Angular Overview

Fall 2026, Lecture xx.x.x

Today

- HTML & CSS & TS & SPA & What is Angular?
- Building Blocks - Components
- Information Handling - Data Binding
- Separation of Responsibilities - Services
- Navigation - Routing
- Reactivity - Observables
- Performance - Signals

Hypertext Markup Language (HTML)

- Language to describe a Web page structure
  - Information – text, tables, lists, paragraphs
  - Navigation – hyperlinks and buttons
  - Input – forms, input boxes, dropdowns, checkboxes
  - Media – audio, video, drawing
- .html – text file with HTML tags (markers)
  - Parsed by browsers

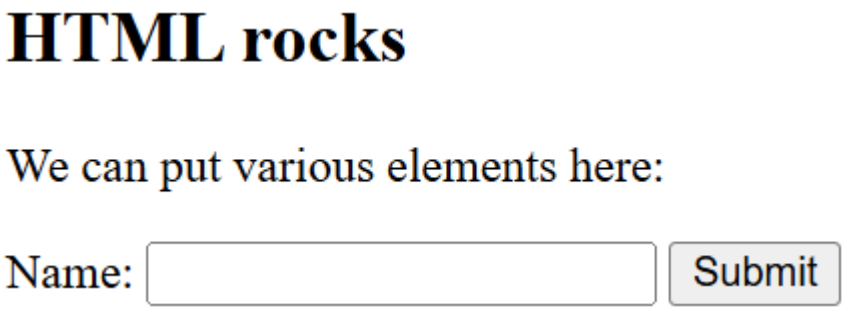
Hypertext Markup Language (HTML)

- Every HTML page has
  - **<html>** - root element, everything within is a HTML document
  - **<head>** - meta information for a browser (encoding, language, etc)
  - **<body>** - visible part of a HTML document

```
<html>
  <head>
    <!-- META INFORMATION -->
  </head>
  <body>
    <!-- PAGE CONTENT -->
  </body>
</html>
```

Hypertext Markup Language (HTML)

```html
<html>
  <body>
    <h2>HTML rocks</h2>
    <p>We can put various elements here:</p>
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">
    <button type="submit">Submit</button>
  </body>
</html>
```



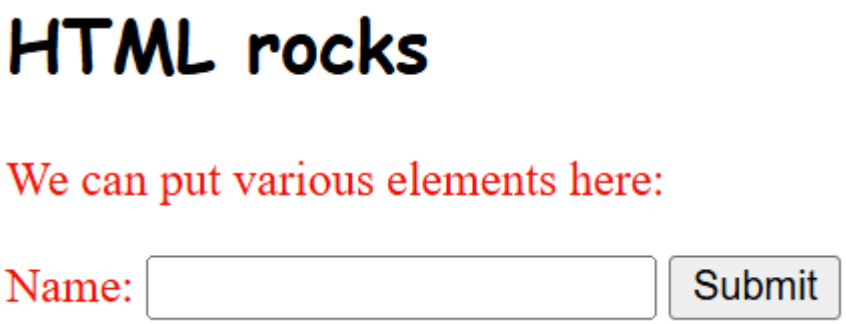
- 140+ tags in HTML5 - HTML Tutorial
  - Plus attributes

Cascading Style Sheets (CSS)

- Language to style the Web page content
  - HTML – what is on the page
  - CSS – how it looks (can modify placement)
- Can be applied to
  - tags
  - sets of elements
  - specific elements

Cascading Style Sheets (CSS)

```html
<html>
  <head>
    <style>
      h2 {
        font-family: "Comic Sans MS", cursive, sans-serif;
      }
      .danger {
        color: red;
        font-size: 16px;
      }
    </style>
  </head>
  <body>
    <h2>HTML rocks</h2>
    <p class="danger">We can put various elements here:</p>
    <label for="name" class="danger">Name:</label>
    <input type="text" id="name" name="name">
    <button type="submit">Submit</button>
  </body>
</html>
```



- 200+ CSS properties - CSS properties - CSS | MDN

Cascading Style Sheets (CSS)

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

```
<p style="color: red; font-size: 16px;">We can put various elements here:</p>
```

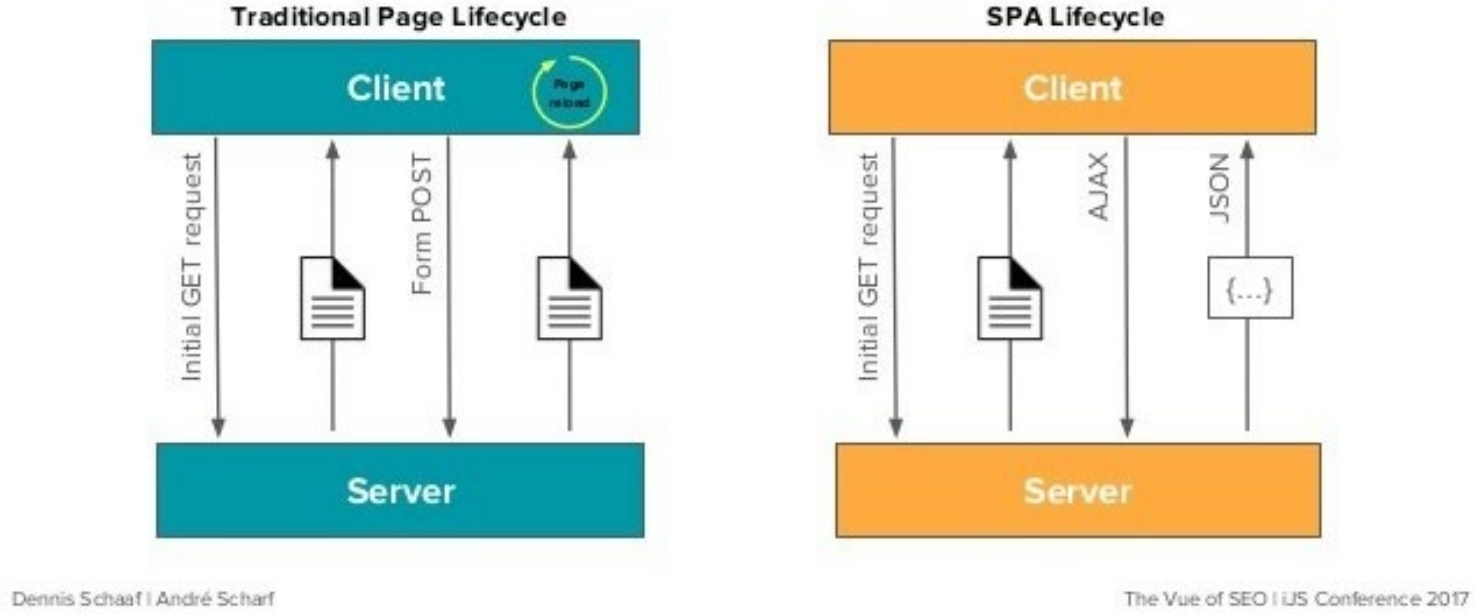
TypeScript

- Superset of JavaScript
  - JS code is valid TS code
  - Transpiled
  - Static typing
    - JS code violating typing is NOT valid TS code

```
let age = 25;
age = "twenty-five"; // Works in JS!

let age: number = 25;
age = "twenty-five"; // Error in TS!
```

```
function add(a: number, b: number): number {
  return a + b;
}

add(5, "10"); // Error in TS!
```

Single-Page Application (SPA)



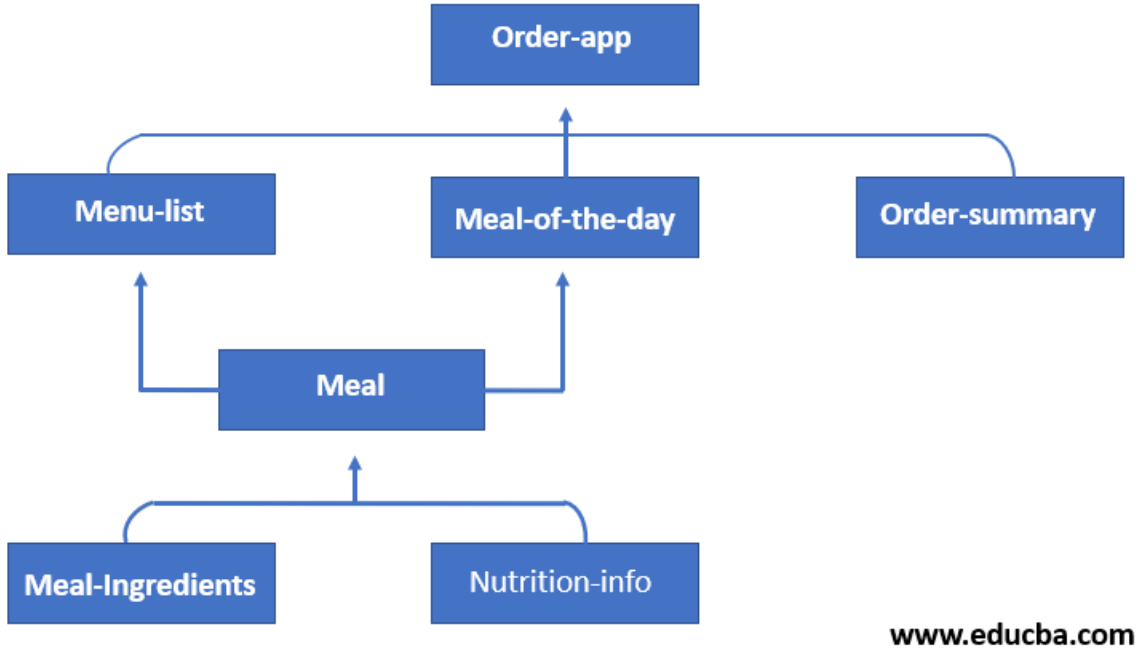
- Everything in one HTML file
  - Parts of the page are changed / data is loaded when needed
  - => less server workload
  - => UI is more responsive

What is Angular?

- Framework (DI) & Platform
- TypeScript – primary language
- HTML & CSS – template languages
  - Extended syntax (loops, conditions, etc) for HTML
- As a platform
  - Component-based framework
  - Integrated libraries (routing, forms, client-server communications, etc)
  - Developer tools (development, building, testing, updating)

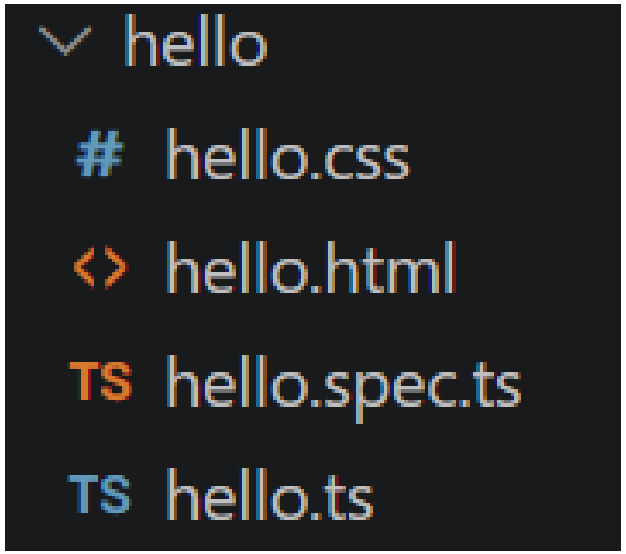
Angular Components

- Everything is a component! (no)
- Reusable
  - Can be used in different places of the application
- Can be placed into each other
- Responsible for their behavior
- Independent
  - Can be tested separately



Angular Components

- TS – TypeScript class
  - data & behavior & metadata
- HTML – template
  - extended with Angular expressions
- CSS – styling
- SPEC.TS – unit tests
  - ignored in this course



Angular Components

- @Component – metadata decorator
  - selector – HTML tag of the component
  - import – child components and so on
  - templateUrl – path to the template
  - styleUrl – path to the CSS file
- Constructor
  - can inject without variable declaration
- ngOnInit() – component lifecycle part
  - called when all dependencies are ready

```
import { Component, OnInit } from '@angular/core';
import { MirrorService } from '../mirror.service';

@Component({
  selector: 'app-hello',
  imports: [],
  templateUrl: './hello.html',
  styleUrl: './hello.css',
})
export class Hello implements OnInit {

  name: string | undefined = "Corwin of Amber";
  weapon: string | undefined;
  trumps: string[] = ["Caine", "Deirdre", "Julian"];

  constructor(private mirrorService: MirrorService) {
    this.weapon = "Grayswandir";
  }

  ngOnInit() {
    this.mirrorService.change();
  }

}
```

Angular Components

- Pure HTML is static
  - HAVE to predefine all items
- Expressions make it a template
  - Control flow • Angular
  - Variables in templates • Angular
  - Expression syntax • Angular

```
<ul>
  @for(trump of trumps; track trump) {
    <li>
      @if (trump === "Deirdre") {
        My favorite sister
      } @else {
        {{ trump }}
      }
    </li>
  }
</ul>
```

- Caine
- My favorite sister
- Julian

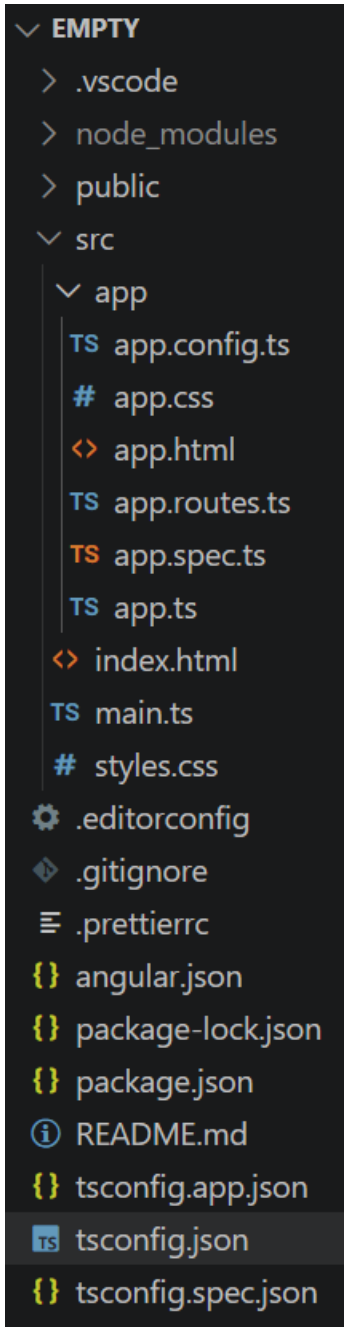
Angular Components

- To create
  - ng generate component [name]
  - ng g c [name]
  - Will create a component folder

Project Structure

- src/main.ts – entry point
- src/main.html – THE single page
- src/app/main.html – first component
- src/app/main.ts – first component
- src – components
- node_modules – dependencies

```
index.html > ...
<!doctype html>
<html lang="en">
<body>
  <app-root></app-root>
</body>
</html>
```



Data Binding

- Connects data in TS and HTML
  - quantum entanglement
  - text, image URL, variable state, objects, etc
- 3.5 types
  - Interpolation
  - Property binding
  - Event binding
  - Two-way data binding

Data Binding – Interpolation

- One-way (TS to HTML)
  - Evaluates an expression
  - Transforms into a string
  - Injects instead of {{ … }}

```
name: string | undefined = "Corwin of Amber";
weapon: string | undefined;
trumps: string[] = ["Caine", "Deirdre", "Julian"];

constructor(private mirrorService: MirrorService) {
  this.weapon = "Grayswandir";
}
```

```
{{ name }}'s favorite weapon is {{ weapon }}.
```

Corwin of Amber's favorite weapon is Grayswandir.

```
{{ 100 * 5 }} => 500

{{ getBalance() - 50 }} => some number

{{ name | uppercase }} => CORWIN OF AMBER
```

Data Binding – Property Binding

- One-way (TS to HTML)
  - HTML tag properties only
  - Injects a value instead of [...]
  - Variety of use
    - Toggle buttons / switches
    - Replace images / videos / etc
    - Share state between elements
    - etc

```
isBlind: boolean = true;
```

```
<button [disabled]="isBlind">Kill Eric</button>
```

```
<button disabled="true">Kill Eric</button>
```

Corwin of Amber's favorite weapon is Grayswandir.

- Caine
- My favorite sister
- Julian

Kill Eric

```
catUrl: string = "https://internet.com/cat.jpg";
```

```
<img [src]="catUrl">
```

```
<img src="https://internet.com/image.jpg">
```

Interpolation vs Property Binding – why?

- Can use interpolation to inject property values

```
<button disabled="{{ isBlind }}">Kill Eric</button>
```

- Interpolation converts values into strings
  - true => “true”
  - 42 => “42”
- Will it work?
  -
  -

Data Binding – Event Binding

- One-way (HTML to TS)
  - Notifies TS that something happened in HTML
  - Reacts on events
    - Key up / down
    - Mouse movements
    - Value change
    - Focus change
    - etc (dozens)

```
<button (click)="killEric()">Kill Eric</button>
```

```
killEric(): void {
  alert("Eric has been killed!");
}
```

Data Binding – Two-way Binding

```
weapon: string | undefined;
```

```
{{ name }}'s favorite weapon is <input [(ngModel)]="weapon" />
```

Corwin of Amber's favorite weapon is Grayswandir

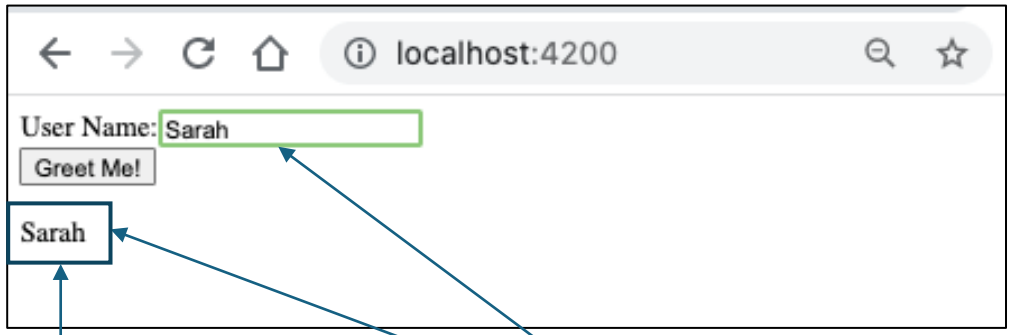
- Two-way (who knew?!)
  - Connects an input in HTML and a variable in TS
  - Updates go in both directions
- Property Binding + Event Binding
  - Syntax sugar

```
<input [(ngModel)]="weapon" />
<input [ngModel]="weapon" (ngModelChange)="weapon = $event">
```

Data Binding – Two-way Binding

Two-way data binding for “name” element

```
    Enter Your Names: <input type="text" [(ngModel)]="name"><br/>
    <button (click)="greet()">Greet Me!</button>

    <p>{{name}}</p>
</div>
```

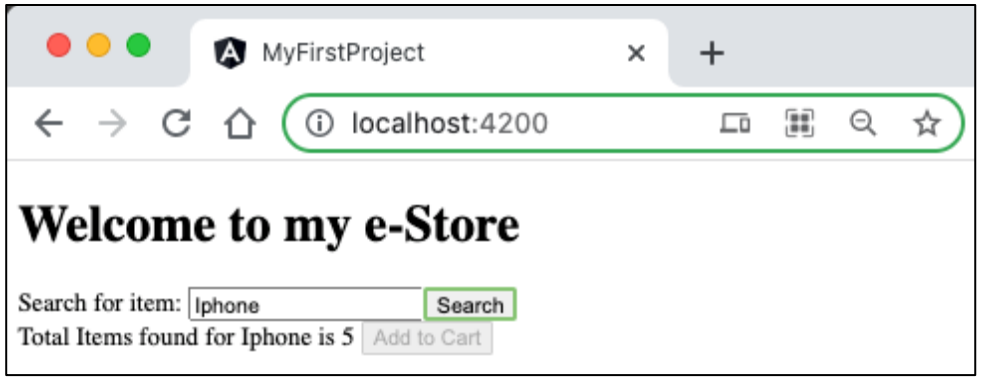


As you type in a new value for “name”, all references are immediately updated in template and component class

Data Binding – Two-way Binding

greet.component.ts

```ts
import { Component, OnInit } from '@angular/core';
import { LogService } from '../log.service';

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

  constructor(private logger: LogService) { }

  ngOnInit(): void {
  }

  title: string = "Welcome to my e-Store";
  isDisabled = true;
  item: string = "item";
  searchItem: string ="";
  numItems = 0;

  searchItems(): void {
    this.numItems = 5;
    this.searchItem = this.item;
  }
}
```



greet.component.html

```html
<h1 [innerText]="title"></h1>

<div>
    Search for item: <input [(ngModel)]="item">
    <button (click)="searchItems()">Search</button><br/>
    Total Items found for {{searchItem}} is {{numItems}}
    <button [disabled]="isDisabled">Add to Cart</button>
</div>
```

Event binding

Property binding

Interpolation

Two-way data binding

Services

- Singletons
  - What is a singleton?
- Contain
  - Data / state
  - Business logic
  - Utility methods
- Share data and functionality among components
  - Decreases coupling
  - Otherwise, components depend on each other more

Services

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

@Service()
export class LogService {

  log(msg: any): void {
    console.log(new Date() + ": " + JSON.stringify(msg));
  }

}
```

```
import { Component } from '@angular/core';
import { LogService } from '../log.service';

@Component({
  selector: 'app-logrus',
  imports: [],
  templateUrl: './logrus.html',
  styleUrl: './logrus.css',
})
export class Logrus {

  constructor(private logService: LogService) {}

  complete(): void {
    this.logService.log("Logrus is passed!");
  }

}
```

Services

- To create
  - ng generate service [name]
  - ng g s [name]
  - Will create [name].service.ts and [name].service.spec.ts

Routing

- SPA, but…
  - login page, profile page, dashboard page, data page,..
  - user needs to navigate and bookmark / share “page” URLs
- Changes the content (and the URL) when a user navigates around
- Creates a map “url – component”
  - nested mapping is possible
- Built-in

Routing

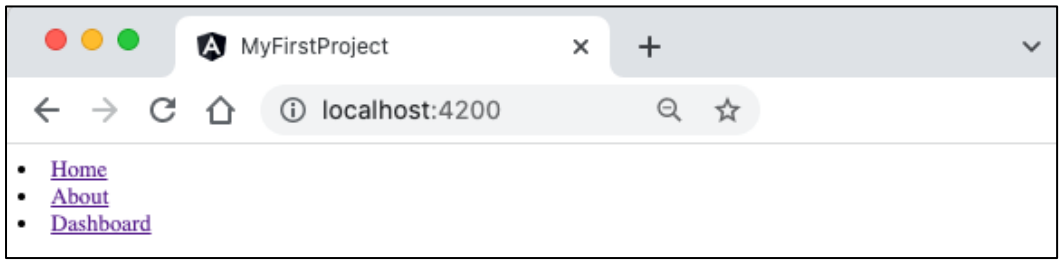
- app.routes.ts - URL mapping

```
import { Routes } from '@angular/router';
import { Hello } from './hello/hello';
import { Logrus } from './logrus/logrus';

export const routes: Routes = [
  { path: '', redirectTo: '/hello', pathMatch: 'full' },
  { path: 'hello', component: Hello},
  { path: 'logrus', component: Logrus}
];
```

- app.html – navigation
  - plus imports

```
<nav>
  <a routerLink="/hello">Hello</a>
  <a routerLink="/logrus">Logrus</a>
</nav>
<router-outlet></router-outlet>
```

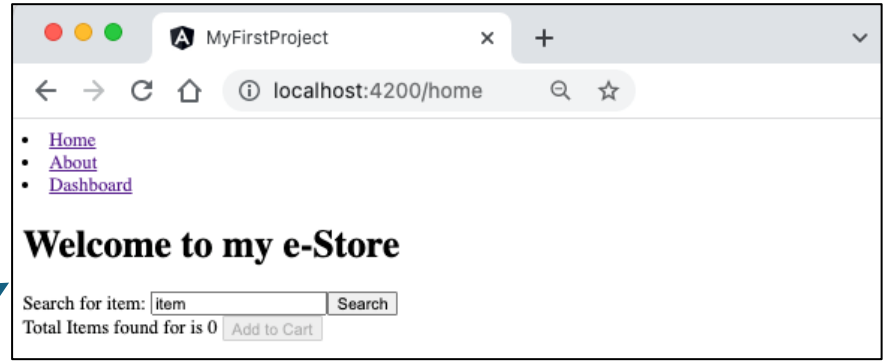
Routing

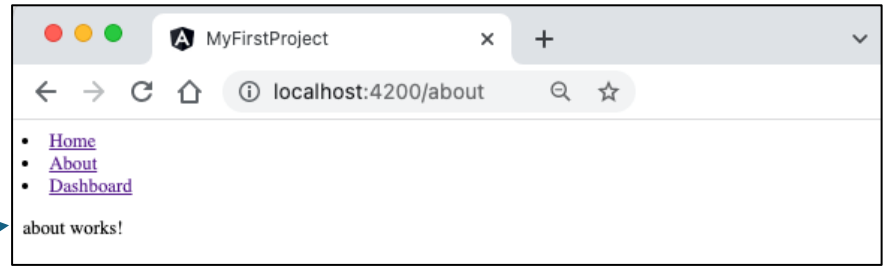


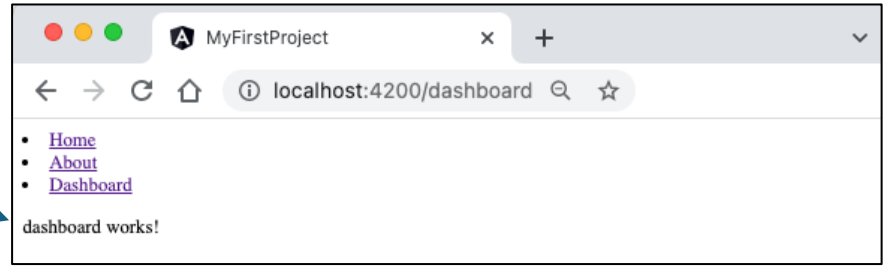
app.html

```
<div>
<nav>
  <li><a routerLink="home">Home</a></li>
  <li><a routerLink="about">About</a></li>
  <li><a routerLink="dashboard">Dashboard</a></li>
</nav>
  <router-outlet></router-outlet>
</div>
```

The <router-outlet> tells the router where to display routed views







Pull vs Push

- Pull
  - define a function
    - produces some value – producer
  - call the function and process the result
    - consumer
- Push
  - create a producer
    - something generating a value(-es)
  - define a consumer
    - something catching produced values

Pull vs Push

- Pull

```
function getData() {
  return "Hello";       // producer
}
const data = getData(); // consumer
console.log(data);      // consumer handles the data
```

- Push

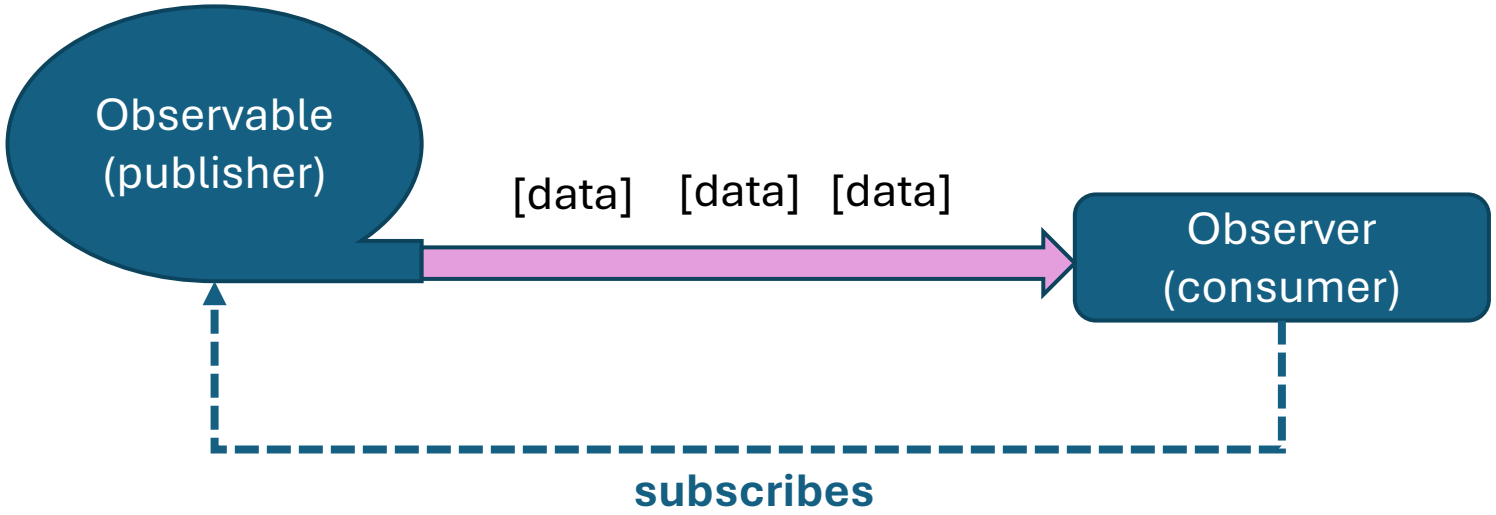
```
const promise = new Promise(resolve => {
  const data = "Hello";   // producer
  resolve(data);          // pushes the data
});
promise.then(data => {    // consumer is triggered when data is produced
  console.log(data);      // consumer handles the data
});
```

Streams

- Sequence of data / events
  - user printing a symbols in a search input
  - user clicking on a page
  - scheduled events (every 5 seconds)
  - server-sent events
  - etc
- Push
- Promise works for ONE event

Observables

- Push, but for a sequence



Observables

```
const observable = of(1, 2, 3, 4, 5); // producer
observable.subscribe({
  next: data => console.log(data),    // consumer handles the next item
  error: err => console.error(err),   // consumer handles the error
  complete: () => console.log('Observable completed') // no more items
});
```

```
const shorty = of(1, 2, 3, 4, 5);
shorty.subscribe(data => console.log(data));
```

Next – next value
Error – on the side of the observable, no further values
Complete – no more values

Observables

- Pre-process every value / event
  - Multiply each price by a constant
  - Process only specific types of events
  - Check duplications
  - etc

```
observable
  .pipe(
    map(x => x * 2),
    filter(x => x > 5)
  )
  .subscribe(result => console.log(result));
```

- Operators – chain of operators for every item
  - Observable 1 -> map -> Observable 2 -> filter -> Observable 3 -> observer
  - 100+ operators - https://rxjs.dev/guide/operators

Observables

- Hot vs Cold
  - Cold – data / event source is within an observable
    - Emits only if there is a subscriber
  - Hot – data / event source is outside of an observable
    - If there is no subscriber, emits anyway (lost)

```
const observable = new Rx.Observable<number>(observer => {
  observer.next(Math.random());
});
observable.subscribe(data => console.log(data));
```

```
const random = Math.random()
const observable = new Rx.Observable<number>(observer => {
    observer.next(random);
});
observable.subscribe(data => console.log(data));
```

Observables

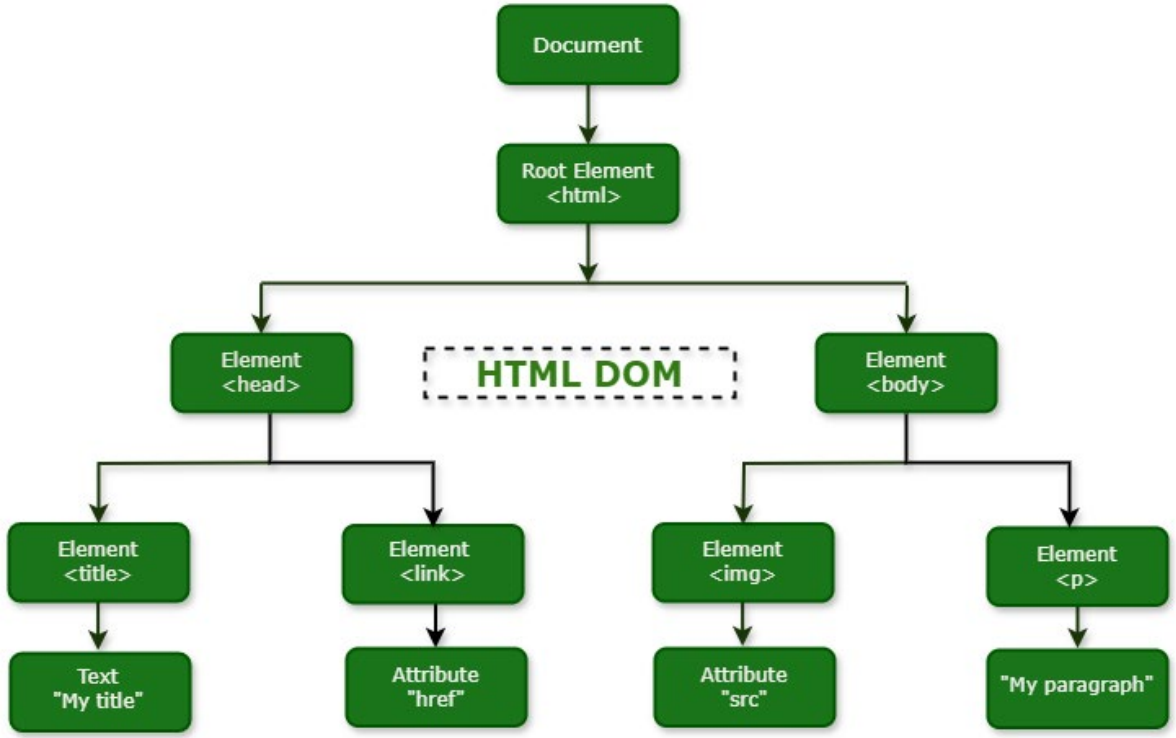
- Multiple subscribers
  - multiple components should react on some events
  - hot – all subscribers receive the same value
  - cold – each subscriber gets its own (separate observable executions)

```
const observable = new Rx.Observable<number>(observer => {
  observer.next(Math.random());
});
observable.subscribe(data => console.log(data)); // 5
observable.subscribe(data => console.log(data)); // 43
```

```
const random = Math.random()
const observable = new Rx.Observable<number>(observer => {
    observer.next(random);
});
observable.subscribe(data => console.log(data)); // 5
observable.subscribe(data => console.log(data)); // 5
```

Zone.js (Harry Potter and Data Binding)

- Document Object Model (DOM)
- ANY event / change happens:
  - go through ALL nodes
  - compare ALL values in HTML and TS
  - update if different
- Issues in real projects
  - a LOT of events / changes
  - large tree
  - performance overhead



Signals

```
protected x: number = 5;
protected y: number = 3;
protected z: number = this.x + this.y;
console.log(this.z); - ???
```

➢ 8

```
y = 5;
console.log(this.z); - ???
```

➢ 8

A lot of variables => a lot of manual updating =>

=> easy to forget something => bugs

```
protected x: Signal<number> = signal(5);
protected y: WritableSignal<number> = signal(3);
protected z: Signal<number> = computed(() => this.x() + this.y());
console.log(this.z());
```

➢ 8

```
this.y.set(5);
console.log(this.z());
```

➢ 10

Signals

- protected x: Signal<number> = signal(0);
- x – signal, not a number!
- console.log(x) -> () => signalGetFn(node)
- **x()** <- get a value of the signal x

- **protected** – component template (HTML) cannot reach private fields (Angular 19+)
  - can keep **private** if is not used in HTML
  - {{ x() }} <= works if x is NOT private
  - can use private if only referenced in the class

Signals

- Signal – value cannot be changed
- WritableSignal – value can be changed
- Computed – read only, by the value will be recalculated
  - lazy – calculated only when called
  - memoized – recalculated only if a dependency changes
  - can have a complex expression

```
computed(() => {
        if (condition())
                return ${value()};
        else
                return “go away”;
}
```

Signals

- Trick to expose a signal and prevent modifications

```
const writable: WritableSignal<number> = signal(10);
const readOnly: Signal<number> = writable;
readOnly(); // 10
readOnly.set(20); // error
writable.set(20); // still works
```

- More types and nuances online

Change Detection

- Angular 21+ is zoneless
- Uses signals to track changes
- Knows precisely what changed
  - no full tree traversal
- Use signals for data binding

Pipes

- Have `protected name: string = "Corwin of Amber";`
- Want Name: CORWIN OF AMBER
- Don’t want to change the variable
  - used in different places
- Use pipes `Name: {{ name | uppercase }}`
  - can chain them
  - can create custom ones
  - there are built-in pipes - Pipes • Angular

Modules

- 1500 components in the app - how to manage?
- Modules – include components, services, pipes, etc
- Typically decomposed based on functionality, e.g.,
  - UserModule (profile management, user settings, etc)
  - ProductModule (details, listings, reviews, etc)
  - OrderModule (create, view, track, history, etc)