Angular

Getting Started

New Horizons®

Introduction

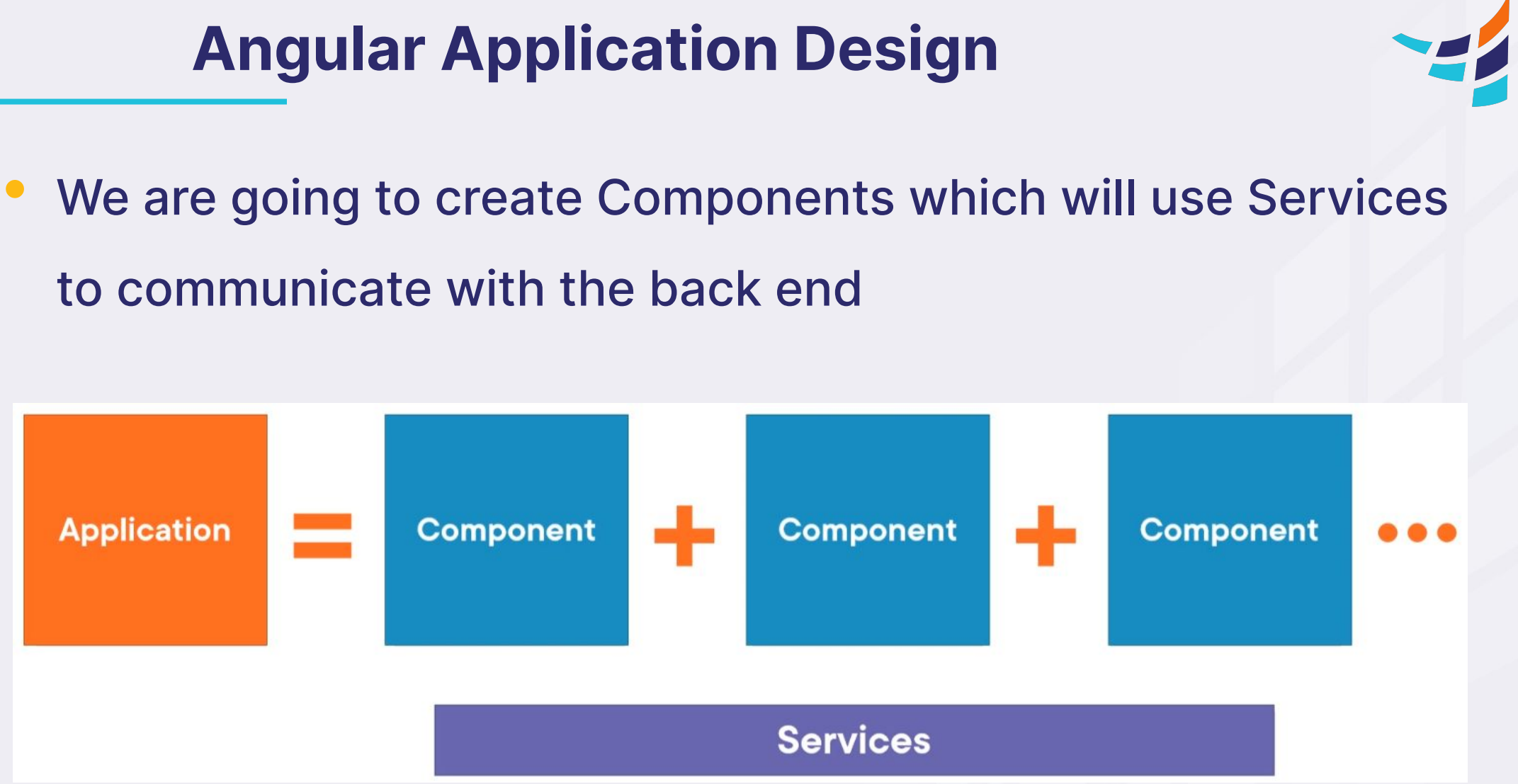
- We are going to be building client-side applications using the Angular Framework
- Prerequisites: HTML, CSS and TypeScript

Introduction

- We are going to create:

  - Expressive HTML with dynamic content
  - Powerful data binding
  - Modular design
  - Using built-in back-end integration to communicate with a server

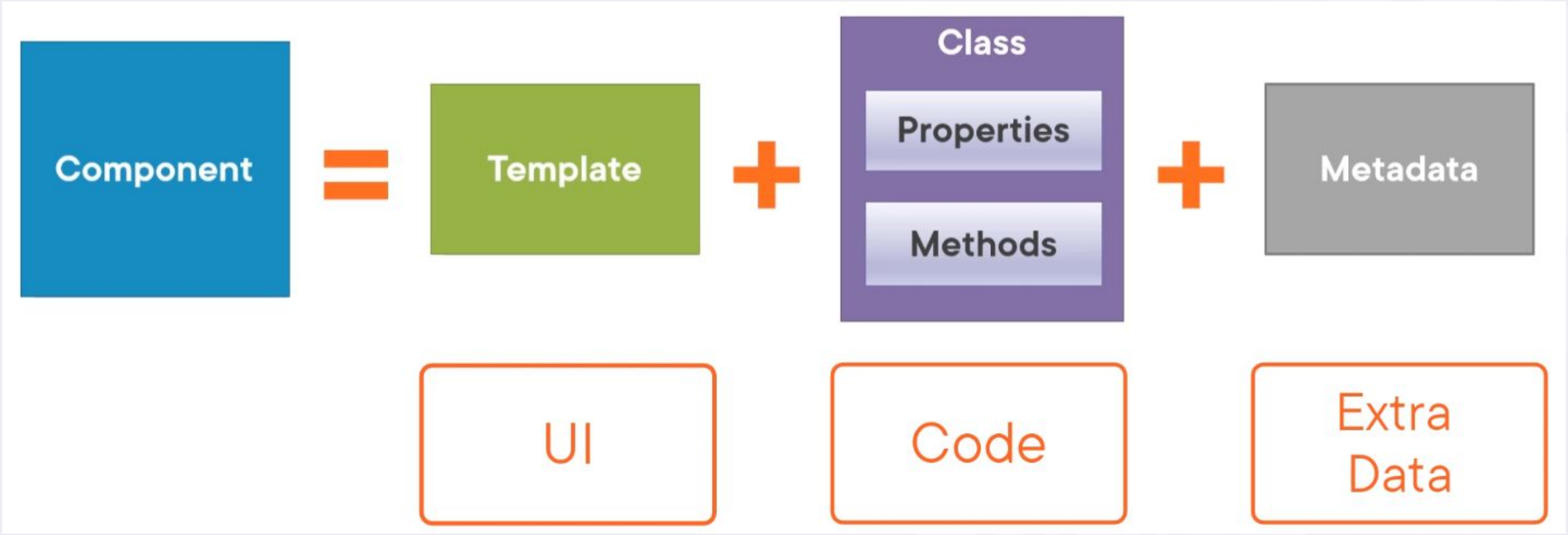
Angular Application Design



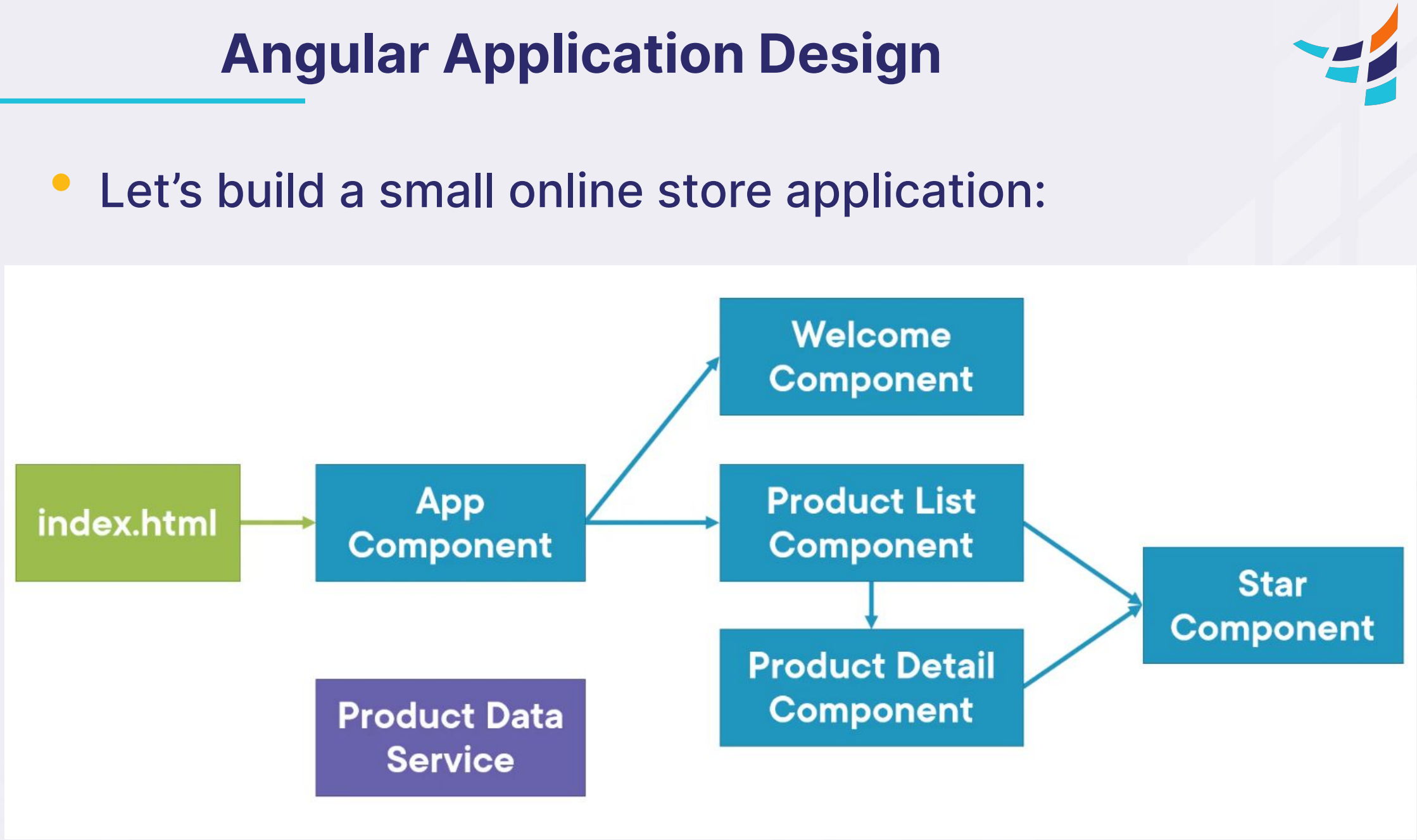
- We are going to create Components which will use Services to communicate with the back end

Angular Application Design

- Components will be the main building blocks



Angular Application Design



- Let’s build a small online store application:

Outline

- Setup and environment
- Components Introduction
- Templates, Interpolation & Directives
- Data Binding & Pipes
- Building Nested Components & Extra Techniques
- Services & Dependency Injection - retrieving data from HTTP
- Navigation & Routing
- Angular Modules
- Building, Testing and Deploying with the CLI

Toolbox

- Visual Studio (VS) Code:

  https://code.visualstudio.com/download

- Node Package Manager (npm):

  https://nodejs.org/en/download

Starter Project

- Unzip the APM-Start.zip in a workspace folder

  The file will be provided by your instructor

- Open the folder in VS Code and explore the contents

- In your terminal/cmd run: npm install

Starter Project

- package.json contains a list of all used packages
- Main code is situated in folder: src/app
- The command to start the project is listed in the package.json file as: npm start
- Browser should open automatically - go and make a change in the app.component.html file

Starter Project

- You can also create a project from scratch using the Angular CLI tool (Command Line Interface)
- Install the CLI tool globally by running the command:

  npm install -g @angular/cli
- Navigate to a new folder of your choice and run:

  ng new apm-new

Angular Components

- We already have the index.html file and the root App component created for us in the starter project (or via the CLI tool)
- The component has its HTML in a template called app.component.html
- The code supporting the template is created with TypeScript and placed in app.component.ts

Angular Components

- The component is a class which has some metadata defined with a Decorator
- The decorator is a function that adds metadata to a class, its members or its methods
- Let’s explore in detail the App component defined the in the app.component.ts file

Angular Components

- At the bottom we have the class defined and exported so it can be used elsewhere in the application
- Common naming convention is to use the name of the component followed by the Component word
- That name is going to be used in other places in the code as long as we export it

Angular Components

- Within the body of the class we can define properties and methods
- In this App component we only have one property and no methods
- Note the name of the property followed by its type in TS before assigning a value

Angular Components

- This class alone is not enough to create a component
- We need to associate the class with a template and provide additional metadata that describes it
- We provide this data via the Angular's Component function that we use as a decorator (@ up front)

Angular Components

- The decorator is a TS feature implemented in JS
- We attach the decorator **before** the piece of code that it decorates or describes and it only works for it
- There is no closing ; at the end of the decorator
- We pass an object as parameter to the decorator function

Angular Components

- The decorator's object has a selector property which defines a directive name to be used in HTML to render this component
- In other words, the selector is the name of the custom HTML element that we are creating in order to access and render the component
- The template property points to the HTML content

Angular Components

- In order to use the Component function or any other Angular core feature, we need to import it first:

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

- You can import multiple members of the same library by enlisting them separated by commas

Single Page Applications

- Now that we are familiar with the App component, check the index.html file and see what happens in the body
- We are using the directive declared in the selector property of the decorator to render the template of the App component via the custom element pm-root

App Modules

- Wait, but how does Angular know where to look for the pm-root and its corresponding component?
- The Angular application is divided into Modules
- Each Module imports and lists every component and library that it uses
- Modules help us organize the code better and they also provide template resolution environment - when the compiler sees a directive, it goes and finds where it belongs inside of the module

App Modules

- The module also defines which is the bootstrap component - what is loaded initially
- It also takes care of running the application correctly in a browser by importing the external BrowserModule library (assuming we develop web application)
- Check the app.module.ts file for the module definition

App Modules

- We define the module as a class with the NgModule decorator
- The import property of the decorator lists all external and third-party libraries that the application will use
- The declarations property lists all main components used in this module
- The bootstrap property defines the starting component

Templates

- We can define the template in several different ways:

Inline Template

```
template:
"<h1>{{pageTitle}}</h1>"
```

Inline Template

```
template: `
<div>
 <h1>{{pageTitle}}</h1>
 <div>
     My First Component
 </div>
</div>
`
```

ES 2015 Back Ticks

Linked Template

```
templateUrl:
'./product-list.component.html'
```

Templates

- Keeping it in the same file can provide more concise code but you get no additional benefits like IntelliSense
- Linked Templates are the preferred way to go especially for longer HTML content

Templates

- Let’s build a Product List component for our application

- To make things easier for the UI, let’s use:

  - Bootstrap: https://getbootstrap.com/

  - Font Awesome: https://fontawesome.com/

- We can easily install these via npm by typing in cmd:

  npm install bootstrap font-awesome

Templates

- We would like to make this new libraries accessible to every template in our application
- Import them up top into our main CSS file at src/styles.css:

```
@import “~bootstrap/dist/css/bootstrap.min.css”;
@import “~font-awesome/css/font-awesome.min.css”;
```

Templates

- We can now create a new template for our Product List
- This is a separate logic in our application so let’s first create new folder in app called: products
- We can now create the template following the naming conventions that we already saw: product-list.component.html

Templates

- Build a product list template using Bootstrap classes
- Include a header and a filter with input field which will be used by the user to search in the product table
- Create empty table with columns for image, product, code, available, price and 5-star rating

Displaying the Template

- The next step is to build a component for this template
- Build a file called: product-list.component.ts
- Add the selector as directive in our App component
- In order to make the directive work (error in browser right now) you need to modify your AppModule

Check next slide for details

Displaying the Template

- Note: every Angular application needs to have at least one module - the root module
- Every component must belong to one and only one Angular module
- If you include a component into a module then you can use it inside other components from this module
- If a component belong to another module, you can instead import it in the component that you want to use it in

Data Binding

- Coordinates communication between the component's class and its template by passing data and invoking methods
- Interpolation is one-way binding from the class property to the template
- In order to use a variable from the class in the template you use double curly braces: {{ pageTitle }}

Data Binding

- The interpolation method allows you to execute any JS expression within the bracers:

  {{ ‘Title: ‘ + pageTitle}}

  {{ 2*5 + 10*10 }}

- You can also use it to bind data to element’s properties:

  <h1 innerText={{pageTitle}}></h1>

Interpolation

- Add page title property to your ProductList component:

  pageTitle: string = ‘Product List’;

- Display that property on the top of the template as a header using interpolation

Data Binding

- The interpolation method allows you to execute any JS expression within the bracers:

  {{ ‘Title: ‘ + pageTitle}}

  {{ 2*5 + 10*10 }}

- You can also use it to bind data to element’s properties:

  <h1 innerText={{pageTitle}}></h1>

Angular Built-in Directives

- Angular provides predefined directives that we can use to manipulate the content of a template
- There are structural directives that let us display content conditionally or based on data:
  - *ngIf: provides if logic in the template
  - *ngFor: provides loop logic in the template

Angular Built-in Directives

- Let’s display our product table only if we have available products in the store
- First we define a new property in our component’s class:

  `products: any[] = [];`
- `any` means any data type since we don’t know yet what exactly our products are going to look like

Angular Built-in Directives

- Now let's modify the template and display the table only if that product array is not empty:
  `<table class='table' *ngIf='products.length'>`
- Check that the table is gone in the browser
- The products are usually going to be accessed and populated via API but for now let's go and hardcode a couple until we learn how to work with APIs

Your instructor can provide sample JSON product data

Angular Built-in Directives

- Now we can populate our table body by creating rows using the *ngFor directive with the product array:

```
<tr *ngFor='let product of products'>
    <td></td>
    <td>{{ product.productName }}</td>
    <td>{{ product.productCode }}</td>
    <td>{{ product.releaseDate }}</td>
    <td>{{ product.price }}</td>
    <td>{{ product.starRating }}</td>
</tr>
```

Quick JS Reminder

for...of vs for...in

for...of

- Iterates over iterable objects, such as an array.
- Result: di, boo, punkeye

```
let nicknames= ['di', 'boo', 'punkeye'];

for (let nickname of nicknames) {
  console.log(nickname);
}
```

for...in

- Iterates over the properties of an object.
- Result: 0, 1, 2

```
let nicknames= ['di', 'boo', 'punkeye'];

for (let nickname in nicknames) {
  console.log(nickname);
}
```

Angular Built-in Directives

- Let’s display our product table only if we have available products in the store
- First we define a new property in our component’s class:

  `products: any[] = [];`
- `any` means any data type since we don’t know yet what exactly our products are going to look like

Property Binding

- It allows us to set the property of an element to a template expression
- We can display the images source path using this approach, note the syntax:

```
<img [src]='product.imageUrl' />
```

Property Binding

- For comparison, interpolation could be used here:

  `<img src={{product.imageUrl}} />`

- However, property binding evaluates the expression while interpolation always assigns a string

- Property binding is preferred as it allows you to do:

  `<input type='text' [disabled]='isDisabled' />`

Property Binding

- Once you use property binding, everything in the quotes after the = sign is considered JS and not a string
- We can now display our images in the product table:

  ```
  <img [src]='product.imageUrl'
  [title]='product.productName' />
  ```

- Note: if you are missing the assets folder get it from your instructor

Property Binding

- In order to make the images smaller we can define two more properties in our component's class:

```
imageWidth: number = 50;
imageMargin: number = 2;
```

- You can now bind them to the img element like this:

```
[style.width.px]='imageWidth'
[style.margin.px]='imageMargin'
```

Event Binding

- It allows us to connect an event to a method in the component listening for JS events
- Let’s capture the clicks on the “Show Image” button and connect it to our component so it is fully functional

Event Binding

- We can add new property to track the current state:

```
showImage: boolean = false; //let's hide by default
```

- We can then add method at the bottom that will simple change the value of this property:

```
toggleImage(): void {
    this.showImage = !this.showImage;
}
```

Event Binding

- In the template, let’s bind and call that new method once the button is clicked. Note the syntax:

  ```
  <button class='btn btn-primary' (click)="toggleImage()">
  ```

- Again, everything after the = sign here is considered JS
- The parenthesis annotate binding to an event
- Event names are usually the JS names of the events

Event Binding

- The button now invokes the class method

- In order to see the results though, we need to specify in the img element additional *ngIf directive:

```
<img *ngIf='showImage' ... />
```

Event Binding

- We can also use interpolation to change the text of the button itself so it doesn't always say "Show Image":
  {{ showImage ? 'Hide' : 'Show'}} Image
- Angular's change detection automatically updates everything once the value of showImage changes

Two-way Binding

- If we want to connect the filter input in our template, we are going to need to connect it to a variable in the class
- Changes to the variable will be updated as value to the input
- However, changes to the input also need to update the variable in the class
- This is called two-way binding

Two-way Binding

- The syntax for that in Angular is combination of everything we learned so far
- Let’s define a variable in the class to hold the input value:

  listFilter: string = ‘cart’; //should be empty by default but let’s add a value for now just to see the binding working

Two-way Binding

- The two-way binding now looks like this:

  `<input type='text' [(ngModel)]="listFilter" />`

- ngModel is a directive indicating two-way binding

- The square brackets indicate property binding from the class property to the input value

- The parenthesis indicate event binding that will send notification back to the class upon changes to the value

Two-way Binding

- Why is it not working?
- Most of the directives that we used so far were from the BrowserModule that we already imported in AppModule
- The ngModel directive is part of a different module called FormsModule so we need to have that available

Two-way Binding

- Go to app.module.ts
- Add an import:

  import { FormsModule } from '@angular/forms';
- Don’t forget to add the new import in the imports section within the decorator’s function

Two-way Binding

- To visualize the two-way binding, let's go and add interpolation to the same property next to the "Filtered by" text like this:
  `<h4>Filtered by: {{listFilter}}</h4>`
- Now you can see that typing in the input changes the class property which then updates on the front end
- We will add filter logic that works in just a few slides

Transforming Data with Pipes

- They transform **bound** properties before they are displayed
- Built-in pipes are available for dates, numbers, percents, currencies, json, etc.
- We can also build our own custom pipes

Transforming Data with Pipes

- We add pipes to a bound value by using the | sign
- For example, we can transform the product codes to be all in lowercase letters like this:

  {{ product.productCode | lowercase }}
- Note: lowercase is one of the predefined pipes in Angular

Transforming Data with Pipes

- We can also use pipes in property bindings

- The image that we display in the table could have its title product name all in uppercase like this:

```
<img *ngIf='showImage'
    ...
    [title]='product.productName | uppercase'
    ...
```

Transforming Data with Pipes

- We can chain pipes:

  {{ product.price | currency | lowercase }}

- By default the currency pipe displays the price in the local currency and if there are letters involved they are all capital so the second lowercase pipe will transform them into small ones

Transforming Data with Pipes

- Some pipes support input parameters:

  {{ product.price | currency:'USD':'symbol':'1.2-2' }}

- We add the parameters to the pipe by specifying a colon after the pipe and enumerating all the parameters if there are more than one

Transforming Data with Pipes

- The currency pipe supports 3 parameters
- First one is currency code, second one is how to display the currency - symbol or letters
- The third one is rounding determining that we should see a minimum of 1 digit before the decimal point, minimum of 2 digits after it and maximum of two digits after it

Interfaces

- An interface is a specification identifying a related set of properties and methods
- Properties are the data required for the class
- Methods contain the logic to work with the data for the given class

Interfaces

- There are two way to use interfaces
- We can use them as types or if you'd like - a custom made class
- We can use them as feature sets which define a certain behaviour and leave the implementation to the component that implements that interface

Interfaces

```
export interface IProduct {
  productId: number;
  productName: string;
  productCode: string;
  releaseDate: string;
  price: number;
  description: string;
  starRating: number;
  imageUrl: string;
}
```

As a type

```
products: IProduct[] = [];
```

```
export interface DoTiming {
    count: number;
    start(index: number): void;
    stop(): void;
}
```

As a feature set

```
export class myComponent
             implements DoTiming {
  count: number = 0;
  start(index: number): void {
    ...
  }
  stop(): void {
    ...
  }
}
```

Interfaces

- Let’s create an interface which will define what a Product means in our application
- We can create the interface in the products folder in a file called product.ts
- We need to export it so it could be used elsewhere

Interfaces

- In order to see why the interfaces are useful - go and make a typo in one of the property values in our products array containing the display data
- We get no error messages until we actually render the data on the screen and see the missing pieces there
- These bugs are usually very hard to find

Interfaces

- Let’s now replace the any[] array to be of time IProduct[]
- Note that the IntelliSense catches the error right away
- The I in the name stands for Interface but many developers choose to omit it and name it just Product instead - both approaches are fine

Encapsulating Component Style

- Templates often require unique styles
- We can inline them directly into the HTML but that makes it hard to change and maintain
- We can build external stylesheet and link it inside the index.html file but that makes the component harder to reuse

Encapsulating Component Style

- The component decorator allows us to add styles directly by just adding a property which is an array in which we can add multiple rules:

  styles: [‘thead {color: #337AB7;}’]

- Even better is to add separate stylesheets again as an array to the styleUrls property:

  styleUrls: [‘./product-list.component.css’]

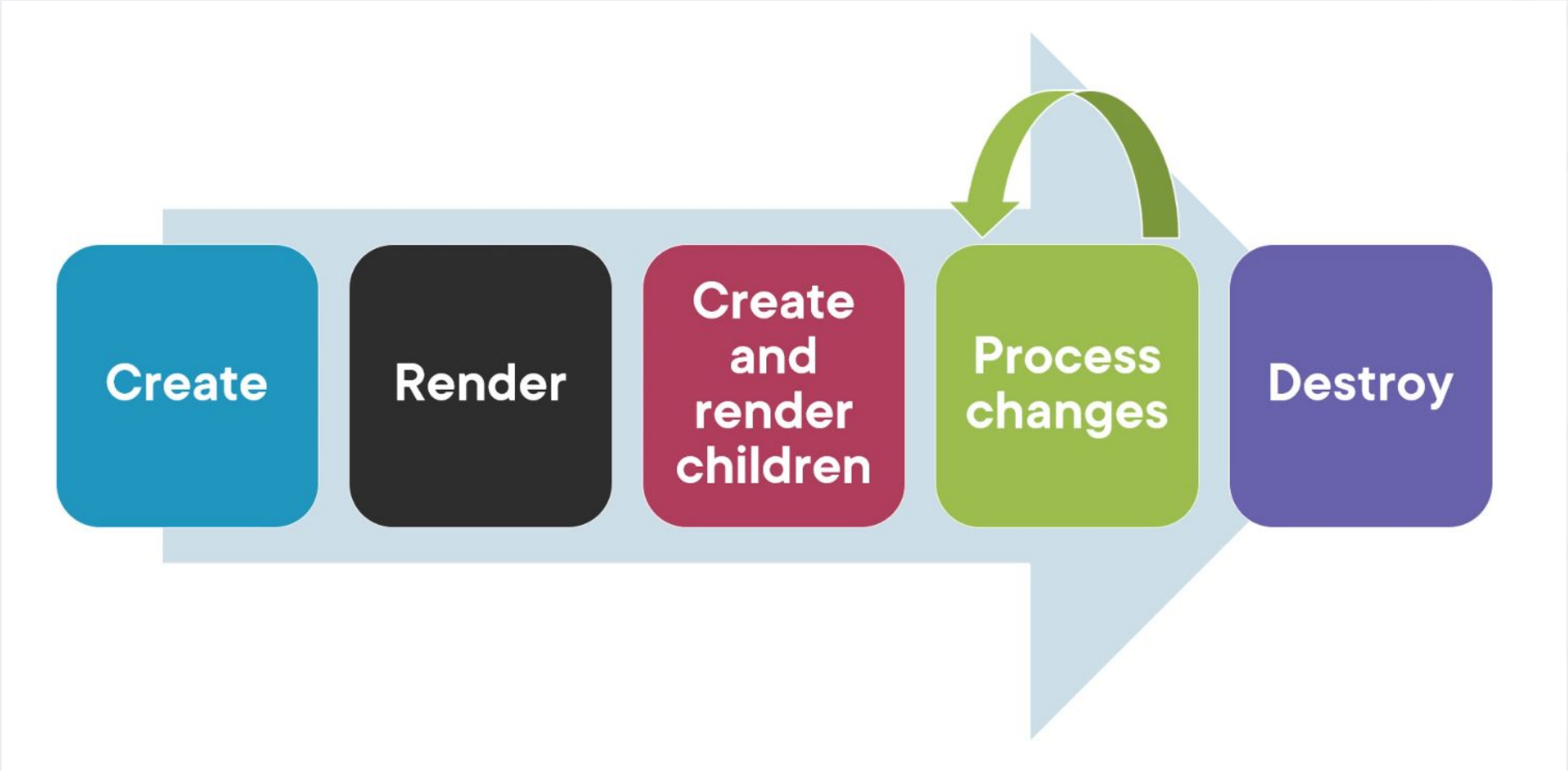
Encapsulating Component Style

- Create the product-list.component.css file
- Link it to the component via the styleUrls property
- Add inside a change to the color of the header section of the table to confirm that it works:

```
thead th {color: #337AB7;}
```

Lifecycle Hooks

- Components have lifecycle events:



Lifecycle Hooks

- A lifecycle hook is an interface provided by Angular that we implement to write code when a component lifecycle event occurs
- For example, once the component loads on the screen, we can send data requests to the server to retrieve data to be displayed on the screen

Lifecycle Hooks

- There are several lifecycle hooks but the most popular are:

  - OnInit: perform component initialization and/or retrieve data

  - OnChanges: perform action after a change to input properties occurs

  - OnDestroy: perform cleanup tasks

- Full list with explanations is available here:

  https://medium.com/ngconf/angular-lifecycle-hooks-977b73fc66af

Lifecycle Hooks

- Let’s import and implement the init lifecycle hook:

  export class ProductListComponent implements OnInit

- IntelliSense tells us what is the name of the function that we need to add to our code:

  ngOnInit(): void { console.log("Init lifecycle hook called"); }

We will implement more complex logic shortly

Custom Pipes

- We can define our own transformation logic to be used with bound properties
- We need to implement the PipeTransform interface and annotate our class with the Pipe decorator
- The logic performed will be contained in a transform method defined by the interface

Custom Pipes

- Create a custom pipe that will replace a character in a string with an empty space
- Use the custom pipe on the product code to replace the dashes with empty spaces
- Note: the first parameter passed to the transform function is the value that we will be working with
- Add the pipe to the AppModule to make it accessible

Custom Pipes

- Notes:

  - in our application we used the shared folder but the structure is entirely up to you
  - naming conventions are also optional
  - the name property in the decorator defines the name that you will be using to refer to the pipe in your code
  - pipes are added in the declarations property of the module along with components

Getters and Setters (JS Review)

```
amount: number = 0;
```

```
private _amount: number = 0;
```

```
get amount(): number {
    // process the amount
    // return amount from private storage
    return this._amount;
}
set amount(value: number) {
    // process the amount
    // retain amount in private storage
    this._amount = value;
}
```

Getters and Setters (JS Review)

- Getters and setters allow us to write specific code to process the value of a variable when it’s get or set
- We can then access the value by using this.amount
- We can utilize this to add a working filter functionality to our application by adding custom logic when changes happen to the filter variable

Getters and Setters

- Transform the listFilter property into a private one with get and set functionality
- Add new property for filtered products (why?): filteredProducts: IProduct[] = [];
- Create a perform filter function with logic to filter the products by name and call it in the set function of the listFilter property

Getters and Setters

- We need the filteredProducts property so that we do not lose our original data which will be needed for future filter executions
- Make sure to substitute products with filteredProducts now in your template's table as you want to display the filtered data instead of the original one
- You can provide this.products as default value for your filteredProducts in order not to see empty screen on load

Nested Components

- Components can be used as a full blown page display view (we will do that later)
- Root / App component
- Directive which is rendered within another component
- Small nested components can be created as part of larger views with tools that allow them to communicate with others

Nested Components

- Let’s create a nested component to present the star rating for our products as icons instead of a number
- We will put it in the shared folder as the idea here is for it to be reused in many other parts and components
- There should already be template and CSS file for it
- Create star.component.ts file as well

Nested Components

- Let’s create a nested component to present the star rating for our products as icons instead of a number
- We will put it in the shared folder as the idea here is for it to be reused in many other parts and components
- There should already be template and CSS file for it
- Create star.component.ts file as well

Nested Components

- Decorate the component connecting the template and the CSS file
- Set some initial values for the properties that we need:

  rating: number = 4; //hardcoded - will change later

  cropWidth: number = 75; //width of the div with the stars
- Next we would like to change the width to correspond to the rating (the number of stars)

Nested Components

- We can achieve that by implementing the OnChanges lifecycle hook and calculating the width via the rating:

```
ngOnChanges(): void {
    this.cropWidth = this.rating * 75 / 5;
}
```

- The 75/5 is basically the width of a single star

Nested Components

- We can now substitute the rating in our products list template with the newly created component: `<td><pm-star></pm-star></td>`
- We also need to import our StarComponent in the AppModule in order for it to be useable
- We have two problems - rating is hardcoded and the OnChanges event never fires as we always see 5 stars

Input Properties

- If a component wants to receive data from its parent

  then the @Input() decorator needs to be used before the property: @Input() rating = 0;

- The parent then passes the data using property binding:

  <pm-star [rating]='product.starRating'></pm-star>

- The @Input() decorator is a function that takes no params

Output Events

- What if we want to go the other way around and pass that from the child component to the parent?
- This is achieved by emitting an event that can be caught and processed by the parent component
- In Angular these events are created via EventEmitter object using the @Output() decorator

Output Events

- Let’s make so when the user clicks on the star rating of a product, we are going to display it in the title of the product list page
- First create onClick function within the StarComponent and bind it to the template’s div:

```
<div class="crop" ... (click)='onClick()'>
```

Output Events

- Within the StarComponent define an emitter object:

  @Output() ratingClicked: EventEmitter<string> = new EventEmitter<string>();

- The <string> annotation is a generic type which means that the event is going to emit data in the form of a string

Output Events

- The created object now has an emit function that could be called with the payload to be emitted
- This will conveniently happen in the implementation of the onClick function from earlier:

```
onClick(): void {
    this.ratingClicked.emit(`The rating ${this.rating} was clicked!`);
}
```

Output Events

- We can now use event binding to capture this event from the parent component
- In order to capture the payload (string in this case) we can use a parameter called $event:

```
<pm-star ... (ratingClicked)='onRatingClicked($event)'>
```

Output Events

- We can now implement that method in the ProductListComponent and update the title:

```
onRatingClicked(message: string): void {
    this.pageTitle = 'Product List: ' + message;
}
```

Services

- A service is a class with a focused purpose
- We use them for features that are independent from any particular content, to provide shared data or logic across components or to encapsulate external interactions
- Examples include login functionality, API calls to retrieve data, authorization, etc.

Services

- If we were to use the services as regular classes then each component would create its own instance
- Instead we want to register the services with Angular which will create a single instance of it for all components called singleton
- Angular gives us access to an Injector which is responsible for registering and holding the services

Services

- If a component needs a service, it defines that service as a dependency and once the component is instantiated the Injector provides (or injects) the single instance of the service to the component
- The process is called Dependency Injection
- Since there is only one instance, all the data and the logic of the service is shared across all components

Services

- A formal definition of Dependency Injection is:

A coding pattern in which a class receives the instances of objects it needs (called dependencies) from an external source rather than creating them itself

Services

- Let’s create a service to retrieve the product data for us
- Since it’s managing product data only we can add it to the products folder as a new file: product.service.ts
- It will be an export of a class annotated with the @Injectable() decorator up top

Services

- Let’s add a method inside that will return our products:

  `getProducts(): IProduct[] { return []; }`

- For now let’s add the hardcoded array we have in the `ProductListComponent` so the method does not return an empty array and we will add HTTP few slides later

Services

- Now we need to register the service with the Injector
- Each component in Angular has its own injector mimicking the structure of our application
- If you register a service with the root component, it is then available across the entire application
- If you register it on a lower level, the scope is limited

Services

- In our case we are going to use the Root Injector and make the service available globally (recommended)
- All we need to do is provide a providedIn property to the decorator function with value of root as this:

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

Services

- If you instead want to register a service to be available only at certain components, you can inject them as part of the component decorator providers property:

```
@Component({...
    providers: [ProductService]
...})
export class ProductListComponent { ...
```

Services

- Also note that the providedIn property is relatively new in Angular but is the recommended way to register the service in the root of the application
- The old way included a providers property in the AppModule instead which is still a valid syntax

Dependency Injection

- How do we inject the service in our component?
- We are going to use the plain old constructor method which is base TS functionality
- Empty constructor is created for every component class by default but now we will explicitly take over to use the constructor for the service injection

Dependency Injection

- The service comes as a parameter to the constructor which then we can assign to a local variable:

```
private _productService;
constructor (productService: ProductService) {
    this._productService = productService;
}
```

Dependency Injection

- Creating a private variable for every service and then piggybacking all the hard work in the constructor is quite some work, not to mention the underscores
- Angular provides a shorthand for the code on the previous slide, automatically creating a private variable for us and assigning the service to it:

```
constructor (private productService: ProductService) { }
```

Dependency Injection

- We now have a reference to the service and we can get the products data from there so we can safely delete the hardcoded array and leave it empty
- Remember the OnInit lifecycle hook that we left unused?
- We can now make our service call there:

```
this.products = this.productService.getProducts();
```

Dependency Injection

- Voila but why is the screen empty?
- We are displaying the filtered products and not the original list which comes at a later stage
- We can fix that by adding another line in the OnInit:

```
this.filteredProducts = this.products;
```

Reactive Extensions (RxJS)

- A library for composing data using observable sequences and transforming that data using operators
- Angular uses Reactive Extensions for working with data, especially asynchronous data

Reactive Extensions (RxJS)

- A library for composing data using observable sequences and transforming that data using operators
- Angular uses Reactive Extensions for working with data, especially asynchronous data
- Observable sequences are collection of items over time
- Unlike array, it doesn’t retain them but observes them over time

Observable

- An observable sequence or stream is also called

  Observable

- It does nothing until we subscribe to it

- Once we subscribe, it starts emitting notifications

Observable

- An observable can emit 3 types of notifications:

  - next: next item is available and emitted
  - error: an error occurred
  - complete: no more items are emitted

- Observables can also be connected to pipes in order to transform the received data (examples to follow)

Observable

- Angular has an HttpClientModule that will allow us to send HTTP requests over the web
- The requests will return an Observable
- We can then subscribe to these observables and process the results

Setting up an HTTP Request

- Let’s start by importing the HttpClientModule into our AppModule under the imports property
- We can now use the HttpClientModule in every component or service in our application
- The import is as follows if the IntelliSense doesn’t help:

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

Setting up an HTTP Request

- We can now go into our ProductService and inject an instance of that module into a constructor like this:
  constructor(private http: HttpClient) {}
- We can simulate a remote server by passing a local URL that points to a JSON file with data. Let's add:
  private productUrl = 'api/products/products.json';

The file can be provided by your instructor

Setting up an HTTP Request

- We can now delete the hardcoded data from the return part of our getProducts method and replace it with a call to the HTTP get method as this:
  `return this.http.get<IProduct[]>(this.productUrl);`
- The generic type `<IProduct[]>` specifies that we are expecting the call to return an array of `IProduct` items

Setting up an HTTP Request

- However, the return data will be wrapped inside of the Observable and in fact, the get method returns an Observable so we need to change the return type of our getProducts method as such:

  getProducts(): Observable<IProduct[]> { ... }

- The generic type here tells us that the observable will return an array of IProduct items

Handling Exceptions

- Before we subscribe to this observable, we need to add some code to handle errors as you never know what will happen when you make HTTP requests
- We can chain the RxJS pipe method after the get call which will allow us to work and process some of the notifications received by the observable

Handling Exceptions

- Inside of the pipe function we can pass comma separated operators which will execute once we get a response from the get request
- One operator is the RxJS tap which allows us to peek at the incoming stream data without transforming it
- Another one is the catchError which allows us to react once an error occurs

Handling Exceptions

- We can implement the pipe and its operators like this:

```
return this.http.get<IProduct[]>(this.productUrl).pipe(
    tap(data => console.log('All: ', JSON.stringify(data))),
    catchError(this.handleError)
);
```

- Don’t forget to import the operators up top:

```
import { Observable, catchError, tap } from "rxjs";
```

Handling Exceptions

- tap takes as parameter the incoming data without modifying it and all we are going to do is log it to the console
- In catchError we will call a named function instead
- We will call that function handleError and it also receives an input parameter of the type HttpErrorResponse
- Error handling in that scenario can be quite sophisticated or simple instead: your instructor can provide a sample code for the method

Subscribing to an Observable

- Angular provides a subscribe method to handle observables
- The method has 3 handlers:

```
x.subscribe({
    nextFunction,
    errorFunction,
    completeFunction
})
```

Subscribing to an Observable

- The nextFunction will be called multiple times when the observable emits data
- The errorFunction will be executed if an error occurs
- You can also provide completeFunction to react and execute some code if you want to catch the complete notification

Subscribing to an Observable

- The subscribe function returns a subscription that you can save in a variable
- You can then call unsubscribe() through that variable if you want to abandon the observable at some point

Subscribing to an Observable

- We can now replace the OnInit method code in the ProductList component to use subscribe:

```
this.productService.getProducts().subscribe({
    next: products => { //reacting on next notification
        this.products = products;
        this.filteredProducts = this.products;
    }
});
```

Subscribing to an Observable

- Since we have code in the service to handle errors, let's also add error handling logic in the component
- Add property up top: errorMessage: string = '';
- We can then add code for the error notification inside the subscribe method in OnInit:

  error: err => this.errorMessage = err

Unsubscribing to an Observable

- Let’s also handle unsubscribing which can happen when the component is being destroyed
- We can implement the OnDestroy hook up top
- Down in the code we will have to define a method:

  ngOnDestroy(): void {}

Unsubscribing to an Observable

- We need a variable to hold the subscription

- Let’s create a property of type Subscription:

  ```
  sub!: Subscription;
  ```

  Don’t forget up top:

  ```
  import { Subscription } from "rxjs";
  ```

- Wonder what the ! is for?

Unsubscribing to an Observable

- When we create the sub property we need to assign a value in TS by providing a default
- The ! is called the non-null assertion operator and it basically tells TS to ignore the value if it's null or undefined which will do just fine for us as we are going to provide the value for it as soon as the OnInit method fires up

Unsubscribing to an Observable

- So let's do that, we assign the return from the subscribe method to our new property in OnInit:

```
this.sub = this.productService.getProducts().subscribe({
```

- We then call the unsubscribe in the OnDestroy:

```
ngOnDestroy(): void {
    this.sub.unsubscribe();
}
```

Navigation & Routing

- An application usually expands beyond having just one page or view
- Let’s use the Angular CLI tool to generate a component for Product Details page and we will then create routes so the user can visit that page. In terminal:

  ng g c products/product-detail --flat//flat means no folder

  g is for generate, c is for component, followed by path

Navigation & Routing

- Note that your AppModule is updated automatically
- Place some basic HTML structure with pageTitle on the top of the template for now
- Add corresponding property to the TS file:

  pageTitle = 'Product Detail';
- Let’s also delete the selector: we are going to use routing to display this component so we don’t need a selector

Navigation & Routing

- Check the AppModule file and see how the new component is automatically added there
- Let’s also add the home/welcome component so we have more options for routes to play with

Navigation & Routing

- We are going to configure routes which the user can activate via actions or links on the page
- Once a route is activated it, it will just replace the currently displayed component on the screen with the component corresponding to the route
- It will also update the URL in the browser
- Technically though we are staying on the same page

Navigation & Routing

- The browser’s URL is updated based on your server configuration
- Back in the day hashtag routing was a default: www.mysite.com/#/products
- It is still supported nowadays although not utilized much
- Don’t be surprised if you see it being used somewhere

Navigation & Routing

- To start we need to register the RouterModule in our AppModule under the imports property
- It gives us access to different router directives that will help us set up the routes in the application
- It also allows us to configure routes and expose them to component so they can be used

Configuring Routes

- Next we need to configure our routes by providing an expected path for each route and the corresponding component that would be loaded
- We specify the list of routes as an array parameter to the `forRoot()` function
- The function is called and attached to the RouterModule:

```
RouterModule.forRoot([]) //route config array in here
```

Configuring Routes

- Configuration route array could look like this - the order matters!

```
[     { path: 'welcome', component: WelcomeComponent },
      { path: 'products', component: ProductListComponent },
      { path: 'products/:id', component: ProductDetailComponent },
      { path: '', redirectTo: 'welcome', pathMatch: 'full' },
      { path: '**', component: PageNotFoundComponent }]
```

- path and component determine the route specifics
- The :id denotes an expected parameter in the route
- The empty route simply redirects to a default component
- The last one is a wildcard which will match any unmentioned route

Configuring Routes

- We don’t have a Page Not Found Component so we can either make one or just substitute the wild card route:

  `{ path: '**', redirectTo: 'welcome', pathMatch: 'full' }`

- Next we can create a menu with links that will allow us to change the route upon clicks

Configuring Routes

- Let’s go to the AppComponent and replace it’s template:

```
<ul class='nav nav-pills'>
    <li><a class='nav-link' [routerLink]=”['/welcome']”>Home</a></li>
    <li><a class='nav-link' [routerLink]=”['/products']”>Product
List</a></li>
</ul>
```

- We bind the routerLink attribute which is a RouterModule directive to an array. The first value in the array is the path we want to go to and you can have extra items passed in (more for that later)

Configuring Routes

- In our case we are not using the array's extra parameters so we can use a shorthand syntax instead:

  ```
  <li><a class='nav-link' routerLink='/welcome'>Home</a></li>
  <li><a class='nav-link' routerLink='/products'>Product List</a></li>
  ```

- Feel free to add some extra CSS to the menu to make it look good

- We now need to specify where the selected route component will be displayed so for that purpose we add another directive to replace the current pm-products:

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

Configuring Routes

- The RouterOutlet will be the place where the components matching the current route will be automatically rendered
- In other words, we are always staying on the same page and looking at the same router outlet but the content inside is changing
- Angular will handle all route changes in the URL and it will also make it so the back and forward buttons work!

Passing Parameters to a Route

- Let's connect that ProductDetailComponent to be displayed when the users clicks on the names of one of the items in the ProductListComponent
- We can add links to the names with the routerLink directive:

  <a [routerLink]="['/products', product.productId]">
  {{product.productName}}

- We need the array here as the second parameter is the :id we specified in the router configuration array

Passing Parameters to a Route

- That works but the Details page would like to read the value of that id parameter and use it
- To access route details from a component, we need to subscribe to the ActivatedRoute service
- Easily doable from the constructor as we already know:

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

Passing Parameters to a Route

- We can use the route Snapshot to read the parameter just one time assuming it doesn't ever change:
  this.route.snapshot.paramMap.get('id');
- If route params are changed, the new values will be emitted as an Observable so we can just subscribe to it:
  this.route.paramMap.subscribe(params ⇒ console.log(params.get('id')));

Passing Parameters to a Route

- When do route params change?

  You can have a page that you always stay on because let's say, you have a "Show Next Product" button

  Pressing the button will not change the component shown on the screen but the parameter in the URL will change

  In that scenario you need to capture the new value

  Note: directly typing different :id in the URL will reload the whole application and render the component from scratch

Passing Parameters to a Route

- In the Details page we can use the snapshot approach
- We can extract the :id and cast it to a Number in the OnInit:

  ```
  const id = Number(this.route.snapshot.paramMap.get('id'));
  ```

- For now, let's just add it to the pageTitle so we can confirm that this code works:

  ```
  this.pageTitle += `: ${id}`; //note the backtick annotation
  ```

Handling Nulls and Undefined

- We are going to need a product property in the Details page which will hold the data for the displayed product
- The data will come from an API call so we won't have it immediately on page (component) load
- So we can define the property to be of time IProduct (which will be invalid until we have the data) OR undefined so it doesn't generate an error like this:
  product: IProduct | undefined;

Handling Nulls and Undefined

- Let’s now remove the id from the pageTitle and substitute the line in the template to show the name: {{pageTitle + ': ' + product.productName}}
- We get an error that the value could possibly be undefined (which is correct)

Handling Nulls and Undefined

- To solve this we can use the safe navigation operator:

  {{pageTitle + ': ' + product?.productName}}

- This is great and it works but it’s not always a solution as it won’t work, for example, with ngModel two-way binding

- We will also have to add this operator dozens of times in the page as we will be displaying many of the product’s properties and we will have to make each of them “safe”

Handling Nulls and Undefined

- A better approach here is to utilize the *ngIf directive and tie it to the containing element like this:
  `<div class='card-header' *ngIf='product'>`
- If the product is undefined then the whole div will not be displayed and checks for undefined are not going to be performed in the data inside

We will add the rest of the page's content shortly

Activating a Route with Code

- We often need to perform some actions before routing - save some data on button press then go to next page
- In these scenarios we want to trigger the routing manually through the code
- Let’s use this approach to implement a Back button in the product details page

Activating a Route with Code

- We need another service to access this functionality:

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

- We can now use its navigate method to trigger a routing event through the code like this:

```
onBack(): void {
    this.router.navigate([‘/products’]);
}
```

Activating a Route with Code

- The passed array is the same structure as in the RouterLink component that we saw earlier
- We can now go and add a footer section in our product details page and bind the onBack function to be executed upon the click event

Protecting Routes with Guards

- We often want to limit the access to certain routes
- Make them available only to logged in users or users with certain permissions
- Angular provides route Guards that we can implement to achieve that functionality

Protecting Routes with Guards

- CanActivate - guard navigation **to** a route
- CanDeactivate - guard navigation **from** a route
- Resolve - pre-fetch data before activating a route
- CanLoad - prevent asynchronous routing

Protecting Routes with Guards

- We can define specific guards in separate classes
- These guards can then be used across the application
- The follow the same Angular structure - implement the interface in interest (i.e. CanActivate) and use a decorator to determine scope

Protecting Routes with Guards

- Let’s create a product-detail.guard.ts page guard:

```
@Injectable({ providedIn: 'root' })
export class ProductDetailGuard implements CanActivate {
    constructor(private router: Router) {}
    canActivate(): boolean { ... }
}
```

- canActivate can return a boolean in a simple setup, true allows access to the route and false does not
- In more complex scenarios it can return an Observable (login, etc)

Protecting Routes with Guards

- To make a guard work for a specific route, we can add it to the respective property in the route configuration in our AppModule file:

  ```
  { path: 'products/:id', canActivate: [ProductDetailGuard], component: ProductDetailComponent }
  ```

- We can also use the CLI tool to generate the guard for us (second g stands for guard here):

  ```
  ng g g products/product-detail
  ```

Protecting Routes with Guards

- The CLI added two parameters to the canActivate function
- route: ActivatedRouteSnapshot - provides current route information details
- state: RouterStateSnapshot - provides general router state information
- The return type is either an Observable, a Promise or a simple boolean value

Protecting Routes with Guards

- We can now write our own logic to guard this route
- Let’s inject a router in the constructor first because we want to redirect the user in case we decide we don’t want to let them in:

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

- We can then get the id from the route and try to parse it to a Number like before:

  ```
  const id = Number(route.paramMap.get('id'));
  ```

Protecting Routes with Guards

- We can now check if the id is valid:

```
if (isNaN(id) || id < 1) {
    this.router.navigate(['/products']);
    return false;
}
return true;
```

- If it’s not we redirect and return false - you might want to redirect to an error page instead

Angular Modules

- The Modules purpose is to organize the pieces of our application and arrange them into blocks
- They also extend the application with capabilities from external libraries
- In a large application it would make no sense to dump everything into one single module

Angular Modules

- A Module declares all the components, directives, pipes and libraries used inside of it and bootstraps a default
- A module can import and export functionalities, including other modules
- Ideally you want a module to include only the pieces needed for the components inside and not more

Angular Modules

- The AppModule will quickly get out of hand once we start adding more to the application - payments, invoicing, feedback, customer service, etc.
- So we can extract the separate functionalities in Feature Modules that work with only what they need
- To demo this we can create a separate ProductModule

Feature Module - ProductModule

- We can use the CLI tool to generate the module:

  ng g m products/product --flat -m app

  m stands for module

  flat means same folder, don't create a new one

  -m app means import the new module in AppModule

- We want to add(import) the new feature ProductModule

  into the AppModule so the application is aware of it

Feature Module - ProductModule

- By default the CommonModule is imported - it contains all the basic Angular matter such as the ngIf and ngFor directives
- Let’s update the declarations array to include all the product related items in our application:

```
declarations: [
    ProductListComponent,
    ProductDetailComponent,
    ConvertToSpacesPipe,
    StarComponent  ],
```

Feature Module - ProductModule

- We can now safely remove those same items from the AppModule as they are already part of the ProductModule and we don’t want to duplicate them
- Remember that every component can belong to only one module and if you want to use them elsewhere you can just export it from that module and import it wherever you need it

Feature Module - ProductModule

- We can now also add FormsModule and RouterModule to the imports section of the ProductModule and remove the FormsModule from the AppModule
- What about routing?

  Remember we called forRoot() in the AppModule?
  In subsequent models we call forChild() instead to denote that we are not in the root of the app anymore

Feature Module - ProductModule

- In order to fix the routes we can cut the product related routes from the forRoot() section in AppModule and add them to the forChild() in the ProductModule
- At this point the application should be back to a working state with now two available modules instead of one
- However, we will have to duplicate many imports in the future, for example, CommonModule, FormsModule, pipes and probably the StarComponent for rating and that's a problem

Shared Module

- To avoid redundant imports of the same module over and over again, we can create Shared Modules that will do all these imports for us at once
- We can then simply import that shared module when we need it
- You can create several different shared modules based on the needs of the application

Shared Module

- Let’s create a Shared Module in our application:

  ng g m shared/shared --flat -m products/product.module

  //we already have the shared folder so use flat to not create another and we want to import this module to the product one

Shared Module

- In the new module let's add the StarComponent in the declarations section and remove it from the ProductModule declarations
- Next we need to define the exports property and expose everything that we want this module to share like this:

```
exports: [
    CommonModule,
    FormsModule,
    StarComponent
]
```

Shared Module

- Note that you can export the FormsModule without importing it first in the imports property
- We can now go and delete these items from the ProductModule and simply add the SharedModule to the imports property there
- You can add the pipe to the shared module as an exercise if you'd like to try it by yourself

Questions?

Thank You!