Angular Fundamentals

With your host ***Mark Techson***

What to expect

By the end of this workshop you'll understand the Angular fundamentals and be able to build an application.

Workshop Key Takeaways

- Learn how to build applications with components
- Learn how to add routing to your application
- Learn how to capture user input with forms
- Learn how use dependency injection
- Learn app optimization techniques

About me?

I'm alright.

Meet your instructor

- Award winning university instructor
- Angular team at Google
- I love videography, guitars/music and video games

About this course

Course Structure

- We'll discuss a topic
- I'll go over an example
- You'll some activities

Important Links

- These slides (goo.gle/fem-slides)
- Project Code (goo.gle/fem-code)

Software Installation

- Latest version of Node or Active LTS
- Angular CLI
- Visual Studio Code
- Angular Language Service Plugin for VS Code

What is Angular?

Angular is a web framework used to be build scalable web apps with confidence.

00 Let's try Angular

Hello Angular.dev

01 Navigate to angular.dev/playground

02 Select the "Hello World" template from the menu

03 Change Hello World to Hello Universe in the template

01 Project Setup

Local installation

Software Installation

- Latest version of Node or Active LTS
- Angular CLI
- Visual Studio Code
- Angular Language Service Plugin for VS Code

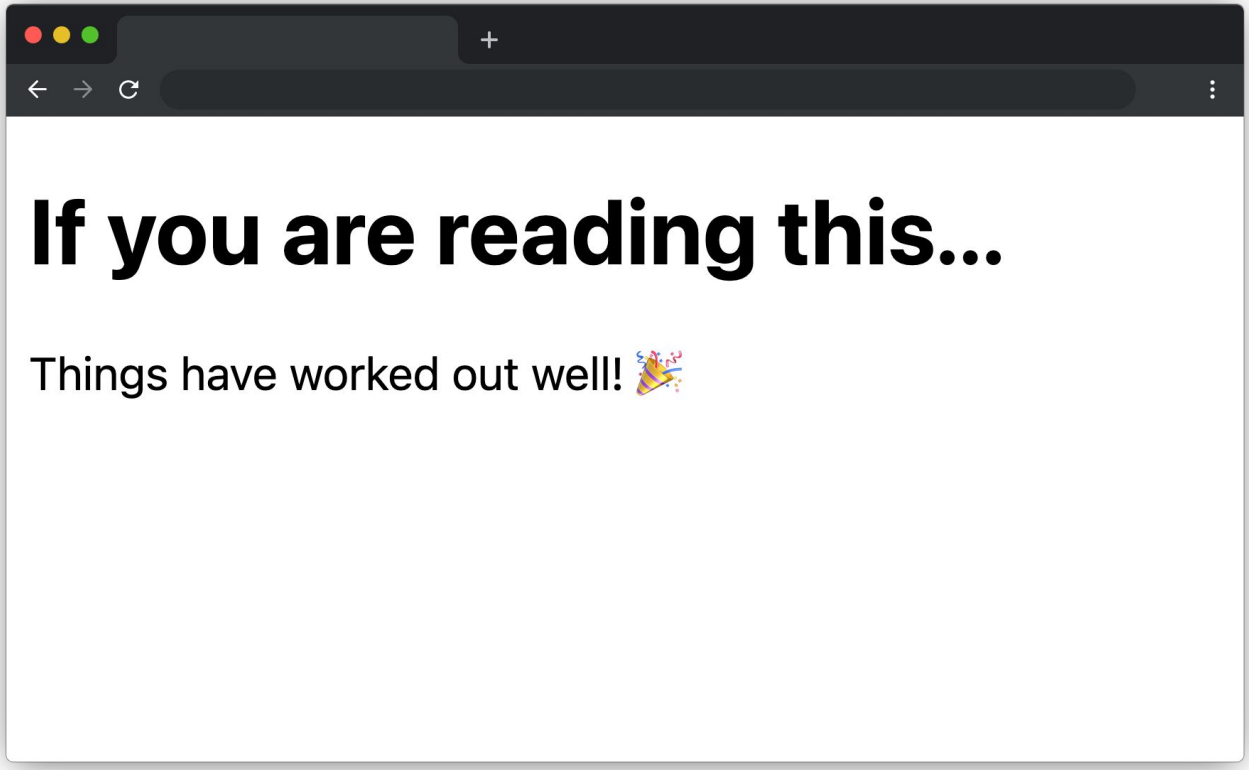
```
$ ng version

Angular CLI: 17.0.1
```

```
$ git clone
https://github.com/MarkTechson/angular-fundamentals-lessons.git
```

(you can also use ssh)

```
1 $ cd angular-fundamentals-lessons
2 $ npm install
3 $ ng serve 01-hello-angular
4
5
```

http://localhost:4200



Explore an Angular project

01 Open VS Code

02 Open the project folder in VS Code:

File > Open Folder > [Choose angular-fundamentals-lessons]

Hello, Angular

You build Angular apps with TypeScript, HTML and CSS.

At the core of Angular is the component

TypeScript

Programming logic in your application

HTML

This is how you define your markup in templates

CSS

Styling your templates

How do you build components in Angular?

app.component.ts

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

Editor Preview Both

app.component.ts

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

@Component({

})
```

Editor Preview Both

app.component.ts

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

@Component({
  selector: 'app-root',
})
```

Editor Preview Both

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
})
```

Editor Preview Both

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  template: `<h1>Hey, Frontend Masters!</h1>`,
})
```

Editor Preview Both

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  template: `<h1>Hey, Frontend Masters!</h1>`,
  styles: `h1 { color: red }`,
})
```

Editor Preview Both

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  template: `<p> Hey, Frontend Masters!</p>`,
  styles: `h1 { color: red }`,
})
class AppComponent {}
```

Editor Preview Both

01 - Hello Angular

01 Open the project README for instructions.

02 Complete the tasks, save your code.

03 Confirm the output in the browser.

Displaying dynamic values in components

app.component.html

```
<section>
    <p> Welcome back, USER</p>
</section>
```

Editor Preview Both

app.component.html

```
<section>
    <p> Welcome back, USER</p>
</section>
```

Editor Preview Both

app.component.ts

```
@Component ({ ... })
export class WelcomeComponent {
    userName = 'codingChamp';
}
```

Editor Preview Both

app.component.html

```
<section>
  <p> Welcome back, USER</p>
</section>
```

Editor Preview Both

app.component.html

```
<section>
  <p> Welcome back, {{ ?? }}</p>
</section>
```

interpolation

Editor Preview Both

app.component.html

```
<section>
  <p> Welcome back, {{userName}}</p>
</section>
```

Editor Preview Both

02 - displaying-dynamic-data

A single component is like this block

Leveraging the power of multiple components is where the power resides.

Component Composition

dashboard.component.html

```html
<!-- DashboardComponent template -->
<section>
    <p>Welcome back</p>
</section>
```

Editor Preview Both

userInfo.component.html

```html
<!-- UserInfo template -->
<article>
    <p>{{user.userName}}</p>
    <p>{{user.email}}</p>
    <p>{{user.lastLogin}}</p>
</article>
```

Editor Preview Both

dashboard.component.ts

```
import {UserInfoComponent} from './user-info.component';

@Component({
    selector: 'app-dashboard',
    template: `
        <section>
            <p>Welcome back</p>
        </section>
    `,
})
export class DashboardComponent {}
```

Editor Preview Both

dashboard.component.ts

```
import {UserInfoComponent} from './user-info.component';

@Component({
    selector: 'app-dashboard',
    template: `
        <section>
            <p>Welcome back</p>
        </section>
    `,
    imports: [UserInfoComponent]
})
export class DashboardComponent {}
```

Editor Preview Both

dashboard.component.ts

```
import {UserInfoComponent} from './user-info.component';

@Component({
    selector: 'app-dashboard',
    template: `
        <section>
            <p>Welcome back</p>
            <app-user-info />
        </section>
    `,
    imports: [UserInfoComponent]
})
export class DashboardComponent {}
```

Editor Preview Both

03-component-composition

How do you make decisions in your template?

Control Flow with `@if`

home.component.html

```html
<!-- HomePageComponent template -->
<section>
    <p>Please login</p>
    <p>Welcome back</p>
</section>
```

Editor Preview Both

home.component.html

```
<section>
    <!-- user.isLoggedIn -->
    <p>Please login</p>
    <!-- !user.isLoggedIn -->
    <p>Welcome back</p>
</section>
```

Editor Preview Both

home.component.html

```
<section>
    @if( expr ) {
      <p>Please login</p>
    }

    <p>Welcome back</p>
</section>
```

Editor Preview Both

home.component.html

```
<section>
    @if(user.isLoggedIn) {
    <p>Please login</p>
    }

    <p>Welcome back</p>
</section>
```

Editor Preview Both

home.component.html

```
<section>
    @if(user.isLoggedIn) {
    <p>Please login</p>
    } @else {
      <p>Welcome back</p>
    }
</section>
```

Editor Preview Both

home.component.html

```
<section>
    @if(orderAmount < 50 ) {
        <p>Your discount amount is 0</p>
    } @else if (orderAmount < 100 ) {
        <p> Your discount amount is {{ orderAmount * .1 }}</p>
    } @else {
        <p>Your discount amount is {{ orderAmount * .2}}</p>
    }
</section>
```

Editor Preview Both

How do you make decisions in your template?

04-control-flow-if

order-details.component.html

```
<!-- What's the issue with this template? -->
<article>
    <p>{{cart[0].price}}</p>
    <p>{{cart[1].price}}</p>
    <p>{{cart[2].price}}</p>
</article>
```

A loop would be better

Editor Preview Both

Control Flow with `@for`

order-details.component.html

```
<article>
    @for(item of cart; track item.id) {
        <p>{{item.price}}</p>
    }
</article>
```

Editor Preview Both

What if the list is empty?

order-details.component.html

```
<article>
    @for(item of cart; track item.id) {
        <p>{{item.price}}</p>
    } @empty {
        <p>Your cart is empty</p>
    }
</article>
```

Editor Preview Both

05-control-flow-for

Property Binding

Property binding in Angular enables you to set values for properties of elements in your templates.

form.component.ts

```
@Component() {
  template: `
    <button type="button" [disabled]='isDisabled'>
      Submit
    </button>
  `
}
export class AppComponent {
  isDisabled = false;
}
```

Editor Preview Both

Event Handling

Event handling in Angular enables you to respond to events in your templates.

app.component.ts

```
@Component() {
  template: `
    ...
    <button type="button">Save Progress</button>
  `
}
export class AppComponent {}
```

Editor Preview Both

app.component.ts

```
@Component() {
  template: `
    <button type="button" (click)="handleClick()">
        Save Progress
    </button>
  `
}
export class AppComponent {
    handleClick() { … }
}
```

Editor Preview Both

Customizing Components with @Input

@Input

Send information into a component (like props)

app.component.ts

```
@Component({
   selector: 'app-cmp',
    template: `<app-user-card />`,
    imports: [UserCardComponent],
})
export class AppComponent {
     user: User = { name: 'Ashley', bio: 'Cool developer',};
}
```

Editor Preview Both

app.component.ts

```
@Component({
    selector: 'app-cmp',
    template: `<app-user-card [userData]="user"/>`,
    imports: [UserCardComponent],
})
export class AppComponent {
    user: User = { name: 'Ashley', bio: 'Cool developer',};
}
```

Editor Preview Both

user.component.ts

```
@Component({
   selector: 'app-user-card',
    template: `
        <section>
            <p>{{userData.name}}</p><p>{{userData.bio}}</p>
        </section>`,
})
export class AppComponent {
    @Input() userData: User = {...}; // default user data
}
```

Editor Preview Both

Custom events with @Output

@Output

Send information from a child component to a parent via custom events

product-list.component.ts

```
@Component({
  template: `
    <button class="btn" (click)="addItem()">Add Item</button>
  `,
})
export class ProductListComponent {
  @Output() addItemEvent = new EventEmitter<string>();
  ...
}
```

Editor Preview Both

product-list.component.ts

```
@Component({
  template: `
    <button class="btn" (click)="addItem()">Add Item</button>
  `,
})
export class ProductListComponent {
  @Output() addItemEvent = new EventEmitter<string>();

  addItem() { this.addItemEvent.emit('🐢'); }
}
```

Editor Preview Both

app.component.ts

```
@Component({
  template: `
    <app-child (addItemEvent)="addItem($event)" />
  `,
  imports: [ChildComponent],
})
export class AppComponent {
  items: string[] = [];
  addItem(item: string) { this.items.push(item); }
}
```

Editor Preview Both

06-inputs-and-outputs

App screenshot of what we'll build.

App Project: Components

Routing

Angular has a built-in, complete router.

`@angular/router`

routes.ts

```
import { Component } from '@angular/core';
import { Routes } from '@angular/router';
import { DetailsComponent } from './details/details.component';

export const routes: Routes = [
    {
        path: 'details',
        component: DetailsComponent,
    }
];
```

Editor Preview Both

app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <router-outlet />
  `,
  styles: '',
  imports: [RouterModule]
})
export class AppComponent {}
```

Editor Preview Both

app.config.ts

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

Editor Preview Both

07-routing-basics

How do you make links?

`routerLink`

app.component.ts

```
@Component({
  template: `
    <a routerLink="/details">Details</a>
    <router-outlet />
  `,
  standalone: true,
  imports: [RouterOutlet, RouterLink],
})
export class AppComponent {}
```

Editor Preview Both

0#-router-link

How do you create dynamic routes?

```
/details/1
```

routes.ts

```
import { Component } from '@angular/core';
import { Routes } from '@angular/router';
import { DetailsComponent } from './details/details.component';

export const routes: Routes = [
    {
        path: 'details/:id',
        component: DetailsComponent,
    }
];
```

Editor Preview Both

details.component.ts

```
@Component({...})
export class DetailsComponent {
    productId = -1;//dest for route info

    @Input()
    set id(value: number) {
      this.productId = value;
    }
}
```

Editor Preview Both

app.config.ts

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes,
              withComponentInputBinding())]
};
```

Editor Preview Both

08-router-recap

Forms

How do you gather user input?

`Forms`

A tale of two systems

Template Driven Forms

- Quick to setup and use
- Best for small one-time use forms
- Requires more configuration for testing

Reactive Forms

- Supports typing
- Reusable, can share models
- More robust testing configuration

app.component.html

```html
<form name="loginForm">
    <label>Username:
        <input type="text" />
    </label>

    <label for="password">Password:
        <input type="password" />
    </label>
    <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.html

```
<form name="loginForm">
    <label>Username:
        <input type="text" [(ngModel)]="username"/>
    </label>

    <label for="password">Password:
        <input type="password" />
    </label>
    <button type="submit">Login</button>
</form>
```

Banana in a box
Property binding
+
Event

Editor Preview Both

app.component.html

```
<form name="loginForm">
    <label>Username:
        <input type="text" />
    </label>

    <label for="password">Password:
        <input type="password" [(ngModel)]="password"/>
    </label>
    <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.ts

```
@Component({
  imports: [FormsModule],
  templateUrl: 'app.component.html',
})
export class AppComponent {
  username = "";
  password = "";
}
```

Editor Preview Both

09-template-driven-forms

What about reactive forms?

app.component.ts //Define the model

```
@Component({
  imports: [ReactiveFormsModule],
  templateUrl: 'app.component.html',
})
export class AppComponent {
    loginForm = new FormGroup({
        name: new FormControl(''),
        email: new FormControl(''),
    });
}
```

Editor Preview Both

app.component.html //Connect the model to the template

```html
<form name="loginForm">
    <label>Username:
        <input type="text" />
    </label>

    <label for="password">Password:
        <input type="password" />
    </label>
    <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.html

//Bind the model to the formGroup

```
<form name="loginForm" [formGroup]="loginForm">
    <label>Username:
        <input type="text" />
    </label>

    <label for="password">Password:
        <input type="password" />
    </label>
    <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.html //Define the form controls

```
<form name="loginForm" [formGroup]="loginForm">
    <label>Username:
        <input type="text" formControlName="username"/>
    </label>

    <label for="password">Password:
        <input type="password" />
    </label>
    <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.html

```
<form name="loginForm" [formGroup]="loginForm">
    <label>Username:
        <input type="text" formControlName="username"/>
    </label>

    <label for="password">Password:
        <input type="password" formControlName="password"/>
    </label>
    <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.html

//Handle the submit with ngSubmit

```
<form name="loginForm" [formGroup]="loginForm"
      (ngSubmit)="handleSubmit()">
    <label>Username:
        <input type="text" formControlName="username"/>
    </label>

     <label for="password">Password:
        <input type="password" formControlName="password"/>
     </label>
     <button type="submit">Login</button>
</form>
```

Editor Preview Both

app.component.ts

```
@Component({...})
export class AppComponent {
    loginForm = new FormGroup(...);

    handleSubmit() {
      this.loginWithCredentials(this.loginForm.value);
    }
}
```

Editor Preview Both

10-reactive-forms

Dependency Injection

Dependency Injection (DI)

"DI" is a design pattern and mechanism for creating and delivering some parts of an app to other parts of an app that require them.

Dependency Injection (DI)

"DI" is a design pattern and mechanism for **creating and delivering** some parts of an app to other parts of an app that require them.

Dependency Injection (DI)

"DI" is a design pattern and mechanism for **creating and sharing** some parts of an app to other parts of an app that require them.

app.component.ts

// Make the service injectable

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

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

'root' means available to the entire application

Editor Preview Both

app.component.ts //Make the service available

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

@Component({...})
export class AppComponent {
    carService = inject(CarService);
}
```

Editor Preview Both

app.component.ts

```
@Component({...})
export class AppComponent {
    carService = inject(CarService);
    cars: string[]

    constructor() {
        this.carService.getCars();
    }
}
```

Editor Preview Both

11-dependency-injection

App Optimizations

Angular Signals

Three reactive primitives

signal

computed

effect

1. Signal

A value that can tell Angular when it changes

capable of notifying its context of future changes in its value

signal

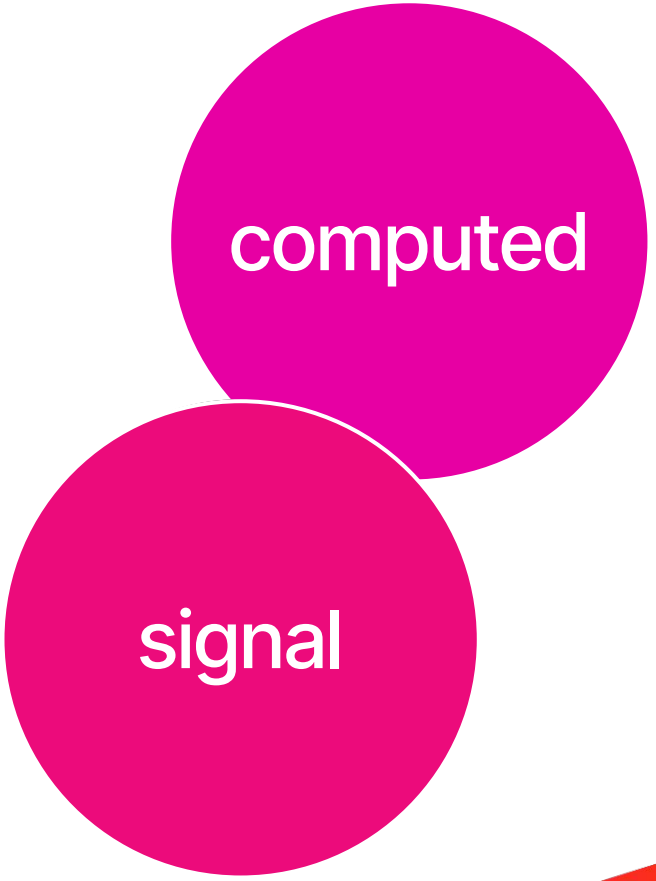
app.component.ts //Defining writable signals

```
@Component({
    template: `<p>{{ lastName() }}, {{ firstName() }}</p>`
})
export class AppComponent {
    firstName = signal('Jessica');
    lastName = signal('Wesley');
}
```

Editor Preview Both

2. Computed

Derive new value when one of the dependent signals change



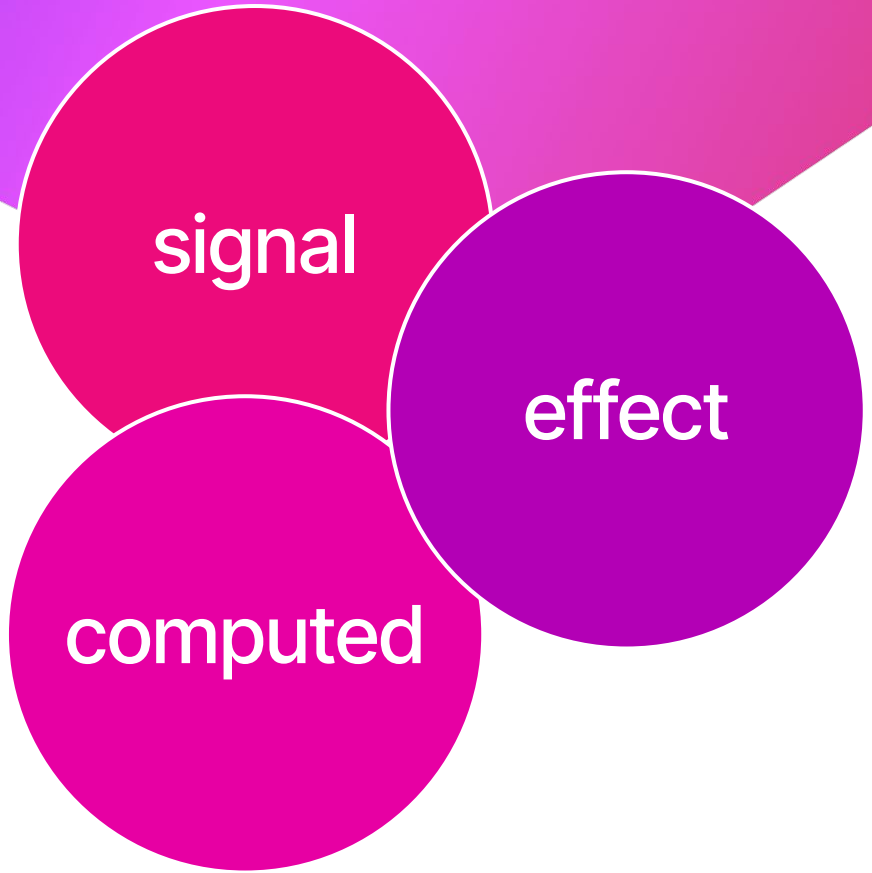
app.component.ts //Defining computed signals

```
@Component({
    template: `<p>{{ fullName() }}</p>`
})
export class AppComponent {
    firstName = signal('Simona');
    lastName = signal('Cotin');
    fullName = computed(() => `${firstName()} ${lastName()}`);
}
```

Editor Preview Both

3. Effect

An effect is a side-effectful operation which reads the value of zero or more signals



app.component.ts //Defining effect signals

```
@Component({
    template: `<p>{{ fullName() }}</p>`
})
export class AppComponent {
    firstName = signal('Simona');
    lastName = signal('Cotin');
    effect(() => console.log('Updated: ' +  lastName()));
}
```

Editor Preview Both

12-signals-todo

Deferrable Views

Lazy loading helps keep initial bundle sizes smaller

app.component.html

```
<button #trigger>Load Recommend Movies</button>

@defer (on interaction(trigger)) {
  <recommended-movies />
} @loading {
  <p>Loading ⏳</p>
} @error {
  <p>Oops, sorry 🤕</p>
} @placeholder {
  <img src="placeholder-image.png" />
}
```

Editor Preview Both

app.component.html

```
<button (click)="count = count + 1">
Add one
</button>

@defer (when count > 5) {
  <recommended-movies />
@placeholder {
  Count is {{ count }}.
}
```

Editor Preview Both

Robust powerful triggers

- `on idle`
- `on immediate`
- `on timer(...)`
- `on viewport(...)`
- `on interaction(...)`
- `on hover(...)`

Deferrable Views + Prefetching

app.component.html

// Deferrable views w/prefetching

```
@defer (on interaction(trigger);
        prefetch on idle ) {
  <recommended-movies />
}

@defer (on interaction(trigger);
        prefetch when count > 5 ) {
  <recommended-movies />
}
```

Editor Preview Both

13-deferrable-views

The Angular CLI
Bonus

The Angular CLI

- Create applications
- Create resources like components, services and more
- Launch local development server
- So much more

Course Wrap up

What to do next?

- angular.dev for documentation and tutorials
- Stay connected
  - x.com/angular
  - youtube.com/@angular

Thank you

Sincerely, your host ***Mark Techson***