

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Introduction

Course Description

How is this course valuable to a Full Stack Engineer (FSE)?

- Today we can be “continuously connected” by using a variety of devices that go well beyond the traditional web browser running on a desktop computer
- A dynamic web application is one which is connected, interactive, and responsive
 - It is connected via the Internet to other people or other systems
 - They make effective interaction possible by utilizing audible, visual, and physical modes, which facilitates your ability to communicate
 - A short, quick response time helps produce a pleasant, useful, and useable application
- If you can think of an idea, there is probably “an app for that;” if not, then perhaps you should create one
- This course will show you how to take advantage of the current technologies to develop dynamic web applications
- Coming up with the next cool idea for an app is left to you!

Course Outline

- Chapter 1 Introduction to Visual Studio Code
- Chapter 2 Standardizing Presentation with HTML and CSS
- Chapter 3 Advanced HTML and CSS
- Chapter 4 Client-Side JavaScript Programming
- Chapter 5 Working with jQuery
- Chapter 6 Introduction to Angular
- Chapter 7 Angular Components
- Chapter 8 Angular Modules and Binding
- Chapter 9 Angular Services

Course Outline (continued)

Chapter 10 Angular End-to-End (E2E) Testing Applications

Chapter 11 Building an Application

Chapter 12 Pipes

Chapter 13 Angular Routing

Chapter 14 Angular Forms

Chapter 15 Angular Deployment

Appendix A Angular Directives

Appendix B Observables

Course Objectives

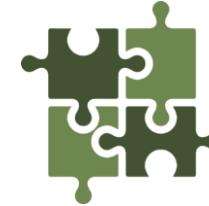
In this course, we will:

- Learn to develop responsive web applications
- Understand what makes a web application responsive and interactive
- Build interactive websites with HTML
- Employ behavior in web pages using JavaScript and AJAX
- Standardize presentation using Cascading Style Sheets (CSS)
- Leverage the capabilities of HTML5 and CSS3
- Understand the need for JavaScript frameworks such as jQuery, node.js, and Angular
- Perform End-to-End (E2E) testing with Jasmine and Protractor

Key Deliverables



Course Notes



Project Work



There is a Skills Assessment for this course

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 1: Introduction to Visual Studio Code

Chapter Overview

In this chapter, we will explore:

- Visual Studio Code

- Explorer
- Extensions
- Work area
- Launching web page from Visual Studio Code

Chapter Concepts

Visual Studio Code

Chapter Summary

Visual Studio Code

- While it is possible to create web pages using any old text editor, it is much better to use one that makes your job easier
- One of the more popular choices is Visual Studio Code (VSC)
- VSC provides many features
 - Lightweight
 - Powerful editor
 - Supports JavaScript, TypeScript, with extensions for many more languages
 - Intellisense
 - Others that we will discuss later

Online Support for Visual Studio Code

- There is a large amount of information online that supports working with VSC
- Getting started introductory videos:
 - <https://code.visualstudio.com/docs/getstarted/introvideos>
- Tips and Tricks:
 - <https://code.visualstudio.com/docs/getstarted/tips-and-tricks>



HANDS-ON
EXERCISE

40 min

Exercise 1.1: Introduction to Visual Studio Code

- Your instructor will demonstrate some of the features of Visual Studio Code
- Please read the instructions for this exercise in your Exercise Manual
- Follow along with your instructor

Chapter Concepts

Visual Studio Code

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Visual Studio Code

- Explorer
- Extensions
- Work area
- Launching web page from Visual Studio Code



Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 2: Standardizing Presentation with HTML and CSS

Chapter Overview

In this chapter, we will explore:

- Standardizing presentation with HTML and CSS
 - HTML essentials
 - Styling with CSS
 - Creating CSS in external files

Chapter Concepts

Introduction to HTML

Styling with CSS

Chapter Summary

Recap: A Basic HTML Document

- A basic HTML document has a document type declaration and an `html` element that contains one `head` and one `body` element

```
<!doctype html>
<html>

<head>
  <title>A Basic Web Page</title>
</head>

<body>
  <h1>Basic HTML Elements</h1> <!-- comment won't show up in browser view -->
  <p>This is a paragraph containing <em>emphasized</em> text.</p>
</body>

</html>
```

Basic Syntax

■ These are the basic elements seen on the previous slide:

- `<!doctype html>`: specifies HTML5 and higher (more on this later)
- `<html>...</html>`: the root element of the document
- `<head>...</head>`: contains information about the document, such as title
- `<title>...</title>`: title of document, shown as window title in web browser
- `<body>...</body>`: contains all the (visible) content of the web page
- `<h1>...</h1>`: is a level 1 heading (important)
 - Also `<h2> ... <h6>`
- `<p>...</p>`: is a paragraph
- `...`: emphasis

■ Most elements are indicated by an opening and closing tag

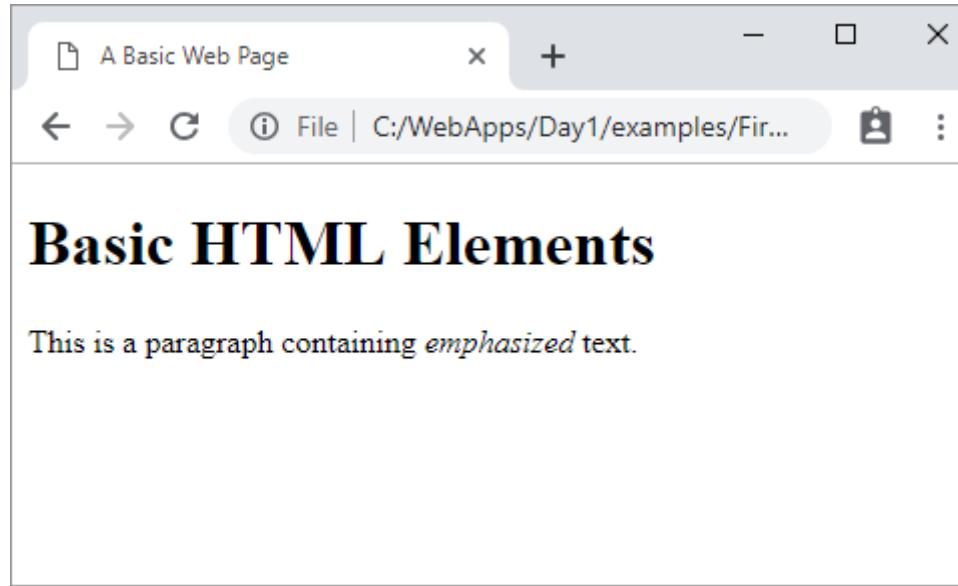
- E.g., opening tag `<head>`, closing tag `</head>`

■ HTML comments are enclosed in `<!-- ... -->`

- May not be nested

Appearance

- The browser (technically known as the User Agent) gives each element a default appearance
 - Later we will see how to customize this



DOs and DON'Ts

```
<html>
  <head>
    Use only for styles, loading JS code, title , meta elements
    Never put page related HTML elements in here!
  </head>

  <body>
    ALL HTML related tags belong in here including semantic tags!
    Some JS code is OK, some JS code can be loaded here as well
    Don't put style tags in body
  </body>
</html>
```

There is no neck!
No code or tags EVER go between </head> and <body>!

- Browsers forgive a lot, but bad or faulty coding styles can lead to unpredictable behavior!

When Is a Browser Not a Browser?

- Most web content is consumed by browsers where a human reads the content
- Not all browsers are created equal, especially mobile browsers
 - Some provide an “accelerated” mode where pages are pre-processed on a server
 - Reduces bandwidth usage and optimizes pages for smaller screen size
- Search companies scan web pages using programs called web crawlers
- Many people, especially those with sight impairment, use screen readers
 - A program that interprets the page and reads it out
- Personal assistants (e.g., Siri, Alexa, Google Assistant) are very popular
 - When you ask a question, the assistant often scans web pages for the answer
- All these features rely on being able to understand the structure of the web page
 - This course uses the term “screen reader” to indicate relevant features

Images

- Images cannot be embedded in or attached to the HTML
 - HTML is a purely text format
 - Unicode text, to account for international languages, but just text
- Images are “sourced” from another document using a URL
 - Either a path relative to the current page or an absolute path
 - Browser downloads and places images in the document

Attribute of the `img` tag

Tag does not require closing, or
may be marked as self-closing (`/>`)

```


```

- The `src` attribute is mandatory, all other attributes are optional
 - The `alt` information is displayed if the end-user has disabled images
 - And by screen readers

Image Formats

- It is recommended that you use one of four image formats:
 - **JPEG** (jpg) for photographs and smoothly varying images
 - Lossy compression based on how the eye sees the world
 - Good for smooth variations and complex scenes where edge detail is not important
 - **PNG** or **SVG** for images that contain mostly lines or text (e.g., diagrams, charts)
 - PNG is a “bitmap” format that supports lossless compression and alpha transparency
 - SVG is a “vector” format that can be scaled perfectly to any size
 - **GIF** when simple animation is required
 - Supports a limited set of colors and limited transparency
 - Has support for animation
- If an image format supports interlacing, use that
 - Users initially see a low-resolution image, while the rest of the image loads
- Reduce the image size to the desired width and height
 - Downsampling an image to a smaller size will help pages load faster

Scalable Vector Graphics

- XML-based vector format for 2D images
- Defines the image as a series of vector primitives:
 - E.g., circle, line, text
 - Since the primitives are vectors, they can be scaled to any size
 - Usually much smaller than a “bitmap” format for appropriate content
- Requires special software to create manually
 - Programmatically simple to create, good for graphs
- Contents of an SVG image become part of the page that displays them
 - Can be styled (e.g., colored) in the same way as any other HTML element
 - Can interact programmatically, e.g., react as the mouse moves



10 min

Try It Now: Review Image Formats

- In the folder Ch02\ImageDemo, open image-demo.html
 - Launch it in the default browser
 - When you move the mouse pointer over any of the images, they will all zoom 8x
- What can you see? Perhaps something like this:

	Graph	Photo
JPEG	Some blurring around text is clear when magnified, leads to softness at normal size.	Photo has some edge artefacts, not noticeable at normal size. Blocks of color are smooth.
GIF	Some blurring of text, but better than JPEG.	Worse edge artefacts. Speckling in blocks of color that is visible even at normal size.
PNG	Similar textual quality to GIF. Similar file size.	Almost as good as JPEG, but much large file size.
SVG	Pin sharp at all magnifications. Much smaller file size.	Cannot handle photos at all.

Links

- The “hyper” in HTML refers to the capability to click links
 - A link takes the user to a different web resource
 - A link is “anchored” (<a>) by some part of an HTML page, often text or an image
 - Specify the URL of the destination in the href attribute of the <a> tag
 - Could be a relative path, as here, or an absolute path

<p>

Here is an example of a JPG image, which is a particularly good format for photographs.

```
<a href="links.html">  
    </a>
```

This photo is also a link, if you click on it, you will be taken to the page about links.

</p>

Link to Sections of a Document

- Can link to a particular section of a document
 - Give the section to be linked an id
 - Any element can have an id and it must be unique within that document

```
<h2 id="pre">Pre-formatted Text</h2>
```

- Link to this from the same document as:

```
<p>
  You can jump ahead to <a href="#br">Line Breaks</a>, <a href="#pre">Pre-formatted Text</a>,
  <a href="#hr">Horizontal Rule</a> or <a href="#nbsp">Non-breaking space</a>
</p>
```

- Link to a section from a different document by adding section ID to path
 - Can add section ID to both absolute and relative paths

```
<a href="https://en.wikipedia.org/wiki/Chicken_Itza#Location">Where is it?</a>
```

Lists

- Can create ordered and unordered lists of items
 - Use `` for items and `` for ordered lists; `` is unordered

```
<h1>Lists</h1>
<p>
    This is an ordered list:
    <ol>
        <li>Reusable</li>
        <li>Maintainable</li>
        <li>Clear</li>
    </ol>
</p>
<p>
    This is an unordered list:
    <ul>
        <li>Unique</li>
        <li>Custom</li>
        <li>Complicated</li>
    </ul>
</p>
```

Lists

This is an ordered list:

1. Reusable
2. Maintainable
3. Clear

This is an unordered list:

- Unique
- Custom
- Complicated

By default, list items in an ordered list are given numbers, whereas items in an unordered list are given bullets

Line Breaks and Horizontal Lines

- Can force a line break by using

 - Use this when the line break is part of the content, as in poetry
 - Use <p> or <hr> to indicate a thematic break
- Can have the browser draw a horizontal line by using <hr>
 - This should not be used just for appearance, it should be thematic

```
<hr>
<h2>Line Breaks</h2>
<p>They could be used to lay out poetry:</p>
<h3>A Poem Never Says Anything</h3>
<p>A poem never says anything.<br>
It just opens a door, quietly.<br>
<br>
Sleepless and bent<br>
just like my aged father<br>
waiting for me in a lonely winter night.<br>
<br></p>
<p><em>Uttaran Chaudhuri</em></p>
<hr>
```

Preformatted Input

- If you have plain text that should not be wrapped, can use `<pre>`
 - Preserves line breaks and whitespace
 - Often used for code listings, but can also be used for poetry

```
<pre>
```

I will arise and go now, and go to Innisfree,
And a small cabin build there, of clay and wattles made;
Nine bean-rows will I have there, a hive for the honey-bee,
And live alone in the bee-loud glade.

And I shall have some peace there, for peace comes dropping slow,
Dropping from the veils of the morning to where the cricket sings;
There midnight's all a glimmer, and noon a purple glow,
And evening full of the linnet's wings.

I will arise and go now, for always night and day
I hear lake water lapping with low sounds by the shore;
While I stand on the roadway, or on the pavements grey,
I hear it in the deep heart's core.

```
</pre>
```

Non-Breaking Space

- Normally, the web browser wraps text by looking for spaces
 - If you have a space separating words that should not be broken, use

```
<p>Try re-sizing the browser and see whether the company name is ever split over two lines.</p>
```

```
<p>  
    Fidelity&nbsnbsp;Investments&nbsnbsp;Inc Fidelity&nbsnbsp;Investments&nbsnbsp;Inc  
    Fidelity&nbsnbsp;Investments&nbsnbsp;Inc Fidelity&nbsnbsp;Investments&nbsnbsp;Inc  
    Fidelity&nbsnbsp;Investments&nbsnbsp;Inc Fidelity&nbsnbsp;Investments&nbsnbsp;Inc  
    Fidelity&nbsnbsp;Investments&nbsnbsp;Inc Fidelity&nbsnbsp;Investments&nbsnbsp;Inc  
    Fidelity&nbsnbsp;Investments&nbsnbsp;Inc Fidelity&nbsnbsp;Investments&nbsnbsp;Inc  
    Fidelity&nbsnbsp;Investments&nbsnbsp;Inc Fidelity&nbsnbsp;Investments&nbsnbsp;Inc  
</p>
```

- This is an example of an “entity”
 - Entities start with the ampersand and end with the semicolon
 - Entities are used to display special characters

Some Entities in HTML

- Some entities are used to escape special characters in HTML
 - E.g., less than (<) and greater than (>) that otherwise are interpreted as tags
- Entities can also be used to represent Unicode characters
 - Useful for characters that aren't on your keyboard
 - See <http://www.w3.org/TR/REC-html40/sgml/entities.html> for full list

Double Quote	"	"
Apostrophe	'	'
Less Than	<	<
Greater Than	>	>
Ampersand	&	&

Copyright	©	©	
Registered	®	®	
Trademark	™	™	
Indian Rupees	₹	₹	
U umlaut	ü	ü	

decimal or hex



20 min

Try It Now: Basic HTML

- In Visual Studio Code, open the folder Ch02\SimpleExamples
 - Open the file index.html
- Also open the file in a web browser
 - Click the links in the table of contents
 - Which files are being displayed?
 - View the source of those files
 - Make changes and verify that you see what you expect in the browser
 - Remember to save the files with CTRL-S
 - Optimize formatting by pressing Shift-Alt-F
 - Let Visual Studio Code improve your code format for you!

Chapter Concepts

Introduction to HTML

Styling with CSS

Chapter Summary

From Attributes to Style

- In older versions of HTML, appearance was specified through attributes

```
<hr align="center" width="50%">
```

- These are deprecated in modern HTML, in favor of styles:

```
<hr style="width: 50%; margin-inline-start: auto;">
```

- This type of style is known as an **in-line style**

There is no direct equivalent of align, and setting the margin to auto (in other words, centering) is the default, but shown here for comparison

- In-line styles need to be set on every element individually
 - Hard to maintain consistency
 - When creating a website, we usually have a “house style”

Centralizing Presentation

- We can use style rules to apply a consistent style to a range of elements
 - We will explore style rules in more detail shortly
 - This rule applies the same style declarations to all `hr` elements

```
hr {  
    width: 50%;  
    margin-inline-start: auto;  
}
```

- One way to use these is to put a `style` element in the `head`

```
<head>  
...  
    <title>Breaks</title>  
    <style>  
        hr {  
            width: 50%;  
            margin-inline-start: auto;  
        }  
    </style>  
</head>
```



Try It Now: Style

15 min

- View the file Ch02\StyleExamples\index.html
 - Load it into the browser
 - Click the “Basic” link

- Also open the file basic.html
 - Delete everything to the right of font-family
 - Hit CTRL-SPACE and look at the options
 - Choose something from the list, save the file, and then click the link again
 - Try a few options—don’t neglect those further down the list (such as cursive)
 - Do the same with font-weight
 - Make sure to try both numeric and text values

Setting Style on Multiple Pages

- A common requirement is to have a consistent presentation style across many pages
 - Ideally, we would apply exactly the same style definitions
- We can do this by writing a separate stylesheet

```
hr {  
    width: 50%;  
    margin-inline-start: auto;  
}
```

style.css

- And linking to it from each HTML page's head

```
<head>  
...  
    <title>Breaks</title>  
    <link rel="stylesheet" href="style.css">  
</head>
```

Simple CSS Selectors

- The part of the CSS rule that indicates which elements it should apply to:
 - `E`: match all elements of type `E`
 - `*`: match all elements
 - Avoid this, see later
 - `#id`: match element with that id
 - `selector, selector`: match either
 - Can be multiple
 - `.class`: match all elements of class
 - See next slide

```
hr {  
    width: 50%;  
    margin-inline-start: auto;  
}  
  
* {  
    font-family: Verdana, Arial, Helvetica, sans-serif  
}  
  
#bl {  
    background-color: blue;  
}  
  
#rp {  
    background-color: rebeccapurple;  
}  
  
#bl, #rp {  
    color: white;  
}  
  
.intro {  
    color: #333;  
}
```

HTML Class

- Useful to be able to apply styles to all elements of a type and to a single element
 - But often want to apply to a collection of elements of the same or different types
 - Apply a class to the HTML elements

```
<h1 class="intro">Breaks</h1>
<p class="intro">We have already seen the p element, used to create a paragraph, but
there are other types of breaks available in HTML.</p>
```

- (Note that elements of different types can share the same class)
- Then apply style rules to that class

```
.intro {
    color: #333;
}
```

- Note that the selector starts with a dot (period or full stop)
- To select just the h1, could use h1.intro

- Classes are also useful for programmatic interaction with the DOM

Common Declarations

- Each name-value pair is called a declaration
 - Declarations end with a semicolon

Property	Description
height	Sets the height of an element. Not every element can have a defined height, see discussion of layout later. <code>height: 100px;</code>
width	Sets the width of an element, as with height. <code>width: 50%;</code>
max-height, min-height, max-width, min-width	Prevent size of element from varying outside certain limits, but otherwise let it vary with the display size.
background-color	Set the fill color of an element. <code>background-color: blue;</code>

Units for Dimensions

- There are many ways to set dimensions, these are some of the most used
 - Generally, favor relative dimensions as these will scale better
 - There are also absolute dimensions

Unit	Description
em	Font size (historically the size of capital M)
rem	Font size of root element
ex	Size of lower case x, $1\text{ex} \approx 0.5\text{em}$ in many fonts
%	Not technically a unit, usually sets relative to parent

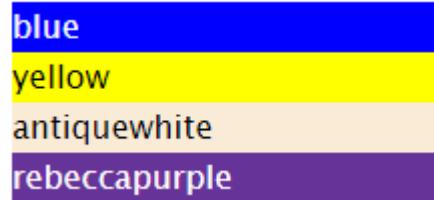
Unit	Description
px	pixel (screen dot ¹)
in	inch, defined as 96px on screens
cm	centimetre, defined as 96px / 2.54
pt	point = 1/72 nd inch (96px / 72)

¹ In fact, the standard defines a *recommended* “reference pixel”, being the whole number of actual pixels that best approximate the visual angle of a device with pixel density 96ppi at arm’s length (say, 28 inches) – approx. 0.0213 degrees

Colors

■ By (case-insensitive) name, e.g.:

- blue
- yellow
- antiquewhite
- rebeccapurple¹



■ By 3-, 4-, 6-, or 8-digit hex code

- Represent RGB (00 – ff, 0 – 255) and transparency (ff = 100% opaque)
- All these are rebeccapurple
- #639
- #663399
- #639f or #639F
- #663399ff or #663399FF

■ By `rgb` or `rgba` function (equivalent since CSS4), all these are rebeccapurple:

- `rgb(102, 51, 153)`
- `rgb(40%, 20%, 60%)`
- `rgb(102, 51, 153, 100%)`
- `rgb(102, 51, 153, 1.0)`
- `rgb(102 51 153 / 1)`

■ Using the `transparent` keyword

■ By `hsl` or `hsla` function

- Hue, saturation, and lightness model
- Easy to create family of colors
- Beyond the scope of this course, but favored by web designers

¹ https://en.wikipedia.org/wiki/Eric_A._Meyer#Personal_life

Text Styles

Property	Description
color	Sets the font color.
font-family	Sets the font of an element. If a particular font is not available, can specify fall-backs. <code>Html { font-family: Arial, Helvetica, sans-serif; }</code>
font-size	Sets the size of a font to a pre-defined keyword (such as <code>medium</code>), a percentage of the parent, or a fixed size. <code>font-size: 14px;</code>
font-style	<code>normal</code> , <code>italic</code> , or <code>oblique</code> (in most cases there is no difference between <code>italic</code> and <code>oblique</code>).
font-weight	E.g., <code>normal</code> , <code>bold</code> , or a numeric value (100, 200, 300, 400, 500, 600, 700= <code>bold</code> , 800, 900).
line-height	Vertical spacing of text, distinct from <code>font-size</code> . <code>line-height: 1.5;</code>
text-align	<code>left</code> , <code>right</code> , <code>center</code> , <code>justify</code>
text-decoration	<code>underline</code> , <code>overline</code> , <code>line-through</code>

Pseudo-Class Selectors

- Pseudo-classes may be used to match an element based on state or relationship:
 - E.g., `:hover` matches an element when the mouse is over it

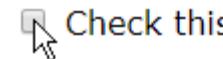
```
button:hover {  
    background: #437c43;  
}
```



- Examples for pseudo-classes include:

- `:checked`

```
input:checked + label {  
    color: #437c43;  
    text-decoration: underline;  
}
```



Label adjacent to input
(we will cover this later)

- `:valid, :invalid`

```
input:valid { background-color: #437c43; }  
input:invalid { background-color: #7c4343; }
```

Enter 3 or more characters

- `:first-child, :nth-child`

```
ol li:first-child { color: red; }  
ol li:nth-child(even) { color: blue; }
```

- Dog bites
 - Bee stings
 - Feeling sad
 - Stack traces
- Raindrops on roses
 - Whiskers on kittens
 - Bright copper kettles
 - Warm woolen mittens

This is an ordered list:

- Reusable
- Maintainable
- Clear

Styling Links

- It is better to leave link colors and cues (underlining, etc.) alone
 - But if the default blue, underlined links will be hard to read on your background, you can change the color scheme using pseudo-classes:
 - :link (all links, used for unvisited)
 - :visited
 - :hover
 - :active (being selected)
 - Always specify in order LVHA
 - Can apply limited styling to visited links to avoid information leak

```
a:link {  
    font-family: cursive;  
}  
  
a:visited {  
    color: #437c43;  
}  
  
a:hover {  
    color: #7c4343;  
}  
  
a:active {  
    text-decoration: underline overline;  
}
```

[Styled link](#)

There is no programmatic way to detect the difference between a visited and unvisited link

Pseudo-Element Selectors

- Select a part of an element
 - In contrast to pseudo-classes, which select based on state or relationship
 - Historically, both were indicated with a single colon, but they are now distinguished
- Examples of pseudo-elements:
 - ::after, ::before

```
.symbol::before {  
    content: '\2021';  
}
```

‡ This paragraph has a ::before pseudo-element.

‡ As does this one.

- ::first-letter, ::first-line
- ::selection

```
.selection::selection {  
    color: yellow;  
    background-color: rebeccapurple;  
}
```

This paragraph has a drop-cap. It usually needs a little tuning to make this work properly, so there is a new mechanism being debated, but for now, the CSS way to achieve it is with ::first-letter and float. Lorem ipsum dolor sit amet, consectetur

nisl vel pretium. Sem et tortor consequat id. Nisi est sit amet facilisis. Quam viverra orci sagittis eu volutpat odio facilisis. In hendrerit gravida rutrum quisque non tellus.

Selections in this paragraph use an unusual color. Tellus pellentesque eu tincidunt tortor aliquam. Cursus euismod quis viverra nibh cras pulvinar. Ac odio tempor orci dapibus ultrices in



15 min

Instructor Demo: Browser Developer Tools

■ Your instructor will show how to debug CSS problems including:

1. Verify that the CSS has loaded for a page with Chrome developer tools
2. Determine what CSS rules are being applied to elements
3. Change CSS properties on the fly



HANDS-ON
EXERCISE

20 min

Exercise 2.1: Applying CSS Styling

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Introduction to HTML

Styling with CSS

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Standardizing presentation with HTML and CSS
 - HTML essentials
 - Styling with CSS
 - Creating CSS in external files



Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 3: Advanced HTML and CSS

Chapter Overview

In this chapter, we will explore:

- Standardizing presentation with HTML and CSS
 - How HTML lays out a web page
 - Additional HTML elements
 - Advanced styling with CSS
 - Semantic tags
 - Forms and validation

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

Normal Flow Layout

- In the Normal Flow model, the browser positions elements in the order they are seen
- The `display` property determines how elements interact with the normal flow model:

block	The element has line breaks before and after. It normally occupies all the available width, with height being set according to contents. Can specify <code>height</code> and <code>width</code> properties.
inline	The element has no line breaks before or after. The <code>height</code> and <code>width</code> are both set by the contents and cannot be specified.
inline-block	Formatted as <code>inline</code> , but can specify <code>height</code> and <code>width</code> .
none	The element takes up no space in the page layout. Compare to <code>visibility: hidden</code> where the element still takes up space.

- There are many other options, but these are the most commonly used
- All element types have a default value for `display`

The Div and Span Tags

- The `div` element is used to break an HTML document into sections
 - Typically to give it a specific style or allow programmatic access
 - It has no specific thematic meaning and should be an element of last resort
 - By default, `div` elements are `block` elements
- The `span` element is similar, but by default, `span` elements are `inline`

```
<div class="text">
  <h2>Combinators</h2>
  <p>A paragraph as next sibling of level 2 heading</p>
  <p>A paragraph that is not the next sibling of level 2 heading</p>
  <p>A paragraph with a <span>span element as immediate child (grandchild of div)</span></p>
    <span>A span as immediate child of a div</span>
</div>
```

Combinators

A paragraph as next sibling of level 2 heading

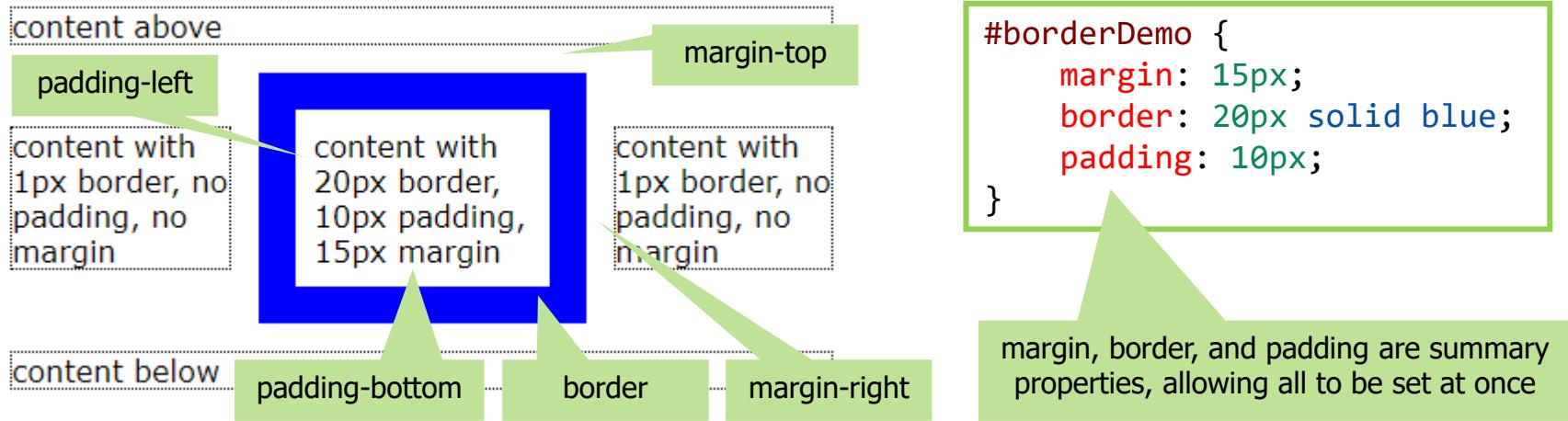
A paragraph that is not the next sibling of level 2 heading

A paragraph with a `span` element as immediate child (grandchild of div)

A span as immediate child of a div

CSS Box Model

- CSS box model allows borders to be added around elements
 - Wraps HTML elements and provides borders, padding, and margins



- By default, all sizes (`height`, `width`, or `content sizing`) apply to the content alone
 - Border and padding are added afterwards
 - Ideal in most cases, but can be hard to line up elements
 - Can change to `box-sizing: border-box`, which includes border and padding in size

Positioning Within Normal Flow

- A detailed discussion of positioning is beyond the scope of this course
- Can control position within normal flow using:
 - position property
 - Allows positioning relative to other elements or fixed on page
 - Specify position with `left`, `right`, `top`, `bottom`, and `z-index`
 - float property
 - Allows positioning to left or right of containing element
 - Text and inline elements flow around the element (e.g., an image in text)
- In many cases, these techniques are better replaced with Flex Box or Grid Layout

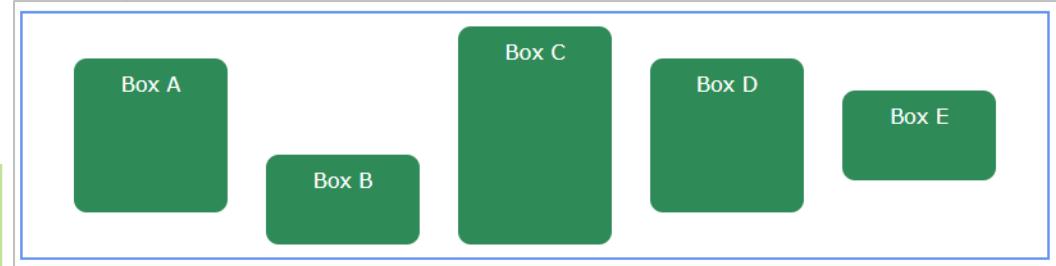
Flex Box and Grid Layout

- It can be hard to align items on the page using the box model
 - CSS provides two additional layout models
 - Specify `display: flex` or `display: grid` on a containing element
- **Flex Box** is designed to lay out items in one dimension (e.g., equally across page)
 - There is one principal direction (`flex-direction`) along which items are laid out
 - They may also be aligned along the other direction (`align-items`)
 - Control whether items resize or wrap in the available space (`flex`, `flex-wrap`)
 - https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbox
- **Grid Layout** is designed to control the overall page layout with items presented in a grid
 - Specify a grid template that includes rows and columns (`grid-template-columns`)
 - Control how rows and columns resize to fit content (`minmax`)
 - Control how items are positioned in the grid (`grid-column-start`, `grid-area`)
 - https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Basic_Concepts_of_Grid_Layout

Flex Box Example

```
#container {  
    display: flex;  
    width: 80%;  
    justify-content: space-evenly;  
    align-items: center;  
}  
  
.item {  
    max-width: 100px;  
    flex-grow: 1;  
    flex-basis: auto;  
    text-align: center;  
}  
  
.b {  
    align-self: flex-end;  
}
```

How to arrange items in the container. In this case, the principal axis is horizontal (default), and items are spaced evenly in that direction while being center-aligned on the vertical axis.



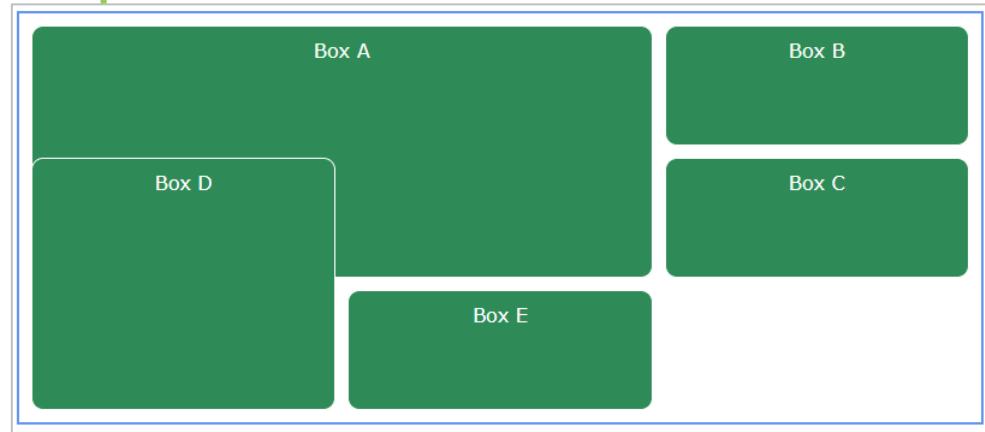
This item will be aligned differently

How items change size when the container changes size. In this case, they will all vary at the same rate (`flex-grow`) and size will be set by the item size (`flex-basis`)

Grid Layout Example

```
#container {  
    display: grid;  
    width: 80%;  
    grid-template-columns: repeat(3, 1fr);  
    grid-auto-rows: 100px;  
    grid-gap: 10px 10px;  
}  
  
#a {  
    grid-area: 1 / 1 / 3 / 3;  
}  
  
#d {  
    grid-row: 2 / 4;  
    grid-column: 1 / 2;  
}
```

How to arrange items in the container. In this case, defines a grid with 3 columns, each an equal fraction of the space. Rows are 100px high. Rows and columns have a 10px gap.



This item will span rows and columns 1 and 2

This item will span rows 2 and 3, but will fit in column 1 (alternative syntax)

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

What's in a Version?

- Earlier HTML standards had version numbers:
 - HTML 2.0, 3.2, 4.0, 4.01
 - HTML4 had 3 variations: strict, transitional, and frameset
 - After HTML4, W3C started to focus on a stricter approach (XHTML)
- WHATWG was formed because browser writers were dissatisfied with the W3C process
 - They considered it too slow
 - They felt the W3C was heading in the wrong direction with XHTML
- WHATWG maintains the HTML Living Standard (what browsers actually do)
 - <https://html.spec.whatwg.org/multipage/>
 - There are no version numbers and never will be!

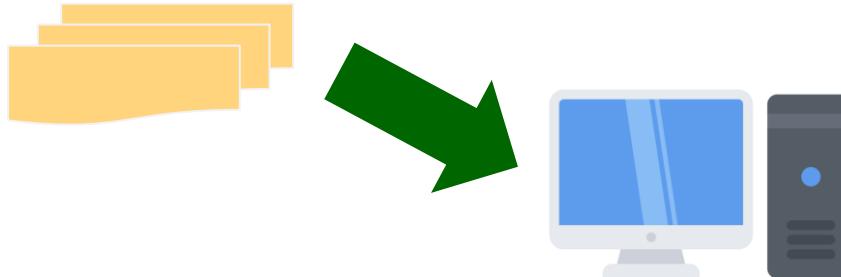
How HTML Works

- The web browser receives a document
 - Document is written in HTML
 - May have style specified using CSS (in-line or in external files)
 - May have behavior specified using JavaScript (in-line or in external files)
 - Refers to images, which are external files
- The browser starts to parse the document, constructing a Document Object Model (DOM)
 - Script files are retrieved immediately, when they are encountered
 - Possible to change this to make them load in parallel
 - Other files are loaded in parallel as parsing of the document continues



How HTML Works (continued)

- Script files are executed immediately
 - May be changed to force them to wait for the DOM to be created
 - Any script may change the DOM by adding, removing, or updating elements
 - Important to write scripts so they do not interact with DOM elements that have not been parsed yet
- Finally, the DOM is “rendered” (displayed to the user)
 - External files may still be loading at this point
 - Appearance may change as they are loaded



Older Browsers

- The original HTML5 was designed as a superset of HTML
 - Pages degrade gracefully when browsers do not support features
 - Still usable in older browsers (not all features will be present)
- As the name “HTML Living Standard” acknowledges, browsers move at their own pace
 - Before deciding to use a feature, check compatibility on websites such as:
 - <https://caniuse.com/> (and check browser usage at <https://caniuse.com/usage-table>)
 - <https://developer.mozilla.org>
- Do not attempt to check browser version – check feature support instead
 - https://developer.mozilla.org/en-US/docs/Web/HTTP/Browser_detection_using_the_user_agent
 - A common library to do this is Modernizr
- Provide compatibility using polyfills or shims
 - JavaScript libraries that provide new features in older browsers
 - CSS workarounds that minimize incompatible implementations
 - https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/HTML_and_CSS

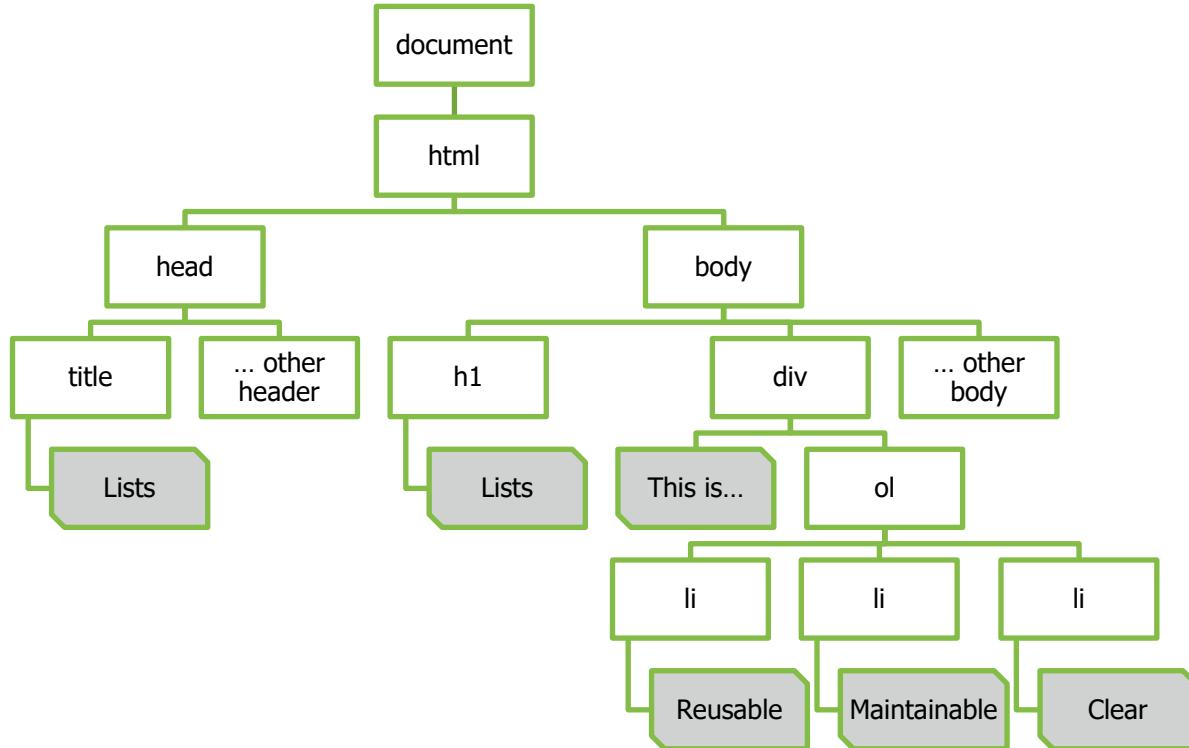
The Document Object Model (DOM)

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

<head>
  <title>Lists</title>
  ... other header content
</head>

<body>
  <h1>Lists</h1>
  <div>
    This is an ordered list:
    <ol>
      <li>Reusable</li>
      <li>Maintainable</li>
      <li>Clear</li>
    </ol>
  </div>
  ... other body content
</body>

</html>
```



Relationship-Based Selectors (Combinators)

- Selectors can be used to select elements based on their relationships to other elements

- E F
 - All F that are descendants of E (at any level)
- E > F
 - All F that are immediate children of E
- E + F
 - All F that are the next sibling of E

```
div span {  
    color: rebeccapurple;  
    font-weight: bold;  
}  
  
div > span {  
    color: #437c43;  
    font-style: italic;  
}  
  
h2 + p {  
    text-decoration: line-through;  
}
```

Combinators

A paragraph as next sibling of level 2 heading

A paragraph that is not the next sibling of level 2 heading

A paragraph with a **span element as immediate child (grandchild of div)**

A span as immediate child of a div

Attribute Selectors

- E[attr]
 - All E with attribute attr
- E[attr=value], E[attr=value i]
 - All E with attribute attr having value (exactly or case insensitive)
- E[attr^=value]
 - All E with attribute attr starting with value
- E[attr\$=value]
 - All E with attribute attr ending with value
- E[attr*=value]
 - All E with attribute attr containing value

Cascading and Specificity

- Style rules “cascade” in four ways:
 - More local styles are used in preference to external styles
 - Children inherit their parents’ styles
 - More specific rules take precedence over less specific rules
 - The last style specified at a given specificity wins
- Local styles
 - Inline styles over internal document styles over external styles
- Children inherit their parents’ styles
 - A rule for `html` will apply the rule to every element that doesn’t have a conflicting rule
- More specific rules
 - Rules specified for `#id` take precedence over `.class` over `element`
 - Pseudo-classes and attribute selectors make rules more specific
 - `element.class` is more specific than either `element` or `.class`
 - <https://css-tricks.com/strategies-keeping-css-specificity-low/>
 - The `*` selector has high specificity and is less efficient than a rule for `html`

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

iframe

■ Embed an external web page

```
<a href="colors.html" target="main">Colors</a>  
  
<iframe name="main" src="blank.html"></iframe>
```

■ Supports a sandbox attribute to specify a list of space-separated permissions:

- allow-same-origin: allow frame access to resources from same origin
- allow-scripts: run scripts, but not open popups windows
- allow-top-navigation: navigation in top-level browsing context (`target="_top"`)
 - See next slide
- allow-popups: allow frame to open popup windows (`target="_blank"`)
 - Any attempt to navigate to a disallowed context will be treated as a popup

Link Targets

- `a` element supports an optional `target` attribute
 - Indicates where to open the link

"" or omitted	Current “browsing context” (current tab or <code>iframe</code>)
<code>_blank</code>	A new tab, the tab remains anonymous (no <code>name</code>)
<code>_self</code>	Current tab or <code>iframe</code>
<code>_parent</code>	Parent if there is one, current tab or <code>iframe</code> if there isn’t
<code>_top</code>	Top-level context (usually the tab)
<code>name</code>	If <code>name</code> doesn’t exist, create a tab with that name. If <code>name</code> exists and either has same origin, or was opened by this context, then open in <code>name</code> .

- Note that browser configuration or `iframe` parameters may change the behavior
 - Pop-up blocker
 - Sandboxing

Tables

Display tabular data

Table Row

Table Header

Table Data

```
<table class="outline-only">
  <caption>Countries of the Americas whose capital city has changed</caption>
  <tr>
    <th>Country</th>
    <th>Old capital</th>
    <th>New capital</th>
  </tr>
  <tr>
    <td>Bermuda</td>
    <td>St George</td>
    <td>Hamilton</td>
  </tr>
  <tr>
    <td>Brazil</td>
    <td>Rio de Janeiro, RJ</td>
    <td>Bras&iacute;lia, DF</td>
  </tr>
  <tr> <td>Honduras</td> <td>Comayagua</td> <td>Tegucigalpa</td> </tr>
  <tr> <td>USA</td> <td>New York, NY</td> <td>Washington, DC</td> </tr>
</table>
```

Countries of the Americas whose capital city
has changed

Country	Old capital	New capital
Bermuda	St George	Hamilton
Brazil	Rio de Janeiro, RJ	Brasília, DF
Honduras	Comayagua	Tegucigalpa
USA	New York, NY	Washington, DC

Blue lines show outline
of individual elements

Table Styles

- Each cell (`td` or `th` element) is treated individually when applying styles
- `border`: cell border is not shared with adjacent cells, consider:
 - `border-collapse`
 - `border-spacing` distance between borders of adjacent cells if border is not collapsed
- Extremely common to apply different styles to alternate rows

```
table, th, td {  
    border: 1px solid black;  
    border-spacing: 2px;  
    padding: 2px;  
}
```

Countries of the Americas whose capital city has changed		
Country	Old capital	New capital
Bermuda	St George	Hamilton
Brazil	Rio de Janeiro, RJ	Brasília, DF
Honduras	Comayagua	Tegucigalpa
USA	New York, NY	Washington, DC

```
/* Collapsed borders. Default is separate */  
.collapse {  
    border-collapse: collapse;  
}  
  
/* Striped rows for readability */  
.collapse tr:nth-child(even) td {  
    background: #dedede;  
}
```

Countries of the Americas whose capital city has changed		
Country	Old capital	New capital
Bermuda	St George	Hamilton
Brazil	Rio de Janeiro, RJ	Brasília, DF
Honduras	Comayagua	Tegucigalpa
USA	New York, NY	Washington, DC

Table Structure

- Merge cells using colspan or rowspan
- All content of table is in implied tbody
 - For more control, specify an explicit thead and tbody

```
/* Striped rows for readability, but only rows
in the tbody (not in the thead) */
.interesting tbody tr:nth-child(even) td {
    background: #dedede;
}

th.old-capital {
    background: #dd99ee;
}

th.new-capital {
    background: #99eedd;
}
```

Country	Capitals	
	Old capital	New capital
Bermuda	St George	Hamilton
Brazil	Rio de Janeiro, RJ	Brasilia, DF
Honduras	Comayagua	Tegucigalpa
USA	New York, NY	Washington, DC

Countries of the Americas whose capital city has changed

```
<table class="simple interesting yellow-headers">
    <caption>Countries of the Americas whose capital
city has changed</caption>
    <thead>
        <tr>
            <th rowspan="2">Country</th>
            <th colspan="2">Capitals</th>
        </tr>
        <tr>
            <th class="old-capital">Old capital</th>
            <th class="new-capital">New capital</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Bermuda</td>
            <td>St George</td>
            <td>Hamilton</td>
        </tr>
        ...
    </tbody>
</table>
```

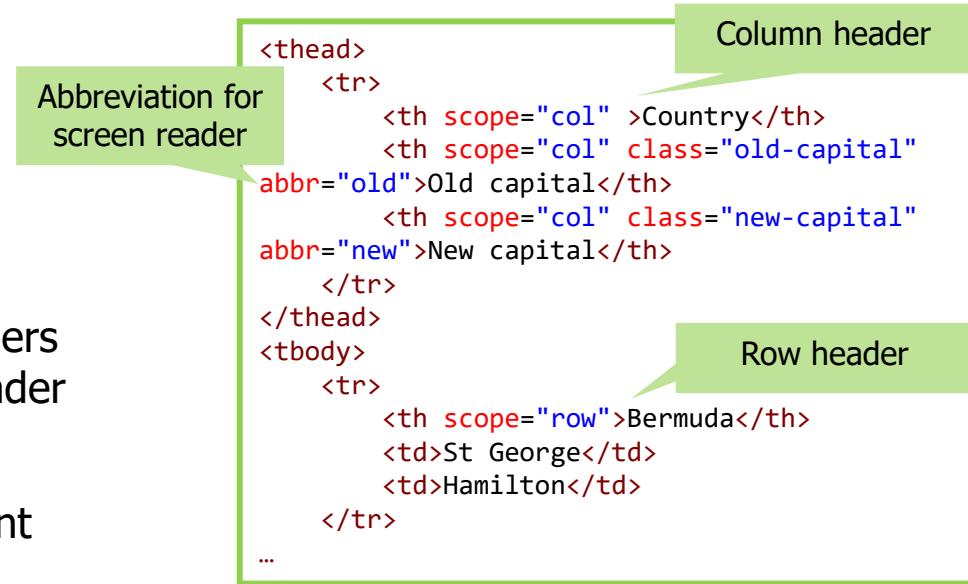
Accessibility Considerations for Tables

For screen readers:

- Provide a caption for the table
- Annotate header cell to specify whether it is for a row or a column
- Avoid complex tables with many merged cells
- Specify an abbreviation for long headers
 - Because screen readers repeat header before reading each value

Do not use tables to lay out page content

- Use them to present tabular data
- Use Flex Box and Grid Layout to control the page





10 min

Try It Now: Tables

- **Launch** Ch03\AdvancedStyleExamples\index.html
 - Click on “Tables”
 - View the output

- **Open the files** Ch03\AdvancedStyleExamples\tables.html and Ch03\AdvancedStyleExamples\css\style.css
 - Make changes to the layout and formatting
 - See the impact of changes in the browser

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

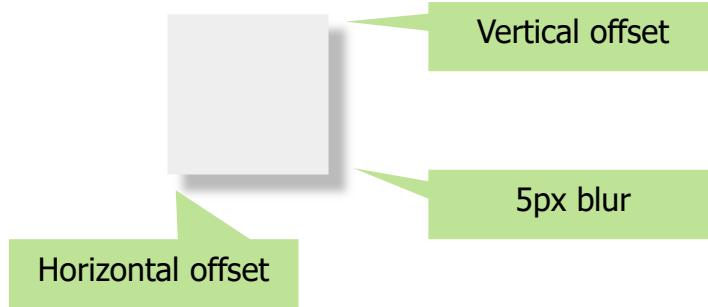
Advanced CSS Effects

- CSS Level 3 added some new features
 - Backward compatible since CSS always ignores anything unrecognized
 - Note that CSS Level 3 (or CSS3) is really a collection of standards
 - CSS3 is a useful shorthand, but not an official standard
- Enables CSS to produce effects previously only achievable with graphic tools such as PhotoShop or animation tools like Flash:
 - Advanced borders (rounded corners, shadows)
 - Advanced text decoration (wavy underlines)
 - Gradients
 - Create new shapes through transformations
 - Add “life” to interactive elements through transitions

Borders

Box shadow

- Specify horizontal offset, vertical offset, and blur of shadow



```
#shadow {  
    width: 100px;  
    height: 100px;  
    background-color: #eeeeee;  
    box-shadow: 10px 5px #bbb;  
}
```

Round rectangle

- Use border-radius

Do it!

```
#rounded-corners {  
    border: 2px solid black;  
    padding: 5px;  
    background: #eeeeee;  
    width: 100px;  
    border-radius: 15px;  
}
```

Linear Gradient

Linear gradient property declaration format is:

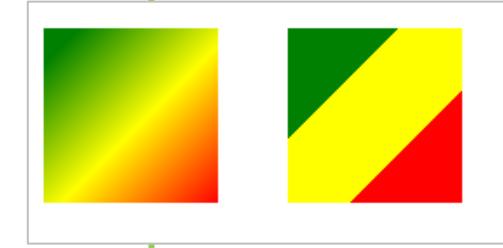
- background: linear-gradient(direction, color1, color2, color3, ...)
- direction **is a pre-defined string, starting to, or an angle**

```
#linear-gradient {  
    height: 100px;  
    width: 100px;  
    background: linear-gradient(to bottom right, green 10%, yellow, red);  
}  
  
#linear-stripes {  
    height: 100px;  
    width: 100px;  
    background: linear-gradient(to bottom right, green 32%, yellow 32% 68%, red 68%);  
}
```

Direction of gradient

Hints move the transition away from
the mid-point between the colors

Overlapping hints cause hard
boundaries between colors



- Also, repeating-linear-gradient
- conic-gradient arranges colors around a circle

Radial Gradient

Linear gradient property declaration format is:

- background: radial-gradient(shape, color1, color2, color3, ...)
- shape **is** circle **or** ellipse

```
#radial-gradient {  
    height: 100px;  
    width: 100px;  
    background: radial-gradient(circle, #0055A4, white, #EF4135, white);  
}
```

```
#radial-stripes {  
    height: 100px;  
    width: 100px;  
    background: radial-gradient(circle, #0055A4 24%, white 24% 48%, #EF4135 48% 71%, white 71%);  
}
```

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="-3 -3 6 6">  
    <circle fill="#EF4135" r="3" />  
    <circle fill="#FFFFFF" r="2" />  
    <circle fill="#0055A4" r="1" />  
</svg>
```

Can also move the center point
using, e.g., circle at top



But if you really want to create a
roundel, use SVG instead

Transitions

- In normal use, style property changes occur instantly
- Transitions allow changes to happen gradually with intermediate states created automatically
 - Do not use this for animation, it is intended for “one-off” changes



- Easing functions:
 - <https://easings.net/>
 - <http://cubic-bezier.com/#0,0,0,0>

```
#transition div {  
    width: 100px;  
    transition-property: width;  
    transition-duration: 2s;  
    margin-left: 0px;  
}  
  
#transition:hover div {  
    width: 200px;  
}  
  
#transition1 {  
    transition-timing-function: ease-in-out;  
}  
  
#transition2 {  
    transition-timing-function: cubic-bezier(0.68,  
    -0.55, 0.265, 1.55);  
}  
  
#transition3 {  
    transition-timing-function: linear;  
}
```

Apply transition when width changes

Change width on mouseover of parent

How “fast” to apply the change (easing function)

Transformations

- Change the appearance of an existing object
 - Can create new shapes
 - Can rotate text or image
- Can be combined with transitions, but this is not their primary purpose
 - Primary purpose is to give more control over presentation of individual object



```
#transform div {  
    width: 50px;  
    height: 50px;  
    background: green;  
    margin: 25px 6px;  
    display: inline-block;  
}  
  
#transform:hover #transform1 {  
    transform: translateY(-20px);  
}  
  
#transform:hover #transform2 {  
    transform: rotate(30deg);  
}  
  
#transform:hover #transform3 {  
    transform: skewX(-30deg);  
}
```

Draw squares

On mouseover,
transform squares



HANDS-ON
EXERCISE

20 min

Exercise 3.1: Applying Advanced CSS Styling Effects

- Please complete this exercise in your Exercise Manual



HANDS-ON
EXERCISE

30 min

Exercise 3.2: Static Web Pages (Optional)

- This is an optional exercise
 - Your instructor will decide if you will do this exercise during class time
- This exercise can be found in your Exercise Manual

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

Adding Document Structure with Semantic Elements

- Historically, web authors used a variety of means to signal the meaning of their content
 - E.g., `<div class="footer">`
 - Easy for humans to understand, not useful for screen readers
- HTML now contains many elements known as semantic elements
 - Convey better meaning of role of an element in a document
- Examples include:
 - header, footer
 - nav
- Benefits of new elements include:
 - More maintainable HTML
 - Easier to style with CSS
 - Better search engine rankings

body, head, and Overall Structure

- body and head are **not** semantic elements, they are required parts of an HTML document
 - May be omitted, but all practical documents include body and head
 - If included, there may be exactly one of each
 - head must precede body
- The heading levels, h1-h6, represent headings for a section of a document
 - Also, not semantic elements, but level should not be chosen for presentation reasons
 - Should start at a high level and work down if hierarchical headings are required
- <header>
 - Used to contain the headline for a page or section (typically an h1-h6)
 - May also contain navigation information or other useful introductory material
 - Can have several header elements per document
- <footer>
 - Contains summary information for a page or section

Semantic Elements

<nav>

- Contains navigation links
- Type of navigation may be determined by where it is
 - Global site navigation as direct child of body
 - In-page navigation within an article or section
- Not all links need to be in a nav

<article>

- Represents self-contained content, e.g., for syndication
- Should make sense on its own regardless of other content on the page
- Nested articles make sense in some cases (e.g., replies to a blog post)

<section>

- Contains content that should be grouped together
- May be used to break up articles
- Use when thematically correct, not as a styling convenience

Semantic Elements (continued)

■ <main>

- Dominant contents of a document
- Must not be more than one unless hidden

■ <aside>

- Content related to the page but can exist independently
- Typically used for sidebars

■ <figure>

- Used to display images, audio, tables, code examples, etc.
- Enables addition of a caption
- Figures may be nested to create a montage

■ <figcaption>

- Used to define caption for a figure
- Must be first or last child of a <figure>

Semantic Elements Example

- The purpose is to define the structure of the document
- Semantic elements do not have default styles
 - However, they are defined as block-level elements

```
<body>
  <header>
    <h1>Semantic Elements</h1>
    <nav><a href="">Top-level Navigation</a></nav>
  </header>
  <article>
    <header>
      <h2>First Article</h2>
    </header>
    <section>
      <p>Something interesting.</p>
    </section>
    <section>
      <p>Something else interesting.</p>
    </section>
  </article>
  <footer>
    Copyright © 2019 The Company
  </footer>
</body>
```

Header for page

Header for article

Sections within article

Footer for page

Exercise 3.3: Creating an HTML5 Page Using Semantic Tags



20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

HTML Forms

- HTML forms are collections of input controls that have a single Submit action
 - A submit button sends form data to the `action`, a URL on the server
 - Most forms have `method="post"`, which sends form data in the body
 - Not all input controls must be in forms, but if the intention is to send the data to a server, then a form is appropriate
 - Controls have a `name`, and this is the key when values are sent to the server
- Input controls may have a label associated with them
 - Strongly prefer a label over a simple text element
 - Screen readers can understand the relationship between the label and the control
 - Either explicit with `for` keyword
 - Or implicit by enclosing the control

```
<label for="name">Name:</label>
<input type="text" id="name" name="userName">
```

```
<label>Name:
    <input type="text" id="name" name="userName">
</label>
```

Structure Within Forms

- Within a form, controls can be grouped using <fieldset>
 - Optional first child can be a <legend>
 - If supplied, appears in the <fieldset> frame and is used by screen readers

```
<fieldset>
  <legend>Contact Method:</legend>
  <input type="radio" id="byEmail" name="contact" value="email" checked>
  <label for="byEmail">Email</label>
  <input type="radio" id="byPhone" name="contact" value="phone">
  <label for="byPhone">Phone</label>
</fieldset>
```

Contact Method:

Email Phone

- Can use <fieldset> as a purely thematic grouping by hiding the border
 - Advantageous for screen readers
- Radio buttons do not have to be related by a <fieldset>
 - Radio groups are defined by the name, not the <fieldset>

HTML Form Example

```
<form action="results.html">
  <label for="name" class="col1">Name:</label>
  <input type="text" id="name" name="userName" class="col2">
  <label for="email" class="col1">E-mail:</label>
  <input type="email" id="mail" name="userMail" class="col2">
  <label for="phone" class="col1">Phone:</label>
  <input type="phone" id="phone" name="userPhone" class="col2">
  <label for="msg" class="col1">Message:</label>
  <textarea id="msg" name="userMessage"></textarea>
  <input type="checkbox" id="optin" name="optIn" class="col1">
  <label for="optin" class="col2">Please contact me with
special offers</label>
  <div class="col1">By:</div>
  <fieldset class="col2">
    <input type="radio" id="byEmail" name="contact" value="email" checked>
    <label for="byEmail">Email</label>
    <input type="radio" id="byPhone" name="contact" value="phone">
    <label for="byPhone">Phone</label>
  </fieldset>
  <button type="submit" class="col2">Send your message</button>
</form>
```

Name:

E-mail:

Phone:

Message:

Please contact me with special offers

By: Email Phone

Send your message

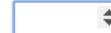
HTML5 Forms

- HTML5 provides new features that simplify building robust interactive forms
 - Specific input fields for certain data types
 - Declarative validation and error messages using element attributes
- As a result, the variety of input types has increased
 - For example:
 - Number
 - Email
 - Telephone number
 - URL
 - Date
 - Time
 - Color
 - Search

HTML Input Types

type="..."	Description
button, reset, submit	A simple button, one that resets the form, or one that submits the form.
color, date, file, time	Provide special selector according to the type of value.
checkbox	A simple on/off choice. <code>value</code> attribute specifies the "on" value, otherwise <code>on</code> .
number	Numeric input. Generally has a spinner on the right. Supports <code>min</code> and <code>max</code> .
radio	Grouped into a radio group where controls have a common <code>name</code> attribute. <code>value</code> attribute specifies the "on" value, otherwise it is <code>on</code> .
range	A numeric value between <code>min</code> and <code>max</code> , use where precision is not important.
tel	A telephone number, but no validation is usually imposed. User agents, such as mobile browsers, can provide special interaction.
text, search, email, url	Simple single line text input. <code>search</code> is functionally identical, but may be styled differently. The <code>email</code> and <code>url</code> versions validate according to the appropriate standard (e.g., email addresses must have an @ sign), but not that the value exists.
hidden	Value is submitted with the form, but not visible.
password	Single line text input with obscured text.

Please contact me



Email Phone



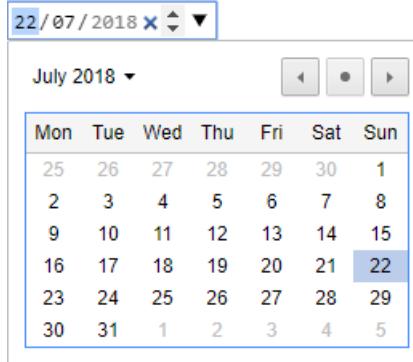
Password (8 characters minimum):



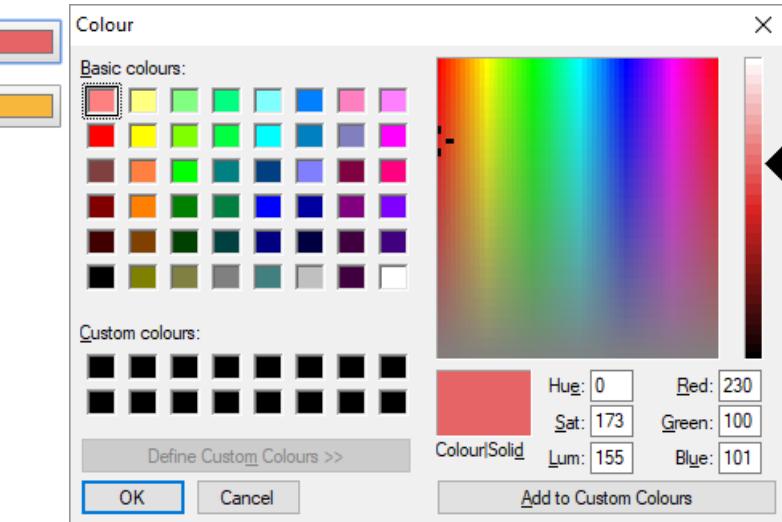
Date, Time, and Color Pickers

Date

Start date:



Color



Time

Choose a time for your meeting:

09 : 03 Office hours are 9am to 6pm

Also variations for date and time, month and year, and week number

Other Form Controls

■ <textarea>

- A multi-line field
- By default, <textarea> uses a monospaced font

■ <select>

- A <select> control is a drop-down list
- It contains a series of <option> entries representing the individual values
- It may contain <optgroup> entries to group <option> values

■ <button>

- A clickable button
- Supports an optional type that takes a value submit (**default**), reset, or button
- Functionally equivalent to an input control with type="button"
- Allows more versatile styling than input control

Form Validation

Some input types have built-in validation

- `required` indicates that an input element must have a value
- `title` can be used to customize the error message

Phone:

Message:  Please match the requested format.
Number must follow North American format

Otherwise validation can be added through attributes:

- `pattern` is a regular expression that must be matched
- `minlength` and `maxlength` for text fields
- `placeholder` is text that will appear faintly inside the field when it is empty
 - *Do not* use this instead of a label

```
<input type="phone" id="phone" name="userPhone"  
      required  
      pattern="[2-9][0-9]{2}-[0-9]{3}-[0-9]{4}"  
      placeholder="XXX-XXX-XXXX"  
      title="Number must follow North American format">
```

Phone:

Pseudo-classes `:invalid` and `:valid` can also be used for visual feedback



Exercise 3.4: Creating an HTML Form with Validation

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

HTML Layout

The Document Object Model

More HTML and CSS

Advanced CSS

Semantic Tags

Forms and Validation

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Standardizing presentation with HTML and CSS
 - How HTML lays out a page
 - Additional HTML elements
 - Advanced styling with CSS
 - Semantic tags
 - Forms and validation

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 4: Client-Side JavaScript Programming

Chapter Overview

In this chapter, we will explore:

- Adding behavior to a web page with JavaScript
- Basic JavaScript syntax
- Manipulating a document via DOM
- Handling events
- The JavaScript security model

Chapter Concepts

Why JavaScript?

Basic Syntax

Working with DOM

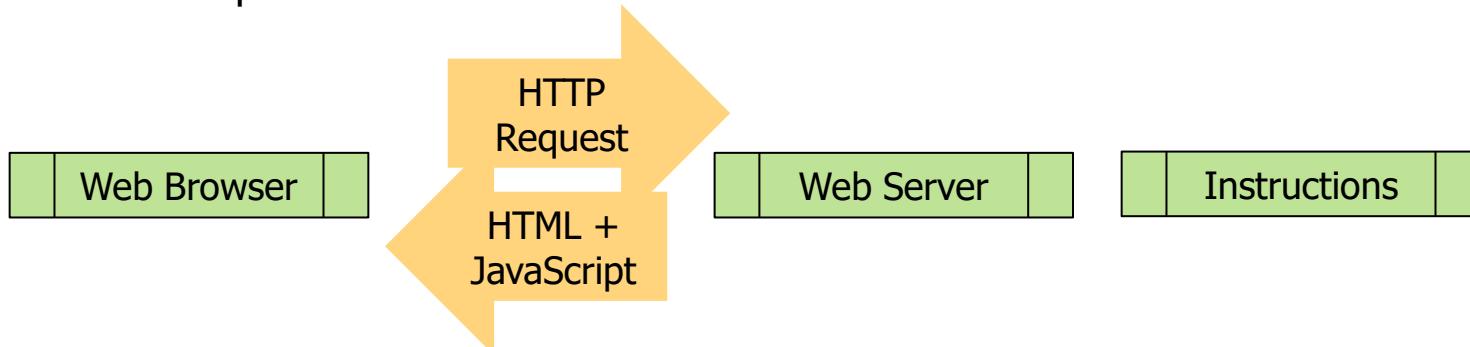
Handling Events

Web Application Security Client Side

Chapter Summary

Behavior

- JavaScript is a way of adding behavior to web pages
 - HTML5 makes JavaScript a standard
 - Prefer JavaScript to Flash, etc.
- A dynamic web application uses:
 - HTML for structure
 - CSS for presentation
 - JavaScript for behavior
 - This is what provides interaction



Static vs. Dynamic

- Most web pages do not serve out the exact same content to users
 - For example, different users have different bank balances
 - Cannot use a static HTML page to serve out account information
- Three types of dynamic behavior:
 - Client-side dynamic behavior achieved through the use of JavaScript
 - Browser executes script on client's machine
 - Server-side dynamic behavior where the server runs logic in reaction to a user query
 - Often written in Java, C# or a template language like PHP
 - Typically, user completes some action (e.g., fills out a form)
 - Web page sends a request to the server where any associated data is processed
 - Server generates HTML including appropriate data and returns it
 - Ajax (Asynchronous JavaScript and XML) is a hybrid
 - Server-side sends data in XML (or, more commonly now, in JSON) to client side
 - JavaScript code on client side works with this data

Recap: How HTML Works

- The web browser receives a document
 - Document is written in HTML
 - May have style specified using CSS (in-line or in external files)
 - May have behavior specified using JavaScript (in-line or in external files)
 - Refers to images, which are external files
- The browser starts to parse the document, constructing a **Document Object Model (DOM)**
 - Script files are retrieved immediately, when they are encountered
 - Possible to change this to make them load in parallel
 - Other files are loaded in parallel as parsing of the document continues



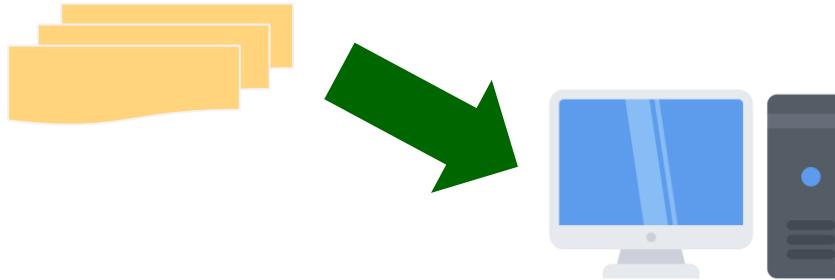
Recap: How HTML Works (continued)

Script files are executed immediately

- May be changed to force them to wait for the DOM to be created
- Any script may change the DOM by adding, removing, or updating elements
- Important to write scripts so they do not interact with DOM elements that have not been parsed yet

Finally, the DOM is “rendered” (displayed to the user)

- External files may still be loading at this point
- Appearance may change as they are loaded



What JavaScript Can Do

- JavaScript is a full-fledged programming language
 - Use it to carry out computations
 - Is the entered date a weekday?
 - Use it to make conditional decisions
 - On what to display, when to display it, etc.
 - Use it to repeat processing
 - If user changes a value, recalculate total
- Use JavaScript to provide a rich, interactive experience
 - Runs directly in the browser
 - Avoids round-trip to server, or page refresh

Where JavaScript Goes

- Place JavaScript within a script tag
 - Can occur anywhere in the document, for now put it in the <head>

```
<script>  
    // code goes here  
</script>
```

- The JavaScript can also be loaded from external URL
 - Unlike many HTML elements, it requires the closing tag

```
<script src=".js/imageCycle.js"></script>
```

The code in the js file must not have <script> tags

- Having JavaScript in a separate file improves:
 - Reusability: if you have multiple pages but want to use same or similar functions
 - Readability: increases the readability of your HTML code
- In this course, most JavaScript will be in external files

Example JavaScript: Cycling Images

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<link rel="stylesheet" href="./css/style.css">
<title>JavaScript Example</title>

<script src=".js/imageCycle.js"></script>

</head>

<body onload="cycleBanner()">
<h1>Banner Images</h1>
<p>This image below should cycle repeatedly and clicking on the image at any time should take you to an appropriate page</p>
<a href="javascript:getImageDestination()"></a>
</body>
</html>
```

```
var banner = ['images/600x399-1.jpg', 'images/600x399-2.jpg', ...];
var bannerDestination = ['Algodones_Dunes', 'Antelope_Canyon', ...];
var imageno = 0;
function cycleBanner() {
    if (++imageno == banner.length) {
        imageno = 0;
    }
    document.bannerImage.src = banner[imageno];
    // change to next image every 3 seconds
    setTimeout('cycleBanner()', 3000);
}
function getImageDestination() {
    document.location.href = 'https://en.wikipedia.org/wiki/' +
        bannerDestination[imageno];
}
```



5 min

Try It Now: JavaScript

- View the file Ch04\ImageCycle\image-cycle.html in VSC
 - Load it in a browser and verify that it works

Chapter Concepts

Why JavaScript?

Basic Syntax

Working with DOM

Handling Events

Web Application Security Client Side

Chapter Summary

Variables

- Variables in JavaScript are untyped
 - Will be coerced into appropriate type at runtime
 - Are case sensitive
- Naming conventions vary according to the programmer's first language

```
let javaProgrammerHere = 10.32;
let $usually_write_perl = 'Hello';
let _systemsProgrammer = true;
```

- Best practice is to use Java naming conventions
 - Map better to XML and JSON
- Uninitialized variables are assigned a special value of `undefined`
 - Can also assign a variable to `null`

Statements and Operators

■ JavaScript belongs to the C-family of languages

- Like C, C++, C#, Java, etc.
- Statements end in a semicolon
- Use curly braces to group a set of statements
- Arithmetic operators: +, -, *, /, %, ++, --, +=, -=
- Comparison operators: ==, !=, >, >=
- Logical operators: &&, ||, !
- Ternary operator

```
var greeting = 'Good ' + (hour < 12) ? 'Morning' : 'Evening';
```

■ Comments

- // comment until end of line
- /* block comment */
- <!-- HTML comment also works -->

Loops and Conditions

- Loops and conditions are also like the C-family of languages

We'll talk about
this shortly

```
var numEntries = 100;  
while (numEntries > 10) {  
    // do something  
    --numEntries;  
}
```

```
for (let numEntries = 100; numEntries > 10; --numEntries) {  
    // do something  
}
```

```
if (numEntries > 100) {  
    alert('Too many people in exhibit.');//  
} else if (numEntries == 0) {  
    closeExhibit();  
}
```

String in JavaScript

- The String API is very close to that of Java

```
var name = 'O\'Keefe';

var lastChar = name.charAt(name.length - 1);

var aposPosition = name.indexOf('\''); // -1 if not there

var num = parseInt('6');
```

- Strings may use single or double quotes
 - Single quotes generally preferred in applications that handle a lot of embedded HTML
 - Because double quotes are used in HTML
 - Consistency is most important

Arrays

- Arrays grow dynamically
 - Unlike in C or Java
 - More like Java ArrayList
 - Syntax is generally like C array
 - Note the array initializer is [] rather than {} as in Java and C

```
// dynamically grows
var itemCosts = new Array();
itemCosts[0] = 51.2;
itemCosts[1] = 60.6;

// alternative, note: [], not {}
var itemCosts = [51.2, 60.6];

// length of an array
var numItems = itemCosts.length;

// resize array
itemCosts.length = 1;

// sparse array allowed
var itemCosts = [51.2,,,60.6];
```



Exercise 4.1: Jumping JavaScript

10 min

- Please complete this exercise in your Exercise Manual

Functions

- Functions are declared with the keyword `function`
- Parameters are always passed by *value*
 - For simple types (numbers, strings), changes to parameters don't affect arguments
 - But if parameter is a reference to an object, function can modify the referenced object

Parameter is a number

Parameter is reference to an object

```
function calcTotal(price, taxRate) {  
    price = price * (1.0 + taxRate);  
}  
let cost = 100;  
calcTotal(cost, 0.06);  
console.log(cost); // 100
```

Argument was not changed

```
function calcTotal(bill, taxRate) {  
    bill.total = bill.charges * (1.0 + taxRate);  
}  
let check = {  
    charges: 100,  
    total: 0  
};  
calcTotal(check, 0.06);  
console.log(check.total); // 106
```

Object's property was changed

Variable Scope

- Variables declared with `var` are in the scope of the function that encloses them
 - Global variables are those outside of any function
 - Function parameters are within the scope of the function
- What is the scope of each of the variables below?

```
var fullDetails = false;
function showDetails(zipcode) {
    var url = '../jaxrs/zipcode/json/' + zipcode;
    if (fullDetails) {
        var request = new XMLHttpRequest();
        request.open('GET', url, false);
        request.send();
    }
};
```

- Note that variables declared without `var` are always global: avoid this!

```
function showDetails(zipcode) {
    url = '../jaxrs/zipcode/json/' + zipcode; // oops global
    ...
}
```

Other Variable Scoping Rules

- Newer browsers understand the keywords `let` and `const`
 - Support true block-level scope, variable scope is set by the `{}` that contains it

```
var fullDetails = false;
function showDetails(zipcode) {
    const url = '../jaxrs/zipcode/json/' + zipcode;
    if (fullDetails) {
        let request = new XMLHttpRequest();
        request.open('GET', url, false);
        request.send();
    }
};
```

- Generally better than `var`, but it depends on the application scope
 - In a corporate environment, they are usually supported
 - In the wider world, supported by about 96% of all browser installations
 - For this, and other features, check <https://caniuse.com>

Careful with Global Variables!

- Global variables are all within the same global scope
 - Even if they are loaded from different files:

```
var fullDetails = false;
function showDetails(zipcode) {
    var url = '../jaxrs/zipcode/json/' + zipcode;
    if (fullDetails) {
        ...
    }
};
```

show-zipcodes.js

```
var fullDetails = new Object();
function updateDetails(zipcode) {
    fullDetails.zipcode = zipcode;
}
```

update-zipcodes.js

- Best practice is to reduce/eliminate global variables

JavaScript Idioms

- The `||` operator evaluates to `false` if a variable is `undefined`

First “true” value is assigned

```
var numEntries = numStocks || pref.portfolioLength || 10;
```

- Note that `0`, `null`, `NaN` (not a number) and `''` all also evaluate to `false`
- Can also use the special value `undefined`

```
if ( numEntries == undefined ){
    numEntries = 10;
}
```

- Three uses of `=`

```
numEntries = 3;           // assignment
if (numEntries == 3);     // check equality
if (entry === this.entry); // strict equality
```

Objects

- Objects in JavaScript are not quite as formal as objects in Java or C#
 - Simply a collection of properties and methods
 - In fact, one alternative syntax is like a Perl associative array

```
// creating an Object and setting its values
var item = new Object();
item.id = 23024;
item.cost = 60.6;

// alternative Perl-like syntax
var item = new Object();
item['id'] = 23024;
item['cost'] = 60.6;

// object-literal alternative syntax
var item = {id: 23024, cost: 60.6};
```

Objects Are Hierarchical

- Objects can be composed of other objects

```
var item = new Object();
item.id = 23024;
item.cost = 60.6;
item.location = new Object();
item.location.lat = 35.3;
item.location.lon = -97.1;
```

Objects Are Like Associative Arrays

- Objects are also like associative arrays
 - Can loop through all the properties of an object

```
var item = {  
    id: 23024, cost: 60.6,  
    location: {lat: 35.3, lon: -97.1}  
};  
var prop;  
for (prop in item) {  
    if (prop == 'cost') {  
        // do something with item[prop]  
    }  
}
```

- Properties can be added or deleted

```
item.salePrice = item.cost * 1.05;  
delete item.location;
```

Defining “Classes”

- JavaScript idiom for constructor, private properties, and public methods

```
function Item(id_, cost_, price_) {  
    var id = id_;  
    var cost = cost_;  
    var price = price_;  
    this.getId = function () {  
        return id;  
    }  
    this.getCost = function () {  
        return cost;  
    }  
    this.getPrice = function () {  
        return price;  
    }  
    this.getProfit = function () {  
        return (price - cost);  
    }  
}
```

```
var item = new Item(23021, 30.2, 31.4);  
var profit = item.getProfit();
```

Dialog Box

- To pop up a dialog box:

```
alert('Please enter a valid state code');
```

- Use sparingly as most users find alerts annoying

- For confirmation:

```
var resp = confirm('Are you sure?');
if (resp) {
    // delete all the work
}
```

- To get a single value from the user

```
var resp = prompt('How many zipcodes?', '10');
if (resp != null && resp != '') {
    numZips = parseInt(resp);
}
```



HANDS-ON
EXERCISE

15 min

Exercise 4.2: JavaScript Arrays and Objects

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Why JavaScript?

Basic Syntax

Working with DOM

Handling Events

Web Application Security Client Side

Chapter Summary

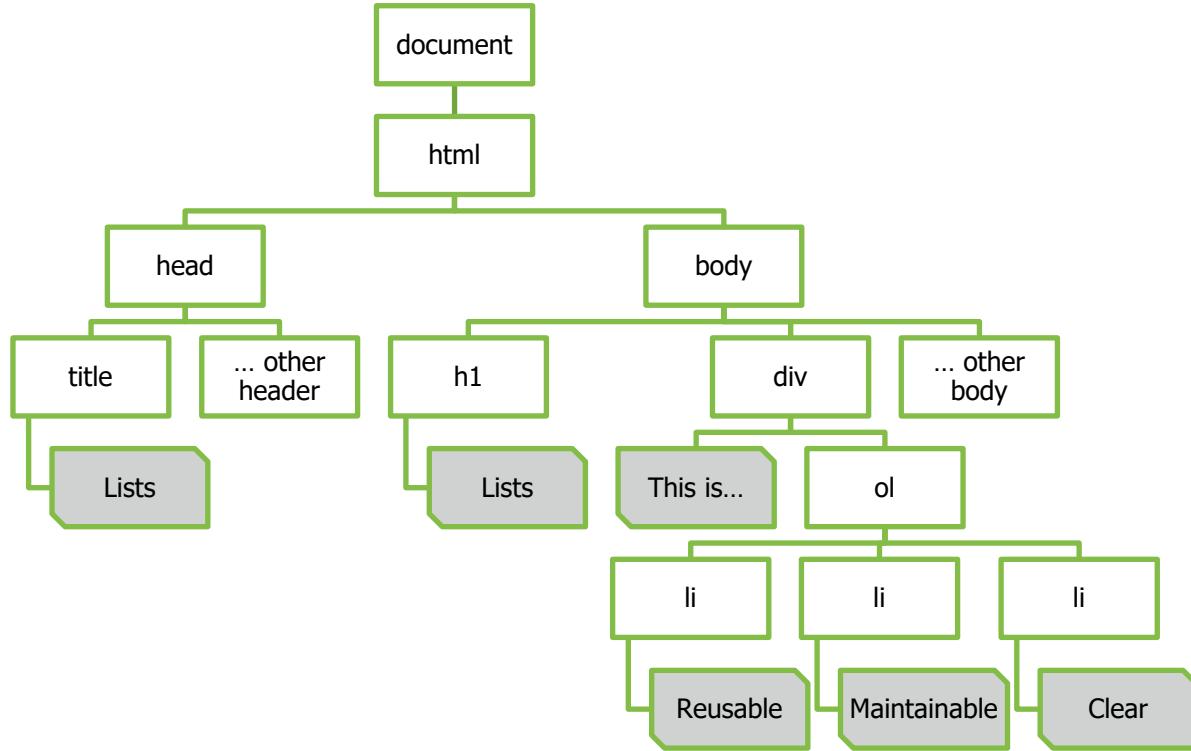
The Document Object Model (DOM)

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

<head>
  <title>Lists</title>
  ... other header content
</head>

<body>
  <h1>Lists</h1>
  <div>
    This is an ordered list:
    <ol>
      <li>Reusable</li>
      <li>Maintainable</li>
      <li>Clear</li>
    </ol>
  </div>
  ... other body content
</body>

</html>
```



Steps to Manipulate the DOM

- Can use JavaScript to manipulate the DOM using the DOM API

- Write a JavaScript function
 - In a script tag or in an external file sourced from a script tag
 - Function will access one or more DOM elements using the API
- Specify that function is to be invoked when a particular event occurs
 - onload, onmousemove, etc.
 - We will cover these events later
- Write HTML page with the required elements

```
function insertSong() {  
    numZips = getLoopLength();  
    document.getElementById('song').innerHTML  
        = createSong(numZips);  
}
```

```
<body onload="insertSong()">  
    ...  
</body>
```

```
<h1>10 Little Zipcodes Event Driven</h1>  
<div id="song">  
    Song replaces this text when JavaScript finishes executing.  
</div>
```

Manipulating the DOM

```
<body onload="insertSong()">
```

Run this JavaScript code when the element has been loaded. Important not to try to manipulate the DOM before the document has been properly parsed.

```
<h1>10 Little Zipcodes Event Driven</h1>
```

```
<div id="song">
```

Song replaces this text when JavaScript finishes executing.

```
</div>
```

```
</body>
```

The id of element

```
function createSong(zipCount) {  
    var song = '';  
    while (zipCount > 0) {  
        song += zipCount + ' little zipcodes jumping on the bed<br>;  
        song += ' one fell down and broke his head.<br>;  
        --zipCount;  
    }  
    song += 'No more zipcodes jumping on the bed<br>;  
    return song;  
}
```

The id of element

```
function insertSong() {  
    document.getElementById('song').innerHTML = createSong(10);  
}
```

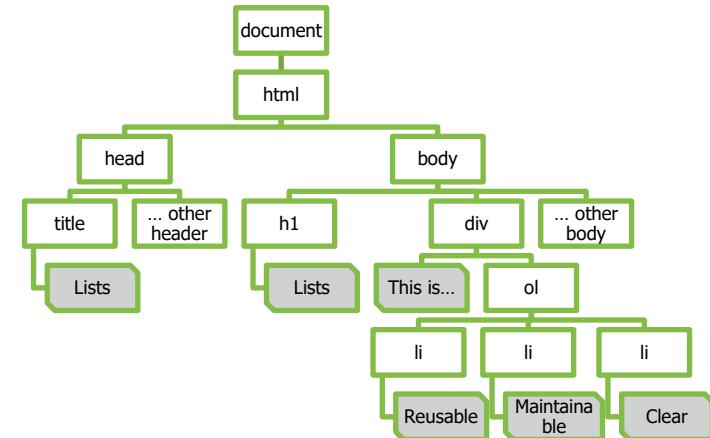
Alternative way of declaring
the event handler (we will
cover this later)

```
window.onload = insertSong;
```

Not `document.write()` but ...

DOM API

- The DOM is represented as a tree of nodes and elements
 - Every HTML element is a node
 - Nodes are also created for element contents (e.g., textual content)
- The normal entry point is the global `document` object to get one or more elements
- Methods that return a “live” list of elements
 - `getElementsByClassName()`
 - `getElementsByTagName()`
- Method that returns a single element
 - `getElementById()`
- Methods that return a static list of nodes from CSS selectors
 - `querySelector()`
 - `querySelectorAll()`



Examples of Manipulating Elements

- Replace the tree below an element
 - All the HTML contained in it
- Change the HTML class associated with an element
 - It is really a list of classes, better to use `classList`
- Change the contents of a text node
- Change an attribute

```
document.getElementById('song').innerHTML = song;
```

```
document.getElementById('songtitle').className = 'highlight';
```

```
var classes = document.getElementById('songtitle').classList;  
classes.add('highlight');  
classes.remove('highlight');
```

```
var title = document.getElementsByTagName('title')[0];  
title.childNodes[0].nodeValue = 'No more zipcodes!';
```

```
document.getElementById('ad').setAttribute('src', urlToShow);
```

Navigating from an Element

■ Can navigate from an Element:

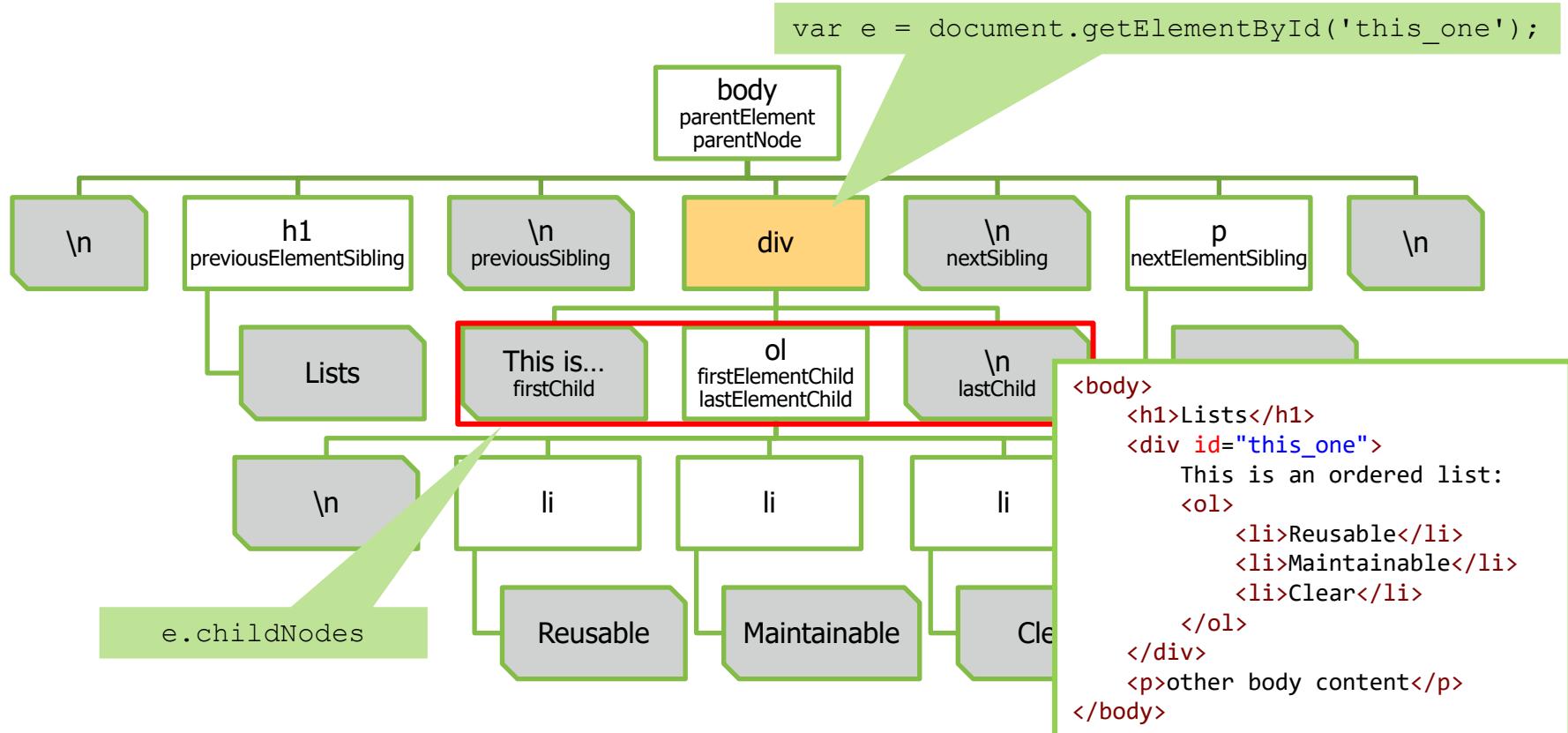
- childNodes
- firstChild, firstElementChild
- lastChild, lastElementChild
- nextSibling, nextElementSibling
- parentNode, parentElement
- previousSibling, previousElementSibling

■ To remove an element from the document:

```
var sep = document.getElementById('separator');
sep.parentNode.removeChild(sep);
```

- Similarly, appendChild(), replaceChild(), insertBefore()

Relative Position in the DOM





HANDS-ON
EXERCISE

10 min

Exercise 4.3: Manipulating the DOM

- Please complete this exercise in your Exercise Manual

JavaScript Built-in Classes

Math

```
var lower = Math.floor(32.2); // 32  
var rnd = Math.random(); // 0 to 1
```

Date

```
var now = new Date();      // a Date object  
var alsoNow = Date.now(); // a number (milliseconds since epoch)  
var fixedTime = new Date(2019, 1, 31, 12, 1, 31, 300);  
var timeString = fixedTime.toLocaleString('de-DE');
```

Number

```
var maxInt = Number.MAX_SAFE_INTEGER;  
var num = Number.parseFloat("1234.56");  
var output = num.toFixed(1);           // "1234.6"  
var exp = num.toExponential(3);       // "1.235e+3"  
var fmt = num.toLocaleString('fr-FR'); // "1 234,56"
```

Optional after month,
default to lowest value

Prefer primitive numbers, but Number
is helpful for type conversions



15 min

Exercise 4.4: Working with Built-In Classes

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Why JavaScript?

Basic Syntax

Working with DOM

Handling Events

Web Application Security Client Side

Chapter Summary

The `onload` Event Handler

■ Have already seen this

- Define a JavaScript function to execute when the page has finished loading
- `onload` declares an event handler for the `load` event
- *Note:* `onload` attribute in `body` tag and `window.onload` are the same

■ Can also use `onload` with other items that are “loaded”, such as:

- `img`
- `iframe`

■ Why?

- Remember JavaScript executes as soon as it is encountered

HTML Parsing

Parsing paused

Fetch script

Execute script

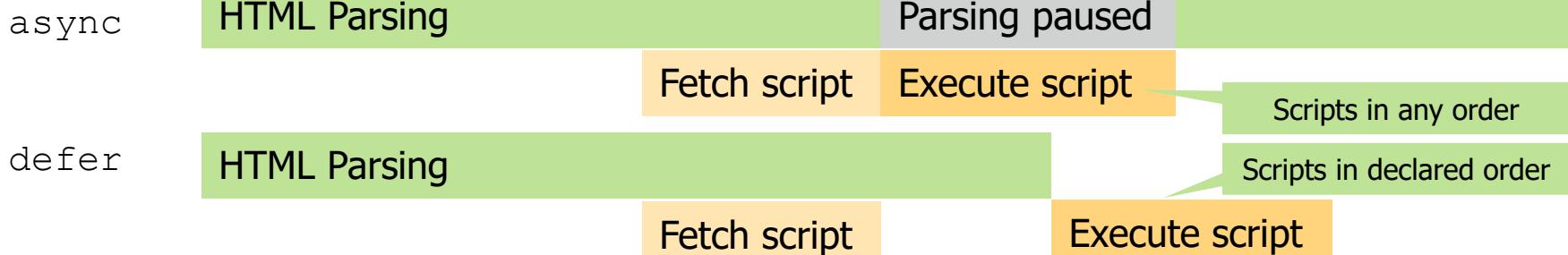
- If the script refers to a DOM Element that has not been parsed, it will fail

When?

- When is the `load` event fired?
 - After document has been parsed *and* other resources (images etc.) have been loaded
- If the purpose is to wait until the DOM has been parsed, there are better options
 - `DOMContentLoaded` (no "on" attribute)

```
window.addEventListener('DOMContentLoaded', (event) => {  
    console.log('DOM fully loaded and parsed');  
});
```
 - `defer` and `async`

```
<script defer src="randomDefer.js"></script>
```



JavaScript Event Handlers

- Here are some of the event handlers that can be registered:

Event handler name	Event name	Example of when it would be triggered
onabort	abort	User aborts the loading of an image
onblur	blur	Item loses focus
onchange	change	User checks on/off a checkbox
onclick	click	User clicks item
oncontextmenu	contextmenu	User right-clicks, before menu is shown
ondblclick	dblclick	User double-clicks item
onerror	error	Link/URL not available
onfocus	focus	Item gains focus
onload	load	Page is loaded
onmouseenter	mouseenter	When mouse enters the area of an element

Attaching Event Handlers

- Prefer assigning event handlers in code: keep separate from HTML

- Attach event handlers after the DOM has been parsed

- Either use `defer` on a script
- Or assign the handler in the `onload` event handler

Function reference, no parentheses

```
window.onload = function () {
    document.getElementById('active').onmouseover = insertSong;
}
```

- This code uses an anonymous function
 - Hard to test, restrict use to trivial code that does not need to be tested, as here

- As we saw earlier, can also attach a handler using `addEventListener`

```
document.getElementById('active').addEventListener('mouseover', insertSong);
```

- Handlers can be attached to a wider range of objects
- Some events cannot have handlers attached any other way

Passing Parameters to an Event Handler

- As already noted, the event handlers are function references
 - There are no parentheses and no parameters

Function reference, no parentheses

```
document.getElementById('clickMe1').onclick = insertNewSong;
```

- The handler may be defined with an optional parameter of type `Event`

```
function insertNewSong(e) {  
    console.log(e);  
}
```

- To pass other parameters, need to create an anonymous function

```
document.getElementById('clickMe2').onclick = function() {  
    insertNewSong(2);  
}
```

As well as constants, has access to any variables in scope when it is defined

This is the event handler, not the function it calls



HANDS-ON
EXERCISE

15 min

Exercise 4.5: Responding to Events

- Please complete the following exercise in your Exercise Manual

Event Bubbling this and event.target

- After an event triggers on an element, it then triggers on its parents in nesting order
- The deepest element (the one triggered) is the target
 - In W3C compliant browsers: event.target
 - In IE8: event.srcElement
- The element the event has bubbled up to is this
 - The element currently handling the event

```
<div id="d1">
    Grandparent div 1
    <div id="d2">
        Parent div 2
        <div id="d3">
            Child div 3
        </div>
    </div>
</div>
```

```
let divs = document.getElementsByTagName('div');
for (let i = 0; i < divs.length; i++) {
    divs[i].onclick = function (e) {
        e = e || event;
        let target = e.target || e.srcElement;
        console.log('target=' + target.id + ', this=' + this.id);
    }
}
```

Stopping Events

- Can stop event bubbling using the method `event.stopPropagation()`
 - In IE8, this was `event.cancelBubble()` on the global event object
- This may seem like a good idea sometimes, but it almost never is
 - Avoid doing it! Find another way
 - Stopping event propagation makes it hard to have event handlers for standard behavior
 - Hard to predict every single case when events should, or should not, propagate
- In most cases, there is a way to achieve a similar effect by stopping the default behavior `event.preventDefault()`
 - E.g., stop a form being submitted, prevent a checkbox from changing state
 - Events that can be stopped have `event.cancelable == true`
 - Can check whether it has been done: `event.defaultPrevented`
 - Event handlers can decide for themselves whether they should handle such an event



Event Bubbling

5 min

- Your instructor will demonstrate event bubbling

Form Validation

- JavaScript used to be the only way to validate forms
 - Since HTML5, most validation is better done declaratively in HTML
- Continue to use JavaScript for more advanced validation
 - Cross-field validation
 - Validation that cannot be implemented as a regular expression or limit
- When working with an individual form control, can override validation
 - `setCustomValidity(message)`
 - If `message` is not empty, this marks the control as invalid and displays `message`
 - Must manually reset validity when the control value changes
 - Enables the pseudo-classes `:invalid` and `:valid` to give visual feedback

Form Events

Common events in form validation

Event	Triggered When...
change	... a form control changes value. For a text field, this is when tabbing out or clicking away. For a SELECT, when choosing a value. For a checkbox or radiobutton, when clicking on the control.
input	... the item value changes (for every change).
keydown, keyup	... a key is pressed in the control. These events occur before change or input.
invalid, valid	... validation is run, depending on the result.
reset	... the reset button is pressed.
submit	... the submit button is pressed.

- The submit event is a common place for complex validation
 - Use preventDefault() to stop the form being submitted
- Or use the submit button click event and call setCustomValidity() on a control

Exercise 4.6: Form Validation with JavaScript (Optional)



20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Why JavaScript?

Basic Syntax

Working with DOM

Handling Events

Web Application Security Client Side

Chapter Summary

How JavaScript Security Works

- Executable content within web pages are a security issue
 - Much more than static content
 - Arbitrary code can be executed on user's computer
- Have to be aware of security constraints on JavaScript
- Who executes the code?
 - The browser (trusted by the user)
- Where is the code?
 - Served out from a website (not trusted by the user)
- JavaScript security relies on the fact that the code is interpreted and run by a trusted program
 - Not possible for JavaScript code to access resources that the JavaScript engine does not allow access to

JavaScript Sandbox

- JavaScript does not provide:
 - Any way to write or delete files or directories on client computer
 - No way to plant a virus on user's system
 - No way to delete a user's data
 - Any networking primitives
 - Has a way to load URLs and send HTML form data to URLs
 - But cannot open a socket connection
 - Therefore, safe when accessed behind firewall
- The sandbox model fails if JavaScript can control a browser plugin which can write or delete files or open socket connections
 - This is a frequent issue with Microsoft ActiveX controls

JavaScript Restrictions

- Browsing history
 - The JavaScript program can tell the browser to navigate `back()`, `forward()`, or `go()`, but the actual URL being navigated to is not known to the program
 - Otherwise, privacy issues arise
- Restricts the reading of files
 - The filename to be uploaded can be set by the user via a dialog box
 - The filename cannot be set by the script
 - Otherwise, the script can read arbitrary files on user's machine
- Privacy of email addresses
 - The script cannot submit a form to a mailto: URL
- Protection of browser sessions
 - A script cannot close a window it did not open
 - Windows opened by script have to be larger than 100px on a side, cannot be larger than screen size and cannot be moved off screen

Same-Origin Restriction

- Prevent event spoofing
 - A script cannot set properties of an Event object
 - Cannot register listeners for documents loaded from a different server than the script
 - A specific case of what is known as the same-origin restriction
- Cannot access predefined properties of client-side objects from a different origin
 - If your web page is served out from `widgets.acme.com`, the only documents you have unrestricted access to are documents from `widgets.acme.com`
- If you are serving out a document from `widgets.acme.com` and you want the document to be available to scripts in documents served out from `sales.acme.com`, then you need to explicitly set the domain property on both the documents to `acme.com`

```
document.domain = 'acme.com';
```

Chapter Concepts

Why JavaScript?

Basic Syntax

Working with DOM

Handling Events

Web Application Security Client Side

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Adding behavior to a web page with JavaScript
- Basic JavaScript syntax
- Manipulating a document via DOM
- Handling events
- The JavaScript security model

Key Points

- JavaScript is a way of adding behavior to web pages
 - Can be used to carry out client-side computation and to update the web page in-place
 - JavaScript is a dynamically typed language
 - Syntax is like the C-family of languages
- The browser parses a HTML document and creates a DOM
 - Can manipulate the DOM using JavaScript
 - Can assign a JavaScript function to be called when an event happens
- JavaScript security relies on a sandbox where dangerous operations are restricted
 - Events, document properties, etc. are subject to a same-origin policy

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 5: Working with jQuery

Chapter Overview

In this chapter, we will explore:

- JavaScript is very powerful
 - But core API is not that large
 - Many reusable classes and functions available as frameworks
- Why should we use these frameworks rather than rolling our own?
 - Robustness
 - These frameworks are very widely used
 - Speed of development
 - Can quickly build web applications
 - Maintainability
 - Less code to maintain
 - Easier for new developers to get onboard on a project when popular frameworks are employed in project
 - Cost considerations
- One of the most popular JavaScript frameworks is jQuery

Chapter Concepts

Getting Started with jQuery

jQuery Selectors

Event Methods

Filtering

Ajax with jQuery

Chapter Summary

jQuery

- jQuery one of the most popular JavaScript libraries
 - Provides cross-browser support and many utilities
- To use jQuery:
 - Download `jquery.js` and place it within your web application
 - From your HTML page, do:

```
<script src='../../lib/jquery-3.4.1.min.js'></script>
```

- A better idea is to use a Content Delivery Network (CDN)
 - User will not need to repeatedly download jQuery
 - Different libraries are available: minified, non-minified, slim, slim-minified
 - *Note:* use non-minified for course → easier debugging

```
<script src='https://code.jquery.com/jquery-3.4.1.js' integrity='sha256-CSXorXZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo=' crossorigin='anonymous'></script>
```

<http://jquery.com/>

Prevents 'spoofing' attacks

jQuery Syntax

- jQuery syntax is specialized for selecting HTML elements and performing some action
- Basic syntax is: `$(selector).action()`
 - A `$` to access jQuery
 - A `(selector)` to find HTML elements
 - A `.action()` something to be done with that element
- Examples:
 - `$(this).hide()` – hides the current element
 - `$('.div').hide()` – hides all `<div>` elements
 - `$('.museum').hide()` – hides all elements with class='museum'
 - `$('#song').hide()` – hides the element with id='song'

The Document Ready Event

- All DOM related actions require that the selected DOM element is already loaded
 - For this purpose, it is a good idea to make sure that the DOM has been parsed

```
$(document).ready(function() {  
  
    // jQuery methods go here...  
  
});
```

- Alternative equivalent syntax

```
$(function() {  
    // jQuery methods go here...  
});
```

- With this code it also does not matter where your script is located in your HTML page

Chapter Concepts

Getting Started with jQuery

jQuery Selectors

Event Methods

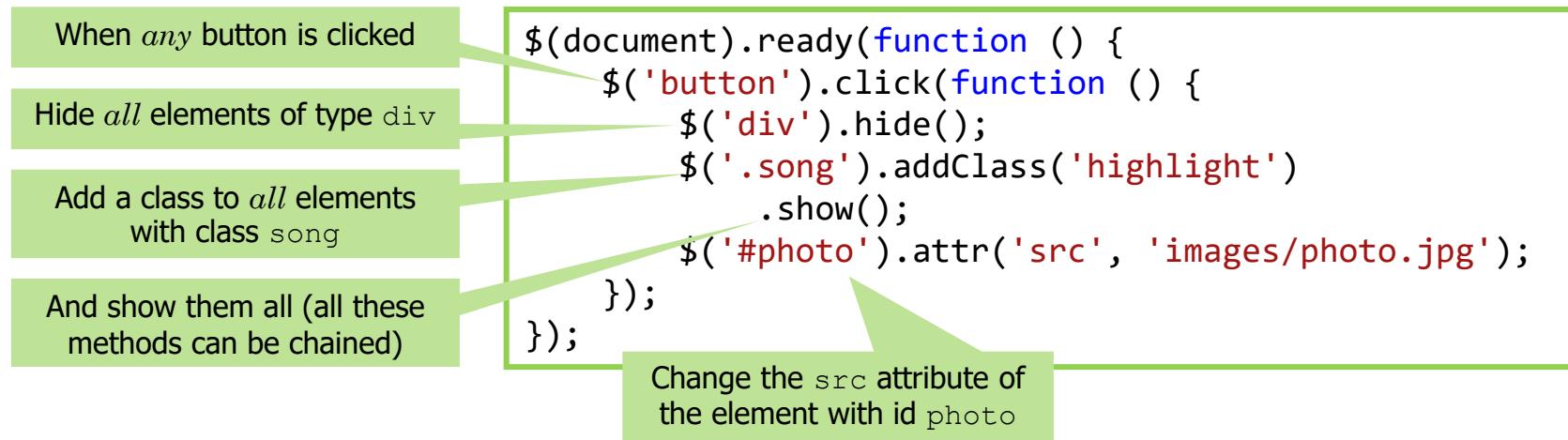
Filtering

Ajax with jQuery

Chapter Summary

jQuery Selectors

- There are multiple ways to access your HTML elements
 - Most jQuery Selectors are also CSS Selectors, and *all* CSS Selectors are jQuery Selectors
 - Just like in CSS, jQuery Selectors address *all* matching elements
- jQuery Selectors are enclosed in the dollar-parentheses syntax: `\$()`



More Selectors

Selector	Description	Selector	Description
<code>\$('*')</code>	All elements	<code>\$('[href]')</code>	All elements with an <code>href</code> attribute
<code>\$('.p.intro')</code>	All <code>P</code> elements with class="intro"	<code>\$('.a[target=_blank"]')</code>	All <code>A</code> elements with a <code>target</code> attribute value equal to <code>_blank</code>
<code>\$('.p:first')</code>	The first <code>P</code> element	<code>\$('.a[target!=_blank"]')</code>	All <code>A</code> elements with a <code>target</code> attribute value not equal to <code>_blank</code>
<code>\$('.ul li:first')</code>	The first <code>LI</code> element of the first <code>UL</code>	<code>\$('.button')</code>	All <code>BUTTON</code> elements and <code>INPUT</code> elements of <code>type="button"</code> , but not <code>reset</code> or <code>submit</code>
<code>\$('.ul li:first-child')</code>	The first <code>LI</code> element of every <code>UL</code>		
<code>\$('.div:contains(Greg)')</code>	All <code>DIV</code> elements that contain the text "Greg"		
<code>\$('.tr:even')</code>	All even <code>TR</code> elements		
<code>\$('.tr:odd')</code>	All odd <code>TR</code> elements		

Converting To and From jQuery

- Do not mix jQuery and non-jQuery when the same outcome can be achieved either way
 - Confuses people
- Convert any Element, or array of Elements, into a jQuery object:
 - `$(this)`
 - `$(varName)`
 - Also, to a limited degree, plain JavaScript objects: e.g., allowing events to be attached
- Get the Element(s) underlying a jQuery object:

Code	Description
<code>\$('.div').get(0)</code>	Get the first DIV
<code>\$('.div')[0]</code>	Alternative syntax
<code>\$('.div').get(-1)</code>	Get the last DIV (cannot use alternative syntax in this case)
<code>\$('.div').get()</code>	Get all DIV Elements



HANDS-ON
EXERCISE

15 min

Exercise 5.1: First Steps with jQuery

- Please complete this exercise in your Exercise Manual

jQuery Equivalents

■ jQuery equivalents for some common DOM interactions

- jQuery interface is made up of methods rather than properties
- Where the “property” is writable, there are two versions of the method
 - One takes a value (setter), one does not (getter)
- `.html([htmlString])`
 - Get or set the HTML of the element
 - Equivalent to `innerHTML` property
- `.val([value])`
 - Get or set the value of the element(s),
e.g., contents of a form control
 - Equivalent to `value` property
- `.text([string])`
 - Get or set the text value
 - Setter escapes any embedded HTML
- `.attr(attributeName[, value])`
 - Get or set value of attribute
- `.css(propertyName[, value])`
 - Get or set value of the CSS property
- `.css(propertyObject)`
 - Set all the properties
 - Object contains property-value pairs

Chapter Concepts

Getting Started with jQuery

jQuery Selectors

Event Methods

Filtering

Ajax with jQuery

Chapter Summary

jQuery Event Handling

- We have already used `.click()`
- Unlike JavaScript event handlers, jQuery event handlers are named directly for the event
 - They are functions that take the handler as a parameter, rather than properties
 - Anonymous function

```
$('button').click(function () {  
    $('div').hide();  
});
```

As with JavaScript event handlers, this is a function reference, not a function invocation (there are no parentheses)

- Named function

```
function clickHandler() {  
    $('div').hide();  
}
```

```
$('button').click(clickHandler);
```

- Generally, prefer the named function
 - It is easier to test

Commonly Used Event Methods

■ Among the events that have a jQuery event handler method with the same name as the event are some familiar names:

- blur
- change
- click
- contextmenu
- dblclick
- keydown, keyup
- focus
- mouseenter
- submit

■ The following do not have one:

- input
- invalid, valid
- reset

■ In addition, there is the jQuery-specific `hover()` event handler

- Maps to `mouseenter` and `mouseleave`

```
$('button')
  .hover(applyColorChange, removeColorChange);
```

Simple Event Binding

- Can bind any event handler using the `on()` method
 - Pass a space-separated list of event names to bind one handler to all the events
- There is also a version that accepts an object literal
 - Bind multiple handlers in a single call

```
$( 'img' ).on( 'click', function () {
    console.log( 'clicked on an image' );
});
```

```
$( 'div' ).on({
    mouseenter: function () {
        console.log( 'mouse entered a div' );
    },
   mouseleave: function () {
        console.log( 'mouse left a div' );
    },
    click: function () {
        console.log( 'clicked on a div' );
    }
});
```



HANDS-ON
EXERCISE

30 min

Exercise 5.2: Putting It All Together

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Getting Started with jQuery

jQuery Selectors

Event Methods

Filtering

Ajax with jQuery

Chapter Summary

Filtering with jQuery

- Real-world web pages are composed of a large, complex DOM tree
 - Web developers often need to filter out unwanted DOM elements from search results
- jQuery provides support for complex element selection with methods like the following:
 - `.eq()`, `.first()`, `.last()`
 - `.children()`, `.contents()`, `.find()`
 - `.filter()`
 - `.has()`
 - `.not()`
- In most cases, jQuery also provides extended pseudo-classes (`:eq()`, etc.)
 - It is usually more efficient to select using a native selector and then filter the selection using these methods than to use the extended pseudo-classes

The .eq(), .first(), and .last() Methods

The .eq() method

- Filter elements by an index
- Indexing is 0-based
- A negative index starts from the end

```
$('li').eq(2).css('color', 'seagreen');  
$('li').eq(-1).css('color', 'coral');
```

```
<ul>  
  <li>To forgive and forget means to throw away dearly bought experience.</li>  
  <li>Oh, what a tangled web we weave...when first we practice to deceive.</li>  
  <li>The real problem is not whether machines think but whether men do.</li>  
  <li>We must accept finite disappointment, but never lose infinite hope.</li>  
  <li>You cannot think about thinking without thinking about thinking about something.</li>  
</ul>
```

Also:

- .first() → .eq(0)
- .last() → .eq(-1)

- To forgive and forget means to throw away dearly bought experience.
- Oh, what a tangled web we weave...when first we practice to deceive.
- The real problem is not whether machines think but whether men do.
- We must accept finite disappointment, but never lose infinite hope.
- You cannot think about thinking without thinking about thinking about something.

The `.children()`, `.contents()`, and `.find()` Methods

- `.children()`, `.contents()`, and `.find()` all replace the list with a list of children
 - We will see some examples on the following slides
- `.children(selector)`
 - Returns all the direct element children
 - The selector is optional
- `.contents()`
 - Returns all the direct children including text nodes
- `.find(selector)`
 - Returns all the descendants (at any level) that match the selector

The `.filter()` Method

- Use the `.filter()` method to select desired elements from a jQuery object
 - Can take a simple selector as a parameter
 - Or a function that filters out elements when it returns false
- The following example performs these steps:
 - Collects all the `li` elements inside all the `div` elements
 - Keeps every third element (note, `index` starts at 0)
 - Applies a css style to the remaining elements

```
$('div')
    .find('li')
    .filter(function (index) {
        return index % 3 == 2;
})
.css({
    'font-style': 'italic',
    'color': 'goldenrod'
});
```

- To forgive and forget means to throw away dearly bought experience.
 - Oh, what a tangled web we weave...when first we practice to deceive.
 - *The real problem is not whether machines think but whether men do.*
 - We must accept finite disappointment, but never lose infinite hope.
 - You cannot think about thinking without thinking about thinking about something.
-
- *To forgive and forget means to throw away dearly bought experience.*
 - Oh, what a tangled web we weave...when first we practice to deceive.
 - The real problem is not whether machines think but whether men do.
 - *We must accept finite disappointment, but never lose infinite hope.*
 - You cannot think about thinking without thinking about thinking about something.

The `.has()` Method

- The `.has()` method filters elements based on their descendants
 - Descendants are specified by a selector
 - Or jQuery object
 - This example selects every `h2` that contains a `span`

This finds all `span` elements in the document and checks to see if any are descendants of any `h2`.

An h2 element without a span

Aliquam ut porttitor leo a. At varius vel pharetra vel turpis in fermentum posuere. Eget nulla facilisi etiam dignissim.

An h2 element with a span

Mi in nulla posuere sollicitudin aliquam. Consequat nisl neque viverra justo nec. Fames ac turpis egestas integreget. Varius duis at consectetur lorem. Non pulvinar ne

```
let spans = $('span');
$('h2')
    .has(spans)
    .css({
        'font-family': 'sans-serif',
        'color': 'seagreen'
    });

```

- By contrast, the selector `h2 span` selects all spans that are descendants of an h2



The .not() Method

- The .not() method filters out undesired elements from the target element list
- This example:
 - Selects all the 'div' elements in the page that do not have the class `wisdom`
 - Then applies the css style to the element children of those 'div' elements

```
$('div')
  .not('.wisdom')
  .children()
  .css({
    'font-style': 'italic',
    'color': 'brown'
 });
```

- Note, this is not equivalent to the native selector `div:not(.wisdom)` since it excludes non-element children, but if it used `contents()`, the selector would be more efficient

Chapter Concepts

Getting Started with jQuery

jQuery Selectors

Event Methods

Filtering

Ajax with jQuery

Chapter Summary

Ajax

- AJAX stands for Asynchronous JavaScript and XML
 - A mechanism by which a web page can request data from a server
 - The basis of single-page applications where the content changes, but the page does not
 - An acronym, but now usually rendered Ajax rather than AJAX
- Asynchronous JavaScript and XML
 - The process does not block the user because it is asynchronous
 - The code that requests the data is written in JavaScript
 - XML was the original transport format of choice
 - Now largely replaced by JSON (JavaScript Object Notation)
- Enabled by the XMLHttpRequest object
 - Often abbreviated to XHR

Sending Off an Ajax Request Without jQuery

- The parameters to open():
 - HTTP method (usually GET or POST)
 - The URL
 - Whether to send asynchronously
 - false: block and wait for result
- Request is created and populated, but only sent on call to send()
- HTTP status of 200 indicates success
 - 2XX codes are success, 3XX and 4XX indicate failure (e.g., 404 is page not found)
 - Can get error message from request.statusText
- Can get full response as either text or XML (request.responseXML)

```
const request = new XMLHttpRequest();
request.open('GET', url, false);
request.send();
if (request.status === 200) {
    const result = JSON.parse(request.responseText);
    // process response
    document.getElementById('details')
        .innerHTML = '...';
} else {
    console.log('Error: ' + request.statusText);
    document.getElementById('details')
        .innerHTML = 'Error';
}
```

Asynchronous Request

- When sending asynchronous request, need to register an event handler
 - Here, an anonymous function is registered
 - Make sure to register handler before sending request (why?)

```
const request = new XMLHttpRequest();
request.open('GET', url, true);
request.onreadystatechange = function () {
    if (request.readyState === 4) {
        if (request.status === 200) {
            const result = JSON.parse(request.responseText);
            // process response
            document.getElementById('details').innerHTML = '...';
        } else {
            console.log('Error: ' + request.statusText);
            document.getElementById('details').innerHTML = 'Error';
        }
    } else {
        console.log('readyState: ' + request.readyState);
        document.getElementById('details').innerHTML = 'In progress';
    }
}
request.send();
```

Function defined anonymously. Can access the XMLHttpRequest object by the variable name in the enclosing scope (`request`), or as `this`.

readyState	Meaning
0	Not initialized
1, 2	Loading
3	Communicating with server
4	Complete

JSON

- JSON has largely displaced XML in Ajax
 - XML is complex and verbose
 - JSON is simple and compact
- Although the name (JavaScript Object Notation) may imply that JSON objects are JavaScript objects, that is not strictly true
 - More accurate to say that JSON is *inspired* by JavaScript

JSON Format

- JSON objects do look very like JavaScript object literals
 - Enclosed in braces { }
 - Name-value pairs separated by commas
 - Name and value separated by a colon
- But they differ
 - The property name is always enclosed in double quotes
- Values may be:
 - String (enclosed in double quotes)
 - Number
 - Another object (enclosed in { })
 - An array (enclosed in [])

```
{  
  "firstName": "Joe",  
  "lastName": "Smith",  
  "title": "Mr.",  
  "company": "Dev Inc.",  
  "jobTitle": "Developer",  
  "employeeNumber": 1,  
  "primaryContactNumber": "+359777123456",  
  "otherContactNumbers": [  
    "+359777456789",  
    "+359777112233"  
  ],  
  "primaryEmailAddress": "joe.smith@xyz.com",  
  "emailAddresses": [  
    "j.smith@xyz.com"  
  ],  
  "groups": [  
    "Dev",  
    "Family"  
  ]  
}
```

How to Access JSON Data

- `JSON.parse()` function turns a JSON string into a JavaScript object

```
const result = JSON.parse(request.responseText);
// process response
const message = result.title + ' ' + result.firstName + ' ' + result.lastName + ' '
+ ' works at ' + result.company + ' as ' + result.jobTitle;
```

- The resulting object has a property for every name-value pair
- JSON is case sensitive

- The opposite process can be achieved by `JSON.stringify()`

Simple Ajax with jQuery

jQuery simplifies Ajax:

One of several Ajax shorthand methods:
jQuery.get(), jQuery.post(), etc.

```
$getJSON(url, function(result) {  
    const message = result.title + ' ' + result.firstName + ' ' + result.lastName + ''  
    + ' works at ' + result.company + ' as ' + result.jobTitle;  
    $('#details').text(message);  
});
```

Some things to note:

- There is no selector: \$ is actually an alias for the jQuery object itself
- The function is only called on success: need another way of handling errors
- JSON is automatically parsed, and the resulting object is passed to the function
- Here the code is trivial, if you need more functionality, use a named function
 - It can be tested more easily

The getJSON Function

The syntax for the getJSON function:

```
$.getJSON(url [, data] [, callback]);
```

The parameters for getJSON:

- **url**: The URL of the server-side resource contacted via the GET method
- **data**: An object whose properties serve as the name-value pairs used to construct a query string to be appended to the URL, or a preformatted and encoded query string
- **callback**: A function invoked when the request succeeds
 - `function callback(data, textStatus, jqXHR)`
 - **data**: The parsed JSON string from the response body
 - **textStatus**: The description of the result, same meaning as in XHR
 - **jqXHR**: A jQuery extension to the XHR object, may be useful to get access to the headers, for example

What about failures?

Handling Outcomes from Ajax Operations

- Functions like `getJSON` return the `jqXHR` object as well as passing it to the callback

- In addition to wrapping a standard XHR, `jqXHR` is a Promise
 - An object that may have a value in the future
 - Implements a well-defined interface for interacting with the future value

jQuery Promise interface

- `.done(onSuccess)` is another way of specifying the success callback
 - `.fail(onFailure)` is called for any sort of failure
 - `.always(onCompletion)` is always called on completion
 - `.then(onSuccess, onFailure)` is shorthand for `.done()` and `.fail()`
 - In all cases, the parameters are functions with the following signatures:
 - `onSuccess(data, textStatus, jqXHR)`
 - `onFailure(jqXHR, textStatus, errorThrown)`
 - `onCompletion` receives the parameters of `onSuccess` or `onFailure`, depending on outcome

jQuery Promises

```
$.getJSON(url)
  .done(function (result) {
    const message = result.title + ' ' + result.firstName + ' ' + result.lastName + ' '
      + ' works at ' + result.company + ' as ' + result.jobTitle;
    $('#details').text(message);
  })
  .fail(function (jqXHR, textStatus, errorThrown) {
    $('#details').text('Error');
    console.log('getJSON request failed: ' + textStatus + ' (' + jqXHR.status + ')');
    console.log(errorThrown);
  })
  .always(function () {
    console.log('getJSON request ended');
  });
});
```

The post Function

- To send Ajax data to a server, use the http POST method
 - Implemented by the jQuery post function:

```
$.post(url [, data] [, callback] [, type]);
```

- The parameters for post:
 - url: The URL of the server-side resource contacted via the GET method
 - data: An object whose properties serve as the name-value pairs passed in the body of the request (in other words the data sent to the server), or a preformatted string
 - When working with forms, a suitable string can be produced by calling serialize()
 - E.g., const str = \$('#contactform').serialize();
 - callback: A function invoked when the request succeeds, as before
 - type: The type of data passed to the callback function:
 - 'xml', 'html', 'script', 'json', 'jsonp', or 'text'
- Again, it returns the jqXHR

Other Ajax Functions

- As well has `$.post()`, there is `$.get()` with similar options
- `$.ajax()` allows any request
 - All the other methods are wrappers round this method
 - Full control over headers
 - Http methods other than GET and POST
- `$.param(object)`
 - Returns an encoded parameter string based on the properties of `object`



HANDS-ON
EXERCISE

30 min

Exercise 5.3: Ajax with jQuery

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Getting Started with jQuery

jQuery Selectors

Event Methods

Filtering

Ajax with jQuery

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- JavaScript is very powerful
 - But core API is not that large
 - Many reusable classes and functions available as frameworks
- Why should we use these frameworks rather than rolling our own?
 - Robustness
 - These frameworks are very widely used
 - Speed of development
 - Can quickly build web applications
 - Maintainability
 - Less code to maintain
 - Easier for new developers to get onboard on a project when popular frameworks are employed in project
 - Cost considerations
- One of the most popular JavaScript frameworks is jQuery

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 6: Introduction to Angular

Chapter Overview

In this chapter, we will explore:

- What is Angular?
- TypeScript
- A 'Hello World' Angular application
- Automated testing with Jasmine and Karma
- Writing tests using Angular-specific utilities

Chapter Concepts

What Is Angular?

Introducing TypeScript

A Simple Angular Application

Angular Test Utilities

Chapter Summary

What Is Angular?

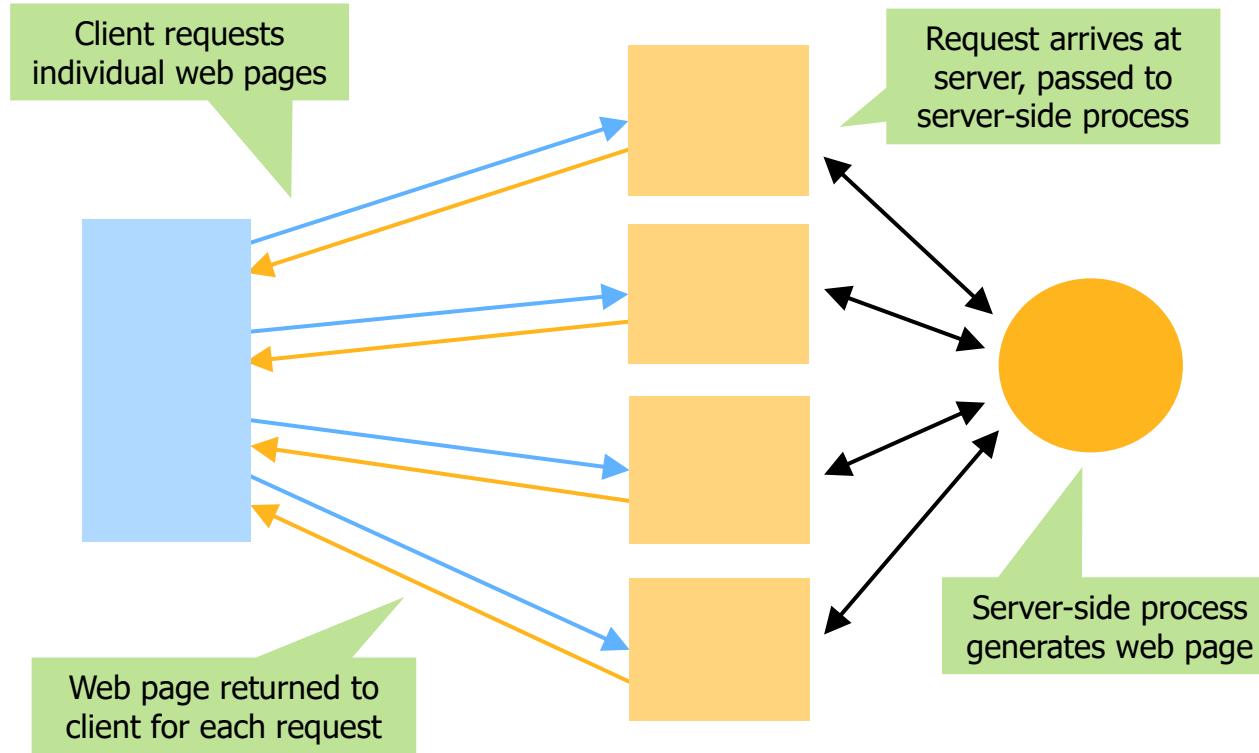
- Angular is a Single Page Application framework
 - Open-source, created by Google
- Can be used to build applications for:
 - Web
 - Mobile web
 - Native mobile
 - Native desktop



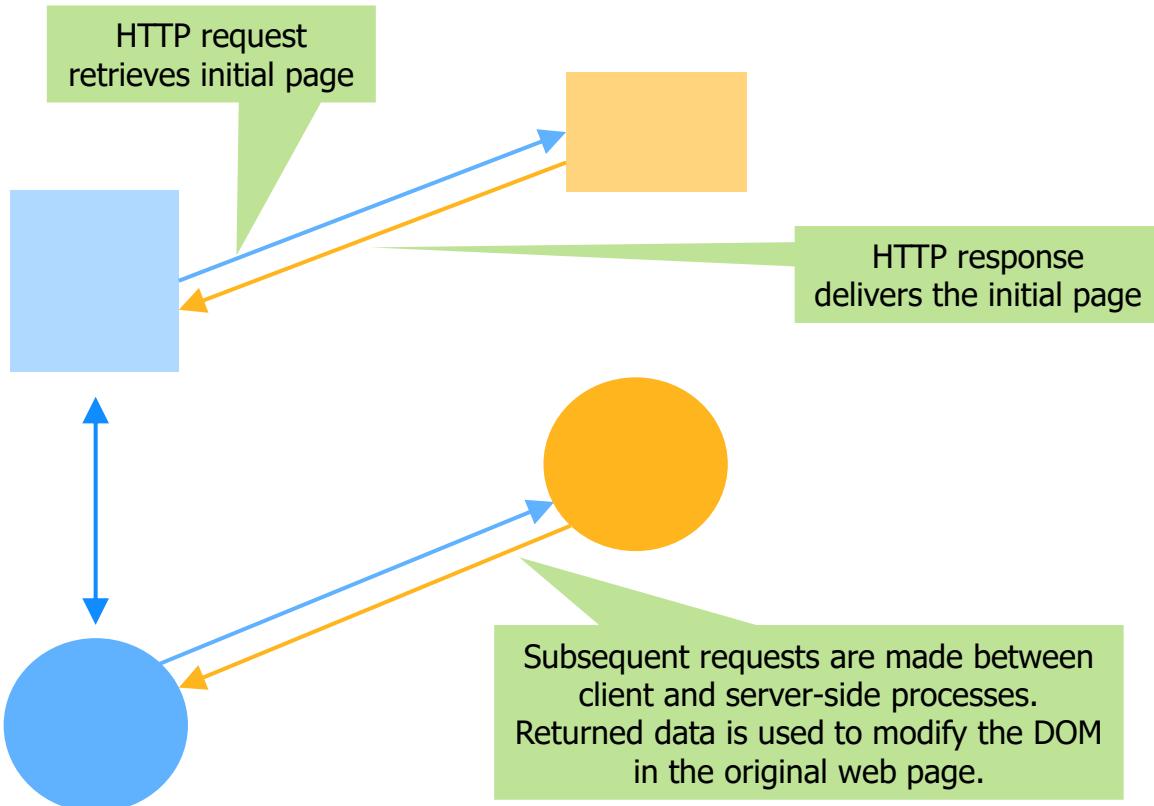
Single Page Applications (SPAs)

- Traditional web applications are comprised of multiple pages
 - Hyper Text Transfer Protocol (HTTP) is used to request web pages
 - Each request involves a complete page refresh
- SPAs use a single shell page
 - HTTP requests use Ajax to fetch JavaScript Object Notation data
 - JavaScript uses the JSON data to update the page's Document Object Model (DOM)
- Many JavaScript SPA Frameworks available:
 - Angular
 - React
 - Ember
 - Vue

Traditional Web Applications



Single Page Applications



Why Angular?

- Widely used—large ecosystem of developers and support
- Provides a complete solution
 - Modules for HTTP, animations, etc.
- Popular alternative frameworks are often more limited
 - React is a (very fast) UI rendering engine
 - Requires third-party support for HTTP, etc.
 - AngularJS was, and still is, popular and widely used
 - Easy to build simple forms
 - Hard to build significant applications
 - Many design and syntax quirks
 - Challenged on performance and scalability

Node.js

- A JavaScript runtime engine
- Angular development relies upon Node and npm
 - Must be installed on development computer
 - *npm* is Node command line package manager

The screenshot shows the official Node.js website. At the top, there's a dark header with the Node.js logo and navigation links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below the header, a main heading states: "Node.js® is an open-source, cross-platform JavaScript runtime environment." Two large download buttons are prominently displayed: one for "18.12.1 LTS" (labeled "Recommended For Most Users") and another for "19.2.0 Current" (labeled "Latest Features"). Below these buttons, smaller links for "Other Downloads", "Changelog", "API Docs", and "API Docs" are visible. A note below the download buttons says: "For information about supported releases, see the release schedule." At the bottom of the page, there's a copyright notice from the OpenJS Foundation and links to various legal documents.

Angular CLI

■ Command line tools for Angular

- Installed globally via npm
- npm install -g @angular/cli

■ Commands to generate components, routes, and services

- Even an entire application start point

The screenshot shows the official Angular CLI website. At the top, there's a navigation bar with links for ANGULAR CLI, DOCUMENTATION, GITHUB, RELEASES, and GET STARTED. Below the navigation, there's a large image of a Mac OS X terminal window displaying the following commands:

```
> npm install -g @angular/cli  
> ng new my-dream-app  
> cd my-dream-app  
> ng serve
```

To the right of the terminal image, the text "Angular CLI" is displayed, followed by "A command line interface for Angular" and a "GET STARTED" button. Below this section, there are two examples of Angular CLI commands:

ng new
The Angular CLI makes it easy to create an application that already works, right out of the box. It already follows our best practices!

ng generate
Generate components, routes, services and pipes with a simple command. The CLI will also create simple test shells for all of these.

Chapter Concepts

What Is Angular?

Introducing TypeScript

A Simple Angular Application

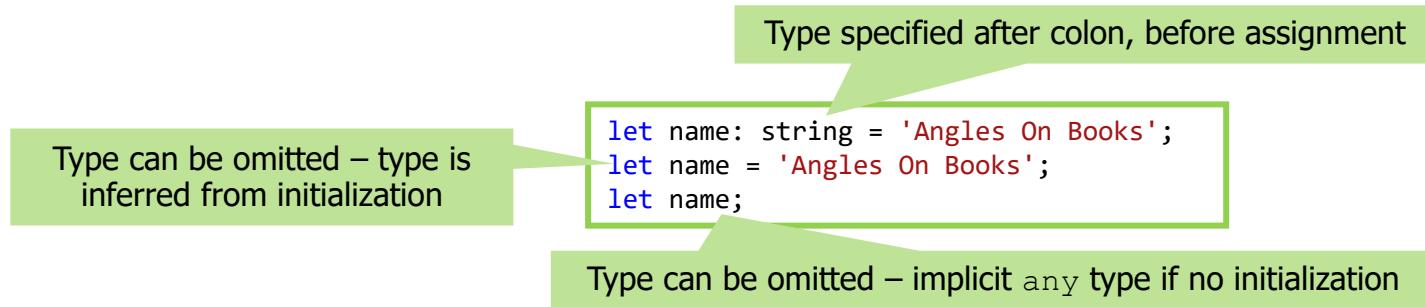
Angular Test Utilities

Chapter Summary

TypeScript

- TypeScript is a superset of JavaScript
 - Adds strong typing and other features
 - Transpiles into JavaScript
 - Configuration setting determines which version
 - Currently browsers support ECMAScript 6
- Many TypeScript features have become part of JavaScript/ECMAScript 6
 - Classes
 - Constructors
 - Modules
 - `import` and `export` statements
- This course will only cover necessary features of TypeScript
 - Typescript is recommended for Angular, but not required
 - Can also be developed using JavaScript or Dart
 - TypeScript features will mostly be introduced only as they are needed

Strong Typing



Declaration	Value
aBool: boolean	true/false
aString: string	'either' "works" Single quotes are usually considered better style
aNumber: number	Any number—integer, floating, hex, and octal
anArray: string[], number[]	Strongly typed array
anything: any	Any type
anObject: Thing	Instance of class Thing

Classes

Define objects

- let myClass: SampleClass;

export so it is available outside this file (next slide)

Parentheses indicate a method

aString is not a property of the object: it is just a parameter to the constructor

Constructors are specially named

```
sample-class.ts
export class SampleClass {
    private stringProperty: string;
    public numberProperty: number;
    public visibleMethod(): number {
        return this.numberProperty * 2;
    }
    private internalMethod(numberArgument: number) {
        this.numberProperty += numberArgument + this.anotherNumber;
    }
    constructor(aString: string, private anotherNumber: number) {
        this.stringProperty = aString;
    }
}
```

Properties are typed

private properties and methods are only available inside the class

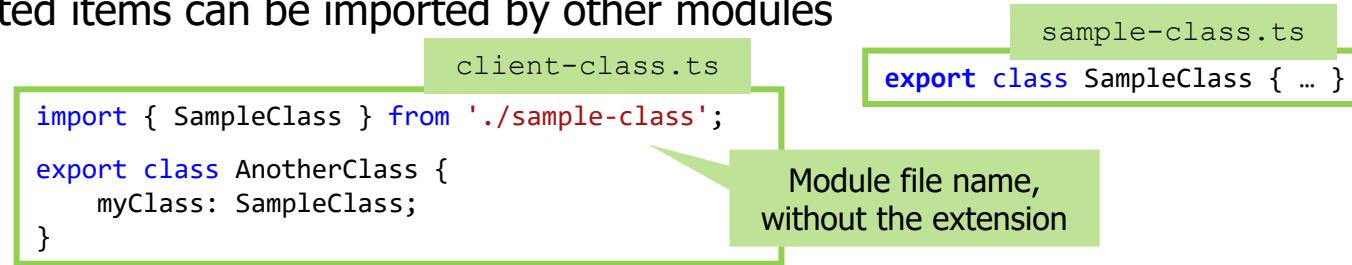
public methods are visible outside the class, methods are typed with their return type

Arguments are typed

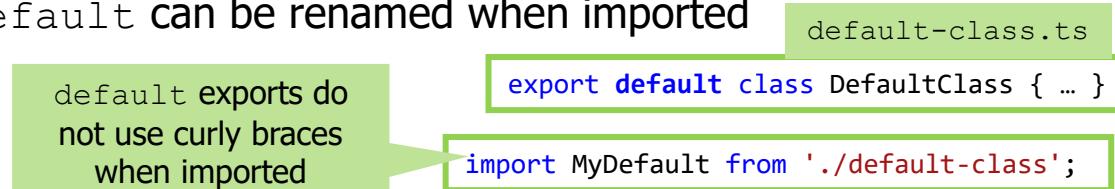
Adding a public or private modifier to a constructor argument creates a property

TypeScript Modules

- TypeScript and JavaScript define modules in the same way
- Every TypeScript file defines a module
 - Everything declared in a module is hidden unless exported
 - Exported items can be imported by other modules



- Exports with `default` can be renamed when imported



Note: For the purposes of the Exercise Manual, TypeScript module imports will be referred to as *file imports* in order to avoid confusion with Angular modules

Chapter Concepts

What Is Angular?

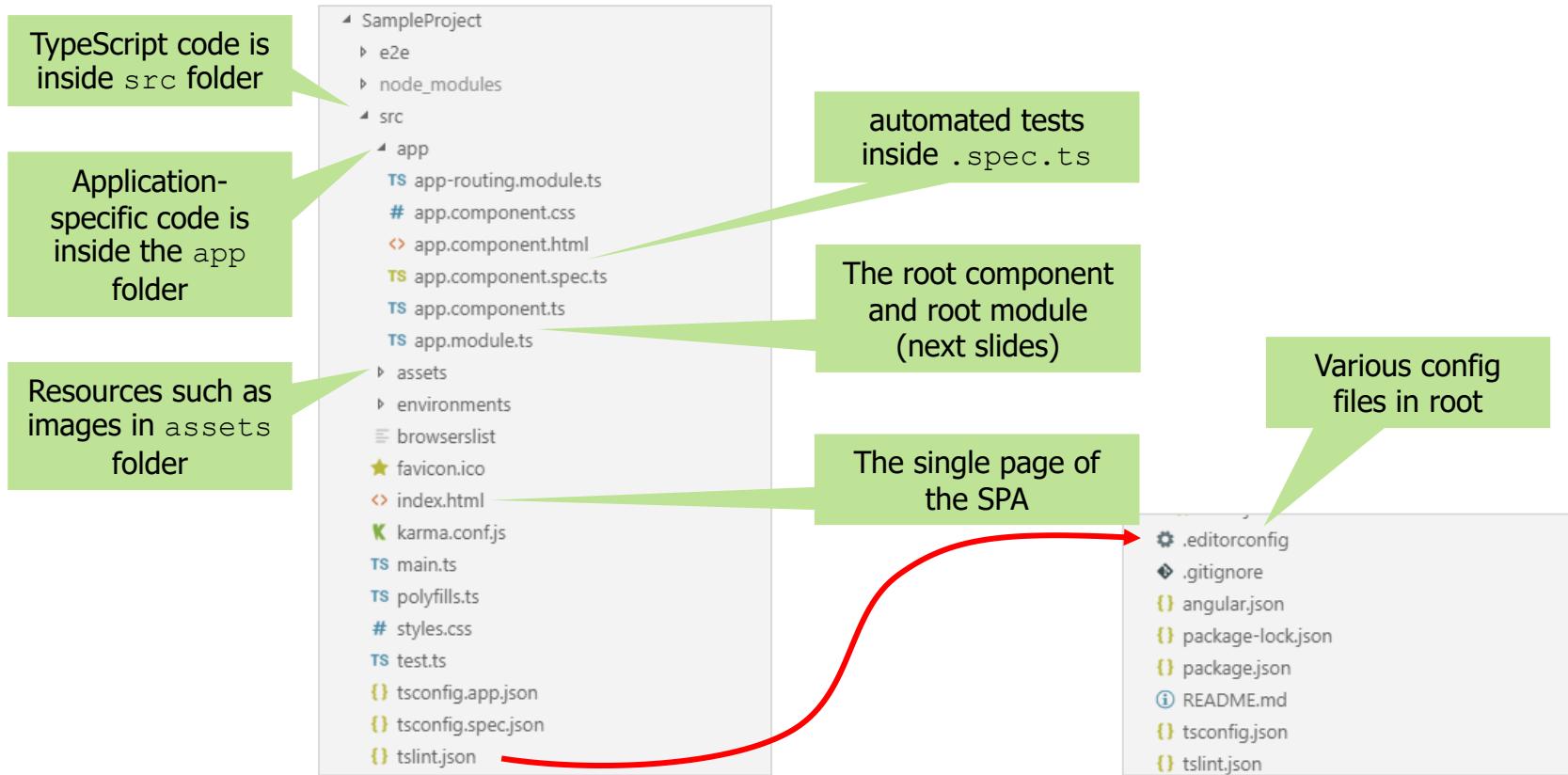
Introducing TypeScript

A Simple Angular Application

Angular Test Utilities

Chapter Summary

File Structure



The Root Module

- Every Angular application has a Root Module
 - Called `AppModule` by convention, defined in `src/app/app.module.ts`
 - Generated by Angular CLI, but you will need to maintain the imports
- All other modules are children of the root module
 - Must appear in the imports array to be available throughout the application
- Declarations are components that belong to this Module

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

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

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

`@NgModule()` means this is an Angular module

The root module contains a single component, `AppComponent`, which will be bootstrapped as the initial component

`export` means `AppModule` is also a TypeScript module

The Root Component

- Components are the building blocks of Angular application user interfaces
 - Combine controller code (properties and methods) with HTML fragments
- By convention, `AppComponent` is the root component
 - Defined inside `src/app/app.component.ts`
 - Generated by Angular CLI, but need to modify it to suit the application

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

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

The component is a TypeScript class containing properties (as shown here) and methods

The selector is used to include this component in HTML

HTML template is the visible content of this component

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

`<h1>`
Welcome to {{ title }}!
`</h1>`

Build Tools

- The individual TypeScript files do not run in the browser
- Must be converted into JavaScript
- Should also be bundled together into larger files and minified
 - Huge reduction in download time
- Requires a build tool
 - Gulp
 - Angular CLI
 - Webpack
- This course will use Angular CLI
 - Uses webpack behind the scenes
 - Code is “live compiled” in the “live server”
 - As each source code file is saved, it is automatically re-compiled and re-deployed
 - The browser application is aware of this process and re-loads the page



HANDS-ON
EXERCISE

20 min

Exercise 6.1: Getting Started with Angular

- Please complete this exercise in your Exercise Manual

Chapter Concepts

What Is Angular?

Introducing TypeScript

A Simple Angular Application

Angular Test Utilities

Chapter Summary

Jasmine

- Jasmine is a test framework
 - Originally for testing JavaScript code
 - Also used for TypeScript
- Clean syntax makes it easy to write tests
- Comprehensive set of matchers means no other libraries are required
- Contains a powerful mocking capability
 - allows sophisticated unit and integration tests to be created

Test Suite

- A test suite starts up with a call to describe
 - A global Jasmine function
 - Takes two parameters
 - A string and a callback function
- The string is a name or title for a spec suite
 - What is being tested
- The callback function is a block of JavaScript code
 - Implements the suite

```
// This is our test suite
describe('MathUtils', () => {
  let calc;
  // This will be called before running each spec
  beforeEach(() => {
    calc = new MathUtils();
  });
  // This will be called after running each spec
  afterEach(() => {
    calc = null;
  });
  // describes may be nested to group related tests together
  describe('when calc is used to perform math operations', () => {
    // Specs for sum and subtract operations
    it('should calculate sum of 3 and 5', () => {
      expect(calc.sum(3, 5)).toEqual(8);
    });
    it('should calculate difference of 17 and 6', () => {
      expect(calc.subtract(17, 6)).toEqual(11);
    });
    ...
  });
});
```

Specs

- Specs are defined by calling another global Jasmine function
 - The function `it` also takes a string and a callback function
- The string is the title of the spec
- The callback function is the spec
 - That is, the test itself
- A spec contains one or more *expectations*
 - Test the state of the code
- An expectation is an assertion
 - Is either true or false
- A spec with all true expectations passes
 - Otherwise, it fails

```
describe('A suite', () => {
  it('contains spec with an expectation', () => {
    let expected = ...;
    let actual = testObject.methodBeingTested();
    expect(actual).toBe(expected);
  });
});
```

Expectations

- Expectations are built with the `expect` function
 - Which takes a value
 - Called the actual
- Chained with a `matcher` function
 - Which contains the expected value
- Each matcher compares the actual and expected values
 - To negate a matcher, chain `expect` to `not` before the matcher
- Jasmine has many matchers included
 - Custom matchers can also be defined

```
it('exercises Jasmine matchers', () => {  
  let pi = 3.1415926;  
  let e = 2.78;  
  
  expect(e).toBe(2.78);  
  expect(e).toEqual(2.78);  
  
  let d1 = new Date(1970, 1, 1);  
  let d2 = new Date(1970, 1, 1);  
  
  expect(d1).not.toBe(d2);  
  
  expect(d1).toEqual(d2);  
  
  expect(e).toBeLessThan(pi);  
  expect(pi).not.toBeLessThan(e);  
  expect(pi).toBeGreaterThan(e);  
});
```

For numbers and strings, these are equivalent

`toBe()` compares object identities

`toEqual()` compares object property values

Expecting an Error

- How do you write an `expect()` for a function that throws an Error?
 - Pass a callback function that calls the function that throws the Error

```
class MathUtils {  
    factorial(num: number) {  
        if (num == 0 || num == 1) {  
            return 1;  
        } else if (num > 1) {  
            return num * this.factorial(num - 1);  
        } else {  
            throw new Error(`factorial is not defined for negative number ${num}`);  
        }  
    }  
}
```

```
let calc = new MathUtils();  
it('should return 1 when the number is 0', () => {  
    expect(calc.factorial(0)).toBe(1);  
});  
  
it('should throw an error when the number is negative', () => {  
    expect(() => {  
        calc.factorial(-7);  
    }).toThrowError(Error);  
});
```

Test Fixtures

- Jasmine provides functions for setting up and tearing down code

- beforeEach
- afterEach
- beforeAll
- afterAll

```
describe('A spec using beforeEach and afterEach', () => {
  const theAnswer = 42;
  let foo;

  beforeEach(() => {
    foo = theAnswer;
  });

  afterEach(() => {
    foo = 0;
  });

  it('does foo equal theAnswer', () => {
    expect(foo).toEqual(theAnswer);
  });
});
```

Disabling a Test Suite

- A test suite can be disabled with the `xdescribe` function
 - The suite is skipped; specs inside it are not run
 - Results show as pending
- You can focus on one suite with `fdescribe`
 - Only the specs in that suite will run
 - Useful for debugging one suite

```
xdescribe('Neither of these specs will run', () => {  
  it('...', () => { ... });  
  it('...', () => { ... });  
});  
  
describe('But both these specs will run', () => {  
  it('...', () => { ... });  
  it('...', () => { ... });  
});
```

```
fdescribe('Only these specs will run', () => {  
  it('...', () => { ... });  
  it('...', () => { ... });  
});  
  
describe('These specs will not run', () => {  
  it('...', () => { ... });  
  it('...', () => { ... });  
});
```

Disabling a Spec

- A spec can be disabled with the `xit` function
 - That one spec is skipped
 - Results show as pending
- You can focus on one spec with `fit`
 - Only that spec will run
 - Useful for debugging one spec

```
describe('The last spec will not run', () => {
  it('...', () => { ... });
  it('...', () => { ... });
  xit('...', () => { ... });
});

describe('These specs will run as usual', () => {
  it('...', () => { ... });
  it('...', () => { ... });
});

describe('Only the last spec will run', () => {
  it('...', () => { ... });
  it('...', () => { ... });
  fit('...', () => { ... });
});

describe('These specs will not run', () => {
  it('...', () => { ... });
  it('...', () => { ... });
});
```

Jasmine Matchers

- So far, we have seen `toEqual`,
`toThrow`, `toBeLessThan`
- See a few more in the list

<code>toBe()</code>	passed if the actual value is of the same type and value as that of the expected value. It compares with <code>==</code> operator
<code>toEqual()</code>	works for simple literals and variables; should work for objects too
<code>toMatch()</code>	to check whether a value matches a string or a regular expression
<code>toBeDefined()</code>	to ensure that a property or a value is defined
<code>toBeUndefined()</code>	to ensure that a property or a value is undefined
<code>toBeNull()</code>	to ensure that a property or a value is null.
<code>toBeTruthy()</code>	to ensure that a property or a value is <code>true</code>
<code>toBeFalsy()</code>	to ensure that a property or a value is <code>false</code>
<code>toContain()</code>	to check whether a string or array contains a substring or an item.
<code>toBeLessThan()</code>	for mathematical comparisons of less than
<code>toBeGreaterThan()</code>	for mathematical comparisons of greater than
<code>toBeCloseTo()</code>	for precision math comparison
<code>toThrow()</code>	for testing if a function throws an exception
<code>toThrowError()</code>	for testing a <i>specific</i> thrown exception



Exercise 6.2: Write Your First Test Specs

20 min

- Please complete this exercise in your Exercise Manual

Jasmine Docs

- Jasmine has many other features
 - We will meet more of them as we need them
- For more details, documentation, and examples, visit the Jasmine website
 - <https://jasmine.github.io/>

Types of Angular Test Utilities

- There are a number of different types of utilities required to test Angular
 - In each case, there are many options available
- A test runner (Angular default → Karma)
 - Run the tests and display results (Karma displays them in a browser)
- A test framework (Angular default → Jasmine)
 - Provide a way of specifying tests
 - Jasmine is also popular for JavaScript
- A matcher library (Angular default → Jasmine)
 - Methods to compare data and indicate whether the test has passed or failed
 - Not every test framework includes matchers
- A browser simulator
 - An environment to imitate the browser, gives more control over tests
 - Angular provides its own for unit testing; later, we will use Cypress for E2E tests

Karma Test Runner

- Karma integrates with Node
- Simply call `npm test` on the command line
 - Opens Chrome and runs tests
- If a spec fails, the stack trace will pinpoint the source of the error

The screenshot shows the Karma Test Runner interface. At the top, it says "Karma v 6.4.1 - connected; test: complete;". Below that, it says "Chrome 108.0.0.0 (Windows 10) is idle". The main area shows a test result for "AppComponent": "3 specs, 0 failures, randomized with seed 78893 finished in 0.049s". The test description is "AppComponent > should render title in a h1 tag". The error message is "Expected 'Welcome to Angles on Books!' to contain 'Welcome to Beyond Books!'.". The stack trace is as follows:

```
Expected 'Welcome to Angles on Books!' to contain 'Welcome to Beyond Books!'.  
at <Jasmine>  
at UserContext.apply (<http://localhost:9876/_karma_webpack_/webpack:/src/app/app.component.spec.ts:30:17>)  
at _ZoneDelegate.invoke (<http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:375:26>)  
at ProxyZoneSpec.onInvoke (<http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:287:39>)  
at _ZoneDelegate.invoke (<http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:374:52>)
```

Using Jasmine to Test Angular

- We can use Jasmine to test all parts of an Angular application
 - Except Modules: these are groupings of components rather than functional elements
- Recall that an Angular application starts with the Root Module (`AppModule`)
 - Imports all the dependencies
 - Declares all the available components
 - Bootstraps the application through the `AppComponent`
- To test a component (actually any Angular directive), we need something that behaves a lot like the `AppModule`, but:
 - Creates a simulated application to host the component
 - To allow us to isolate the unit under test
 - Gives us more control over how the application starts and runs
 - Allows us to create test fixtures and determine when data binding occurs
 - Interacts with a simulated browser rather than requiring a full browser

TestBed

- Creates an Angular testing module
 - Replaces the component's module (in this case, `AppModule`) for testing purposes
- Is configured via an object literal argument to `configureTestingModule()`
- `async` and `await` serialize asynchronous operations
 - Required here so `compileComponents()` is finished before specs are executed

```
describe('BookListComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [
        BookListComponent,
        BookFormComponent
      ],
      imports: [
        ReactiveFormsModule
      ]
    }).compileComponents();
  });
});
```

TestBed configuration inside `beforeEach()`

We'll discuss `await` and `async` in detail later

book-list.component.spec.ts

Declare components used by the component under test

Like `@NgModule()` uses `imports` for module dependencies

ComponentFixture<T>

- Represents the test environment surrounding the component
- Gives access to the component itself
 - Via the `componentInstance` property
- Also provides a handle for the associated DOM element
 - Via the `debugElement` property
- `debugElement` provides query methods to locate specific DOM element(s)

```
const fixture: ComponentFixture<AppComponent> = TestBed.createComponent(AppComponent);
fixture.detectChanges(); // detectChanges() performs data binding

// Look up the text content of the <h1> tag
const h1Content: string = fixture.debugElement // Angular DebugElement
  .nativeElement // AppComponent's HTMLElement
  .querySelector('h1') // <h1> tag's HTMLElement
  .textContent; // content of <h1> tag

expect(h1Content).toContain('Welcome to Angles on Books!');
```

A Sample Test

```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';

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

  it('should render title in a h1 tag', () => {
    const fixture = TestBed.createComponent(AppComponent);

    fixture.detectChanges();

    const domElement = fixture.debugElement.nativeElement;
    expect(domElement.querySelector('h1').textContent).toContain('Welcome to Angles on Books!');
  });
});
```

describe() specifies what is being tested

beforeEach() sets up the TestBed

it() defines an individual spec

detectChanges() triggers data binding

expect() determines success or failure



Exercise 6.3: Unit Testing Angular

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

What Is Angular?

Introducing TypeScript

A Simple Angular Application

Angular Test Utilities

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- What is Angular?
- TypeScript
- A 'Hello World' Angular application
- Automated testing with Jasmine and Karma
- Writing tests using Angular-specific utilities

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 7: Angular Components

Chapter Overview

In this chapter, we will explore:

- Creating an Angular component
- Defining strongly typed classes
- Passing data between a parent and child component

Chapter Concepts

Creating Angular Components

Built-In Directives

An Angular Application

Chapter Summary

Components

- Components are the building blocks of Angular applications
 - Angular applications are a tree of components
- At a minimum, components must have:
 - A *template*
 - Defines UI content
 - Uses data binding to display class properties
 - Can be internal or housed in external file
 - A *selector*
 - String defining HTML-like element to identify component
 - So can be included in another component's template
- Defined by adding `@Component()` decorator to a class

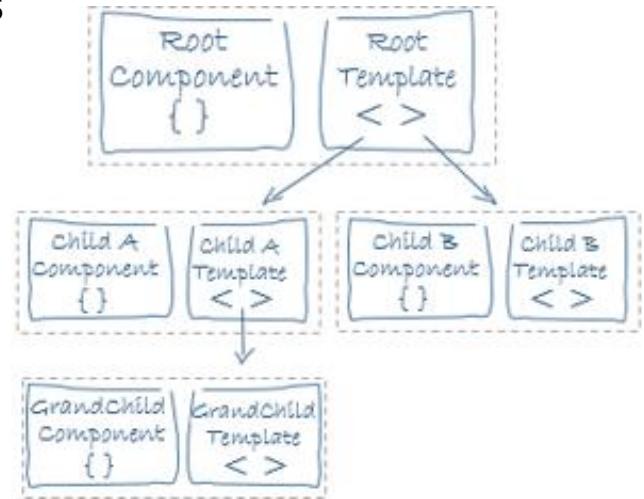


Image source: <https://angular.io/guide/architecture>

Decorators

- Decorators add meta-information to classes, methods, parameters, etc.
 - Similar to attributes in C#, annotations in Python and Java
- Have the form `@expression()`
 - `@expression()` evaluates to a function that can be called at runtime
 - Can be used to modify/replace decorated item
 - E.g., to convert a standard class into an Angular component
 - Exact behavior determined by arguments passed to `@expression()`
- Decorators are currently proposed as a new JavaScript standard
 - Already available in TypeScript
- Used extensively in Angular
 - `@NgModule()`
 - `@Component()`
 - `@Input()`
 - More



The @Component() Decorator

- Transforms a standard class into an Angular component
- Behavior determined by object literal passed to the decorator

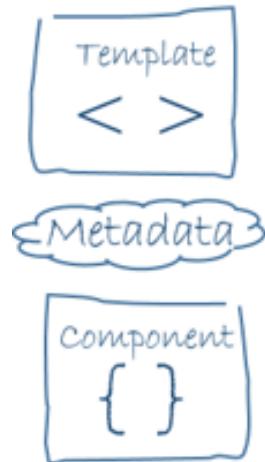
```
Make @Component() available
```

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

@Component({
  selector: 'app-book-list',
  template: `<h2>
    This is the Book List Component
  </h2>`
})
export class BookListComponent {
  // Component implementation code here
}
```

The selector `app-book-list` becomes
`<app-book-list></app-book-list>`
in parent component

Use of ` (back tick) allows
string to span multiple lines



- Normally, we replace the "app-" prefix (for example, book-list, book-form)
 - When doing that, we should change the ESLint configuration to match

Internal vs. External Templates

- Templates can be internal
 - Use template property with string value

```
@Component({  
  selector: 'app-root',  
  template: `<h1>Hello {{ title }}!</h1>  
    <app-book-list></app-book-list>`  
})
```

- Or external
 - Use templateUrl to point at a separate file

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html'  
})
```



```
<h1>  
  Hello {{ title }}!  
</h1>  
  
<app-book-list></app-book-list>
```

- Some of the example code will use internal templates for convenience
 - In the real world, prefer external templates for longer templates
 - Angular style guide recommends external for > 3 lines

Creating a Custom Component

Steps to creating a component:

1. Write the class (shown previously)
2. Declare the component in an Angular module
 - All components must be declared by exactly 1 module
3. Add a file import for the component
4. Use the selector inside the parent template

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { BookListComponent } from './book-list/book-list.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    BookListComponent
  ]
})
export class BooksModule { }
```

3: import

2: declare

```
@Component({
  selector: 'app-root',
  template: './app.component.html'
})
export class AppComponent {
  ...
}
```

app.component.ts

app.component.html

```
<h1>Hello {{ title }}!</h1>
<app-book-list></app-book-list>
```

Creating a Component Using Angular CLI

- Angular CLI provides methods to create most Angular types, including components
 - With reasonable defaults (e.g., internal structure, updates module appropriately)
 - Automatically converts between kebab case for HTML and title case for TypeScript
 - So these commands are equivalent:
 - `ng generate component book-list`
 - `ng generate component BookList`
 - The keyword can be abbreviated:
 - `ng g component BookList`
 - There are many options:
 - Some are common to all generated items
 - E.g., `-d` is a “dry run”: says what changes would be made, but doesn’t make them
 - Some are unique to a particular type of item
 - (Don’t worry about them for now, but good to know they’re there)

Data Binding

- A core advantage of SPAs is the abstraction of the DOM
 - Developers bind data to templates rather than interacting directly with the DOM
 - As data changes in code, the web page automatically updates
- In Angular, a component is TypeScript code
 - Represents the controller of a (small part of a) user interface
 - Data items and methods in the component are bound to elements in the HTML template
- Angular supports many types of data binding
 - We have already seen interpolation binding
 - It uses the moustache notation

```
 {{book.title}} by {{book.author}}
```

Creating Custom Data Types

- Most of the time, we will be passing complex data
 - Create a `class` to define a custom data type
- Declare constructor arguments as `public` to create properties
 - Private data is visible only inside the class

```
export class Book {  
    constructor (  
        public title: string,  
        public author: string  
    ) {}  
}
```

```
"use strict";  
exports.__esModule = true;  
var Book = /** @class */ (function () {  
    function Book(title, author) {  
        this.title = title;  
        this.author = author;  
    }  
    return Book;  
}());  
exports.Book = Book;
```

JavaScript object generated from
TypeScript class above

- You can use `ng generate class` to do this
 - Limited benefit since there are no complex dependencies or structure

Binding to Custom Data Type

Steps to bind to custom data:

1. Write the class (previous slide)
2. Import the class into the Angular Component
3. Add a property with the custom data type to the component class
4. Use interpolation binding syntax with dot notation inside the template

```
<tr>
  <td>{{book.title}}</td>
  <td>{{book.author}}</td>
</tr>
```

Name must match

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../models/book';

@Component({
  selector: 'app-book-list',
  templateUrl: './book-list.component.html',
  styleUrls: ['./book-list.component.css']
})
export class BookListComponent implements OnInit {

  book: Book = {
    title: 'The Lord of the Rings',
    author: 'J R R Tolkien',
    cover: '',
    bookId: 1
  };

  constructor() { }

  ngOnInit() {
  }
}
```

2: import

3: property

Naming Guide

- Angular CLI imposes certain Angular best practices for naming

- Define each component in its own file
 - Include *component* in filename
 - E.g., `book-list.component.ts`
 - Same pattern with all Angular types
 - `feature.type.ts`
 - `some-feature.component.ts`
 - `some-feature.module.ts`
 - `some-feature.pipe.ts`
 - Use hyphens for selector and filenames
 - E.g., `book-list`
 - Use title case for class name, including type as part of name
 - E.g., class `BookListComponent`

- Angular understands how and when to convert between title case and “kebab case”



HANDS-ON
EXERCISE

30 min

Exercise 7.1: Creating a Component

- Please complete this exercise in your Exercise Manual

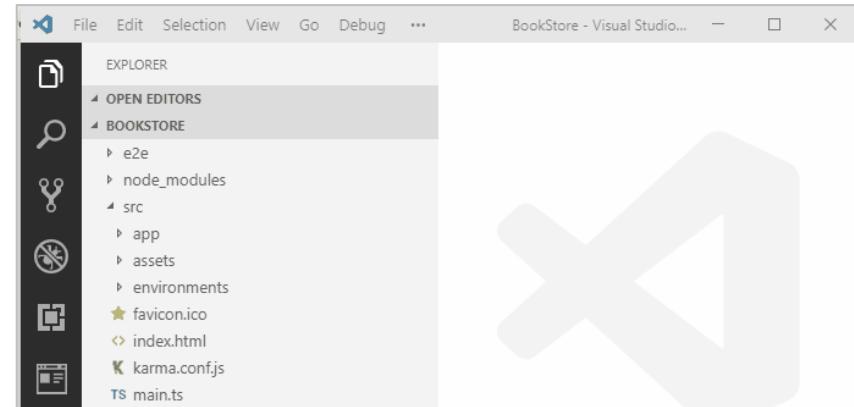
Debugging Angular Applications in VS Code

■ VS Code can debug code running in Chrome

■ Click VS Code's **Debug** icon

- A launch configuration is defined in `.vscode/launch.json`
- Click the gear icon on the debug toolbar to view the config

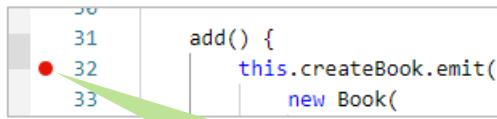
```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "name": "Launch Chrome against localhost",  
      "type": "chrome",  
      "request": "launch",  
      "url": "http://localhost:4200",  
      "webRoot": "${workspaceFolder}",  
      ...  
    }  
  ]}
```



`${workspaceFolder}` is the folder open in VS Code.
`webRoot` must be the folder that contains `package.json`

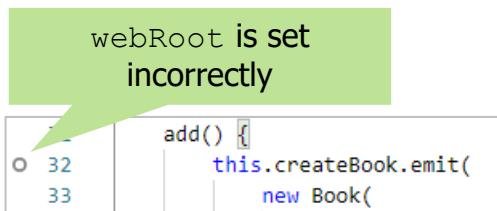
Debugging Angular Applications in VS Code (continued)

- Set breakpoints in the gutter to the left of source code
- Run the application as usual
 - Using `npm start`
- In the debug panel, click the **Play** button by the configuration name (Angular app)
 - A new window should open connected to the application



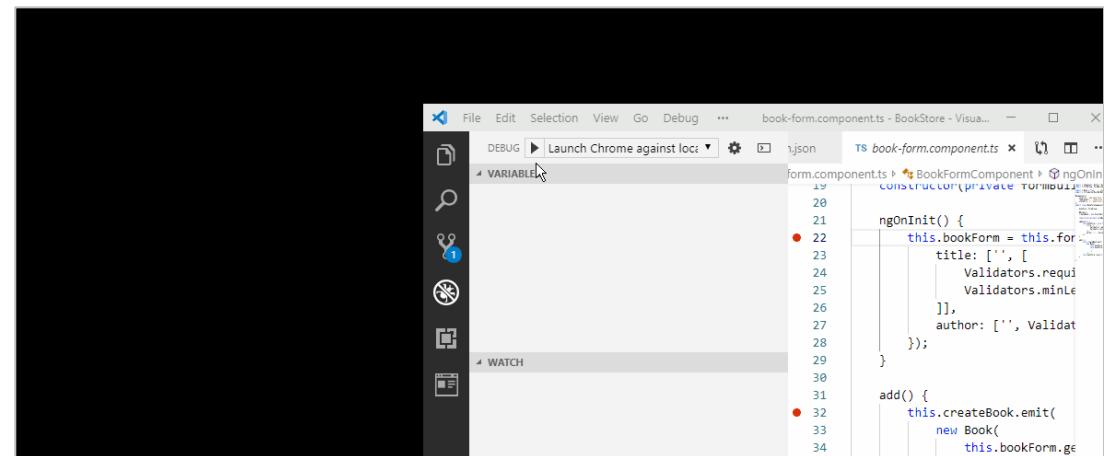
```
31
32 add() {
33     this.createBook.emit(
            new Book()
```

Click here to set and
unset breakpoints



```
31
32 add() {
33     this.createBook.emit(
            new Book()
```

webRoot is set
incorrectly



The screenshot shows the VS Code interface during debugging. The top bar has "DEBUG" selected. The left sidebar has icons for file, search, and variables. The main area shows the code for `book-form.component.ts`. Two red dots indicate breakpoints at lines 32 and 33. The code is as follows:

```
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
```

The "VARIABLES" pane shows the state of variables, and the "WATCH" pane shows monitored expressions.

Debugging Angular in Chrome

■ Get access to a component

- Highlight an Angular component in Developer Tools
- In the console, type `ng.getComponent($0)`
 - Returns the `DebugElement` that corresponds to that component
 - View a method implementation: `inspect(ng.getComponent($0).ngOnInit)`
 - View property values by expanding the component

```
> ng.getComponent($0)
<  ▼ BookPageComponent {bookService: BookService, books: Array(4), errorMessage: '', __ngContext__: LRootView(31)} ⓘ
  ► bookService: BookService {http: HttpClient, url: 'http://localhost:8080/BookService/jaxrs/books'}
  ▼ books: Array(4)
    ► 0: {author: 'Gamma, Helm, Johnson, Vlissides', bookId: 4, cover: '', title: 'Design Patterns'}
    ► 1: {author: 'Martin Fowler', bookId: 3, cover: '', title: 'UML Distilled'}
    ► 2: {author: 'Robert Martin', bookId: 2, cover: '', title: 'Clean Code'}
    ► 3: {author: 'Neal Stephenson', bookId: 1, cover: '', title: 'Cryptonomicon'}
    length: 4
    ► [[Prototype]]: Array(0)
  errorMessage: ''
```

■ Use Chrome's Angular DevTools extension to view component tree and profile execution

Try It Now: Debugging Angular

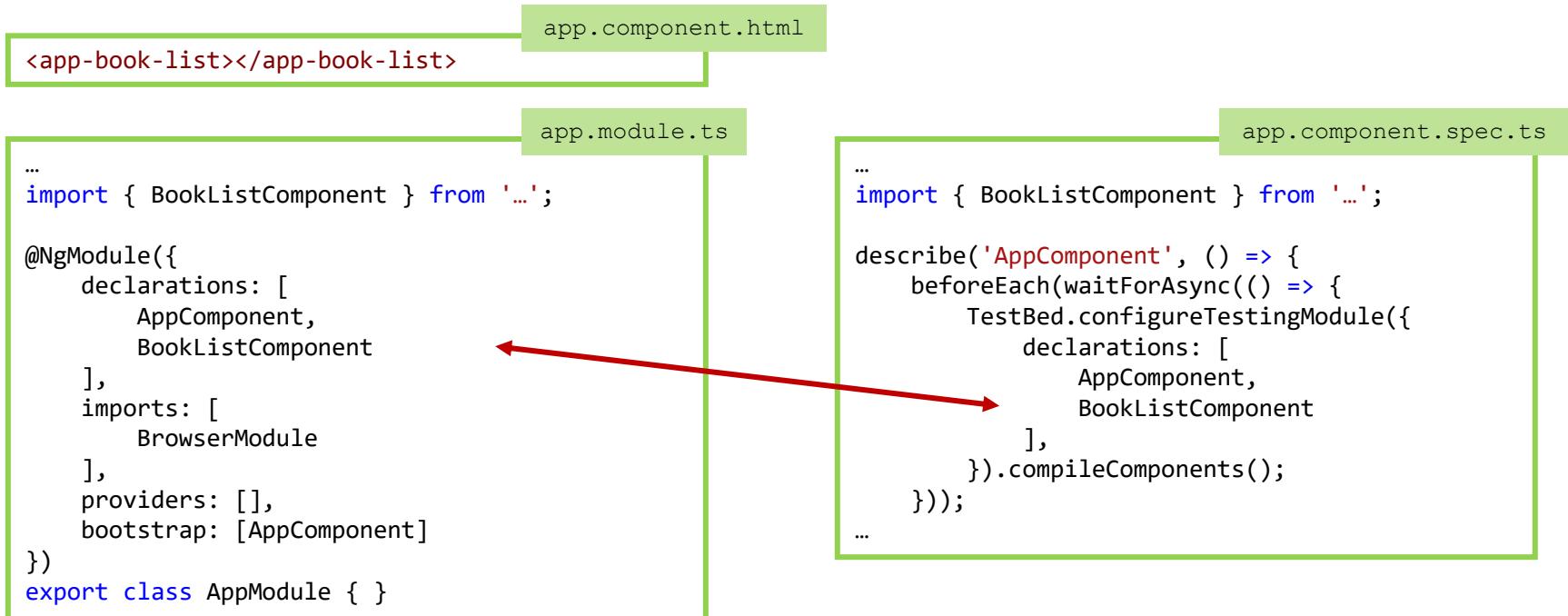


10 min

- Try out the techniques from the previous slides

Unit Testing Components

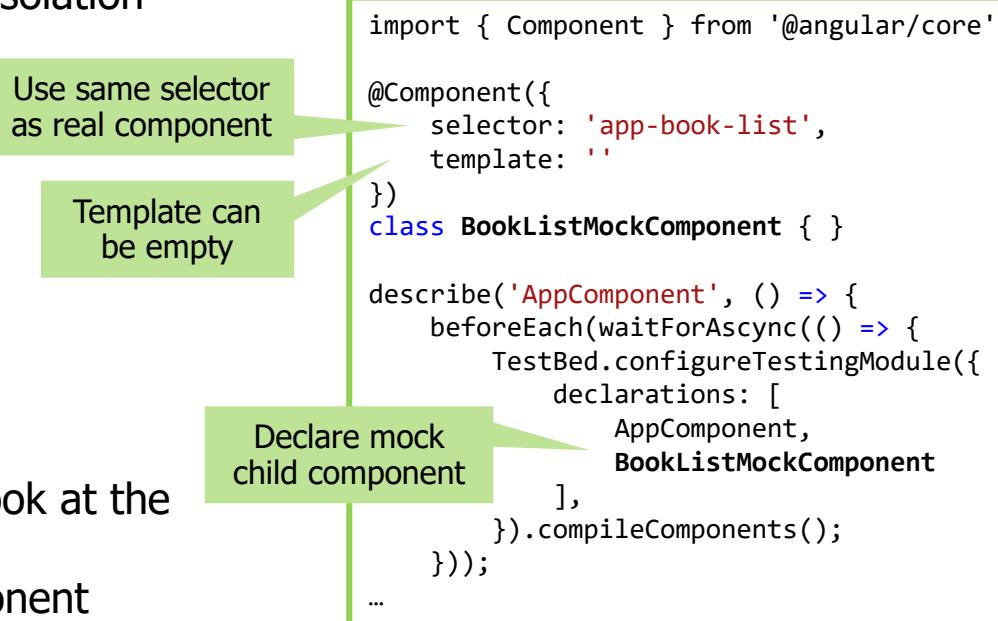
- Whenever we add one component to the template of another, we create a dependency
 - Tests will not compile unless we add the same dependency to the test module



Isolating Components for Testing

- However, if we just add the component to the test, then the component is not being tested in isolation
 - Not really a unit test
- Instead, create a mock component
 - Give it the same selector
 - In many cases, an empty class will suffice
- Future changes to the dependency will not affect the test
- To know which mocks are required, look at the component's HTML template
 - Each `<app-*>` tag is a child component
 - Each child component needs a mock

```
import { Component } from '@angular/core'  
  
@Component({  
  selector: 'app-book-list',  
  template: ''  
})  
class BookListMockComponent {}  
  
describe('AppComponent', () => {  
  beforeEach(waitForAsync(() => {  
    TestBed.configureTestingModule({  
      declarations: [  
        AppComponent,  
        BookListMockComponent  
      ],  
    }).compileComponents();  
  }));  
  ...  
});
```



What Tests Are Needed for a Component?

- Tests that depend heavily on the layout of the component's template will be fragile
 - Any change to the layout will require changes to the tests
- What can we test without involving the template?
 - Interaction between methods and properties
 - Input and output bindings (coming up shortly)
 - The API of the class
- However, the component consists of a component class *and* a template
 - Goal: test interaction of component's class and HTML template
 - To see that the data appear correctly
 - To see that controls trigger the right methods
 - No need to test static layout: concentrate on the dynamic content

Testing the Component Template

- After any component data changes, need to ensure the Angular event loop has run
 - `fixture.detectChanges();`
- Use CSS selectors to identify the appropriate element and test properties
 - Use `debugElement.nativeElement` with `querySelector()`
 - Then work with the native DOM methods
 - Or `debugElement` with `query()`
 - Returns another `debugElement`, eventually access `nativeElement`
 - Both of these techniques resemble the way jQuery accesses elements
- This code snippet checks properties of an HTML `<table>`:

Get the DOM element corresponding to the component

Find the first `table` element in the component

```
const pageComponentDomElement = fixture.debugElement.nativeElement;
const table = pageComponentDomElement.querySelector('table');
console.log(table);
expect(table.rows.length).toBe(1);
expect(table.rows[0].cells[0].textContent)
  .toBe('The Lord of the Rings');
```

Check properties of the DOM element

Debugging Angular Unit Tests in VS Code

- launch.json has a launch configuration for debugging Karma unit tests
- Set breakpoints in either application code or test code
 - Be sure the keyboard focus is in the spec file
 - Run npm test, leave the Karma window open
 - In the Debug view, select **Karma unit tests**
 - Click the **Play** button
 - Specs will run again and execution will pause at the next breakpoint

```
19    it(`should have as title 'Angles on Books'`, () => {
20      const fixture = TestBed.createComponent(AppComponent);
21      const app = fixture.debugElement.componentInstance;
22      expect(app.title).toEqual('Angles on Books');
23    });

```

```
{
  "name": "Karma unit test",
  "type": "chrome",
  "request": "launch",
  "url": "http://localhost:9876/debug.html",
  "sourceMapPathOverrides": { ... },
  ...
}
```

webRoot is set incorrectly or the sourceMaps are defined incorrectly

31	add() {
32	this.createBook.emit(
33	new Book(



HANDS-ON
EXERCISE

20 min

Exercise 7.2: Unit Testing a Component

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Creating Angular Components

Built-In Directives

An Angular Application

Chapter Summary

Built-In Directives

- Angular provides several directives to help build templates
 - Both attribute and structural directives
- Structural directives modify the template by adding or removing elements
 - NgIf
 - NgSwitch
 - NgFor
- Attribute directives add behavior to existing template structure
 - NgClass
 - NgStyle

The NgIf Directive

- Adds and removes elements from the DOM
 - Including any child elements

Structural directives begin with *

Use quotes, not braces

```
<tr *ngIf="book.isABestSeller">
  <td colspan="2">One of our best sellers</td>
</tr>
<tr *ngIf="!book.isABestSeller">
  <td colspan="2">A cult classic </td>
</tr>
```

Note logic reversing value

The screenshot shows a browser's developer tools with the 'Elements' tab selected. A table is displayed with two rows. The first row has two cells and contains the text 'One of our best sellers'. The second row is collapsed, showing its template binding information in the developer tools interface. The binding information includes the element type, its attributes, and the condition being evaluated.

```
Elements Console Sources Network Timeline >| ...
}-->
▼<tr _ngcontent-wgk-1>
  ▶<td _ngcontent-wgk-1 colspan="2">...</td> == $0
</tr>
<!--template bindings= {
  "ng-reflect-ng-if": null
}-->
</tbody>
```

Only element matching *ngIf included in the DOM

The NgFor Directive

- Used for repeating data
- Creates an instance of a template for each item in an array

```
<tr *ngFor="let country of countries">
  <td>{{country.name}}</td>
  <td>{{country.language}}</td>
</tr>
```

country is a template input variable, only available inside the loop

```
countries: Country[] = [
  {
    id: 1,
    name: 'United Kingdom',
    language: 'English'
  },
  {
    id: 2,
    name: 'United States of America',
    language: 'English'
  }];
]
```

Countries

Country	Language
United Kingdom	English
United States of America	English

NgFor Exported Values

- NgFor provides a set of exported values

- index (number)
- first (boolean)
- last (boolean)
- even (boolean)
- odd (boolean)

Declare local variable and set equal to required exported value

- Automatically populated

- Assign to local variable

```
<tr *ngFor="let country of countries; let i = index">
  <td>{{country.name}} index = {{i}}</td>
  <td>{{country.language}}</td>
</tr>
```

Value of **i** is current index

United Kingdom index = 0	English
United States of America index = 1	English

The NgClass Directive

- Allows CSS classes to be assigned dynamically
 - Argument to [ngClass] can be an object literal

book-list.component.html

Object's keys are CSS class names

```
<tr *ngFor="let book of books">
  <td [ngClass]="{ bestSeller: book.isABestSeller }">{{book.title}}</td>
```

book-list.component.css

```
.bestSeller {
  color: green;
  font-weight: bold;
}
```

book-list.component.ts

```
book: Book = {
  isABestSeller: true,
  title: 'The Lord Of The Rings',
  author: 'J R R Tolkien'
};
```

If book.isABestSeller property is true, add bestSeller class to <td>

book-list component rendered output:

Title	Author
The Lord Of The Rings	J R R Tolkien

Browser DevTools Elements tab:

- <tbody _ngcontent-fyf-1>
- <tr _ngcontent-fyf-1>
- <td _ngcontent-fyf-1 ng-reflect-ng-class="[object Object]" class="bestSeller">The Lord Of The Rings</td>
- <td _ngcontent-fyf-1>J R R Tolkien</td>

The NgStyle Directive

- Allows inline CSS values to be assigned dynamically

```
<td [ngStyle]="calculatedStyles">{{book.title}}</td>
```

```
...
```

```
calculatedStyles = {  
  color: this.book.isABestSeller ? 'green' : 'black',  
  fontWeight: this.book.isABestSeller ? 'bold' : 'normal'  
}
```

Inline CSS assigned based on object literal
created dynamically inside component

The screenshot shows a browser's developer tools with the 'Elements' tab selected. A table row is selected in the DOM tree. The 'ngStyle' attribute of the first `td` element is highlighted with a red oval, and its corresponding value in the 'style' attribute is also highlighted with a red oval. The value is `color: green; font-weight: bold;`. The table has two columns: 'Title' and 'Author'. The row contains the data: 'The Lord Of The Rings' and 'J R R Tolkien'.



Exercise 7.3: Using Built-In Directives

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Creating Angular Components

Built-In Directives

An Angular Application

Chapter Summary

Angular Architecture Overview

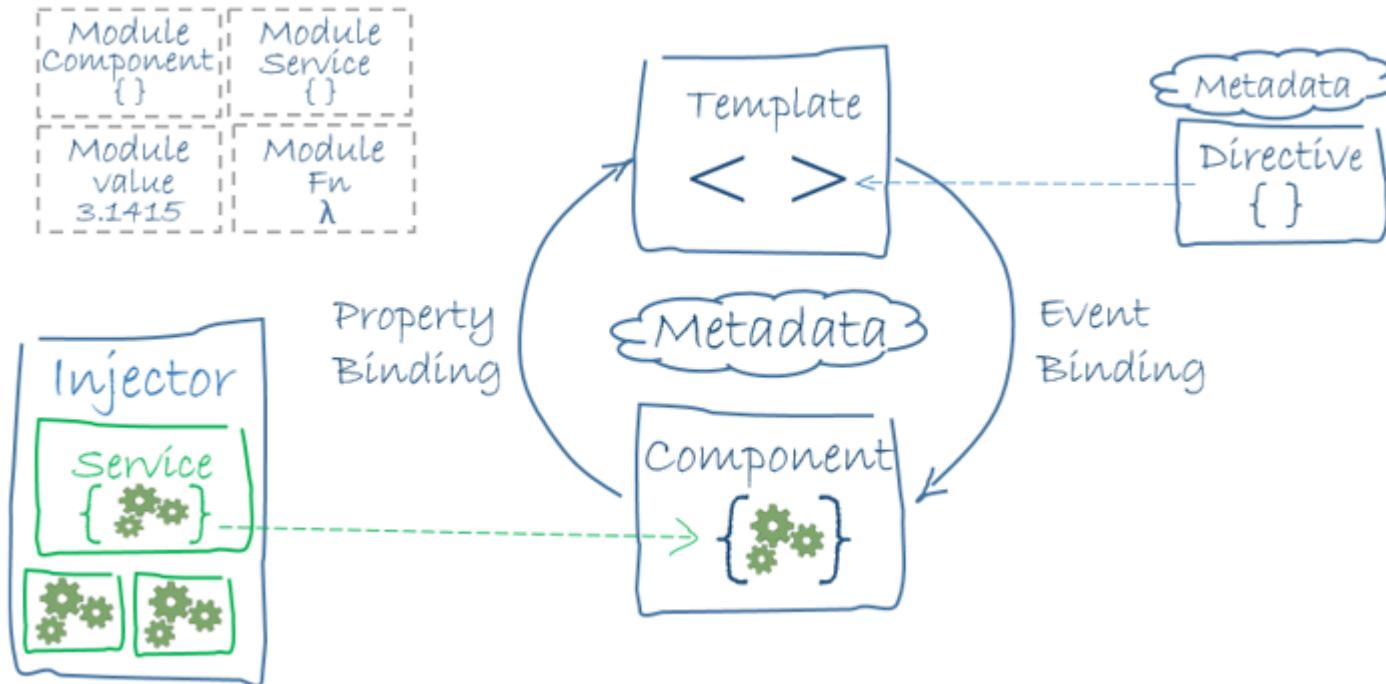


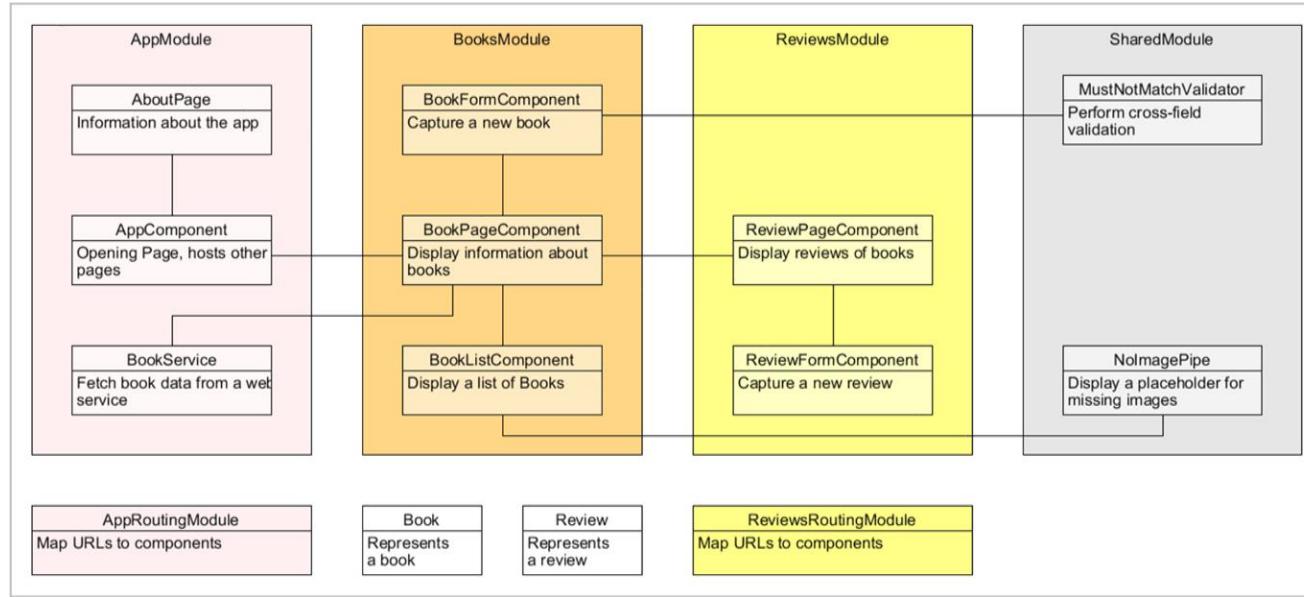
Image source: <https://angular.io/guide/architecture>

Elements of Angular Architecture

- Component (an important type of Directive)
 - Controller code for a section of user interface
 - Linked to a template (fragment of HTML)
 - Components should be thin and focused on the user interface
- Service
 - Re-usable business logic that Angular will inject as needed
- Directive
 - Re-usable code that implements standard user interface code
 - E.g., applying styles conditionally, formatting numbers
- Module
 - A grouping of thematically-related components
 - Hide components from, or expose them to, other parts of the application
 - Distinct from a TypeScript module

The BookStore Application

- It is useful to have some idea of the application you are building
 - What components it contains
 - How they relate

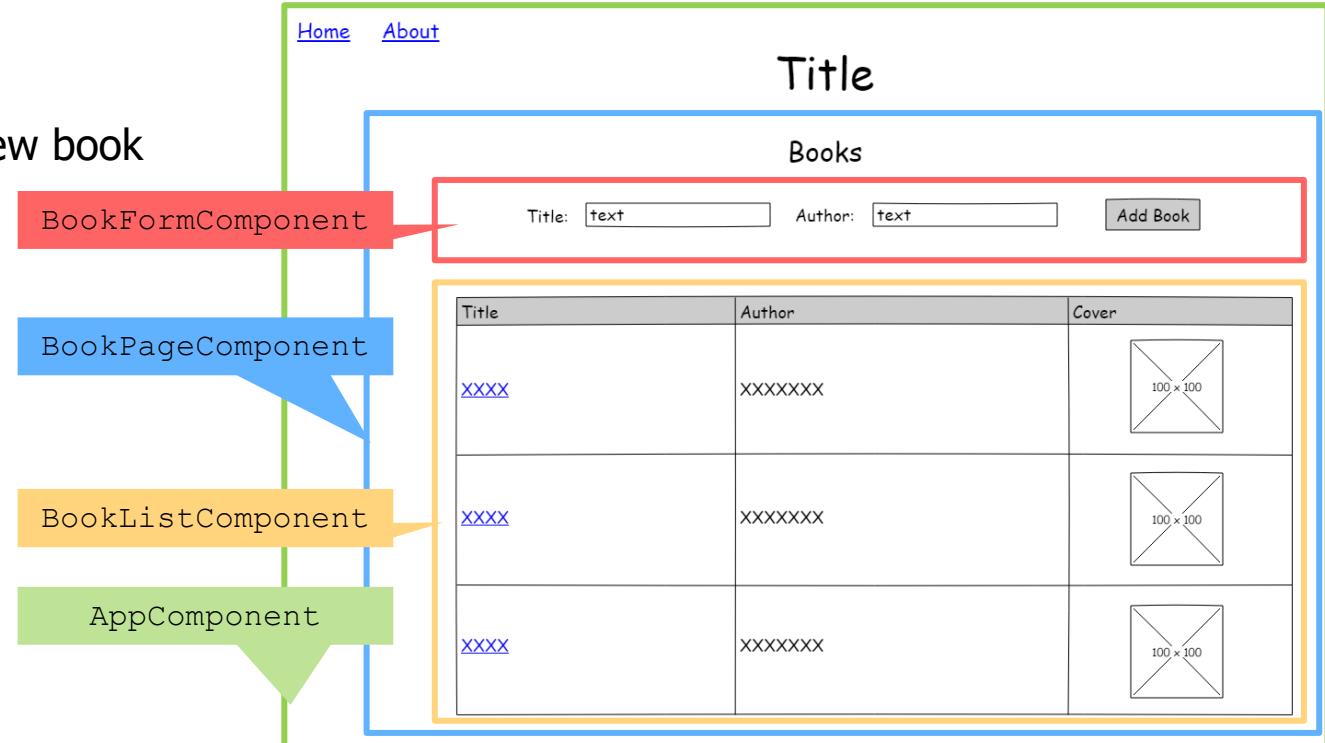


The BookStore Application (continued)

The BookPageComponent will contain most of the content of our application

A form to enter a new book

A list of books
– Title
– Author
– Cover picture



Page Components and Simple Components

- In many applications, there are two types of Component:
 - Page Components represent a whole page
 - They are mapped to a URL (we will see how in the section on Angular Routing)
 - Contain other components
 - Interact with Services
 - Simple Components represent individual blocks of the user interface
 - May be reusable across many pages, but do not have to be
 - Receive data from their host page
- By making this division, each individual component is simplified

Passing Data to Child Components

- To pass data from parent to child components:

- Child component decorates required data with `@Input()`

Import `Input` from core

```
import { Component, Input } from '@angular/core';
import { Book } from '../models/book';

@Component({
  selector: 'app-book-list',
  templateUrl: './book-list.component.html',
  styleUrls: ['./book-list.component.css']
})
export class BookListComponent {
  @Input() currentBooks: Book[] = [];
  ...
}
```

Decorate required data
with `@Input()`

- Parent component passes data in via an attribute binding
 - Name of variable = name of attribute

Name of `@Input`
annotated child property

```
<app-book-list [currentBooks]="books"></app-book-list>
```

Name of parent property



Exercise 7.4: Refactoring Components

30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Creating Angular Components

Built-In Directives

An Angular Application

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Creating an Angular component
- Defining strongly typed classes
- Passing data between a parent and child component

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 8: Angular Modules and Binding

Chapter Overview

In this chapter, we will explore:

- Creating an Angular module
- Binding properties, attributes, and events
- Using two-way binding with HTML inputs

Chapter Concepts

Organizing Code into Modules

Binding Data, Methods, and Events

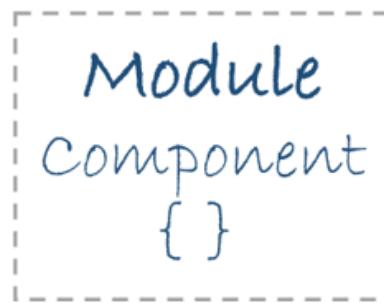
Chapter Summary

Debrief

- Currently, all the application code is inside the root module, `AppModule`
- This pattern would quickly become very complex to maintain
- Angular applications should contain multiple modules
 - One for each distinct area of functionality
- Each module imports the modules it relies on
 - `AppModule` imports the custom feature modules
 - Custom feature modules import modules they depend on

Angular Modules

- Are containers for blocks of functionality
 - Distinct from JavaScript modules
 - JavaScript modules are used to provide closures and a private scope
 - Exposing chosen properties and functions via `import` and `export`
- Angular modules are application building blocks
 - All Angular features are declared as part of a module
 - Components
 - Pipes
 - Services
 - Directives
 - Other modules consume these features by importing the parent module
- Many Angular features are themselves provided as modules
 - `FormsModule`, `HttpModule`



The `@NgModule()` Decorator

- Angular modules are classes decorated with `@NgModule()`
- Has a single object literal argument with many properties
 - imports
 - exports
 - declarations
 - providers
 - Others

The Root Module

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

import { AppComponent } from './app.component';
import { BooksModule } from './books/books.module';

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

This module relies on two other modules –
BrowserModule,
and the custom feature module BooksModule

BrowserModule makes Angular built-in directives available

AppComponent belongs to this module, and is passed to bootstrap as the root component

Custom Feature Modules

- By convention, modules are declared inside their own folder and filename includes module
 - books.module.ts should be in the books folder

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms'

import { BookListComponent } from './book-list/book-list.component';
import { BookPageComponent } from './book-page/book-page.component';
import { BookFormComponent } from './book-form/book-form.component';

@NgModule({
  declarations: [
    BookListComponent,
    BookPageComponent,
    BookFormComponent
  ],
  imports: [
    CommonModule,
    FormsModule
  ],
  exports: [
    BookPageComponent
  ]
})
export class BooksModule { }
```

The declared components belong to this module and have access to each other

CommonModule also makes Angular built-in directives available. BrowserModule should only be imported by the root module.

The exported components can be used by other modules that import this module



HANDS-ON
EXERCISE

20 min

Exercise 8.1: Creating a Module

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Organizing Code into Modules

Binding Data, Methods, and Events

Chapter Summary

Data Binding

- A core advantage of SPAs is the abstraction of the DOM
- Developers bind data to templates
 - Do not interact directly with the DOM
 - As data changes in code, the web page automatically updates
- Angular has extensive support for data binding
 - One-way binding
 - Interpolation (Chapter 7)
 - Input binding (Chapter 7)
 - Event binding
 - Output binding
 - DOM property binding
 - HTML attribute binding
 - Two-way binding
 - Form elements

Angular Binding

Two-way input binding

```
<input [(ngModel)]="country.officialName"  
        type="text"  
        name="newCountry"  
        required  
        #countryInput="ngModel" />
```

DOM Property binding

Event binding

```
<button [disabled]="!countryInput.valid"  
        (click)="submit()">  
    {{actionName}} Country  
</button>
```

Interpolation binding



Text Add comes from the
actionName property in the class

View is bound to properties
and methods of the class

```
export class CountryComponent {  
    countries: Country[] = [];  
  
    country: Country = {  
        officialName: ""  
    };  
  
    submit() { ... }  
  
    actionName: string = "Add";  
}
```

Button disabled until input valid
because of [disabled] binding

Event Binding

- Allows browser events to be bound to methods in the class

- onclick
- onkeyup
- onmouseover
- onchange
- Etc.

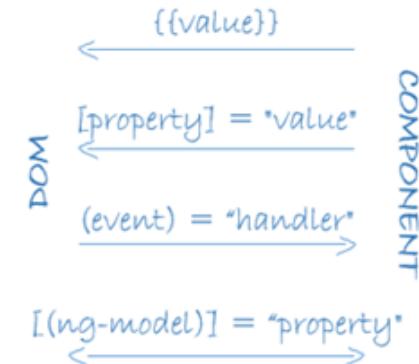
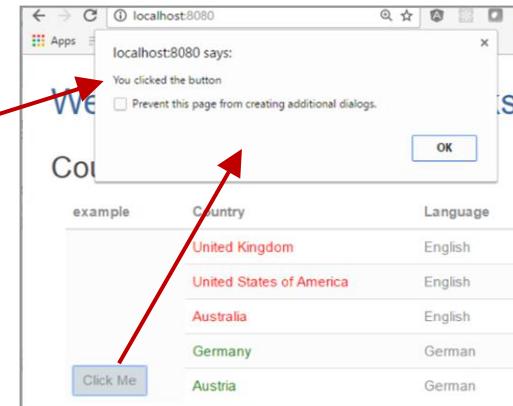
Event bindings use parentheses

country.component.html

```
<button (click)="raiseAlert()">Click Me</button>
```

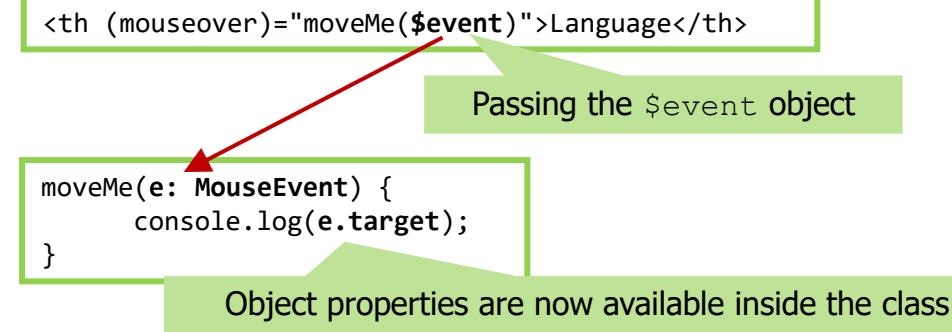
country.component.ts

```
class CountryComponent {  
  raiseAlert() {  
    alert('You clicked the button');  
  }  
}
```



The \$event Object

- Makes the event object available
- For browser events, \$event is a DOM event object
 - Components can also raise their own events and pass specialized objects
- Allows access to event properties
- \$event is passed from the binding in the template



The \$event Object Illustrated

```
raiseAlert() {
    alert("You clicked the button");
}

moveMe(e: any) { e = MouseEvent {isTrusted: true, screenX: 1040, screenY: 172}
    console.log(e.target);
}

track(r: any) {
    r.addEventListener("click", moveMe);
}
}
```

A tooltip for the `MouseEvent` object is displayed, listing its properties:

- altKey: false
- bubbles: true
- button: 0
- buttons: 0
- cancelBubble: false
- cancelable: true
- clientX: 1040
- clientY: 172

Output Binding

- Allows components to raise events
 - Enables communication *from child to parent component*
 - Opposite of `@Input()`, which flows from parent to child
- Combination of two Angular features
 - `@Output()` decorator
 - `EventEmitter` object
- The process:
 - Define a child component class property as a `new EventEmitter()`
 - Decorate the property with `@Output()`
 - Call `emit()` to trigger the event
 - Consume the event inside the parent component

The Child Component

Import Output and EventEmitter

```
import { Component, EventEmitter, Output } from '@angular/core';
import { Book } from '../../../../../models/Book.model';
```

Object to be emitted

Declare the event emitter with @Output()

```
@Component({
  selector: 'app-book-form',
  templateUrl: './book-form.component.html',
  styleUrls: ['./book-form.component.css']
})
export class BookFormComponent {
  book: Book = new Book('', '', '', -1);
```

```
@Output()
bookEventEmitter: EventEmitter<Book> = new EventEmitter<Book>();
```

```
add() {
  this.bookEventEmitter.emit(this.book);
}
```

Method that emits the event

Pass event object to be emitted

Bind the method inside the HTML template

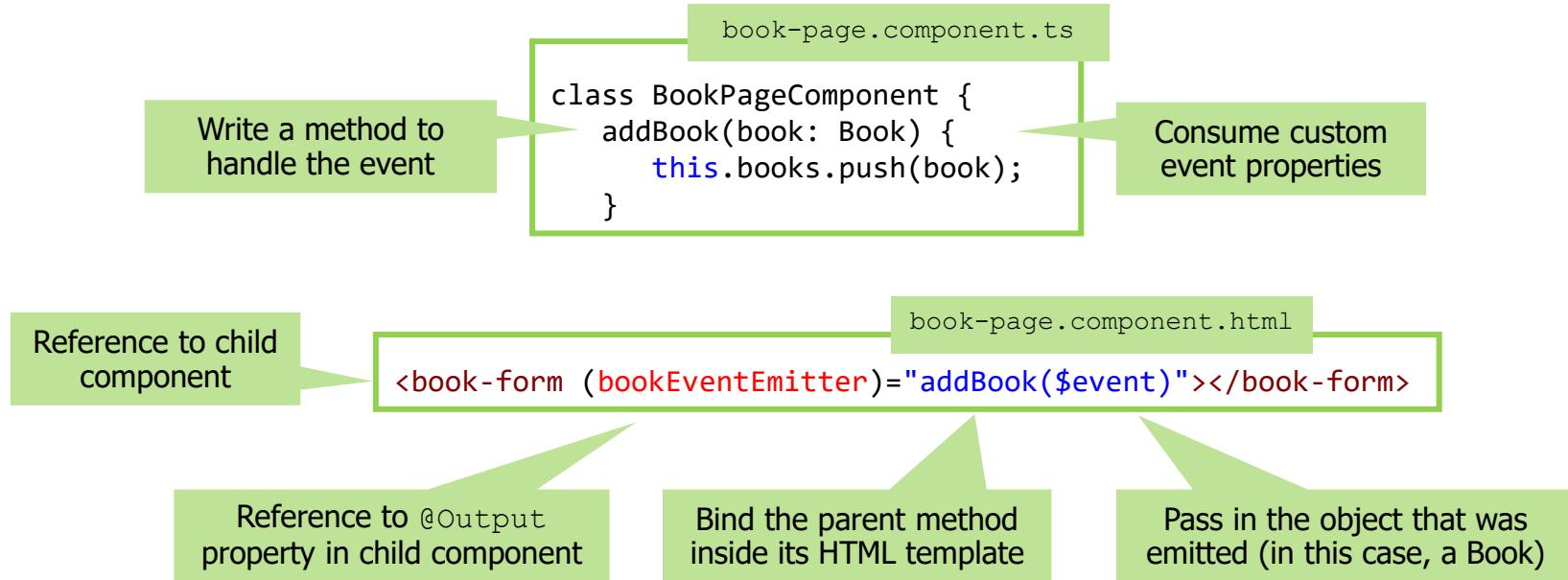
```
<button type="button"
        (click)="add()">Add Book</button>
```

book-form.component.html

book-form.component.ts

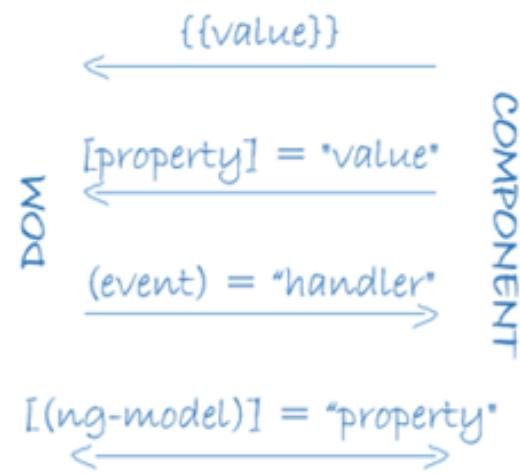
The Parent Component

- The `@Output` property is bound using the same syntax as a built-in event



NgModel and Two-Way Binding

- Two-way binding is a combination of property and event bindings
 - The “banana in a box” syntax: [()]
- Normally used with the built-in NgModel directive
 - For reading/writing values in input controls
- Functionality is provided by separate FormsModuleModule
 - Part of @angular/forms
 - Import to root module to make available to custom components



NgModel Code Sample

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { BookListComponent } from '...';
import { BookFormComponent } from '...';

@NgModule({
  ...,
  imports: [
    ...,
    FormsModule
  ]
})
export class AppModule { }
```

Add FormsModule to the module imports

Use NgModel with two-way binding syntax inside a component template

book-form.component.html

```
Title: <input [(ngModel)]="book.title" type="text" />
Author: <input [(ngModel)]="book.author" type="text" />
```

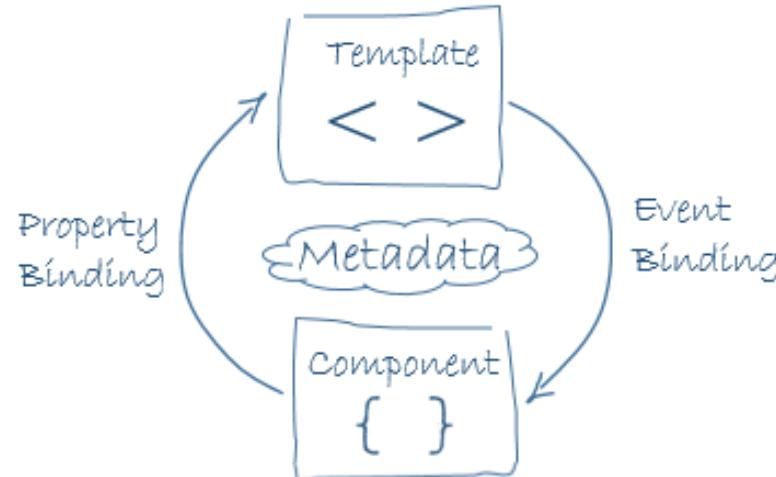
The book.author property will set the initial value of the input and will be updated as the user enters text

book-form.component.ts

```
@Component({
  selector: 'app-book-form',
  templateUrl: './book-form.component.html'
})
class BookFormComponent {
  book: Book = {
    title: string = "",
    author: string = ""
  };
}
```

Angular Binding

- Angular processes all bindings once per JavaScript event cycle
 - Starting at the root application component
 - Processing down the component tree through all child components



Testing Angular Input Events

- Test input events by triggering them
 - Get the native element that owns the event
 - Call `dispatchEvent` passing in an event of the appropriate type
 - Remember that `detectChanges()` is needed before checking any output changes

```
// trigger the click
const nativeElement = fixture.debugElement.nativeElement;
const button = nativeElement.querySelector('button');
button.dispatchEvent(new Event('click'));

fixture.detectChanges();
```

- There is no easy way to trigger an `NgModel` two-way binding programmatically
 - Changing the value of the form control does not trigger it
 - Generally, the easiest way is to set the component properties directly

Testing Angular Output Events

- Test output events using Jasmine *spies*
 - A spy is a test double function
 - Can mock or stub any function
 - Tracks calls to the function
 - Tracks function arguments
- A spy can be defined:
 - In the `describe` block
 - Or the `it` block
- A spy is removed after each spec
 - Create spies in `beforeEach()`
- Special matchers are provided to interact with spies

```
book-form.component.spec.ts
```

```
spyOn(component.bookEventEmitter, 'emit');

// trigger a click of the form's Add Book button
...
BookFormComponent.add() calls
bookEventEmitter.emit(this.book)

// check the output event was triggered
expect(component.bookEventEmitter.emit)
    .toHaveBeenCalledWith();

// check it was called with a specific argument
expect(component.bookEventEmitter.emit)
    .toHaveBeenCalledWith(book1);
```

Jasmine will track calls to the `@Output` property's `emit()` method



HANDS-ON
EXERCISE

45 min

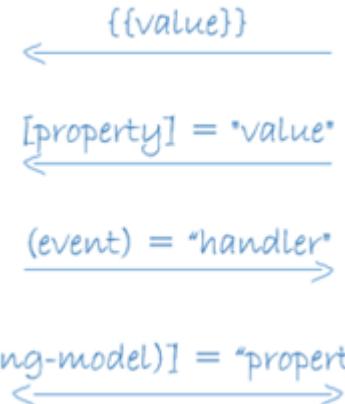
Exercise 8.2: Two-Way and Event Binding

- Please complete this exercise in your Exercise Manual

Bindings for DOM Properties and HTML Attributes

- DOM properties: <https://developer.mozilla.org/en-US/docs/Web/API#interfaces>
 - Example: look up `HTMLTableCellElement`, find `rowSpan`
 - You may need to look in parent interface to find some properties
- HTML attributes: <https://developer.mozilla.org/en-US/docs/Web/HTML/Reference>
 - Example: under "HTML Elements", look up `<td>`, find `rowSpan`
- Set DOM *properties* with `<tag [property] = "calculated-value" ...>`

```
<td [rowSpan] = "countries.length + 1" ></td>
```



- Set HTML *attributes* that are *also* DOM properties with interpolation

OK: HTML *width attribute*
is also a DOM *property*

```
<td width = "{{countries.length + 1}}" ></td>
```

- Don't use interpolation for attributes that are *not* DOM properties

ERROR: `rowspan` is
not a DOM property

```
<td rowspan = "{{countries.length + 1}}" ></td>
```

Attribute Binding Error Illustrated

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. The 'Preserve log' checkbox is checked. A red error message is displayed:

```
✖ ► Unhandled Promise rejection: Template parse errors:  
Can't bind to 'rowspan' since it isn't a known property of 'td'.  
("ry of countries; let i = index; trackBy:trackCountry">  
          <td [ERROR ->]rowspan="  
{{countries.length}}"></td>  
          <td>{{country.name}}  
ind"): CountryListComponent@14:44 ; Zone: <root> ; Task:  
Promise.then ; Value: Error: Template parse errors:  
Can't bind to 'rowspan' since it isn't a known property of 'td'.  
("ry of countries; let i = index; trackBy:trackCountry">  
          <td [ERROR ->]rowspan="  
{{countries.length}}"></td>  
          ...  
          ...
```

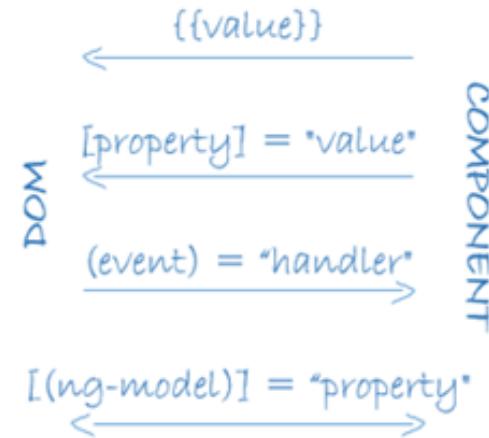
Attribute Binding Syntax

- Non-property attributes can be bound with `<tag [attr.attribute]="value" ...>`

Note no `{ { } }`. The value is an expression that resolves to a string.

```
<td [attr.rowspan]="countries.length + 1"></td>
```

Prefixing the attribute with `attr.` allows non-property attributes to be bound



- Particularly useful for Accessible Rich Internet Applications (ARIA) attributes for assistive technologies

```
<div id="percent-loaded" role="progressbar" [attr.aria-valuenow]="progress.current"  
      [attr.aria-valuemin]="progress.min" [attr.aria-valuemax]="progress.max">  
  </div>
```

Chapter Concepts

Organizing Code into Modules

Binding Data, Methods, and Events

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Creating an Angular module
- Binding properties, attributes, and events
- Using two-way binding with HTML inputs

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 9: Angular Services

Chapter Overview

In this chapter, we will explore:

- The role of services
- Building a service to retrieve data
- Making GET and POST calls using the HttpClient service
- Processing data returned from the HttpClient service

Chapter Concepts

Dependency Injection

Creating Services

RESTful Services

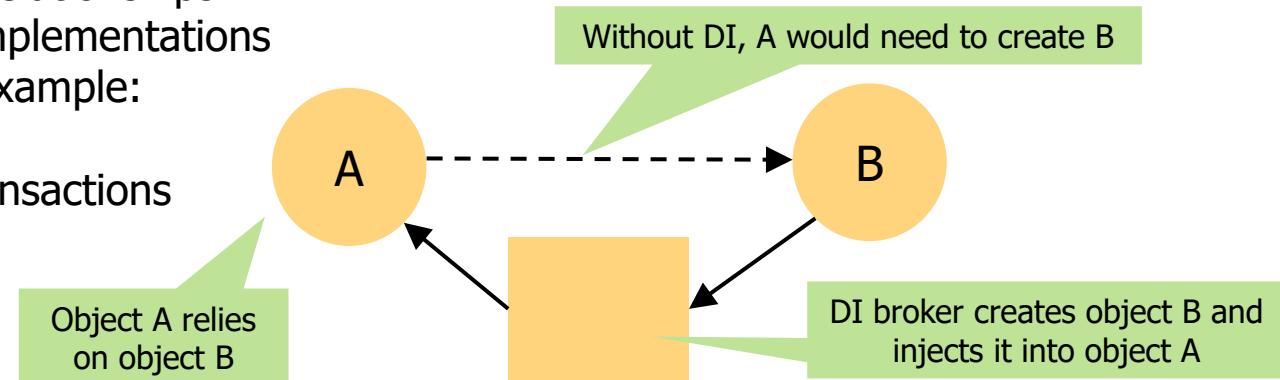
Using the HttpClient Service

Handling Errors

Chapter Summary

Dependency Injection

- An implementation of the Inversion of Control pattern
- Objects often require other objects in order to do their work
 - E.g., UI component relies on business object to supply logic
- With Dependency Injection, these objects are provided by a broker
 - Usually via constructor call or property settings
 - Avoids hard-coded relationships
 - Allows alternative implementations to be injected; for example:
 - Mocks for testing
 - Proxies for DB transactions



Angular Dependency Injection

- Angular has its own DI framework
 - Used extensively in Angular
 - Can also be used independently of Angular
- Injection is managed implicitly via constructors and providers
- Services are injected by injectors at the module level
 - injectors are configured by providers
 - Add a service to a provider using the providedIn property
 - Best practice is to provide services in the root module
- Add a service as an object property in the constructor
 - Angular injects the dependency into the object

```
@Injectable({  
    providedIn: 'root'  
})  
export class BookService {  
    ...  
}
```

BookService will be injected into the component as bookService variable, which will be a private property

```
class BookPageComponent {  
    constructor(private bookService: BookService) { }  
}
```

Lifecycle Hooks

- Components have a lifecycle
 - Managed by Angular
- Component is created, rendered, destroyed, removed from DOM, etc.
- Key stages in the process are accessible via hooks
 - Allow developers to trigger behavior at set points in process
 - Hooks are optional, and are only called if defined in the component
- Three steps to defining and using a hook:

1. Import the hook from @angular/core

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

2. Implement the hook interface

```
export class BookListComponent implements OnInit {
```

3. Implement the interface method
(hook name prefixed with ng)

```
ngOnInit() {  
    this.getBooks();  
}
```

Lifecycle Hooks (continued)

Hook	Called
OnChanges	Before OnInit and whenever data-bound content changes
OnInit	Once, after initial OnChanges and before initial DoCheck – a place to initialize property values
DoCheck	After OnChanges on every change detection run, allows for response to changes Angular doesn't detect
AfterContentInit	After transcluded/projected content initialized
AfterContentChecked	After transcluded/projected content DoChecked
AfterViewInit	After child component view initialized
AfterViewChecked	After child component view DoChecked
OnDestroy	Just before Angular destroys component

Chapter Concepts

Dependency Injection

Creating Services

RESTful Services

Using the HttpClient Service

Handling Errors

Chapter Summary

The Role of Services

- Services are simply classes that provide reusable functionality
- Used throughout the framework and in every Angular application
- Components should be thin, focused on the user interface
- Other functionality should be delegated to services, e.g.:
 - Fetching data from the server
 - Application logic
- Services are singletons
 - Only one instance per provider, usually only one per application
 - May be used to share data between components



Creating Services

- Services should have a narrow, well-defined role
 - Do one job completely
- Are simply classes
 - No base class
 - No @Service decorator
 - No service interface
- Decorate service with `@Injectable()`
 - Adds metadata to associate the service with an injector (usually a module)
 - Angular Directives (e.g., Components) do not need decoration – already support injection
- Create with Angular/CLI:
`ng generate service services/Book`
 - Services generated this way are always associated with the root module provider (Angular best practice since v6)

```
@Injectable({  
  providedIn: 'root'  
})  
export class BookService {
```

Service Example

```
import { Injectable } from '@angular/core';
import { Country } from './models/country';

const countries: Country[] = [
  { id: 1, name: 'United Kingdom', language: 'English' },
  { id: 2, name: 'United States of America', language: 'English' }
];

@Injectable({
  providedIn: 'root'
})
export class CountryService {

  constructor() { }

  getCountries(): Country[] {
    return countries;
  }

  addCountry(country: Country) {
    countries.push(country);
  }
}
```

country.service.ts

Dummy data source

This service now provides all data access relating to countries. Data retrieval and updating are exposed to components via methods.

```
export class CountryListComponent implements OnInit {

  countries: Country[] = [];

  constructor(private countryService: CountryService) { }

  ngOnInit() {
    this.countries = this.countryService.getCountries();
  }
}
```

country-list.component.ts

Inject the service

Use service methods

Asynchronous Data

- Services often return data asynchronously
 - E.g., after retrieving data from RESTful web services
 - Covered in the next section
- Synchronous calls return when task is complete (preferably immediately)
 - Method returns data
- Asynchronous calls return when task has been started (usually immediately)
 - But method does not return data
 - Returns an object that *may* contain data in the future
- Two mechanisms:
 1. Promises – a standard part of JavaScript
 2. Observables – provided via the RxJS library

The Promise Object

- Many common asynchronous JavaScript library functions return Promises
- Promises let you control the order of asynchronous calls in a clean way
 - Solves the problem of “callback hell”
- A Promise has `resolve()` and `reject()` methods
 - `resolve()` is called when an asynchronous operation succeeds
 - `reject()` is called on failure
- Client code calls `Promise.then(onFulfilled)` to retrieve data
 - `onFulfilled` is a callback function
 - The Promise runs `onFulfilled` if `resolve()` has been called
 - Argument to callback function is the data returned by the asynchronous call
- Client code calls `Promise.catch(onRejected)` to handle errors
 - The Promise runs `onRejected` if `reject()` has been called
 - A single `catch()` at the end of chain of `thens` will handle any rejection in the chain

Promise Illustrated

```
book.service.ts  
getBooks(): Promise<Book[]> {  
    return ...  
}
```

book.service.ts

Return the Promise from an asynchronous call,
e.g., a web service request or file system read

```
book-list.component.ts  
class BookList {  
    books: Book[] = [];
```

book-list.component.ts

Data is initially an empty array

```
constructor(private bookService: BookService) { }
```

```
ngOnInit() {  
    this.bookService.getBooks()  
        .then(books => {  
            this.books = books;  
        })  
        .catch(err => {  
            console.log(`Error getting books from service: ${err}`);  
        });  
}
```

Call to asynchronous method
made in ngOnInit()

Callback function passed to then()
runs when the Promise resolves

Callback function sets
component's property value with
data from asynchronous call

The Observable Object

- An Observable usually represents a series of future values
 - Promise represents a single future value
 - However, some Observables do represent a single value
 - Can convert to Promise using `.toPromise()`
- Construction is more complex than Promise
 - A simple Observable can be constructed from one value with the function `of()`
- Process results using `Observable.subscribe(next)`
 - Observable calls `next()` when new data is available
- Observables **must** be subscribed to, or they are never executed
- For more information about Observables, see Appendix B

Observable Illustrated

```
getBooks(): Observable<Book[]> {  
    return of(books);  
}
```

Usually, this would return the Observable produced by an asynchronous call...

....but for our examples, we'll call rxjs.of() to create an Observable from an array

```
class BookList {  
    books: Book[] = [];
```

Data is initially an empty array

```
constructor(private bookService: BookService) { }
```

Call to asynchronous method made during ngOnInit()

```
ngOnInit() {  
    this.bookService.getBooks()  
        .subscribe(data => this.books = data);
```

Callback function passed to subscribe() runs when the Observable produces data

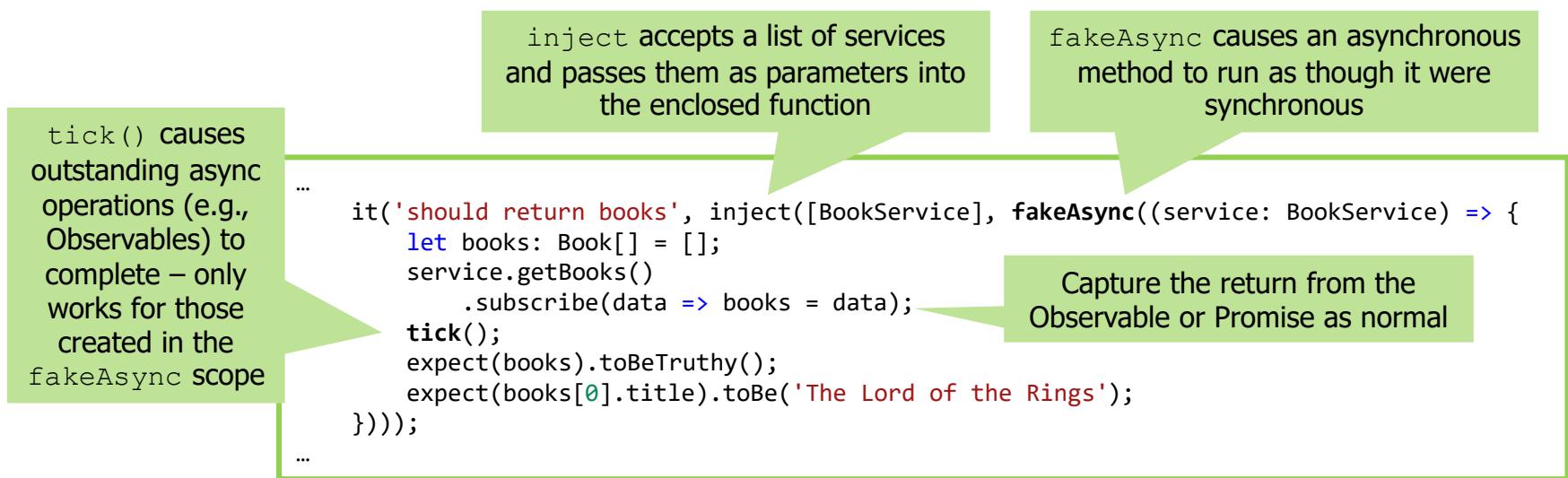
```
addBook(book: Book) {  
    this.bookService.addBook(book)  
        .subscribe(() => this.getBooks());  
}
```

Refresh the book list with the new book added

Callback function sets component's property value with data from asynchronous call

Testing Asynchronous Services

- Since services are just classes, we can generally test them like any other class
- However, asynchronous methods, Promises and Observables need a little extra work
 - inject and fakeAsync are wrapper functions that themselves take a function as a parameter



Testing Components that Depend on Services

- In general, we should be unit testing

- Do not test service while testing the component
- Use Jasmine's spy capability to create test doubles (mocks)
- Example: BookPageComponent has a dependency on BookService

book-page.component.spec.ts

```
let mockBookService = jasmine.createSpyObj('BookService', ['getBooks', 'addBook']);
const testBooks: Book[] = [{title: '...', ...}, {title: '...'}];
mockBookService.getBooks.and.returnValue(of(testBooks));
let addBookSpy = mockBookService.addBook;

TestBed.configureTestingModule({
  providers: [
    { provide: BookService, useValue: mockBookService }
  ]
  ...
})

expect(addBookSpy).toHaveBeenCalled();
```

Create a spy that implements two methods from the service

For getBooks(), act as a stub (return known data) – if using Promises, use Promise.resolve()

For addBook(), act as a spy (track invocations)

Have the TestBed return the mock service in place of the real one



Exercise 9.1: Creating and Injecting a Service

45 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Dependency Injection

Creating Services

RESTful Services

Using the HttpClient Service

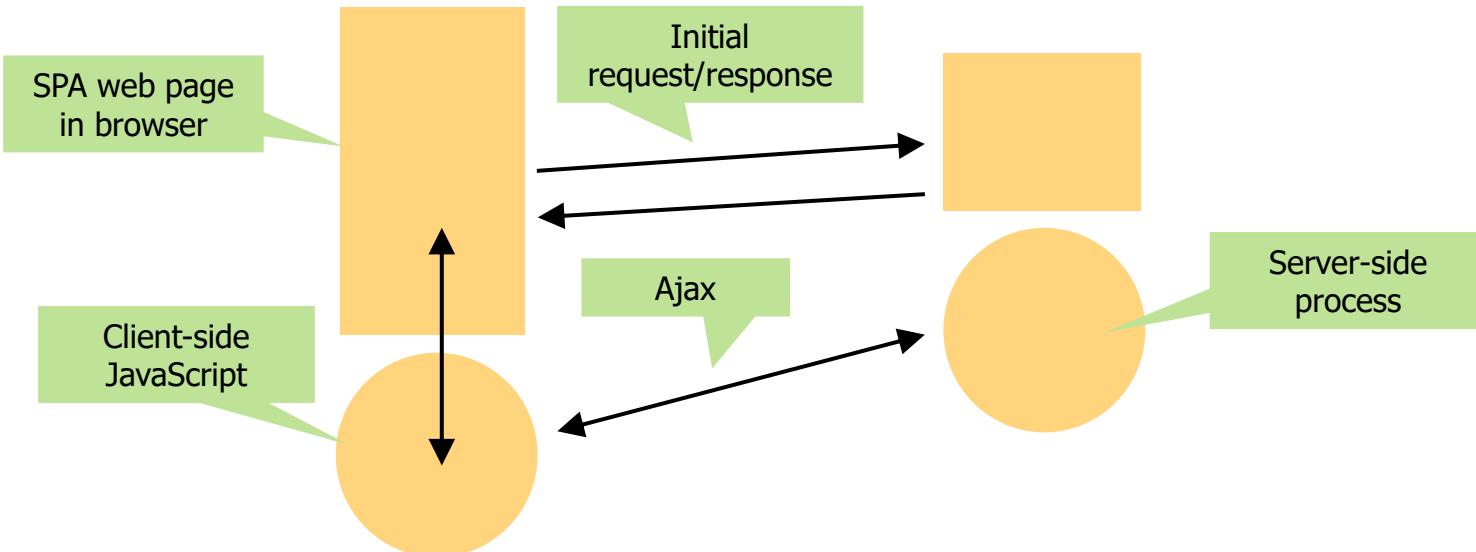
Handling Errors

Chapter Summary

SPAs and HTTP Requests

■ Recall that in Single Page Applications:

- Initial page is loaded via a normal web request
- Subsequent interaction via Ajax
 - HTTP requests made on a background thread



RESTful Services

- REST stands for Representational State Transfer
 - An alternative to “traditional” (SOAP) web services
 - Much simpler, with no complicated XML vocabulary
- Uses HTTP verbs:
 - GET, POST, PUT, DELETE
- Can return any payload
 - JSON
 - XML
 - HTML
 - Plain text
- Typically, Angular applications will make RESTful Ajax calls
 - And receive JSON data from the server

Anatomy of a REST Call

- REST maps HTTP verbs to server-side CRUD operations
 - Create = HTTP POST
 - Retrieve = HTTP GET
 - Update = HTTP PUT (or MERGE or PATCH)
 - Delete = HTTP DELETE
- Actions on the server are accessed via simple hierarchical URLs
- GET requests:
 - /Api/Books – retrieves data for all books
 - /Api/Reviews/67 – retrieves reviews for bookId 67 (path parameter)
 - /Api/Reviews/67?summary=true – retrieves review summary (query parameter)
- POST requests:
 - /Api/Books – insert new book
 - URL is same as for GET
 - Use of POST method tells the server what action to take

REST HTTP POST Illustrated

X Headers Preview Response Timing

▼ General

Request URL: <http://localhost:55979/Api/Books>
Request Method: POST
Status Code: 200 OK
Remote Address: [::1]:55979

► Response Headers (10)

▼ Request Headers view source

accept: application/json
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
Connection: keep-alive
Content-Length: 85
content-type: application/json
Host: localhost:55979
Origin: http://localhost:55979
Referer: http://localhost:55979/index.html
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) C

▼ Request Payload view source

▼ {title: "The Druid of Boston Common", author: "E.N. McMahon", cover: "", bookId: -1}
 author: "E.N. McMahon"
 bookId: -1
 cover: ""
 title: "The Druid of Boston Common"

Same URL as GET

POST tells server to call insert method

content-type tells server how the object has been serialized

JSON payload

REST Technologies

- There are two parts to any REST communication
 - Client-side and server-side
- Any server-side technology can be used to create REST services
- Angular provides a client-side solution
 - The `HttpClient` service from `HttpClientModule`
 - Found in `@angular/common/http`
 - Can choose to use alternative REST clients
 - There is an older version called `Http` from `@angular/http`
 - When looking at an unfamiliar application, always check, since these are not compatible

Chapter Concepts

Dependency Injection

Creating Services

RESTful Services

Using the HttpClient Service

Handling Errors

Chapter Summary

The HttpClient Service

■ Part of HttpClientModule

- Import to root module to make available throughout application
- Inject HttpClient into custom service classes
 - Hiding details of data access from components

■ Provides methods to call web services

- HttpClient.get<T>()
- HttpClient.post<T>()
- HttpClient.put<T>()
- Others

■ Wraps the underlying XMLHttpRequest object or Fetch API request

■ Returns Observables

- Can be converted to Promise objects
 - Via RxJS toPromise() method
- Or can use the Observable directly by calling subscribe()

Making a GET Request

1. Import `HttpClientModule` in `app.module.ts`:

```
import { HttpClientModule } from '@angular/common/http';
```

```
@NgModule({ imports: [ BrowserModule, HttpClientModule, ... ], ... })
```

2. Inject into the custom data service in `book.service.ts`

```
constructor(private http: HttpClient) { }
```

3. Pass REST service URL to `get()`

- The service should return `Observable<T>` or convert it to a Promise
 - This observable only returns once, so it is very like a Promise anyway

```
getBooks(): Observable<Book[]> {  
  return this.http.get<Book[]>(this.url);  
}
```

4. In the component, subscribe to the `Observable<T>`

```
this.bookService.getBooks()  
.subscribe(data => this.books = data);
```

Making a POST Request

Code for the `post()` method is very similar to `get()`

Key differences:

- Need to pass in the data to be sent to the server
- Need to set HTTP headers for the `content-type`

```
addBook(book: Book): Observable<Book> {  
    const httpHeaders = new HttpHeaders({  
        'Content-Type': 'application/json',  
        Accept: 'application/json'  
    });  
    return this.http.post<Book>(this.url, book, { headers: httpHeaders });  
}
```

Third argument sets
HTTP headers using
`HttpHeaders` object

Second argument is object to be
serialized to JSON

- Note that this method returns the `Observable` and must be subscribed for the `POST` to take place

Testing HttpClient Services

■ You can test HttpClient services in three ways:

1. Against the real backend
 - Reliable in identifying mapping problems or misunderstandings
 - Slow, requires backend to be available
 - Note that `fakeAsync` calls cannot involve `HttpClient`
2. Against the In-Memory Web API
 - Fast, no requirement for service
 - Must make the response look exactly like the real thing
3. Using the `HttpClientTestingModule`
 - No real requests are made
 - Good for inspecting the details of the requests (check headers, etc.)
 - Good for testing error handling, including network errors
 - Again, must be careful to reproduce the behavior of the real service

■ In practice, they all have their place in a project

The Angular In-Memory Web API

- Diverts HttpClient service REST calls to an in-memory mock solution
 - Allows development to proceed without access to REST service
- A separate package and module
 - Can be installed using npm
 - `npm install angular-in-memory-web-api --save-dev`
- Simple to use:
 1. Create mock backend service that implements `InMemoryDbService`
 - See next slide
 2. Add to test module imports (or root module to use development-wide)

```
InMemoryWebApiModule.forRoot(MockBackendService)
```

Your service implementation
 3. Write `Http` method calls as if calling remote REST service

The In-Memory Web API Illustrated

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

export class MockBackendService implements InMemoryDbService {
  createDb() {
    const countries: Country[] = [
      {
        countryId: 1,
        name: "United Kingdom",
        language: "English"
      },
      {
        countryId: 2,
        name: "United States of America",
        language: "English"
      }];
    return { countries };
  }
}
```

createDb() method implements
InMemoryDbService

Create the data structure that will be
exposed by the REST service and
return it from createDb()

HttpClientTestingModule Setup

- Add HttpClientTestingModule to the test instead of the HttpClientModule
 - Allows tests to intercept calls and respond with fake data

```
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { TestBed } from '@angular/core/testing';
...

describe('BookService', () => {
  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [
        HttpClientTestingModule
      ]
    }).compileComponents();
  });

  httpTestingController = TestBed.inject(HttpTestingController);
});
```

HttpTestingController intercepts
calls to HttpClient.get()

HttpClientTestingModule Test

- Use the `HttpTestingController` to examine the request

```
it('should return books', inject([BookService], fakeAsync((service: BookService) => {
  let books: Book[] = [];
  service.getBooks()
    .subscribe(booksFromService => books = booksFromService);

  const req = httpTestingController.expectOne('http://localhost:8080/BookService/jaxrs/books');
  // Assert that the request is a GET
  expect(req.request.method).toEqual('GET');

  // Respond with mock data, causing Observable to resolve.
  req.flush(testBooks);
  // Cause all outstanding asynchronous events to complete before continuing
  tick();

  expect(books).toBeTruthy();
  expect(books[0].title).toBe("The Lord of the Rings");
})));
```

Get the real request and inspect it

Call the method under test

Expect exactly one call to this URL. Return a `TestRequest` that contains the real `HttpRequest`.

Test data is set up in `beforeEach()`



45 min

Exercise 9.2: Retrieving and Adding Data with REST

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Dependency Injection

Creating Services

RESTful Services

Using the HttpClient Service

Handling Errors

Chapter Summary

Handling Errors—In the Service

- If there's an error, the service still must return an Observable to its client
 - pipe passes an Observable to a list of functions, and then returns the Observable
 - catchError executes if there's an error, returns a new Observable in place of the original

```
getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>(this.url)
        .pipe(catchError(this.handleError));
}
```

From rxjs/operators

```
handleError(response: HttpErrorResponse) {
    if (response.error instanceof ProgressEvent) {
        console.error(`A client-side or network error occurred: ${response.message}, ${response.status}, ${response.statusText}`);
    } else {
        console.error(`Backend returned code ${response.status}, ${body was: ${JSON.stringify(response.error)}}`);
    }
    // return an observable with a user-facing error message
    return throwError(() => 'Unable to contact service; please try again later.');
}
```

Backend returned an unsuccessful response code

From rxjs

Handling Errors—In the Component

- The component should display an error message to the user
- Steps to handling errors:

- Add a component property to hold the error message
- Use `NgIf` to display the message when it exists
- Add an error callback to `subscribe`'s argument

```
book-page.component.ts  
errorMessage: string = "";
```

```
book-page.component.html  
<div *ngIf="errorMessage" class="error">  
  {{errorMessage}}  
</div>
```

```
book-page.component.ts  
getBooks() {  
  this.bookService.getBooks()  
    .subscribe({  
      next : data => {  
        this.books = data;  
        this.errorMessage = '';  
      },  
      error: err => this.errorMessage = err  
    });  
}
```

Testing Errors in a Service

- The `HttpTestingController` provides mechanisms to simulate errors
 - HTTP errors can be returned using `request.flush()`, just like good data

```
req.flush('Forced 404', {  
  status: 404,  
  statusText: 'Not Found'  
});
```

- Network errors can be simulated using `request.error()`
 - Pass a `ProgressEvent` argument

```
const mockError = new ProgressEvent('simulated network error');  
// Respond with mock error  
req.error(mockError);
```

- Capture the error from the `subscribe` in the test
 - If you have an error handler, the error is the value returned by the `throwError` callback
 - Otherwise, it will be the exception that was thrown

Simulating an HTTP Error in a Service Test

```
it('should handle a 404 error', inject([BookService], fakeAsync((service: BookService) =>
{
  let errorMessage: string = "";
  const errorHandlerSpy = spyOn(service, 'handleError').and.callThrough();
  service.getBooks()           Call the method under test
    .subscribe({
      next: () => fail('Should not succeed'),
      error: err => errorMessage = err
    });
  const req = httpTestingController.expectOne(service.url);
  expect(req.request.method).toEqual('GET'); // verify request was a GET
  req.flush('Forced 404', { status: 404, statusText: 'Not Found' });
  tick(); // Cause all Observables to complete
  expect(errorMessage).toBe('Unable to contact service; please try again later.');
  expect(errorHandlerSpy).toHaveBeenCalled();
  let handleErrorArgs: HttpErrorResponse[] = errorHandlerSpy.calls.argsFor(0);
  let httpErrorResp: HttpErrorResponse = handleErrorArgs[0];
  expect(httpErrorResponse.status).toBe(404);
}));
```

Create a spy on a single method: it records calls, but passes them through to the real method

Call the method under test

Capture the error from the Observable returned by handleError

Return a mock error

Get the arguments of the first call to the handleError spy

Get the first argument (the HttpErrorResponse)

Exercise 9.3: Handling Errors in a RESTful Service (Optional)



20 min

- Your instructor will determine if you should do this exercise in your Exercise Manual

Chapter Concepts

Dependency Injection

Creating Services

RESTful Services

Using the HttpClient Service

Handling Errors

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The role of services
- Building a service to retrieve data
- Making GET and POST calls using the HttpClient service
- Processing data returned from the HttpClient service

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 10: End-to-End (E2E) Testing Angular Applications

Chapter Overview

In this chapter, we will explore:

- End-to-end tests
- An introduction to the Cypress tool
- Implementing E2E tests for Angular

Chapter Concepts

End-to-End Testing

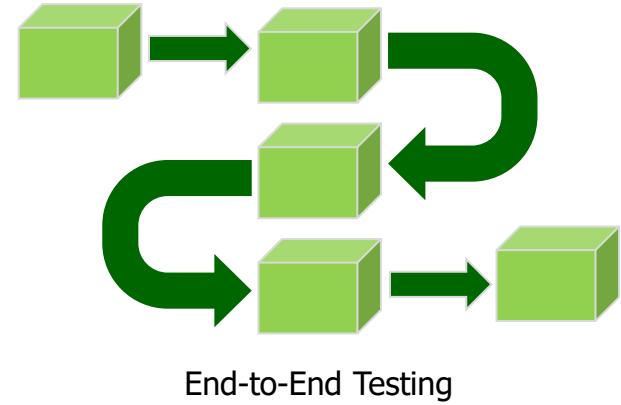
Introducing Cypress

Advanced Cypress

Chapter Summary

End-to-End Testing

- End-to-end testing (E2E)
 - A methodology used to test the flow of an application
 - From start to finish
- Testing the application from the user point of view
 - Treats the application as a blackbox
 - Only the user interface is exposed to the user (tester)
- Usually performed after functional and system testing
- Elsewhere, we will discuss different types of end-to-end testing
 - For now, we are focused on interactions of a single application
 - This is typical of E2E testing applied to a single-page application



E2E Testing vs. Unit Testing

- All the tests we have written so far have been unit tests
 - Aim to isolate a single component
 - Test every piece of functionality, including edge cases
 - Reasons for unit testing:
 - Ensure the component meets requirements
 - Eliminate regression bugs by defining a behavioral baseline
 - Create living documentation of individual classes
- By contrast, end-to-end tests:
 - Aim to test the whole application together
 - Concentrate on whether user can accomplish their goals, not edge cases
 - Reasons for end-to-end testing:
 - Ensure components work together as expected
 - Simulate the user's experience of the application (e.g., timing)

Why Perform E2E Testing?

■ E2E tests:

- Increase the overall confidence that the application will meet requirements
 - And, if there are failures, how those failures will impact end users
- Can identify integration issues that cannot be detected any other way

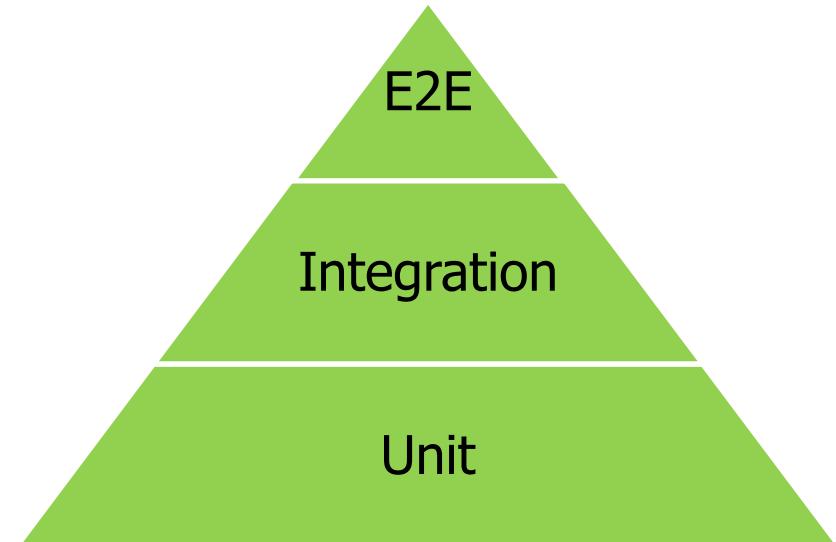
■ Issues with E2E tests:

- Too broad to be a useful diagnostic tool on their own
- Slow to execute
 - Require significant set up time and resources (e.g., external services)
- Fragile
 - More points of failure means they are more likely to fail
- Expensive to maintain
 - As the application changes, so must the tests

■ How does E2E testing fit into the overall application testing strategy?

Testing Pyramid

- This well-known graphic shows that most effort should be in creating unit tests
 - Google's rule-of-thumb: 70/20/10
 - 70% unit tests
 - 20% integration tests
 - 10% E2E tests
- As a rule, do not use higher-level tests to re-test functionality that has already been proved at a lower level
 - Use higher-level tests to test something different (e.g., how the components work together)



End-to-End Testing Scenarios: Web Shopping

- Next step: generate Use Cases from User Stories in the Product Backlog
 - Use cases use a simple, consistent text format to describe user interaction scenarios
 - Written in non-technical language so all stakeholders can review and approve them

Use case: User Places Order

Main Success Scenario:

1. User logs on
2. Application displays welcome page
3. User selects Search for an Item
4. Application displays matching items
5. User adds an item to the cart
6. User selects Proceed to Checkout
7. Application displays cart contents with options for payment and shipping methods
8. User selects payment and shipping method
9. User selects Place Your Order
10. Application displays Order Placed message
11. Application sends confirmation email

Use case: User Returns an Item

Main Success Scenario:

1. User logs on
2. Application displays welcome page
3. User selects My Orders
4. Application displays previous orders
5. User opens a previous order from the list
6. User requests a return label for an item
7. Application displays return label and instructions
8. User prints return label
9. User logs off
10. Application displays “good-bye” message

Designing End-to-End Testing

- When writing use cases, start out simple: main success scenario only
- After your team has reviewed and approved the main success scenario, add more detail
 - Alternate paths: different choices the user can make at each step
 - Error paths
 - Conditions
 - For each user function, document its conditions
 - Timing, data conditions, etc.
- Use cases are the basis for your end-to-end testing
 - For each scenario, create one or more test cases
 - Each condition should be covered in a separate test case
- For details, see *Writing Effective Use Cases* by Alistair Cockburn
 - Chapter 1 gives an excellent overview of the writing process

E2E Testing Best Practices

- Focus on the user
 - Use user stories, acceptance tests, BDD scenarios
 - Put most effort into tests that cover high-risk or high-impact scenarios
- Concentrate on the happy path
 - Leave most exception testing to lower-level tests
- Optimize the test environment
 - Should match production environment as much as possible
 - Should be quick to set up and tear down
- Make tests as reliable as possible
 - Separate test logic from presentation by using Page Object pattern (more later)
 - Ensure the test waits appropriately for the application to respond
 - Too short a wait time will result in many failures
 - Too long a wait time will mean long test cycles



Exercise 10.1: Designing E2E Testing

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

End-to-End Testing

Introducing Cypress

Advanced Cypress

Chapter Summary

Cypress

- Cypress is a JavaScript/TypeScript automation testing framework created for performing front-end testing
 - Runs the application in a real browser
 - Drives the browser, reading the screen and interacting with controls
- Can natively drive common browsers such as Chrome, Firefox, Edge
- Interacts with Angular, automatically waits for Angular to complete operations
- Cypress uses the Mocha framework under the hood
 - Mocha is a JavaScript testing framework
 - An alternative to Jasmine
 - Mocha defines its own `it()`, `describe()`, and `beforeEach()` methods

How to Use Cypress

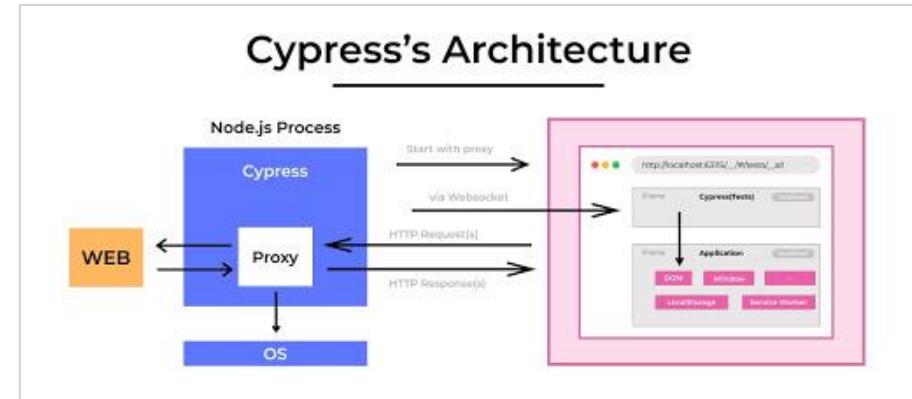
- Cypress can be used as: End-to-End, Integration, and Unit Test
 - However, we will mainly focus on the End-to-End test portion

- Cypress allows you to:

- Set up tests
- Write tests
- Run tests
- Debug tests

- Cypress ecosystem

- Download the open-source and **locally installed Test Runner**
 - Apply TDD
- Integrate Cypress with your CI/CD provider and use **Dashboard Service**, which can record your test runs



Courtesy: <https://www.lambdatest.com/blog/cypress-test-automation-framework/>

Adding Cypress to An Angular Project

- Install the official Cypress Angular schematic to add Cypress to your project:

- ng add @cypress/schematic

- Run Cypress UI for E2E testing:

- ng e2e

- Schematic: template-based code generator

- Used by ng generate **sub-commands**
 - E.g., ng generate component

```
C:\All_Code\ROI\AngularSolutionsVersion13\ManageCars_Ex14_4>ng add @cypress/schematic
i  Using package manager: npm
✓ Found compatible package version: @cypress/schematic@1.6.0.
✓ Package information loaded.

The package @cypress/schematic@1.6.0 will be installed and executed.
Would you like to proceed? Yes
✓ Package successfully installed.
? Would you like the default `ng e2e` command to use Cypress? [ Protractor to Cypress
otractor-to-cypress?cli=true ] Yes
CREATE cypress.json (298 bytes)
CREATE cypress/tsconfig.json (139 bytes)
CREATE cypress/integration/spec.ts (178 bytes)
CREATE cypress/plugins/index.ts (180 bytes)
CREATE cypress/support/commands.ts (1377 bytes)
CREATE cypress/support/index.ts (651 bytes)
UPDATE package.json (1223 bytes)
UPDATE angular.json (3902 bytes)
✓ Packages installed successfully.

C:\All_Code\ROI\AngularSolutionsVersion13\ManageCars_Ex14_4>
```

Cypress schematic:

<https://www.npmjs.com/package/@cypress/schematic>

A Simple Cypress Test

- What can we learn from a simple Cypress test?

```
describe('My First Test', () => {
  it('Visits the initial project page', () => {
    cy.visit('/')
    cy.get('h1').contains('Welcome')
  });
});
```

- `describe()` and `it()` are defined by the Mocha framework
- Cypress provides an abstraction for the browser
 - Commands to navigate to pages
 - `cy.visit('...')` – navigate to a URL
- Cypress provides an abstraction for the DOM
 - Use CSS selectors to access DOM objects and interact with them
 - `cy.get('h1')` – find the `<h1>` tag on the current page

Cypress Assertion should

- Cypress's `should()` assertion function takes a Chai *chainer* argument
 - `.should(chainer)`
 - `.should(chainer, value)`
- Usage:
 - `cy.contains('Login').should('be.visible');`
 - `cy.get('tbody tr').should('have.length', 4);`
- “Chainers” are derived from Chai or Chai-jQuery or Sinon-Chai
 - Explore some “chainers” at this link:
 - <https://docs.cypress.io/guides/references/assertions#Chai>
- Explore the powerful capabilities of `should`, and how it can be chained
 - <https://docs.cypress.io/api/commands/should#Examples>

Verifying an HTML Table Contents

- This spec tests the table of books on the BookStore main page

```
it('contains a table of books', () => {
  cy.visit('/');
  cy.get('app-root h2').contains('Books');
  cy.get('tbody > tr').should('have.length', 4);
  cy.get('tbody > tr:nth-child(1) > :nth-child(1)')
    .invoke('text').should('eq', 'Design Patterns');
  cy.get('tbody > tr:last-child > :nth-child(1)')
    .invoke('text').should('eq', 'Cryptonomicon');
});
```

Verify the book list header

Verify the number of rows in the table

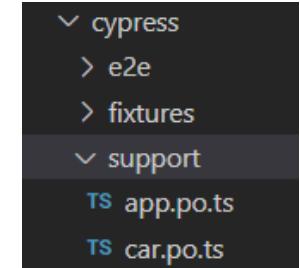
Verify the contents of some table columns

invoke() calls the DOM element's standard text() function

- Problem: Complex CSS selectors make the spec hard to read and understand
 - Specs might be brittle: what happens if the page layout changes?

Page Object Design Pattern

- Goal: Separate test logic from details of how to access individual controls
 - Allow tests to be written in terms that describe the function of the application
 - Allow test logic to remain unchanged when simple layout changes occur
- Solution: implement the Page Object design pattern
 - Create a class that represents the page
 - Defines methods to access important properties or perform actions
 - Page Object class can be added to the `cypress/support` folder
- Consequences: Each page in the application will have a separate page object
 - Page object doesn't need to expose every detail of page layout
 - Page object will change as page layout changes
 - Test logic will remain unchanged when page layout changes
- This pattern can be applied to unit testing as well as E2E testing
 - Can also be applied in technologies other than HTML and CSS



Spec Re-Written to Use Page Object

```
import { AppPageObject } from '../support/app.po';
import { TablePageObject } from '../support/table.po';

let appPage: AppPageObject = new AppPageObject();
let tablePage: TablePageObject = new TablePageObject();

it('simplifies the spec by using page objects', () => {
  appPage.navigateToHomePage();

  tablePage.checkBookTableHeader('Books');

  tablePage.getBookTableRows()
    .should('have.length', 4);
  tablePage.getBookTableFirstRowTitleColumn()
    .should('eq', 'Design Patterns');
  tablePage.getBookTableLastRowTitleColumn()
    .should('eq', 'Cryptography');
});
```

Spec is easier to read,
easier to maintain

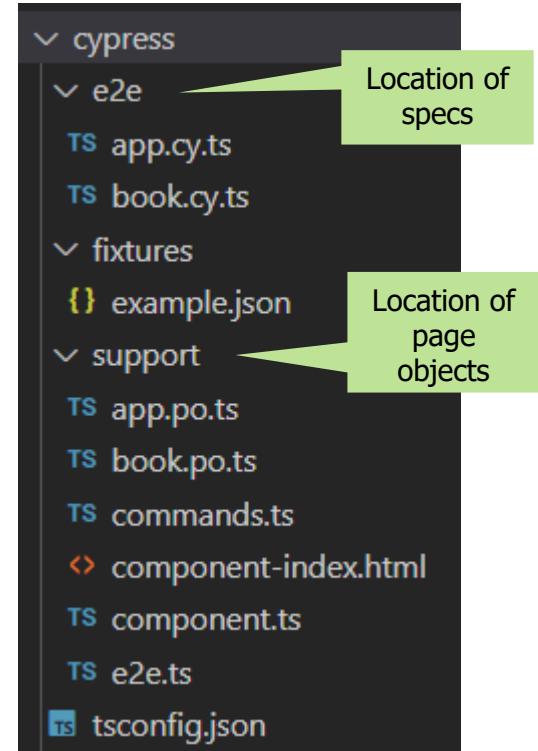
```
export class AppPageObject {
  navigateToHomePage() {
    cy.visit("/");
  }
  checkTitle(title: string) {
    cy.get('app-root h1')
      .should('contain.text', title);
  }
}
```

```
export class TablePageObject {
  checkBookTableHeader(header: string) {
    cy.get('app-root h2').should('contain.text', header);
  }
  getBookTableRows() {
    return cy.get('tbody > tr');
  }
  getBookTableFirstRowTitleColumn() {
    return cy.get('tbody > tr:nth-child(1) > :nth-child(1)')
      .invoke('text');
  }
  getBookTableLastRowTitleColumn() {
    return cy.get('tbody > tr:last-child > :nth-child(1)')
      .invoke('text');
  }
}
```

Page objects can be
reused in many specs

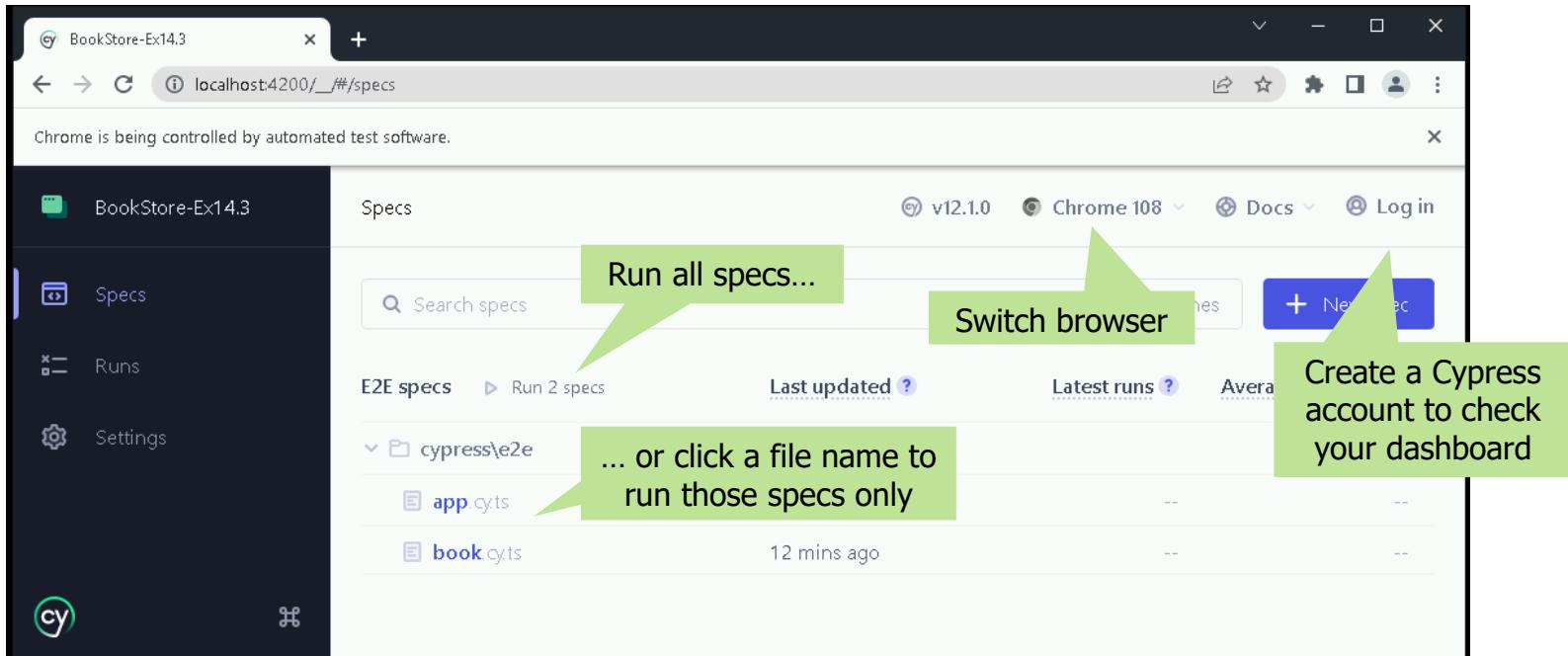
Running Cypress Tests in an Angular Project

- Cypress in Angular has a predefined folder structure
 - Tests are stored in `cypress/e2e` under the project folder
 - Tests have the extension `*.cy.ts`
 - Page objects can be stored in the `support` folder
 - Best practice: use extension `.po.ts`
- Run Cypress specs using Angular CLI
 - `ng e2e`
- Specs are compiled and executed in the browser
 - Cypress provides a UI so you control which specs are run
- For more information about Cypress, check out:
 - <https://docs.cypress.io/guides/introduction/introduction/why-cypress>



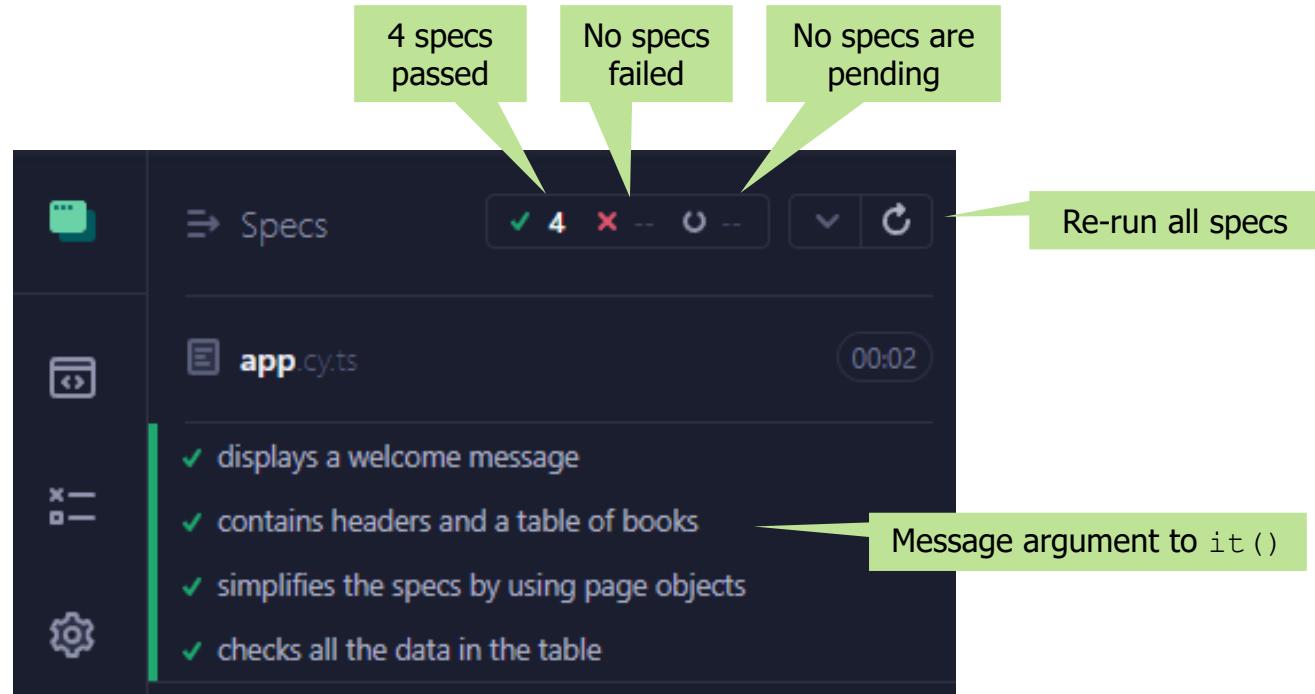
Cypress Spec Explorer

- The first time you run `ng e2e`, it starts the Cypress Launchpad
 - Select **E2E Testing**, then click **Start E2E Testing** to launch Cypress Spec Explorer



Cypress Test Runner

- After running specs, Cypress Test Runner displays the spec **Status Menu**



Debugging Specs with Cypress

Replay a spec by clicking lines in the Spec Runner

The screenshot shows the Cypress Spec Runner interface on the left and a browser window on the right.

Cypress Spec Runner:

- Shows a test named "app.cy.ts" with 4 passing tests and 0 failing tests.
- The test "My First Test" contains the following code:

```
1 visit '/'
2 get app-root h1
3 -contains Welcome to Angles on Books!
4 get app-root h2
5 -contains Books
6 get tbody > tr
7 -assert expected [ <tr>, 3 more... ] to have length of 4
(xhr) GET 200 http://localhost:8080/BookService/jaxrs/books
8 get tbody > tr:nth-child(1) > :nth-child(1)
9 invoke .text()
10 -assert expected Design Patterns to equal **Design Patterns**
```

A green callout bubble points to the line "3 -contains Welcome to Angles on Books!" with the text "Click a line to replay it". Another green callout bubble points to the line "10 -assert expected Design Patterns to equal **Design Patterns**" with the text "Cypress highlights selected element".

Browser Window:

The browser displays a web application titled "Welcome to Angles on Books!". The "Books" section lists four books:

Title	Author	Cover
Design Patterns	Gamma, Helm, Johnson, Vlissides	NO IMAGE AVAILABLE
UML Distilled	Martin Fowler	NO IMAGE AVAILABLE
Clean Code	Robert Martin	NO IMAGE AVAILABLE
Cryptonomicon	Neal Stephenson	NO IMAGE AVAILABLE

Display of Failing Specs

- When a test fails, Cypress displays an error message
 - You also get a detailed breakdown of where and why the test failed
 - Replay the spec by clicking lines in the test body above the **AssertionError**

```
11  - assert expected [ <tr>, 3 more... ] to have a length of 3 but got 4
!
! Assertion Error
Timed out retrying after 4000ms: Too many elements found. Found '4', expected '3'.
cypress/e2e/car.cy.ts:47:31
45 |         // Then click the "All Cars" button and verify the result
46 |         car.getAllCarsButton().click();
> 47 |         car.getCarTableRows().should('have.length', 3);
        ^
48 |
49 |         car.getCarTableRowColumn(1, 1).should('eq','Tesla');
50 |         car.getCarTableRowColumn(1, 2).should('eq','Roadster');

> View stack trace
Print to console
```



30 min

Exercise 10.2: Writing a Simple Cypress Test

- Please complete this exercise in your Exercise Manual



Finding an Element on a Page

- Most interaction with a page is through `cy.get('CSS-selector')`
 - Returns DOM elements that match the selector (may be more than one)
 - Waits for Angular to settle before evaluating
- `get()` returns a DOM element, but not the element's text
 - Use `invoke()` to call the DOM element's `text()` function

```
getReviewText() {  
    return cy.get('app-review-page p').invoke('text');  
}
```

- Find a specific element by the text it contains
 - `cy.get('button').contains('Add Book')`
 - `cy.contains('Add Book')`

■ Discuss the difference!

```
clickAddBook() {  
    cy.get('button').contains('Add Book').click();  
}
```

Additional Features

- Sometimes processes take longer than the default Cypress timeout (4000 ms)
 - Default value can be increased (not recommended)
 - Timeout for a specific control can be increased if needed

```
cy.contains(title, {timeout: 6000}).should('exist');
```

- Inputting text in an HTML field
 - Using CSS selectors to identify the field and call `type()`
 - Simulates keyboard input

```
cy.get('app-book-form input#title').type(title);
cy.get('app-book-form input#author').type('William Gibson');
```

Interacting with Elements

- Activate buttons or links with `click()`
- Use `type()` to enter text
 - Accepts a list of arguments
 - May be strings or special key presses
- Use `clear()` to clear an `<input>` control
- Submit a form with `submit()`
 - Only applies to a form or an element within a form
 - For a better emulation of a human user, get a reference to a button and `click()` it

Testing the Add Book Function

- This spec adds a book to the list and verifies that the new book has been added

```
import { v4 as uuid } from 'uuid';
describe('Book Page', () => {
  it('should add a book', () => {
    const title = uuid();
    cy.visit('/');
    cy.contains(title).should('not.exist');
    cy.get('button').contains('Add Book').should('be.disabled');
    cy.get('app-book-form input#title').type(title);
    cy.get('app-book-form input#author').type('William Gibson');
    cy.get('button').contains('Add Book').should('not.be.disabled');
    cy.get('button').contains('Add Book').click();
    cy.contains(title).should('exist');
  });
});
```

book.cy.ts

Use a random string for title so spec is repeatable

Verify title is not yet on the page

Verify button is disabled

Simulate keyboard input

Simulate button click

Test Re-Written to Use Page Object

```
import { v4 as uuid } from 'uuid';
import AppPage from '../support/app.po'
import BookPage from '../support/book.po'

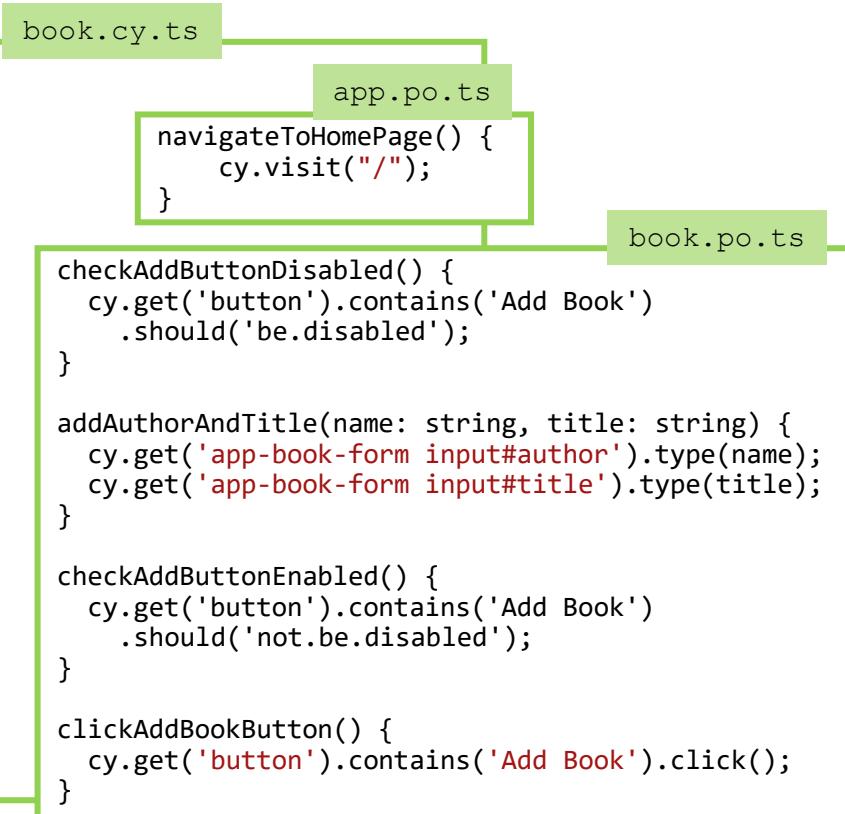
describe('Book Page', () => {
  let app : AppPage = new AppPage();
  let book : BookPage = new BookPage();

  it('should add a book', () => {
    const title = uuid();

    app.navigateToHomePage();
    cy.contains(title).should('not.exist');
    book.checkAddButtonDisabled();

    book.addAuthorAndTitle('William Gibson', title);
    book.checkAddButtonEnabled();
    book.clickAddBookButton();

    cy.contains(title).should('exist');
  });
});
```





Exercise 10.3: E2E Tests that Enter Data in HTML Inputs

30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

End-to-End Testing

Introducing Cypress

Advanced Cypress

Chapter Summary

Better Test Isolation

- Objective: Make the test independent of basic application layout and configuration
 - Text contained on individual pages that does not depend on test input
 - URLs that do not depend on test input
- In other words, make test code as self-contained as possible
 - Test should only change when application behavior changes
- Solution: Ramp Page Object up to the max!
 - Move all application layout and display information into Page Object
 - Anything that does not directly depend on test input data
 - Many tests look exactly like ours: there is no common agreement on the “right” scope
- What about external dependencies?
 - This is an E2E test, external dependencies are part of the test!
 - But some items, e.g., pre-configured test data, should not affect the test
 - In a small project, leave these in the tests
 - In a larger project, move to a separate test configuration class

Chapter Concepts

End-to-End Testing

Introducing Cypress

Advanced Cypress

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- End-to-end tests
- An introduction to the Cypress tool
- Implementing E2E tests for Angular



Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 11: Building an Application

Chapter Overview

In this chapter, we will explore:

- Building an application using everything we have learned so far!

The ManageCars Application

- The application will retrieve information about cars from a web service
- It will have the ability to select an overall list of cars, or a list of the most expensive

Title

All Cars Top 3 Cars by Price

Make	Model	Year	Doors	Price
XXXX	XXXXX	9999	9	\$999,999
XXXX	XXXXX	9999	9	\$999,999
XXXX	XXXXX	9999	9	\$999,999
XXXX	XXXXX	9999	9	\$999,999



60 min

Exercise 11.1: Building an Angular Application

- Please complete this exercise in your Exercise Manual

Chapter Summary

In this chapter, we have explored:

- Building an application using everything we have learned so far!

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 12: Pipes

Chapter Overview

In this chapter, we will explore:

- An introduction of pipes
- Building a custom pipe

Chapter Concepts

Introducing Pipes

Building Custom Pipes

Chapter Summary

The Role of Pipes

- Pipes transform data for display inside the template
 - Often necessary in order to display useful, not just accurate, data
- Without a pipe, raw dates ...

```
<dt>Date</dt><dd>{{ myDate }}</dd>
```

- display like this:

Date

Sun Jan 29 2017 09:58:41 GMT+0000 (GMT Standard Time)

- Once passed through the `date` pipe, dates...

```
<dt>DatePipe</dt><dd>{{ myDate | date }}</dd>
```

- ...display like this:

DatePipe

Jan 29, 2017

Pipes are applied to data by adding
`| name-of-pipe` after the data
to be transformed

Built-In Pipes

■ Limited number of built-in pipes

- CurrencyPipe
 - | currency
- DatePipe
 - | date
- LowercasePipe
 - | lowercase
- NumberPipe
 - | number
- PercentPipe
 - | percent
- UppercasePipe
 - | uppercase

■ Currency and date pipes rely on ECMAScript Internationalization API

Passing Parameters to Pipes

- Pipes can accept parameters to modify behavior
 - How many digits to display
 - What currency to display
- Parameters are passed as arguments after a colon

```
<!-- output '$100.23' -->  
<p>{{Value|currency}}</p>  
  
<!-- output '₹100.23' -->  
<p>{{Value|currency:'INR'}}</p>
```

```
<dt>PercentPipe with 2 digits</dt>  
<dd>{{ myNumber | percent:'2-2' }}</dd>
```

PercentPipe to 2 digits
42.70%

0-2 would mean *between* 0
and 2 digits (e.g., 42.7%)

- PercentPipe, NumberPipe argument:

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Chaining Parameters

- Multiple parameters can be chained

```
<dt>CurrencyPipe USD with $ symbol and 2 digits</dt>
<dd>{{ myLargeNumber | currency:'USD':symbol:'2-2' }}</dd>
```

CurrencyPipe USD with \$ symbol and 2 digits
\$365.00

Use code instead of symbol
for currency code (USD)

- Multiple pipes can be applied to the same data

- date formats: full (date and time), fullDate, mediumDate, yyyy-MM-dd

```
<dt>DatePipe with 'fullDate' and uppercase</dt>
<dd>{{ myDate | date:'fullDate' | uppercase }}</dd>
```

DatePipe with 'fullDate' and uppercase
SUNDAY, JANUARY 29, 2017

Chapter Concepts

Introducing Pipes

Building Custom Pipes

Chapter Summary

Custom Pipes

- Very common to create custom pipes
- Implement the `PipeTransform` interface
 - Has single `transform()` method
 - Accepts one or more arguments of any type
 - Returns transformed data of any type
- Decorate the class with `@Pipe`
- Add the pipe to the module declarations
- Test pipes like any regular class

Custom Pipes Illustrated

```
import { Pipe, PipeTransform } from '@angular/core';  
  
name is used in the template  
@Pipe({name: 'reverse'})  
export class ReversePipe implements PipeTransform {  
  
    transform(value: string): string {  
  
        return value.split("").reverse().join("");  
  
    }  
}  
  
Transform can accept and  
return any type. This custom  
pipe restricts to string.  
  
Add the class to the  
module declarations  
  
app.module.ts  
  
declarations: [ AppComponent, AboutComponent, ReversePipe ],  
  
<dt>ReversePipe</dt><dd>{{ lowerCaseText | reverse }}</dd>  
  
Reference the @Pipe in the template using the name
```



Exercise 12.1: Creating a Custom Pipe

30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Introducing Pipes

Building Custom Pipes

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- An introduction of pipes
- Building a custom pipe

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 13: Angular Routing

Chapter Overview

In this chapter, we will explore:

- The role of routing in an SPA
- The RouterModule
- Adding routes to the application
- Creating a parameterized route

Chapter Concepts

Routing in SPAs

Introducing the Angular Router

Parameterized Routes

Advanced Routing

Chapter Summary

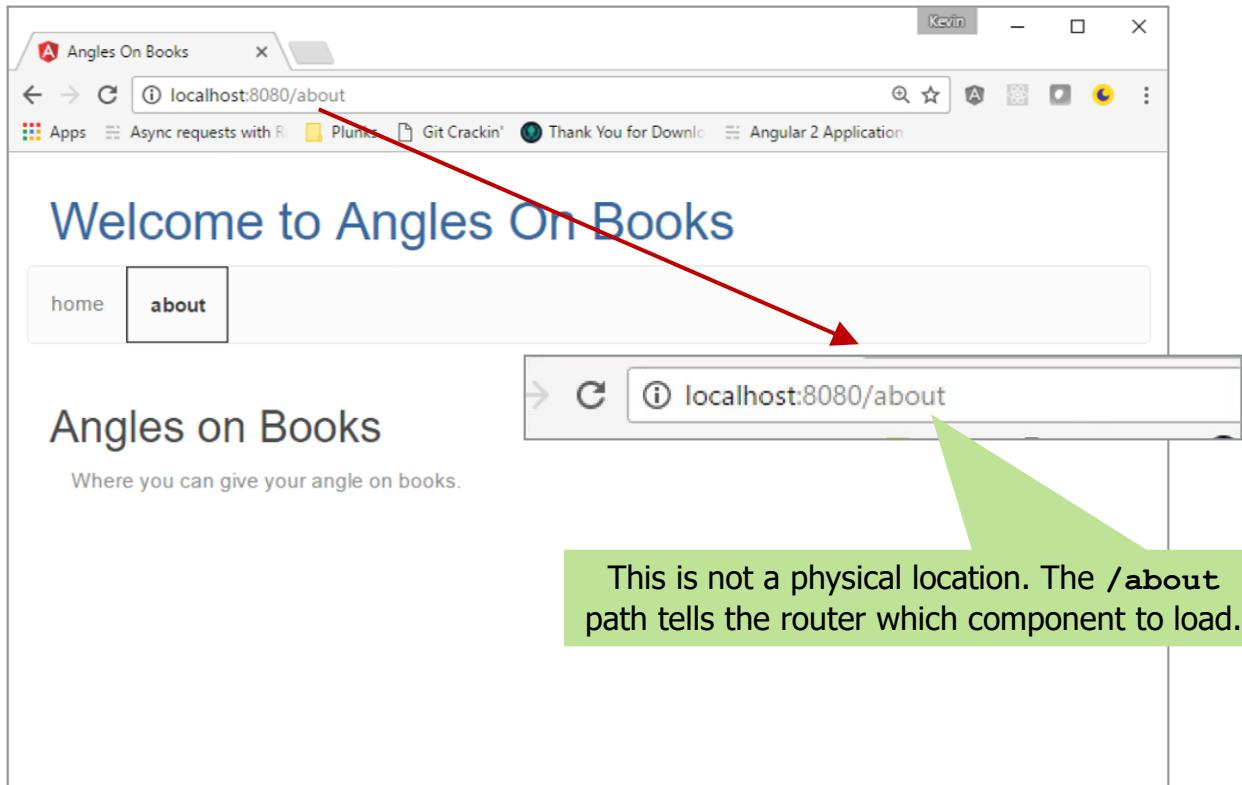
SPA Routes

- Single Page Applications are just that:
 - They only contain one web page
- All additional 'pages' are created dynamically using Ajax data
 - We consider a page to be something with its own URL
- An SPA without routing has many limitations
 - Users cannot bookmark any content except default home page
 - Very complex to manage navigation between content
 - Realistic only for very small applications
- Real-world SPA require routing

Purpose of Routing

- *Routing*: how your users navigate from one component to another
- In our app now, all components are associated with a single URL "/"
 - "/" maps to AppComponent via `index.html`
- We want to set up links so users can navigate to different components
 - `About` -- HTML physical path -->
 - `About` -- Angular logical path -->
- Your routing configuration maps logical paths to physical paths
 - `"/about"` → AboutComponent
 - `"/shoppingCart"` → ShoppingCartComponent
- The Angular Router is what makes your app an SPA
 - Allows creation of links that simulate behavior of traditional web apps ("deep linking")
 - Simplifies navigation between different parts of the application
 - Supports automation of login redirects

Routing in SPAs Illustrated



Chapter Concepts

Routing in SPAs

Introducing the Angular Router

Parameterized Routes

Advanced Routing

Chapter Summary

Routing in Angular

- Provided through a separate module `RouterModule`
- Routes defined as an array of `Route` configurations
 - Passed to `RouterModule` methods
 - `.forRoot()` used by the root module
 - `.forChild()` used by additional feature modules
 - Allows each module to manage its own routes
 - Includes everything except the router service, which is shared
- `RouterModule` provides extensive functionality
 - `RouterLink` to create HTML links
 - Access to *parameters* passed with the route
 - *Guards* to allow navigation to be cancelled or confirmed
 - Support for child routes and route transition animations
 - Programmatic redirection

Route Configuration

- Application routes are comprised of an array of individual Route objects
- Many properties available
 - path – URL segment to match
 - component – Angular component to load into <router-outlet>
 - redirectTo – navigate to existing route
 - Often used for successful login
 - pathMatch – required for redirects
 - Values: prefix (default) or full
 - canActivate – can the user access this path now?
 - Used to prevent user from leaving a page with unsaved changes
 - children – array of routes relative to another component

```
const routes: Routes = [
  {
    path: 'about',
    component: AboutComponent
  },
  {
    path: 'add',
    component: CountryFormComponent
  },
  {
    path: 'countries',
    component: CountryListComponent
  },
  {
    path: '',
    redirectTo: '/countries',
    pathMatch: 'full'
  },
  {
    path: '**',
    component: AboutComponent
  }];

```

URL path:
http://myhost/myapp/about

Which component to load for this path

Navigates to an existing route

Wildcard: matches all unmatched routes

Creating Routing Modules

- Possible to add routing directly to the root module, but not recommended
 - Managing all routes from the root module quickly becomes unwieldy
 - Routes configuration becomes very large and hard to maintain
- Better for individual feature modules to manage their own routes
- Routes can be divided into multiple specialized modules
 - One for the root
 - One for each feature module
- Specialized routing modules are then imported into the matching module
- One key difference between root and feature module routes
 - Feature module configurations use `RouterModule.forChild()`
 - Ensures same router service shared by root and feature modules

Using the RouterModule

- In most cases, add routing automatically when creating a module with CLI
 - Possible to add manually if not added when module was generated
 - We'll create an AppRoutingModule and add it to the AppModule
- Steps to routing with RouterModule:
 - In AppRoutingModule, add imports from @angular/router

```
import { Routes, RouterModule } from '@angular/router';
```

- 2. In AppRoutingModule,
define an array of Routes

```
const routes: Routes = [  
  {  
    path: 'about',  
    component: AboutComponent  
  },  
  //more routes in here . . .  
  {  
    path: '**',  
    component: AboutComponent  
  }  
];
```

Using the RouterModule (continued)

3. Add RouterModule to the @NgModule() definition of AppRoutingModule
 - Import the configured Route objects via the forRoot() method
 - Export the final RouterModule

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

4. Add the AppRoutingModule to the AppModule imports

```
imports: [  
    BrowserModule,  
    AppRoutingModule  
]
```

5. In head section of index.html, set base href to /

- Enables HTML5 History API
- Browser can navigate back without reloading

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

6. In AppComponent, remove all child components
 - Replace with <router-outlet>

Router will replace outlet with appropriate component

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

Creating Routing Modules for Feature Modules

1. Create routing modules for each feature module
 - CountryModule routes inside country-routing.module.ts
2. Define feature module specific routes inside the new module

```
const countryRoutes: Routes = [ {  
    path: 'add', component: CountryFormComponent }, {  
    path: 'countries', component: CountryListComponent }];  
country-routing.module.ts
```

3. Use `forChild()` to pass routes to RouterModule

```
@NgModule({  
    imports: [ RouterModule.forChild(countryRoutes) ],  
    exports: [ RouterModule ]  
})  
export class CountryRoutingModule { }  
CountryRoutingModule.ts
```

4. Import feature routes module to the feature module country.module.ts

```
@NgModule({  
    imports: [ CommonModule, FormsModule, CountryRoutingModule ],  
    declarations: [ CountryListComponent, CountryFormComponent ]}  
)  
export class CountryModule { }  
CountryModule.ts
```

The RouterLink Directive

- Allows HTML `a` elements to integrate with Angular routes
- Used as an attribute in place of `href` inside the opening tag

```
<a routerLink="/about">about</a>
```

Matches the path in route configuration

- Can be combined with `routerLinkActive` directive
 - Attaches a class when the path matches, permits styling of the active path
 - `routerLinkActiveOptions` controls how to match the path
 - Setting `exact` to `true` means entire path must match *exactly*
 - Not just contain the path

```
<a routerLink="/" routerLinkActive="active"  
[routerLinkActiveOptions]="{exact: true}">home</a>
```

Testing RouterModule

- RouterModule and its associated directives are complex and intertwined
 - The easiest way to test is to use RouterTestingModule
- Fortunately, most tests do not require actual routing
 - Import RouterTestingModule from @angular/router/testing
 - Add to TestBed configuration: imports: [RouterTestingModule]
- To test routing fully, need to configure RouterTestingModule with routes

```
imports: [
  RouterTestingModule.withRoutes(routes)
]
```

- The challenge:
 - Must either re-define the routes using mock components
 - Or use the real routes and real components, increasing complexity and dependencies
- Limited value in testing “normal” routing this way
 - Useful for testing programmatic routing (see later)



Exercise 13.1: Routing with the Angular Router

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Routing in SPAs

Introducing the Angular Router

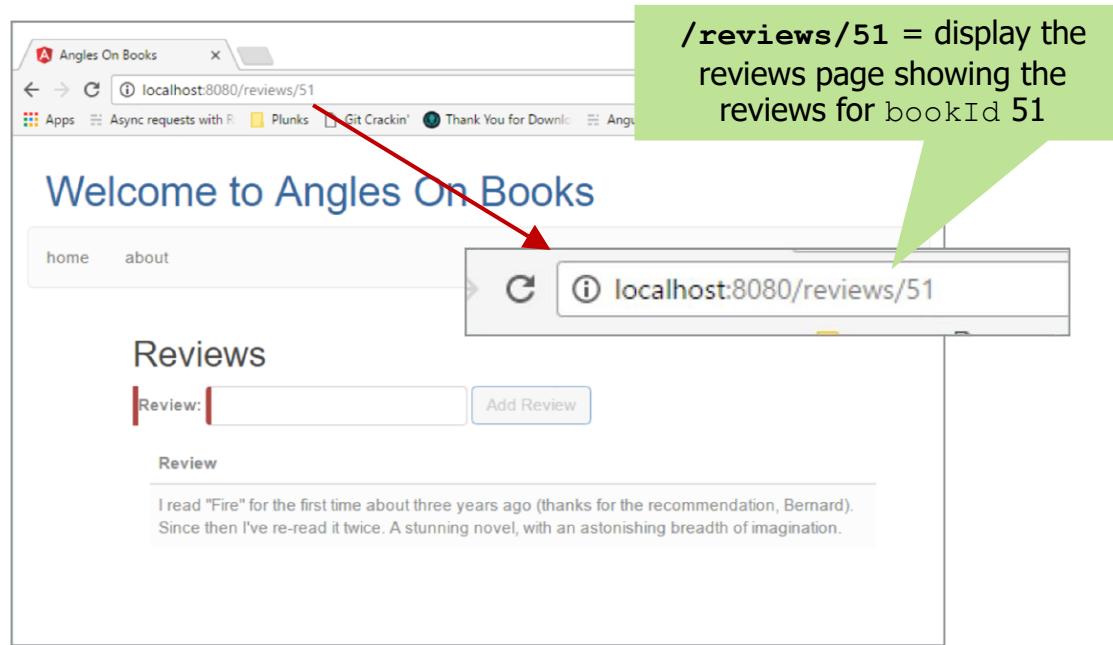
Parameterized Routes

Advanced Routing

Chapter Summary

Route Parameters

- Some routes need parameters in order to be useful
 - Can't ask for book reviews without knowing which book



Defining Route Parameters

- Parameters are defined as part of the path
 - Each parameter has its own path segment, prefixed with a colon
 - The colon is not included in the actual URL

```
const countryRoutes: Routes = [  
  {  
    path: 'country/:id/:name',  
    component: CountryComponent  
  }  
];
```

Each /:text is a parameter

- Create optional parameters by defining multiple routes
 - Most specific must come first
 - Routing uses match-first approach

```
{ path: 'country/:id/:name', component: CountryComponent },  
{ path: 'country/:id', component: CountryComponent }
```

If no name provided, second path will match.
The name parameter is therefore optional.

Passing and Retrieving Parameters

- The `routerLink` directive allows parameters to be passed via:

- An array of values

```
<a [routerLink]=["/country", country.id]> {{country.name}} </a>
```

 - The second argument in the array becomes the first parameter in the path
 - Concatenation

```
<a [routerLink]="'/country/' + country.id">{{country.name}}</a>
```

 - The directive must be in square brackets (property binding) or it will be treated as a string literal
- Parameters can be retrieved in a component via `route.snapshot`
 - Using Dependency Injection and `OnInit` (see details on next slide), or
 - With `Observable` (preferred when component may have different parameters during its lifetime)

Dependency Injection and OnInit

Dependency Injection

- Add the ActivatedRoute as an object property in the constructor
 - Angular injects the dependency

OnInit Lifecycle Hook

- A place to initialize the component
- Capture the route parameter(s)

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

Import OnInit from @angular/core and
ActivatedRoute from @angular/router

Specify that class implements
OnInit

```
export class CountryComponent implements OnInit {
    countryId: number;
    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
        this.countryId = this.route.snapshot.params['id'];
    }
}
```

Inject ActivatedRoute,
route is made into a property

Retrieve parameter value inside ngOnInit()

Testing ActivatedRoute

- Create a mock object using pre-defined values:

```
review-page.component.spec.ts
const mockActivatedRoute = {
  snapshot: {
    params: {
      id: 42
    }
  }
}
...
TestBed.configureTestingModule({
  providers: [
    { provide: ActivatedRoute, useValue: mockActivatedRoute }
  ],
  ...
})
it('should receive route parameters', () => {
  expect(component.countryId).toBe(42);
});
```



30 min

Exercise 13.2: Passing and Receiving Route Parameters

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Routing in SPAs

Introducing the Angular Router

Parameterized Routes

Advanced Routing

Chapter Summary

Programmatic Routing

- Sometimes necessary to redirect programmatically
 - E.g., redirecting user after successful login
- Steps to achieve imperative routing:

- Import Router from @angular/router

```
import { Router } from '@angular/router';
```

- Inject Router via the constructor

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

- Call `navigate()` and pass in path and additional URL path segments

```
this.router.navigate(['/reviews', this.book.id]);
```

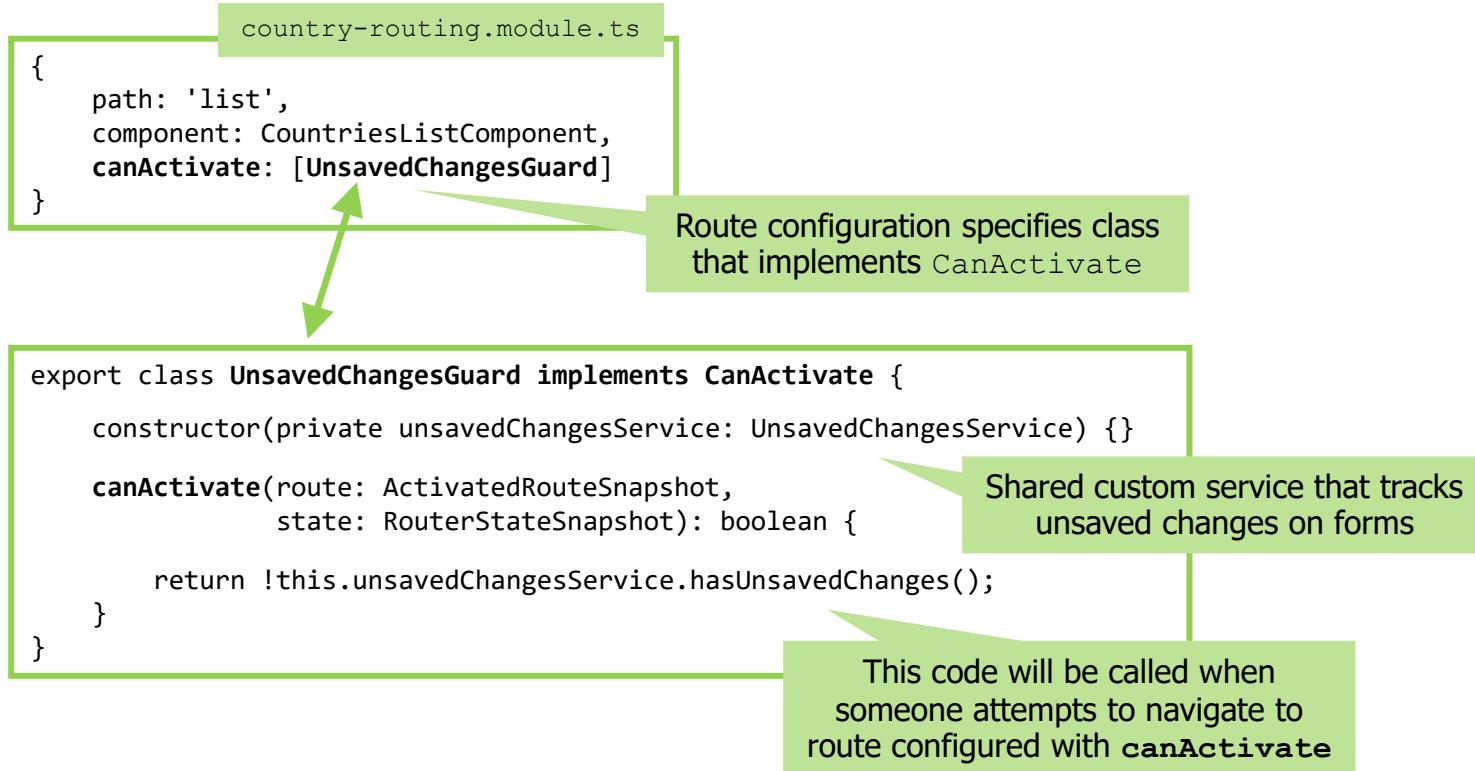
Path to redirect to

Additional path segment.
Generated URL: /reviews/42

Route Guards

- Allow programmatic intervention during routing process
 - Analogous to component lifecycle hooks
- Implemented through the Router service
- canActivate
 - Boolean return determines if change to new route succeeds
- canDeactivate
 - Boolean return determines if change away from current route succeeds
- canActivateChild
 - Similar to canActivate, but for a child route
- canLoad
 - Similar to canActivate, but for modules loaded asynchronously
- resolve
 - Allows pre-fetching of data before activating route

The CanActivate Guard



Child Routes

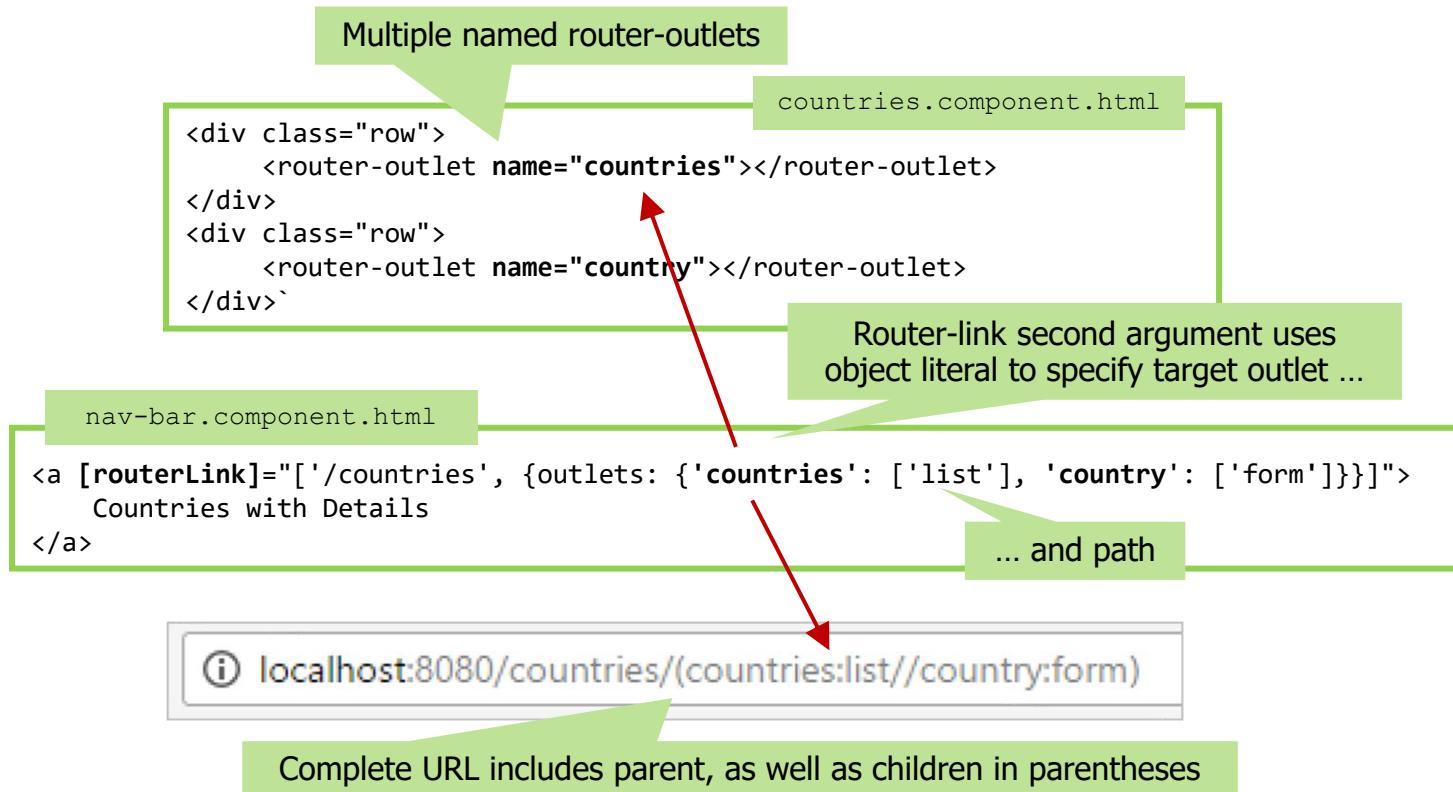
- Allow routes to target subset of a view
 - Often desirable with master-detail relationships
 - Clicking a link should refresh only the detail, not the master list
- Achieved via configuration of children and *named* router-outlets

```
{  
  path: 'countries',  
  component: CountryParentComponent,  
  children: [  
    {  
      path: 'list',  
      component: CountryListComponent,  
      outlet: 'countries'  
    }, {  
      path: 'form',  
      component: CountryFormComponent,  
      outlet: 'country'  
    }]  
,
```

Array of child routes targeting router-outlet elements inside parent component

outlet = name of targeted router-outlet

Child Route Template Syntax



Chapter Concepts

Routing in SPAs

Introducing the Angular Router

Parameterized Routes

Advanced Routing

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The role of routing in an SPA
- The RouterModule
- Adding routes to the application
- Creating a parameterized route

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 14: Angular Forms

Chapter Overview

In this chapter, we will explore:

- An introduction to Angular forms
- Building and validating template-driven forms
- Implementing and validating model-driven forms

Chapter Concepts

Introducing Angular Forms

Template-Driven Forms

Model-Driven Forms

Chapter Summary

Recap: HTML Forms

- Submit name/value pairs to the server
- Outer <form> element contains limited number of widgets
 - input
 - type attribute determines whether text, email, etc.
 - select
 - textarea
 - button
 - Requires type=submit to submit form on older browsers
- Validation provided via attributes
 - required
 - minlength
 - maxlength
 - pattern

Recap: HTML Form Example

Traditional HTML forms use
method and action attributes

```
<form class="form-group form-inline" method="post"  
      action="somescript" >
```

Widgets must have a
label to be well formed

```
<label class="control-label">Country:  
<input required class="form-control" type="text"  
      name="countryName" /></label>
```

Content must be provided

```
<button class="btn btn-primary"  
       type="submit">Add Country</button>
```

type tells input what kind of
control it represents. The name
provides name for name-value pair.

```
</form>
```

Button triggers form
submission when clicked

Angular Forms

- Angular enhances HTML forms, adding:
 - Two-way data binding
 - Control over validation
 - Change tracking
 - Error handling
- Makes it simple to create responsive, intelligent forms
- Two approaches:
 1. Template-driven forms
 - Form is managed largely through markup in template
 2. Model-driven forms
 - Form is managed primarily from the component class
 - Used when creating functional reactive forms with Observables

Chapter Concepts

Introducing Angular Forms

Template-Driven Forms

Model-Driven Forms

Chapter Summary

Template-Driven Forms

- Use form-specific directives
- Contained inside `FormsModule` in `@angular/forms`
 - Import to any module that uses template forms
- Form directives allow template to manage validation, change tracking, etc.
 - Minimal code required
- Some directives are added explicitly
 - E.g., `NgModel` on input elements
- Others are added automatically by Angular
 - `NgForm` directive is attached to every HTML form
 - `FormControl` is attached to named inputs that use `NgModel`
 - `RequiredValidator` directive added automatically
 - Whenever HTML `required` attribute present
 - Other validation directives also map to HTML5 validation attribute

The FormControl Directive

Keeps track of control's change state

- pristine – user has not yet changed the value
- touched – the control has been visited but not necessarily changed
 - Specifically, user has triggered a blur event on it (i.e., control has lost focus)
- dirty – not pristine (i.e., user changed the value)

Also tracks validity of the form

- valid
- invalid
- Based on directives that map to HTML validation attributes

```
<input type="password" #pin="ngModel" pattern="\d\d\d\d" ...>
<div *ngIf="pin.dirty & !pin.valid" class="error">
  PIN must be 4 digits
</div>
```

Include test for `dirty`,
otherwise error message
appears on initial page display

All values are also written into HTML as CSS classes with the prefix `ng-`

- E.g., a pristine control has class `ng-pristine`
- You can apply different styles based a control's state

Control State Tracking Illustrated

```
<input type="text" name="countryName" required minlength="2" .../>
```

The screenshot shows a web page with a form. The 'Country:' field contains a single character 'I'. A red arrow points from the text above to a pink error message box below it, which says 'Country must be at least 2 characters long'. Another red arrow points from the error message to the 'Elements' tab of the developer tools. In the 'Elements' tab, a green arrow points to the DOM element for the 'Country' input field, highlighting the classes 'ng-invalid', 'ng-touched', and 'ng-dirty'.

```
<input class="form-control ng-invalid ng-touched ng-dirty" minlength="2" name="name" required type="text" ng-reflect-minlength="2" ng-reflect-name="name" ng-reflect-model="I"> == $0
```

Classes `ng-invalid`, `ng-touched`, `ng-dirty` added automatically

The NgForm Directive

- Provides access to a collection of FormControl objects
- Has form-level equivalent of FormControl change and validation tracking
- pristine is true only if no inputs have been changed
 - Checks the state of all child controls
- valid is true only if every FormControl is valid
 - Can be used to disable submit button until form is valid

countryForm is a
template reference variable

```
<form #countryForm="ngForm">  
  <input type="text" name="countryName"/>  
  <button [disabled]="!countryForm.valid" type="submit">Add Country</button>
```

countryForm can be referenced
by other form controls

Validation

- Both FormGroup and FormControl have valid property
 - Combine with [disabled] to prevent behavior until valid
- FormControl has errors object
 - Error name is validator type
 - Combine with NgIf to display messages for individual errors
- Adding pristine/dirty/touched test controls when message displays

Only show error message if element is
both not valid and not pristine

```
<div *ngIf="!(countryName.valid || countryName.pristine)">
    <div *ngIf="countryName.errors?.['required']" class="error">
        Country must have a name
    </div>
    <div *ngIf="countryName.errors?.['minlength']" class="error">
        Country name must be at least 2 characters long
    </div>
</div>
```

Template Reference Variables

- Allow access to DOM elements and Angular directives *inside* the template
- Are prefixed with the # character
 - Does not form part of the variable name

```
<form #countryForm="ngForm" (ngSubmit)="submit()">
```

Inside the template, countryForm holds a reference to this instance of the ngForm directive

- Both form and input elements use template reference variables
 - Allows validation messages to reference the control's state

```
<input type="text" required name="language" #language="ngModel" [(ngModel)]="country.language" />
```

```
<div *ngIf="!(language.pristine || language.valid)" class="error">  
    Country must have a language  
</div>
```

Steps to Create Template Forms

1. Import `FormsModule` to the module containing the form

```
import { FormsModule } from '@angular/forms';
```

```
@NgModule({  
    imports: [  
        CommonModule,  
        FormsModule  
    ], ...
```

2. Add a `<form>` element and assign a template reference variable

```
<form #countryForm="ngForm" >
```

3. Add `name`, `[(ngModel)]` and template reference variable to `input`

```
<input type="text" required name="language" #language="ngModel" [(ngModel)]="country.language" />
```

Steps to Create Template Forms (continued)

4. Disable button until form is valid

```
<button [disabled]="!countryForm.valid" type="submit">  
    Add Country  
</button>
```

5. Use NgIf to add specific validation messages

```
<div *ngIf="language.errors?.['required']" class="error">  
    Country must have a language  
</div>
```

6. Implement a method to handle submission and bind to ngSubmit

```
<form #countryForm="ngForm" (ngSubmit)="submit()">
```

```
submit() {  
    console.log(this.country);  
}
```





30 min

Exercise 14.1: Creating a Template-Driven Form

- Please complete this exercise in your Exercise Manual

Testing Template Forms

- Testing template forms is challenging
 - Since all the logic is in the form, need to get reference to `NgForm` to check functionality
 - Forms update asynchronously
- Steps to test functionality:
 - Run `detectChanges()` to update all bindings
 - Wait for bindings to be updated using `whenStable()`
 - `whenStable()` is asynchronous (returns a Promise)
 - Make a change to a data value
 - Run `detectChanges()` to update all bindings and reflect new state
 - Wait for bindings to be updated using `whenStable()`
 - Make another change
 - ...
 - Finally, check form is in expected end state

Reference to NgForm

Using a query

- Instead of the CSS queries seen so far, this one searches by the Angular directive

```
const contentControl = fixture.debugElement  
  .query(By.directive(NgForm))  
  .references['revForm']  
  .controls['content'];
```



```
<form (ngSubmit)="submit()"  
#revForm="ngForm" >
```

Using @ViewChild()

- Decoration can bind any item in the view to a property in the class

```
<input required minlength="3" type="text" name="content" #content="ngModel" [(ngModel)]="review.content" />
```

```
@ViewChild('content')
```

```
contentControl: FormControl;
```

review-page.component.ts

- The issue is that this must be present in the Component class, not the test
 - Access in test as component.contentControl
- Use of @ViewChild() is complex if the form is dynamic (e.g., hidden and revealed)

Testing Asynchronously with `async ... await`

- Tests that call asynchronous methods must handle Promise return values
 - Easiest solution is `async` and `await`
- `async` (JS and TS language feature)
 - Mechanism for indicating that a function is asynchronous (returns a Promise)
 - If method does not return a Promise, result is wrapped in a Promise anyway
- `await`
 - Keyword that waits for an asynchronous function to complete
 - Unwraps the Promise, returning the data value inside, if applicable
 - Only available in `async` functions

```
it('should validate content', async () => {  
  // fixture.whenStable() returns a Promise  
  await fixture.whenStable();  
  
  const contentControl = fixture.debugElement  
    .query(By.directive(NgForm))  
    .references['revForm']  
    .controls['content'];  
  expect(contentControl.valid).toBeFalsy();  
  
  contentControl.setValue('a');  
  
  fixture.detectChanges();  
  await fixture.whenStable();  
  
  expect(contentControl.valid).toBeTruthy();  
});
```



Exercise 14.2: Testing a Template-Driven Form

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Introducing Angular Forms

Template-Driven Forms

Model-Driven Forms

Chapter Summary

Model-Driven Forms

- Are driven from the component `class`, not the `template`
 - Provide programmatic control of validation, change tracking, etc.
- Designed with testability in mind
- Same browser behavior as template-driven forms
 - E.g., adding `ng-pristine`/`ng-dirty` `classes`
- Use low-level API contained inside `ReactiveFormsModule`
 - `FormControl`
 - Programmatic representation of an individual `input`
 - `FormGroup`
 - A group of `FormControl` objects
 - Every form is itself a `FormGroup`
 - Forms can contain nested `FormGroup` objects
 - `FormArray`
 - Similar to `FormGroup`, but an array instead of an object
 - Easier to work with for dynamic forms

The Revised Form

Assign variable name to `formGroup`: *not* using a template reference variable

```
<form [formGroup]="countryForm" (ngSubmit)="submit()">
  <label>Country:
    <input type="text" formControlName="countryName" />
  </label>

  <button [disabled]="!countryForm.valid" type="submit">
    Add Country
  </button>

  <div *ngIf="countryForm.controls.countryName.hasError('required')">
    Country must have a name
  </div>

  <div *ngIf="countryForm.get('countryName').hasError('minlength')">
    Country name must be at least 2 characters long
  </div>
</form>
```

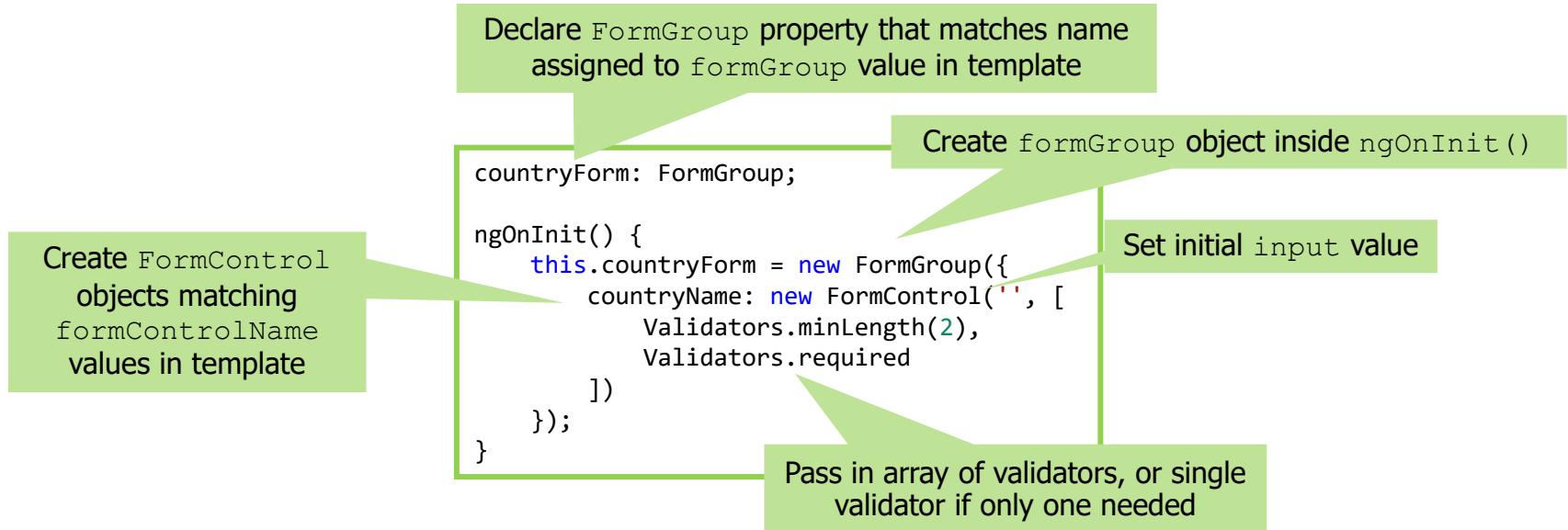
formControlName associates FormControl with name used inside the class

Two different ways of accessing FormControl

The Revised Component Class

- Builds form structure programmatically
 - Inside `ngOnInit`
- Form property inside class is of type `FormGroup`
 - Can contain additional nested `FormGroup` objects
 - *Contact* form might have sub-group *Address*
- `FormControl` objects are added to the `FormGroup` programmatically
 - One for each form control in the template
- Validator objects are passed to the `FormControl` constructor
 - Multiple validators can be passed as an array
 - Can still use HTML5 validation properties
 - If so, they *must* be used with the matching Angular validators

Building a FormGroup Illustrated



Using Validators

■ Angular provides a set of built-in validators

- required
- requiredTrue
 - Value must be *truthy*
- minlength
- maxlength
- pattern
 - For regular expression validation

■ Additional validation can be written using custom validators

- Can be written as simple functions or directives

The FormBuilder Helper

- Designed to simplify creation of forms
 - Reduces the amount of boilerplate code required to create FormGroup
 - Benefits more apparent with larger forms!
- Injected via the constructor
- Used inside ngOnInit

```
constructor(private formBuilder: FormBuilder) { }

ngOnInit() {
    this.countryForm = this.formBuilder.group({
        countryName: ['', Validators.compose([
            Validators.required,
            Validators.minLength(2)
        ]),
        language: ['', Validators.required]
    });
}
```

FormControl created implicitly

Steps for Model-Driven Forms

1. Import ReactiveFormsModuleModule to the module containing the form

```
import { ReactiveFormsModuleModule } from '@angular/forms';
```

```
@NgModule({  
    imports: [  
        CommonModule,  
        ReactiveFormsModule], ...  
})
```

2. Import required classes to the component

```
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
```

3. If using FormBuilder, inject it in the constructor

```
constructor(private formBuilder: FormBuilder) { }
```

4. Add formGroup property binding to the form

```
<form [formGroup]="countryForm">
```

5. Add formControlName to each input

```
<input type="text" formControlName="countryName" />
```

Steps for Model-Driven Forms (continued)

6. Implement ngOnInit and build FormGroup

```
ngOnInit() {
    this.countryForm = this.formBuilder.group({
        countryName: ['', Validators.required]
    });
}
```

7. Use NgIf to add validation messages

```
<div *ngIf="countryForm.controls.countryName.hasError('required')">
    Country must have a name
</div>
```

8. Disable button until form is valid

```
<button [disabled]="!countryForm.valid" type="submit">
    Add Country
</button>
```

9. Bind ngSubmit to method inside class

```
<form [formGroup]="countryForm" (ngSubmit)="submit()">
```

Submitting the Form

- Model-driven forms do not use two-way binding
- FormControl objects are independent of underlying domain objects
- Domain object must be created from form values

```
submit() {  
    const country = new Country(-1,  
        this.countryForm.get('countryName').value,  
        this.countryForm.get('language').value  
    );  
  
    //do something with object  
    console.log(country);  
}
```

Retrieve values from named
FormControl objects inside FormGroup

Testing Reactive Forms

- Testing Reactive Forms is usually easier than testing Template Forms
 - Controls update synchronously
 - Unless they invoke asynchronous validation (not covered in this course)

```
it('should validate title', () => {
  expect(component.bookForm.get('title').hasError('required')).toBeTruthy();

  component.bookForm.get('title').setValue('aaa');

  expect(component.bookForm.get('title').hasError('required')).toBeFalsy();
});
```

- Note that programmatic updates to the value of a control do not mark it as dirty unless you call `markAsDirty()`



HANDS-ON
EXERCISE

30 min

Exercise 14.3: Creating a Model-Driven Form

- Please complete this exercise in your Exercise Manual

Custom Validation

- A validator is a function (may or may not be wrapped in an object)
 - Accepts `AbstractControl` as parameter (superclass of `FormControl`)
 - Returns either `ValidationErrors` or `null` (`null` indicates valid data)
 - The errors are represented by an object with a property named after the error

```
export function mayNotBeDuneValidator(control: AbstractControl): ValidationErrors | null {  
    return control.value === 'Dune' ? { 'mayNotBeDune': true } : null;  
};
```

```
fc = new FormControl('', mayNotBeDuneValidator);
```

- Validators that accept parameters are created by a factory function
 - A function that returns a validator

A reference to the validator function

```
export function mayNotBeValidator(disallowed: string): ValidatorFn {  
    return (control: AbstractControl): ValidationErrors | null => {  
        return control.value === disallowed  
            ? { 'mayNotBe': { 'disallowed': disallowed } } : null;  
    };  
}
```

Invoke the factory function

By convention, error includes additional information as a nested object

```
fc = new FormControl('', mayNotBeValidator('Dune'));
```

FormGroup

- FormGroup can contain FormControls or other FormGroups
 - FormGroup and FormControl share a common ancestor: AbstractControl
 - So FormGroup has many of the same features as FormControl
 - Every FormGroup must have a corresponding element in template
 - Typically form, fieldset or div
- FormGroup has a value
 - An object containing a property for each control
 - getValue() returns the object
 - setValue() sets form control values to matching object literal
 - Any missing FormControl or value raises error
 - patchValue() allows a subset of controls to be updated

```
fg.setValue({  
    title: 'Dust',  
    author: 'Hugh Howey'  
});
```

FormGroup Validators

Can attach validation to FormGroup

- Useful for cross-control validation
- Validator receives the FormGroup as an AbstractControl
- FormGroup has valid property and hasError() method

```
this.bookForm = this.formBuilder.group(  
  {  
    title: ['', [  
      Validators.required,  
      Validators.minLength(3)  
    ]],  
    author: ['', Validators.required]  
  }, {  
    validators: titleAuthorMustNotMatch  
  });
```

Validator attached at
FormGroup level

```
<div class="error" *ngIf="!bookForm.pristine && bookForm.hasError('mustNotMatch')">  
  Title and Author may not have the same value  
</div>
```

Exercise 14.4: Implementing Cross-Field Validation (Optional)



30 min

- Your instructor will determine if you should do this exercise in your Exercise Manual

Functional Reactive Forms

- Uses Observables to view form as a stream of changing values
- Allows real-time interaction with the data
 - Validation of partial matches
 - Suggestions
- More details on this and Observables in Appendix B

Chapter Concepts

Introducing Angular Forms

Template-Driven Forms

Model-Driven Forms

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- An introduction to Angular forms
- Building and validating template-driven forms
- Implementing and validating model-driven forms

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Chapter 15: Angular Deployment

Chapter Overview

In this chapter, we will explore:

- How to use the Angular environment
- Considerations for automated builds
- Using lazy loading of a feature module
- Deploying the application

Chapter Concepts

Angular environment

Building Angular Applications

Deploying Angular Applications

Chapter Summary

Angular Environment

- Prior to Angular 15, ng new generated src/environments/environment.ts
 - Contained an object literal environment with one property production

environment.ts

```
// This file can be replaced during build by using the `fileReplacements` array.  
// `ng build ---prod` replaces `environment.ts` with `environment.prod.ts`.  
// The list of file replacements can be found in `angular.json`.  
  
export const environment = {  
  production: false  
};
```

- Application components could import the environment to read configuration values

```
import { environment } from '../environments/environment';  
  
@Component({ ... })  
export class AppComponent {  
  
  isProductionMode: boolean = environment.production;
```

Available Environments

- Angular 15+ does not generate `environment.ts` by default
 - To generate environment files, run `ng generate environments`
 - Creates `environments/environment.ts` and `environment.development.ts`

```
export const environment = {};// empty object literal
```

`environment.development.ts`

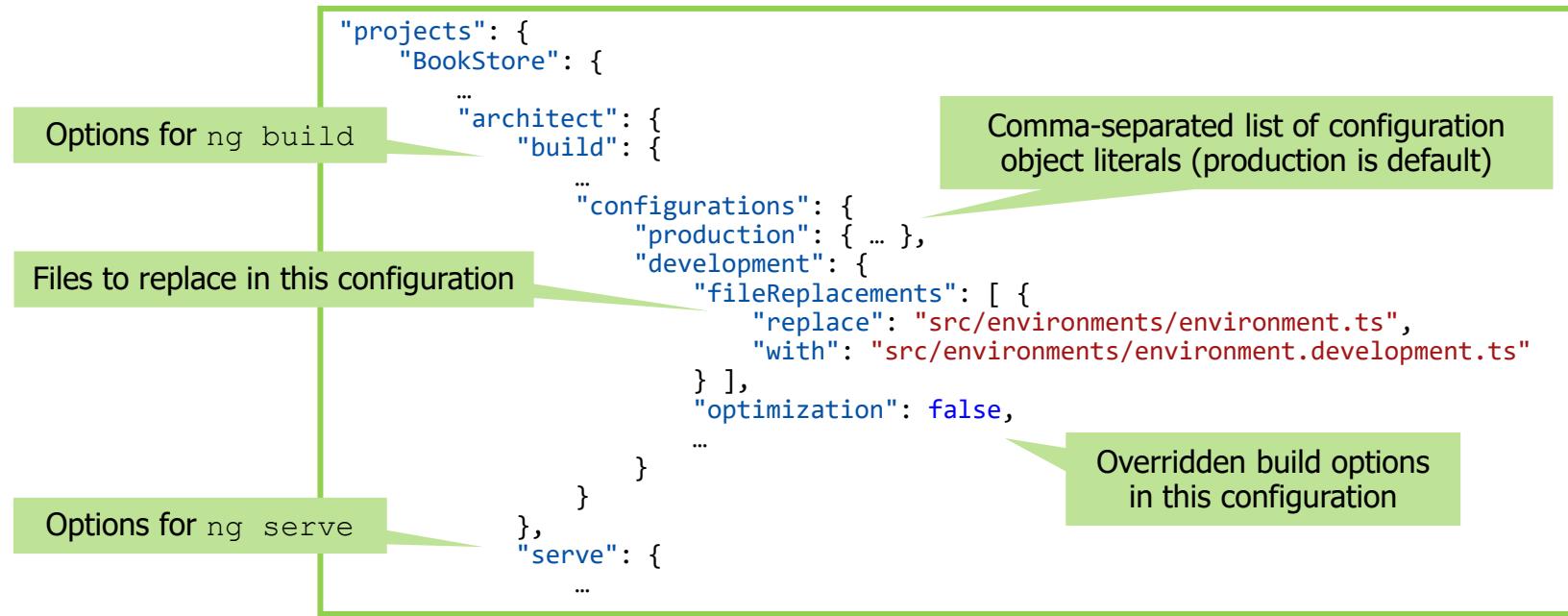
- To add additional environment data items, add properties to the object literal

```
export const environment = {  
  serverBaseUrl: "http://localhost:8080/",  
  debugMode: true  
};
```

- Build process replaces default `environment.ts` with a configuration-specific file
 - `environment.ts` – production configuration
 - `environment.development.ts` – development configuration

Angular Configurations

- Define project configurations in `angular.json` file in the project root directory



Adding Configurations

```
"build": {  
  ...  
  "configurations": {  
    "production": {  
      ...  
    },  
    "qa": {  
      "fileReplacements": [ {  
        "replace": "src/environments/environment.ts",  
        "with": "src/environments/environment.qa.ts"  
      } ],  
      ...  
    }  
  },  
  "serve": {  
    ...  
    "configurations": {  
      "production": {  
        "browserTarget": "HelloWorld:build:production"  
      },  
      "qa": {  
        "browserTarget": "HelloWorld:build:qa"  
      }  
    }  
  }  
}
```

No file replacements for production configuration

New qa configuration defines a new environment file

Using Configurations

- To use a configuration, specify it at the command line

```
ng serve --configuration=qa
```

- Angular also understands the synonym `--prod` for `--configuration=production`
 - In general, you should not run production using `ng serve`
 - Exists to permit configuration testing

Chapter Concepts

Angular environment

Building Angular Applications

Deploying Angular Applications

Chapter Summary

Building Angular Applications

- In development, use the “live” server
 - `ng serve`
 - A Node.js server running on port 4200
 - Permits incremental build
- Do *not* attempt to deploy the “live” server into a production environment
- Use the Angular build process:
 - Transpiles from TypeScript to JavaScript
 - Bundles all JavaScript files into a small number of larger files
 - Reduces size of bundles by removing whitespace, etc. (minifies)
 - Uglifies code: makes identifiers as short as possible for faster downloads
 - Converts to production settings using environments
 - For Angular and application-specific constants
 - Ensures unused dependencies are not included in bundles (tree-shaking)

Build Application With Angular CLI

Simple build

```
ng build
```

- Creates `dist` sub-directory that contains everything needed
 - Copy this to server root
- If copying to a server sub-directory, change the base href:

```
ng build --base-href="/path/on/server/"
```

Angular is very fussy and expects the URL to have leading and trailing /

Optimized build

```
ng build --prod
```

- Uses ahead of time compiler
- Bundles, minifies, and uglifies
- Eliminates dead code and unused imports (“tree-shaking”)
- Enables production mode (`--configuration=production`)

Lazy Loading

- Angular router can be configured to use asynchronous routing
 - Only downloading resources as necessary
 - Also known as lazy loading
 - Reduces the size of the initial bundle
 - Improving start-up performance
- In this approach, modules are loaded via the router
 - Uses `loadChildren()` method in configuration
 - Remove any lazy loaded modules from root module imports (`AppModule`)
 - Lazy loading automatically enabled, provided router configured correctly

```
const routes: Routes = [ ... other routes
, {
  path: 'reviews',
  loadChildren: () => import('./reviews/reviews.module')
    .then(mod => mod.ReviewsModule)
}];
```

Path to feature module relative to `AppModule`. This is now the only reference to the feature module from the `AppModule`.

The `/reviews` part of the path is now taken care of by the `AppRoutingModule`

```
const routes: Routes = [
  path: ':id',
  component: ReviewPageComponent
];
```



Exercise 15.1: Lazy Loading a Feature Module (Optional)

15 min

- Your instructor will determine if you should do this exercise in your Exercise Manual

Chapter Concepts

Angular environment

Building Angular Applications

Deploying Angular Applications

Chapter Summary

Deploying Angular Applications

- A minimal deployment requires moving the following to the server:
 - The JavaScript versions of the application's TypeScript files
 - All non-JavaScript resources
 - index.html
 - CSS files
 - Images
 - Any additional dependencies
- In other words, once built, it looks just like any other website
 - Can deploy to any suitable server
- Which server?

Which Server?

- Most Angular applications use at least one web service
 - Common to deploy to the same server
 - Avoids Cross-Origin Resource Sharing (CORS) issues
 - We will cover CORS in more detail later in the program
 - Defines a safe way to allow access to a service from other servers
 - Each service can specify a CORS policy
 - The test service we have been using permits all cross-origin requests
 - This is unrealistic and undesirable in a production application
- To completely avoid all CORS requests, we can create an integration tier
 - A server that provides a common interface to all the services used by the application
 - Often built using Node.js
 - Angular application can be served from the same server
 - This is not the only reason for creating an integration tier
 - We will cover integration tiers in more detail later in the program

Build and Deployment Tools

- We have seen how the Angular CLI may be used to build the application
 - There are many other tools available
- Ideally, would like the build to create the final deployment package
 - Continue to use Angular CLI natively for development builds
 - Choose a production build tool that integrates with the target environment
 - Integrates with Node.js if deploying with an integration tier
 - Integrates with Java if deploying with a Java web service
 - Use the same build tool on continuous integration servers (e.g., Jenkins)
- Node.js applications can be built using a variety of tools
 - E.g., gulp, grunt, webpack (the tool underlying Angular CLI)
- Angular can be built using Maven, a common Java build tool
 - The maven-frontend-plugin allows Maven to run npm

Configure the Server

- Configure server to return `index.html` for 404 errors
 - When server doesn't recognize an Angular path, it will respond with `index.html`
 - Then Angular routing takes over and navigates to the appropriate Angular component
 - Example: GET `/index.html/reviews/12`
 - Server attempts to find resource `reviews/12`
 - 404 redirects to `index.html`
 - Angular reads response with `index.html`
 - And load the reviews for `bookId 12`
- Allows deep-links to be saved as bookmarks, not just used while the application is running

Changing RESTful Service Address for Production

- Approach to this will depend on deployment approach
 - Do not want to have to modify code when building
- Add URLs to Angular environment
 - A good plan to centralize any configuration variables to the environment anyway
 - Even if not using it to get a different address in development and production
- Use the live server proxy
 - Next slide

Configuring a Server Proxy

- Add a proxy.conf.json file to the src folder

- E.g., to redirect localhost:4200/api to localhost:8080/BookService/jaxrs

When deployed, we expect service to be on same server at URL /api

```
{  
  "/api": {  
    "target": "http://localhost:8080/BookService/jaxrs",  
    "secure": false,  
    "pathRewrite": {  
      "^/api": ""  
    }  
  }  
}
```

Remove /api from the URL before appending it to target

- Add an options element with a proxyConfig entry to angular.json

```
...  
  "serve": {  
    "builder": "...",  
    "options": {  
      "proxyConfig": "src/proxy.conf.json"  
    },  
  ...  
}
```



Exercise 15.2: Building and Deploying the Application

30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Angular environment

Building Angular Applications

Deploying Angular Applications

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How to use the Angular environment
- Considerations for automated builds
- Using lazy loading of a feature module
- Deploying the application

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Course Summary

Course Summary

In this course, we have:

- Learned to develop responsive web applications
- Understood what makes a web application responsive and interactive
- Built interactive websites with HTML
- Employed behavior in web pages using JavaScript and AJAX
- Standardized presentation using Cascading Style Sheets (CSS)
- Leveraged the capabilities of HTML5 and CSS3
- Understood the need for JavaScript frameworks such as jQuery, node.js, and Angular
- Performed End-to-End testing with Jasmine and Protractor

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Appendix A: Angular Directives

Appendix Overview

In this appendix, we will explore:

- Creating directives
- Learning how to build structural directives
- Creating an attribute directive

Appendix Concepts

Introducing Directives

Structural Directives

Attribute Directives

Appendix Summary

Types of Directive

- There are three kinds of directives:
 - Components
 - Structural directives
 - Attribute directives
- Components
 - Directives with a template
 - You've already created several of these!
- Structural directives
 - Directives that change the structure of the DOM
 - NgFor
 - NgIf
- Attribute directives
 - Directives that change the behavior or appearance of a DOM element
 - NgModel
 - NgStyle
 - NgClass

Which Type to Create?

- Far the most common directives are components
 - Flexible, reusable building-blocks of the application
 - Combination of template, data, and methods
 - Usually, the right choice
 - Highly reusable inside an application
 - Minimally reusable *between* applications
- Structural and attribute directives do not contain templates
 - Are focused on DOM elements *within* templates
 - Can create functionality reusable both within and between applications
 - A structural directive for a loop with built-in paging
 - An attribute directive to make an element draggable
- Choose a structural directive to add/remove DOM elements
- Choose an attribute directive to add behavior to *existing* DOM elements

Appendix Concepts

Introducing Directives

Structural Directives

Attribute Directives

Appendix Summary

The NgSwitch Directive

- Tests a single variable against multiple values
 - Similar to a programmatic switch statement
- Adds and removes elements from DOM like NgIf
- Includes a default option to simplify logic

```
<tr [ngSwitch]="book.author">
  <td colspan="2" *ngSwitchCase="'J R R Tolkien'>
    The man himself.
  </td>
  <td colspan="2" *ngSwitchCase="'Christopher Tolkien'>
    His son.
  </td>
  <td colspan="2" *ngSwitchDefault>
    Somebody else altogether
  </td>
</tr>
```

[ngSwitch] placed on container element

Note string value *inside quotes*

*ngSwitchCase placed on element to add to DOM

*ngSwitchDefault applies where no condition matches

Structural Directives

- Structural directives manipulate the DOM:

- Add elements
 - Remove elements
 - Change elements

- Apply to a host element

The table data <td> element is the host of the NgIf directive

```
<td *ngIf="country.language == 'English'>  
    English spoken here  
</td>
```

- Directives have a class name (NgIf) and an attribute name (ngIf)
- Structural directives start with an asterisk (*)
 - A convenience notation that Angular expands

The ng-template Element

- Structural directives use the template element
 - HTML5 feature designed to hold client-side content that is not initially rendered
 - Can be rendered later when bound to data
- Angular uses its own version of the template element (`ng-template`)
 - Converts the `*` syntax to `ng-template` behind the scenes

```
<div *ngIf="errorMessage">  
    Sorry: {{errorMessage}}  
</div>
```

Sorry: Response with status: 0 for GET
Sorry: Response with status: 0 for UPD

Both code samples
have same behavior

```
<ng-template [ngIf]="errorMessage">  
    <div>  
        Sorry: {{errorMessage}}  
    </div>  
</ng-template>
```

```
<!--template bindings={  
    "ng-reflect-ng-if": "  
 }-->  
► <div>...</div>  
<!--template bindings={  
    "ng-reflect-ng-if": "  
 }-->  
► <div>...</div>
```

Converted to an attribute

Creating a Structural Directive

- Create a class and mark it with the `@Directive()` decorator
 - Easiest to use CLI: `ng g directive unless`
- Define a selector to refer to the directive inside templates
 - Prefix the selector with a prefix to ensure uniqueness
 - Do not use prefix `ng`, as that is reserved for Angular
 - Use CSS attribute syntax `[]`
 - Directives are added as attributes inside templates

```
@Directive({  
    selector: '[appUnless]'  
})  
export class UnlessDirective { }
```

- Add the directive to module declarations
 - CLI will add to the `AppModule`

Binding Values to the Directive

- Structural directives normally pass data via the attribute itself
 - `*ngIf="someCondition"`
 - `*ngFor="let country of countries;"`
- Add an `@Input()` to the directive
 - Assign to a write-only setter
- Typescript allows methods to appear to be properties via `get` and `set`
 - `get` method for reading
 - `set` method for writing
- From the point of view of the client, looks like data
 - Inside the class, is just a pair of methods
 - Omit `get` for write-only, `set` for read-only

TypeScript Getters and Setters

- Getters and Setters normally wrap data
 - Allow validation or manipulation before setting/getting value

```
private _data: string = null;  
get fieldName(): string {  
    return this._data;  
}  
set fieldName(name: string) {  
    this._data = name;  
}
```

Value passed in when client sets
value of `objectRef.fieldName`

- Can be methods without backing fields, that can be set like fields
 - Or as attributes inside a template

```
@Input() set appUnless(condition: boolean) {  
    // do something with condition here  
}
```

Boolean assigned to attribute `*appUnless`
will be passed in to the method

Rendering the Template

- Structural directives rely on two types to do their work
 - `TemplateRef` to reference the template to be rendered
 - `ViewContainer` to do the rendering
- Both are injected in the constructor

```
constructor(  
    private tRef: TemplateRef<any>,  
    private viewContainer: ViewContainerRef  
) { }
```

- `ViewContainer` has methods to manipulate DOM
 - `clear()`
 - `insert()`
 - `createEmbeddedView()`
 - Etc.
- `TemplateRef` allows it access to the markup to which it is assigned

Structural Directives Illustrated

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private tRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set appUnless(condition: boolean) {
    if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    } else if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.tRef);
      this.hasView = true;
    }
  }
}
```

Use this Boolean so the directive only does work if work is needed since creating and destroying view elements takes time

If the item is not required, do not hide it, remove it from the DOM completely

Use ViewContainer to access DOM, use TemplateRef to access the host element

Using the Directive

This value will be passed into the `set` method

```
<div *appUnless="(name.valid || name.pristine)" class="error">  
    Country must have a name  
</div>  
  
<div *ngIf="!(language.valid || language.pristine)" class="error">  
    Country must have a language  
</div>
```

AppUnless is used in exactly the same way
as NgIf but has reversed logic. Note the !

ng-container

- Usually attach structural directives to a convenient HTML element
 - If there isn't one, add `span` or `div`
- Sometimes adding an element changes the behavior
 - May change appearance
 - CSS selectors no longer match or match when they should not
 - May change code that parses the DOM
 - Some elements require children to be of a single type
 - All children of `<select>` elements must be `<option>`
- Angular provides `<ng-container>` as a grouping element
 - Does not appear in the finished DOM
 - Allows structural directives to be associated with a group of HTML elements

Object Tracking

- By default, NgFor uses object identity to track objects
 - Can be inefficient, especially with objects retrieved from web services
 - Can lead to DOM tear down and rebuild even when data unchanged
- Provides the `trackBy` option for greater efficiency
 - Developer specifies a function that returns a value used for tracking
 - Function receives index and object as arguments

```
<tr *ngFor="let country of countries; trackBy: trackCountry">
  <td>{{country.name}}</td>
  <td>{{country.language}}</td>
</tr>
```

```
trackCountry(i: number, country: any): number {
  return country.id;
}
```

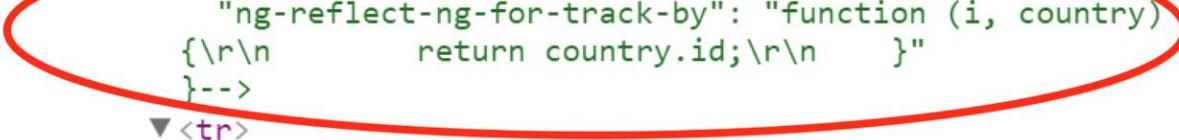
Object Tracking Illustrated

Countries

Country	Language
United Kingdom index = 0	English
United States of America index = 1	English
Australia index = 2	English

Elements Console Sources Network Timeline > | :: X

```
ng-reflect-ng-for-of": "[object Object],[object Object],[object Object],[object Object],[object Object]",  
ng-reflect-ng-for-track-by": "function (i, country)  
{\r\n    return country.id;\r\n}  
-->  
▼ <tr>
```



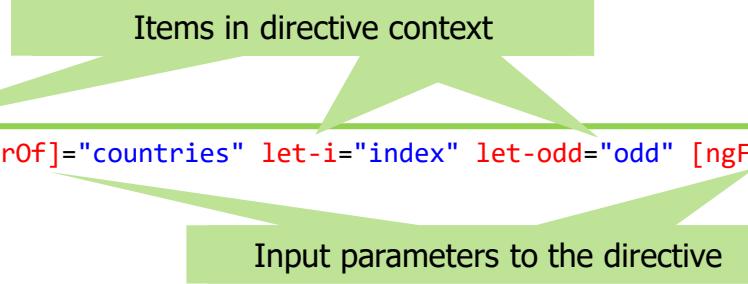
Advanced Structural Directives

- The string assigned to the attribute is treated as a microsyntax

```
<div *ngFor="let country of countries; let i=index; let odd=odd; trackBy: trackById" [class.odd]="odd">  
    ({{i}}) {{country.name}}  
</div>
```

- Becomes:

```
<ng-template ngFor let-country [ngForOf]="countries" let-i="index" let-odd="odd" [ngForTrackBy]="trackById">  
    <div [class.odd]="odd">  
        ({{i}}) {{country.name}}  
    </div>  
</ng-template>
```



- The NgFor directive must support two input parameters (marked with @Input)
- What is the context?

Structural Directive Context

- Each time the directive adds something to the DOM, it attaches data
 - Here's (more or less) how NgFor does it
- Overloaded `createEmbeddedView` accepts context as a parameter
- This is the definition of the context
 - Items in the context can be accessed through let expressions in the string assigned to the property
 - In the previous example:
 - `country` is assigned `$implicit`
 - `i` is assigned `index`
 - `odd` is assigned `odd()`

```
const view = this.viewContainer.createEmbeddedView(this.tRef,  
    new NgForOfContext<T>(item, this.ngForOf, i, ilen), i);
```

```
export class NgForOfContext<T> {  
    constructor(  
        public $implicit: T, public ngForOf: NgIterable<T>,  
        public index: number, public count: number) { }  
  
    get first(): boolean { return this.index === 0; }  
  
    get last(): boolean {  
        return this.index === this.count - 1;  
    }  
  
    get even(): boolean { return this.index % 2 === 0; }  
  
    get odd(): boolean { return !this.even; }  
}
```

Appendix Concepts

Introducing Directives

Structural Directives

Attribute Directives

Appendix Summary

Attribute Directives

- Closely related to structural directives
 - Use the `@Directive()` decorator
 - Specify selector using CSS syntax
- Do not use HTML5 template element
- Do not add or remove elements from the DOM
- Add or change behavior of DOM element
 - Use `elementRef`, not `templateRef`
 - Use `@HostBinding` to access specific properties of the element
- Use `@HostListener` to attach to DOM events on underlying element
 - `@Output()` and `EventEmitter` to communicate with element
 - `@Input` to get properties from the element

The `@HostListener()` Decorator

- Allows directives to listen for DOM events on the element
 - Reduces amount of code required
 - Manages unsubscription
 - Preventing memory leaks
 - Avoids code having to interact directly with the DOM
 - DOM interaction should be through Angular wherever possible
- Has two arguments:
 - Name of event being listened for
 - Array of string arguments
 - Can use reserved `$event` symbol to reference DOM event object

```
@HostListener('mouseover', ['$event']) onMouseOver(e: MouseEvent) {  
    console.log('Mouse position: x=' + e.clientX + ', y=' + e.clientY);  
}
```

e = MouseEvent = \$event

The @HostBinding() Decorator

- Allows directives to update properties of the host element
 - Reduces amount of code required
 - Avoids code having to interact directly with the DOM
 - DOM interaction should be through Angular wherever possible
- Has one argument:
 - Name of property being bound

```
@HostBinding('style.border') border: string;
```

Creating Attribute Directives

Steps to create an attribute directive

1. Start with Angular CLI: `ng g directive Box`
2. Creates class decorated with `@Directive()` and assigned a selector

```
@Directive({
  selector: '[appBox]'
})
export class BoxDirective {
```

3. Either inject `ElementRef` in the constructor and set properties on the element

```
constructor(private el: ElementRef) { }
```

```
this.el.nativeElement.style.border = '2px solid red';
```

4. Or use `@HostBinding()`

```
@HostBinding('style.border') border: string;
```

```
this.border = '2px solid red';
```

Creating Attribute Directives (continued)

5. Attach to events with @HostListener

```
@HostListener('mouseenter') onMouseEnter() {  
    this.el.nativeElement.style.border = '2px solid red';  
}
```

6. Optionally, define and emit events

```
@Output() activated: EventEmitter<MouseEvent> = new EventEmitter();  
  
@HostListener('mouseover', ['$event']) onMouseOver(e: MouseEvent) {  
    this.activated.emit(e);  
}
```

7. Optionally, define @Input() properties

Naming, or aliasing, to the same name as the directive means it can be assigned like this

```
@Input('appBox') boxColor: string;  
@Input() boxWidth: number;
```

Remember, [property]= causes code to be evaluated, property= assigns a value

```
<p [appBox]="color" boxWidth="5">box me in!</p>
```

Testing an Attribute Directive

- Either construct target element programmatically
 - Test as plain class
 - Hard if it refers to the host element
- Or create dummy component
 - Test as directive

Could also use `SpyOn` here to check behavior (e.g., event triggering)

```
@Component({
  template: '<div appBox>aaa</div>'
})
class TestBoxComponent {
  @ViewChild(BoxDirective) directive: BoxDirective;
}

describe('BoxDirective', () => {
  let component: TestBoxComponent;
  let fixture: ComponentFixture<TestBoxComponent>;
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestBoxComponent, BoxDirective]
    });
    fixture = TestBed.createComponent(TestBoxComponent);
    component = fixture.componentInstance;
  });
  it('should add box on mouseenter', () => {
    const divEl = fixture.debugElement.query(By.css('div'));
    divEl.triggerEventHandler('mouseenter',
      new MouseEvent('mouseenter'));
    fixture.detectChanges();
    expect(divEl.nativeElement.style.borderWidth).toBe('2px');
  });
});
```

Define a minimal template

Use `@ViewChild` to get a reference to the directive



HANDS-ON
EXERCISE

30 min

Exercise A.1: Creating an Attribute Directive

- In this exercise, you will create an attribute directive that enables dragging over and dropping onto an HTML element
- Please refer to the Exercise Manual

Appendix Concepts

Introducing Directives

Structural Directives

Attribute Directives

Appendix Summary

Appendix Summary

In this appendix, we have explored:

- Creating directives
- Learning how to build structural directives
- Creating an attribute directive

Fidelity LEAP

Technology Immersion Program

Developing Client-Side Dynamic Web Applications

Appendix B: Observables

Appendix Overview

In this appendix, we will explore:

- Understanding Observables
- Returning Observables from the `Http` service
- Using Observables to build a Functional Reactive Form

Appendix Concepts

Observables and REST

Observables and Forms

Appendix Summary

Introducing Observables

- Observables are used extensively in Angular
 - HttpClient service methods all return Observables
 - The events system uses Observables
 - Routing parameters are provided by Observables
 - Even forms can be Observables
- So, what are they?
- Observables are proposed as a new primitive type in ES7
 - Currently provided via the RxJS library
- Provide a new way of managing asynchronous data
 - Distinct from promises
- Promises are single operations fulfilled at some point in the future
- Observables are sequences of operations over time
 - Cancellable streams of data

Understanding Observables

- Traditional JavaScript function calls are synchronous
 - Responses are *pulled* from the function on request
 - Return immediately
- Where multiple returns are required, an iterator is used
 - Responses are pulled in sequence
- A Promise is an asynchronous single
 - It *pushes* the return when it is ready
 - Client does not control the timing of the return
- An Observable is an asynchronous stream
 - Pushes multiple responses when they are ready
- Clients do not control the timing of returns
 - Subscribe to the stream
- Observables are only triggered when a client subscribes
 - A *cold* Observable has been defined, but not subscribed
 - A *hot* Observable has a subscriber

The Observable Advantage

- Traditionally, web technologies viewed UIs in terms of discrete actions
 - Button click
 - Mouse click
 - Focus
- However, many aspects of UIs are more stream-like
 - Entering sequences of text
 - Moving a mouse
 - Auto-suggesting as a user types
- Observables mean UI can be treated as a combination of streams
 - Significant reduction in code required for many tasks
 - Enhanced behavior
 - Unlike Promises, Observables can be cancelled and retried
 - Many methods to handle complex coding concerns
 - Debouncing to avoid sending too many HTTP requests
 - distinctUntilChanged to avoid repeating an HTTP query
 - Etc.

Observables in Action

- Observables are typed to the objects returned in the stream
 - E.g., HttpClient methods all return Observable<T>
- Have sub-types with specialized behavior
 - Subject
 - BehaviorSubject
- Have many methods to manipulate the return
 - map()
 - flatMap()
 - switchMap()
- General pattern is consistent
 - subscribe() to the Observable to access stream
 - unsubscribe() to avoid memory leaks
 - Not absolutely necessary with a finite series
 - May be managed by framework tools such as AsyncPipe
 - More on AsyncPipe soon

Route Parameters Revisited

- Route parameters are provided as an Observable
 - Because they are values that change over time
- Current solution uses a snapshot
 - Retrieves the initial value of the parameter
 - Does not allow for changes over time
 - Snapshot approach relies on a new component being created each time
 - Works well for standard router-outlet
 - Does not work with child routes
 - Component is no longer recreated each time
 - Only first link would work
- Alternative is to *subscribe* to the parameters Observable
 - Works for both standard *and* child routes

Subscribing to Parameters

Note additional lifecycle method

```
export class CountryComponent implements OnInit, OnDestroy {  
  countryId: number;  
  subscription: any;  
  constructor(private route: ActivatedRoute) { }  
  
  ngOnInit() {  
    this.subscription = this.route.params.subscribe(params => {  
      this.countryId = params['id'];  
    });  
  }  
  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
}
```

Resets `countryId` whenever route parameter changes

Normally, it is important to unsubscribe from Observables. Here, however, this is not strictly necessary, as the `RouterModule` will destroy the `ActivatedRoute` along with the component.

Observables and HttpClient

- The HttpClient service returns Observable<T>
 - A finite Observable/single value series, completed when the HttpClient call returns
- Either subscribe in the component, as we have done until now

```
countries: Country[];  
getCountries() {  
    this.service.getCountry().subscribe(countries => this.countries = countries);  
}
```

- Or use an Observable directly with the AsyncPipe
 - Inside the component, use an Observable type

```
countries: Observable<Country[]>;  
getCountry() {  
    this.countries = this.service.getCountry();  
}
```

- Inside the template, modify NgFor to use AsyncPipe

```
<tr *ngFor="let country of countries | async">
```

The AsyncPipe

- Is another built-in pipe
- Subscribes to an Observable or Promise
- Automatically unsubscribes when component is destroyed
- Unwraps the primitive type emitted from Promise or Observable
- Makes it much easier to work with Observables
 - Simple to add to the template
 - Manages sub/unsub and ensures no memory leaks
 - Can be used with single values or iterable results
- Frequently used inside NgFor

```
<tr *ngFor="let book of books | async; trackBy:trackBook">
```

Taking Advantage of Streams

- Observables make it easy to update the page responsively as user types
- Example: displaying list of matching books as user types in search box

1. Return Observable<Book[]> from the service

```
return this.http.get<Book[]>(this.url + "Books");
```

2. Define a stream to hold user-entered text

```
searchStream: BehaviorSubject<string> = new BehaviorSubject<string>("");
```

- Could be a raw Observable or one of the inherited types
- BehaviorSubject must always return a value on subscription
 - Can facilitate component initialization

3. Call the next() method when the user enters a character

```
search(term: string) {  
    this.searchStream.next(term);  
}
```

search() is bound to a
key up event on an input

Taking Advantage of Streams (continued)

4. Call the http service every time the stream emits a value
 - Inside an Observable mapping function
 - Assign the resulting Observable to a local variable

```
books: Observable<Book[]> = this.searchStream  
  .pipe( switchMap((term: string) =>  
    this.bookService.getBooksByTitle(term)));
```

books updates every time a character
is added to searchStream

pipe() composes Observable operators. The operators
themselves are separately imported from 'rxjs/operators'.
Pass in a list of operators as parameters to pipe().

switchMap() returns a new Observable and
ensures that the results are ordered by the
requests, not the responses from the server

5. Add the AsyncPipe in the template to manage sub/unsub

```
<tr *ngFor="let book of books | async; trackBy:trackBook">
```

Taking Advantage of Streams (continued)

6. Refine the behavior by chaining methods to....

- Delay server requests until n milliseconds after users stops typing

```
.debounceTime(500)
```

- Ensure duplicate requests are not sent if user hits backspace

```
.distinctUntilChanged()
```

- Handle errors

```
.catch(error => this.errorMessage = error)
```

Assumes the service uses `Observable.throw()` to build
errorMessage in the same way as `Promise.reject()`



40 min

Exercise B.1: Making RESTful Calls Using Observables

- In this exercise, you will use Observables to make RESTful web service calls
- Please refer to the Exercise Manual

Appendix Concepts

Observables and REST

Observables and Forms

Appendix Summary

Functional Reactive Programming

- Angular was designed with Functional Reactive Programming in mind
 - Two key concepts:
 - Asynchronous dataflow/streams (the Reactive part)
 - A model for the UI
 - Functions that do not mutate state (the Functional part)
 - Output depends solely on input
- The last exercise treated a textbox as a stream
 - Responded with a function that returned an output based on the input
- In Angular, *the entire form* can be treated as a stream
 - Allowing a model-driven form to become fully reactive
- Can be responded to in functions where output depends solely on input
 - And where state is stored not in the application code, but the DOM

Functional Reactive Forms

- A Functional Reactive form can be viewed as a stream of changing values
 - Exposed in Angular as the `valueChanges` stream
- The stream gives access to the value of the entire form as an object
 - Values of individual controls are properties of the object
- Can be used for validation or real-time data manipulation as the user types

The screenshot shows a browser window with a "Books" application. The application has a title input field containing "This is the", an author input field, and a "Add Book" button. Below the form, the browser's developer tools are open, specifically the "Console" tab. The console displays a series of log entries showing the state of the form's value as it changes over time. The first entry shows the value as an object with "title" and "author" properties. Subsequent entries show the "title" property being updated to "Th", "Thi", "This", "This ", and finally "This i". The "author" property remains empty throughout. The log entries are timestamped at 82f7:46. A green callout box points to the first log entry with the text "Control values are properties on the object". Another green callout box points to the log entries with the text "The form is received as a stream as the user types".

```
Angular 2 is running in the development mode. Call enableProdMode() to enable the production mode.  
Form value = ▶ Object {title: "T", author: ""} book-form  
Form value = ▶ Object {title: "Th", author: ""} book-form.component.ts?82f7:46  
Form value = ▶ Object {title: "Thi", author: ""} book-form.component.ts?82f7:46  
Form value = ▶ Object {title: "This", author: ""} book-form.component.ts?82f7:46  
Form value = ▶ Object {title: "This ", author: ""} book-form.component.ts?82f7:46  
Form value = ▶ Object {title: "This i", author: ""} book-form.component.ts?82f7:46
```

Steps to Use valueChanges Stream

1. Import OnInit and OnDestroy from @angular/core

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

2. Implement both lifecycle methods

```
class CountryFormComponent implements OnInit, OnDestroy {
```

3. Create a variable to hold the subscription

```
sub: Subscription;
```

4. subscribe() to the stream inside ngOnInit()

```
this.sub = this.countryForm.valueChanges.subscribe((value)  
  => { /* do something in here */ });
```

5. Implement ngOnDestroy() and unsubscribe() from the stream

```
ngOnDestroy() { this.sub.unsubscribe(); }
```

Updating the DOM

- The DOM can be updated via the `FormGroup`
 - Or individual `FormControl` objects
- Same two methods on both `FormGroup` and `FormControl`
 - `setValue()` sets form control values to matching object literal
 - Any missing `FormControl` or value raises error
 - `patchValue()` allows a subset of controls to be updated
 - On `FormControl`, makes no difference which method is used
- Second argument uses object literal to control behavior
 - `onlySelf: false` by default, prevents validation check bubbling
 - `emitEvent: true` by default, emits `valueChanges` event on change

Only update specified control(s)

```
this.countryForm.patchValue({language: value.language},  
{ emitEvent: false});
```

Don't emit `valueChanges` for this update



20 min

Exercise B.2: Functional Reactive Forms and Observable

- In this exercise, you will use modify your existing book form to take advantage of streams and Observables
- Please refer to the Exercise Manual

Appendix Concepts

Observables and REST

Observables and Forms

Appendix Summary

Appendix Summary

In this appendix, we have explored:

- Understanding Observables
- Returning Observables from the `Http` service
- Using Observables to build a Functional Reactive Form