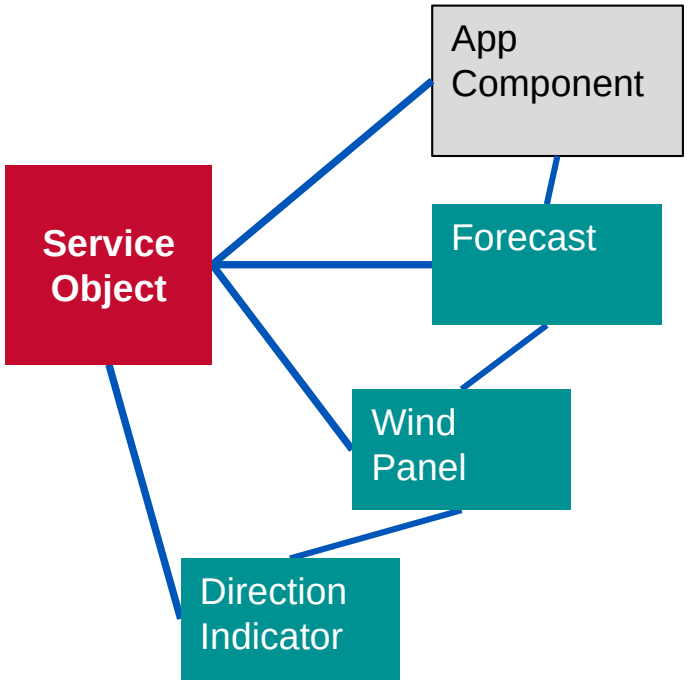
Access members and variables of the view child

DI = dependency injection

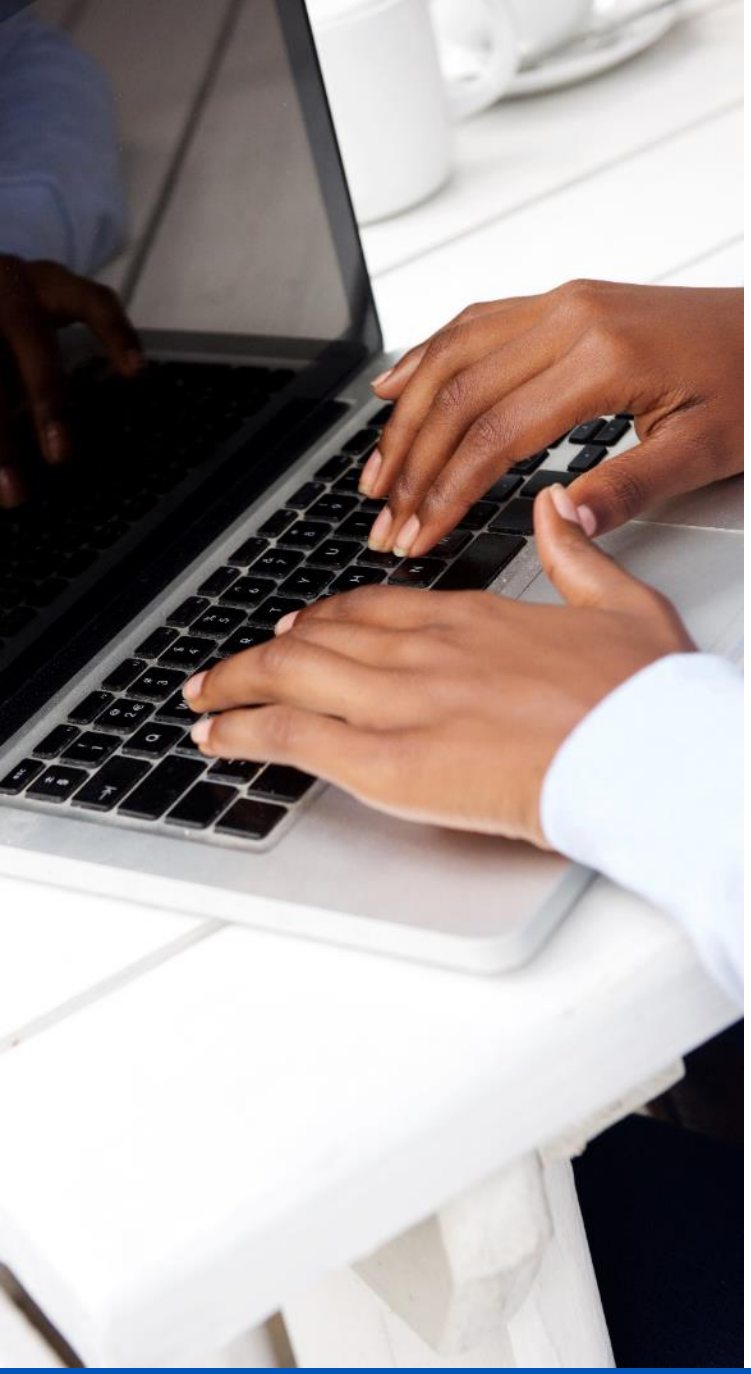
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Communication via Shared Service Objects

- **Dependency injection (DI) provides another communication option**
  - Shared service objects
- **In Chapter 3, we showed that a service can be shared across multiple Components using DI**
  - Can use the service to share data between all Components that access it
- **Sharing model data with an injected service object is best practice**
- **Use event handlers and property bindings for sharing all data other than the model**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 5.2

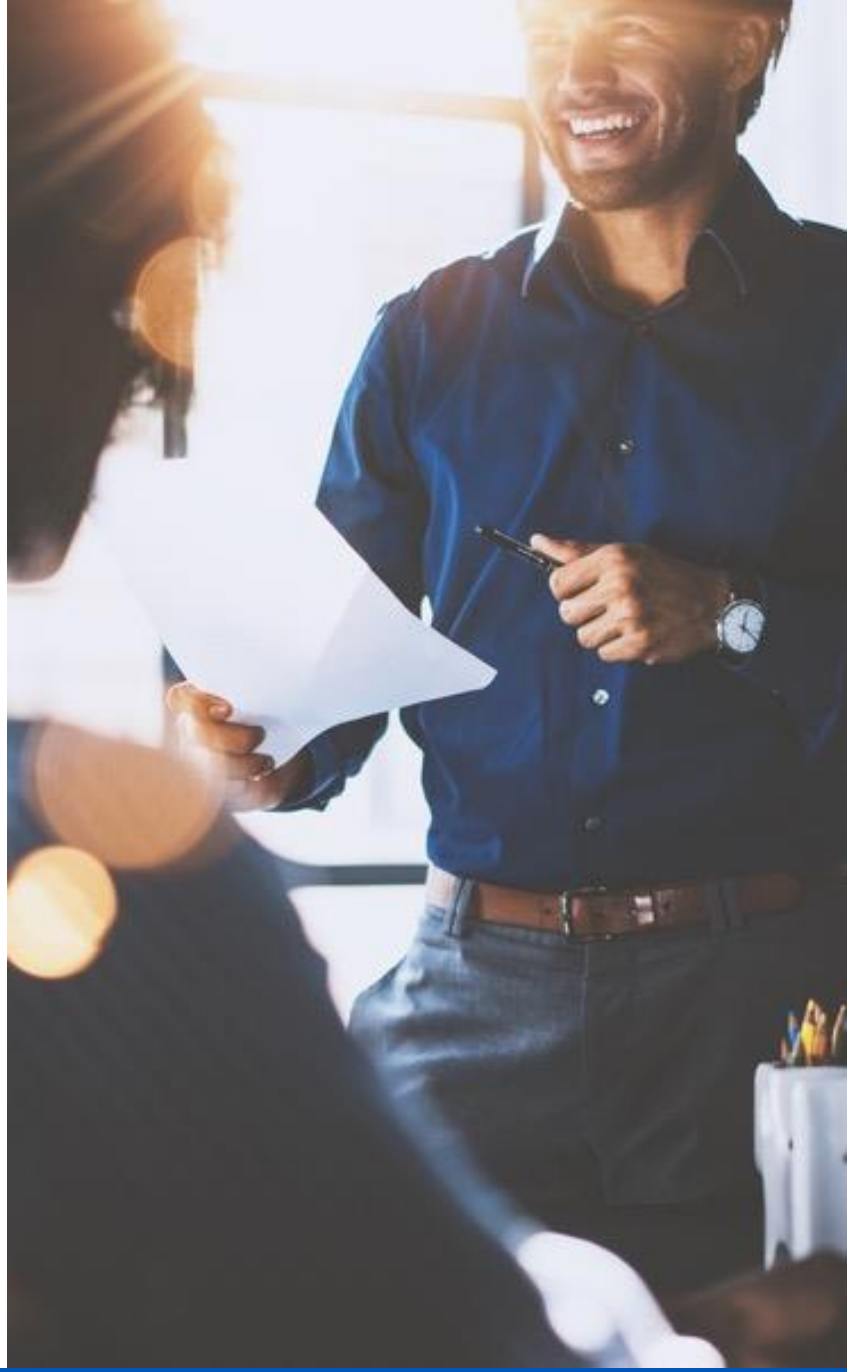
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 5.2: Event Handling

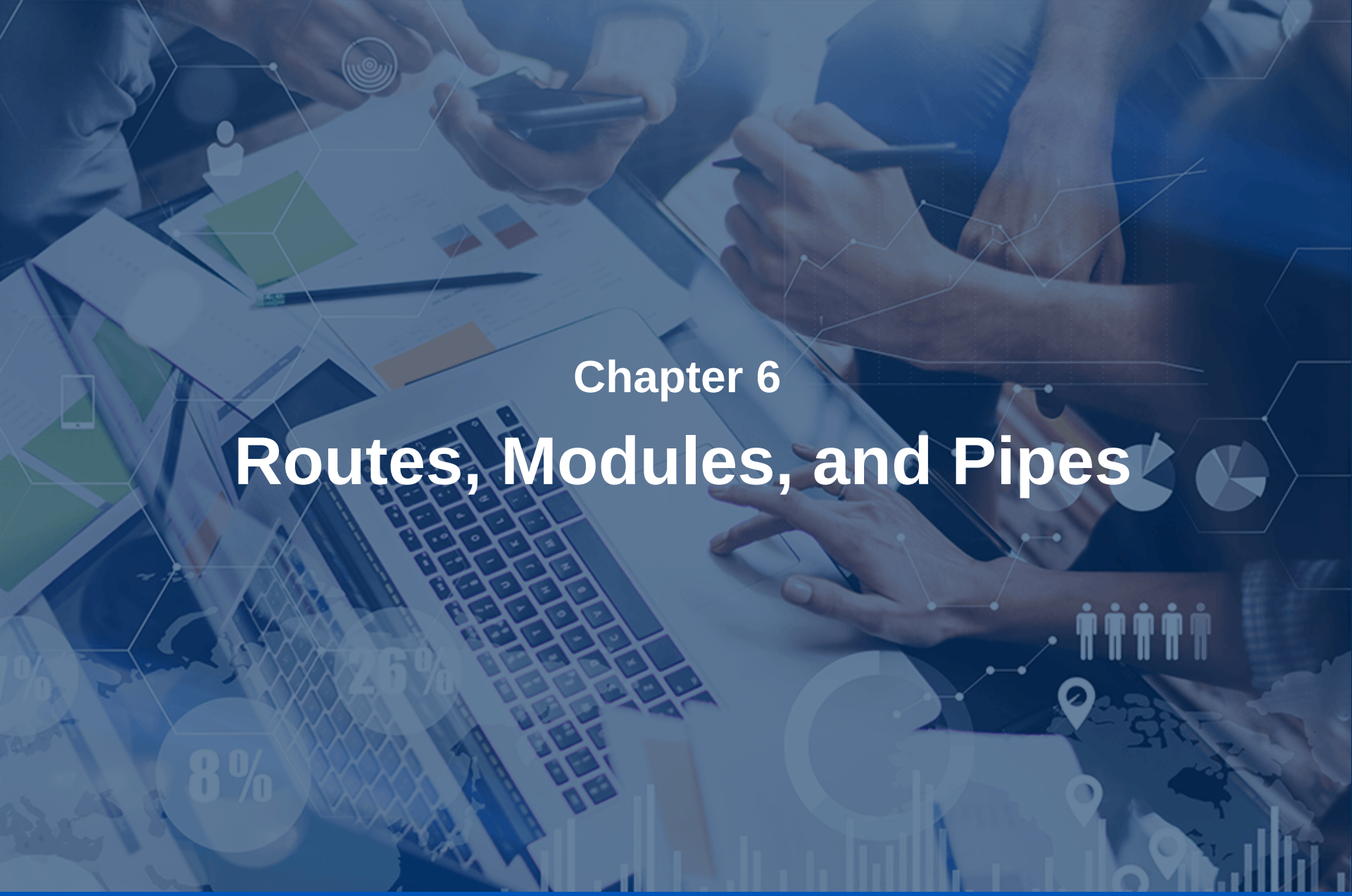
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Exchange information between Components
- Display data from parent Components
- Define event emitters and handlers
- Share information with service Components



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



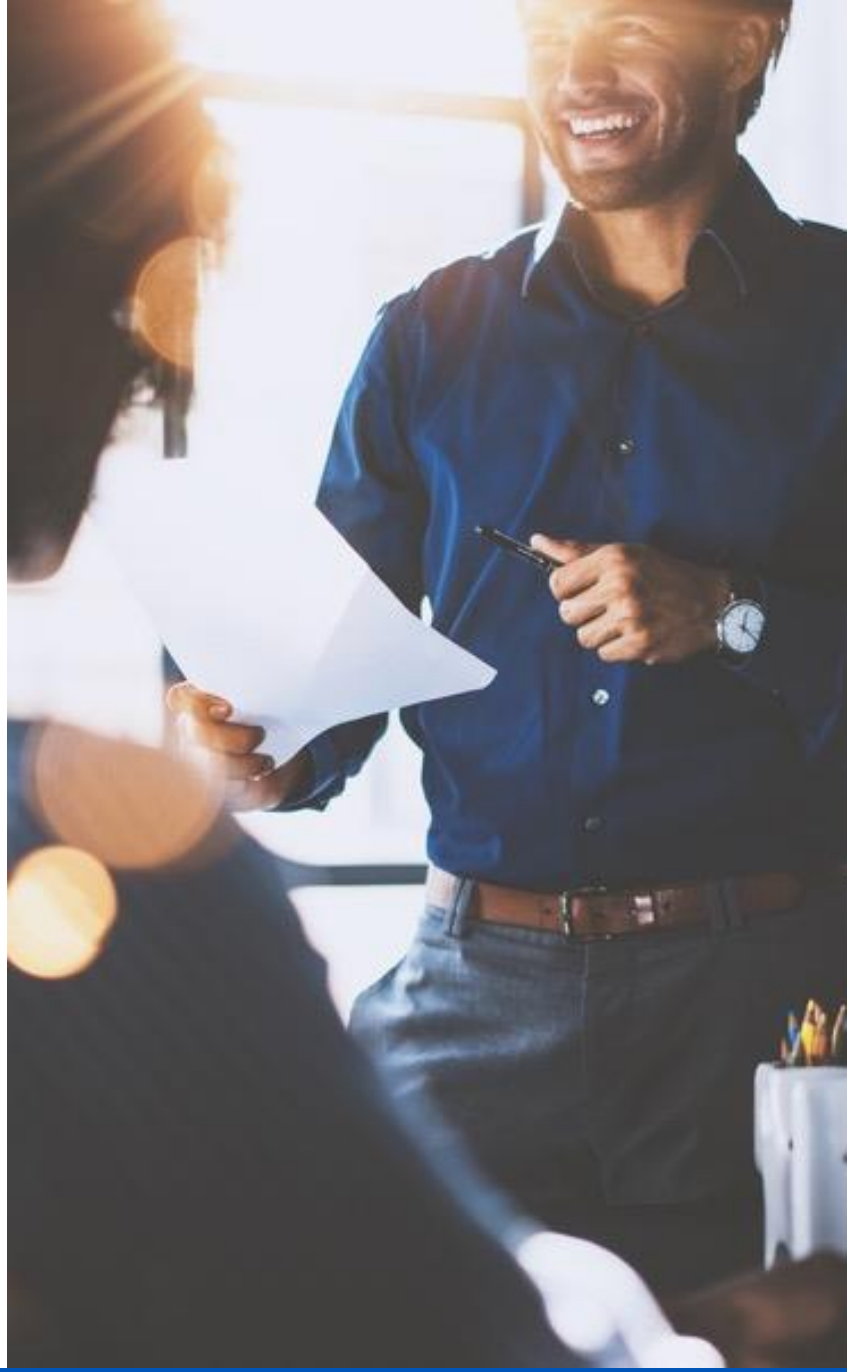
Chapter 6

Routes, Modules, and Pipes

LEARNING TREE INTERNATIONAL

Objectives

- Implement navigation with the Component Router
- Declare routes and configure the router
- Interact with the router programmatically
- Transform data with pipes

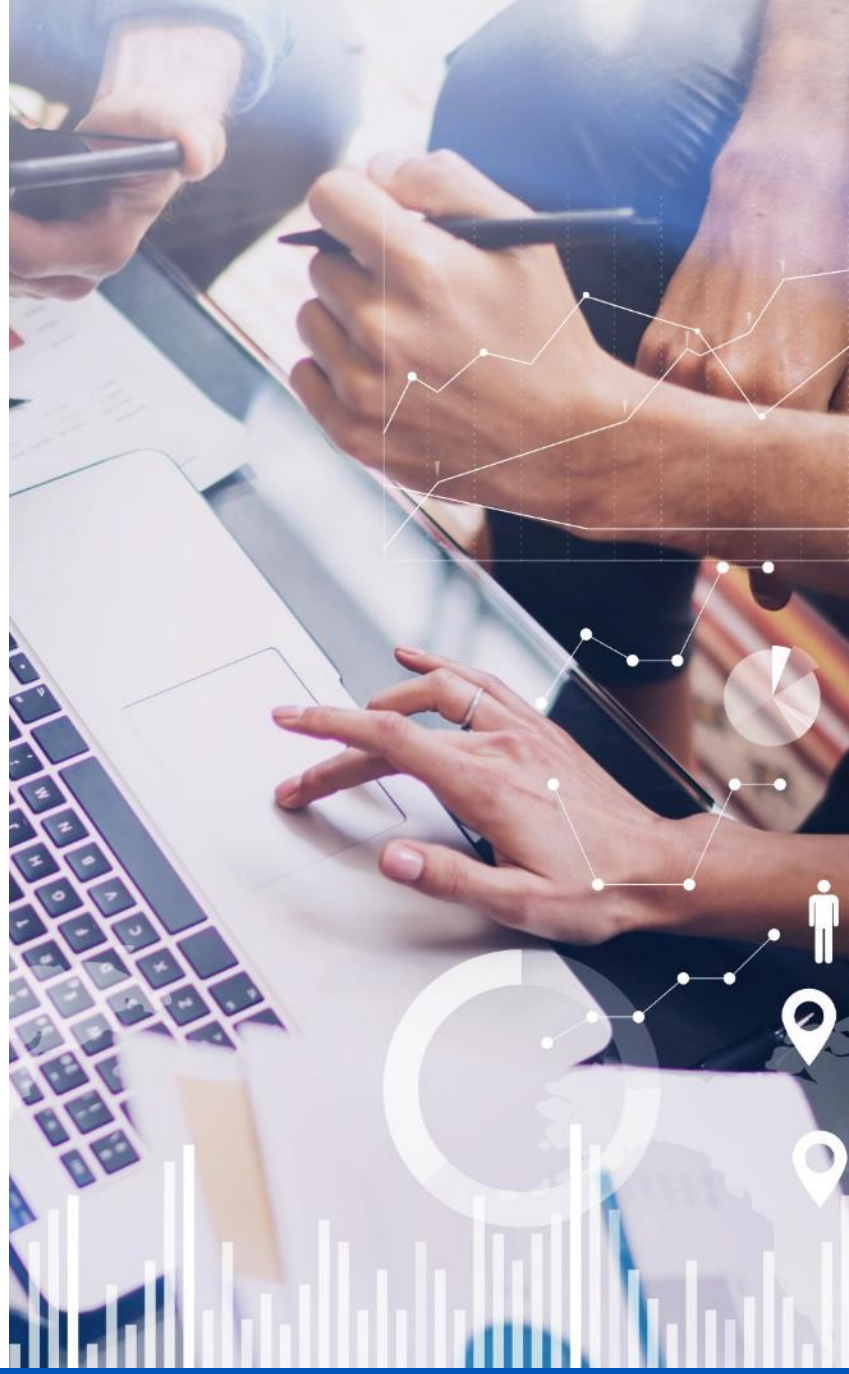


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

The Component Router

- Routing and Feature Modules
- Pipes



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing the Component Router

- **The router provides a navigation mechanism**
  - Browser URLs control which Angular Component to display
- **Routing controlled**
  - By browser URL
  - By Back/Forward buttons
  - By binding to links in a page
  - Programmatically
- **Benefits**
  - Works well with a tab-style view paradigm
  - Provides multiple independent Component trees
  - Reduces load times

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Configuring the Component Router

- **Steps**
  1. Configure the router environment
  2. Define navigation routes
  3. Bind routes to templates
  4. Programmatic navigation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

1. Configure the Router Environment

- **Add a `<base>` element to the `index.html` page**
  - Used to resolve relative paths within the application
  - Required to allow JavaScript to push URLs onto the browser history

```
<base href="/">
```

- **If your application will be hosted on a path within your server**
  - http://www.myserver.com/myangularapp/
  - The `<base>` `href` attribute should include the path

```
<base href="/myangularapp/">
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Router Configuration Module

- **The router is configured by a collection of path/Component pairs**
  - `path: 'home', component: HomeComponent`
    - When the URL is `/home`, the `HomeComponent` will be displayed

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

2. Define Navigation Routes

- **Routes are defined in an array**
- **Each array element consists of**
  - A path
    - Relative to the application base path
  - A Component to activate when the path is matched

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'home', component: HomeComponent },
  { path: 'forecast', component: ForecastComponent },
  { path: 'preferences', component: PreferencesComponent }
];
```

- **A route with a `path` of `''` should be provided**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Bootstrapping the Router

- **To use the router**
  - The application module provides router configuration via dependency injection
- **Add `RouterModule.forRoot(routes)`to the module `imports`**
  - `routes` is the `Routes` array defined previously
- **Dependency injection makes the router configuration available to any Component**

app.module.ts

```
const routes: Routes = […

@NgModule({
 declarations: […]
 imports: [
       RouterModule.forRoot(routes),
  ]
});
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

3. Bind Routes to Templates

- **The top-level (application shell) Component must provide a `<router-outlet>`**
  - The location for the Components specified by the router to be displayed

```
<router-outlet></router-outlet>
```

- **Navigation links are provided by the `<a>` element**
  - Using a binding to `routerLink` instead of an `href` attribute

```
<a routerLink='/home'>Home</a>
```

Matches the path in the `router` config

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Redirecting Routes

- **A route with a path of `''` should be provided**
  - But this path may *not* have an obvious mapping to a Component
- **Redirects offer a solution**
  - `redirectTo`: the path to activate
  - `pathMatch`: how to evaluate the path
    - `full`: must match the entire path
    - `prefix`: matches a path which starts with the path in the route

app.routes.ts

```
const routes: Routes = [
  {
    path: '',
    redirectTo: '/home',
    pathMatch: 'full'
  },
  { path: 'home', component: HomeComponent },…
];
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Parameterizing Routes

- **Routes may be specified as having a data Component**
  - For example, identifying which forecast location to use

```
{ path: 'forecast/:location', component: ForecastComponent }
```

- **If the supplied URL was `http://…./forecast/Toronto`**
  - The `location` parameter would be `Toronto`
- **`path` declarations can have multiple placeholders for parameters**
  - `forecast/:location/:units`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Retrieving Route Parameters

- **Route parameters are accessed via `ActivatedRoute`**
  - Automatically supplied via dependency injection
- **`ActivatedRoute.params` returns an Observable**
  - Calls our function with an array of parameters when available
  - Route parameter can be extracted from the array

Injected

```
constructor(private route: ActivatedRoute,
            private weatherService : WeatherService ){}

ngOnInit(){
   this.route.params.subscribe(params => {
     this.location = params['location'];
   });
}
```

Extract the route parameter called `'location'`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

4. Programmatic Navigation

- **Any Component may use the router for navigation**
  - Add a Router argument to the constructor
    - The dependency injector automatically provides the router
- **Use `Router.navigate()` to perform routing**
  - Router path prefixed with a `'/'`

```
constructor( private router : Router ) {}

onClickWarnings(){
  this.router.navigate(['/warnings'])
}
```

Path must start with /

```
<button (click)="onClickWarnings()">Warnings</button>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Adding Path Parameters Programmatically

- **Route parameters may be added to the array passed to `navigate()`**

```
{ path: 'forecast/:location', component: ForecastComponent }
```

```
 onClickForecast(){
    let theLocation = "Paris";
    this.router.navigate(['/forecast', theLocation ]);

  }
```

- **Path may have multiple parameters**

```
'forecast/:location/:units'
```

- **Supplied array must provide elements for the path**
  - Each placeholder in the path
    - The array must be ordered to match the path specifier

```
this.router.navigate(['/forecast', location, units ]);
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Query Parameters

- **URL data can be sent as query parameters**
  - An alternative to path parameters
  - Useful when parameters are optional
- **Query parameters are sent as `name=value` pairs at the end of the URL**
  - Separated from the URL by a `?`
  - `http://…./forecast?location=Toronto&units=Celsius`
- **Query parameters are not declared in the router configuration**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Retrieving Query Parameters

- **Query parameters are also accessed via `ActivatedRoute`**
  - In the `queryParams Observable`

```
constructor(private route: ActivatedRoute, …){}

ngOnInit(){
  this.route.queryParams.subscribe
      (params => {
              this.location = params['location'];
      });
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Adding Query Parameters Programmatically

- **Query parameters are added to the request as an object with a property `queryParams`**
- **`queryParams` is converted to the query parameters in the URL**
  - Each property becoming a query parameter

```
{ path: 'forecast', component: ForecastComponent }
```

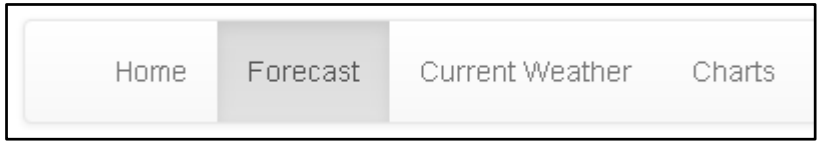
```
onClickForecastByQuery(){
    let location = “Toronto";
    let units = "Celsius";
    this.router.navigate(['/forecast'],
        {queryParams :
        {'location' : location, 'units' : units}});
}
```

- `http://…./forecast?`**`location`**`=Toronto&`**`units`**`=Celcius`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Showing the Active Route in the Template

- **A CSS class may be toggled to show the active route**
- **Add the `routerLinkActive` directive to an element**
- **The specified class is added to the element**
  - If that element or it's child matches the current active route



Class `'active'` is added to the `<li>` if the active route is `/forecast`

```
<li routerLinkActive="active" >
  <a routerLink='/forecast'>Forecast</a>
</li>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Route Guards

- **Prevent routes being activated/deactivated**
- **Based on some programmatic condition**
  - Perhaps whether the user has logged it?
- **Guards are specified in the Routes object**

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'home', component: HomeComponent },
  { path: 'forecast', component: ForecastComponent,
                          canActivate: [RouteGuard] },
  { path: 'preferences', component: PreferencesComponent }
];
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

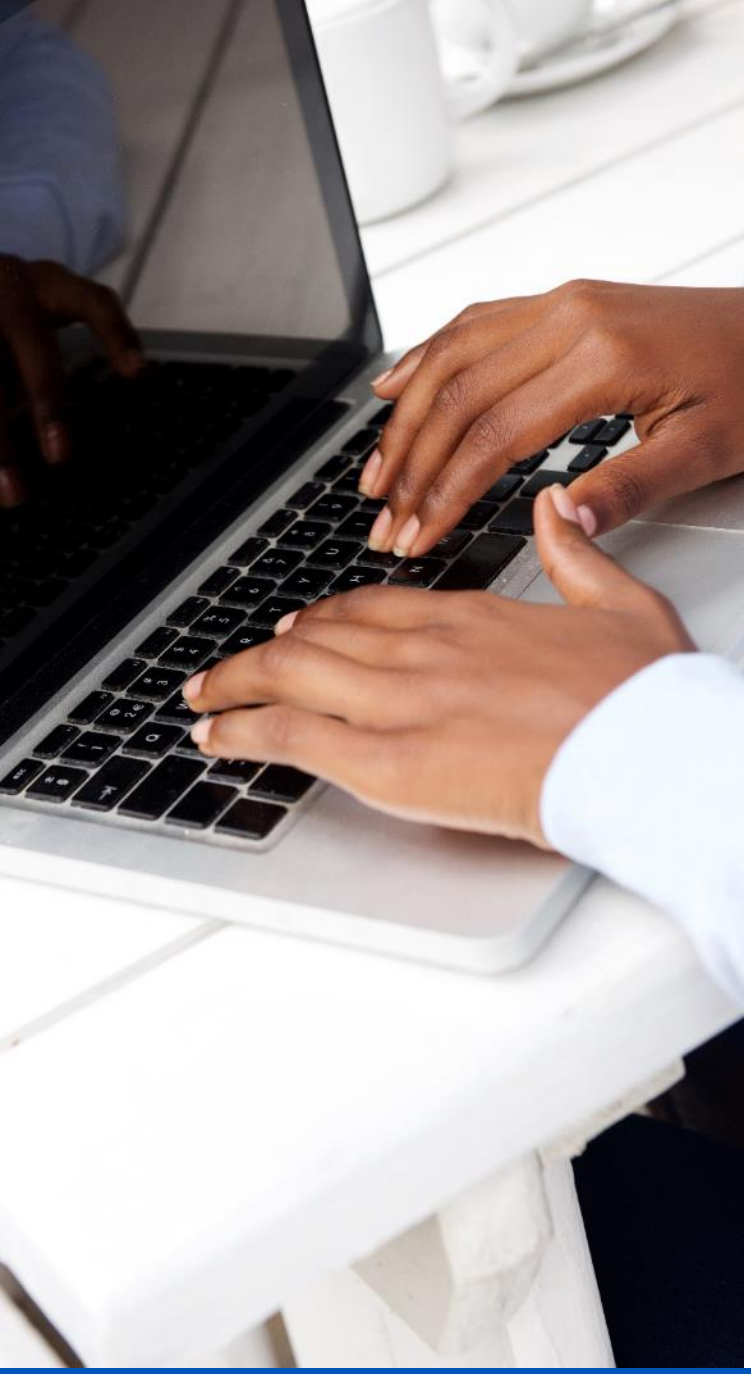
canActivate canDeactivate Guards

- **CanActivate**
  - Prevents routes being activated until a condition is met
  - Returns true to indicate a route can be activated

```
@Injectable()
export class RouteGuard implements CanActivate {
  canActivate() {
    console.log('RouteGuard#canActivate called');
    if(some_condition) return true;
    return false;
  }
}
```

- **canDeactivate**
  - Similar structure, but returns true if it is OK to leave the current page
- **Guards may return a Boolean or an `Observable<Boolean>`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 6.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 6.1: Integrating the Component Router

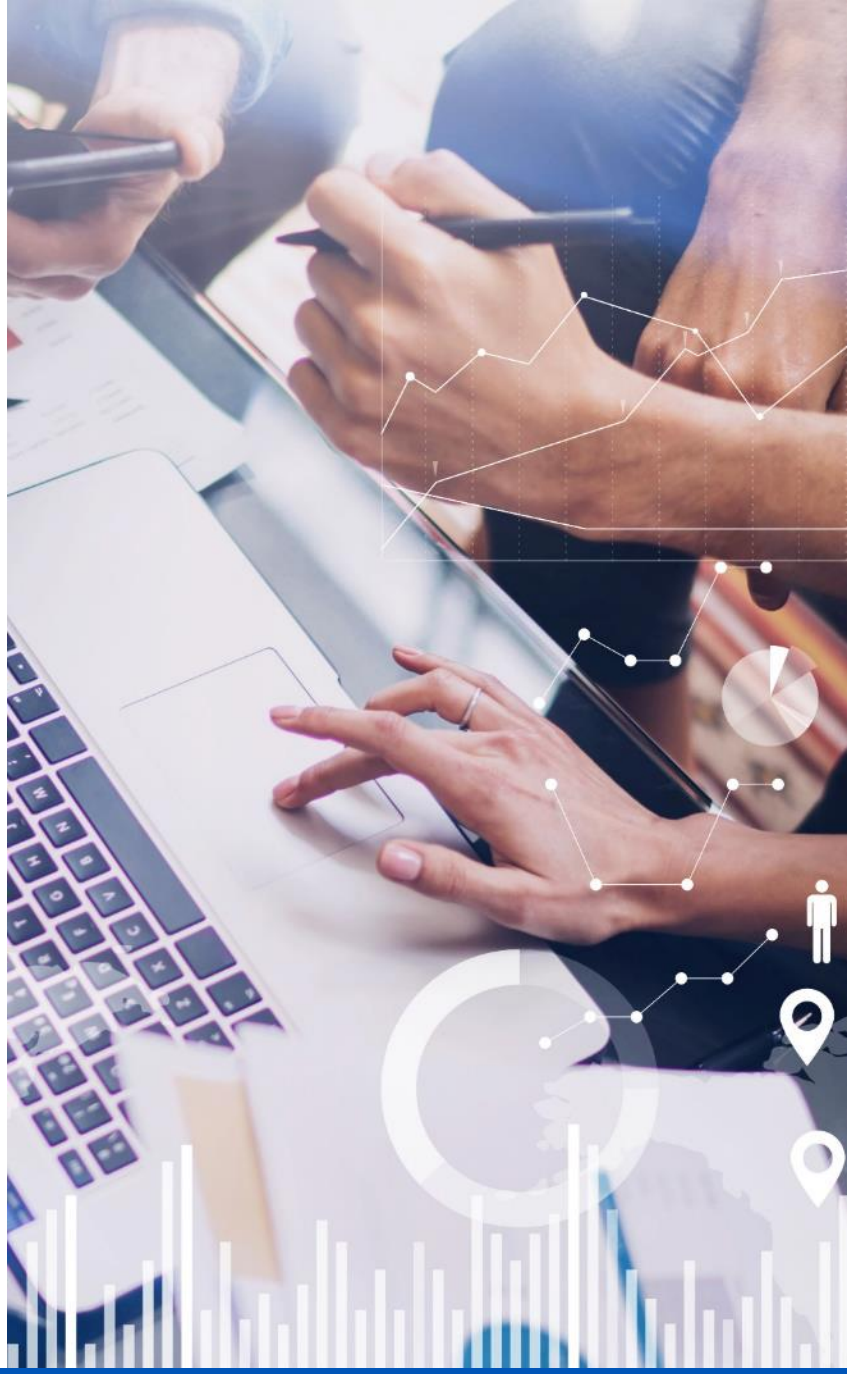
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- The Component Router

Routing and Feature Modules

- Pipes



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Feature Modules Revisited

- **Angular code (components, directives, services, pipes, …)**
  - Must be placed in a module
- **Every application must have a minimum of a root module**
  - It may have feature modules
- **Feature Modules may be used to**
  - Package reusable code
  - Improve performance through lazy loading
  - Provide logical grouping of code

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating a Feature Module

- **Angular CLI makes module creation simple**

```
ng generate module CustomerAccountsModule
```

- **Generated module is very basic**
  - Does not actually do anything

```
@NgModule({
  imports: [
    CommonModule,
]})
export class CustomerAccountsModule { }
```

- **As with the root module, declarations must be added**

```
declarations: [AccountComponent]
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exposing Module Content

- **Module content is made available to the application in various ways**
  - Dependent on the purpose of the module and the content
- **Code is published from a module by**
  - Direct export (using `exports` property of `@NgModule`)
  - As a provider (using `providers` property of `@NgModule`)
  - Lazy loading via the component router

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Routing Modules

- **Routing modules allow routes to be separated from other code declarations**
  - Considered good practice but not required

AccountsRoutingModule.ts

```
const routes: Routes = {
  path: 'account',
  component:AccountComponent
}];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AccountsRoutingModule { }
```

Routes collection defined as normal

`forChild()` used as this is a feature module

`RouterModule` (with the new routes) exported

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Importing Routes From a Routing Module

- **Module to be imported is added to the `imports` property of `@NgModule`**
  - Loads the module eagerly
    - Before it is needed

```
@NgModule({
  imports: [
    CommonModule,
    AccountsRoutingModule,
    ReactiveFormsModule,
    FormsModule
  ],
  declarations: [AccountComponent]
})
export class AccountsModule { }
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lazy Loading

- **Modules may be loaded on-demand by the component router**
- **Uses the `loadChildren` property of the `Routes` object**
  - Instead of component

Route definitions for root component

```
{
   path: 'myflights',
   component: MyFlightsComponent
 },

 {
   path: 'account',
   loadChildren: loadChildren: () => import('./accounts/accounts.module')
       .then(mod => mod.AccountsModule)
}
```

"Normal" route definition

Module loading route

Class

Path

- **`loadChildren` property uses an arrow function to import the module**
- **It then uses the `then` function of a `Promise` to load the module**

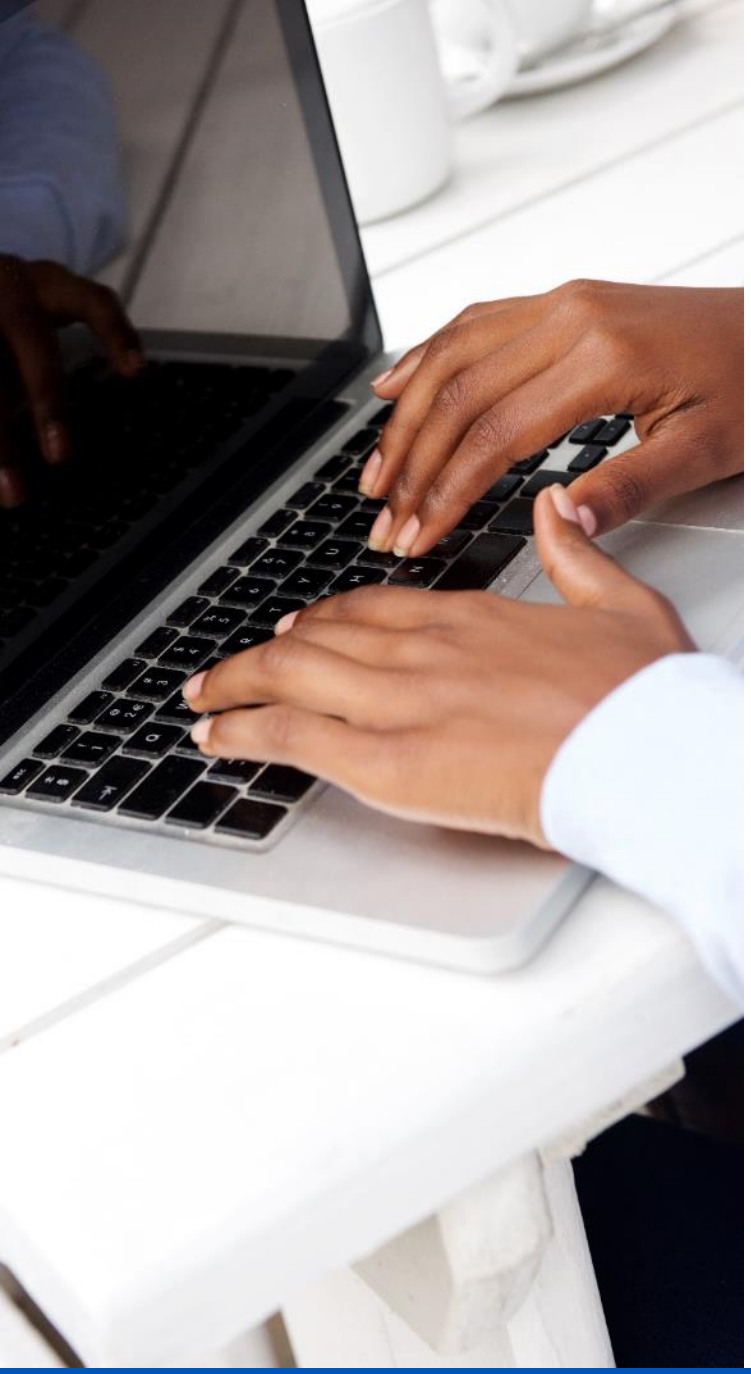
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lazy Loading Route Resolution

- **In our example**
  - The path in the root module is `'account'`
  - The path in the accounts module is also `'account'`
- **The path to the `AccountComponent` is therefore:**
  - `http://localhost:4200/account/account`
- **The feature module may define many routes**
  - Each will define a path relative to the path from the root (or parent) module
  - Examples:

```
http://localhost:4200/account/account
http://localhost:4200/account/address
http://localhost:4200/account/history
…
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 6.2

Exercise Manual

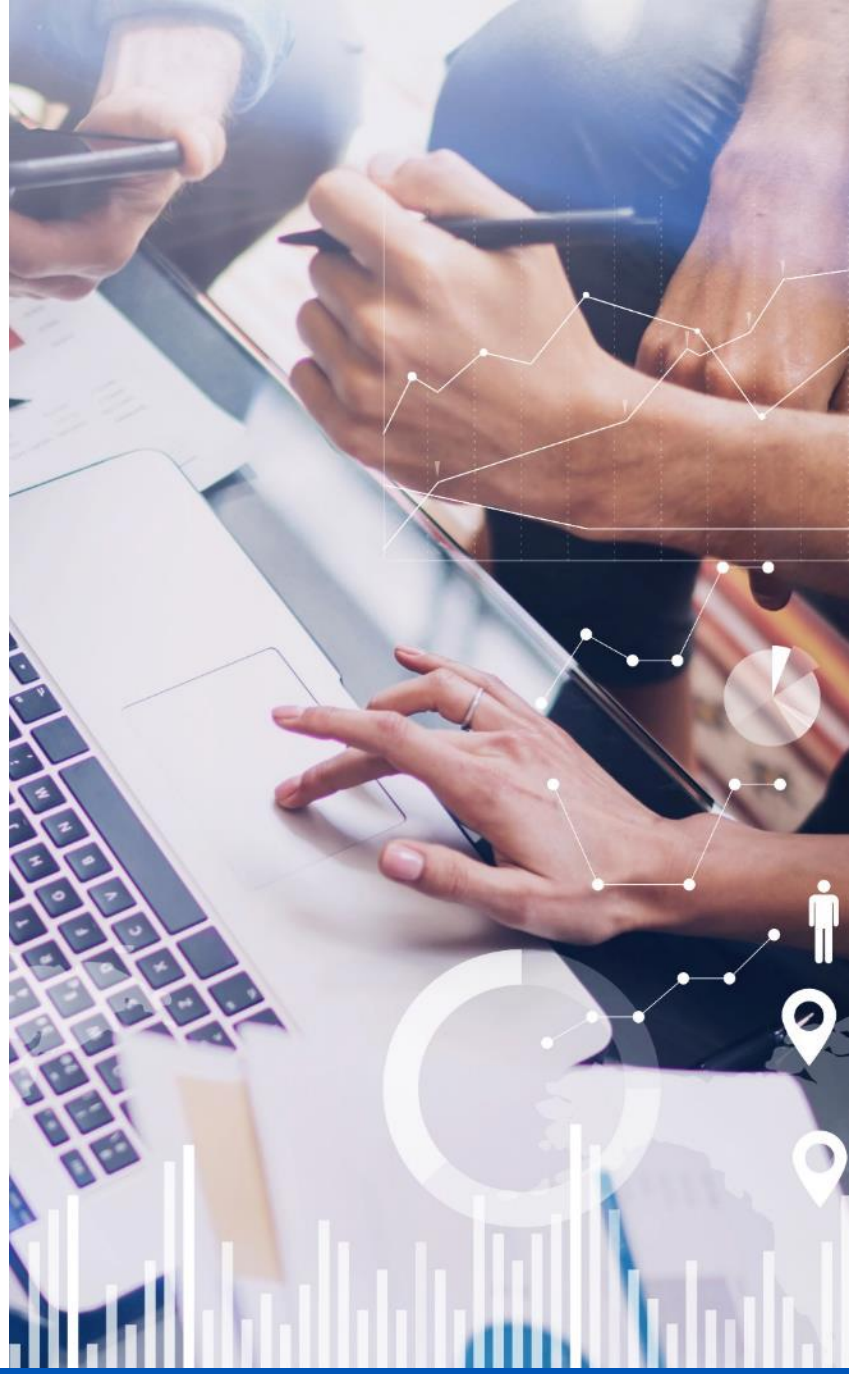
In your Exercise Manual, please refer to Hands-On Exercise 6.2: Creating a Feature Module

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- The Component Router
- Routing and Feature Modules

Pipes



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Pipes

- **Pipes provide data transformation**
  - Formatting dates
  - Calculating and displaying percentages
  - Converting text case
  - Internationalization
- **Many built-in pipes**
  - CurrencyPipe—formats a number as local currency
  - DatePipe—formats a date
  - LowerCasePipe—converts text to lower case
  - JsonPipe—converts input object to JSON
    - Useful for debugging

JSON = JavaScript Object Notation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using Pipes

- **Pipes are used in templates**
- **Within interpolation {{}} expressions**

Built-in date pipe

```
<p>Weather for {{ forecastDate | date }}</p>
```

Pipe symbol

- **Pipes may be parameterized**
  - Parameters separated from pipe name by a `':'`

```
<p>Weather for {{ forecastDate | date : "MM-dd-yyyy" }}</p>
```

- **Standard pipes are automatically available in all modules**
  - Custom pipes must be registered in the module declarations

See documentation on DatePipe for full formatting information

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Binding and Chaining

- **Pipes parameters may be bound to Component properties**

```
<p>Weather for {{ forecastDate | date : dateFormat }}</p>
```

`dateFormat` is a Component property

- **Pipes may be chained**

```
<p>Weather for
{{ forecastDate | date : dateFormat | lowercase }}</p>
```

  - Example formats date, then outputs in lowercase

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom Pipes

- **Custom Pipes**
  - Are a class with a `@Pipe()` decorator
  - `name` property of the decorator is the name used in the template
- **Must implement `PipeTransform`**
  - Provide a method called `transform()`
  - `transform()` returns a string to display
- **`transform()` takes the value to transform and any optional arguments**
- **Example: Appends the wind speed units to the wind speed**
  - Usage:

```
<li>Speed - {{windSpeedKnots  | windSpeed }}</li>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom Pipe Implementation

wind-speed.pipe.ts

```
@Pipe({
  name: 'windSpeed'
})

export class WindSpeedPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    return value + " knots";
  }

}
```

Pipe name

Pipe implementation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Declare Custom Pipes in the Module

- **Custom pipes must be declared in the `@NgModule` metadata**

app.module.ts

```
@NgModule({
  declarations: [
    HomeComponent,…
    WindSpeedPipe
  ],
  imports: […
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Parameters

- **Multiple parameters may be supplied to a pipe**
  - Separated by colons `":"`
  - Order must match `transform()` method

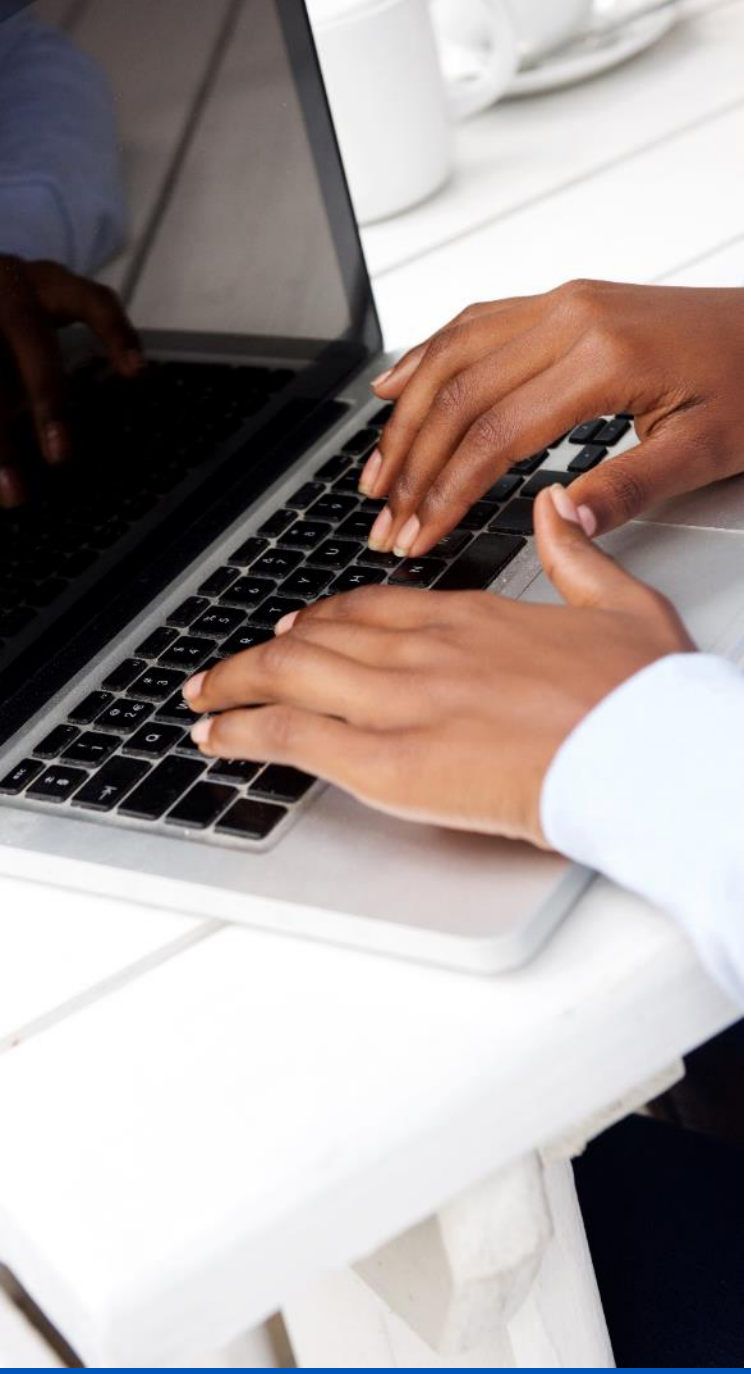
```
{{windSpeedKnots  | windSpeed : 'MPH' : 0.91 }}
```

- **Parameters are passed to the `transform()` method as method arguments**

```
export class WindSpeedPipe implements PipeTransform {
  transform(value: any,
                units : string = "knots",
                conversion : number = 1.0 ): string {
    return value * conversion + " " + units ;
  }
}
```

- **Use default argument values to make parameters optional**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 6.3

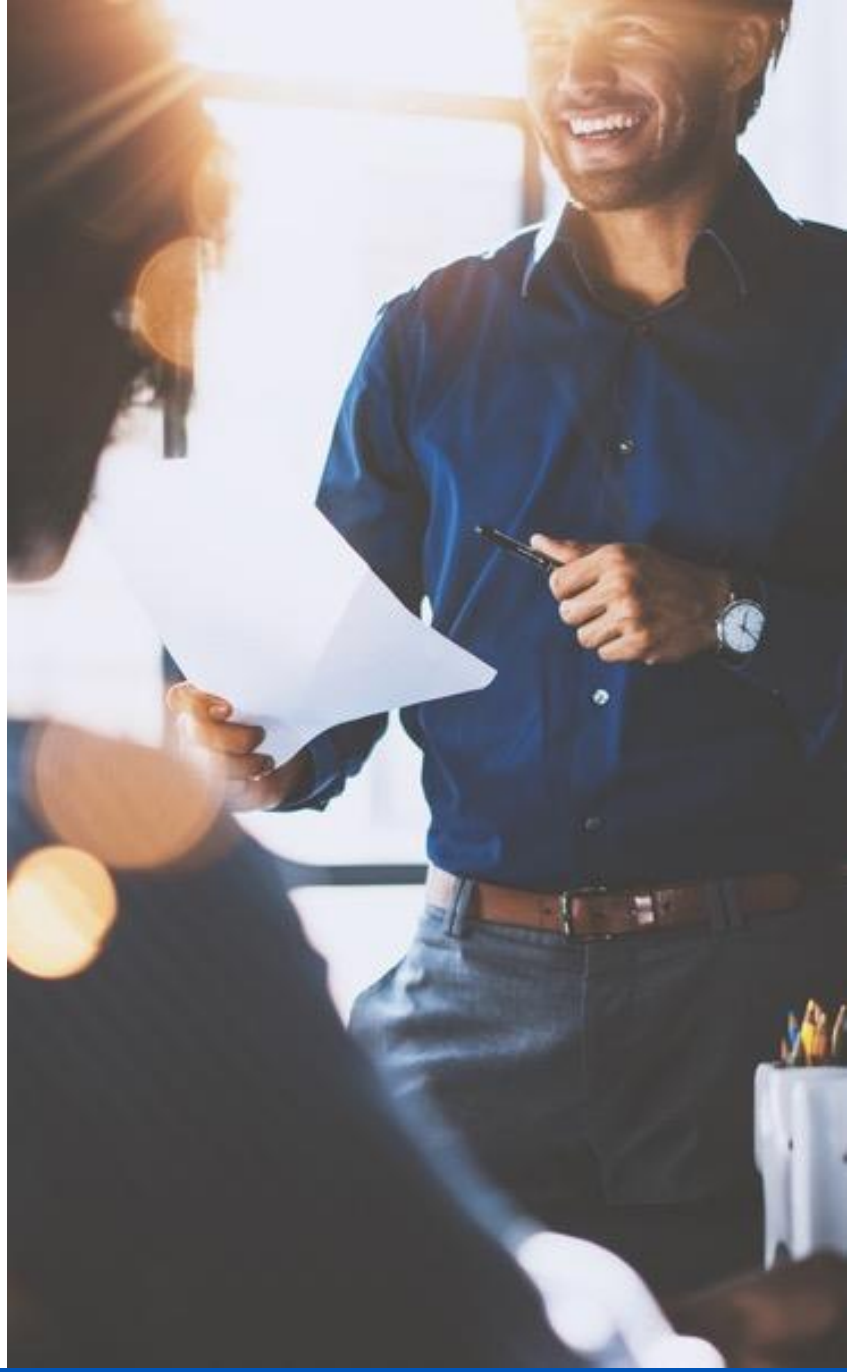
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 6.3: Creating a Custom Pipe

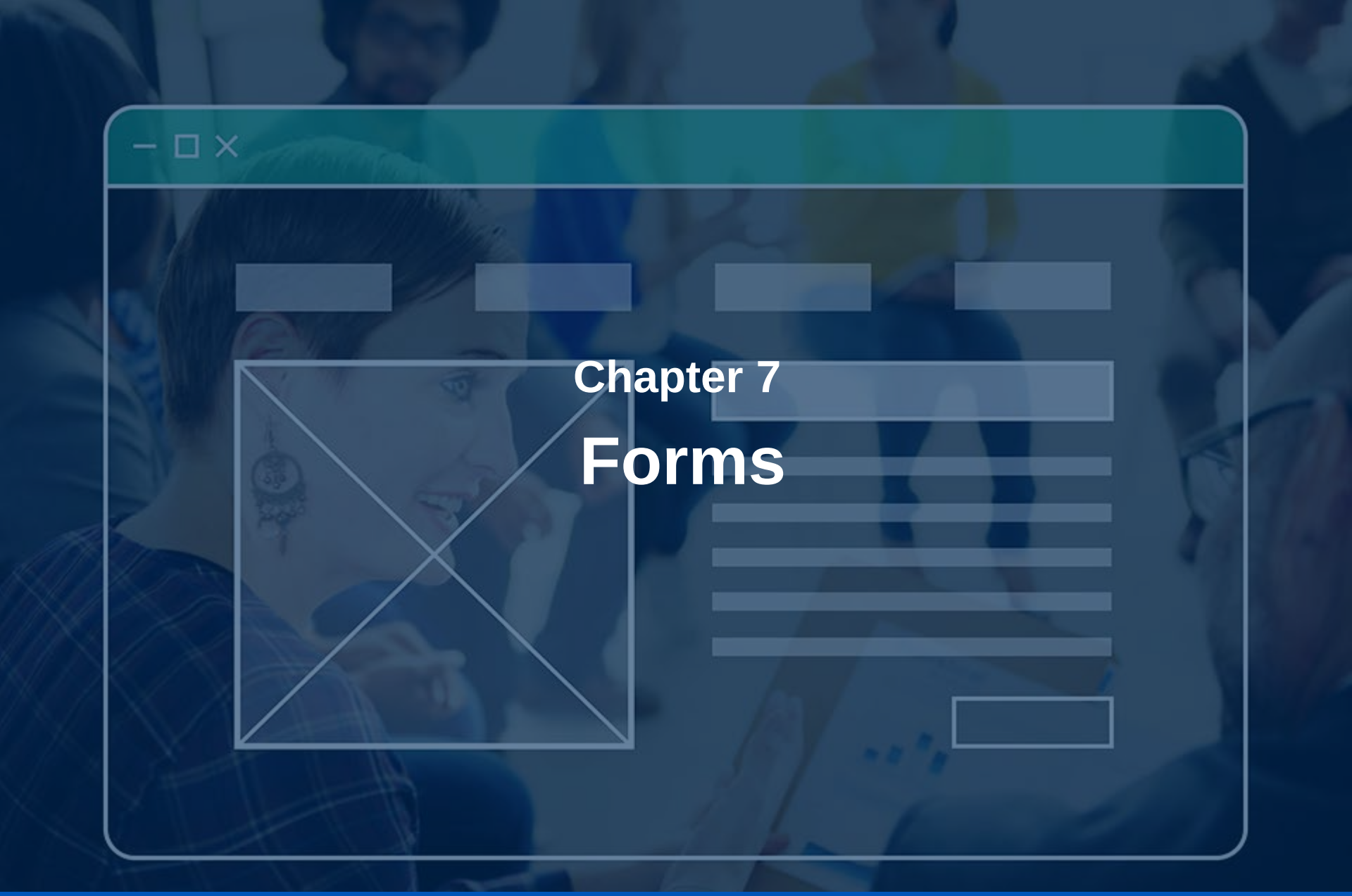
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Implement navigation with the Component Router
- Declare routes and configure the router
- Interact with the router programmatically
- Transform data with pipes



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



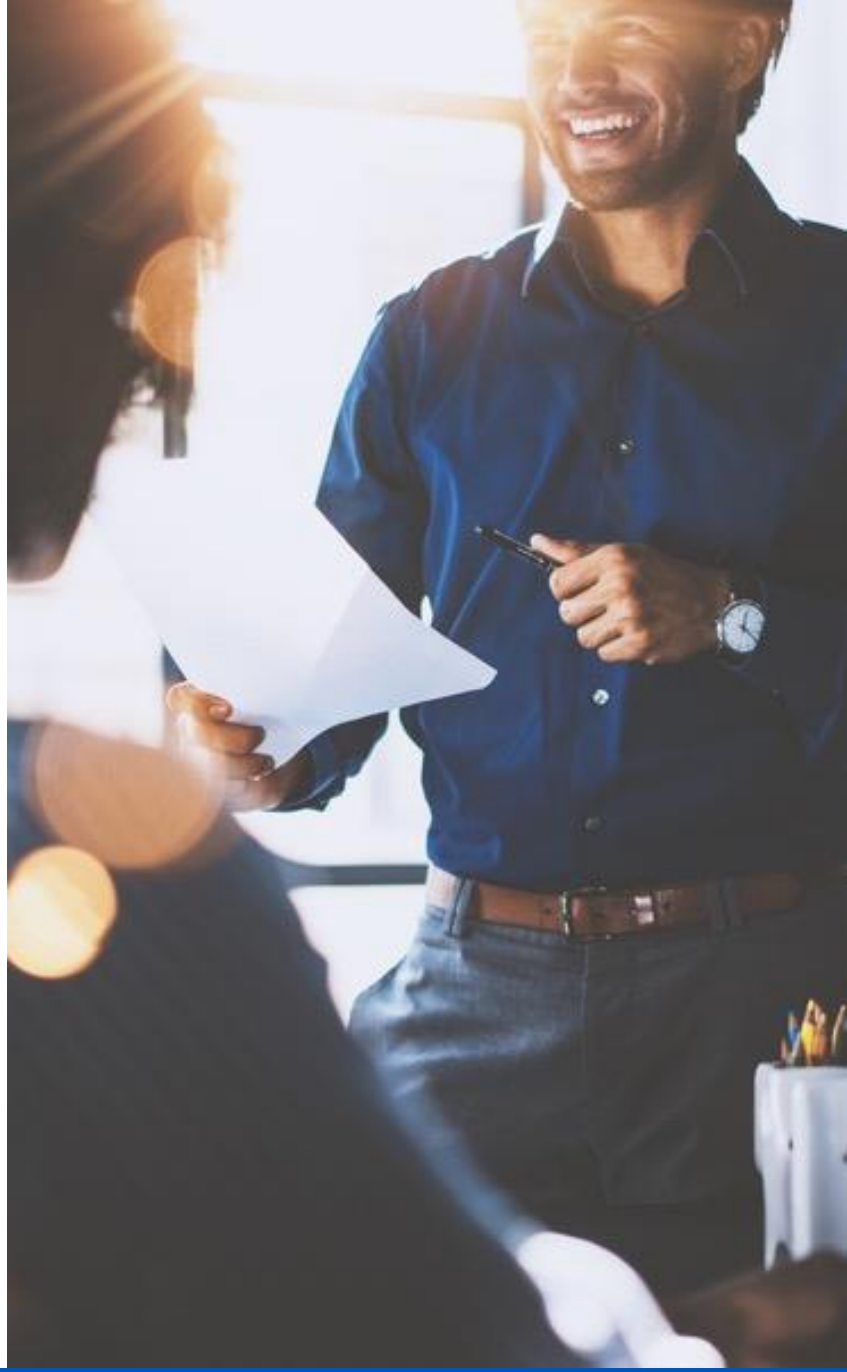
Chapter 7

Forms

LEARNING TREE INTERNATIONAL

Objectives

- Define a form data model
- Build a template-driven form
- Provide validation and feedback
- Develop a reactive form

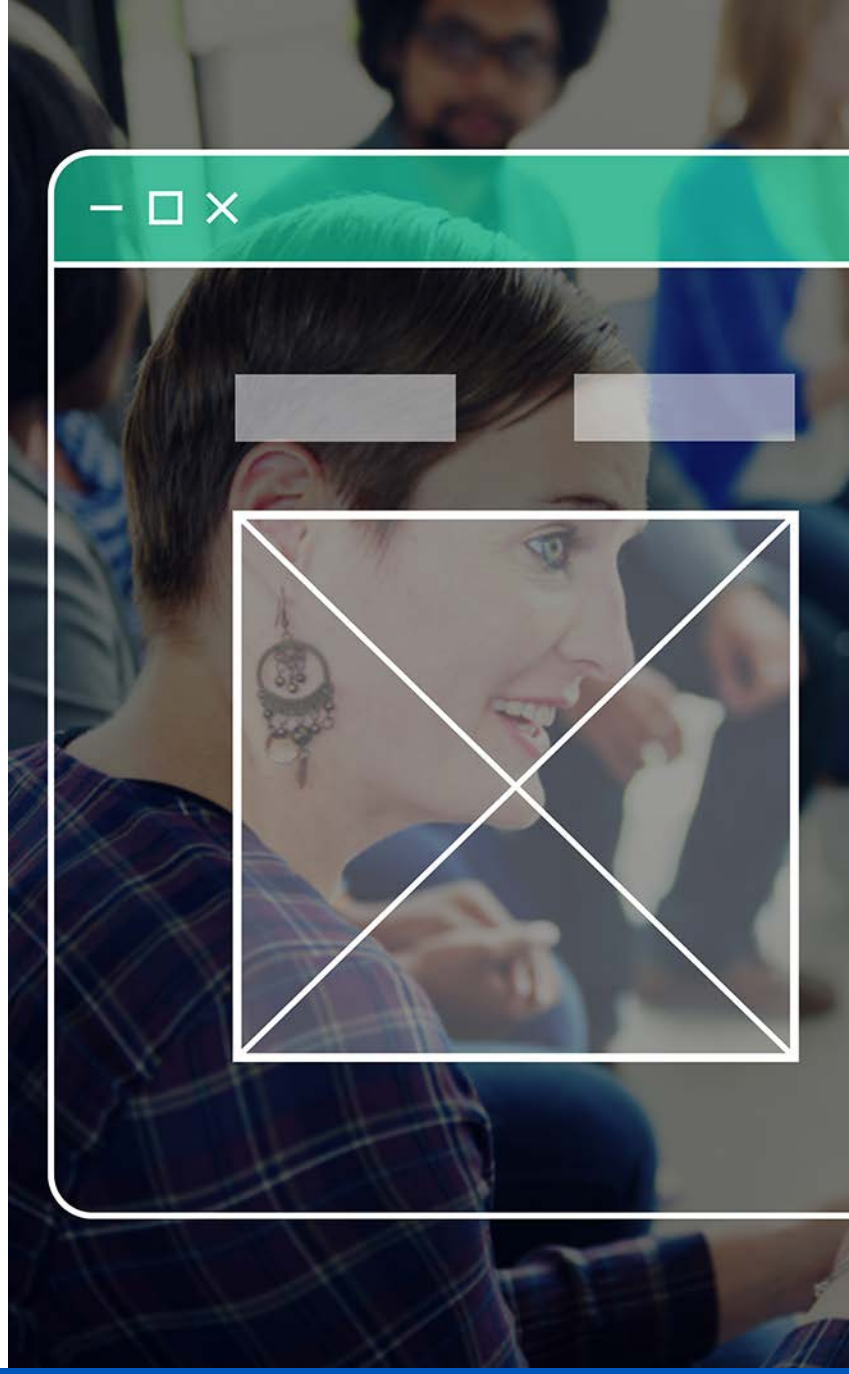


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Form Processing

- Template Driven Forms
- Validation and Feedback
- Reactive Forms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

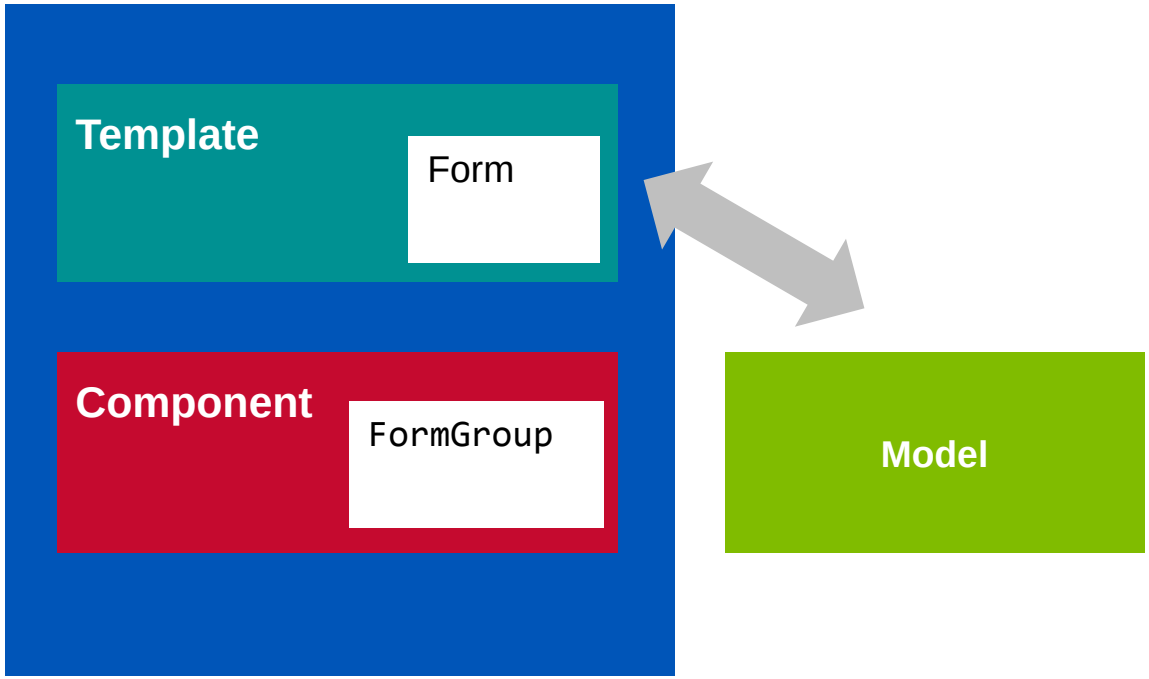
Angular Forms

- **Angular forms provide data-bound input forms**
- **Features**
  - Bi-directional data binding
  - State tracking
  - Visual feedback via CSS classes
  - Display of validation errors
- **Form processing is handled by `FormGroup`**
  - Two related methods of building `FormGroup`
- **Template driven**
  - Angular builds a `FormGroup` from the HTML template
  - The simplest solution
- **Reactive**
  - Developer defines a `FormGroup` with the `FormBuilder`
    - Associates it with the form
  - Offers more flexibility and custom validations

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Form-Processing Architecture

- **Forms build on the Component architecture**
- **Component template defines a form**
  - HTML plus form directives



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Form Data Model

- **Forms are data-bound**
  - Fields in the HTML form are bound to model fields in the Component
  - The form and underlying data model are synchronized
- **Ideally, create a model object to represent the form data**
  - Model object should have fields to represent each control in the form
- **Our example is capturing user preferences for the weather application**

preferences.ts

```
export class Preferences {
  constructor(
    public name: string,
    public location: string,
    public temperatureUnits: string,
    public speedUnits: string,
    public forecastLength: number
  ){}
}
```

`public` arguments to a constructor become properties of the object

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Form Template

- **Minimal form template**
- **Standard HTML 5**
- **Note the use of `required` validation attributes**
  - Angular uses standard HTML 5 validation markup

preferences-form.component.html

```
<form>
  <div>
    <label for="name">Name</label>
    <input id="name" type="text" name="name" required>
  </div>
  <div>
      <label for="location">Location</label>
      <input id="name" type="text" name="location" required>
  </div>
  <button type="submit" >Submit</button>
 </form>
```

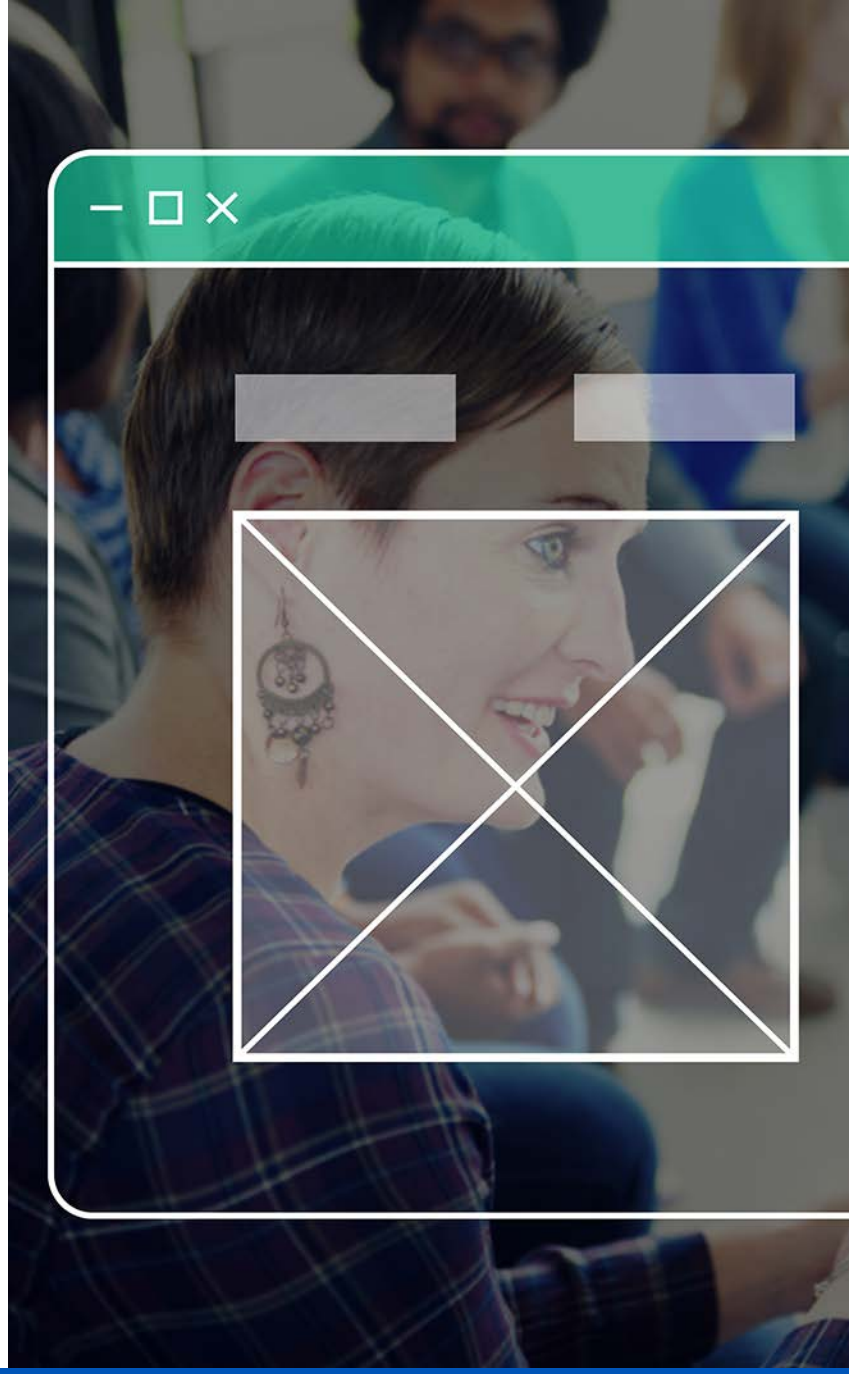
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Form Processing

Template Driven Forms

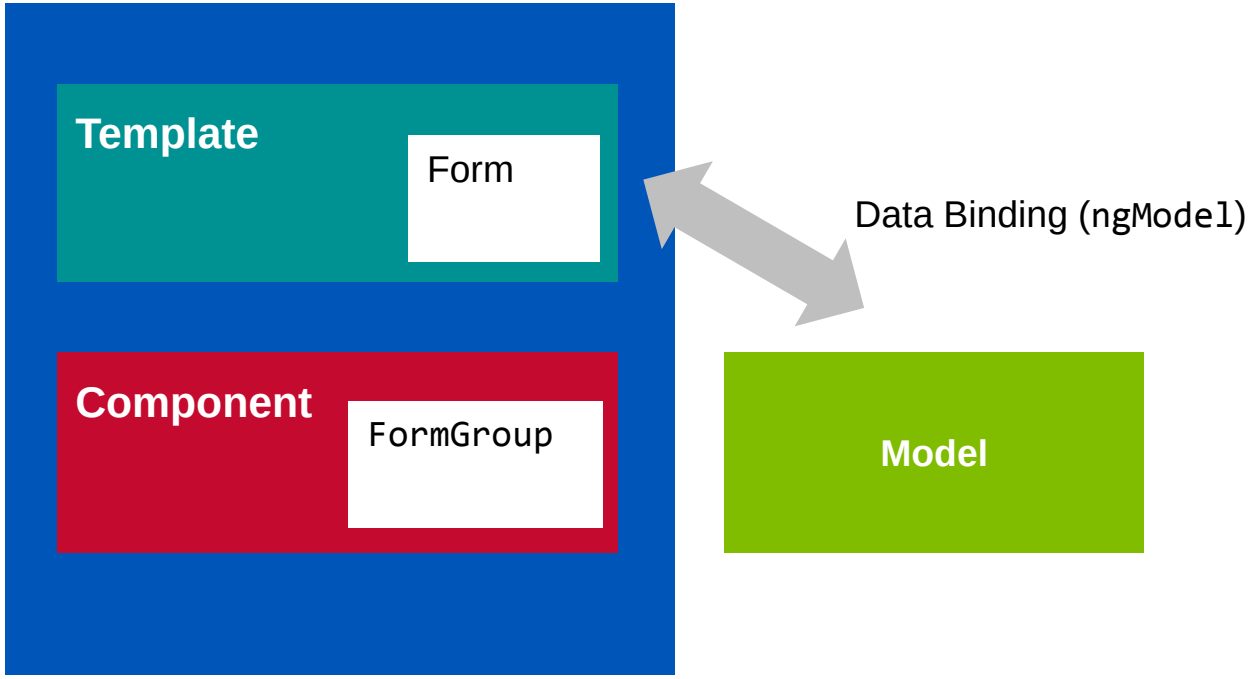
- Validation and Feedback
- Reactive Forms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Driven Form-Processing Architecture

- **In the template driven paradigm**
  - Form data is bound to a model by `ngModel`
  - Angular keeps the model and form synchronized



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Form Hosting Component

- **A normal Component template**
  - With the addition of a model property
- **The example has initial model data**

preferences-form.component.ts

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

@Component({…
  templateUrl: 'preferences-form.component.html',
})
export class PreferencesFormComponent {

  model = new Preferences(  "Long John Silver", "Treasure
        Island", "F", "Knots", 10 );

}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Form Directives

- **Data is bound to the form model using the `NgForm` directive**
  - One of several directives used by forms
- **Specified in `@NgModule` by adding `FormsModule` to the `imports`**

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
],
  imports: [
    BrowserModule,
    FormsModule,
    routing
  ],…})
```

- **Some versions of Angular CLI add the `import` statements automatically**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

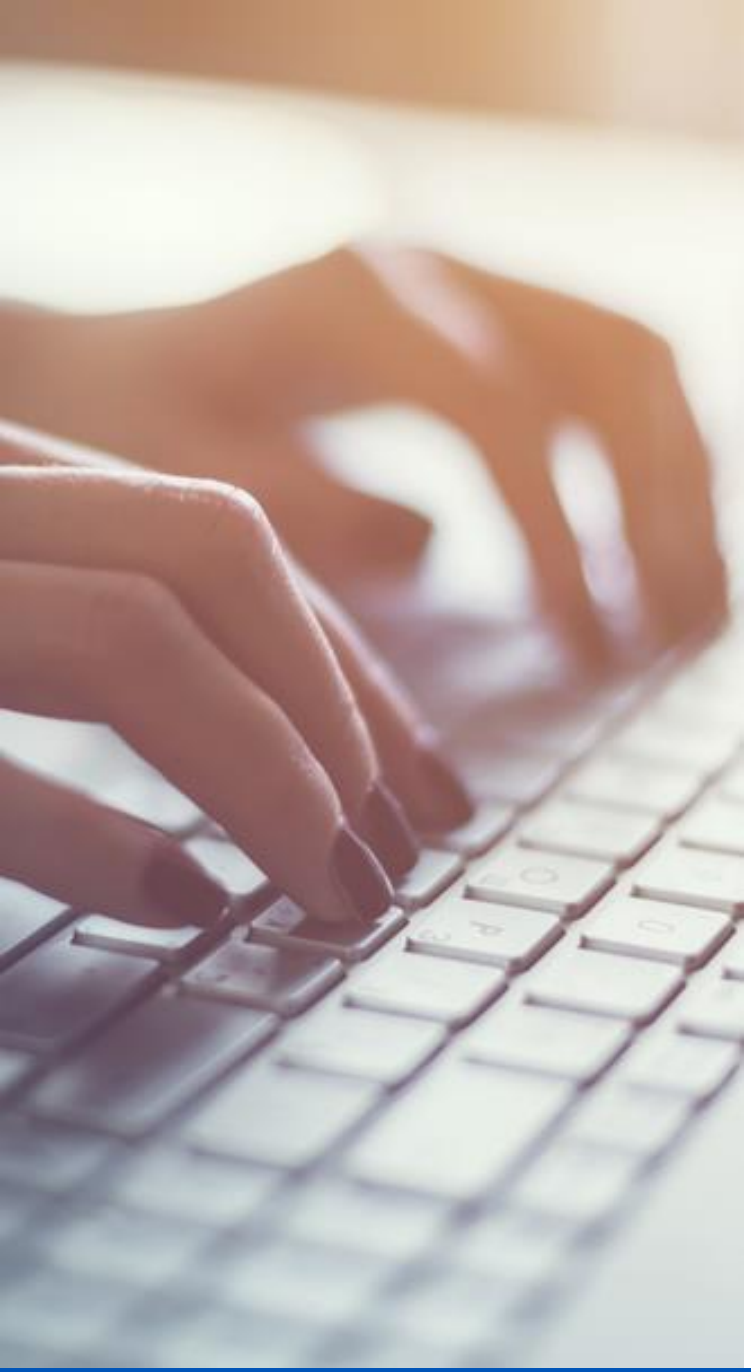
Binding to the Form Model

- **Binding is performed in the template**
  - Add attributes to the form input fields
- **Bind input field to model with `[(ngModel)]="model property name"`**
  - Two-way binding
  - Remember the "banana in a box" binding `[()]`
- **Bound form field elements *must* have a name attribute**

preferences-form.component.html

```
<input type="text" name="units" required
       [(ngModel)]="model.units" >
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



7.1: Form Bindings

Do Now

- In your Exercise Manual, please refer to Do Now 7.1: Form Bindings

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Populating Selection Data

- **Options for select controls may be populated from the model**
  - Use template directives to generate the `<option>` elements

preferences-form.component.html

```
<select … [(ngModel)]="model.temperatureUnits"
  name="temperatureUnits">
  <option *ngFor="let unit of temperatureUnits"
       [value]="unit">{{unit}}</option>
</select>
```

preferences-form.component.ts

```
temperatureUnits = ['F', 'C', 'K'];
```

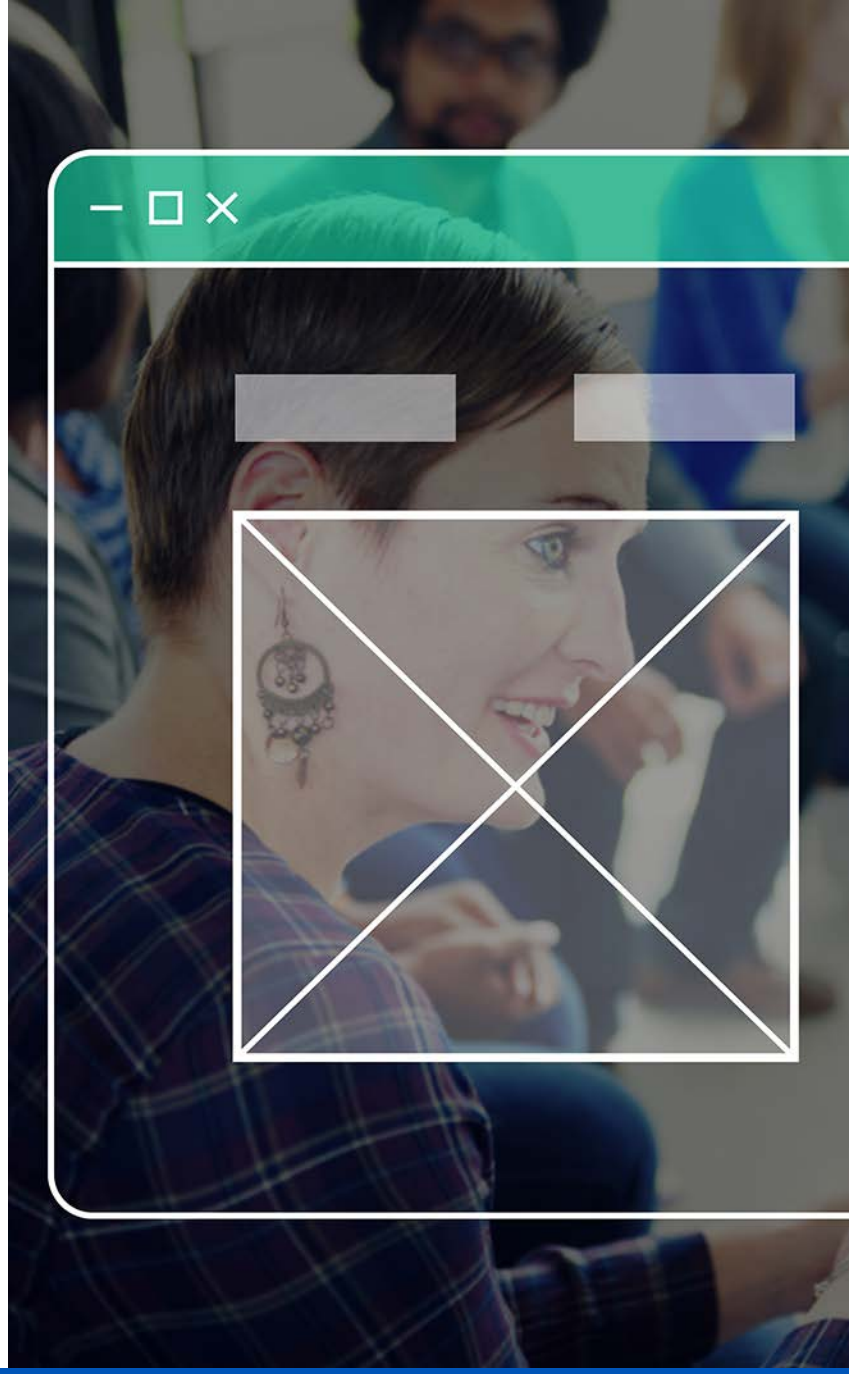
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Form Processing
- Template Driven Forms

Validation and Feedback

- Reactive Forms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Input Validation

- **Input validation assists users in creating the correct content**
- **`NgForm` provides mechanisms to give immediate feedback to users**
  - Through CSS styling
  - Through conditional content display
- **`NgForm` validations are built by processing the HTML 5 validation attributes**

Caution:
Input validation does not protect server-side applications from attack
- There are many tools and techniques that attack servers directly
  - Bypassing client validation

See also Learning Tree Course 940 Securing Web Applications, Services, and Servers

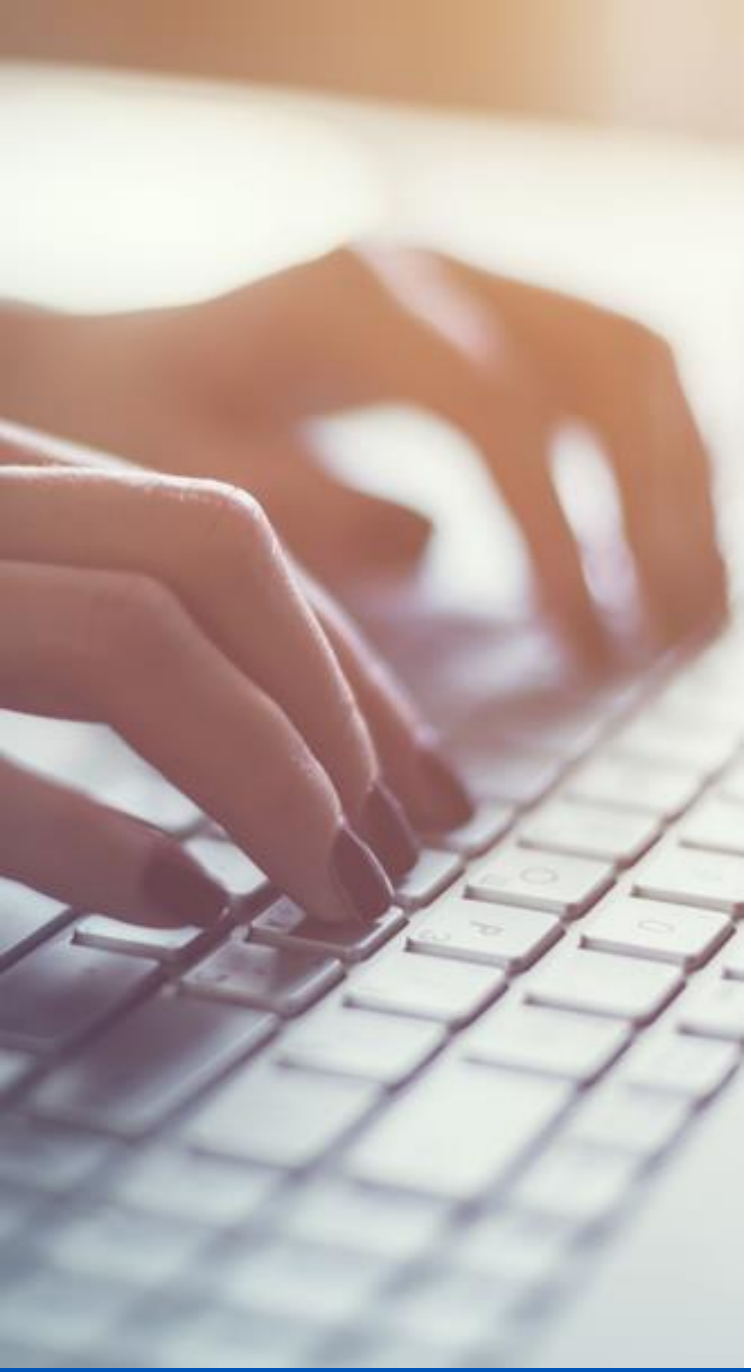
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

CSS Validation Feedback

- **The built-in `ngControl` directive automatically adds CSS classes to controls**
  - Based upon the current validation state

State	Class	
	State True	State False
Control touched	`ng-touched`	`ng-untouched`
Value changed	`ng-dirty`	`ng-pristine`
Control valid	`ng-valid`	`ng-invalid`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



7.2: CSS Validation Feedback

Do Now

- In your Exercise Manual, please refer to Do Now 7.2: CSS Validation Feedback

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Feedback Through Conditional Display

- **May be useful to have detailed feedback messages**

```
<input type="text" class="form-control" required
  [(ngModel)]="model.name" name="name">
<div>To make this valid you must…</div>
```

- **The `ngModel` directive allows this**
  1. Associate a template local variable with the `ngModel` directive
  2. Use the variable to test the control state

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Conditional Display of Error Messages

1. Associate a template variable called `name` with `ngModel`

```
 <input type="text" class="form-control" required
  [(ngModel)]="model.name" name="name" #name="ngModel">
<div [hidden]="name.valid || name.pristine" >
        Name is required
</div>
```

2. Test `ngModel` state

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Form Submission

- **Form submission is handled by the `ngSubmit` directive**
  - Bound to `ngSubmit` attribute of the form element
- **`ngSubmit` invokes the specified handler when the form submit button is pressed**

```
<form (ngSubmit)="onSubmit()" #theForm="ngForm">

  <button type="submit">Submit</button>

</form>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

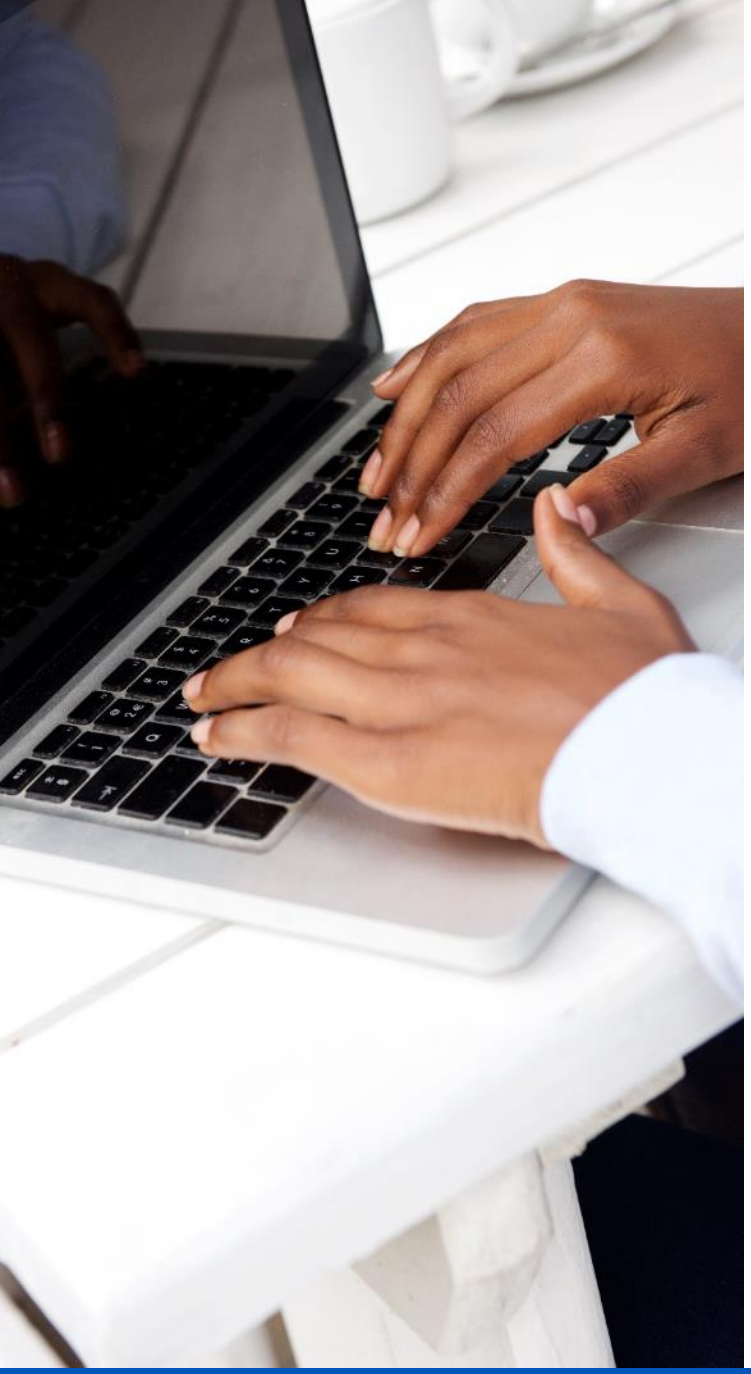
Testing Form Validity With `NgForm`

- **Angular automatically adds an `NgForm` directive to the `<form>`**
  - It is exported as `ngForm`
- **Binding a template variable to `ngForm` allows checking of the form state**
  - `ngForm.form.valid` is true if all the controls are valid

```
<form (ngSubmit)="onSubmit()" #theForm="ngForm">

<button type="submit" [disabled]="!theForm.form.valid">
Submit</button>

</form>
```

Disable the submit button unless the form is valid

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 7.1

Exercise Manual

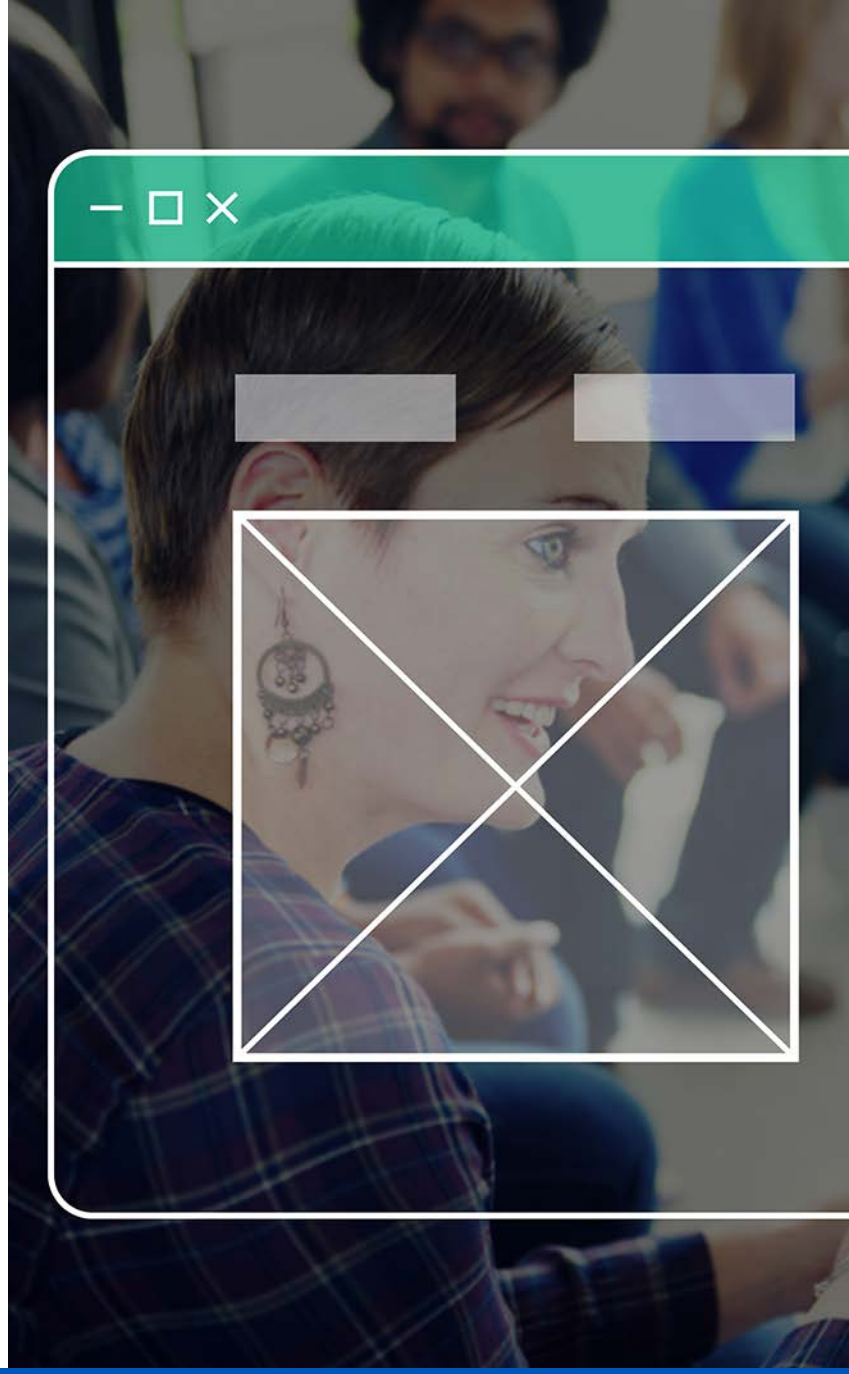
In your Exercise Manual, please refer to Hands-On Exercise 7.1: Implementing a Form With `ngModel`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Form Processing
- Template Driven Forms
- Validation and Feedback

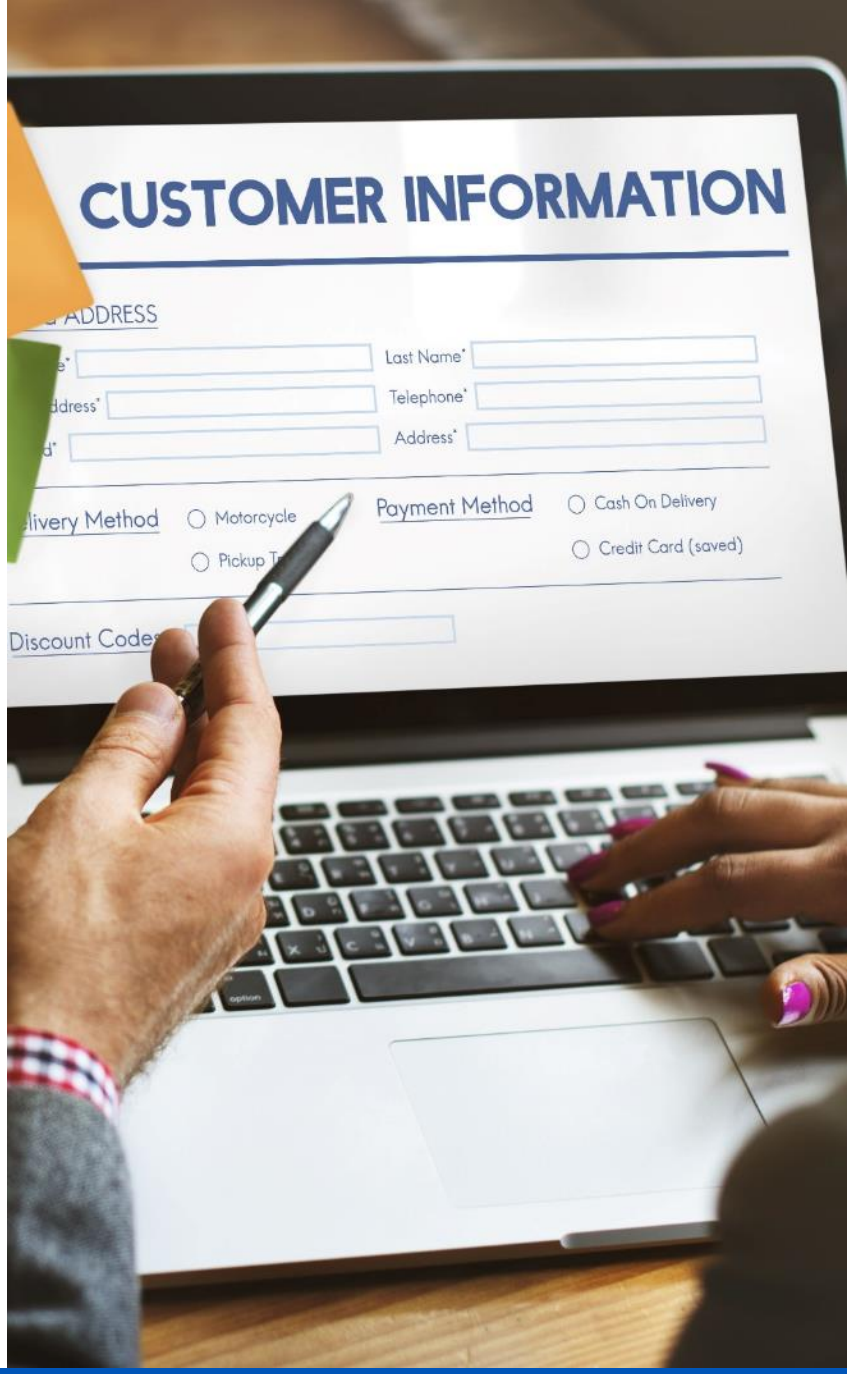
Reactive Forms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Reactive Forms

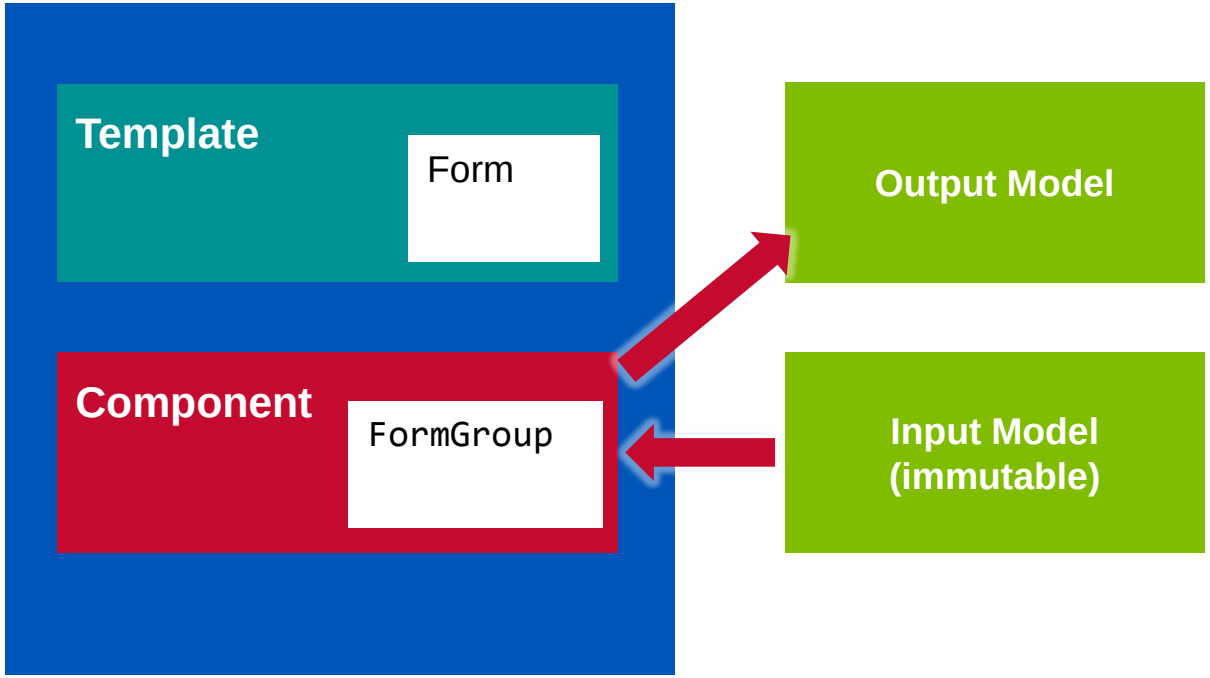
- **Reactive Forms provide an alternative mechanism for building forms**
  - Provides more control at the cost of increased complexity



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Reactive Form Processing Architecture

- **In the reactive form paradigm**
  - The `FormGroup` is created in the Component
    - Then associated with the form in the template
  - Code in the Component then
    - Populates the `FormGroup` from the model
    - Pulls data from the `FormGroup` to an output model



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

`FormBuilder`

When working with reactive forms:

- **`FormGroup` is created using `FormBuilder`**
  - Instead of automatically from the form template
- **`FormBuilder` increases control of form processing**
  - Moves validation from templates into code
- **Building reactive forms with `FormBuilder`**
  1. Create a `FormGroup` using `FormBuilder`
  2. Associate the `FormGroup` with a form template
  3. Provide feedback by interacting with the `FormGroup`
  4. Populate the `FormGroup` with data from the model
  5. Pull data from the `FormGroup` when form is submitted

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

1. Create a FormGroup With `FormBuilder`

1. **Module must import `ReactiveFormsModule`**
   - Can coexist with `FormsModule`
2. **`FormBuilder` is injected into the `Component` constructor**
3. **Use `FormBuilder` to create a `FormGroup`**

payment.component.ts

```
export class PaymentComponent {

  payForm: FormGroup;

  constructor(formBuilder : FormBuilder ){

    this.payForm= formBuilder.group({…})
  }
```

2

3

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Defining the Form

- `FormBuilder.group()` creates a `FormGroup`
- Argument is a collection of key/Control pairs

```
this.payForm=
      formBuilder.group({
      'name': […],
     'address': […],
     'email': […],
  …
});
```

Form field name

Parameter map to create a Control

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Defining the Controls

- **The parameter map for each control consists of**
  - A default value - used if there is no model
  - A validator
    - Specifying the validation to perform on the control
    - Built-in validators
      - `required`, `minLength`, `maxLength`, `pattern`
    - Custom validators may be created

```
… formBuilder.group({
  'name': ['', Validators.required],
  'address': ['', Validators.required],
  'email': ['', Validators.pattern("^[A-Za-z0-._...")],
        …
});
```

Default value—or populate from the model

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Validations

- **Multiple validations may be composed into a single validation**
  - For example, combining required and minimum length validations
- **Use `Validators.compose[validator, validator,…]` to create the validation**

```
… formBuilder.group({
  'name': ['',
       Validators.compose(
              [Validators.required,
               Validators.minLength(5)])],
  'address': ['', Validators.required],
  'email': ['', Validators.pattern("^[A-Za-z0-._...")],
       …
});
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

2. Associate the `FormGroup` With the Form Template

- **To use the `FormGroup`**
  - Use the `formGroup` directive to bind the form to the `FormGroup`
  - Use `formControlName` to bind each control to the corresponding property of the `FormGroup`
    - Replaces the `name="…"` attribute

```
<form [formGroup]="payForm" (ngSubmit)="onSubmit()">

   <div …>
     <label for="name">Name</label>
     <input type="text"
       formControlName='name'>
   </div>
…
```

- **Form controls must not use `ngModel` bindings**
  - Deprecated in Angular 6, removed in 7

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

3. Provide Feedback

- **The same CSS classes as in template-driven forms are available**
  - The controls are all using `ngControl`
    - `ng-touched`, `ng-dirty`, etc.
- **Controls may be accessed in the template via the `FormGroup`**
  - `formGroup.controls`
  - For example, `payForm.controls.email.valid`

```
<form [formGroup]="payForm"
   <div …>
     <label for="name">Name</label>
     <input type="text"
       formControlName='name'>
   </div>
   <div *ngIf="!payform.controls.name.valid">
       The name field is invalid because…
   </div>
```

Conditional error display

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

4. Populate the Form With Model Data

- **Reactive forms do not bind to the model**
- **Developer may populate the `FormGroup` data from the model**
- **`FormGroup` provides 2 methods**
  - `setValue(model : Object)` – model must match the fields in the `FormGroup`
    - Throws an error if there is not a one-to-one match
  - `patchValue(model: Object)` – populates those fields that match
    - Does not throw an error

```
model = new Payment(…);
…
this.payForm.setValue(this.model);
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

5. Pull Data From the `FormGroup`

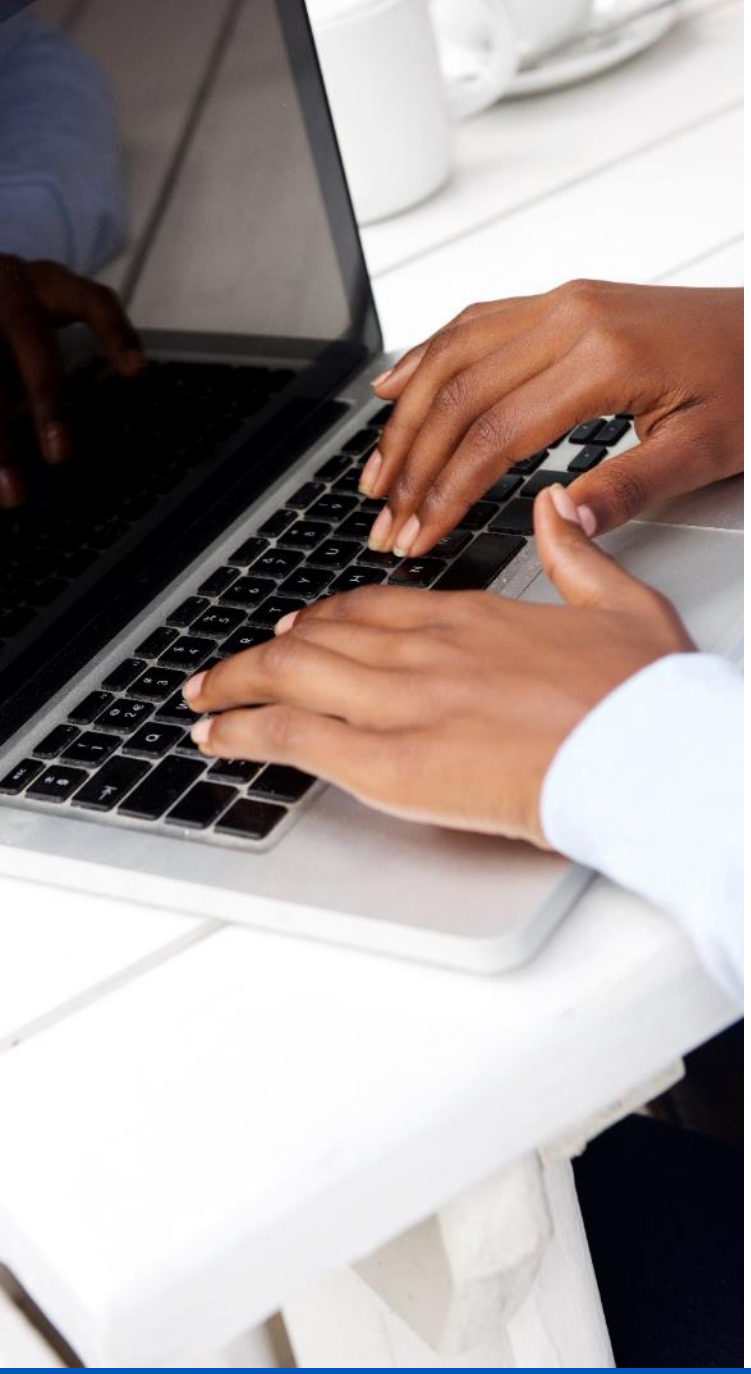
- **Data is typically extracted from the form when the submit button is pressed**
  - But can be done at any time
    - e.g., `ngOnChanges()`
- **The form data may be accessed via the `value` property of `FormGroup`**

```
const formData = this.payForm.value;
```

- **An object with properties which hold the current values of the form**

```
const address = formData.address;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 7.2

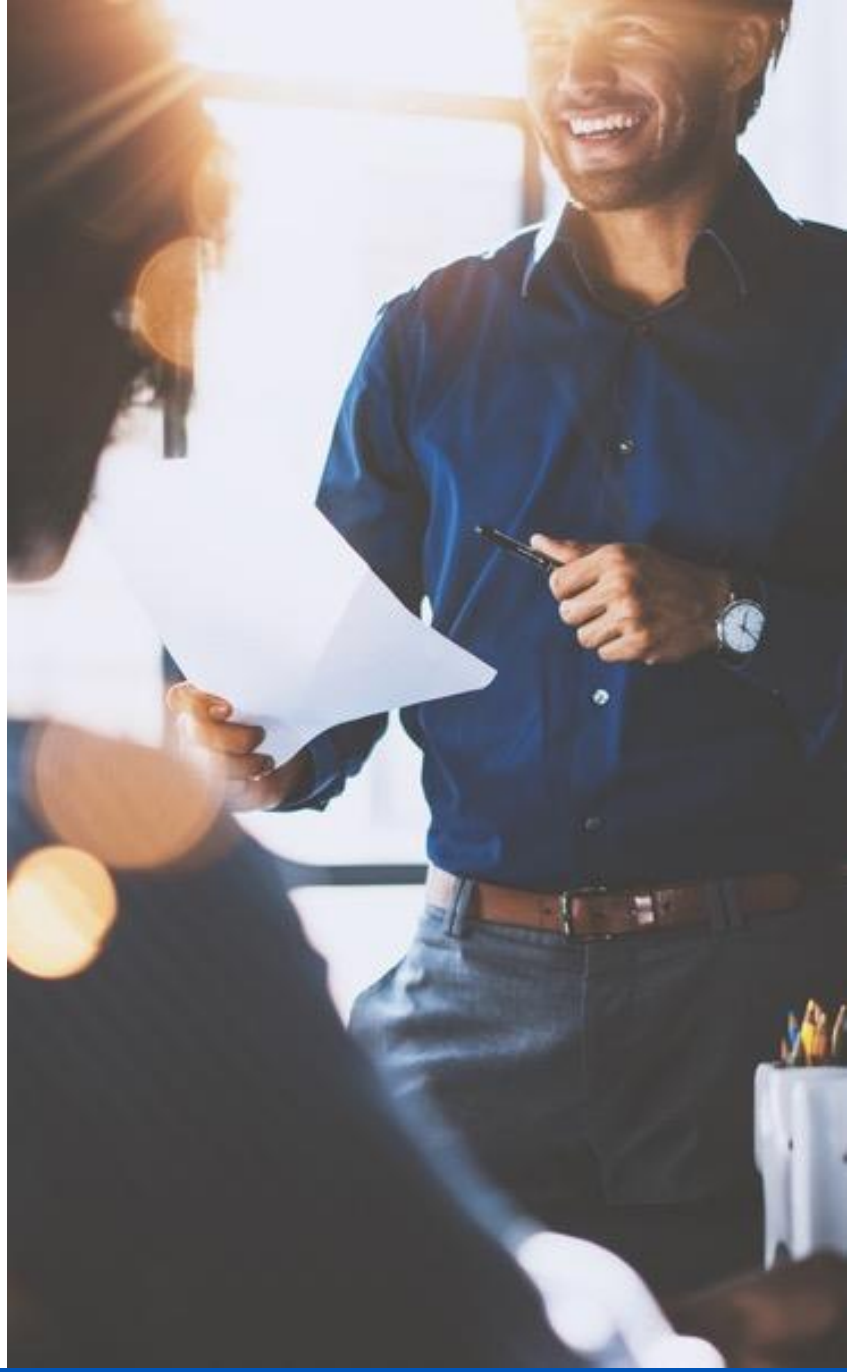
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 7.2: Creating Reactive Forms

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Define a form data model
- Build a template-driven form
- Provide validation and feedback
- Develop a reactive form



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

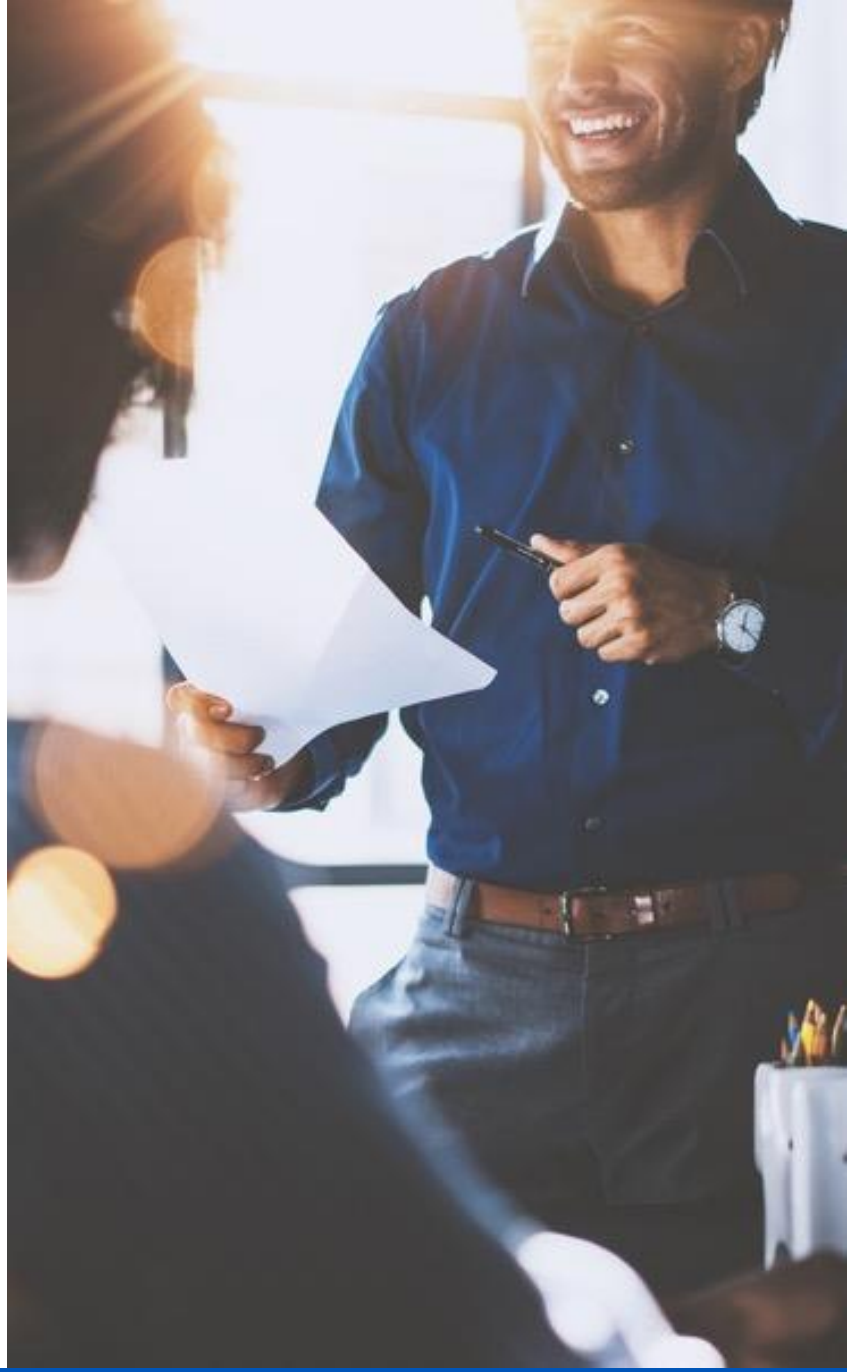
Chapter 8

Managing Asynchronous Behavior

LEARNING TREE INTERNATIONAL

Objectives

- Incorporate Promises and Observables
- Optimize Change Detection
- Communicate with REST APIs

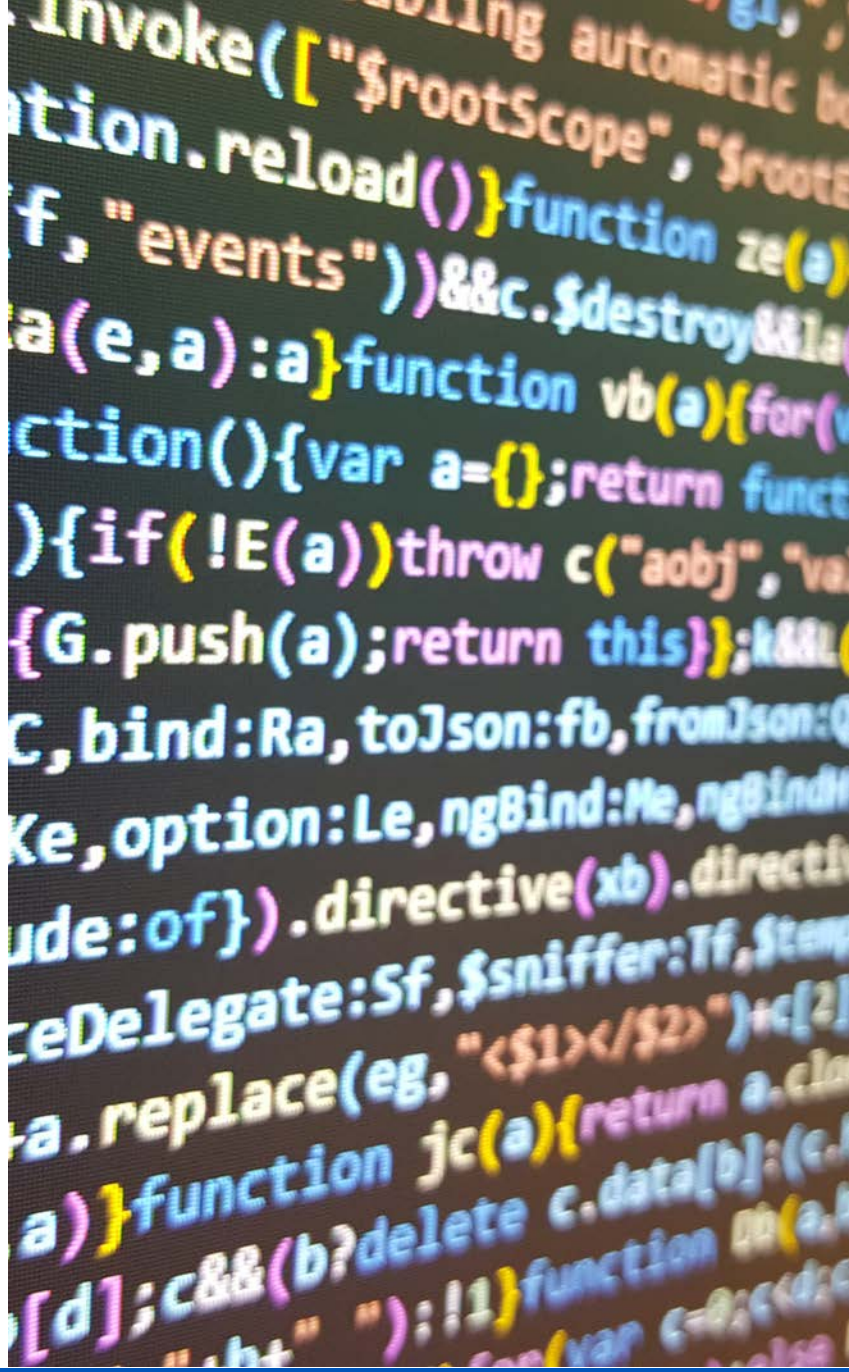


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Asynchronous Behavior

- Change Detection
- Web Service APIs
- Integrating the Http Client



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Asynchronous Behavior

- **Code that performs its work at some future time**
  - When triggered
- **Asynchronous triggers**
  - Timers
    - `setTimeout()`, `setInterval()`
  - DOM event handlers
    - `onClick()`…
  - `XMLHttpRequest` (XHR) object
    - Generates events when data is loaded or an error occurs
  - Geolocation events
  - Web Workers

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Approaches to Asynchronous JavaScript

- **Three options**
  - Callback functions
  - Promises
  - Observables

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Callbacks

- **Calling code requests behavior**
  - A timer in our example
- **Provides a function call when the behavior is complete**

```
setTimeout(function () {
    console.log('timeout 1');
    setTimeout(function () {
        console.log('timeout 2');
    }, 1000);
}, 1000);
```

- **Many limitations**
  - Complex nested code—the pyramid of doom
  - Try/catch cannot be used in the outer block
  - Handlers must be registered before the work is started

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Making Promises

- **Promises were developed specifically to handle XHR requests**
  - Adopted for many other cases
- **Promise consists of**
  - The work to perform
  - A call to the `resolve()` function if it worked
  - A call to the `reject()` function if it did not

```
this.promise = new Promise((resolve, reject) => {
  // do some work – perhaps create an HTTP connection
        …
  if (/* everything ok */) {
    resolve("Stuff worked!");
  }
  else {
    reject("Something failed");
  }
});
```

The code to do the work goes here

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Getting Results From a Promise

- **The work in a Promise starts as soon as the Promise is created**
  - Typically background work such as HTTP request
- **The results can be retrieved when we are ready for them**
- **Call `then()` on the promise**
  - Providing a function to call
    - Function is called as soon as results are ready

```
promise.then((msg)=>{console.log(msg);})
```

Function passed to `Promise` as resolve function

- **Call `catch()`to supply the reject function**

```
promise.then((msg)=>{console.log(msg);})
                .catch((err)=>{console.log(err);}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observables

- **A core component of Reactive programming**
  - Like Promises, they allow background activities to take place
- **Observables return data as a stream**
  - A collection that arrives over time
    - Like an array that is being gradually filled with data
- **Observables are lazy**
  - Work is performed only when a client subscribes
- **Observables can be canceled**
- **Observables clean up after themselves**
  - No need to deregister event handlers

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Observables

- **Observables constructor takes a function**
  - Single parameter—an observer
- **Observer performs its work**
  - Sends results to observer by calling `observer.next(result)`

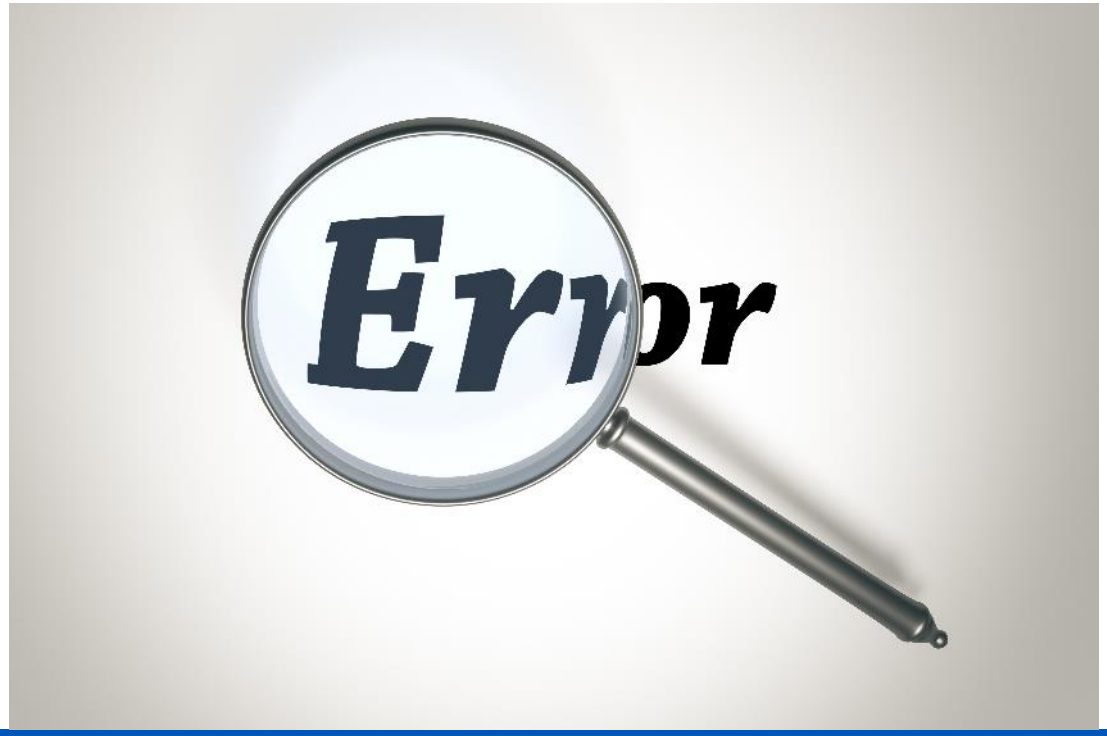
```
 createObserver() {
    this.data = new Observable((observer) => {
      let i=10;
      while(i--){
          setTimeout(() => {
              observer.next(i);
          }, 2000);
      }
      observer.complete();
    });
}
```

Send data to subscriber

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observable Errors and Completion

- **The Observable reports errors by calling `observer.error( err: any)`**
- **May indicate completion by calling `observer.complete()`**
- **`error()` or `complete()` action is the last thing to be observed**
  - The Observer and Subscription are both terminated



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observing Observables

- **Observers subscribe to the Observable**
  - Provide functions to call
    - When data is available
    - When errors occur (optional)
    - When the Observable is finished (optional)

```
let subscription = this.data.subscribe(
     (value) => this.values.push(value),
     (error) => this.errors += " " + error,
     () => this.done = true
   );
```

Data handler

Optional error handler

Optional completion handler

- **The subscription should be terminated**
  - `subscription.unsubscribe()`

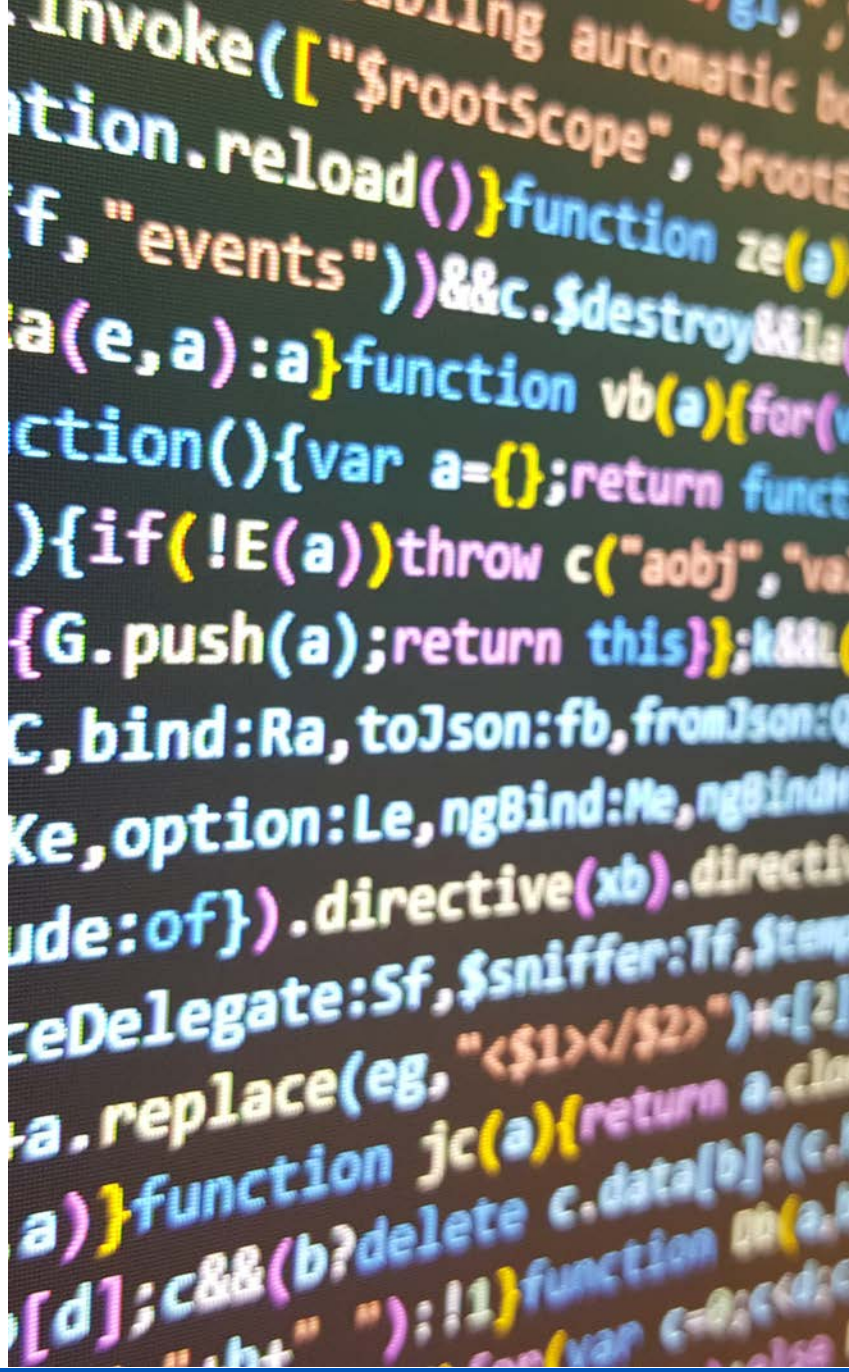
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Asynchronous Behavior

Change Detection

- Web Service APIs
- Integrating the Http Client



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

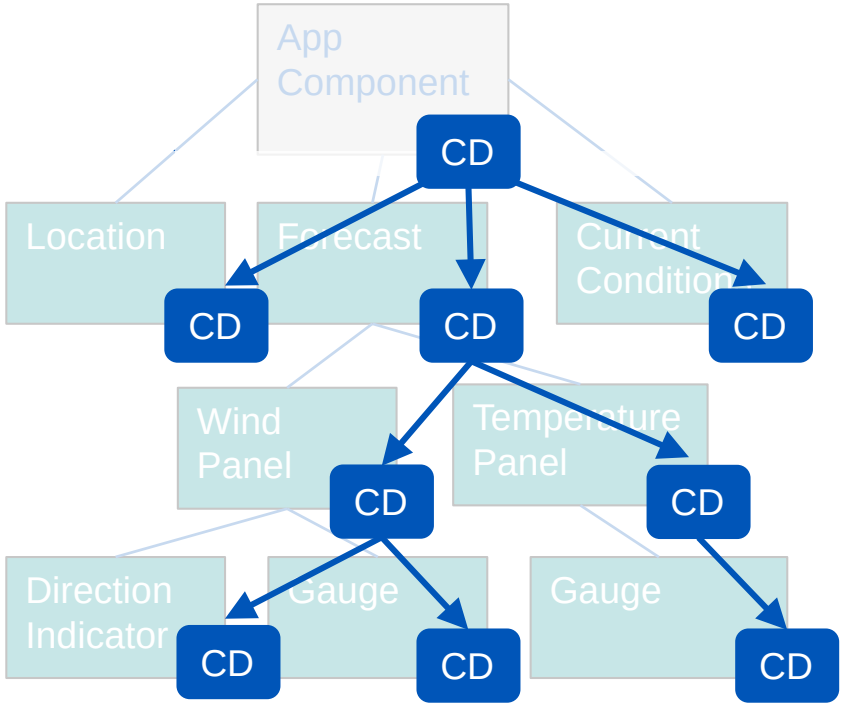
Updating the View

- **Angular maintains a Component tree**
- **Each Component has a state—the model**
  - Which is reflected in the view
- **When the state of a Component changes**
  - Angular must update the view
    - Re-create all or part of the HTML DOM
- **Updating the entire view for every change could be very slow**
  - Resulting in poor response and a flickering display
- **Angular has a change detection strategy**
  - Identifies what has changed and updates the display

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Change Detection

- **Angular automatically checks state periodically**
- **Checks for change**
  - Working top-down through the Component tree
  - Interrogates each Component's change detector
- **The Component change detectors are built automatically by Angular**
- **If any Component reports a change**
  - Angular updates the view



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Optimizing Change Detection

- **Default behavior**
  - Check every Component for any change
  - May give poor performance
    - Large Component trees on slow machines
- **Default change detection checks the content of every variable**
  - Examines the detailed structure of every object referenced from a Component
- **Change detection can be tuned**
  - Specify an alternative detection strategy
  - Add `changeDetection: ChangeDetectionStrategy.OnPush` to the Component metadata
- **`OnPush` changes the strategy to look for changes to object references**
  - Checks the values of the variables in the Component
    - Does not check the values of the variables referenced
- **The `OnPush` strategy applies to all Components in the hierarchy below the one where it is applied**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Example of Change Detection

- **Changes to object references are always detected**

```
private date : Date;
…
doSomething(){
  this.date = new Date();
}
```

Object reference changed

- **Changes to the content of objects are only detected with default change detection**
  - Not when the strategy is `OnPush`

```
private date : Date;
…
doSomething(){
  this.date.setYear(1969);
}
```

Content object changed

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Notifying Angular of Change

- **Data arriving at Observables is not detected when strategy is `OnPush`**
  - Because the Observable's reference does not change
- **Solution: Subscription handler code must notify Angular of change**
  - Inject a `ChangeDetectorRef`
  - Call `markForCheck()` to notify Angular that the Component should be fully checked

```
constructor(
  private weatherService : WeatherService,
  private cd: ChangeDetectorRef){}
…
  this.weatherService.
        updatePreferences(this.model).subscribe(
        (id : number)=> {theId = id;
              this.cd.markForCheck()},
             (err)=> this.error = err
        );
```

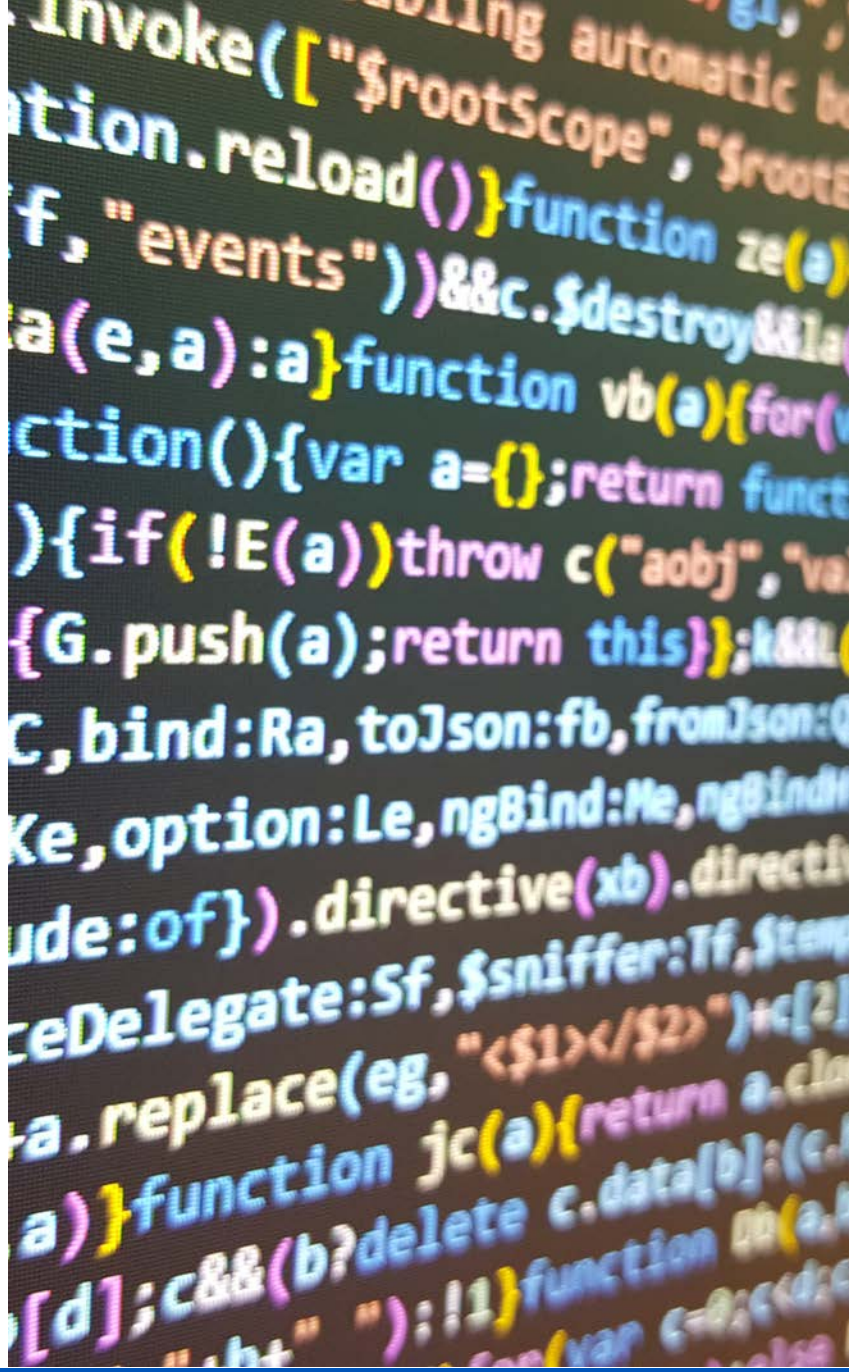
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Asynchronous Behavior
- Change Detection

Web Service APIs

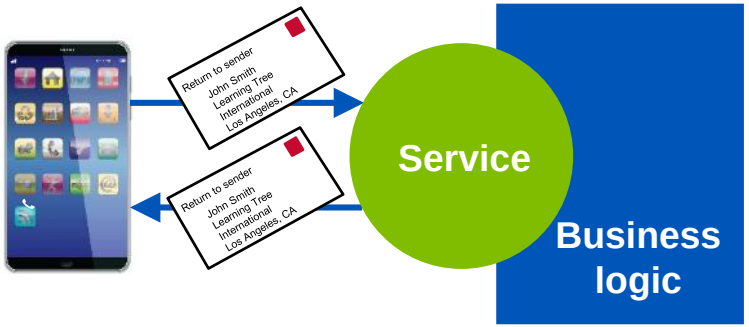
- Integrating the Http Client



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Integrating With the Server

- **Application data is commonly provided by a server-side application**
- **Communicated to the application via web services**
  - Also known as web APIs



- **Representational State Transfer (REST) services**
  - Lightweight, informal services
  - Data typically in JSON format
  - Easy to consume from JavaScript-based applications
- **Web services communicate over HTTP**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Communicating Using HTTP

- **HTTP is the primary protocol for communicating on the web**
  - TCP/IP–based request/response protocol
- **Basic HTTP communication cycle**
  1. Client opens an IP connection to the server
  2. Client sends an HTTP request
  3. Server returns a response
  4. IP connection closed

- **HTTP request methods define behaviors**
  - `GET`: requests data from the server
  - `POST`: sends data to the server—analogous to a create
  - `PUT`: sends data—update
  - `DELETE`: deletes the specified resource

TCP/IP = Transmission Control Protocol/Internet Protocol

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

JSON Format Overview

- **JSON is widely used for REST services**
  - Concise
  - Easy to process
- **Objects**
  - Objects begin and end with curly brackets (`{}`)
- **Object members**
  - Members consist of strings and values
  - Separated by a colon (`:`)
  - Members are separated by commas
- **Arrays**
  - Begin and end with brackets (`[]`) and contain values
  - Values are separated by commas

```
[{"dateTime":1467820008052,
  "skyConditions":"clear",
  "airPressure":1021.0,
  "windSpeed":18.0,
  "gustSpeed":24.5},
{"dateTime":1467820008052,
  "skyConditions":"clear",
  "airPressure":1029.0,
  "windSpeed":8.0,
  "gustSpeed":4.5}]
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Building a Web Service Client

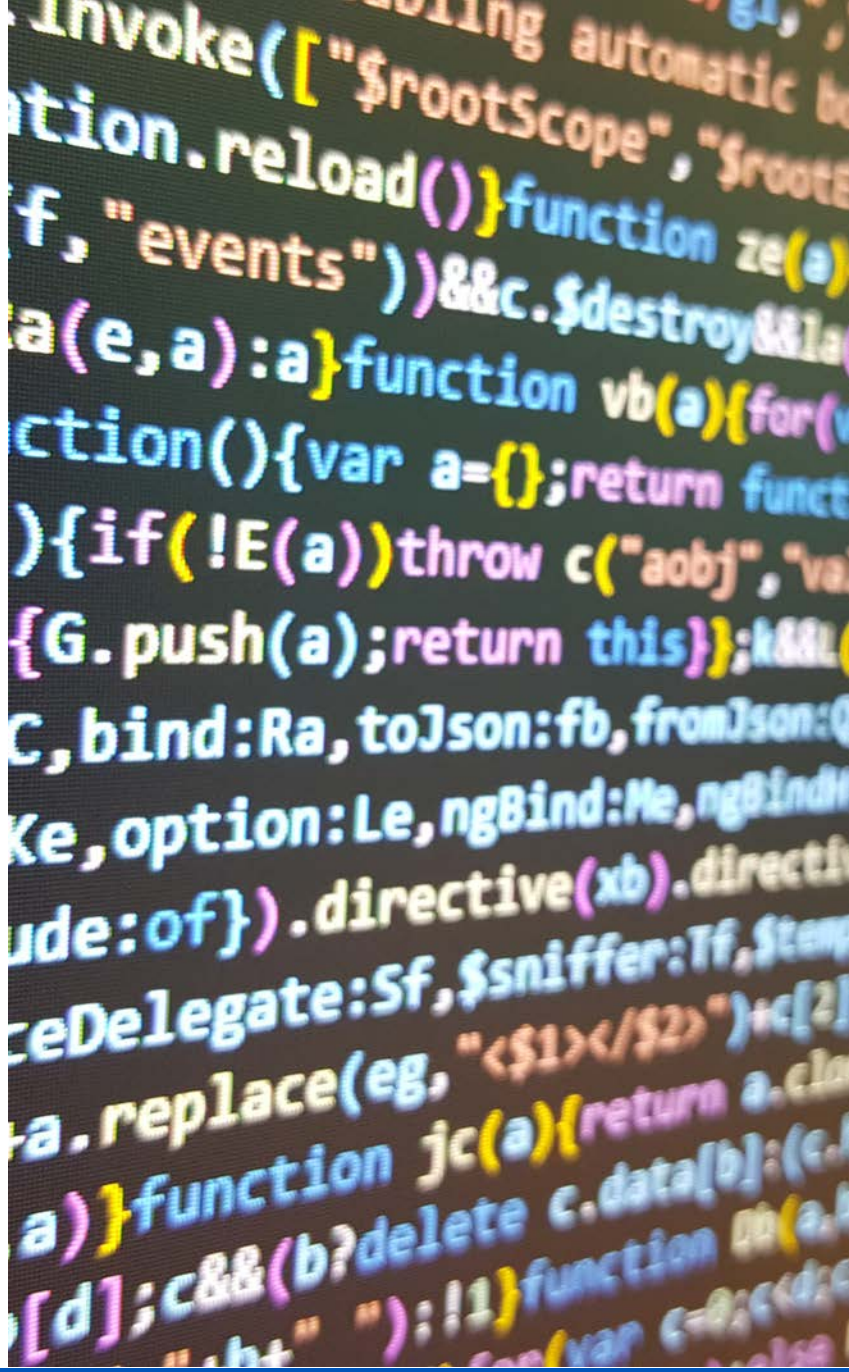
- **To communicate with a web service**
  - Create an HTTP connection
  - Create a request from application data
  - Send the request
  - Parse the results
- **The Angular `HttpClient` class provides extensive support**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Asynchronous Behavior
- Change Detection
- Web Service APIs

Integrating the HTTP Client



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Fetching Data From the Server

- **Weather server returns weather data in response to a `GET`**
  - `http://localhost:8080/weather/latest`
- **HTTP access code should be placed in a service class**

1. **Provide HTTP support**
2. **Inject `HttpClient` into service**
3. **Make the `GET` request**
4. **Convert the response to domain objects**
5. **Handle errors**
6. **Subscribe to the result**


```
[{"dateTime":1467820008052,
  "skyConditions":"clear",
  "airPressure":1021.0,
  "windSpeed":18.0,
  "gustSpeed":24.5},
{"dateTime":1467820008052,
  "skyConditions":"clear",
  "airPressure":1029.0,
  "windSpeed":8.0,
  "gustSpeed":4.5}]
```


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

1. Provide HTTP Support

- **HTTP support is provided by `HttpClientModule`**
  - Must be declared in the `@NgModule imports`

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent, …],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    routing
  ],
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

2. `HttpClient` and Observable Support

- **Inject `HttpClient` into the constructor of the service**
- **HTTP code makes extensive use of Observables**
  - Provide imports for Reactive Extensions Observables
    - Avoid importing all of `rxjs`—it is huge!

weather.service.ts

```
import {Observable, throwError} from "rxjs";
import {catchError} from "rxjs/operators";

constructor(private http: HttpClient) {}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

3. Make the GET Request

- **Call `get()` on the `HttpClient` object**
  - Pass the URL as a parameter
- **Returns an `Observable<any>`**
- **Use `catch()` to specify the method to call if there is an error**

weather.service.ts

```
getWeather() {
  let url = "http://localhost:8080/weather/latest";

  let res : Observable;
  res = this.http.get(url)
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The `Get` Method Return Type

- **The response from `HttpClient` methods that return data is**
  - `Observable<any>`
- **There are multiple overloads that can be used to return different data types within the Observable**

```
get(url: string, options: …
```

`Options` object configures return types

- **Return types are configured by the `observe` and `responseType` properties of the `options` object**
- **Possible return types**
  - `string`
  - `ArrayBuffer`
  - `blob`
  - `HttpEvent`
  - `HttpResponse<any>`
- **See https://angular.io/api/common/http/HttpClient#get**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Get Method Overloads

- **The type of the Observable returned from the `get()` method is defined by the options object**
  - Properties formally documented here

Signature of `HttpClient.get()`

```
get(url: string,
  options: {
    headers?: HttpHeaders | {
        [header: string]: string | string[];
    };
    observe?: 'events' | 'body' | 'response';
    params?: HttpParams | {
        [param: string]: string | string[];
    };
    reportProgress?: boolean;
    responseType?: 'arraybuffer' | 'blob' |
                                   'json' | 'text';
    withCredentials?: boolean;
}): Observable<any>
```

`observe` property

`responseType` property

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Basic `Get` Method Overloads

Reference

- **Return type of `HttpClient` `get` method determined by a combination of**
  - `Observe` (type of observable)
  - `Type` (data type of actual response)

observe	responseType	Return type
'body'	'arraybuffer'	Observable<ArrayBuffer>
'body'	'blob'	Observable<Blob>
'body'	'text'	Observable<string>
'events'	'arraybuffer'	Observable<HttpEvent<ArrayBuffer>>
'events'	'blob'	Observable<HttpEvent<Blob>>
'events'	'text'	Observable<HttpEvent<string>>
'response'	'arraybuffer'	Observable<HttpResponse<ArrayBuffer>>
'response'	'blob'	Observable<HttpResponse<Blob>>
'response'	'text'	Observable<HttpResponse<string>>

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

HttpEvent and HttpResponse

Reference

- **On the previous slide, response may be wrapped in:**
- **`HttpResponse`**
  - Provides full access to the `HttpClient` response object
  - Properties include
    - `body`
    - `type`
    - `headers`
    - `status`
    - `statusText`
- **`HttpEvent`**
  - Union type for all possible events on the response stream
    - `HttpResponse`
    - `HttpSentEvent` (Indicates request sent to server)
    - `HttpHeaderResponse` (Headers, but no body)
    - `HttpProgressEvent` (Progress)
    - `HttpUserEvent` (Base class for user defined events)

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Get Methods for JSON Data

- **If the `responseType` is `'json'` the data is treated as an object**

```
get(url: string, options: {…}) : Observable<Object>
```

- **Return type may be specified using a generic type indicator**

```
get<T>(url: string, options: {…}) : Observable<T>
```

- **The value of observe still indicates if the returned data should be wrapped**
  - By `HttpEvent` or `HttpResponse`

observe	responseType	Return type
'body'	'json'	Observable<Object>
'event'	'json'	Observable<HttpEvent<Object>>
'response'	'json'	Observable<HttpResponse<Object>>

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

4. Convert the Response to Domain Objects

- **The JSON data from `Http.get()` should be converted to domain objects**
  - Weather objects in our example
- **Use the generic version of `HttpClient.get<T>()`**
  - Returns an Observable that holds the specified type

```
public getWeather() : Observable<Weather[]>{
        return this.http.get<Weather[]>(url);
}
```

Defaults for the options object (2nd parameter) are `observe: 'body', type: 'json'`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

5. Handle Errors

- **Use the `Observable.pipe(fn)` to append an error handler to the `Observable` chain**
- **The function specified as the parameter to `catchError()`**
  - Will be invoked if an error occurs

```
"
getWeather() : Observable<Weather[]>; {
  let url = "http://localhost:8080/weather/latest";
  return this.http.get(url)
        .pipe(catchError(this.handleError));
}
private handleError (error: HttpErrorResponse )  {
       // Handle the error conditions
       // See next slide
}
"
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Error Handler

- **`HttpErrorResponse` contains the error information**
- **If `HttpErrorResponse.error` is an `ErrorEvent`**
  - Indicates a client side error
- **Otherwise assume it is a server-side error**
- **Error handler must return an `ErrorObservable`**

```
"private handleError (error: HttpErrorResponse ) {
    if(error.error instanceof ErrorEvent){
      // Client error
      console.error('Http communication error:', error.error.message)
    } else {
      // Server error
      console.error(`Server error: ${error.status}. Message body:
${error.error}`)
    }
    return throwError( 'Server error - is the REST server running?');
  }
"
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

6. Subscribe to the Result

- Code so far has converted the HTTP response to `Observable<Weather[]>`
- Consuming code subscribes to the `Observable` to extract domain data

```
weather : Weather[];
error: any;

ngOnInit() {
  this.weatherService.getWeather().subscribe(
    (weather : Weather[])=>this.weather = weather,
    (error : any)=>this.error = error);
}
```

- Important: The request will not be sent until `subscribe()`is called

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sending Data to the Server

- **Simple data may be sent to the server as query parameters**
  - Name = value pairs appended to the URL
  - `http://localhost:8080/weather/latest?location=Newcastle`
- **Appending values can be achieved**
  - Using string concatenation
  - With `URLSearchParams`

```
let weatherParams = new HttpParams();
weatherParams.set('location', location);
weatherParams.set('units', 'C');

return this.http.get<Weather[]>(url, {params: weatherParams
});
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Posting Data to the Server

- **Object data should be sent using `HTTP Post`, `Put`, or `Patch`**
  - Confidential data should also be sent using these methods
  - `POST` for create operations
  - `PUT` or `PATCH` used for update operations
- **Using `POST` is similar to `GET`**
  - Requires extra data
- **`HttpClient.post()` takes three parameters**
  1. `url`—the URL to send the request to
  2. `body`—the data to send —usually JSON
     - Create using `JSON.stringify()`
  3. `options`—configuration for the `HTTP POST`
     - Must provide an HTTP header of `Content-Type`
       - Use `application/json` for JSON data
     - May provide other HTTP headers such as `Accepts`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Posting Data to the Server

```
savePreferences(preferences: Preferences ) :
                                          Observable<string> {
 let url = "http://localhost:8080/weather/preferences";      1
 let body = JSON.stringify( preferences );      2
 let headers =
     new HttpHeaders({ 'Content-Type': 'application/json' });
 let options = { headers: headers };      3

 return this.http.post<string>(url, body, options)
       .pipe(catchError(this.handleError));
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

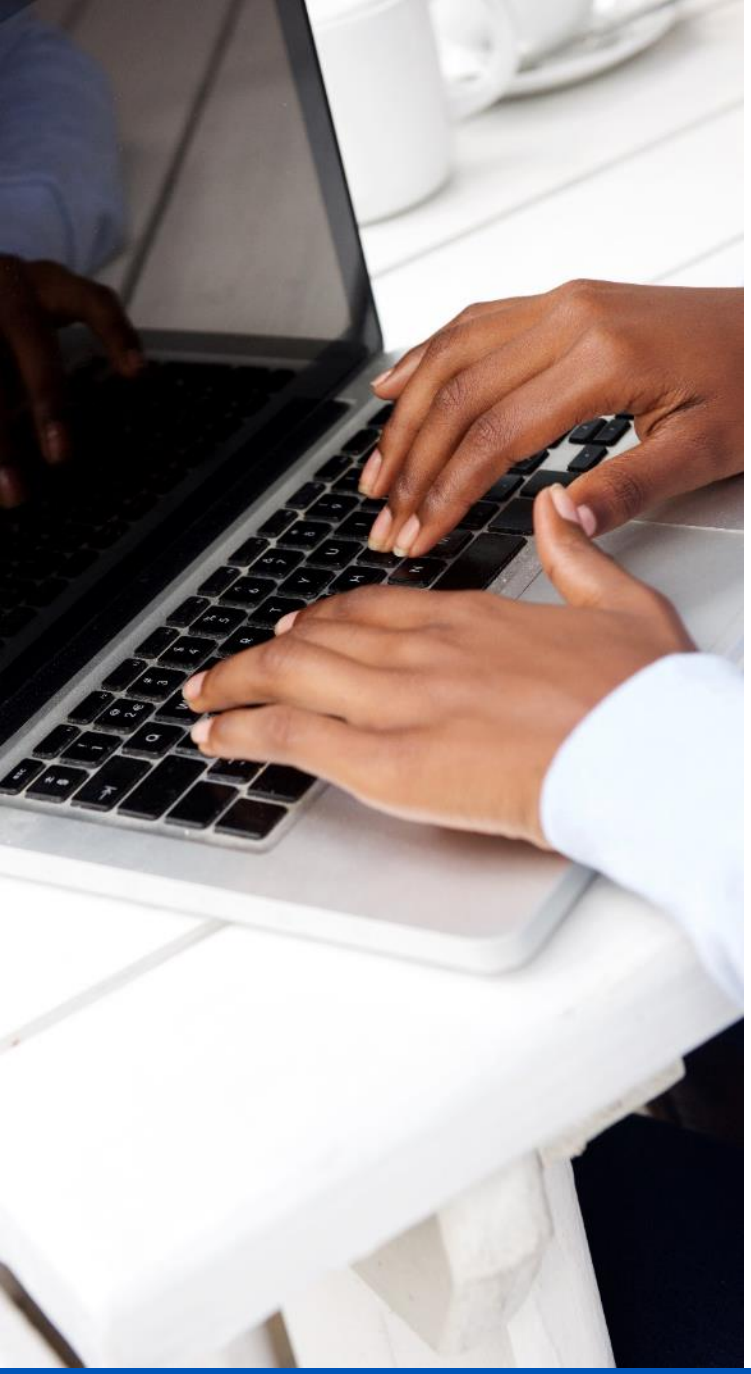
Other HTTP Methods

- **`put()` and `patch()` take the same arguments as `post()`**
- **`delete()` does not have a body**
- **The code is very similar in all cases**

```
post(url: string, body: any,
        options?: Object) : Observable<Response>

put(url: string, body: any,
        options?: Object) : Observable<Response>

delete(url: string,
        options?: Object) : Observable<Response>

patch(url: string, body: any,
        options?: Object) : Observable<Response>
```

- **There is no explicit type declaration for `options`**
  - It is described separately for each method

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 8.1

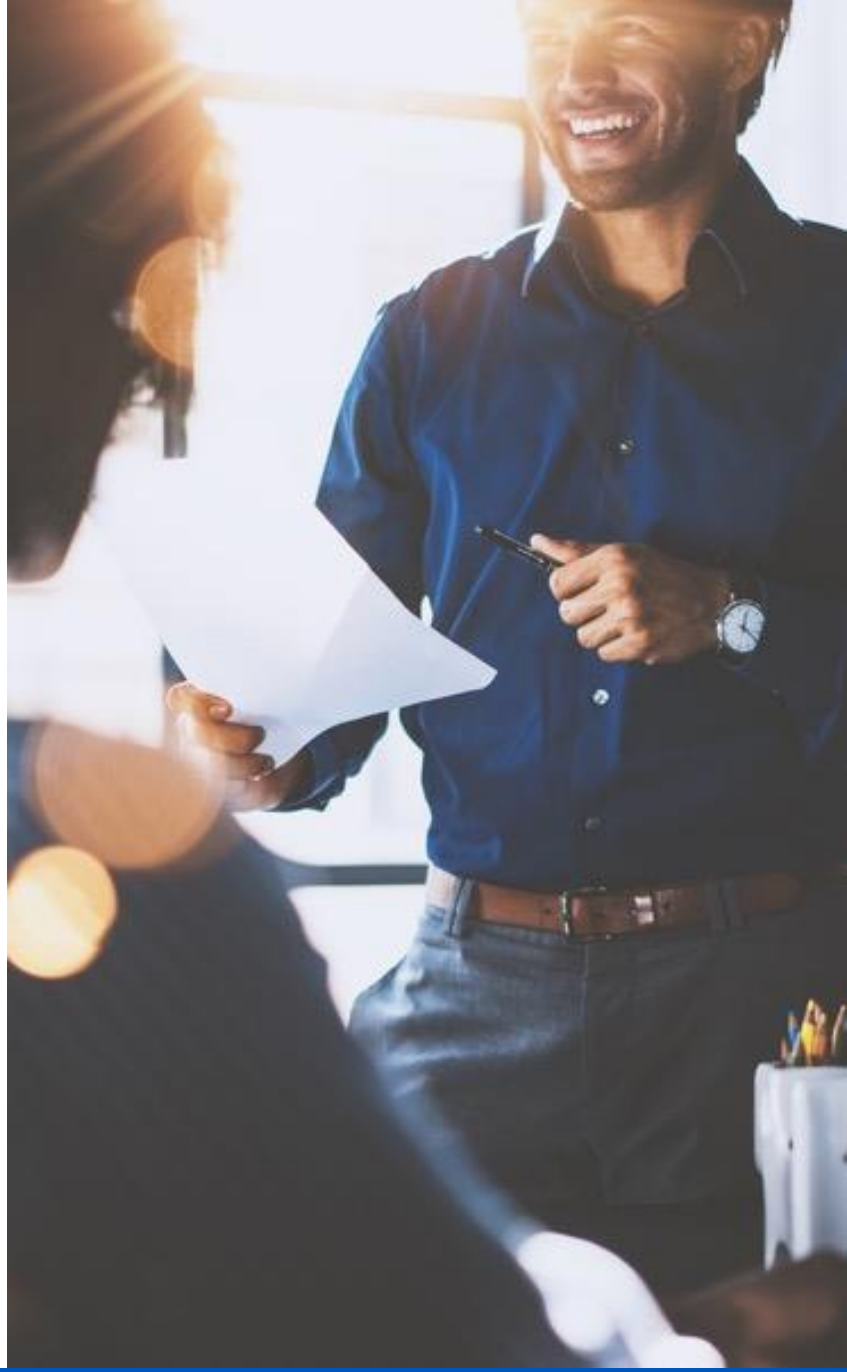
Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 8.1: Communicating With a REST Server

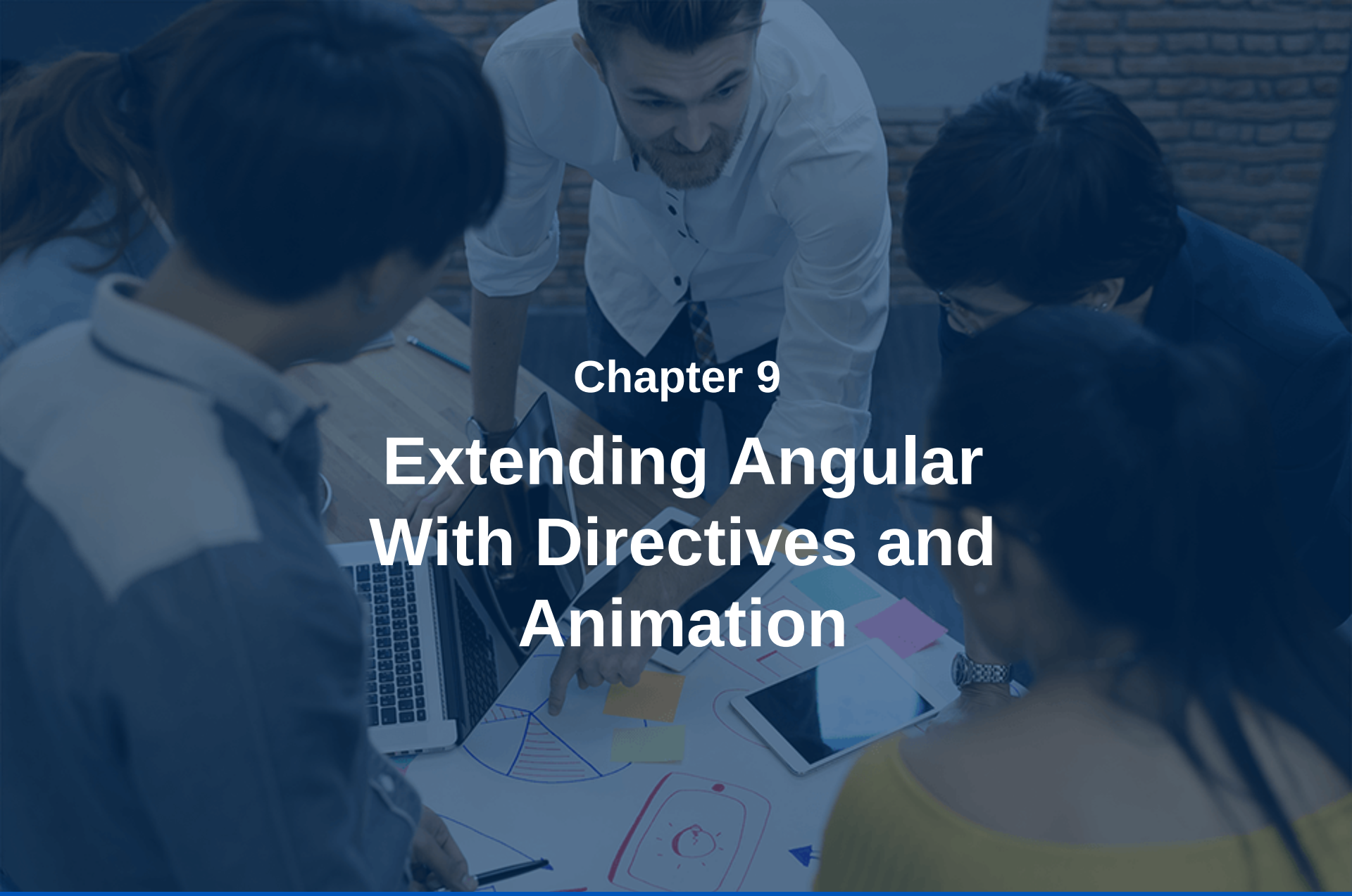
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Incorporate Promises and Observables
- Optimize Change Detection
- Communicate with REST APIs



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



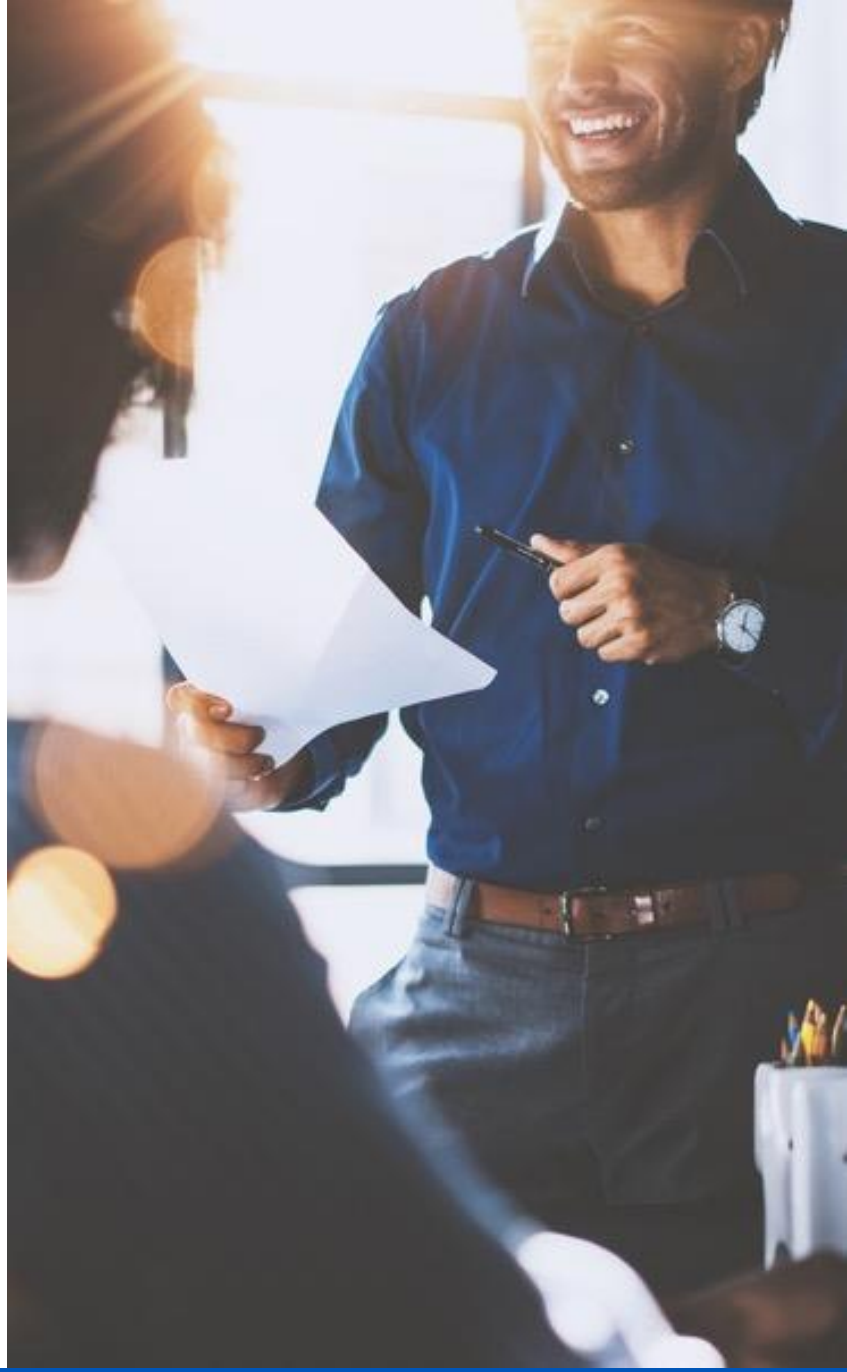
Chapter 9

Extending Angular With Directives and Animation

LEARNING TREE INTERNATIONAL

Objectives

- Differentiate directive types
- Build attribute directives
- Investigate structural directives
- Implement an animation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Attribute Directives

- Structural Directives
- Animation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Directives

- **Angular has three types of directives**
- **Components**
  - We've used extensively
- **Attribute directives**
  - Change the appearance or behavior of a DOM element
- **Structural directives**
  - Modify the DOM structure
    - Add or remove elements

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Attribute Directives

- Functionality applied when a matching attribute is present in a template
- Example: An attribute directive to match the attribute `MakeItBold`

```
<p MakeItBold>Some bold text</p>
```

- A directive is a class with an `@Directive()` decorator

```
@Directive({
  selector: '[MakeItBold]'
})

export class MakeItBold {
  constructor(el: ElementRef) {
    el.nativeElement.style.fontSize = '2em';
    el.nativeElement.style.fontWeight= 'bold';
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Attribute Directive Details

- **The selector property specifies a CSS selector matching the attribute**
  - CSS selectors in `[ ]` match attributes
- **Angular injects a reference to the DOM element `ElementRef`**
- **`ElementRef.nativeElement` provides access to the corresponding DOM element**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Applying the Attribute Directive

- **Two steps**
  1. Add the attribute to the template of a Component

```
<p MakeItBold>Some bold text</p>
```

  2. Add the directive's class name to the declarations property of the `@NgModule`

```
declarations: [MakeItBold]
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Binding Attributes

- **Attribute directives support binding**
  - Allowing configuration of directive behavior

```
<p [MakeItBold]="'2em'">Some bold text</p>
```

- **Directive must have an `@Input()` with a name matching the selector**
  - `MakeItBold` in our example

```
@Directive({
  selector: '[MakeItBold]'
})

export class MakeItBold {
  @Input()
  MakeItBold : string;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aliasing the `@Input`

- **The `@Input()` decorator can specify an alias**
  - Allowing the field name to fit better with the directive logic
- **The `@Input()` decorator can be assigned to a setter**

```
export class MakeItBold {

  constructor(private el: ElementRef) {
    el.nativeElement.style.fontWeight= 'bold';
  }

  @Input('MakeItBold')
  set fontSize(size : string){
    this.el.nativeElement.style.fontSize  = size;
  }

}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Attribute Directives

Structural Directives

- Animation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structural Directives

- **Angular has several built-in structural directives**
- **`ngIf`, `ngFor`, `ngSwitch`**
- **They manipulate the structure of the DOM**
  - Adding or removing DOM fragments as needed
- **Structural directives make use of an HTML5 element `<template>`**
  - A wrapper for an HTML fragment that is not initially rendered
    - Script in the page may display it

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Operation of `ngIf`

- `ngIf` is normally used as `*ngIf="some condition"`

```
<div
*ngIf="contentReady">
  {{content}}
</div>
```

- Angular interprets this as

```
<template [ngIf]="contentReady">
  <div>
    {{content}}
  </div>
</template>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Implementation of `ngIf`

- **Simplified version of `ngIf`**
- **Similar to the attribute directive**
  - Has an `@Input` of `ngIf`

```
@Directive({selector: '[ngIf]'})
export class NgIf {
 constructor(private _viewContainer: ViewContainerRef,
              private _templateRef: TemplateRef<Object>){}
 @Input()
 set ngIf(newCondition: boolean) {
  if (newCondition){
       this._viewContainer
              .createEmbeddedView(this._templateRef);
  } else {
       this._viewContainer.clear();
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Operation of `ngIf`

- **Two references injected into the constructor**
  - `ViewContainerRef`—allows access to the DOM around the element with the structural directive applied to it
  - `TemplateRef<Object>`—wrapper for the `<template>`

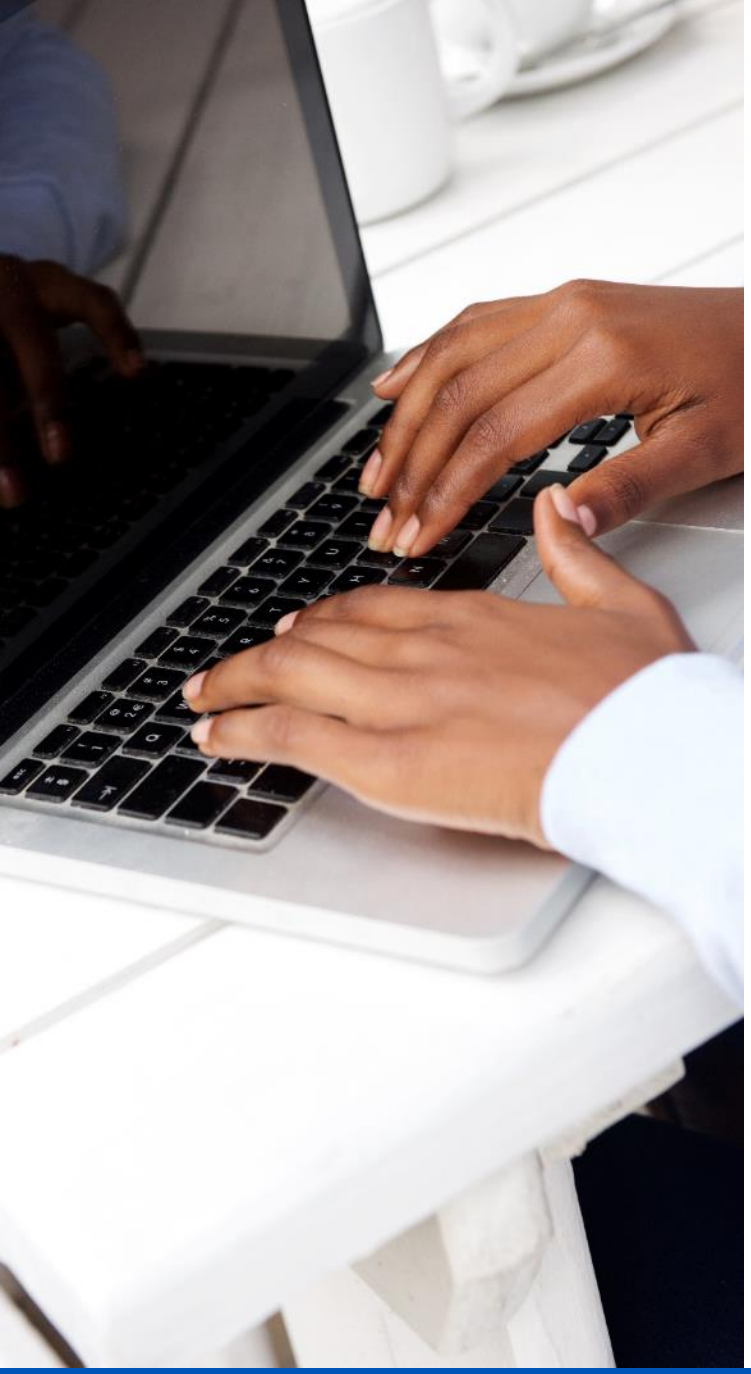
- **If the condition is true**
  - Render the template with:

```
this._viewContainer
            .createEmbeddedView(this._templateRef);
```

- **Else**
  - Replace the `<template>` with `<script></script>`

```
this._viewContainer.clear()
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 9.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 9.1: Creating an Attribute Directive

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Attribute Directives
- Structural Directives

Animation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Adding Animation

- **Animation can improve the user experience**
  - Smooth transitions
  - Gaining the user's attention
- **Animation also helps sell your product!**
- **Angular supports animation built on top of the Web Animation API**
  - Not supported by all browsers
  - Polyfill available: `https://github.com/web-animations/web-animations-js`
- **Animations are defined by states, transitions, and triggers**
  - State: the styling when in that state
  - Transition: defines visual effects when changing from one state to another
  - Triggers: define how an animation relates to a template

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Defining Animations

- **Animations are defined as a property of `@Component`**
  - `animations : [trigger, trigger]`
- **Triggers have a name**
  - One or more states
  - One or more transitions

```
animations: [
  trigger('myanimate', [
    state('on', style({
      opacity : 0
    })),
    state('off',   style({
      opacity : 1
    })),
    transition('on => off', animate('1000ms ease-in')),
    transition('off => on', animate('1000ms ease-out'))
])
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Adding Animation to a Template

- **Animation triggers are added to template elements**
  - As attributes: `@trigger-name`

```
<div [@myanimate]="animate"></div>
```

Binds animation trigger '`myanimate`' to the Component property `animate`

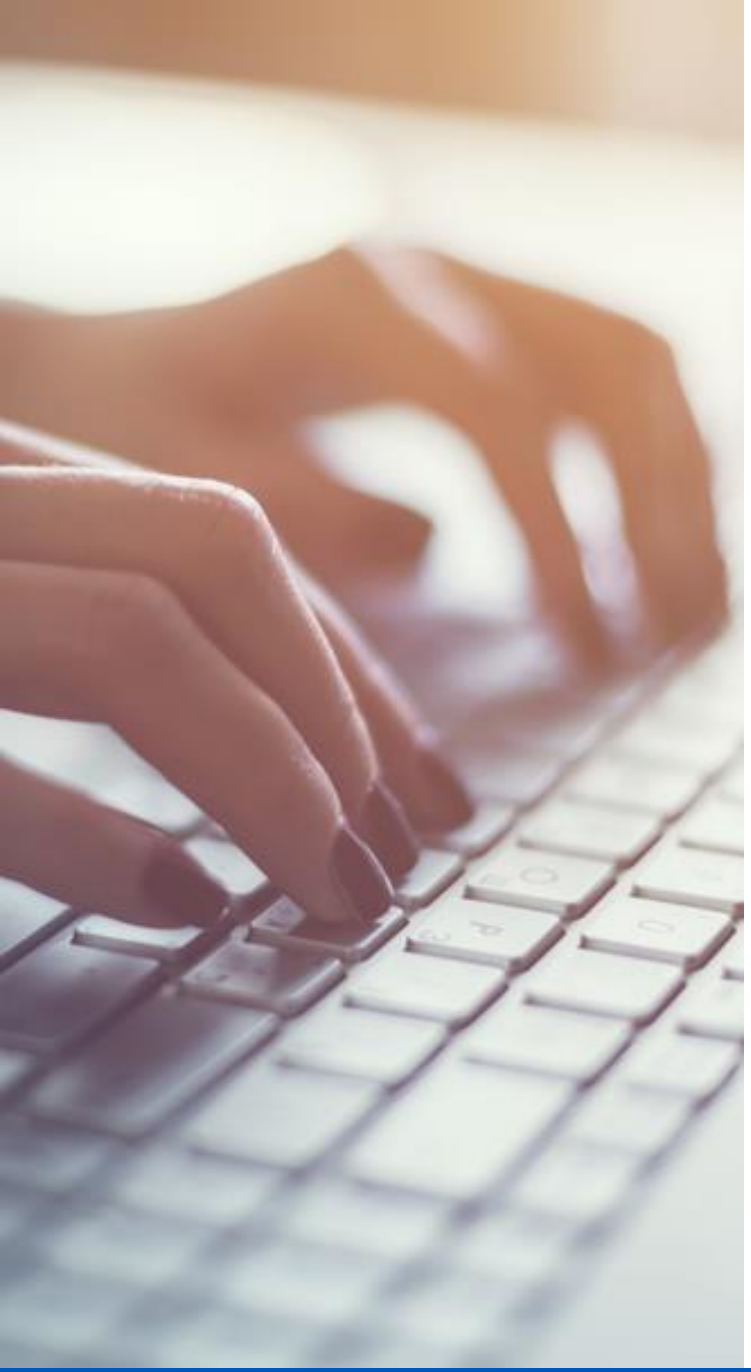
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Enabling Animation

- **To enable animation, your module must include the `BrowserAnimationsModule`**

```
import { BrowserAnimationsModule }
       from '@angular/platform-browser/animations';

@NgModule({
       …
   imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    BrowserAnimationsModule
  ],
       …
})
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



9.1: Animation in Action

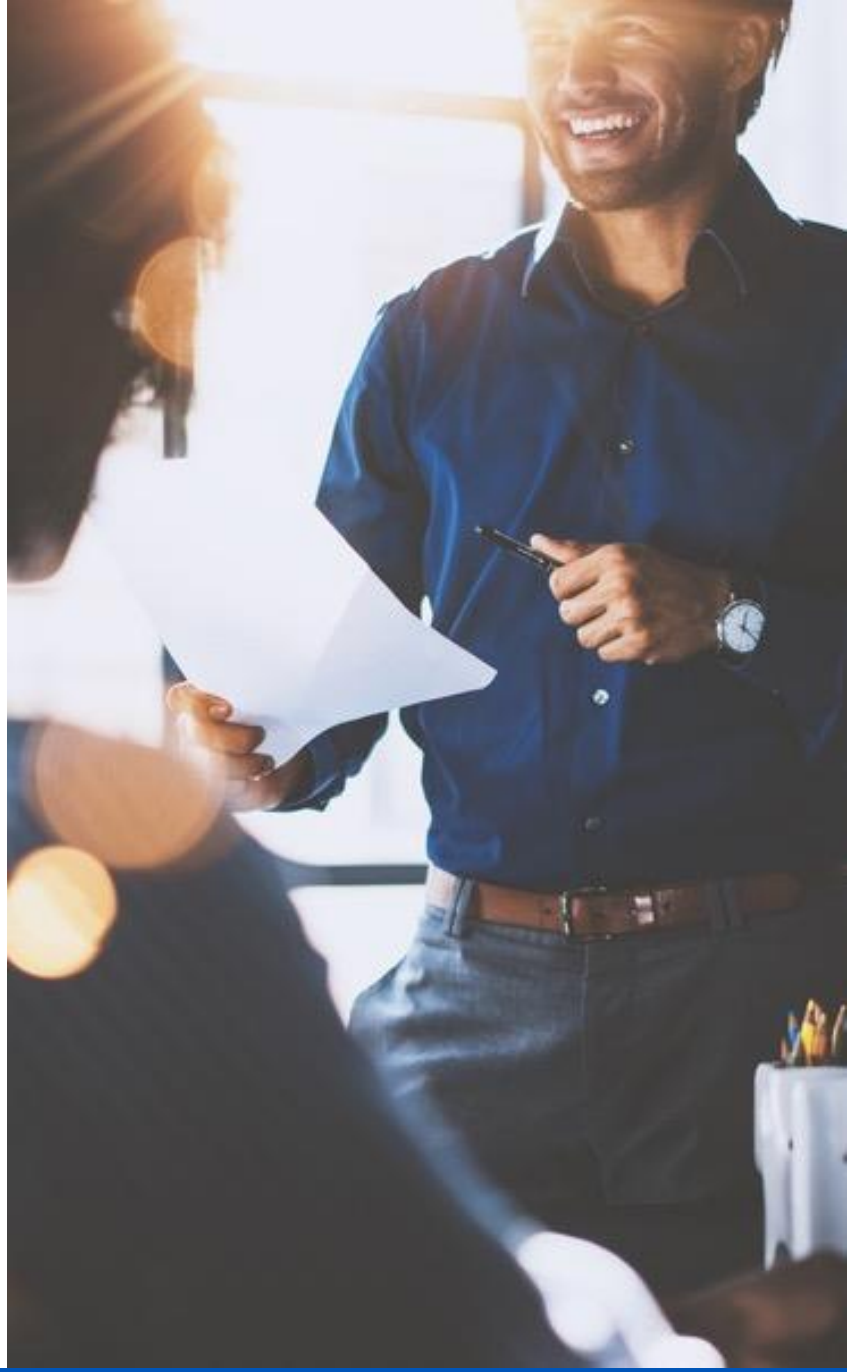
Do Now

- In your Exercise Manual, please refer to Do Now 9.1: Animation in Action

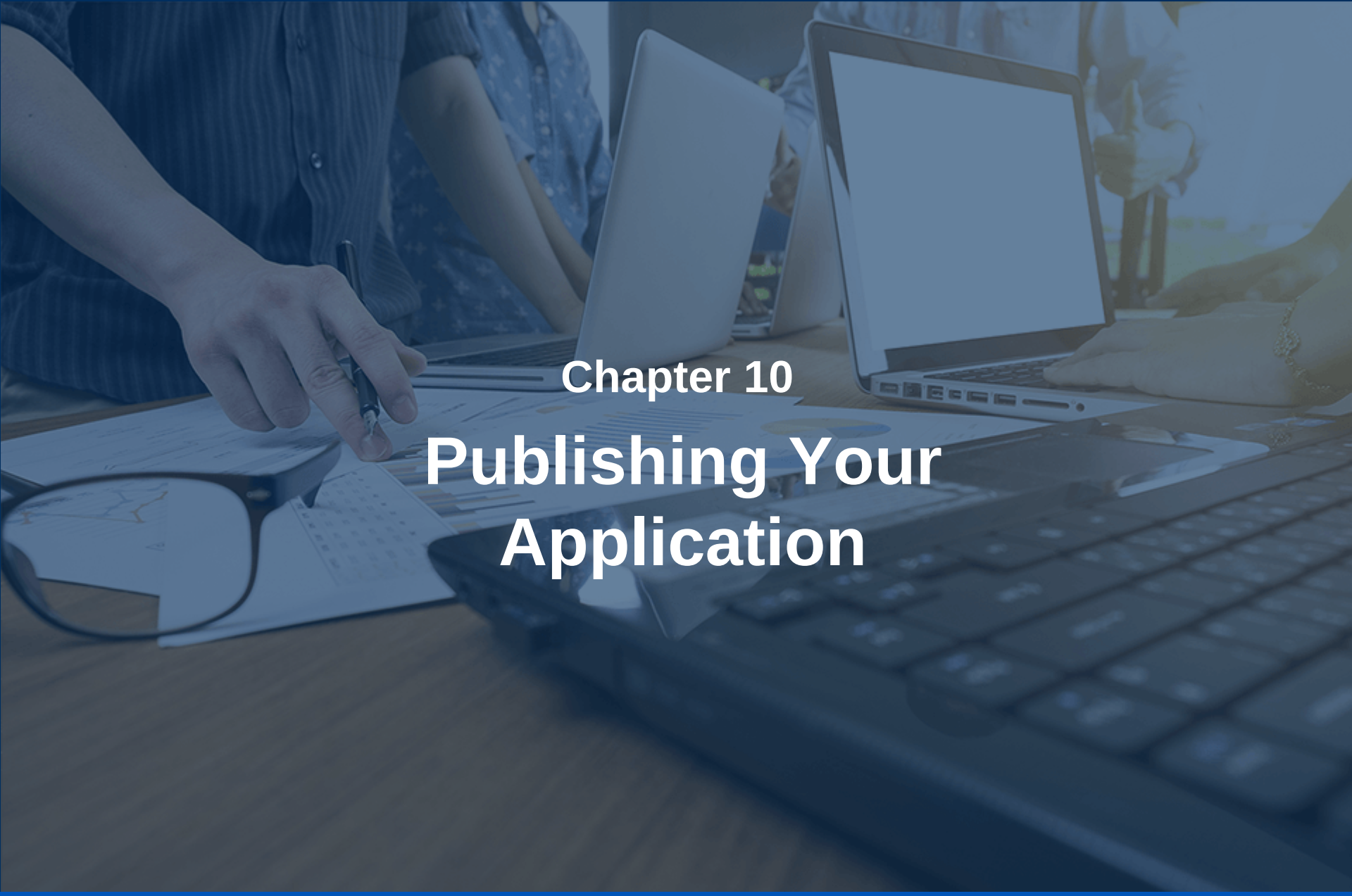
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Differentiate directive types
- Build attribute directives
- Investigate structural directives
- Implement an animation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



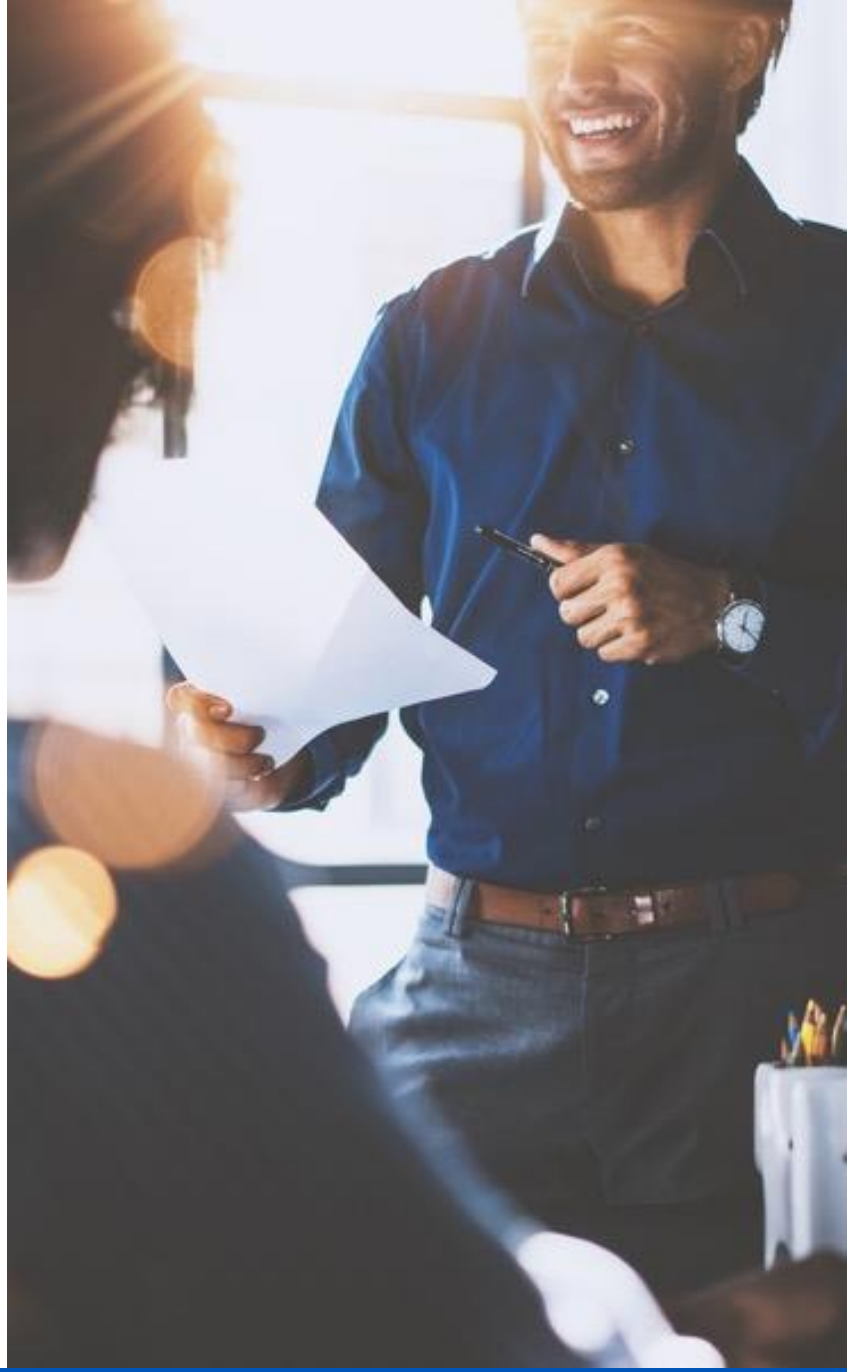
Chapter 10

Publishing Your Application

LEARNING TREE INTERNATIONAL

Objectives

- Prepare your application for publication
- Publish your application on a web server

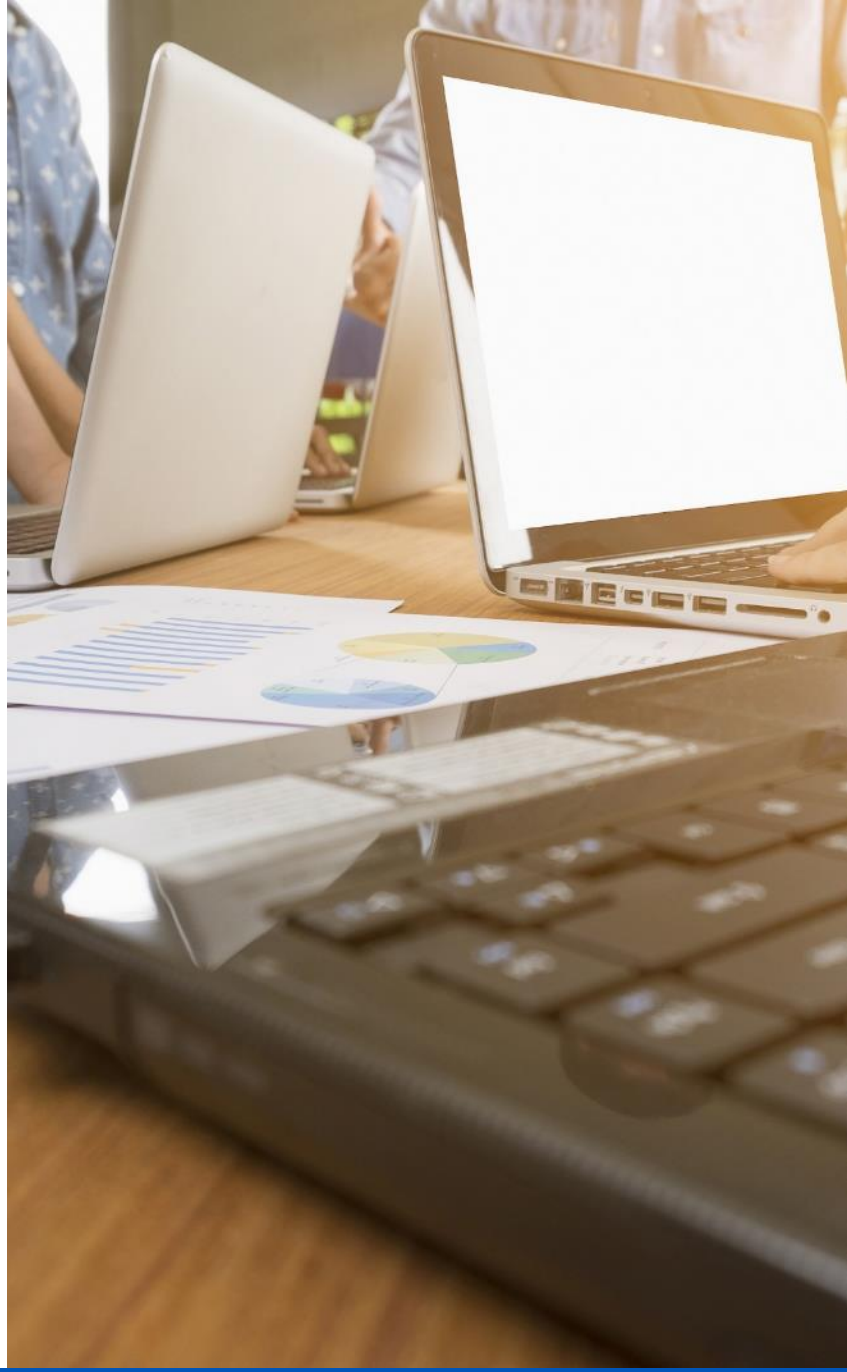


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Preparing for Publication

- ► **Publishing to a server**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Moving to Production

- **We have been working with our application on a development server**
  - Using `ng serve`
- **To publish we need to**
  - Run on a production server
    - IIS, Apache, Nginx, Lighttpd, etc.
  - Compress and bundle Javascript
  - Compress and bundle other resources

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Compressing and Bundling Javascript

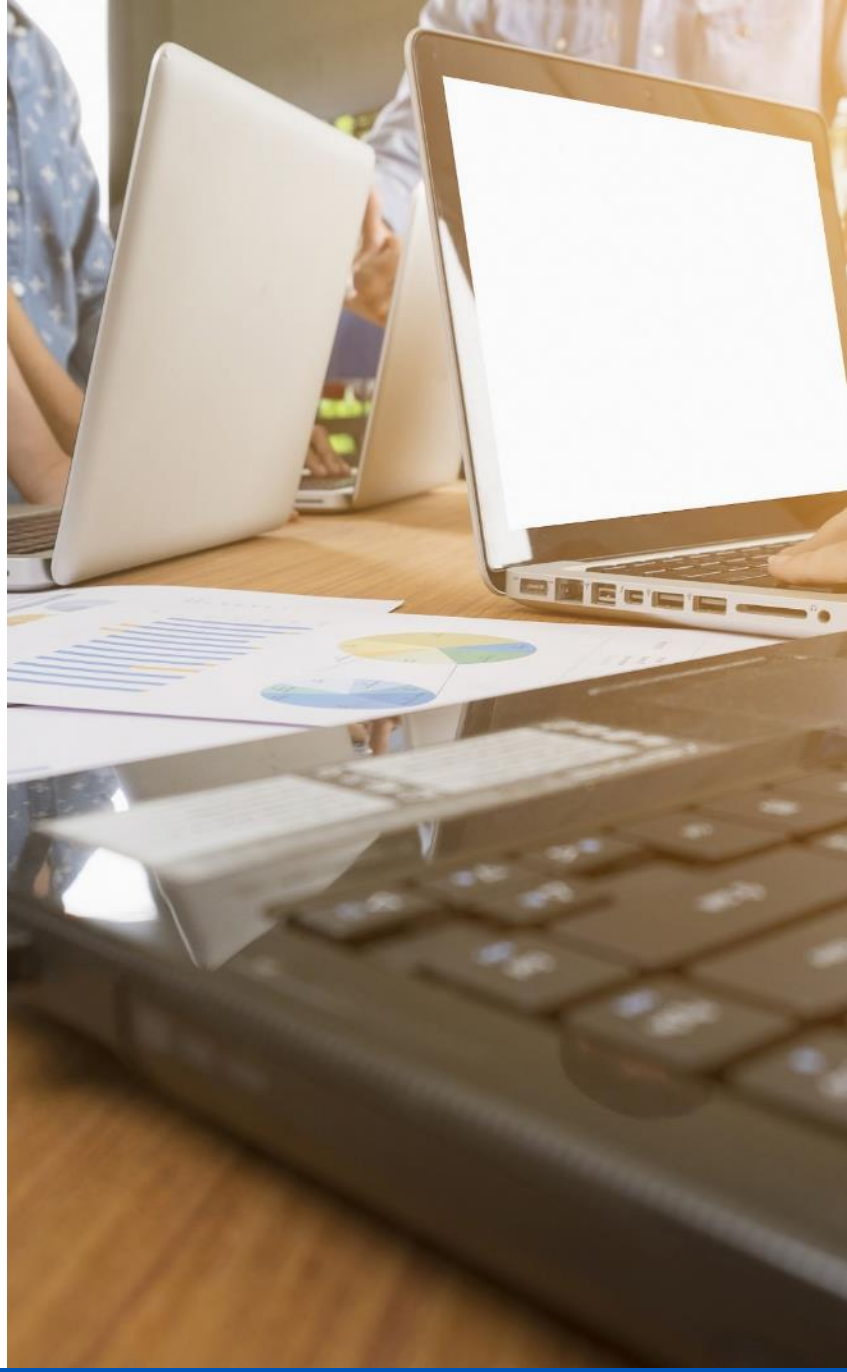
- **An Angular application references a very large number of source files**
  - Referencing each one via a `<script>` tag would be slow and inefficient
- **In the development environment, Webpack bundles the application into a small number of files**
  - To aid debugging, these files are not compressed
- **In production, the bundle files should be as small as possible**
- **Achieved by two processes**
  - Tree-shaking—removal of unused code
  - Uglification—compresses Javascript with a number of techniques including
    - Removal of whitespace
    - Shortening of symbols

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- ► **Preparing for Publication**

Publishing to a server



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Ahead-of-Time Compilation

- **The code we write for Angular components is not actually executable!**
  - It's a declaration of a component's appearance and behaviour
    - Not a set of steps to execute
- **Angular compiles component code to executable code in the form of component factories**
- **Compilation may be performed**
- **Just-in-Time (JIT)**
  - The browser runs the compiler before components are rendered
- **Ahead-of-Time (AOT)**
  - Components are compiled as part of the build process
- **The default behavior is JIT**
- **AOT may be enabled by adding the `--aot` flag to `ng serve` or `ng build`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Should You Use AOT?

- **If AOT is enabled**
  - Applications load faster
  - The downloaded bundles are smaller
    - No JIT compiler
  - Template errors are detected during build
  - Security is enhanced
    - Original component source not sent to the browser
- **But…**
  - There are restrictions on the content of the `@Component` metadata
    - Only a subset of JavaScript is supported
  - For details and work-arounds see: https://angular.io/guide/aot-compiler

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Browser Support—JavaScript

- **Angular code uses the latest JavaScript specifications and extensions**
  - May not be available on all browsers
- ***Polyfills* provide missing support**
  - Defined by contents of `src/polyfills.ts`
- **Code is automatically generated for both ES5 and ES2015 by the build process**
  - The `<script>` tags inserted into the HTML use `nomodule` / `type="module"`
    - To load the correct script dependent on browser support

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Browser Support—CSS

- **Some browsers require vendor specific prefixes for newer CSS features**
  - Example for the transition property:

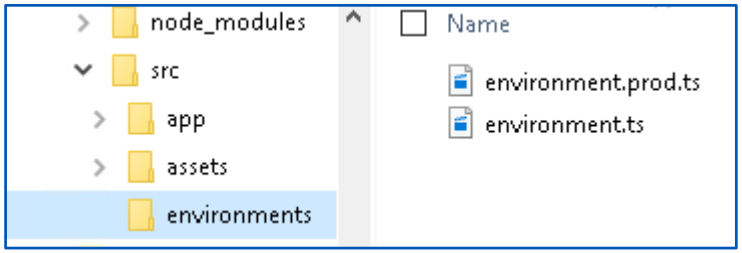
```
-webkit-transition: all 4s ease;
-moz-transition: all 4s ease;
-ms-transition: all 4s ease;
-o-transition: all 4s ease;
transition: all 4s ease;
```

- **Angular applications should just specify the CSS3 standard property**
- **The build process will add vendor prefixes as required**
- **Uses a library called browserlist to define supported browsers**
  - Configured by a file called `browserlist` in the project root directory

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Production vs. Development Builds

- **The environments folder contains two files**
  - `environment.ts`
  - `environment.prod.ts`
- **May be used to define values that change between builds versions**




```
export const environment = {
  production: false,
  apiUrl: 'localhost:1234/flights'
};
```

environment.ts

```
export const environment = {
  production: true.
  apiUrl: 'prodhost.com:1234/flights'
};
```

environment.prod.ts

```
import { environment } from '../environments/environment;

export class FlightsService
  private serviceUrl = environment.apiUrl;
};
```

flights.service.ts

Value from `environment.prod.ts` in production build and from `environment.ts` in development

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

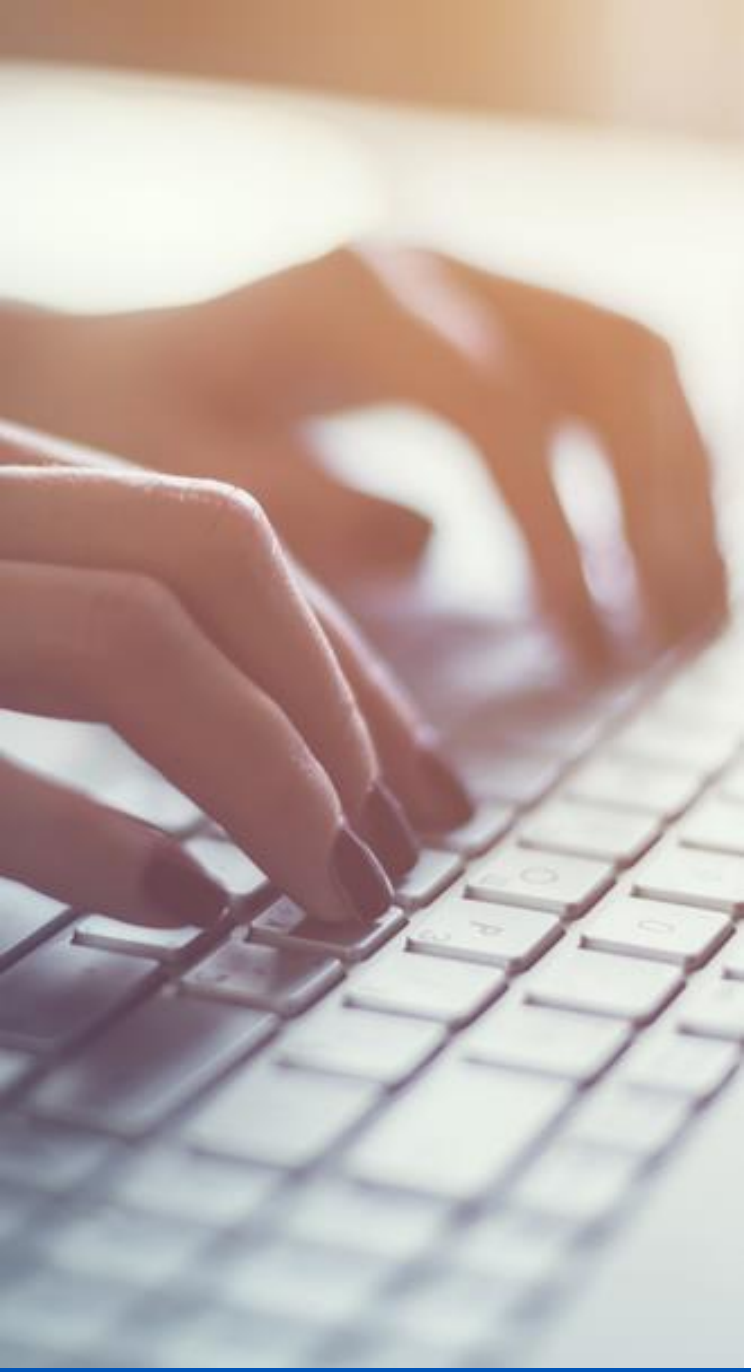
Publishing With Angular CLI

- **Angular CLI can perform the build compression and bundling tasks for us**

```
ng build --prod --aot
```

- **There are many options to `ng build`**
  - View with `ng build –help`
  - `--prod` switch to the production version of `environment.ts`
    - Also suppresses the generation of .map files (used for debugging TypeScript)
  - `--aot` enable ahead-of-time compilation
- **Creates a set of files in the `dist` directory of our project**
- **These files can be copied directly to a production web server**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



10.1: Publishing the `FlySharp` Application

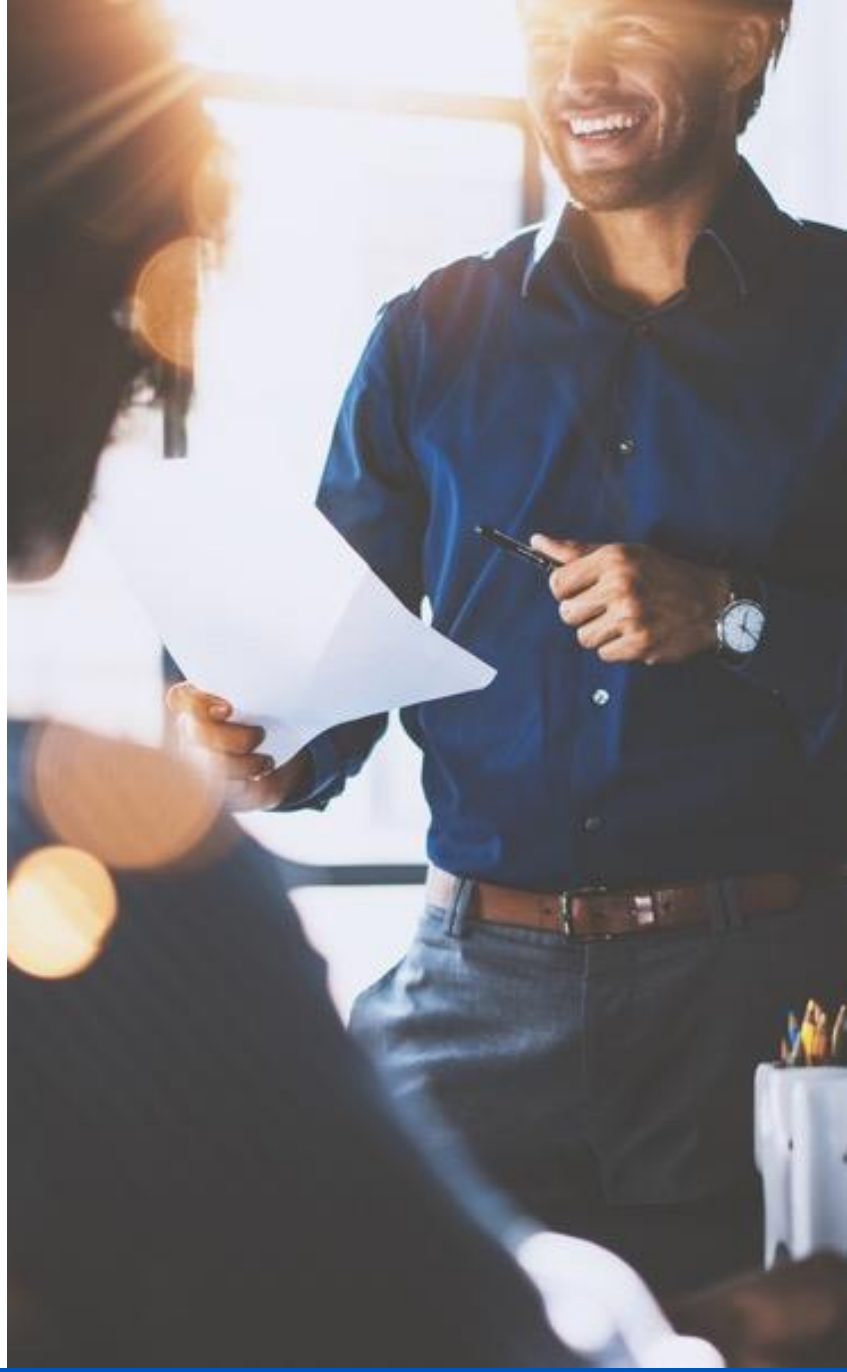
Do Now

- In your Exercise Manual, please refer to Do Now 10.1: Publishing the `FlySharp` Application

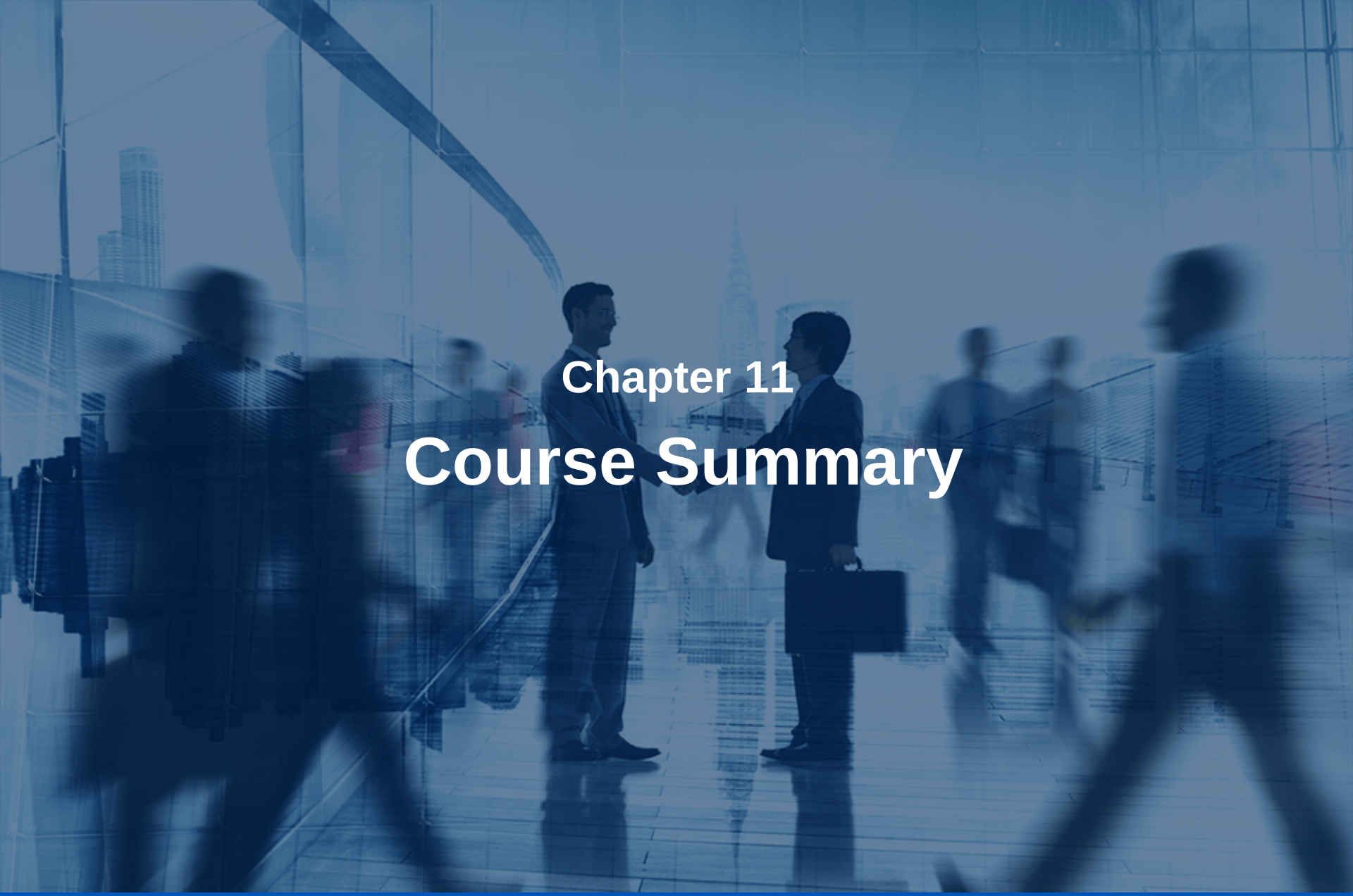
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Prepare your application for publication
- Publish your application on a web server



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Chapter 11

Course Summary

LEARNING TREE INTERNATIONAL

Course Objectives

- Create well-structured Angular applications
- Develop Components using TypeScript, Templates, and Decorators
- Modularize applications with Feature Modules and the Component Router
- Add interactivity with event handlers and bindings
- Capture and validate input with template-driven and reactive forms
- Consume REST services using Observables

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

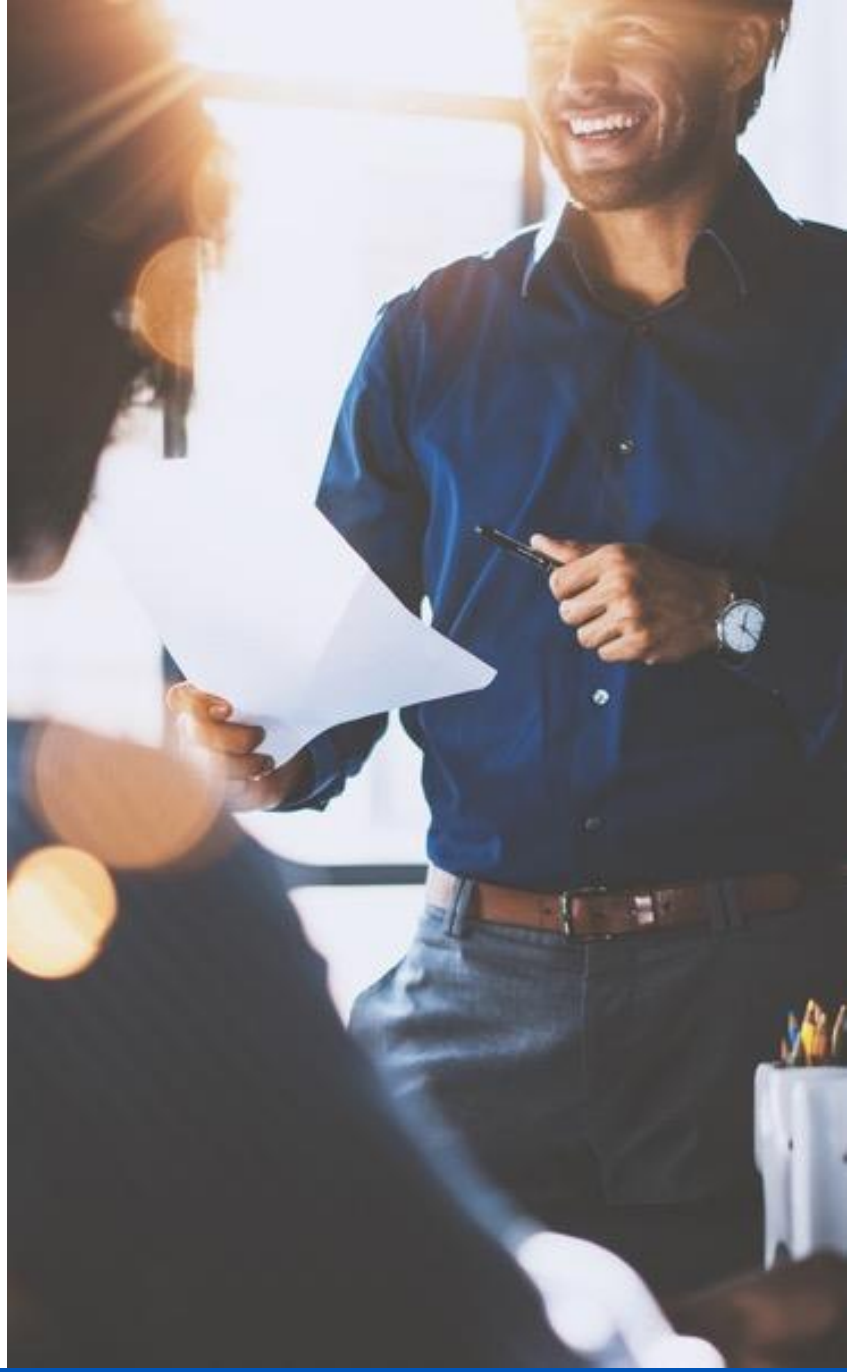
Appendix A

Migrating From Angular JS 1

LEARNING TREE INTERNATIONAL

Objectives

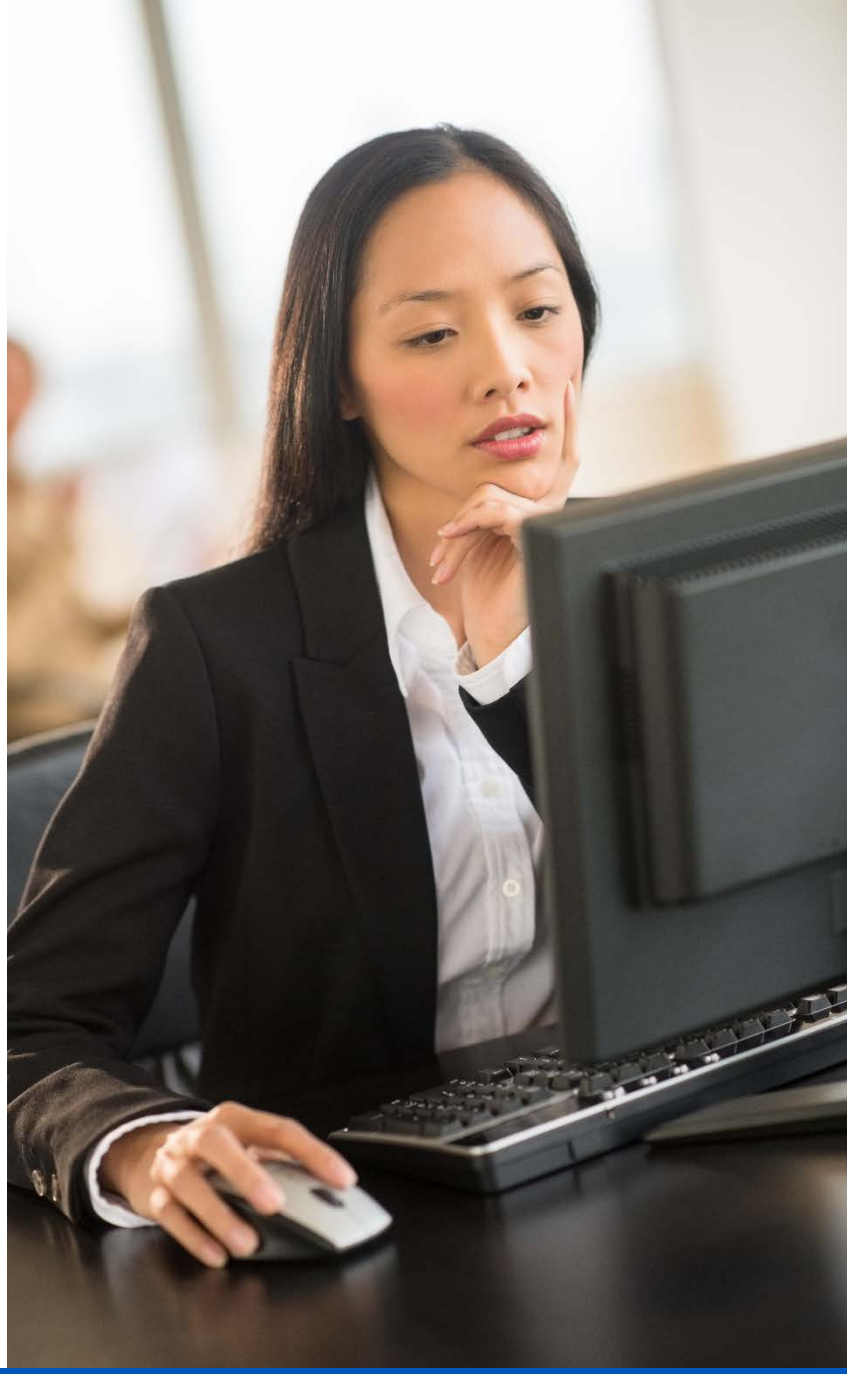
- Combine Angular JS and Angular in a single application
- Incorporate Angular Components in an Angular JS application



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Same But Different!

- **Angular is significantly different from Angular JS**
- **There is no backward compatibility…**
- ***But***
  - The `UpgradeAdapter` allows Angular and Angular JS applications to coexist
- **Angular Components may then be used in an Angular JS application**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Best Approach

- **The best, and sometimes easiest, approach to migration**
  - Take each Angular JS component/directive one at a time
    - Create a new Angular component
    - Copy the Angular JS template into the new component
    - Copy the Angular JS logic into the new component
  - Glue all the components together into a new application
- **Sometimes that is not possible**
  - Use the upgrade adapter

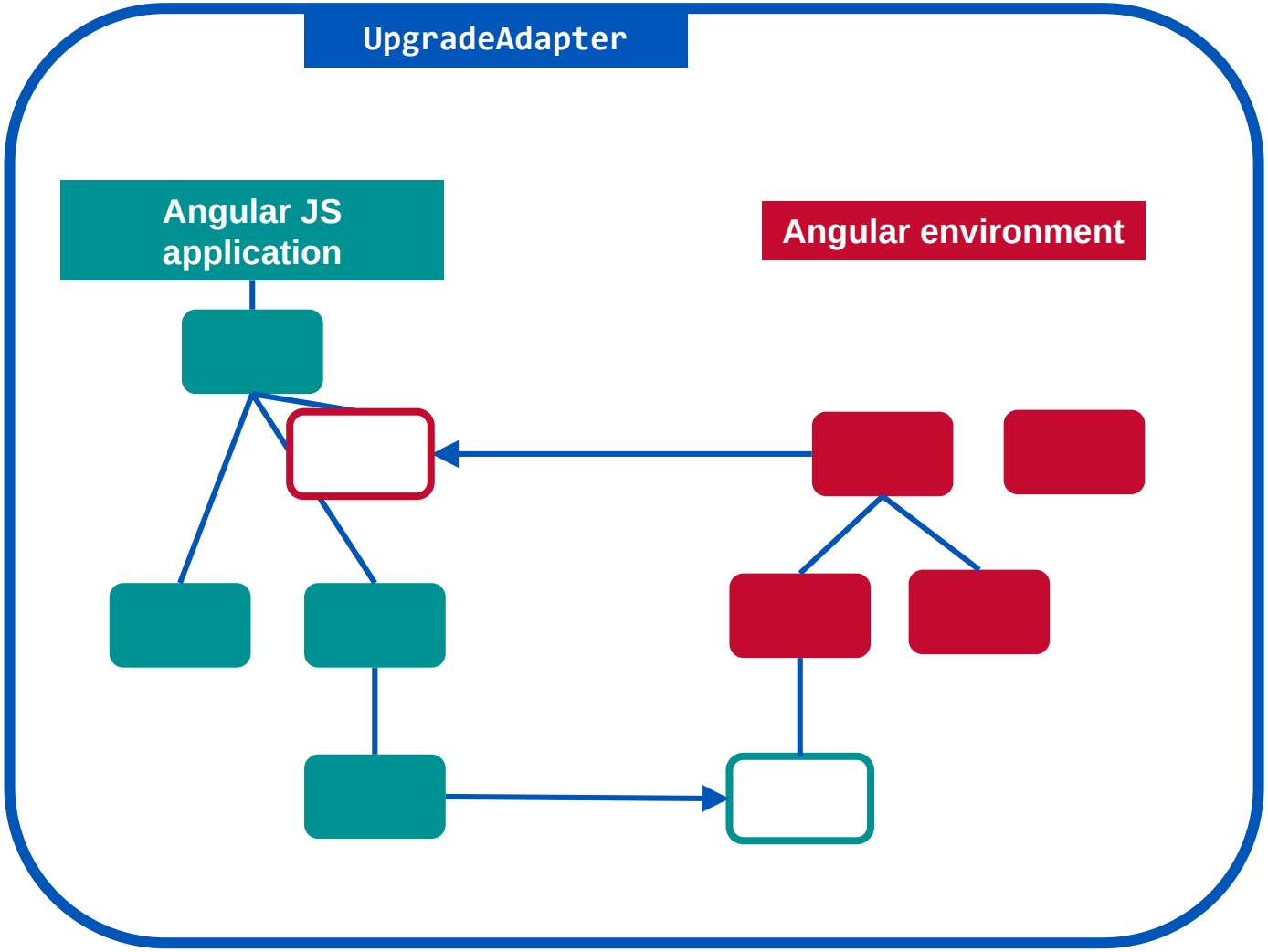
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The `UpgradeAdapter`

- **Runs both the Angular JS and Angular environments**
- **Bootstraps an Angular JS application**
  - In which Angular Components are available
- **Intended to allow gradual migration to Angular**
  - New features added as Angular Components and services
- **May be used as a development environment to safely migrate to A2**
- **Key concepts**
  - Application is booted as an A1 application
    - By the `UpgradeAdapter`
  - A2 Components can be downgraded to be used in A1
  - A1 Components can be upgraded and used by Angular

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The UpgradeAdapter



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

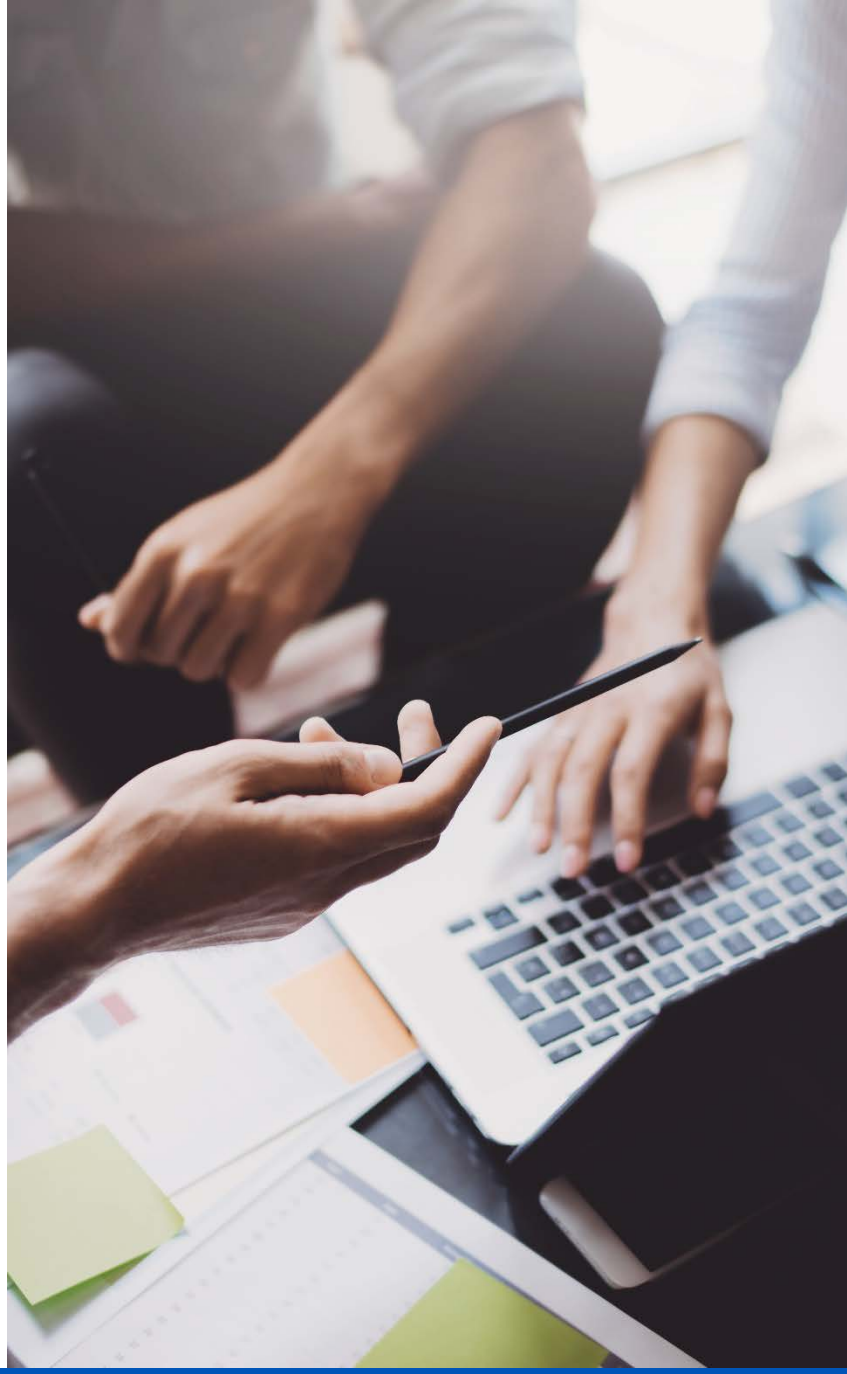
Crossing Boundaries

- **The `UpgradeAdapter` creates a hybrid application**
  - Angular JS 1 processes the root template
  - Angular Components are available via the adapter
- **The boundary between the two frameworks may be crossed**
  - By using a Component from the other framework
    - Angular 1 template uses an Angular Component
    - Angular template uses an Angular 1 Component
- **The adapter manages the injection tree to allow interoperation**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Before you Begin

- **Before using the upgrade adapter, you should**
  - Convert your Angular JS application to TypeScript
  - Convert your Angular JS application to 1.5+
  - Convert your directives to components
- **Not trivial**
  - See Angular JS documentation for converting to components



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Getting Started With the `UpgradeAdapter`

- **To use Angular JS and Angular together**
  - Must boot an Angular JS application with the `UpgradeAdapter`
    - As part of the Angular application
- **Once the two applications are working together**
  - Can run Angular components in the Angular JS application
  - Gradually migrate Angular JS components to Angular
- **Full instructions:**
  - https://angular.io/docs/ts/latest/guide/upgrade.html
  - Has some errors/omissions
- **Alternative: Use our starter app**
- **Clone** `https://mikejrway@bitbucket.org/mikejrway/course2324-js-angular.git`
  - Follow the instructions in the `README.md` file

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Bootstrapping

- **Angular JS template typically starts the application with `ng-app`**
  - The value of the attribute being the initial directive name

```
<body ng-app="phonecatApp" ng-strict-di>
```

- **Hybrid application must be bootstrapped from code**
  - `bootstrap` method of `UpgradeAdapter`

main.ts

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .then(platformRef => {
        const upgrade =
                platformRef.injector.get(UpgradeModule) as
                                                UpgradeModule;
        upgrade.bootstrap(document.documentElement,
                                                ['phonecatApp']);
});
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Bootstrapping—The Angular Module

- **The Angular Module must provide `ngDoBoostrap()`**
  - Can be an empty method

```
@NgModule({
  declarations: [
    PhoneListComponent,
  ],
  imports: [
    BrowserModule, UpgradeModule, HttpModule
  ],
  providers: [
    Phone,
  ],

  entryComponents: [
    PhoneListComponent,
  ]
})
export class AppModule {
  ngDoBootstrap(){}
}
```

Instead of Bootstrap—tells Angular the start of the component hierarchy(ies)

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Making an Angular Service Available to Angular JS

- **Use `downgradeInjectable` to make a service available to Angular JS**

```
import { Injectable } from '@angular/core';
declare var angular: angular.IAngularStatic;
import { downgradeInjectable } from '@angular/upgrade/static';


@Injectable()
export class Phone {
  constructor(private http: Http) { }
  query(): Observable<PhoneData[]> {
    return …
  }
}
angular.module('core.phone')
  .factory('phone', downgradeInjectable(Phone));
```

- **Can now use the service with Angular JS in the usual way**
  - `static $inject = ['phone'];`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Downgrading Angular Components

- **To use an Angular Component from Angular 1**
  - Downgrade it with the `UpgradeAdapter`!

main.ts

```
angular.module('phoneList')
 .directive(
   'phoneList',
   downgradeComponent({component: PhoneListComponent}) as
       angular.IDirectiveFactory
);
```

- **The downgraded Component follows the normal Angular JS 1 rules**
  - The directive name is converted to an element name using Kebab case

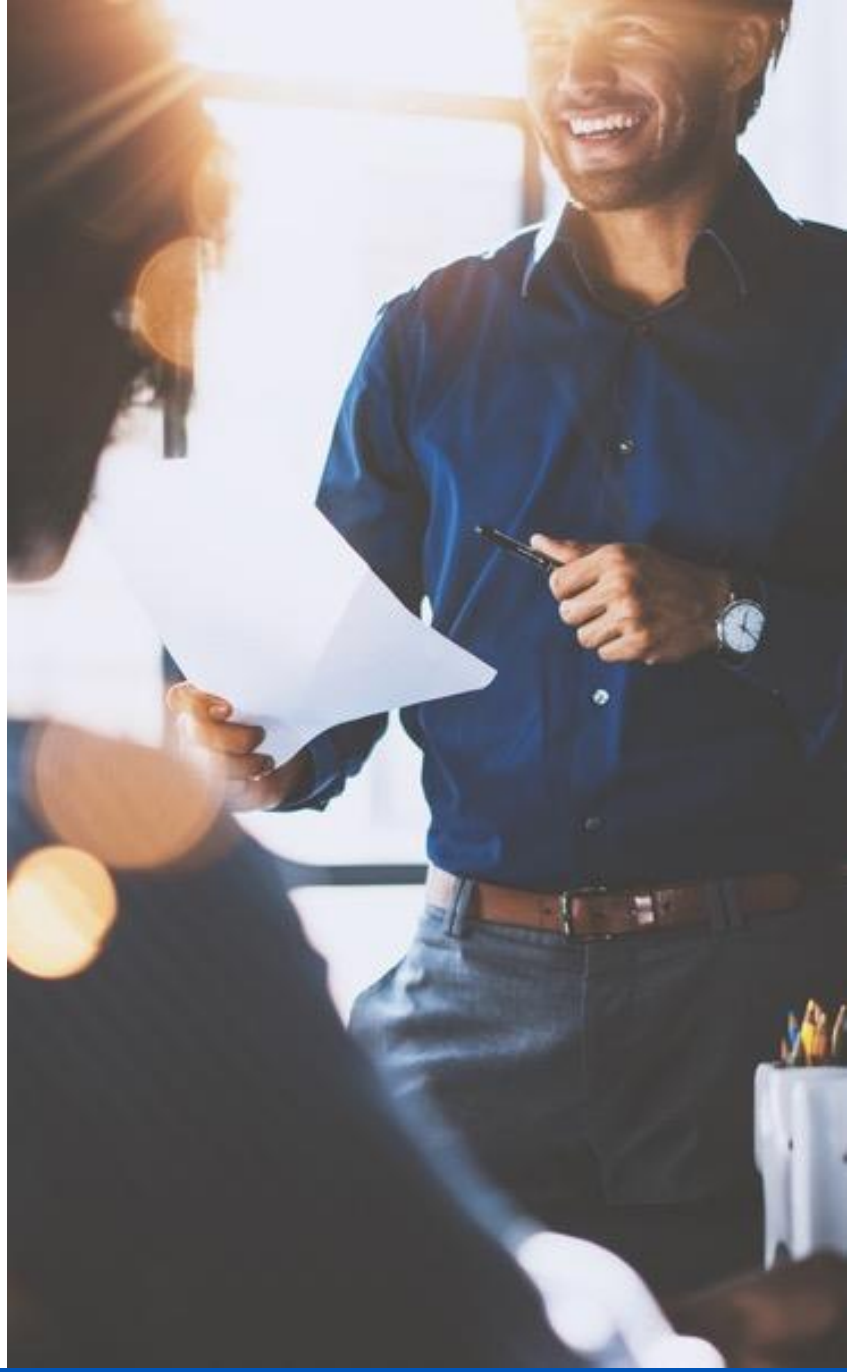
container.ts

```
<phone-list></phone-list>
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Combine Angular JS and Angular in a single application
- Incorporate Angular Components in an Angular JS application



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.