Angular training

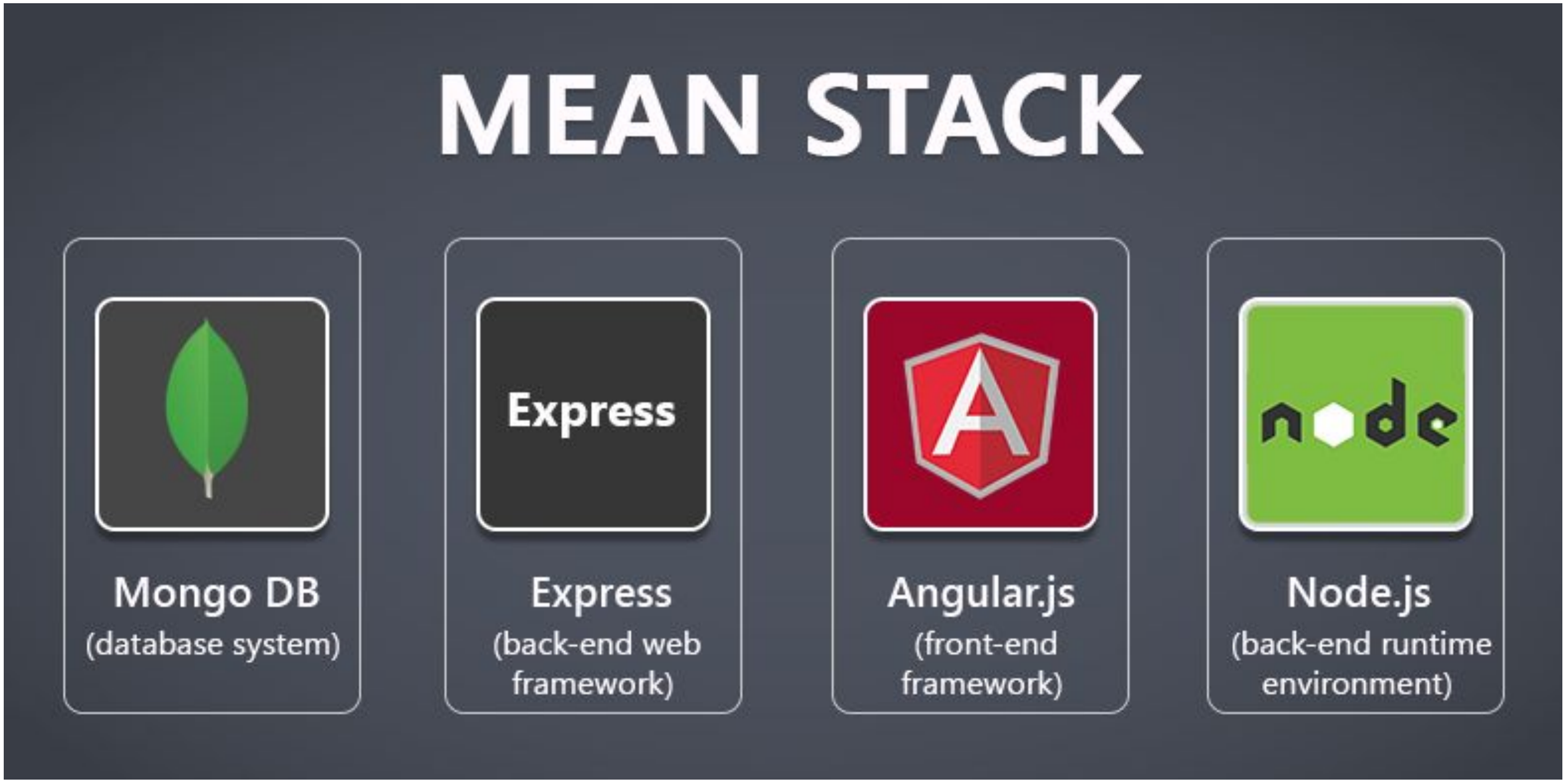
Introduction to Angular


Let's
Get
Started!

Angular

- **Comprehensive framework** for building scalable and robust web applications.
- **Developed and maintained by Google**, ensuring long-term support and continuous improvements.
- **Built with TypeScript**, offering strong typing and modern JavaScript features.
- **Component-based architecture** that promotes modularity and reusability.
- **Foundation for mobile app development** using frameworks like **Ionic**.
- **A key part of the MEAN Stack** (MongoDB, Express, Angular, Node.js), commonly used for full-stack JavaScript applications.

📘 **Official documentation**: https://angular.io


MEAN STACK
Express
node
Mongo DB
(database system)
Express
(back-end web framework)
Angular.js
(front-end framework)
Node.js
(back-end runtime environment)

What is a stack?

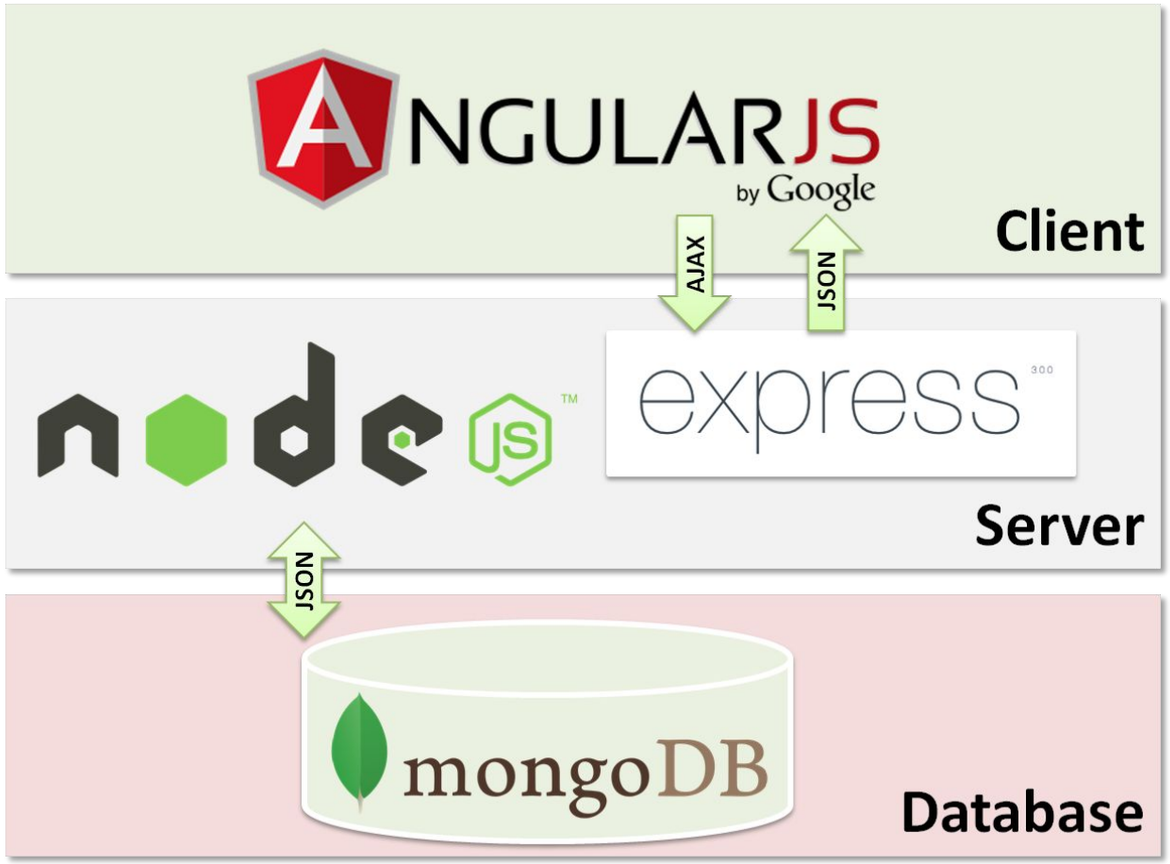
	MEAN / MERN/ MEVN - Javascript	LAMP - Different language	Java Stack	Python
Database	MongoDB	MySQL	Postreql / Oracle	Postresql / MySQL
Backend web framework	Express JS	Apache / Nginx	Spring Boot	Django / Flask
Front end web framework	Angular / React / VueJS	PHP, Python, Perl	JSP, JSF , or SPA	SPA
Backend Runtime environment	NodeJs	Linux	Tomcat server	Linux
Mobile app	Ionic / React Native		Native Android (Java) / IOS	
Data				Can do data science easily

Case Study AirBnB

Language	Java, JS
Database - Normally we will have 2 databases - Transactional, Data processing, Caching	MySQL, RDS , Redis, Hadoop
Backend web framework	
Front end web framework	React, Rails
Backend Runtime environment	EC2 (Linux)
Mobile app	
Data	

Based on https://stackshare.io
https://stackshare.io/airbnb/airbnb/

What is MEAN?



MEAN is an opinionated fullstack javascript framework - which simplifies and accelerates web application development.

Why MEAN?

- **MEAN** stands for **MongoDB, Express.js, Angular, and Node.js** — a full-stack JavaScript framework for developing web applications.
- **Angular** plays the role of the **front-end framework**, handling the user interface and client-side logic.
- Being **TypeScript-based**, Angular brings structure, scalability, and maintainability to large-scale front-end applications.
- Angular seamlessly **communicates with Express.js and Node.js** on the server side using **HTTP and JSON**, ensuring a smooth full-stack JavaScript development experience.
- Its **component-based architecture** and **powerful tooling** make Angular a robust choice for building dynamic, responsive web interfaces in MEAN applications.
- Since all technologies in MEAN use **JavaScript (or TypeScript)**, there's **consistency across the entire stack**, making development faster and easier to manage.

Data stored in MongoDB (example)


```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

MongoDB query example

Find all contacts with at least one work phone or hired after 2014-02-02

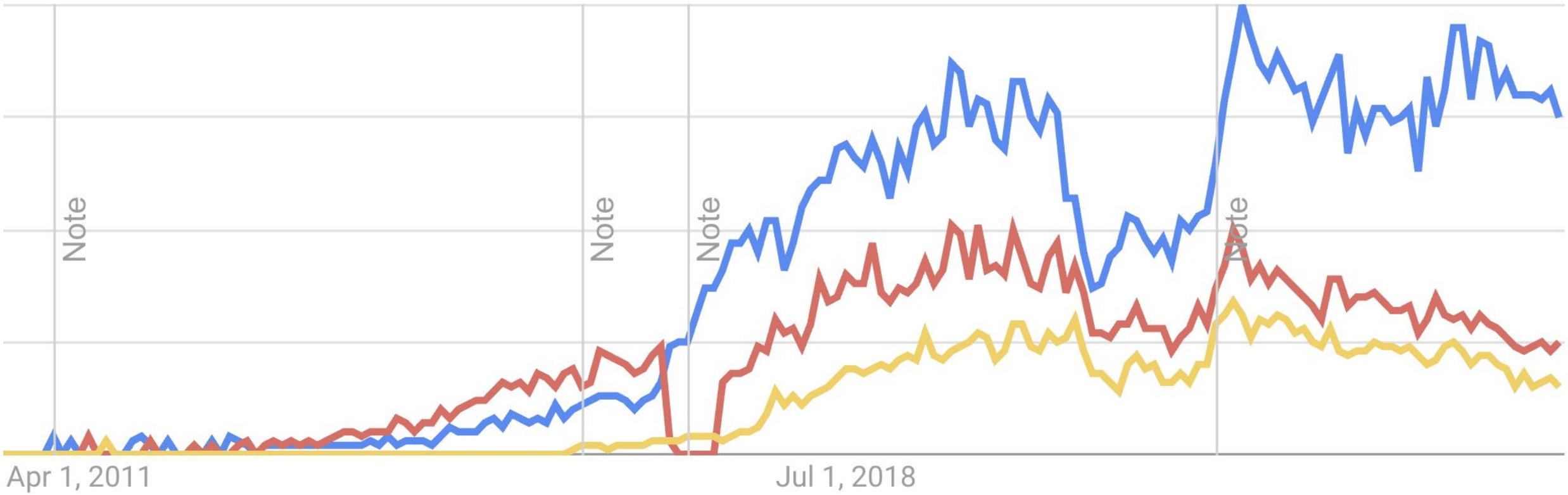
SQL

```
select A.did, A.lname, A.hiredate, B.type,
B.number from contact A left outer join phone
on (B.did = A.did) where b.type = 'work' or
A.hiredate > '2014-02-02'::date
```

MongoDB CLI

```
db.contacts.find({"$or": [
{"phones.type":"work"},
{"hiredate": {"$gt": new ISODate("2014-02-02"
]});
```

React vs Vue vs Angular



Single Page Application

- We don’t refresh the entire page, normally will only have one html which is index.html and all the others are JS files.
- Three main technology: Angular, ReactJs and VueJS

Angular	**ReactJS**	**VueJS**
By Google	By Facebook	
Full Framework - Include HTTP call	Library -Have to use JS method of calling HTTP	Library -Have to use JS method of calling HTTP
Bidirectional (will see tomorrow)	Unidirectional	Unidirectional
Two way data binding	One way data binding	One way data binding
Moderate	Hard	Easy - might even replace Jquery
	Next.js (https://nextjs.org/)	Laravel + VueJS (https://inertiajs.com/)

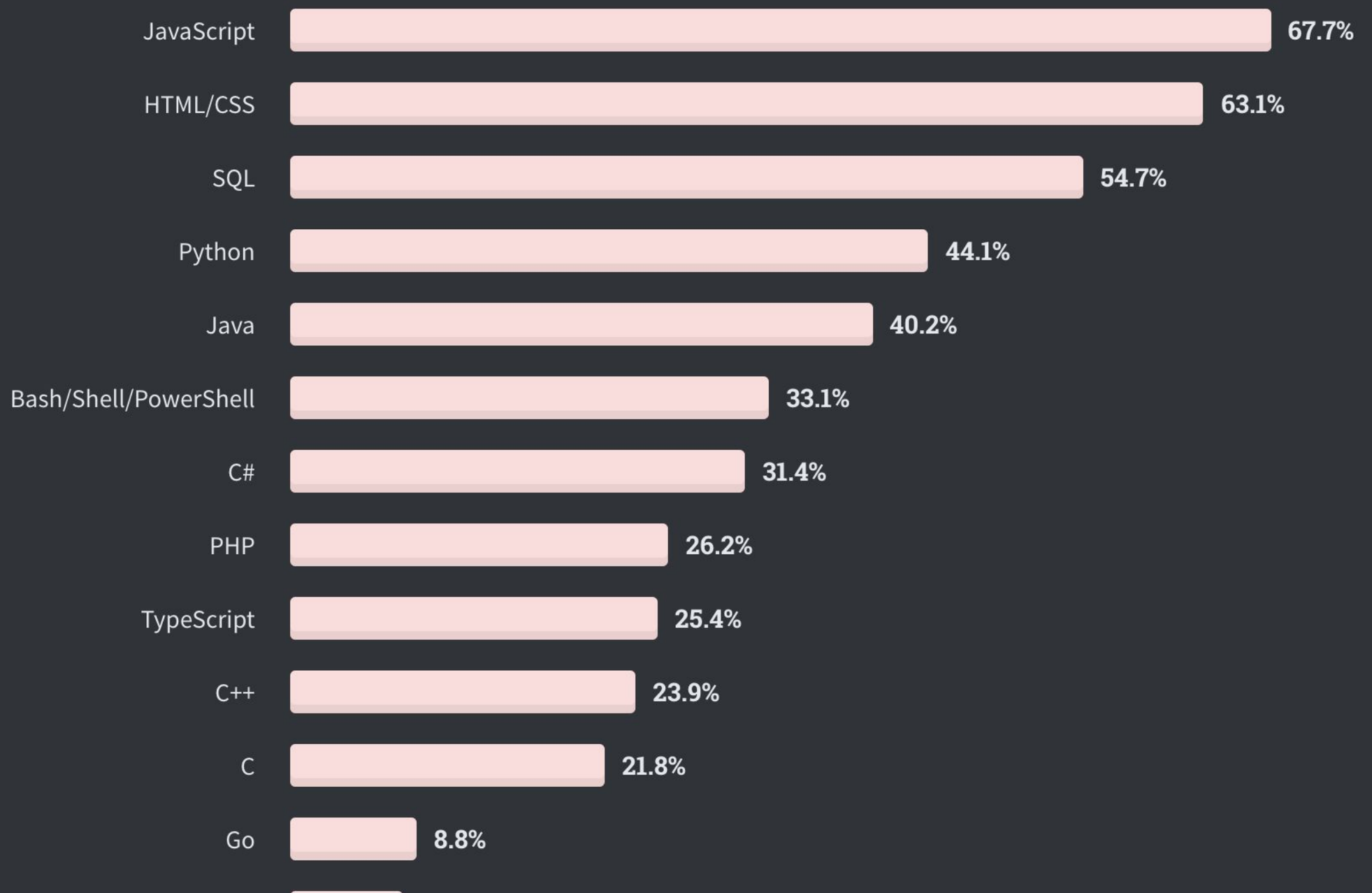
Difference between library & Framework

	Angular	React	Vue
UI Library	Imperative (how to do)	Declarative (what to do) , (map, reduce, filter)	Imperative (how to do)
HTTP Request	Built in HTTP Module/library	JS fetch axios	JS Fetch axios
Routing	Built in Angular Router	React-Router	Vue-Router
FIle structure	Complex	Simple (we can define how to do)	Simple (we can define how to do)

https://developer.mozilla.org/my/

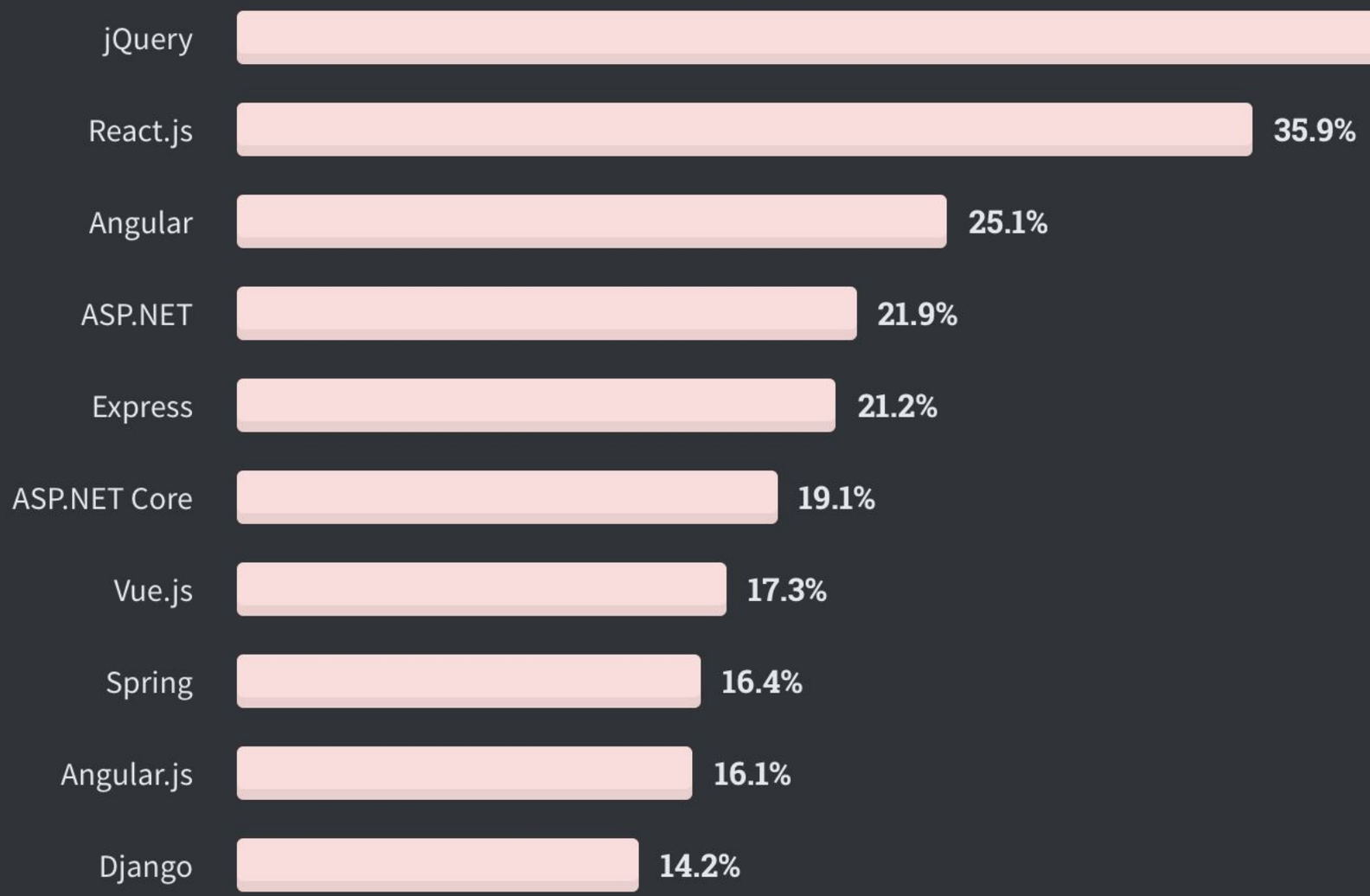
All Respondents | Professional Developers

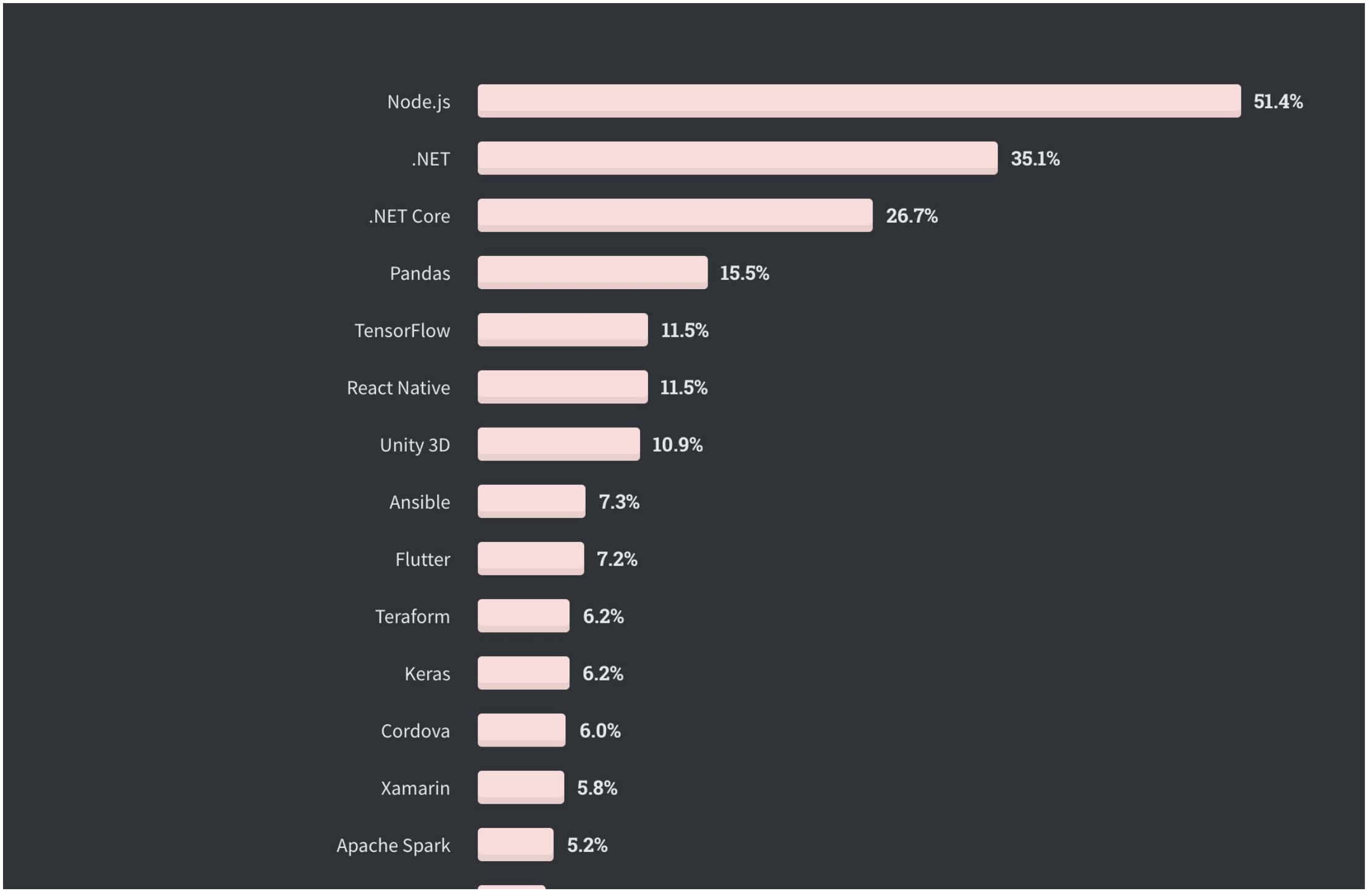
57,378 responses; select all that apply



All Respondents | Professional Developers

42,279 responses; select all that apply




Node.js
51.4%
.NET
35.1%
.NET Core
26.7%
Pandas
15.5%
TensorFlow
11.5%
React Native
11.5%
Unity 3D
10.9%
Ansible
7.3%
Flutter
7.2%
Teraform
6.2%
Keras
6.2%
Cordova
6.0%
Xamarin
5.8%
Apache Spark
5.2%

Application and Data

Application and Data

STACK LAYER

- **Application and Data**
- Utilities
- DevOps
- Business Tools

TOP 10 TOOLS & SERVICES

1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

JavaScript

DevOps

DevOps

STACK LAYER

- Application and Data
- Utilities
- **DevOps**
- Business Tools

TOP 10 TOOLS & SERVICES

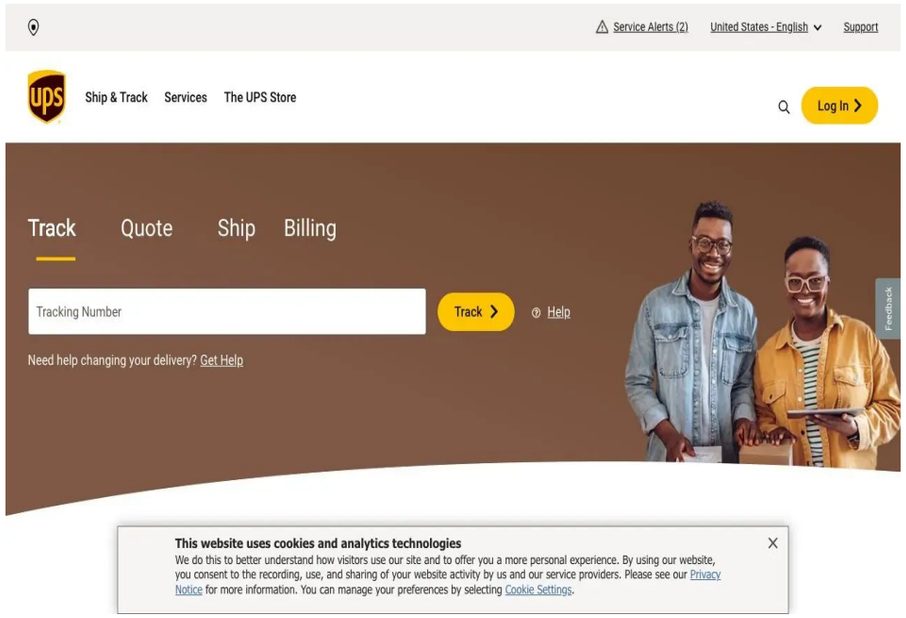
1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

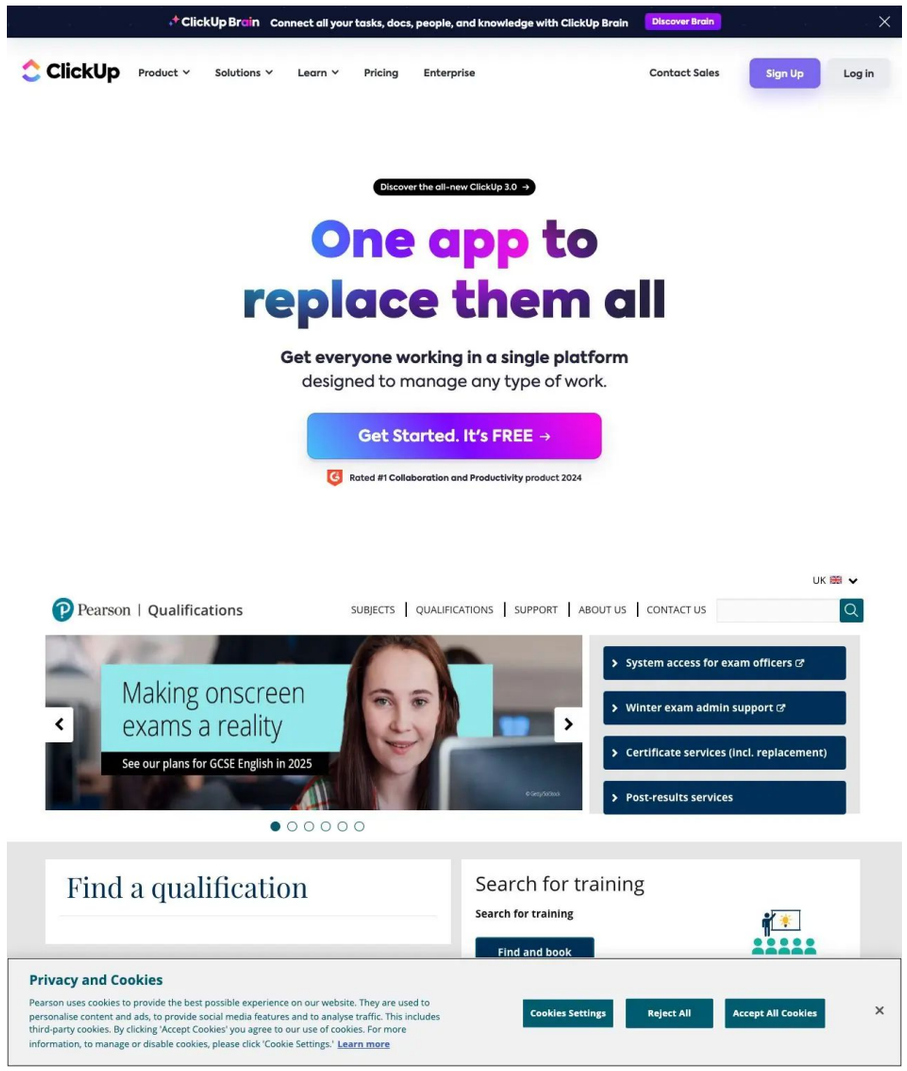
LAMP Stack

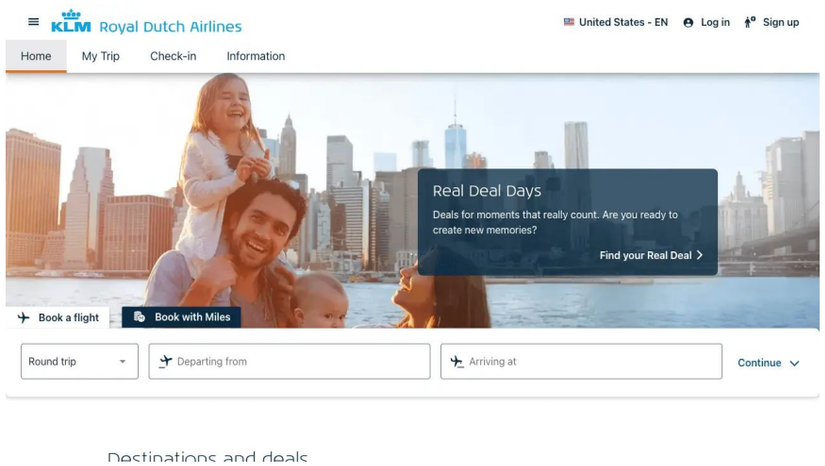
ANGULAR KEY FEATURES

- Full Framework
  - View
  - Model
  - Services
  - Routing
  - Http Requests
- Easy to use
- Scalable
- Fast
- Animations

Who uses Angular?

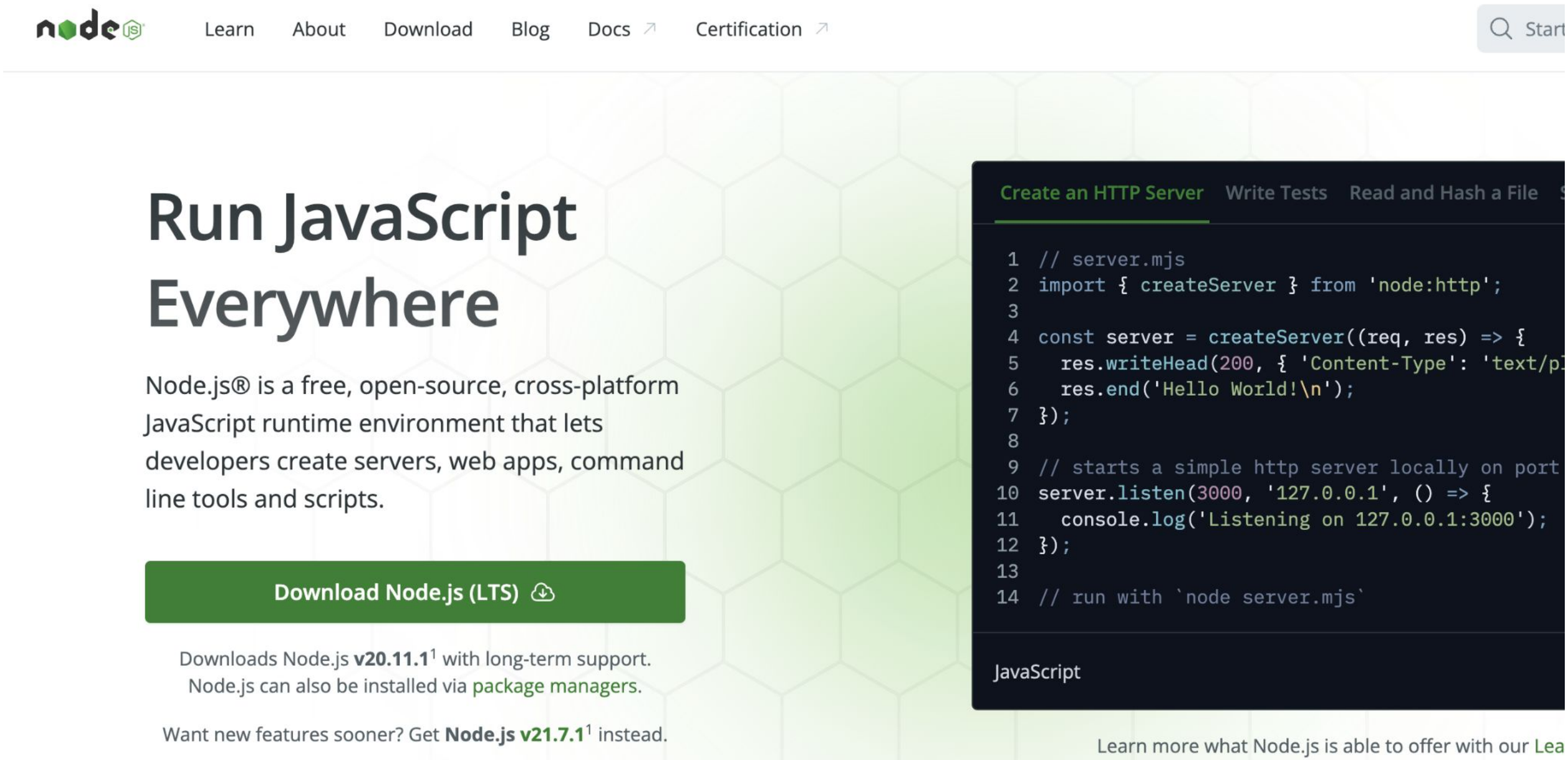






https://www.madewithangular.com/sites/

INSTALLATION AND GET STARTED



INSTALLATION AND GET STARTED

- Installing Angular
  - npm install -g @angular/cli (you might need sudo if you are on mac)
- Creating new Angular project
  - ng new hello-world
- Open in browser
  - cd hello-world
  - ng serve --open

Verify your Angular version

- ng --version

```
Angular CLI: 19.2.8
Node: 20.18.2
Package Manager: npm 10.8.2
OS: darwin arm64

Angular:
...

Package                         Version
---------------------------------------------------------
@angular-devkit/architect       0.1902.8 (cli-only)
@angular-devkit/core            19.2.8 (cli-only)
@angular-devkit/schematics      19.2.8 (cli-only)
@schematics/angular             19.2.8 (cli-only)
```

```
✓ Which stylesheet format would you like to use? CSS             [ https://developer.mozilla.org/docs/Web/CSS
                                                ]
✓ Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
CREATE hello-world/README.md (1473 bytes)
CREATE hello-world/.editorconfig (314 bytes)
CREATE hello-world/.gitignore (587 bytes)
CREATE hello-world/angular.json (2610 bytes)
CREATE hello-world/package.json (1004 bytes)
CREATE hello-world/tsconfig.json (915 bytes)
CREATE hello-world/tsconfig.app.json (424 bytes)
CREATE hello-world/tsconfig.spec.json (434 bytes)
CREATE hello-world/.vscode/extensions.json (130 bytes)
CREATE hello-world/.vscode/launch.json (470 bytes)
CREATE hello-world/.vscode/tasks.json (938 bytes)
CREATE hello-world/src/main.ts (250 bytes)
CREATE hello-world/src/index.html (296 bytes)
CREATE hello-world/src/styles.css (80 bytes)
CREATE hello-world/src/app/app.component.css (0 bytes)
CREATE hello-world/src/app/app.component.html (19903 bytes)
CREATE hello-world/src/app/app.component.spec.ts (931 bytes)
CREATE hello-world/src/app/app.component.ts (287 bytes)
CREATE hello-world/src/app/app.config.ts (310 bytes)
CREATE hello-world/src/app/app.routes.ts (77 bytes)
CREATE hello-world/public/favicon.ico (15086 bytes)
```

```
[wanmuz@Wans-MacBook-Air hello-world % ng serve
Component HMR has been enabled.
If you encounter application reload issues, you can manually reload the page to bypass HMR and/or disable this
 feature with the `--no-hmr` command line option.
Please consider reporting any issues you encounter here: https://github.com/angular/angular-cli/issues

Initial chunk files | Names         |  Raw size
polyfills.js        | polyfills     |  90.20 kB |
main.js             | main          |  47.23 kB |
styles.css          | styles        |  95 bytes |

                    | Initial total | 137.53 kB

Application bundle generation complete. [1.229 seconds]

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
  ➜  Local:   http://localhost:4200/
  ➜  press h + enter to show help
```

localhost:4200

localhost:4200

uery Pagination... | Home | Wix.com | Download Clash of... | Gmail | YouTube | Maps | Instagram

Angular

Hello, hello-world

Congratulations! Your app is running. 🎉

Explore the Docs

Learn with Tutorials

CLI Docs

Angular Language Service

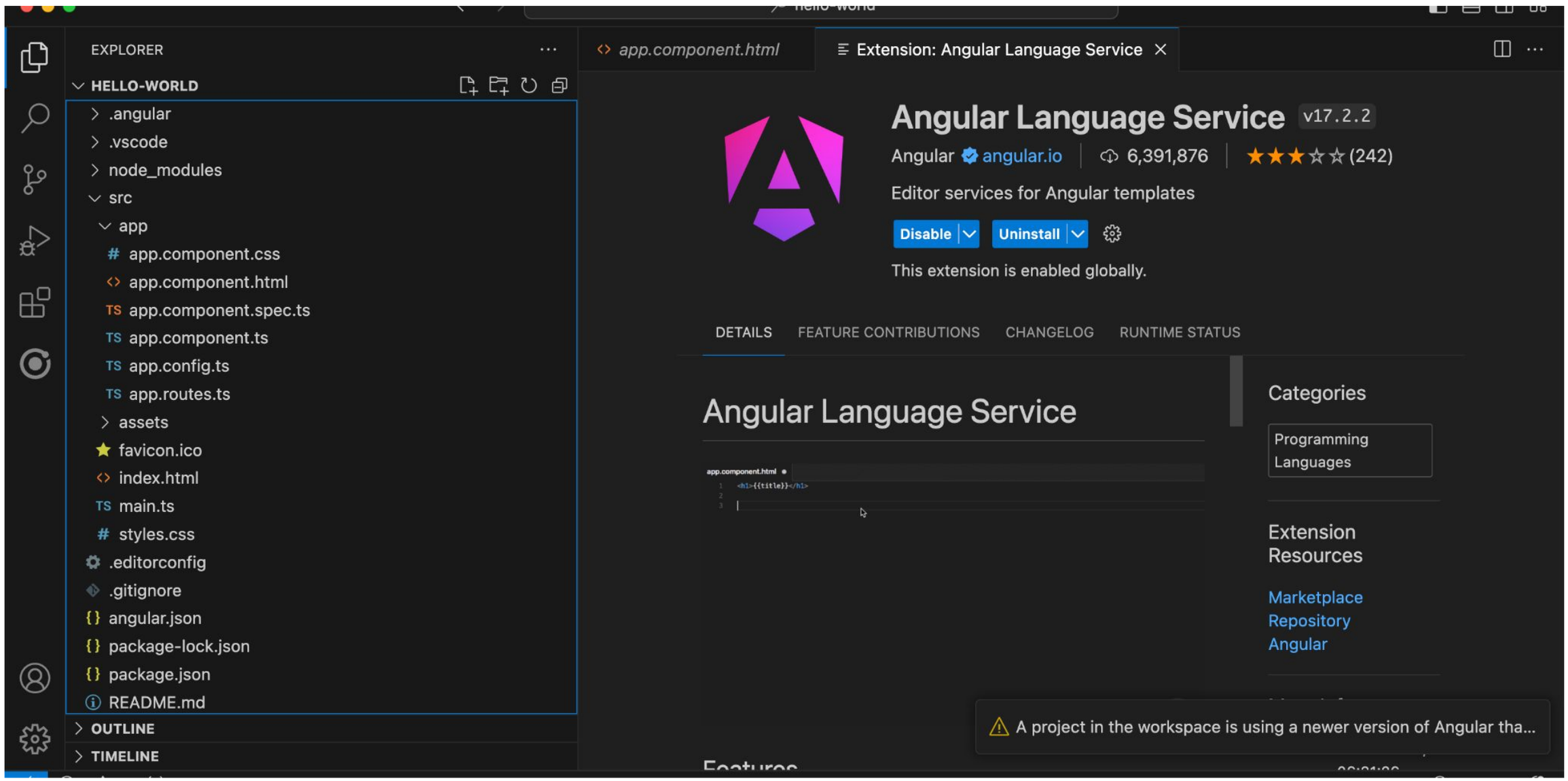
Angular DevTools

Open the project

- Open Visual Studio Code
- Select Open Folder
- Select the folder angular-training/hello-world
- Press Open Folder
- Look for src/app/app-component.html

https://code.visualstudio.com/

Open project with VS Code



Angular folder structure

`.vscode`: Includes VS Code configuration files
•`node_modules`: Includes npm packages needed for development and running the Angular application
•`src`: Contains the source files of the application
• `.editorconfig`: Defines coding styles for your editor
• `.gitignore`: Specifies files and folders that Git should not track
• `angular.json`: The main configuration file of the Angular CLI workspace
• `package.json` and `package-lock.json`: Provide definitions of npm packages, along with their exact versions, which are needed to develop, test, and run the Angular application
• `README.md`: A README file that is automatically generated from the Angular CLI
• `tsconfig.app.json`: TypeScript configuration that is specific to the Angular application
• `tsconfig.json`: TypeScript configuration that is specific to the Angular CLI workspace
• `tsconfig.spec.json`: TypeScript configuration that is specific to unit tests

Angular folder structure(2)

app: Contains all the Angular-related files of the application. You interact with this folder most of the time during development.

•assets: Contains static assets such as fonts, images, and icons.

• favicon.ico: The icon displayed in the tab of your browser, along with the page title.

• index.html: The main HTML page of the Angular application.

• main.ts: The main entry point of the Angular application.

•styles.css: Contains application-wide styles. These are CSS styles that apply globally to the Angular application. The extension of this file depends on the stylesheet format you choose when creating the application.

Sample code in Angular

app.component.css: Contains CSS styles specific for the sample page

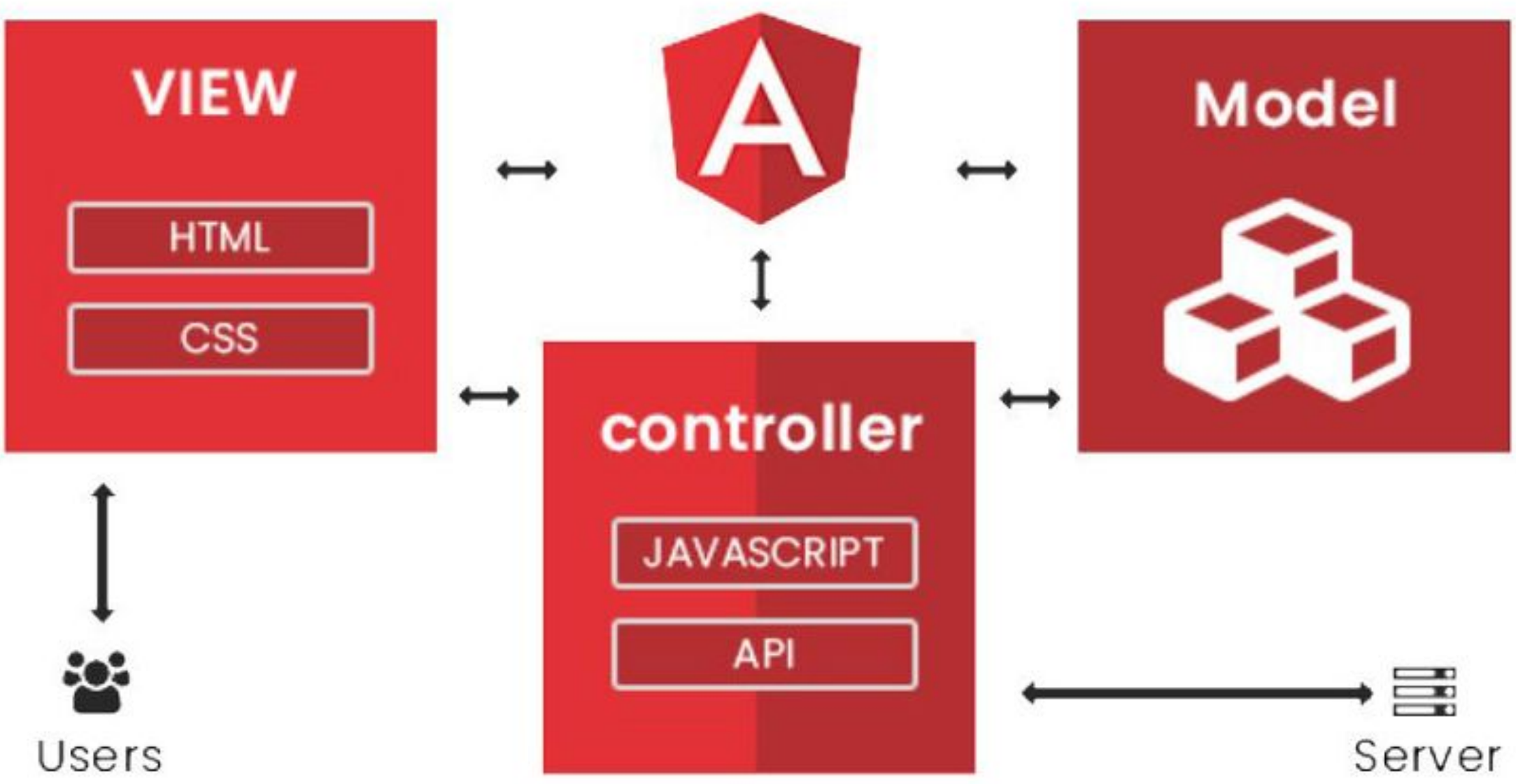
- app.component.html: Contains the HTML content of the sample page

- app.component.spec.ts: Contains unit tests for the sample page

- app.component.ts: Defines the *presentational logic* of the sample page

- app.module.ts: Defines the main **module** of the Angular application

MVC in Angular



Component in Angular

Files prefixed with `app.component` define an Angular component.

In Angular, a **component** is responsible for controlling a specific part of the web page by linking **presentation logic** (written in TypeScript) with the **HTML template** that defines the view.

Every Angular application has at least one **root component**, conventionally named `AppComponent`, which serves as the main entry point for rendering content on the page.

index.html

Every Angular application includes a main HTML file named **`index.html`**, located in the **`src`** folder. This file contains the following `<body>` element:

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

The custom `<app-root>` tag represents the **root component** of the Angular application (typically `AppComponent`). It acts as a **placeholder** where Angular dynamically renders the template and logic of the root component. This is the starting point of the application's UI.

Hello World

Go to app.component.ts and change the title to ‘My First app’

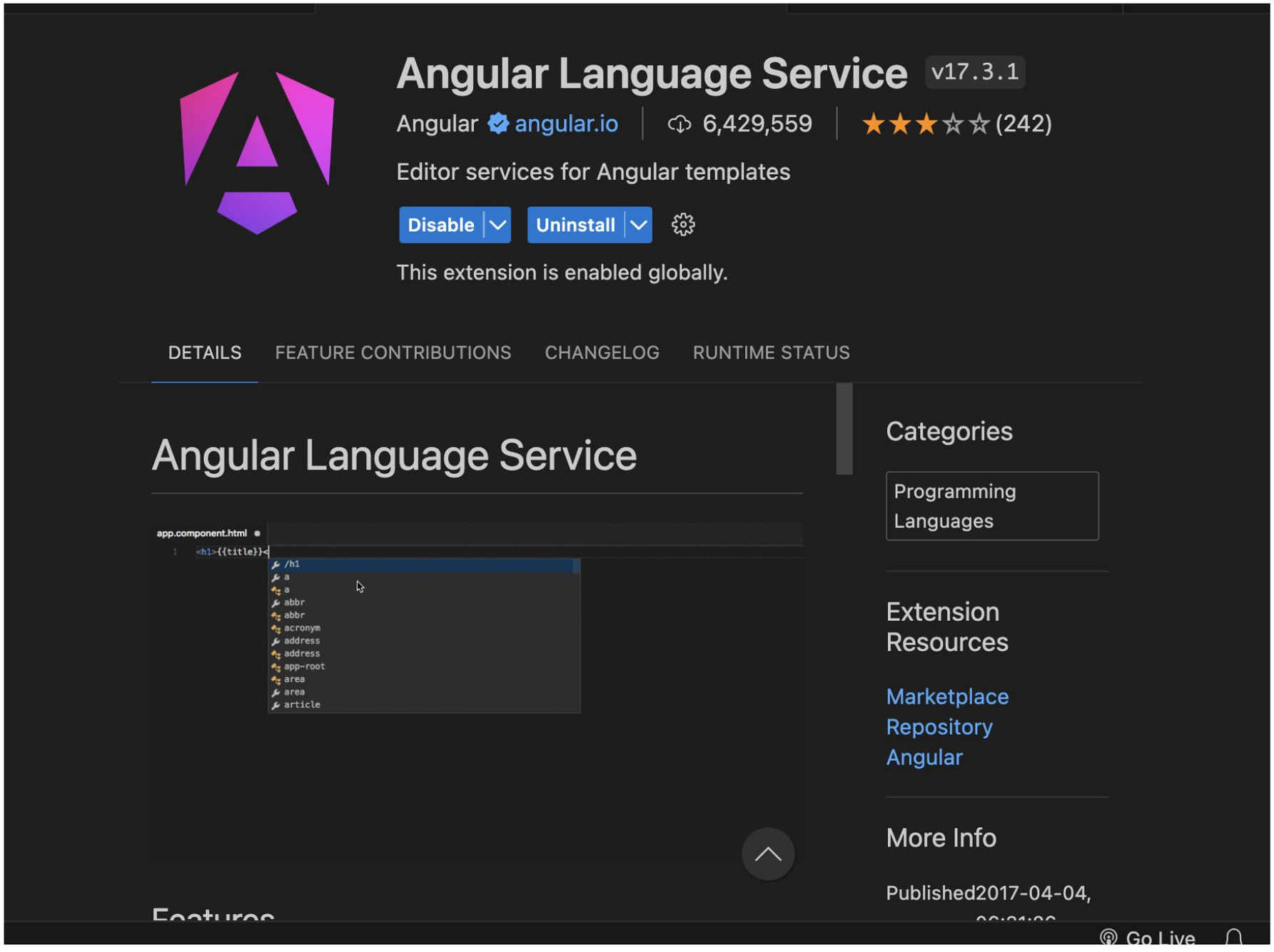
```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'My First App';
}
```

Hello World

Open app.component.html and verify in line 228

```
223          <stop offset="1" stop-color="#7702FF" />
224        </linearGradient>
225        <clipPath id="a"><path fill="#fff" d="M0 0h982v239H0z" />
226      </defs>
227    </svg>
228    <h1>Hello, {{ title }}</h1>
229    <p>Congratulations! Your app is running. 🎉</p>
230  </div>
231  <div class="divider" role="separator" aria-label="Divider"></di
232  <div class="right-side">
233    <div class="pill-group">
```

Angular Extension



Discover autocorrect VSCode

Add the following in app.component.ts

```
})
export class AppComponent {
  title = 'My First App';
  description = 'This is my first Angular app';
}
```

In app.component.html verify the autocorrect:

```
    </svg>
    <h1>Hello, {{ title }}</h1>
    <p>{{description}}</p>
```

Angular Modules

Angular modules are containers for a particular block of code that adheres to the same functionality.

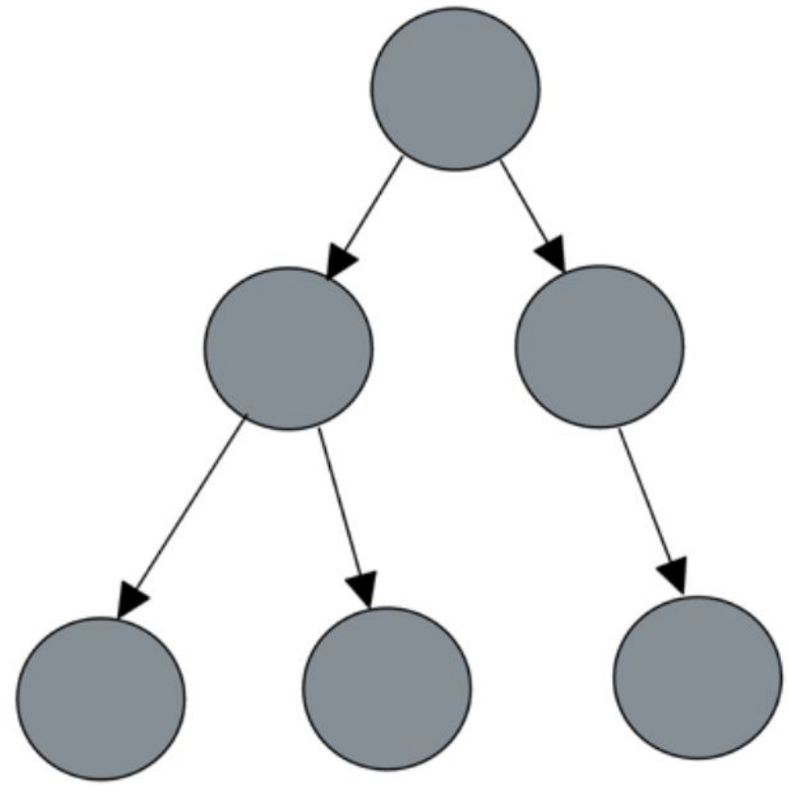
An Angular module is dedicated to an application domain, such as orders or customers for an e-shop application, or a user workflow, such as order checkout or user registration.

Generally, it addresses a particular set of capabilities that an application can have.

The main advantage of the Angular module architecture is that it scales better and is easier to test.

Component

Components are the fundamental building blocks of an Angular application.



Each component controls a specific part of the user interface, known as a **view** — for example, a product list, navigation bar, or checkout form.

Components handle the **presentational logic**, connecting the application's data with its visual structure via templates.

Angular organizes components in a **hierarchical tree structure**, allowing parent and child components to **interact through inputs, outputs, and services**, enabling modular and scalable application design.

Structure of an Angular Component

In Angular, the `@Component` decorator is used to define and configure a **component**. It includes several key metadata properties:

- **`selector`**: A CSS selector that tells Angular where to insert the component in the DOM. Angular looks for a matching HTML tag and renders the component at that location.

- **`templateUrl`**: Specifies the path to an external HTML file that contains the component's template.

  Alternatively, you can use the **`template`** property to define the HTML inline.

- **`styleUrls`**: An array of paths pointing to external CSS files used for styling the component.

  Alternatively, you can use the **`styles`** property to define styles inline.

Structure of an Angular Component

```
@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
```

GENERATING NEW COMPONENT

- Generate new component
  - ng g component product-list --standalone
  - ng g component header --standalone
  - ng g component footer --standalone
- Call the component in app.component.html

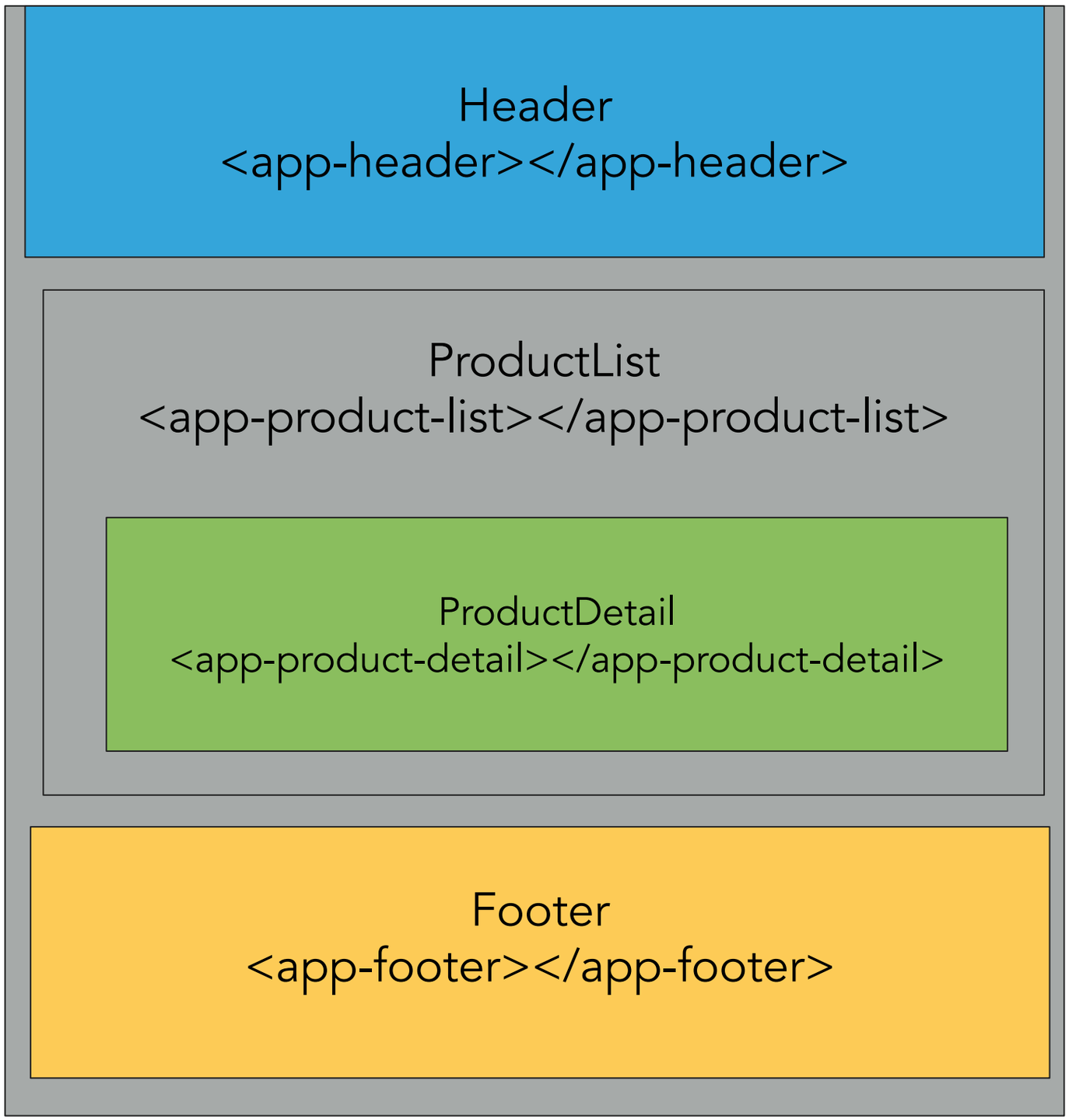
```
Go to component
1   <app-product-list></app-product-list>
```

Add the component as an import in app.component.ts

Importing the component in app.component.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet,ProductListComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'My First app';
  description = 'Hello world';
}
```

Create the following structure



String interpolation

Interpolation in Angular is a technique to **bind component data to the HTML template** using double curly braces (`{{ ... }}`).

It allows you to **display dynamic values** (e.g., strings, numbers, expressions) in the UI.

```
<span>{{title}}</span>
```

Property binding

An alternative way to perform interpolation is to bind the title property to the innerText property of the span HTML element:

```
<span [innerText]="title"></span>
```

We bind to the Document Object Model (DOM) property of an element.

The property inside square brackets is called the target property and is the property of the DOM element into which we want to bind.

The variable on the right is called the template expression and corresponds to the title property of the component.

Example in class [innerHTML]

product-list.component.ts

```ts
export class ProductListComponent {

  Imagine I get this from the internet
  welcome = 'Welcome to <strong>LazadaMall</strong>';
```

product-list.component.html

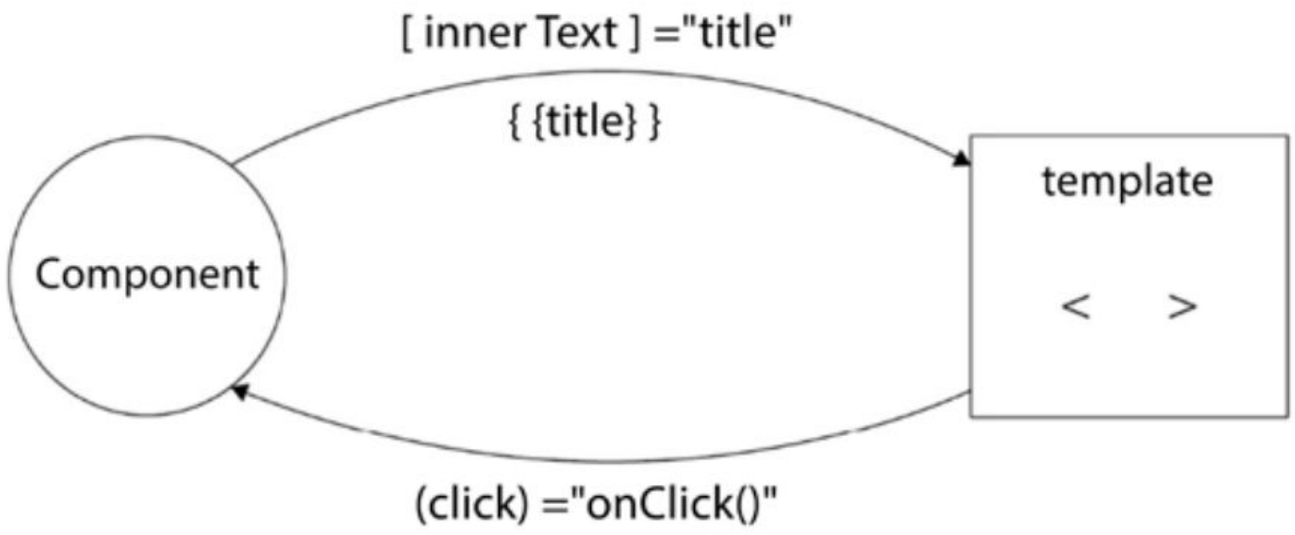
```html
<div class="my-5 container">
    <!-- Property biding with innerHTML -->
    <p [innerHTML]="welcome"></p>
</div>
```

Event Binding in Angular

Event Binding is a technique to get data from the template back to the component class.

An event binding listens for DOM events on the target HTML element and responds to those events by calling corresponding methods in the component class.

Event binding in Angular supports all native DOM events found at https://developer.mozilla.org/docs/Web/Events.



Example of Event Binding

.html

```
>{{number}}</p>
utton (click)="increaseNumber()">Increase</button>
```

.ts

```
export class ProductListComponent {
  number = 0

  increaseNumber(){
    this.number++;
  }
}
```

Control Flow in Components - @if

Introduced in Angular 17+ as part of the **control flow syntax** improvements

Replaces the traditional `*ngIf` structural directive with a **more readable and explicit syntax**

Enables **block-level conditional rendering** using `@if` and `@else`

Better IDE support, type checking, and developer experience

Works inside component templates (inline or external)

Sample code

.html

```
@if (isLoggedIn) {
    <p>Welcome back, user!</p>
  } @else {
    <p>Please log in to continue.</p>
  }
```

ts

```
export class ProductListComponent {
  number = 0
  isLoggedIn = true;
```

Control Flow in Components: @for

Introduced as a modern replacement for `*ngFor`

Provides a **clearer and block-scoped syntax** for iterating over lists

Supports destructuring, `index`, `first`, `last`, `even`, `odd`, and `track` by functionality

Offers **better readability**, **stronger typing**, and improved **tooling support**

Ideal for simplifying nested loops and complex list rendering

Sample code

.html

```
@for (item of items; track item.id) {
   <li>{{ item.name }}</li>
}
```

.

ts

```
items = [
  { id:1,name: 'Item 1', price: 10 },
  {  id:2,name: 'Item 2', price: 20 },
  {  id:3,name: 'Item 3', price: 30 },
  {  id:4,name: 'Item 4', price: 40 }
]
```

@switch and @case

- New **template control flow** syntax introduced in Angular 17+ (stable in Angular 19).
- Alternative to `*ngSwitch` directive.
- Designed for **cleaner**, **declarative**, and **more readable** code in templates.

Code Example

```
@switch (user.role) {
  @case ('admin') {
    <p>Welcome, Admin!</p>
  }
  @case ('editor') {
    <p>Hello, Editor!</p>
  }
  @default {
    <p>Access limited.</p>
  }
}
```

Component Inter-communication

Angular components expose a public API that allows them to communicate with other components.

This API encompasses input properties, which we use to feed the component with data.

It also exposes output properties we can bind event listeners to, thereby getting timely information about changes in the state of the component.

Component Inter-Communication

Add a new component , product-detail:

```
ng generate component product-detail
```

Inside product-detail.component.ts add @Input

```
t { Component, Input } from '@angular/c

export class ProductDetailComponent {

  @Input() product:any;

}
```

Component Inter-Communication (2)

Inside product-detail.component.html bring out the name

```
Go to component
<h2>Product Detail</h2>
<h3>{{product.name}}</h3>
```

Component Inter-Communication (2)

Add new variable *selectedProduct* inside product-list.component.ts

```
selectedProduct: any = null;
```

Create the product list in product-list.component.html

```
@for (item of items; track item.id) {
  <li (click)="selectProduct(item)">{{ item.name }}
  </li>
}
```

Component Inter-Communication (3)

Create selectProduct method in product-list.component.ts

```
selectProduct(product: any) {
  this.selectedProduct = product;
}
```

Component Inter-Communication (4)

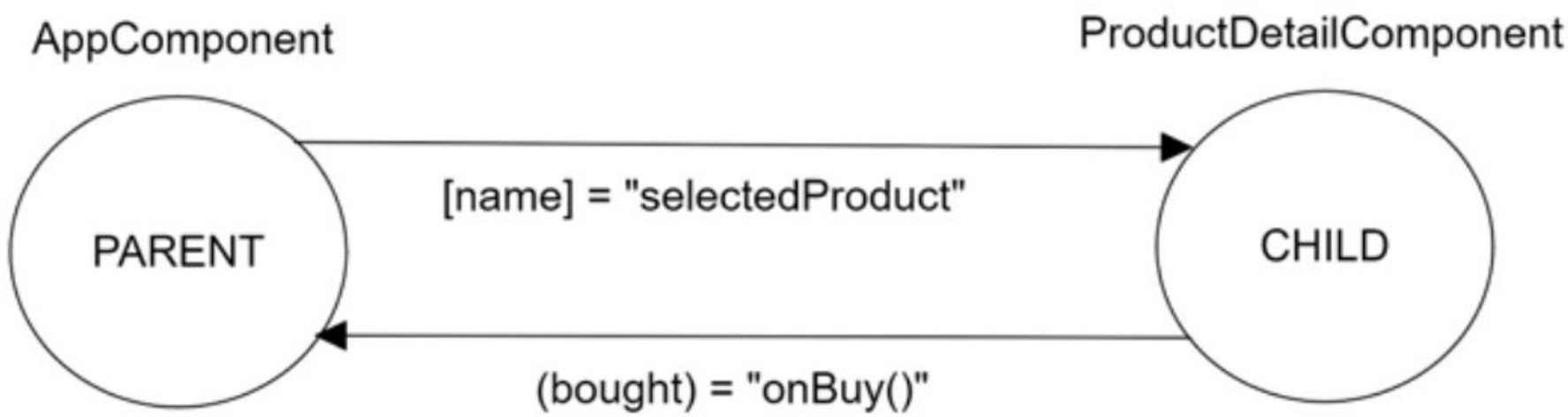
Pass the *selectedProduct* to the *ProductDetail* component . This approach is called input binding.

```
@for (item of items; track item.id) {
  <li (click)="selectProduct(item)">{{ item.name }}
  </li>
}
<app-product-detail
  [product]="selectedProduct" />
```

Listening for events on output bindin

We use an output directive to notify the parent component about specific actions that occur in the child component.

In this example we will add a button “Buy Now” on the child component and notify the parent component of this event .



Listening for events on output binding

In product-detail.component.ts, import the following:

```
1    import { Component, Input, Output,EventEmitter }
2
```

Add an @Output directive with an EventEmitter object:

```
@Input() product:any;
@Output() bought = new EventEmitter();
```

Listening for events on output binding

In product-detail.component.ts, implement the method buy :

```
buy(){
  this.bought.emit();
}
```

Add a button that will be linked to the method buy in product.component.html

```
<h2>Product Detail</h2>
<h3>{{product.name}}</h3>
<p>USD {{product.price}}</p>
<button (click)="buy()">Buy now</button>
```

Listening for events on output binding

Modify the selector on product-list.component.html

```
    [product]="selectedProduct" (bought)="onBuy()"/>
```

Implement the method onBuy in product-list.component.ts

```
onBuy(){
  alert(`You bought ${this.selectedProduct.name}`);
}
```

Emitting data through custom event

The emit method of an EventEmitter property can accept any data to pass up to the parent component. The proper way is initially to define the data type that can be passed to the EventEmitter property.

product-detail.component.ts

```
export class ProductDetailComponent {

  @Input() product:any;
  @Output() bought = new EventEmitter<any>();

  buy(){
    this.bought.emit(this.product);
  }
```

Emitting data through custom event

product-list.component.html

```
[product]="selectedProduct" (bought)="onBuy($event)"/
```

product-list.component.ts

```
onBuy(product:any){
  alert(`You bought ${product.name}
  for ${product.price}`);
}
```

Introduction to component lifecycle

Lifecycle events are hooks that allow us to jump into specific stages in the lifecycle of a component and apply custom logic

Some hooks are considered best practices, while others help debug and understand what happens in an Angular application. A hook comes with an interface that defines a method we need to implement. The Angular framework ensures the hook is called, provided we have implemented this method in the component.

Component Lifecycle

The most basic lifecycle hooks of an Angular component are:

- **OnInit**: This is called when a component is initialized

- **OnDestroy**: This is called when a component is destroyed

- **OnChanges**: This is called when values of input binding properties in the component change

- **AfterViewInit**: This is called when Angular initializes the view of the current component and its child components

A full list of all the supported lifecycle hooks is available in the official Angular documentation at https://angular.dev/guide/components/lifecycle

Performing component initialization

The OnInit lifecycle hook implements the ngOnInit method, which is called during the component initialization. At this stage, all input bindings and data-bound properties have been set appropriately, and we can safely use them.

Using the component constructor to access them may be tempting, but their values would not have been set at that point

Constructors should be relatively empty and devoid of logic other than setting initial variables.

Performing component initialization

```
import { Component,OnInit } from '@angular/core';
import { ProductDetailComponent } from "../product-detail/product-detail.component";

@Component({
  selector: 'app-product-list',
  imports: [ProductDetailComponent],
  templateUrl: './product-list.component.html',
  styleUrl: './product-list.component.css'
})
export class ProductListComponent implements OnInit {
  number = 0
  isLoggedIn = true;

  constructor() {
    console.log('Called in constructor');
  }

  ngOnInit() {
  console.log('Called in OnInit');
  }
```

Performing component initialization

```
Navigated to http://localhost:4200/
Called in constructor                                    product-l
Called in OnInit                                         product-l
```

Detecting input binding change

The OnChanges lifecycle hook is called when Angular detects that the value of an input data binding has changed.

In our example, we will use it in the product detail component to learn how it behaves when we select a different product from the list:

Detecting input binding change

product-detail.component.ts

```
import { Component, Input, Output,EventEmitter, OnChanges, SimpleChanges } from '@angular/core';

@Component({
export class ProductDetailComponent implements OnChanges {


  ngOnChanges(changes: SimpleChanges): void {
    const productChange = changes['product'];
    const previousProduct = productChange.previousValue;
    const currentProduct = productChange.currentValue;
    console.log('Previous Product:', previousProduct);
    console.log('Current Product:', currentProduct);
  }
```

Directives

Angular directives are HTML attributes that extend the behavior or the appearance of a standard HTML element. When we apply a directive to an HTML element or even an Angular component, we can add custom behavior to it or alter its appearance.

There are three types of directives:

- Components: They are directives with an associated template
- Structural directives: They add or remove elements from the DOM
- Attribute directives: They modify the appearance of or define a custom behavior for a DOM element

Transforming elements using directives

Angular framework includes a set of ready-made structural directives that we can start using straight away in our applications:

- ngIf: Adds or removes a portion of the DOM tree based on an expression
- ngFor: Iterates through a list of items and binds each item to a template
- ngSwitch: Switches between templates within a specific set and displays each one depending on a condition

Displaying data conditionally

The ngIf directive adds or removes an HTML element in the DOM based on the evaluation of an expression. If the expression evaluates to true, the element is inserted into the DOM. Otherwise, the element is removed from the DOM.

We can use a <ng-template> element to compose an if-else statement in the template of our component

CommonModule

To use built-in directive or pipes, you need to Import CommonModule in component:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
@Component({
  selector: 'app-product-list',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './product-list.component.html',
  styleUrl: './product-list.component.css'
})
```

Displaying data conditionally

product-list.component.ts

```
selectedProduct : string | undefined;
```

product-list.component.html

```
<app-product-detail [name]="selectedProduct"
(bought)="onBuy($event)"
*ngIf="selectedProduct"></app-product-detail>
```

Displaying data conditionally

product-list.component.html

```
<app-product-detail [name]="selectedProduct"
(bought)="onBuy($event)"
 *ngIf="selectedProduct; else noProduct"></app-product-detail>
<ng-template #noProduct>
 <p>No product selected!</p>
</ng-template>
```

Iterating through data

The ngFor directive allows us to loop through a collection of items and render a template for each one, where we can define convenient placeholders to interpolate item data.

Each rendered template is scoped to the outer context, where the loop directive is placed so that we can access other bindings.

We can think of ngFor as the for loop for HTML templates.

Iterating through data (3)

product-list.component.ts

```
})
export class ProductListComponent {
  title = "Product page";
  products = ['Webcam', 'Microphone', 'Wireless keyboard'];
}
```

product-list.component.html

```
Go to component
<h2 [innerText]="title"></h2>
<ul>
    <li *ngFor="let product of products" (click)="selectedProduct = product">
     {{product}}
</li>
```

Iterating through data (4)

The variable is a template input variable that we can reference later in our template. The property can have the following values:

- index: Indicates the index of the item in the array, starting at 0 (number)

- first/last: Indicates whether the current item is the first or last item in the array (boolean)

- even/odd: Indicates whether the index of the item in the array is even or odd (Boolean)

Iterating through data (5)

```
<div>
    <ul>
        <li *ngFor="let product of products; let i = index" >
        {{i+1}} - {{product}}
        </li>
    </ul>
</div>
```

Iterating through data (6)

During the execution of ngFor, data may change, elements may be added or removed, and the whole list may even be replaced. Angular must take care of these changes by creating/removing elements to sync changes to the DOM tree. This is a process that can become very slow and expensive and will eventually result in the poor performance of your application.

Angular deals with variations within a collection by keeping DOM elements in memory. Internally,it uses something called **object identity** to keep track of every item in a collection. We can, how ever, use a specific property of the iterable items instead of the internal Angular object identity using the trackBy property:

Iterating through data (7)

```
<ul>
    <li *ngFor="let product of products; trackBy: trackByProducts">
    {{product}}
    </li>
</ul>
```

```
export class ProductListComponent {
  title = "Product page";
  products = ['Webcam', 'Microphone', 'Wireless keyboard'];
  trackByProducts(index: number, name: string): string {
    return name;
  }
}
```

Switching through templates

ngSwitch is used to switch between templates and display each one depending on a defined value.

It is like an ordinary switch statement that we use in other programming languages, consists of a set of other directives:

- [ngSwitch]: Defines the property that we want to check when applying the directive

• *ngSwitchCase: Adds or removes a template from the DOM tree depending on the value of the property defined in the [ngSwitch] statement

• *ngSwitchDefault: Adds a template to the DOM tree if the value of the property definedin the [ngSwitch] directive does not meet any *ngSwitchCase statement

Switching through templates

```
<h2>Product Details</h2>
 <h3>{{name}}</h3>
 <div [ngSwitch]="name">
    <p *ngSwitchCase="'Webcam'">
    Product is used for video
    </p>
    <p *ngSwitchCase="'Microphone'">
    Product is used for audio
    </p>
    <p *ngSwitchDefault>Product is for general use</p>
    </div>
<button (click)="buy()">Buy Now</button>
```

Manipulating data with pipes

Pipes allow us to transform the outcome of our expressions at the view level. They take data as input, transform it into the desired format, and display the output in the template.

The syntax of a pipe is pretty simple, consisting of the pipe name following the expression we want to transform, separated by a pipe symbol (|).

Pipes are usually used with interpolation in Angular templates and can be chained to each other.

Angular has a wide range of pipe types already baked into it.

https://angular.io/guide/pipes

Creating product interface

ng g interface product

product.ts

```
src > app > TS product.ts > Product
1    export interface Product {
2        name: string;
3        price: number;
4    }
```

Creating array of products

In product-list.component.ts replace products to array of Product. Verify in the project to take Product type instead of string.

```
products: Product[] = [
  {
  name: 'Webcam',
  price: 100
  },
  {
    name: 'Microphone',
    price: 200
    },
    {
    name: 'Wireless keyboard',
    price: 85
    }
  ];
```

Manipulating data with pipes

`uppercase`/`lowercase`: They transform a string into a particular case.

```
{{product.name | uppercase}}

{{product.name | lowercase}}
```

`percent`: This formats a number as a percentage. For example, the output of `<p>{{0.1234 | percent}}</p>` will be **12%**.

- `currency`: This formats a number as a local currency. We can override local settings and change the symbol of the currency, passing the currency code as a parameter to the pipe.

```
{{product.price | currency:'USD'}}
```

Manipulating data with pipes

slice:This subtracts a subset (slice) of a collection or string. It accepts a starting index,

where it will begin slicing the input data, and optionally an end index as parameters. When the end index is specified, the item at that index is not included in the resulting array. If the end index is omitted, it falls back to the last index of the data.

```
<ul>
<li *ngFor="let product of products | slice:1:3">
{{product.name}}
</li>
</ul>
```

Manipulating data with pipes

date: This formats a date or a string as a particular date format. The time zone of the formatted output is in the local time zone of the end user's machine. The following snippet displays the component property today as a date:

```
<p>{{today | date}}</p>
<p>{{today | date:'fullDate'}}</p>
```

https://angular.io/api/common/DatePipe

json: It takes an object as an input and outputs it in JSON format.

```
<p>{{product | json}}</p>
```

Creating a custom pipe

To create a new pipe, we use the `generate` command of the Angular CLI, passing the word `pipe` and its name as parameters:

```
ng generate pipe sort
```

If you want to create on a particular folder

```
ng generate pipe pipes/sort
```

Creating a custom pipe

```ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'sort',
  standalone: true
})
export class SortPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }

}

```

Creating a custom pipe

The only required property in the `@Pipe` decorator is the `name` of the pipe. A pipe must implement the `transform` method of the `PipeTransform` interface that accepts two parameters:

•`value`: The input data that we want to transform

• `args`: An optional list of arguments we can provide to the transformation method, each separated by a colon

Creating a custom pipe

```
import { Pipe, PipeTransform } from '@angular/core';
import { Product } from './product';
@Pipe({
  name: 'sort',
  standalone: true
})
export class SortPipe implements PipeTransform {

  transform(value: Product[]): Product[] {
    if (value) {
      return value.sort((a: Product, b: Product) => {
      if (a.name < b.name) {
      return -1;
      } else if (b.name < a.name) {
      return 1;
      }
      return 0;
      });
    }
    return [];
  }
```

Example in case (using product)

```
transform(value: string[]): string[] {
    if (value){
      return value.sort((a:string,b:string)=>{
        if (a < b ){
          return -1;
        } else if (b < a){
          return 1
        }
        return 0
      })
    }
    return [];
}
```

Creating a custom pipe

```
import { Component, AfterViewInit, ViewChild } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductDetailComponent } from '../product-detail/product-det
import { Product } from '../product';
import { SortPipe } from '../sort.pipe';
@Component({
  selector: 'app-product-list',
  standalone: true,
  imports: [CommonModule, ProductDetailComponent, SortPipe],
  templateUrl: './product-list.component.html',
  styleUrl: './product-list.component.css'
})
```

Creating a custom pipe

```
<h2 [innerText]="title"></h2>
<ul>
    <li *ngFor="let product of products | sort" (click)="selectedProduct = product">
     {{product.name | uppercase}}
    <p>{{product.price | currency:'USD'}}</p>
</li>
</ul>
```

Custom Directive

Custom Directive allow you to attach advanced behaviors to elements in the DOM or modify their appearance.

Directives mainly fall into two categories: structural and attribute.

```
ng g directive copyright

ng g directive directives/copyright
```

Reference to DOM manipulation (pure JS) :
https://www.w3schools.com/jsref/met_document_createelement.asp

Custom directive

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

@Directive({
  selector: '[appCopyright]',
  standalone: true
})
export class CopyrightDirective {

  constructor() { }

}
```

Custom directive

app > TS copyright.directive.ts > CopyrightDirective > constructor

```ts
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appCopyright]',
  standalone: true
})
export class CopyrightDirective {

  constructor(el: ElementRef) {
    const currentYear = new Date().getFullYear();
    const targetEl: HTMLElement = el.nativeElement;
    targetEl.classList.add('copyright');
    targetEl.textContent = `Copyright ©${currentYear} All Rights Reserved.`;
  }

}
```

Custom directive

The ElementRef class to access and manipulate the underlying HTML element attached to the directive. The nativeElement property contains the actual native HTML element.

Add the `copyright` class in styles.css

```
.copyright {
   background-color: lightgray;
   padding: 10px;
   font-family: Verdana, Geneva, Tahoma,
sans-serif;
  }
```

Custom Directive

product.component.html

```
</li>
</ul>
<app-product-detail [product]="selectedProduct"
(bought)="onBuy($event)"
 *ngIf="selectedProduct; else noProduct"></app-product-de
<ng-template #noProduct>
 <p>No product selected!</p>
</ng-template>
<p appCopyright></p>
```

Property binding and responding to event

The Angular framework provides two helpful decorators that we can use in our directives to enhance their functionality:

- @HostBinding: This binds a value to the property of the native host element
- @HostListener: This binds to an event of the native host element

A good example on how you can create custom directive:

https://codecraft.tv/courses/angular/custom-directives/hostlistener-and-hostbinding/

Property binding and responding to event(2)

In this example we will create a directive that listens to keypress event from a directive and intercept non numerical value .

```
ng generate directive numeric
```

Property binding and responding to event(3)

numeric.directive.ts

```
import { Directive, HostBinding, HostListener } from '@angular/core';
```

```
  @HostBinding('class') currentClass = '';
```

Property binding and responding to event(4)

numeric.directive.ts

```
@HostListener('keypress', ['$event']) onKeyPress(event:
  KeyboardEvent) {
    const charCode = event.key.charCodeAt(0);
    console.log(charCode);
    if (charCode > 31 && (charCode < 48 || charCode > 57)) {
    this.currentClass = 'invalid';
    event.preventDefault();
    } else {
    this.currentClass = 'valid';
    }
  }
```

Property binding and responding to event(5)

product-list.component.html

```
<input appNumeric />
```

product-list.component.ts

```
import { NumericDirective } from '../numeric.directive';
```

```
  standalone: true,
  imports: [CommonModule,RouterModule, ProductDetailComponent,
    SortPipe, CopyrightDirective, ProductCreateComponent,
    NumericDirective],
```

Sibling - Sibling communication