Introduction to Angular

Angular is a powerful and versatile JavaScript framework used to build dynamic web applications. It's maintained by **Google** and has become a popular choice for front-end development, favored for its efficiency, robustness, and comprehensive features. This presentation delves into the fundamentals of Angular, exploring its core concepts, advantages, and practical implementation.

Why Frontend Frameworks?

1. **Code Reusability**

   Frameworks provide pre-built components and structures, promoting code reuse, reducing development time, and fostering consistency across applications.

2. **Simplified Development**

   Frameworks offer standardized patterns and best practices, simplifying development by streamlining common tasks and reducing the need for repetitive coding.

3. **Enhanced Testability**

   Frameworks often include testing tools and frameworks, making it easier to write and run tests, improving code quality and ensuring reliability.

4. **Community Support**

   Popular frameworks have large and active communities, providing access to extensive documentation, support forums, and pre-built libraries.

Benefits of Angular Over Other Frameworks

Component-Based Architecture

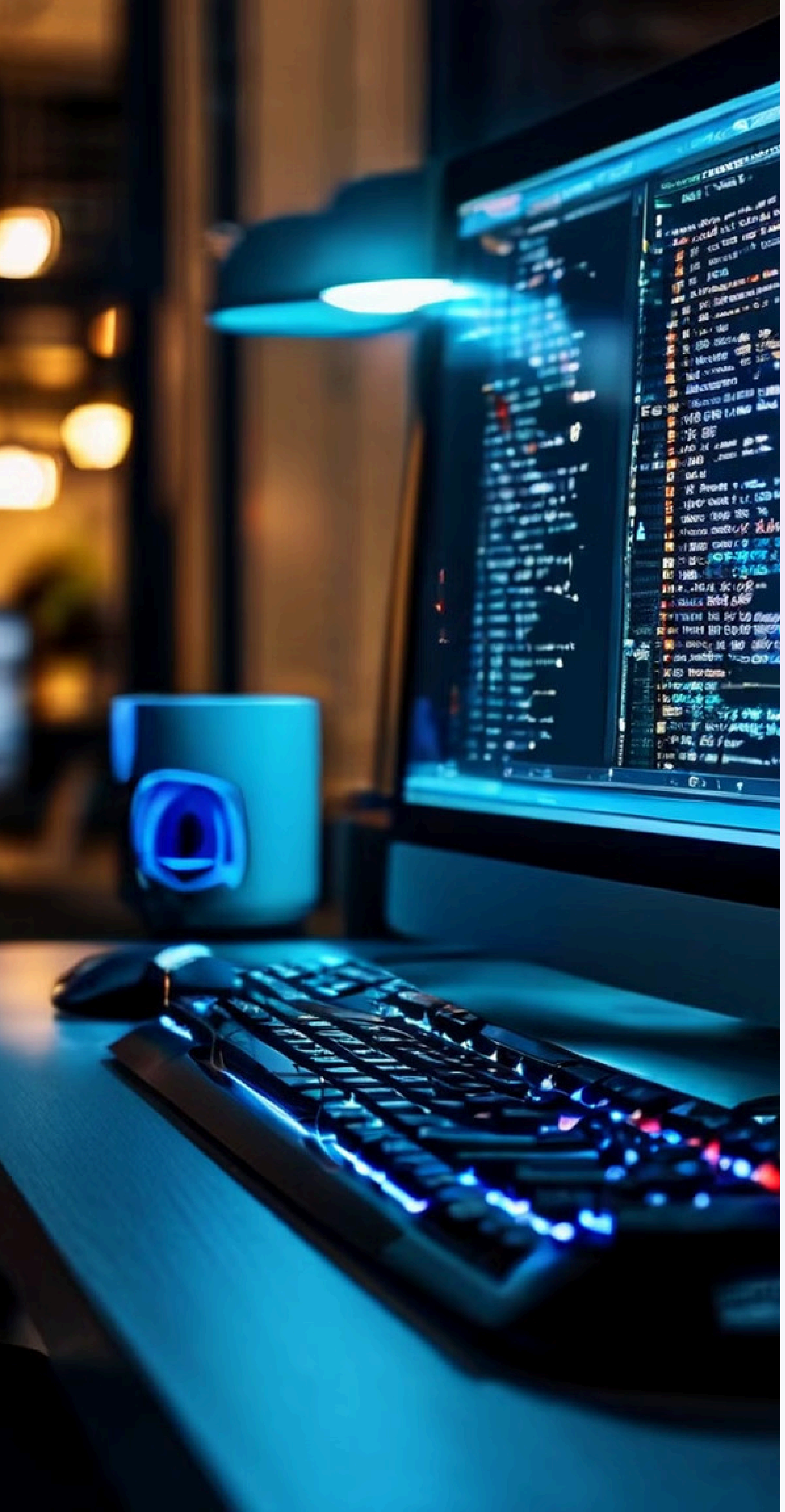
Angular's component-based architecture enables developers to build modular and reusable UI elements, making it easier to maintain and scale applications.

TypeScript Support

Angular utilizes TypeScript, a strongly typed superset of JavaScript, offering better code organization, improved maintainability, and enhanced error detection during development.

Angular CLI

Angular comes with a command-line interface (CLI) that simplifies project setup, development, and deployment, automating common tasks and enhancing productivity.



Setup and Pre-requisites

1

Node.js and npm

Install Node.js, a JavaScript runtime environment, and npm (Node Package Manager), which is used to manage packages and dependencies.

node -v

npm -v

—> Ensure you have Node.js and npm installed.

2

Angular CLI

Install Angular CLI using npm, which provides tools for creating, developing, and deploying Angular applications.

npm install -g @angular/cli

3

Create New Project and Run

ng new my-angular-app cd my-angular-app

ng serve

String Interpolation and Property Binding

String Interpolation

String interpolation allows embedding dynamic data within HTML templates using double curly braces ({{ }}). For example: **<h1>{{ title }}</h1>** (HTML file) **export class AppComponent { title = 'Hello, Angular!'; }** (ts file)

Property Binding

Property binding allows setting HTML attributes or properties based on component data using square brackets ([]). For example: **<img [src]="imageUrl">** (HTML file) **export class AppComponent { imageUrl = 'https://example.com/image.png'; }** (ts file)

Event Binding and Parent/Child Communication

1

Event Binding

Event binding allows triggering component methods in response to HTML events, such as click or change. For example:

```
<button (click)="handleClick()">Click Me</button>
```
(HTML file)

```
export class AppComponent { handleClick() { alert('Button clicked!'); } }
```
(ts file)

2

Input Properties (Parent to Child using @Input)

Parent components can pass data to child components using input properties, decorated with the @Input() decorator. Child components receive the data using the property name.

// parent.component.html

```
<app-child [childData]="parentData"></app-child>
```

// child.component.ts

```
@Input() childData: string;
```

3

Output Properties(Child to Parent using @Output)

Child components can emit events to their parent components using output properties, decorated with the @Output() decorator. Parent components can subscribe to these events and handle them accordingly.

// child.component.ts

```
@Output() childEvent = new EventEmitter<string>();
this.childEvent.emit('data');
```

// parent.component.html

```
<app-child (childEvent)="handleEvent($event)"></app-child>
```

Angular Services

Feature	Description
Data Sharing *//In the service:* **this.dataService.sharedData = 'Shared Data';** *//In a component:* **unknown link = this.dataService.sharedData;**	Services enable sharing data across multiple components, ensuring data consistency and avoiding duplication.
Reusable Logic this.http.get('**https://api.example.com/data').subscribe**(response => this.data = response);	Services encapsulate reusable logic, such as API calls, data validation, or utility functions, making it easier to maintain and reuse code.
Dependency Injection **constructor(private myService: MyService) { }** MyService is injected into the component via the constructor, allowing access to its functionality	Angular's dependency injection mechanism allows injecting services into components, making it easy to access service functionality without tightly coupling components.

Angular Directives

Structural Directives

Structural directives modify the DOM structure, adding, removing, or manipulating elements based on conditions. Examples include *ngIf, *ngFor, and *ngSwitch.

<div *ngIf ="isVisible">Content</div>

<div *ngFor="let item of items">{{ item }}</div>

Attribute Directives

Attribute directives modify the appearance or behavior of existing elements by changing their attributes or styles. Examples include ngClass, ngStyle, and ngModel.

<div [ngClass]="{'active': isActive}">Content</div>

<div [ngStyle]="{'color': isRed ? 'red' : 'blue'}">Styled Content</div>

<input [(ngModel)]="username" placeholder="Enter username">

Custom Directives

Custom directives extend Angular's functionality by creating reusable logic and behavior that can be applied to elements. They allow developers to create custom behaviors and interactions.

<p appHighlight>Hover over this text to see the highlight effect.</p>

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

Angular Forms

1

Template-Driven Forms

Template-driven forms are created using Angular directives and templates, making it easy to build forms using declarative syntax. The **ngModel** directive is central to data binding and validation in template-driven forms.

<form #form="ngForm"> <input name="name" ngModel> </form>

2

Reactive Forms

Reactive forms provide a more powerful and flexible approach to form management, using classes and objects to represent form data and control. They offer more control over form state, validation, and asynchronous operations.

this.form = this.fb.group({ name: [''] });

<form [formGroup]="form"> <input formControlName="name"> </form>

3

Validation

Angular provides built-in validation mechanisms to ensure data quality, such as required fields, email validation, and custom validation logic. Validation errors can be displayed to users to guide them in correcting their input.

this.myForm = new FormGroup({ name: new FormControl('', Validators.required) });

Angular Routing and HTTP Requests

1

Routing

Angular routing allows navigating between different parts of the application without reloading the entire page, creating a smooth and seamless user experience.

const routes: Routes =

[{ path: 'home', component: HomeComponent }, { path: 'about', component: AboutComponent }];

//Router Module:

@NgModule({ imports: [RouterModule.forRoot(routes)], exports: [RouterModule] }) export class AppRoutingModule { }

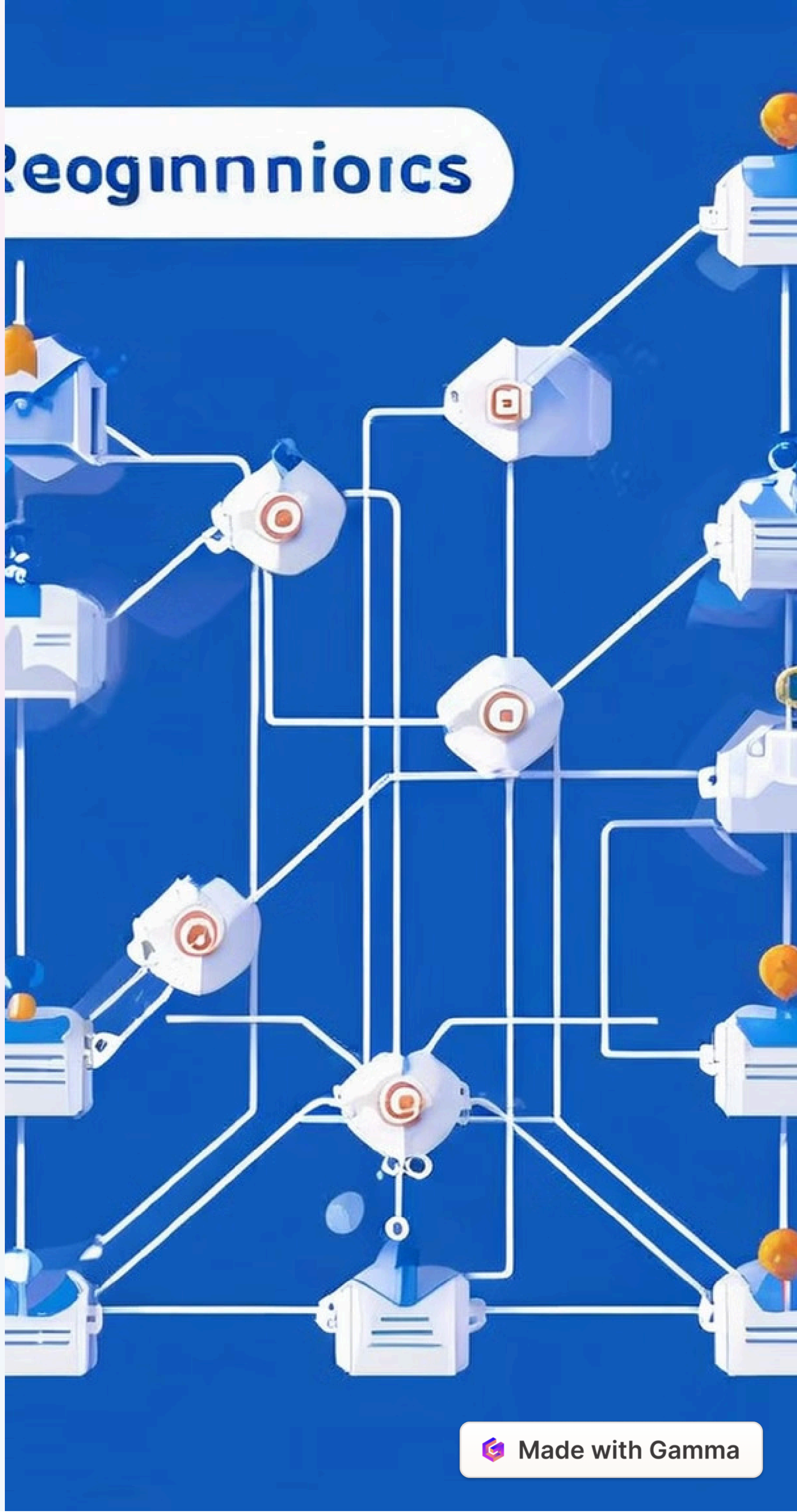
2

HTTP Requests

Angular provides the HttpClient service for making HTTP requests to external APIs or servers to retrieve or send data. This enables communication with backend systems and integration with external services.

this.http.get('https://api.example.com/data').subscribe(response => console.log(response));

(http.get sends a GET request to the specified URL, and the response is logged to the console.)



Conclusion:

Angular provides a rich set of tools and features that streamline frontend development, including efficient data binding and modular design. To maximize your use of Angular, explore its extensive **documentation** and tutorials to unlock its full potential and stay ahead in web development