

Going Deeper with Angular

Angular Basics

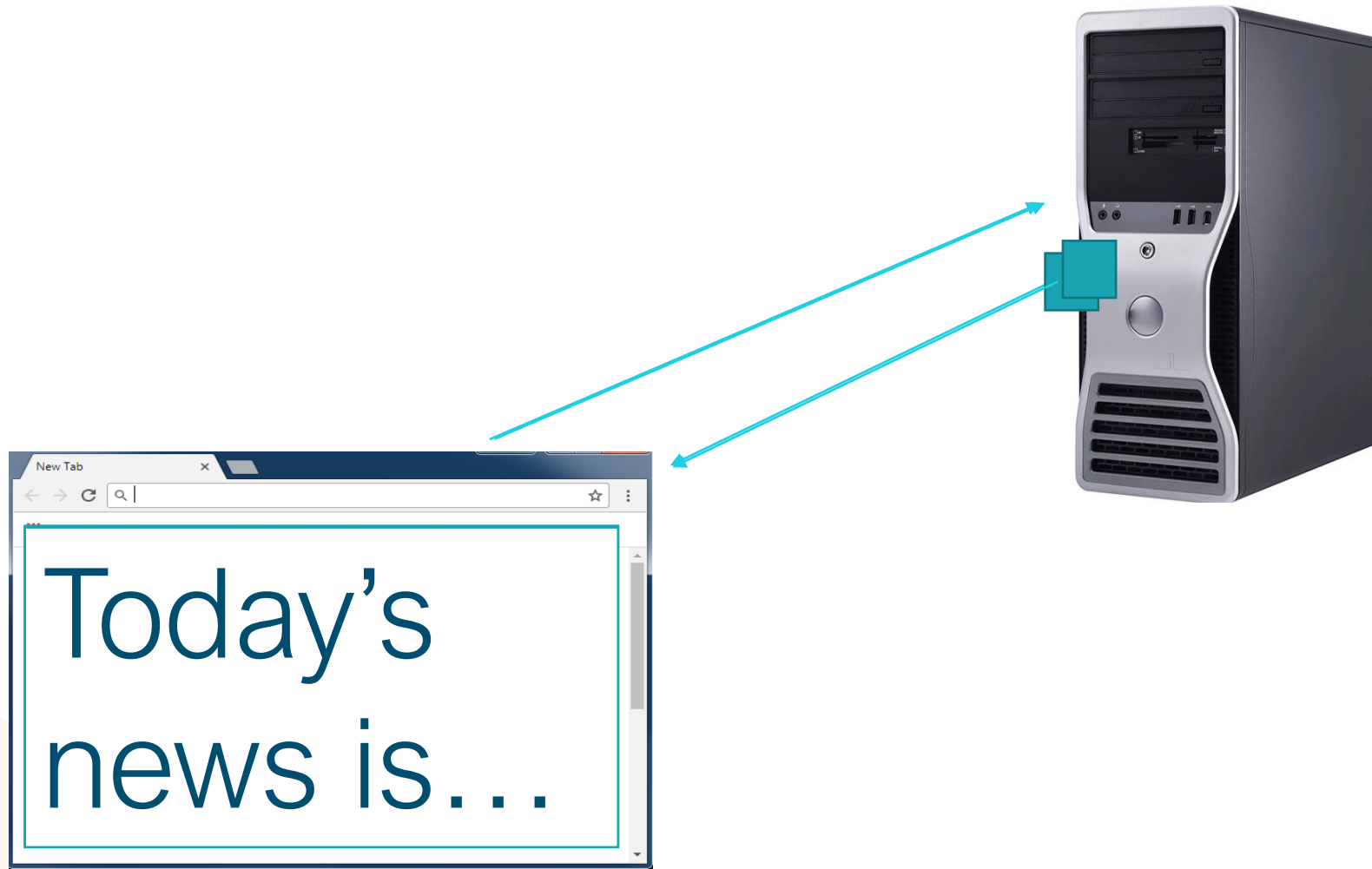
Objectives

- What is Angular?
- Why do we need it?
- What is a Single Page Application?
- Creating an Angular application
- The Angular file structure
- Components
- CSS
- Using typescript in Angular

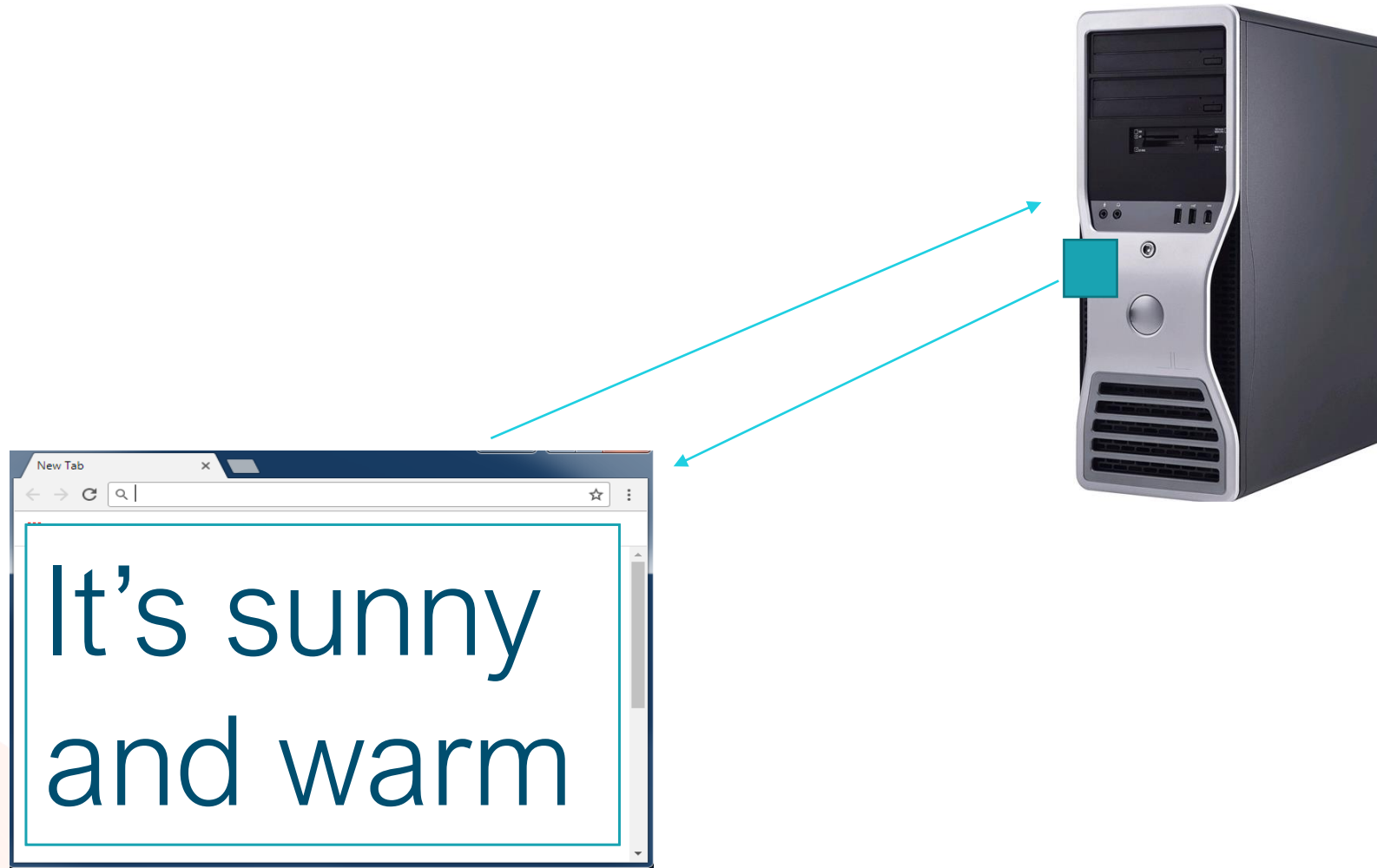
What is Angular?

- Angular is a client-side framework from Google that help us to build interactive, fast, feature-rich websites.
- Sites are built using HTML, CSS and Typescript (a superset of Javascript)
- It is a **single page application** framework

Traditional websites



Single page applications



Single page applications

- The browser will get **one page only** from the server (index.html).
- This page will reference other resources, including javascript files
- The javascript files contain code that **simulates multiple pages** and provides the **full user interface**.
- These can provide a better user experience as we **avoid the server round trip**.
- In Angular, the page we get from the browser is very simple... the complexity is in the Javascript
- Building an application like this from scratch would be very difficult + time consuming. Angular makes it easier.

Transpiling

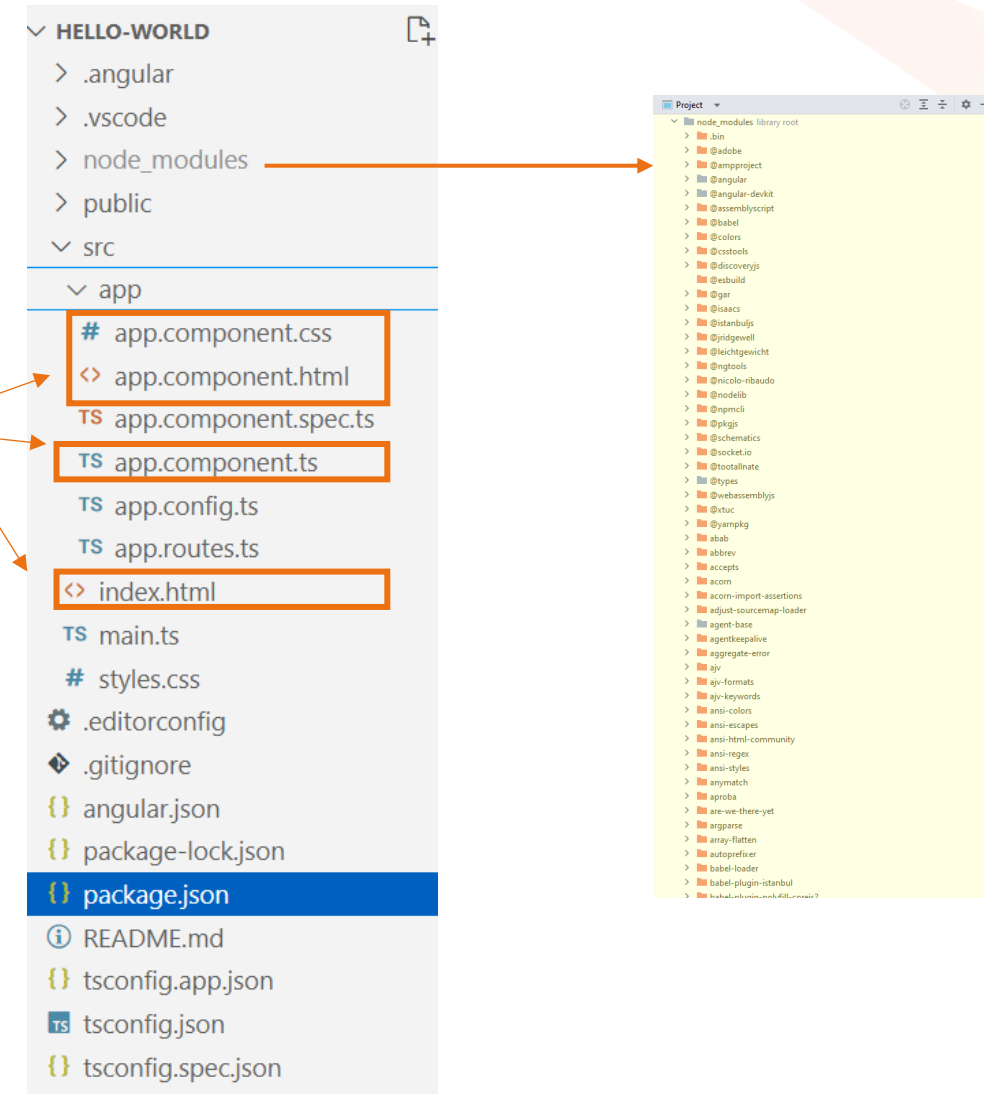
- We will create code in Angular which will be written in **Typescript**
- The application will also contain code from lots of other libraries, and be structured over lots of files.
- We cannot send all the code in a typical Angular project to the browser (it's over 200MB, and **browsers can't run Typescript** by default)
- When we are ready to build the application to put it onto a webserver, the Angular **transpiler** converts our code to plain Javascript that the browser can understand.
- This minimises the code size, and ensures that the browser has everything it needs straight away.
- While we are developing, we can use a **development webserver** that **transpiles live**

Activity – create an Angular app

- Go to your workspace
- Run the command: `ng new hello-world`
- Choose **CSS** for the **stylesheet format**
- Choose **N** for **server side rendering**
- Open the hello-world folder in VS Code
- Start the development server with: `ng serve`
- Open the browser and visit <http://localhost:4200>
- View the page source
- Inspect the page using the developer tools

The Angular File Structure

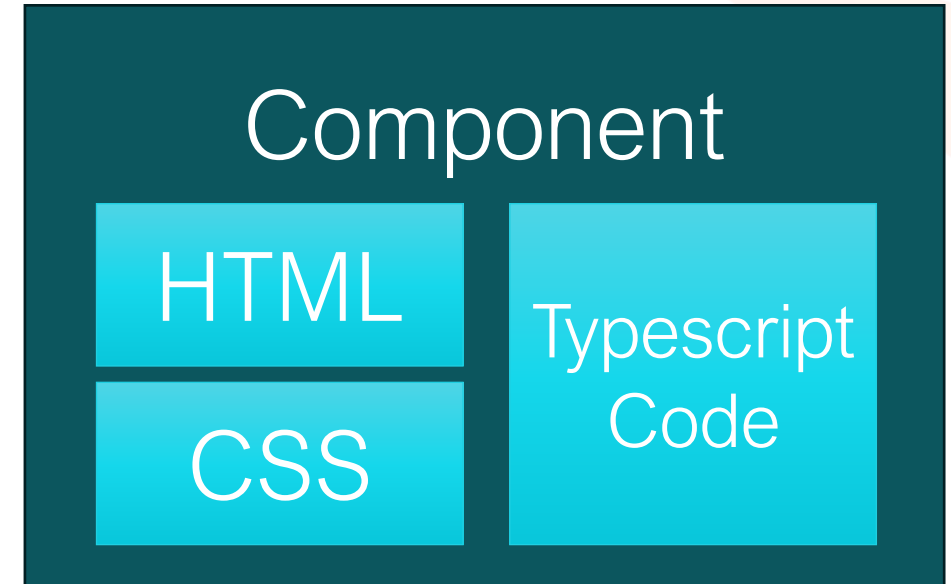
- `Index.html` is the entry point for the application
- The javascript that runs in the browser will execute the file `app.component.ts`
- This file includes `app.component.html` and `app.component.css`
- We can edit the `app.component` files, and add more that can be referenced from here



What are components?

- The core building block of the UI
- Combination of HTML, CSS & Typescript
- Components are placed on the page:
 - Can be re-used on different pages
 - Can be used multiple times on a single page
- Create components with:

```
ng generate component component-name  
ng g c component-name
```



app.component.ts

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

- The selector in a component's ts file defines the html tag used to place a component on the page
- The templateUrl and styleUrls define the locations for the html and css files

css in Angular applications

- Use **styles.css** for global css
- Use **component css files** for component specific css (or a `<style>` tag in the html)
- Angular uses a **shadow DOM** (each component instance has its own DOM) – this enables encapsulation within the component instance
- Use **:host** within css to refer to the element
- Use **--variable-name** within css to define custom variables (typically used for colors)
- Make use of the custom variables with : **var(--variable-name)**

Typescript in Angular applications

- Component level variables are mutable unless defined as **readonly**
- Variables within a function must be defined with **const** or **let**
- Within a function reference component level variables and functions with **this**
- Use the constructor() function to execute code on startup

Summary

- What is Angular?
- Why do we need it?
- What is a Single Page Application?
- Creating an Angular application
- The Angular file structure
- Components
- CSS
- Using typescript in Angular

Going further with Components

Objectives

- Naming Conventions
- Property & Event binding
- Component Interaction
 - @Input
 - @Output
 - @ViewChild
 - @ViewChildren
- Structural Directives
 - @NgIf
 - @NgFor
- Other Directives
- Binding attributes
- Pipes

Naming Conventions

```
room-locator.ts
```

```
room-locator.html
```

```
class RoomLocator {  
  
}
```

Property and event binding

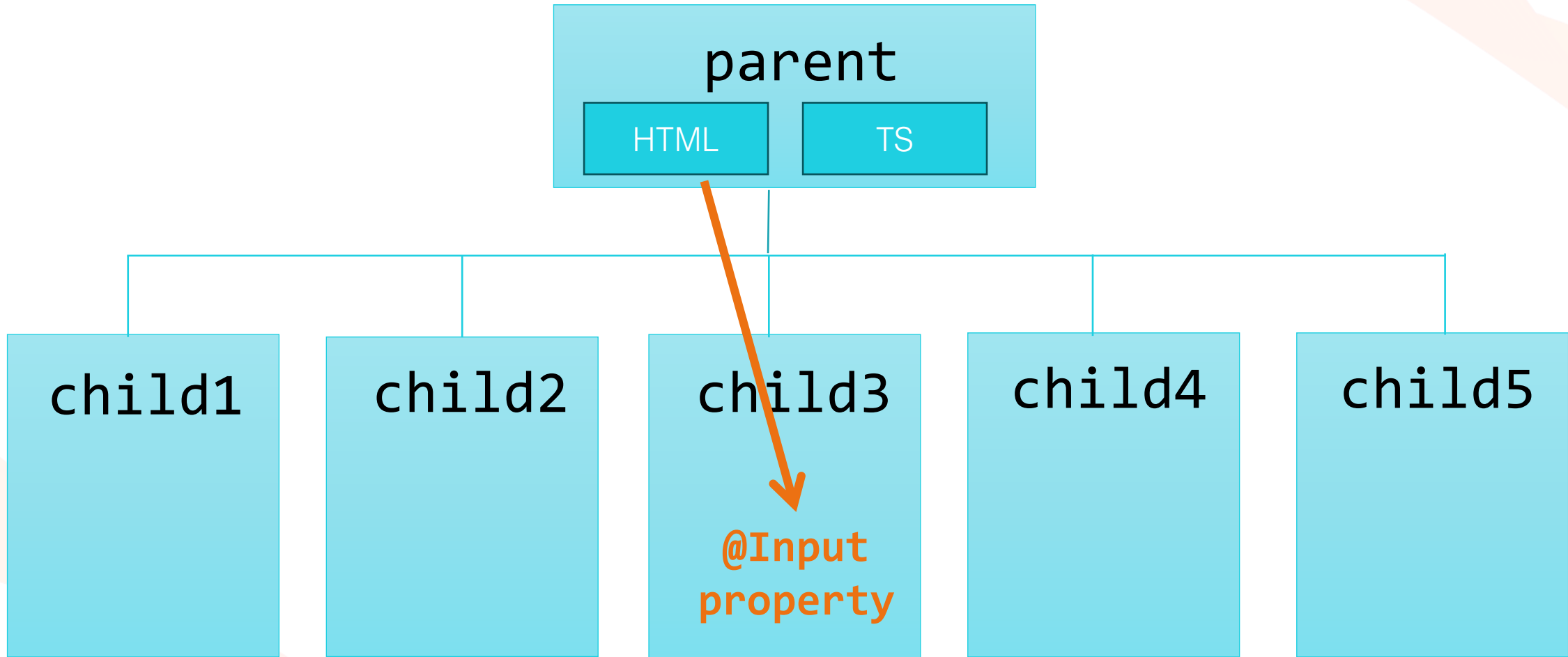
- Class level variables can be inserted into the HTML using `{{ }}`
- We can bind functions to events using `(event)="function()"`
- We can bind to properties of elements using `[property]="variable"`

Activity – Event and property binding

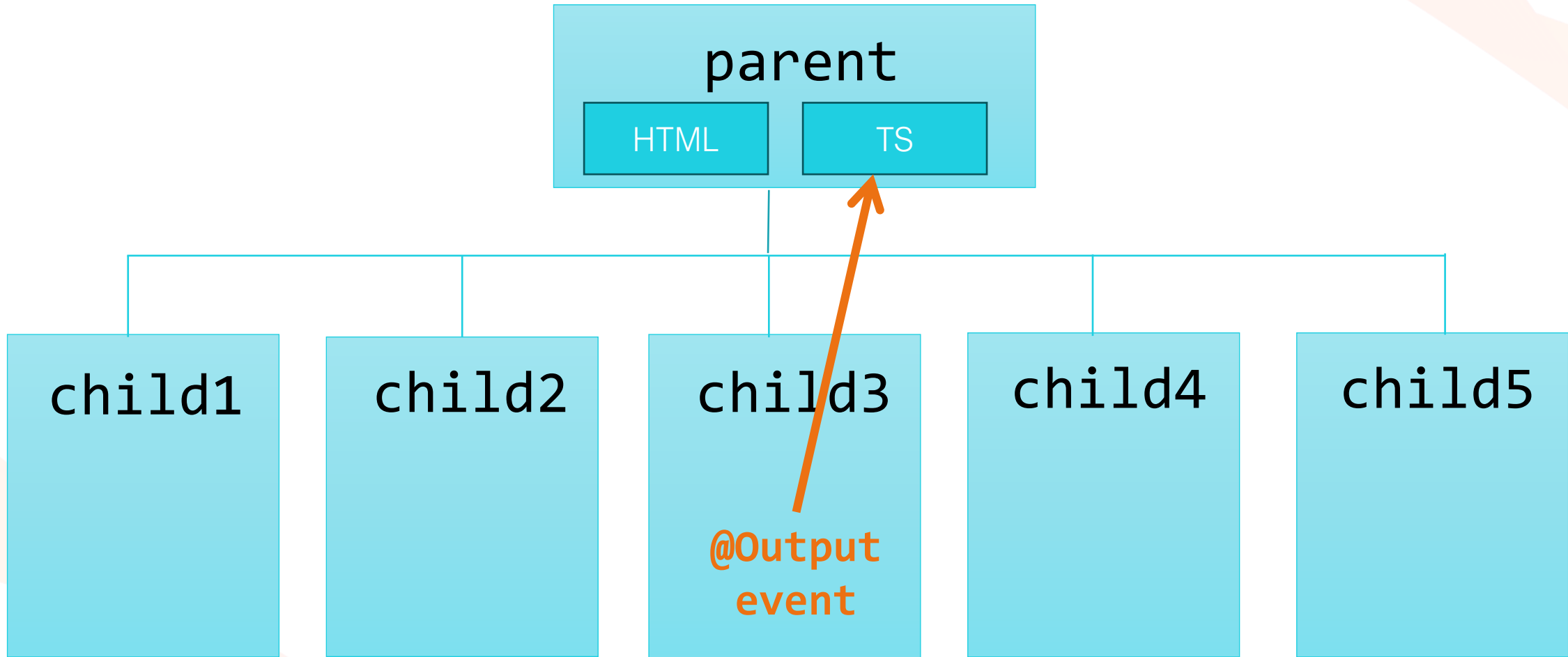
Edit the html and code so that:

- The *title* property name appears in the h1 tag
- The first paragraph displays the *name* property
- Create a new number property called *counter* and display it in the second paragraph
- The button should increment the counter when clicked.
- Add an image to the page – find any image on the internet and save it in the public folder. Place the image in the HTML but use its src and alt properties to bind to variables.

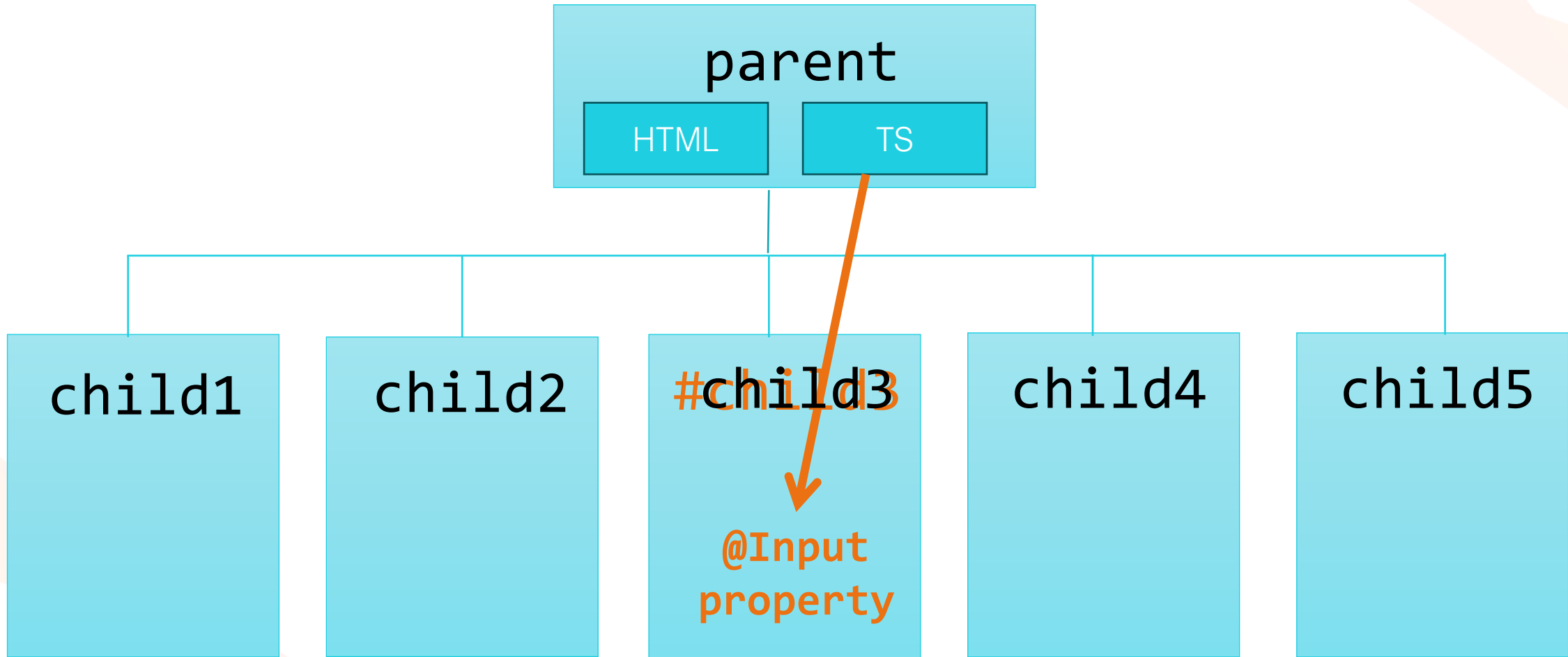
Component Interaction



Component Interaction



Component Interaction



Structural directives

- These are properties that are prefixed with a *
- They alter the layout of the DOM
- *ngIf will display an element if the value evaluates to true

```
<p *ngIf="myVariable">hello!</p>
```

- *ngFor allows us to loop through an array of data

```
<ul>  
<li *ngFor="let name of names">{{name}}</li>  
</ul>
```


Activity – Structural directives

Edit the html and code so that:

- The *isTopSong* property of the Song component is a boolean
- Based on the value of isTopSong, use ngIf to display the words “TOP SONG”

Other Directives

ngClass – bind the class property to a variable or evaluation

```
<p [ngClass]="test ? class1 : class2">hello!</p>
```

ngStyle – bind the style property to a variable or evaluation

```
<p [ngStyle]="{color : test ? 'blue': 'black'}">hello!</p>
```

***ngSwitch** – display the element based on scenarios (expanded version of ngIf)

```
<div [ngSwitch="myVariable"]>
  <p *ngSwitchCase="1">...</p>
  <p *ngSwitchCase="2">...</p>
  <p *ngSwitchDefault>...</p>
</div>
```

Activity – Other directives

Edit the html and code so that:

- The *isTopSong* property of the Song component is used to style the output – put the top songs in bold and red!

Binding css attributes

ngClass / ngStyle – bind the property to a variable, evaluation or array

```
<p [ngClass]="test ? class1 : class2">hello!</p>
```

class / style – bind the property to simple variable

```
<p [class]="myVariable">hello!</p>
```

Dot notation – bind a property based on a variable

```
<p [style.align]="myVariable">hello</p>
```

Transforming output with pipes

- Pipes let us transform the way data is displayed within the HTML
- Simple built in pipes:
 - currency
 - date
 - decimal
 - percent
 - lowerCase
 - uppercase
 - titleCase
- Some pipes take optional parameters – these are provided with a colon separator

```
{{score | percent }}
```

```
{{total | decimal : "1.2-2" }}
```

<https://angular.dev/guide/templates/pipes>

Activity – Pipes

- Add a *price* and a *date released* field to each song. Display this data on the screen using the currency and date pipes.
- Transform the artist into upper case

Complex pipes

- AsyncPipe Read the value from a Promise or an RxJS Observable.
- I18nPluralPipe Maps a value to a string that pluralizes the value according to locale rules.
- I18nSelectPipe Maps a key to a custom selector that returns a desired value.
- JsonPipe Transforms an object to a string representation via JSON.stringify, intended for debugging.
- KeyValuePipe Transforms Object or Map into an array of key value pairs.
- SlicePipe Creates a new Array or String containing a subset (slice) of the elements.

Summary

- Naming Conventions
- Property & Event binding
- Component Interaction
 - @Input
 - @Output
 - @ViewChild
 - @ViewChildren
- Structural Directives
 - @NgIf
 - @NgFor
- Other Directives
- Binding attributes
- Pipes

The Component Lifecycle

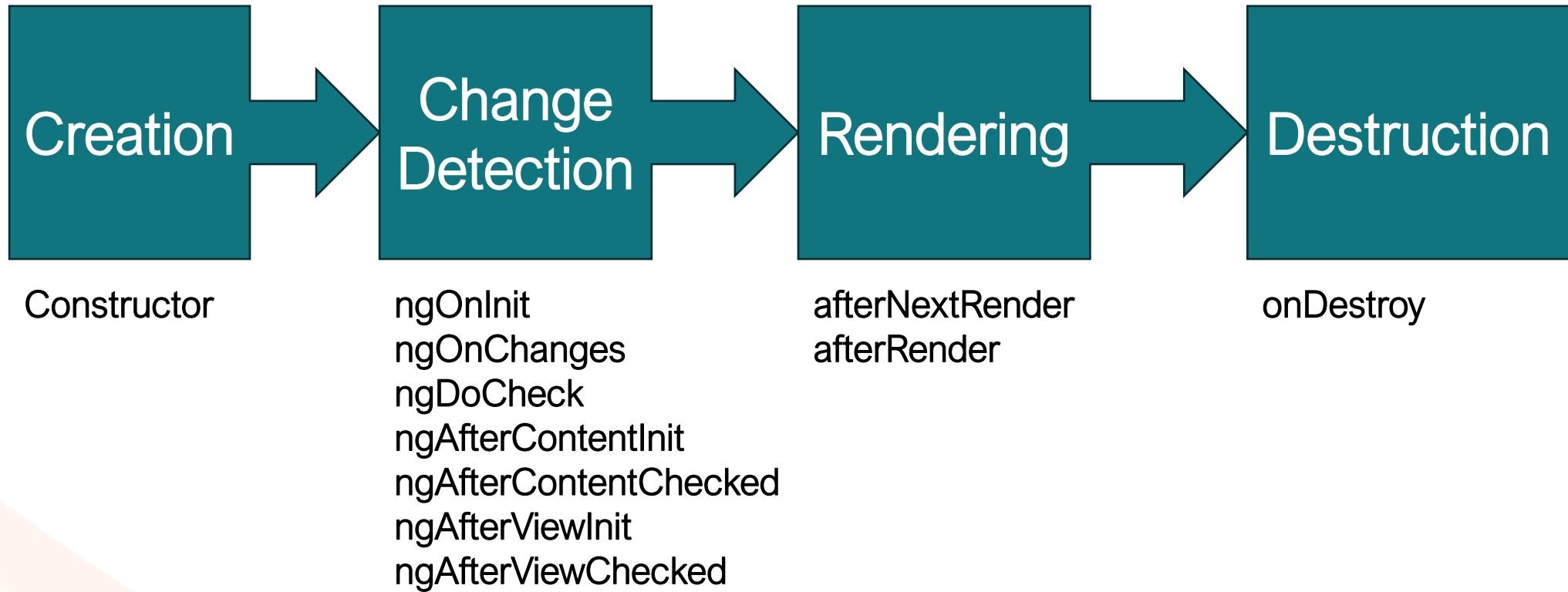
Objectives

- What is the component lifecycle?
- Constructor vs `ngOnInit`
- `ngOnChanges`
- `ngAfterViewCheck`

What is the component lifecycle?

- A series of steps that take place between a component's creation and destruction
- Angular lets us interact with components during this lifecycle using **hooks**
- You must take care if you make state changes in any code that you write inside a hook

Lifecycle phases




Lifecycle phases



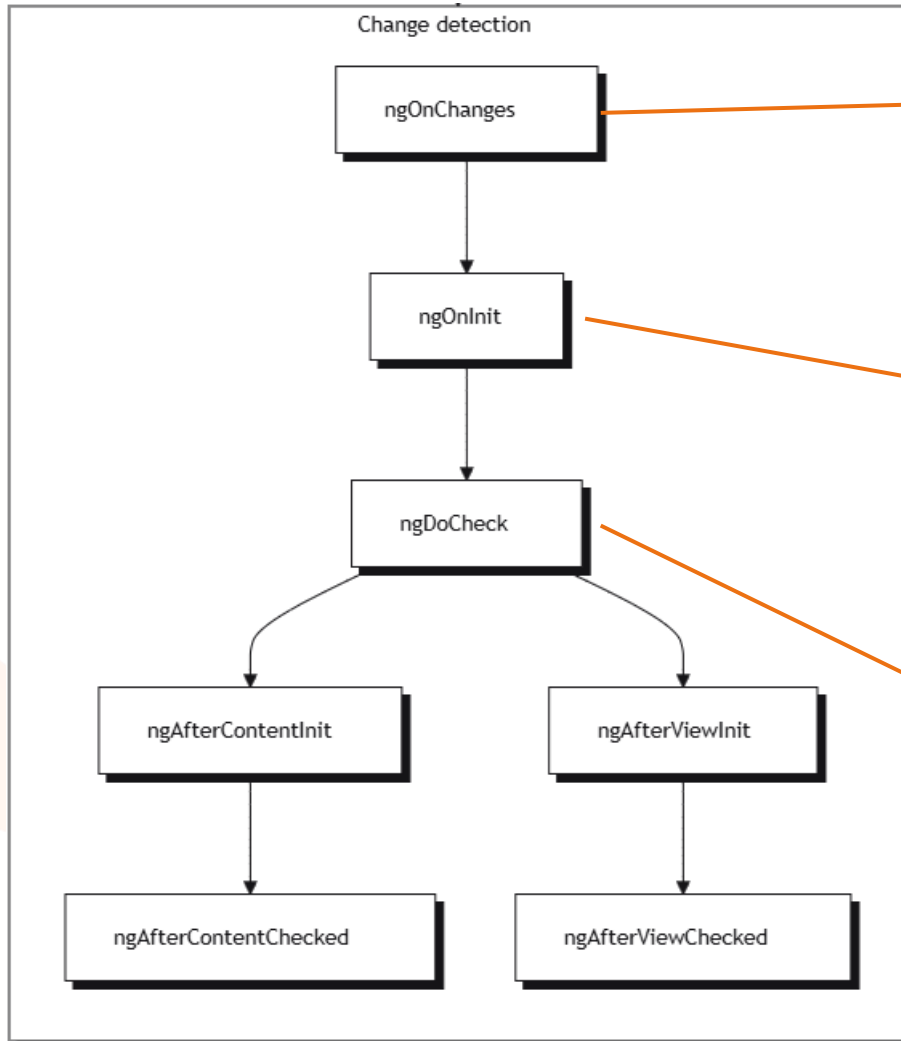
Creation

Constructor

- 
- This is the standard Javascript class constructor
 - Primary use is for dependency injection

Lifecycle phases

Change
Detection



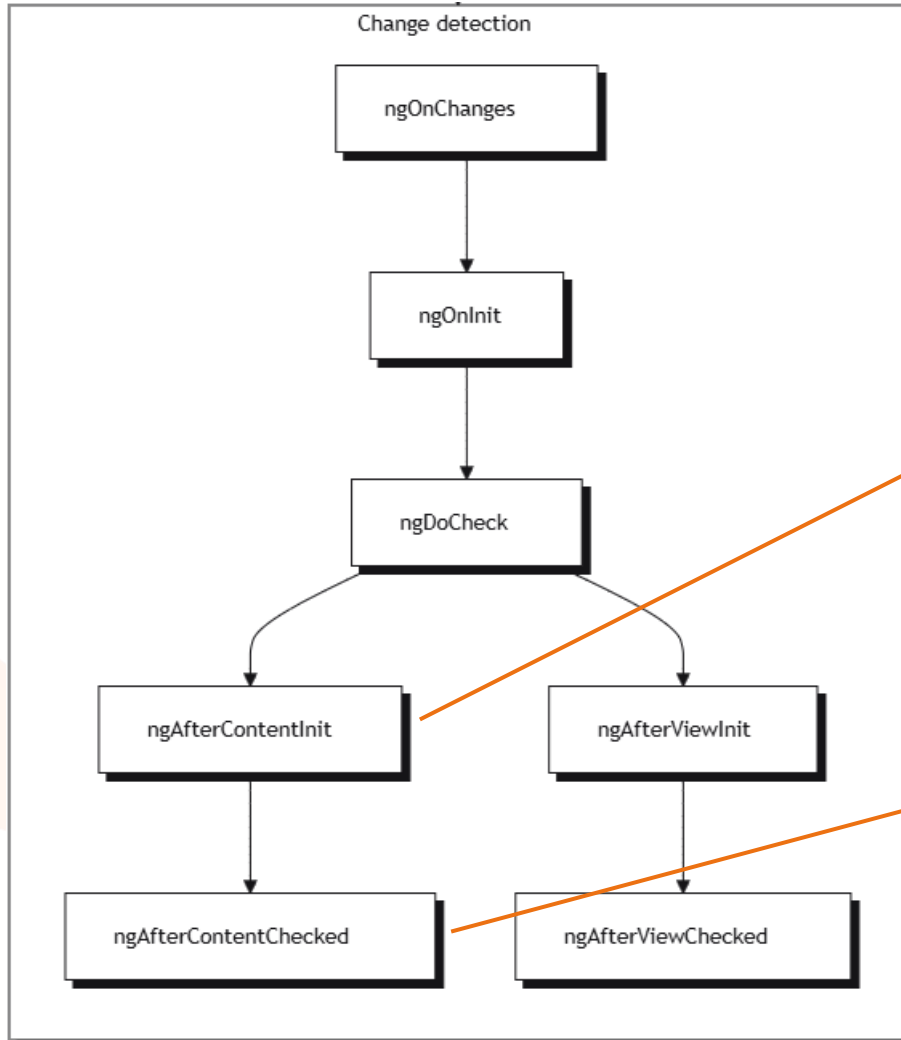
- This runs after any inputs have changed

- This runs after the constructor and all inputs have been initialized
- It's the standard place to put code that must run on initialization

- This runs before Angular checks for template changes – use with caution!

Lifecycle phases

Change
Detection

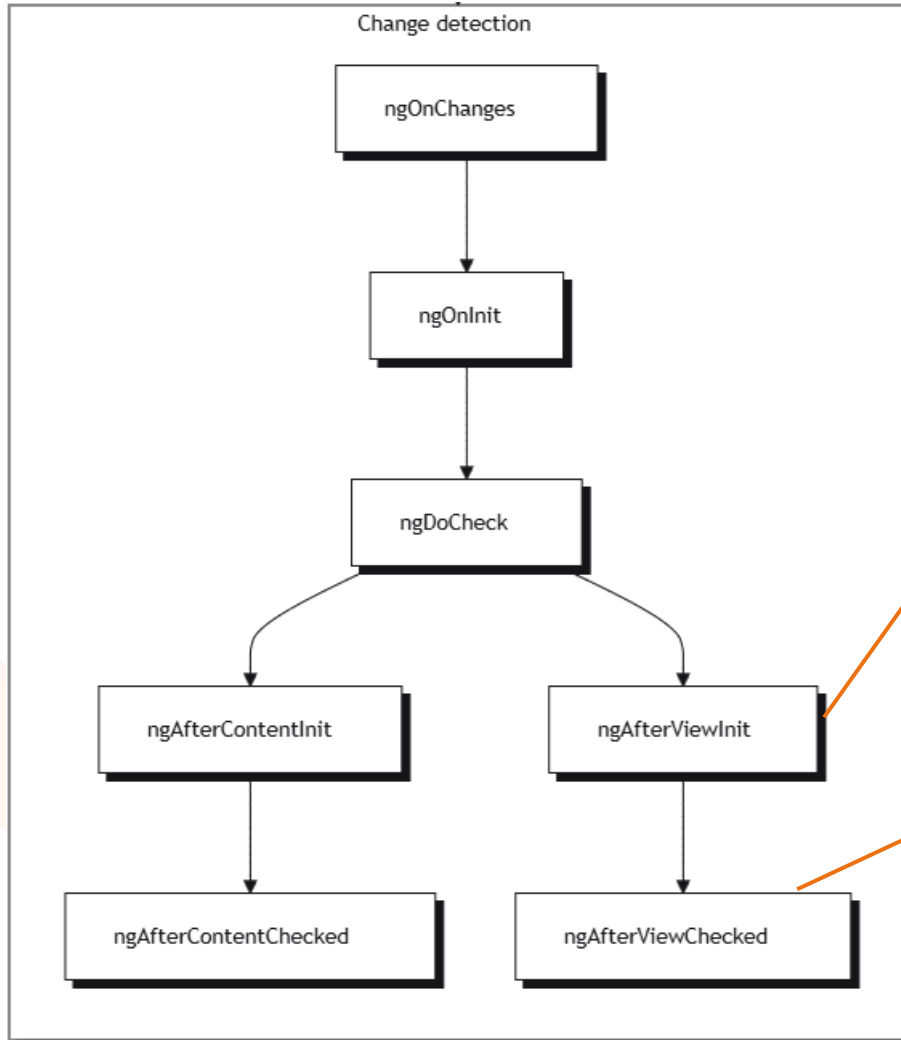


- This runs after the component and all its children have been initialized
- **DO NOT CHANGE STATE IN THIS FUNCTION!**

- This runs before Angular checks for template changes including in the children – use with caution!

Lifecycle phases

Change
Detection



- This runs after the component and all its children have been initialized and the templates created
- **DO NOT CHANGE STATE IN THIS FUNCTION!**

- This runs before Angular checks for template changes including in the children – use with caution!

Summary

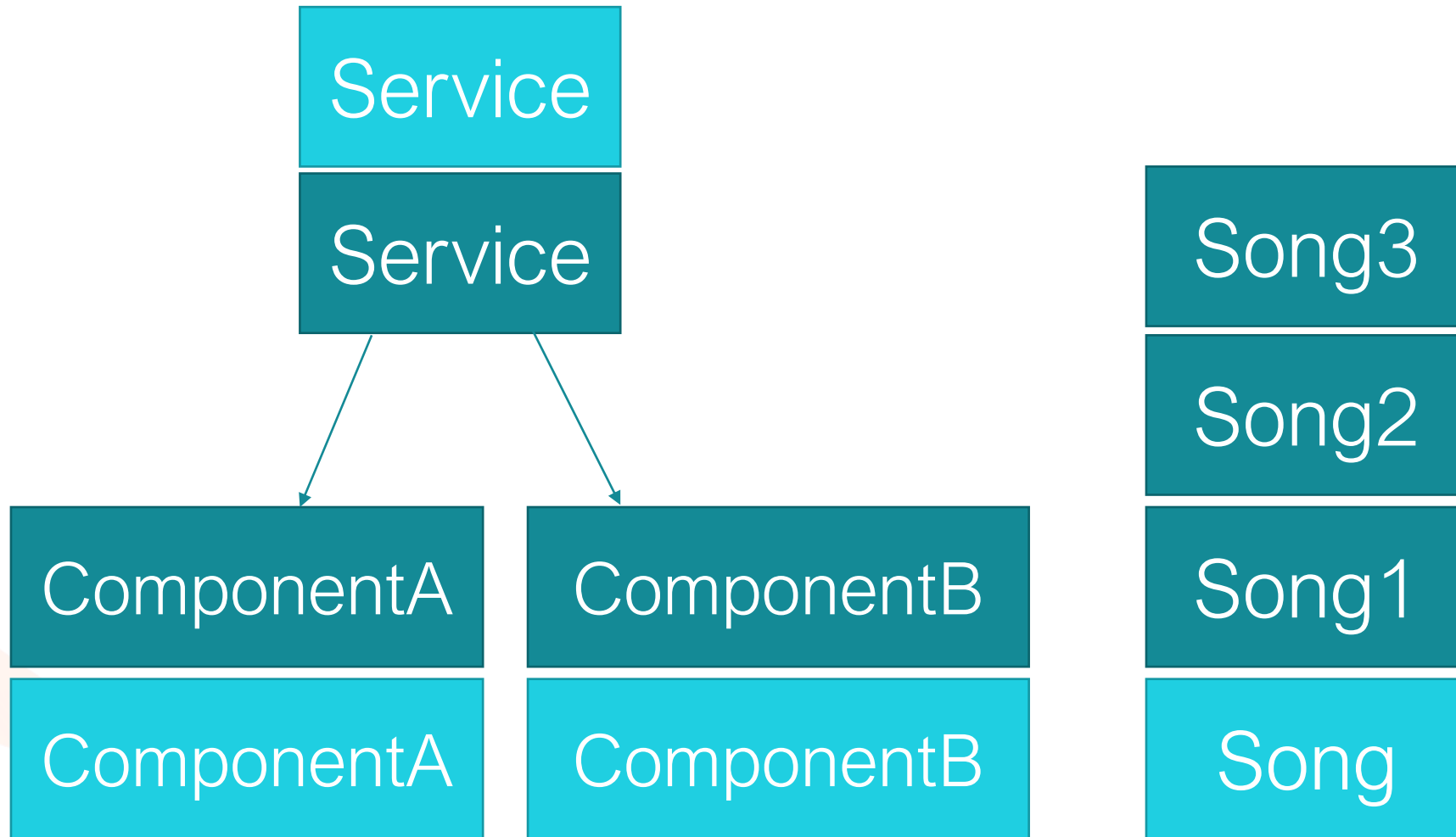
- What is the component lifecycle?
- Constructor vs `ngOnInit`
- `ngOnChanges`
- `ngAfterViewCheck`

Services & Dependency Injection

Objectives

- What are services
- What is dependency injection?
- Standalone mode
- How services work

Services & Dependency Injection



Standalone mode

- Since Angular 17, the cli defaults to standalone mode
- Modules are no longer registered in `app-module.ts`, but must be imported when used
- You can switch to the older format if you create your application with the **--no-standalone flag** – but that's not considered good practice!

How services work

- A service is a regular class
- By using a decorator of @Injectable it means that Angular can instantiate and inject it!
- The @Injectable decorator specifies where in the project the service can be used – typically root. The root injector will instantiate the service at startup if it is used within the application (if not it will remove it from the build tree)
- If you change root to something else, you need to provide an alternative module to do the injecting...

Activity – Creating a service

- Create a new component called SummaryComponent
- Use dependency injection to inject the DataService into the summary component
- In the summary component calculate and display the total price of all songs
- Display the summary component on screen.
- Create a function in the data service that will add 1 euro to the price of each song
- Add a button into the Summary Component that calls that function

Summary

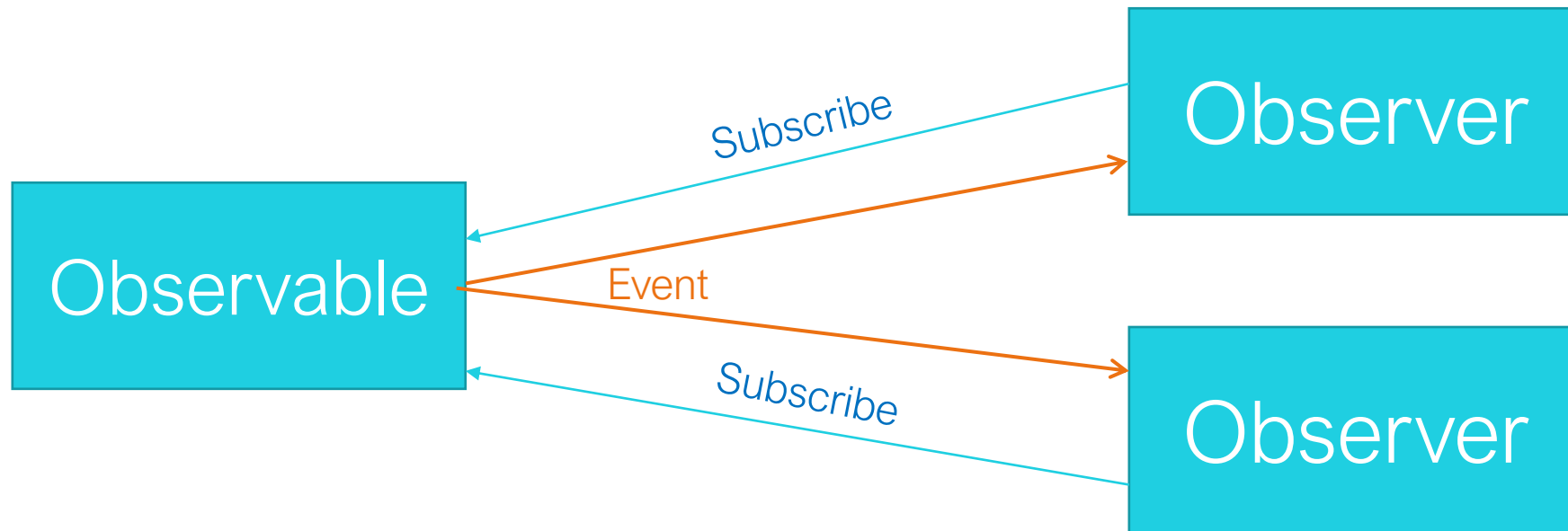
- What are services
- What is dependency injection?
- Standalone mode
- How services work

The Observer Design Pattern

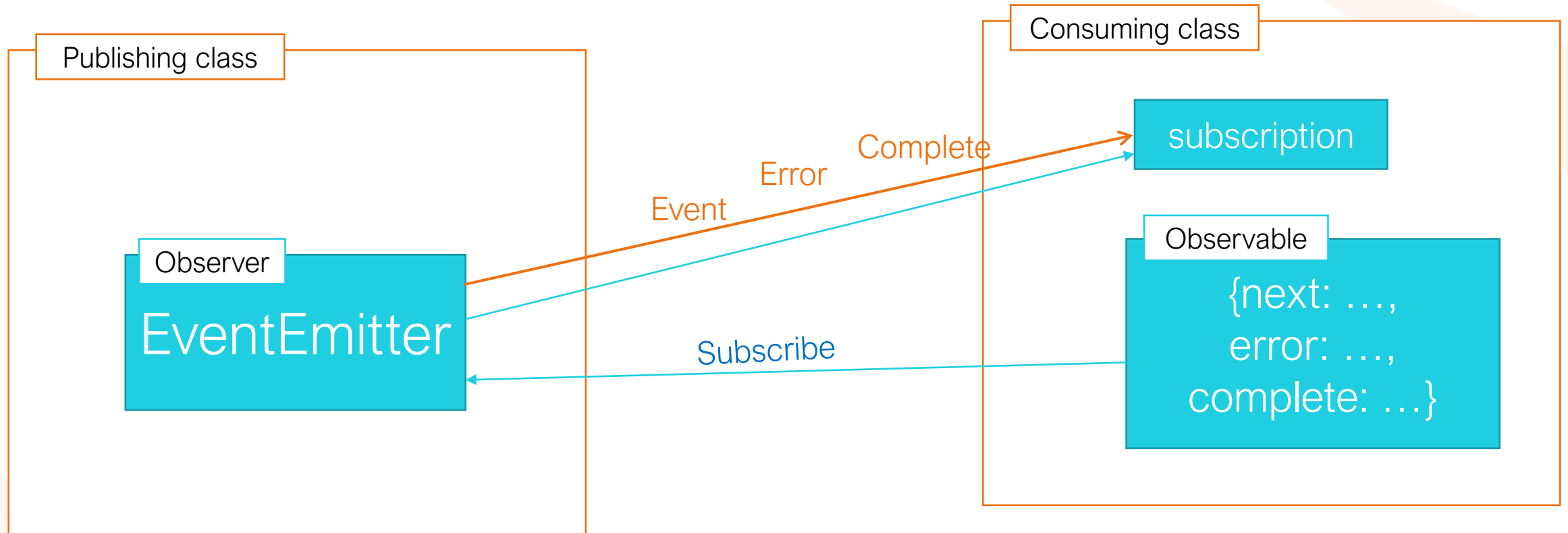
Objectives

- The Observer Design Pattern
- rxjs
- Making a REST request
- Custom Event Emitters
- Subjects

The Observer Design Pattern



rxjs



Making a rest request

- Angular provides an object called HttpClient which makes it easy to make REST requests
- To use this object, you must add the Http Client Provider in the app.config.ts file.
- The httpClient will return an observable that we can subscribe to – when the server responds, we'll get the data, and define in our observer what should happen with it
- This is an appropriate way of working as the rest call is asynchronous – we don't want our application to freeze while it waits for the server to respond.

Custom event emitters

- We can create a custom event emitter by calling `new EventEmitter<type>()`;
- The event emitter has an `emit` function we can call to send out data, together with error and complete functions.
- We must make our event emitter accessible to the places where we want to subscribe to it – this means that it must sit either in a service that can be injected or in a child component.
- Calling the `subscribe` function of the event emitter will allow us to define the observable (as we saw with the rest example)

Subjects

- A Subject in Angular is an object which can act as both an observer and an observable... this means that it can both subscribe to receive events and emit events
- Subjects are useful to act as intermediaries – for example to filter events or transform the data before emitting them
- Subjects also allow us to create observables in a safer way than using event emitters

Activity – Implement the sort!

Implement the sort feature:

- Subscribe to the events in the UserList component
- Respond to the events by changing the sort order of the table
- Remove the subscription from the App component as it's no longer needed here!

Summary

- The Observer Design Pattern
- rxjs
- Making a REST request
- Custom Event Emitters
- Subjects

Using RESTful Webservices

Objectives

- REST basics – Verbs and Status Codes
- Configuring environments
- The HttpClient module
- Making requests and receiving data from a server
- Manipulating data received from the server

(POST & PUT / PATCH coming later!)

Http verbs

- Before we start learning about webservices, there are 2 useful terms to understand, Verbs and Status codes.
- When you send a request to a server over HTTP, such as a request to view a web page, the request always includes a verb. The main verbs are:
GET POST PUT PATCH DELETE OPTIONS
- In standard Html we use GET and POST.
- For example, GET is used when we visit a URL in a browser or click on a link to a web page.
- POST is often used when we send data to a server using a form.
- Technically the difference is that GET is **idempotent** - it won't change any values on the server, so it's safe to repeat. POST is not safe to repeat

Http Status codes

- When a server responds to a client over HTTP it will send back a status code.
- You might have come across the idea of a 404 error page – 404 is a status code.
- The most common status codes are:

200 (OK)

301 (URL has changed)

400 (bad request)

401 (unauthorised)

(403) forbidden

404 (not found)

500 (server error)

(504) timeout

- View the full list at
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

Configuring Environments

- To connect to a server, we will need to know which server to connect to – this will normally be different based on where the application is run (local development environment / test or QA environment / production server)
- Within Angular we can set up a javascript object containing data such as server URL, and have multiple versions of this object – one for each environment
- By default Angular supports 2 environments: production and development but we can add additional environments by editing the angular.json file.
- **ng generate environments** will create the standard environment files
- **ng serve** runs the code using the development environment. To run code using another environment we can do:
 - **ng serve -c environment_name** or
 - **ng build -c environment_name**

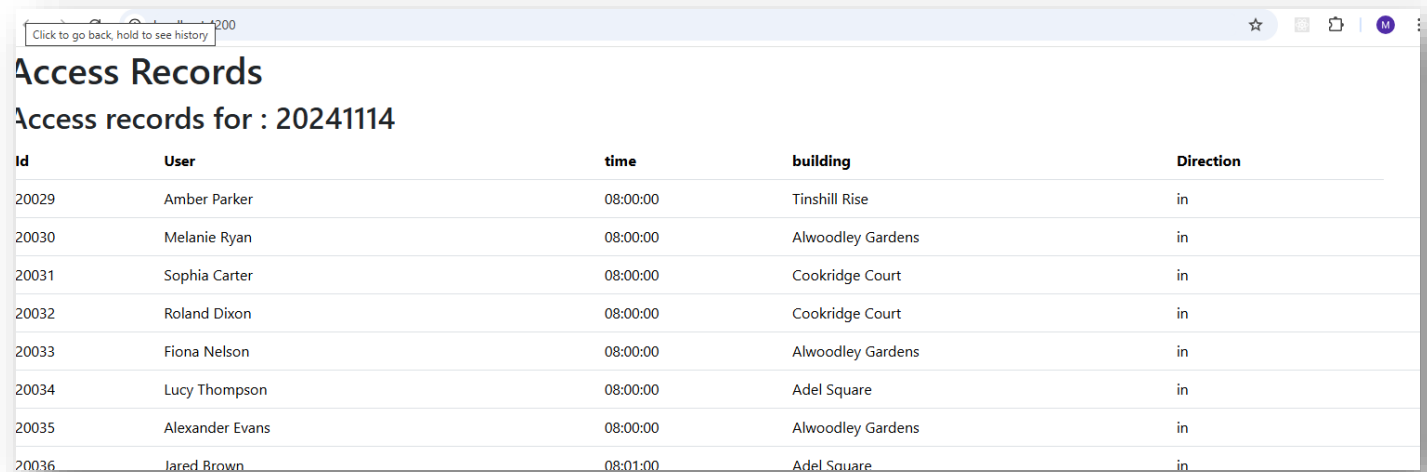
The HttpClient module

- To inject an HttpClient into our project we need to provide the HttpClient module in app.config
- The HttpClient module has methods like get, put, post to match the Http verbs
- We can use generics to specify the data return type – this will enable us to manipulate the data in a meaningful way
- HttpClient returns an observable – we must subscribe to receive the value when the server responds – this is an asynchronous operation.

Activity – Implement a get request

Implement the a get request for the access logs as follows:

- Create a new type to represent an AccessRecord
- Create an function in the DataService to retrieve all access records for a given date (call to /api/logs/\${date}?all=true)
- Create a component that will call this function passing in a hard coded string of yesterday's date and display the data in a table.
- Put this at the top of the App component



The screenshot shows a web browser window with a table titled "Access Records". The table displays access records for the date "20241114". The table has five columns: Id, User, time, building, and Direction. The data is as follows:

Id	User	time	building	Direction
20029	Amber Parker	08:00:00	Tinshill Rise	in
20030	Melanie Ryan	08:00:00	Alwoodley Gardens	in
20031	Sophia Carter	08:00:00	Cookridge Court	in
20032	Roland Dixon	08:00:00	Cookridge Court	in
20033	Fiona Nelson	08:00:00	Alwoodley Gardens	in
20034	Lucy Thompson	08:00:00	Adel Square	in
20035	Alexander Evans	08:00:00	Alwoodley Gardens	in
20036	Jared Brown	08:01:00	Adel Square	in

Manipulating Data received from the server

- Sometimes the data might not be in the right format – we might want to pre-process it in some way, for example if the server schema doesn't match our type, or if we want to manipulate / summarise / filter the data.
- Observables contain a pipe method – this lets us convert the data when it is received – the return value from a pipe is another pipe
- The pipe method takes a transform function called map – this is imported from rxjs

Activity – Who is in the building?

Implement the “who is in the building right now” feature:

- Create a new function in the DataService to retrieve all access records for the current date for a specified building name
- Using the pipe feature, loop through these records and to keep only those records for the current building.
- Create a new component that will call this for the building “Adel Square” (hard code this for now) and display the data in a table.
- Inside this component work out who is in right now by reviewing the data received
 - Suggestion – make a Map of all records with the key being the user Id, so that only the last entry for each user is in the Map
 - All entries in the map where the status is true = users in the building
- Put this at the top of the App component

Note – you can move the ServerAccessRecord type and the convertData method out of the previous method to re-use them

Summary

- REST basics – Verbs and Status Codes
- Configuring environments
- The HttpClient module
- Making requests and receiving data from a server
- Manipulating data received from the server

(POST & PUT / PATCH coming later!)

Routing

Objectives

- What do we mean by routing?
- The routes array
- Implementing basic routing
- Path matching and implementing a 404 page
- Looking at the URL from code
- Programatic navigation
- Working with the Query String
- Thinking about observable ordering

What do we mean by routing?

- Angular is a **Single page application** – this means that there is only 1 HTML file, everything is created in the browser Javascript.
- There are no round-trips to the server to generate new pages as we progress through the application
- We will however want to give the **impression** of navigating through pages to the user – e.g. via a menu
- It can also be helpful to show the user **meaningful URLs**, such as `https://myserver/customers?id=7`

Implementing routing in Angular

- Define a set of routes – this specifies the different URLs in the application and the components that should be displayed when we access the particular URL

```
const routes: Routes = [  
  { path: 'customer', component: CustomerComponent },  
  { path: '', component: HomeComponent }];
```

- Use the router outlet tag to indicate where the components are loaded

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

Implementing navigation in Angular

- Providing an anchor tag – we use the routerLink attribute instead of src. Any component that uses this must import RouterModule.

```
<a routerLink="/help/about">About</a>
```

- Programmatically within a component we need to get access to the Router Module using dependency injection, then we can use the navigate function

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

```
this.router.navigate(['help', 'about']);
```


Path Matching / Implementing a 404 page

- In the list of routes, the first matching path for the URL is used
- We can specify a “catch all” path of “**” to cover pages that are not found.
- This should go at the end of the list, or every path will match against it!

```
const routes: Routes = [  
  { path: '', component: HomeComponent },  
  { path: 'customer', component: CustomerComponent },  
  { path: '**', component: PageNotFoundComponent}];
```

- We can specify parameters in a URL using a :

```
const routes: Routes = [  
  { path: 'customer/:id', component: CustomerComponent}
```

Looking at the route

- Within a component we can find out if there were any parameters in the route by looking at the `ActivatedRoute` object - this can be accessed via dependency injection

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

- The activated route has a `params` property – this is an observable so we need to subscribe to it:

```
this.route.params.subscribe(params => {  
  });
```

- To run this code at the right time, we use **ngOnInit**

Programatic navigation

- Within a component we can change the URL by using the Router object - this can be accessed via dependency injection

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

- The router has a navigate function – provide the path as an array of strings:

```
this.router.navigate(["/abc","def"]);
```

Activity – programmatic routing

In the “Who is in the building” component

- When the component loads, fetch a list of buildings from the back-end and store this in a variable
- Use this variable to populate a dropdown selector
- On the change event of the dropdown selector, navigate to the appropriate URL (hint: inject the Router)
- Only load the data if there is a building selected!

Hint for the change event:

```
<select (change)="handleChangeBuilding($event)">
```

```
handleChangeBuilding(event : Event) {  
  value: String = (event.target as HTMLSelectElement).value;  
}
```

Working with the Query String

- To navigate using the **Router** and supply a query string we can supply this as a JSON object:

```
this.router.navigate(["/abc","def", {queryParams: {value1='ghi', value2=99}}];
```

- To read the query params using the **ActivatedRoute** use the queryParams property (note you need to subscribe to this)

```
this.activatedRoute.queryParams.subscribe(params => {});
```

Activity – using the query string

Edit the Users Sort component so that:

- When you change the sort order the URL is updated with an appropriate query parameter
- When the page loads the query parameter is read from the URL (so that the page can survive a refresh / be bookmarkable / support the back button)

Note: You'll find that when you refresh the page it doesn't sort the data... we'll fix this next

Fixing the sort problem

The reason that the sort is not working when the page is refreshed is:

- Two observables are being subscribed to at the same time:
 - Load data from the server
 - Sort the data
- We cannot sort the data until it is loaded... in the user list component, we need to delay checking for the value in the route until after the data is loaded – we'll therefore move this feature into the user list component

Summary

- What do we mean by routing?
- The routes array
- Implementing basic routing
- Path matching and implementing a 404 page
- Looking at the URL from code
- Programatic navigation
- Working with the Query String
- Thinking about observable ordering

UI Features

Objectives

- AG Grid
- Customising columns
- Filtering
- Creating responsive components
- ngOnChanges
- Styling elements
- Child and auxiliary routes

Introducing AG Grid

- AG Grid is a node module that provides advanced features for displaying tabular data
- It is not part of Angular – it's a separate Node project, but has versions for Angular, React and other JS based projects
- To use AG Grid we must first add it to our project as a dependency and get the node files:

```
npm install ag-grid-angular
```

Getting started with AG Grid

- AG Grid requires a minimum of 2 parameters to be bound to arrays:
 - rowData must be bound to an array of data
 - columnDefs must be bound to an array of column definitions
 - AG Grid supplies a format for the column Definitions (a type called ColDef)
- You must provide css styling for the table to look presentable!

Customising columns

The ColDef supports a number of properties – the most useful ones for displaying data are:

- colDefField – allows us to specify a field using dot notation – useful to get data that is within an object (eg “user.firstname”)
- valueGetter – a function that provides the value to display – useful for calculated values or manipulating the data before display
- valueFormatter – a function that provides a format for display – useful for dates or currencies
- headerName – a string containing the name of the column to display – useful to have a different name from the field name
- See the full list at: <https://www.ag-grid.com/angular-data-grid/column-properties/>

Filtering

AG Grid supports various filtering options:

- Column filter – allow the user to enter values in each column
- Quick filter – allow the user to enter text that filters across the entire table
- External filter – create a filter that works outside the grid
- Advanced filter – column filter that offers additional features (only available with the enterprise version)

Full details are at: <https://www.ag-grid.com/angular-data-grid/filtering-overview/>

Responsive components

- The traditional way to create responsive components is with CSS:

```
@media screen and (min-width: 769px) and (max-width: 1024px) {  
  .myClass {  
    font-size: 24px;  
  }  
}
```

- Angular provides an alternative mechanism called the breakpoint observer – this can be injected using dependency injection
- We can subscribe to the observer to run code whenever changes occur
- This is part of the Angular Material library (a wrapper for Google's Material Design) - <https://v6.material.angular.io/>

ngOnChanges

- Within a component we can find out if there were any parameters in the route by looking at the `ActivatedRoute` object - this can be accessed via dependency injection

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

- The activated route has a `params` property – this is an observable so we need to subscribe to it:

```
this.route.params.subscribe(params => {  
  });
```

- To run this code at the right time, we use **ngOnInit**

Activity - refactoring

To allow us to explore ngOnChanges:

- Refactor the who-is-in-the-building component to create 3 components:
 - Emergency component – this is the wrapper
 - Building-selector component – the dropdown to select a building
 - Who-is-in-the-building component – the list of users for the selected building
- Use an event to output the value from the building-selector to the emergency component
- Pass the value from emergency-component to the who-is-in-the-building component as a property
- Use ngOnChanges to detect when the property changes.

Styling options

- An HTML element can be given a static style as in normal HTML:

```
<div style="background: #ccc; border: 1px solid #000" ></div>
```

- To bind to a variable, we can bind the property with square brackets:

```
divStyle : string = "background: #ccc; border: 1px solid #000"
```

```
<div [style]="divStyle" ></div>
```

- This also allows us to use expressions:

```
<div [style]="myboolean ? divStyle : 'background: #999; border: 1px solid #333'" ></div>
```

- We can also create styles using dot notation:

```
<div [style.background]="myboolean ? '#ccc;' : '#333'"  
      [style.color]="myboolean ? '#000;' : '#fff'"  
></div>
```

ngStyle

- ngStyle works like [style] but expects a JSON object rather than a string

```
divStyle : any = {background: "#ccc" , color: "#000"};
```

```
<div [ngStyle]="divStyle" ></div>
```

```
<div [ngStyle]="myboolean ? divStyle : {background: '#999'}" ></div>
```

- Note that ngStyle must be imported

Child and Auxiliary routes

- By default any route will be mapped to the default `<router-outlet>`
- Auxiliary routes are routes that can be mapped to an alternative router outlet
- Child routes let us create a hierarchy of URL patterns
- Combining these lets us create a parent route which maps to the main router-outlet and a child route which maps to an alternative
- This feature is useful to create:
 - A custom side menu (that changes based on the page)
 - A separation between a view and an edit page

Child and Auxiliary routes

- Child routes can be defined as follows:

```
{path: 'users', children : [  
  {path: 'view', component: UserViewComponent},  
  {path: 'edit', component: UserEditComponent},  
]},
```

- An alternative router outlet is simply a router outlet with a name:

```
<router-outlet name="side-menu"></router-outlet>
```

- An auxiliary route can be defined either by providing the outlet name:

```
{path: 'users', component: MenuComponent, outlet: "side-menu"}
```

- Angular interprets a URL that includes an auxiliary in the format:

```
/users(outlet-name1:aux-url1/outlet-name2:aux-url2)
```

Summary

- AG Grid
- Customising columns
- Filtering
- Creating responsive components
- ngOnChanges
- Styling elements
- Child and auxiliary routes

Going further with data

Objectives

- Slow connections & error handling
- Caching data in the browser
- Using session storage
- Pre-fetching data

Slow connections & error handling

- To create a good user experience, the user needs to know if the application is doing something – we need to give **visual clues**.
- When a server doesn't respond or responds in an unexpected way, we need to handle the error appropriately.

Caching data in the browser

- A page refresh will generally require any data on the page to be re-loaded from the server.
- For data that changes infrequently it can make sense to cache this in the browser's session storage
- This is simple to do – we just use:

```
localStorage.setItem(key,value);
```

```
localStorage.getItem(key,value);
```

- Note that the values are always stored as / read as strings
- You can also create and read cookies from Angular but this is not recommended due to EU cookie laws (and cookies have a limited size)

Activity – session storage

The building-selector component loads a list of buildings – we will cache this information. Edit this component so that:

- When the buildings are fetched, the data is stored in session storage, together with the date and time
- When the component is loaded, session storage is checked, and if the data is present and is less than 2 minutes old, we'll use it, otherwise we'll fetch the data from the server
- Implement appropriate error handling too!
- Use `console.log` for testing so that we know which version is used...

Pre-fetching data

- Angular lets us fetch data needed for a component BEFORE the component is displayed on screen, using a resolver
- This gives us a way of providing a better user experience when we have NOTHING to display until data is available
- We first create a class that implements `Resolve<Observable<DataType>>`
- Then pass this into the route as a resolve parameter
- The data is then loaded before the component, and can be accessed within the component by looking at **`this.route.snapshot.data`**

Summary

- Slow connections & error handling
- Caching data in the browser
- Using session storage
- Pre-fetching data

Testing

Objectives

- Unit testing
- What are providers?
- Creating test methods
- Reading the screen
- Limiting test method execution
- Simulating user interaction
- Interacting with code
- Testing behaviour
- Testing events
- Mocking dependencies

Unit testing

- Angular automatically creates files with the extension **.spec.ts** to contain test methods
- We execute our test methods with the **ng test** command. This uses frameworks called Karma and Jasmine to run our tests.
- A test method is a method called **it** in the format:

```
it('description', () => {  
  const fixture = TestBed.createComponent(MyComponent);  
  const app = fixture.componentInstance;  
  
  expect...  
});
```


What are providers?

- Angular can inject a class using Dependency Injection if it is decorated with `@Injectable` – for our own services, that's all we need to do
- When the full application runs, Angular uses providers to create an instance of objects like Router, ActivatedRoute, HttpClient – these providers are configured in the `app.config.ts` file
- In the test environment, this file is not used, and we need to manually specify the providers required to create the classes that are to be injected into each component we create

```
beforeEach(async () => {  
  await TestBed.configureTestingModule({  
    imports: [AppComponent],  
    providers: [provideRouter([]), provideHttpClient()]  
  }).compileComponents();  
});
```

Test methods

- Tests are contained within a container created by calling the **describe** function.
- A method decorated with **@beforeEach** is the setup method – it runs before each independent test
- A method called **it** will be executed as a test method
- Within a test method assertions are created in the format

```
expect(someValue).evaluationFunction(expectedValue);
```

- The main evaluation functions are:
 - toBe, toEqual, toContain
 - toBeTrue, toBeFalse, toBeTruthy, toBeFalsy
 - toHave...

Reading the screen

- The **fixture** object lets us find HTML elements on the screen with the screen with methods:
 - `nativeElement` = the entire HTML
 - `nativeElement.querySelector('h1')` or `querySelectorAll`
 - `nativeElement.query(By.css('className'))`
 - `nativeElement.query(x => x.attributes[''] === '...')`
- `nativeElement` allows us to explore the DOM
- `debugElement` allows us to explore the Angular component

Activity – test methods

- Create a test method in the App component testing file to check that “Building Access Management System” appears on the screen

Limiting test execution

To limit which tests run

- Use `fit` / `fdescribe` to run just one test / one suite of tests
- Use `xit` / `xdescribe` to exclude one test / one suite of tests

Simulating User Interaction

To simulate a button click / user typing in a box etc.

- Use `debugElement`
- find the relevant object
- set its properties (such as value)
- call its event (such as click)
- Wait for the screen to refresh with: `fixture.detectChanges()`

Interacting with the code

Use the **component** object for:

- Reading and changing properties of the component
- Executing functions within the component

Activity – test methods

Create a test method for the UserSort component:

- First edit the component to store the sortType as a local variable
- Edit the handleClick button to store the sortType in the local variable
- Test that when the button containing the text “Sort by firstName” is clicked, the local variable is set with the value 2

Testing behaviour

To check if functions were called within our components:

- first tell the testing framework that we want to **spy** on the function
- optionally add `.and.callThrough()` to tell the test frame to still run the function

```
spyOn(component, 'functionName').and.callThrough();
```

- use `toHaveBeenCalled` to check if they were called:

```
expect(component.functionName).toHaveBeenCalled();  
expect(component.functionName).toHaveBeenCalledWith(param);
```

- To check the return value of a function, get a reference to the spy:

```
const spy = spyOn(component, 'functionName').and.callThrough();  
//do action  
const returnValue : string = spy.calls.mostRecent().returnValue;
```

Activity – test methods

- Edit the user-sort component so that it calls another method:

```
demoFunction(sortNumber: number) : string {  
    return `Sort by ${sortNumber}`;  
}  
  
handleClick(sortType: number) {  
    this.demoFunction(sortType);  
    ...  
}
```

- Create 3 test methods:
 - Check that clicking on the first name sort button calls the handleClick function with a parameter of 2
 - Check that clicking on the first name sort button calls the demo function with a parameter of 2
 - Check that clicking on the first name sort button calls the demo function which returns the string "Sort by 2"

Testing events

To test events are emitted we can either:

- Subscribe to the event in a test
- Spy on the event's emit function

Mocking dependencies

To mock a dependency (such as a service):

- Use `jasmine.createSpyObj` to create a dummy version of the object with specific methods named

```
const mockService = jasmine.createSpyObj('MyService', ['method1', 'method2']);
```

- For each method provide an implementation using `callFake`

```
mockService.method1.and.callFake( () => {} );
```

- Create a provider to inject the mock into our component

```
await TestBed.configureTestingModule({  
  imports: [MyComponent],  
  providers: [{provide: MyService, useValue: mockService}]  
})  
.compileComponents();
```

Summary

- Unit testing
- What are providers?
- Creating test methods
- Reading the screen
- Limiting test method execution
- Simulating user interaction
- Interacting with code
- Testing behaviour
- Testing events
- Mocking dependencies

Forms

Objectives

- Template driven forms
- Validation in template driven forms
- Reactive forms
- Populating data in a reactive form
- FormBuilder
- FormArray

Template driven forms

- Template driven forms are the simplest type of form we can create:
 - Import the FormsModule into the component
 - Bind the inputs to a backing data field using [(ngModel)]. Ensure the field also has a name

```
<input type="text" [(ngModel)]="myVariable" name="WmyField" />
```

- Bind the form's submit event to a function

```
<form (submit)="handleSubmit()">
```


Template form validation

- Angular automatically applies classes to form elements:
 - ng-valid / ng-invalid
 - ng-untouched / ng-touched
 - ng-pristine / ng-dirty
- The required attribute will impact ng-valid/ng-invalid but will not automatically result in a browser warning on submit
- We must implement the required attribute or custom validation manually*

* Custom validation is better done with Reactive Forms

Activity – implement the required validation

To make the required field work in a meaningful way:

- Define css rules so that an input with ng-dirty & ng-invalid contains a red border
- Create a local boolean variable to store whether the form is valid
- Create a method to check if the value of loginData is valid (both fields have > 0 length) and to update the boolean variable with the result
- Bind this method to the change event of each of the inputs, so that it updates as the user types
- Disable the submit button if the form is not valid.

Reactive Forms

- Reactive Forms offer a more dynamic way to build forms in Angular:
 - Import the ReactiveFormsModule module
 - Create a programmatic structure of FormControl objects to represent the form elements in code:

```
myForm = new FormGroup({  
  field1 : new FormControl('field1'),  
  field2 : new FormControl('field2'),  
});
```

- Bind to the HTML using formGroup on the form and FormControlName on the elements

```
<form [formGroup]="field1" >
```

```
<input type="text" id="field1" FormControlName="field1" >
```

Reactive Forms – populating data

- To populate data into a ReactiveForm, we use the patchValue function of the formGroup

```
this.myFormGroup.patchValue({  
  field1 : value1,  
  field2 : value2  
});
```

The FormBuilder

- The FormBuilder lets us create the programmatic form group with less code:

```
// @ts-ignore
myForm : FormGroup;

ngOnInit() {
  this.myForm= this.formBuilder.group({
    field1: '',
    field2: ''});
}
```

- It also allows us to specify validation rules using validators:

```
this.myForm= this.formBuilder.group({
  field1: ['', [Validators.xxx, Validators.xxx]],
  field2: ''});
```

- We can check if fields are valid in the html:

```
<div *ngIf="myForm.controls['field1'].invalid" ...
```

Activity – finish the Edit User form

To make a good user experience:

- Make any field which is invalid be displayed with a red border
- If a field is < 3 characters display a suitable message below the field
- If any field is invalid, disable the submit button.

Note: Ignore the id field when determining validity.

Form arrays

- We can use FormArrays to create a form of dynamic data (unknown length)
- A FormArray is an array of FormGroup that must itself be contained inside a FormGroup

The general process is:

- Use the form builder to create a new formArray and insert it into a form group
- Define a getter method to enable us to work with the formArray
- Add or remove formGroups from the formArray
- Bind the FormGroup containing the FormArray to HTML using `*ngFor` to loop through the individual FormGroup in the FormArray

Summary

- Template driven forms
- Validation in template driven forms
- Reactive forms
- Populating data in a reactive form
- FormBuilder
- FormArray

Linting

Objectives

- What is Linting?
- Linting in Angular
- Basic configuration

What is linting?

- Code linting is an automated process to help ensure code quality
- It will consider:
 - Code styling errors
 - Code efficiency
 - Some compile errors

Linting in Angular

- Angular doesn't automatically include a linter, but it can be added with **ng lint**
- This will add a file called `eslint.config.js` to your project, which can be used to configure linting rules.
- Add the ESLint extension into your IDE to see the linting rules be dynamically applied to your files as you code.

```
Array type using 'Array<ServerAccessRecord>' is forbidden. Use 'ServerAccessRecord[]' instead. eslint(@typescript-eslint/array-type)
```

```
interface Array<T>
```

```
View Problem (Alt+F8) Quick Fix... (Ctrl+.) Fix using Copilot (Ctrl+I)
```

Activity – fix the linting rules!

- Review the code and improve the quality to remove all linting errors and warnings
- Most of these will be straightforward code styling issues...

Note you should replace

```
//@ts-ignore
```

With

```
//@ts-expect-error('form group is set up in ngOnInit')
```

Summary

- What is Linting?
- Linting in Angular
- Basic configuration

Post, Put and Delete

Objectives

- Rest methods with data
- Creating a meaningful user experience

Rest methods with data

When making a rest request with data in Angular, we:

- Supply the data as a parameter to the method call
- Subscribe to the observable to determine if the change was successful

```
this.http.post<User>('http://.../api/users', newUser);
```

Creating a meaningful UX

- Rest requests are asynchronous – when we submit the request, our code doesn't wait for the response
- We therefore need to establish a process to ensure a meaningful user experience, such as:
 - Disable the submit button so a user doesn't submit twice
 - Display a spinner / message to tell the user something is happening
 - When the result comes back display a message to the user to indicate the status (or navigate to a different page)

Activity – Allow a user to be edited

1 - Using the Use Edit form that we created earlier:

- Use the rest PUT method to send an update for a user to the server
- Display a “saving” message while we wait for the server response
- If the server responds with a valid response, display the message “user saved”, otherwise display the response received from the server

(note if you save a user with just spaces in the
firstname or surname field you will get an error)

2 - Create an “add user” function:

- Add an “add user” button to the user list page
- Use the same form to add a user (set the id to null)
- If a new user is successfully saved, navigate to the edit page for that user

Summary

- Rest methods with data
- Creating a meaningful user experience

Login, Security and Route Guards

Objectives

- Authentication & Authorization
- Storing user credentials
- State Management
- Making secure REST requests
- Route Guards

Authentication vs Authorization

- Authentication = who is our user?
- Authorization = what is our user allowed to do?
- We will rely on the back-end to authenticate a user and tell us their role
- We use their role to determine what they can do in the app
- The back-end must always validate that the user can perform the actions required (via the rest requests)

Storing user credentials

- It's a bad idea to store the user's username and password in memory / localStorage
- Most modern applications use JWTs – the server will provide us with a unique token to use with the user
- When we first authenticate a user, we use BASIC authentication
- On all subsequent REST requests, we use BEARER authentication

State Management

- When we have obtained a JWT from the server, we should consider storing this in localStorage so that it survives a browser refresh

Making secured REST requests

- To make a secure rest request we include the JWT in the header in the format:

{Authorization: Bearer ...}

- If the JWT has expired we should expect to receive back a 401 – in this case we can try and use the refresh token to get a new JWT...

Route Guards

- Route Guards allow us to alter navigation based on (for example) a user's status.
- Typically they are used to redirect users to a login page

Activity - Route Guards

- Create a second Route Guard that checks the user's role is admin, and if it isn't redirects them to a page that says they must have admin access to use this feature
- Protect the edit / add user pages with this route guard

Summary

- Authentication & Authorization
- Storing user credentials
- State Management
- Making secure REST requests
- Route Guards