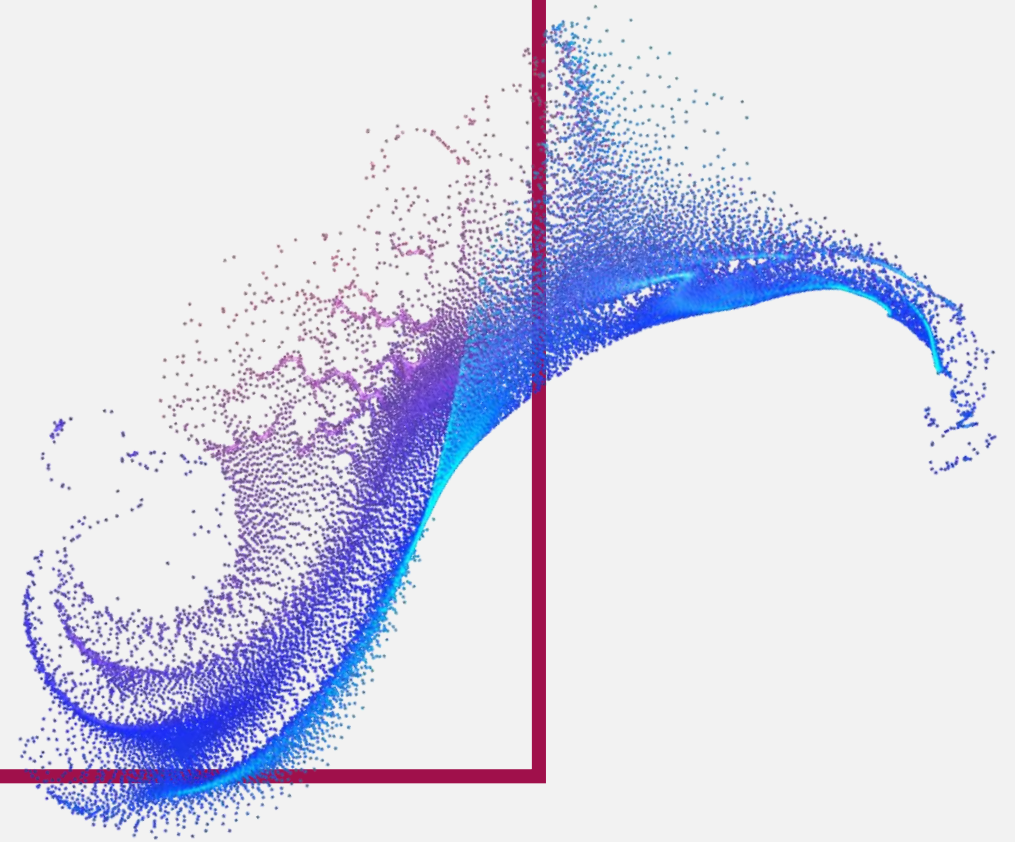


Angular Intensive

The modern web developer's platform



Jun 2024

**Nash
Tech.**

Agenda

1. Introduction to Angular
2. Angular Modules and Services
3. Routing and Forms
4. Advanced Topics and Best Practices

Introduction to Angular

- Overview of Angular
- Setting Up the Environment
- Project Structure and Architecture
- Components and Templates

Angular Modules and Services

- Modules
- Directives and Pipes
- Services and Dependency Injection

Routing and Forms

- Routing and Navigation
- Angular Forms

Advanced Topics and Best Practices

- Angular Change Detection
- HttpClient and APIs
- State Management
- Testing Angular Applications
- Best Practices

Overview of Angular



What is Angular?



Key features



History and evolution

What is Angular?

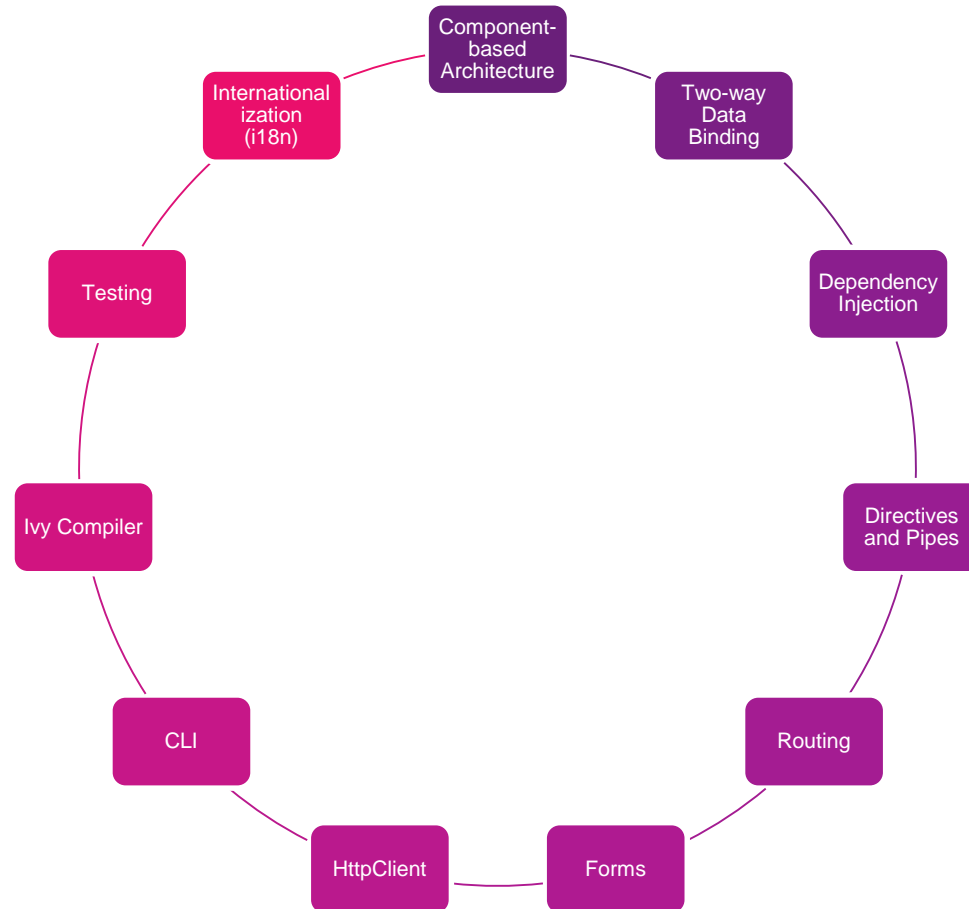


Angular is a platform and web framework for building single-page client applications using HTML and TypeScript



Angular is developed and maintained by Google

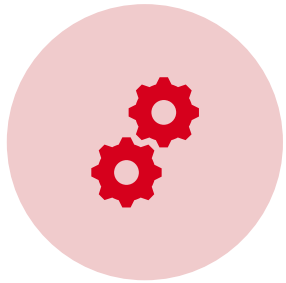
Key Features



History and Evolution



Setting Up the Environment



Installing Node.js and
Angular CLI



Creating your first
Angular project

Project Structure



Understanding the file structure

src/

app/

assets/

environments/



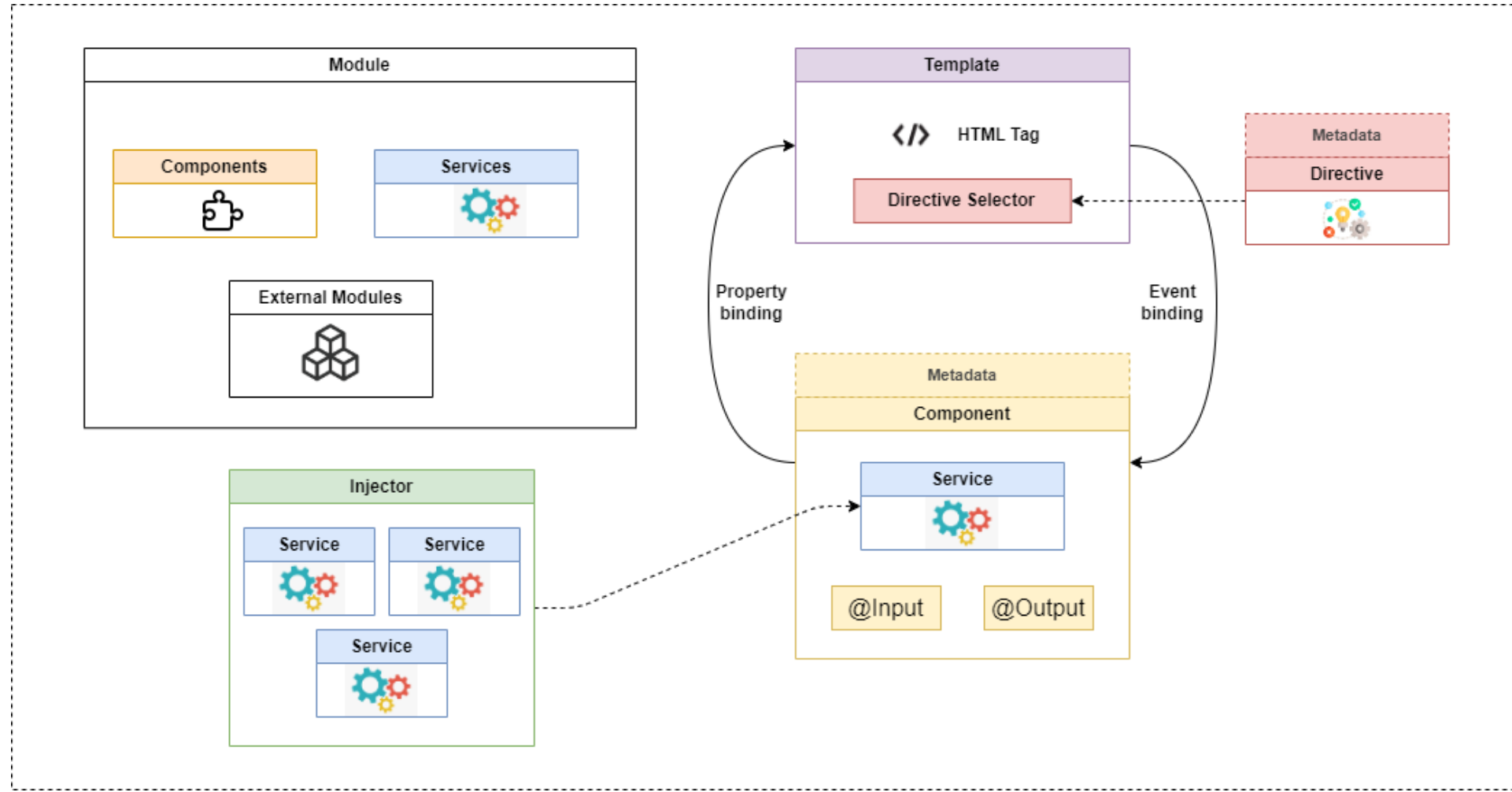
Important files and directories

angular.json

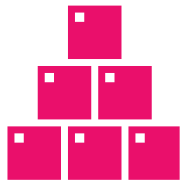
package.json

tsconfig.json

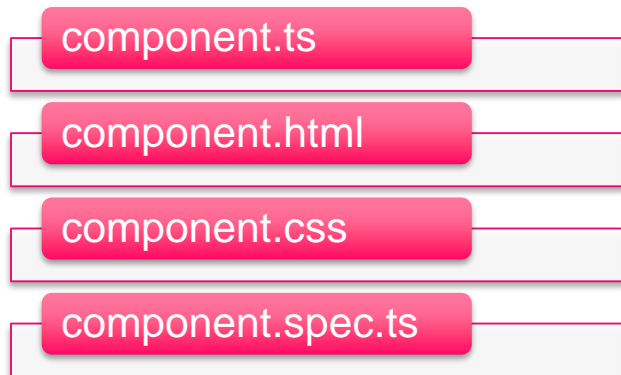
Angular architecture



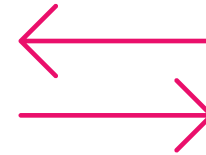
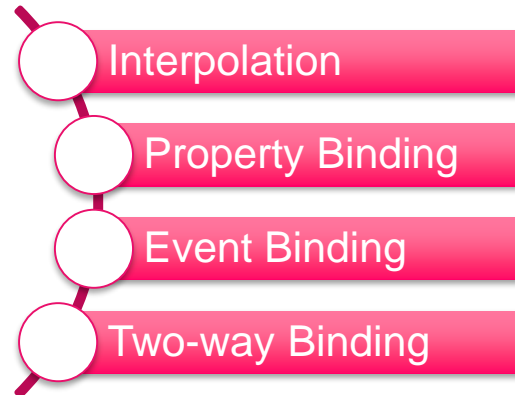
Component and Templates



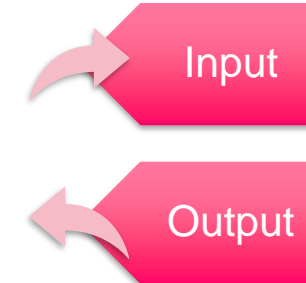
Creating and using components



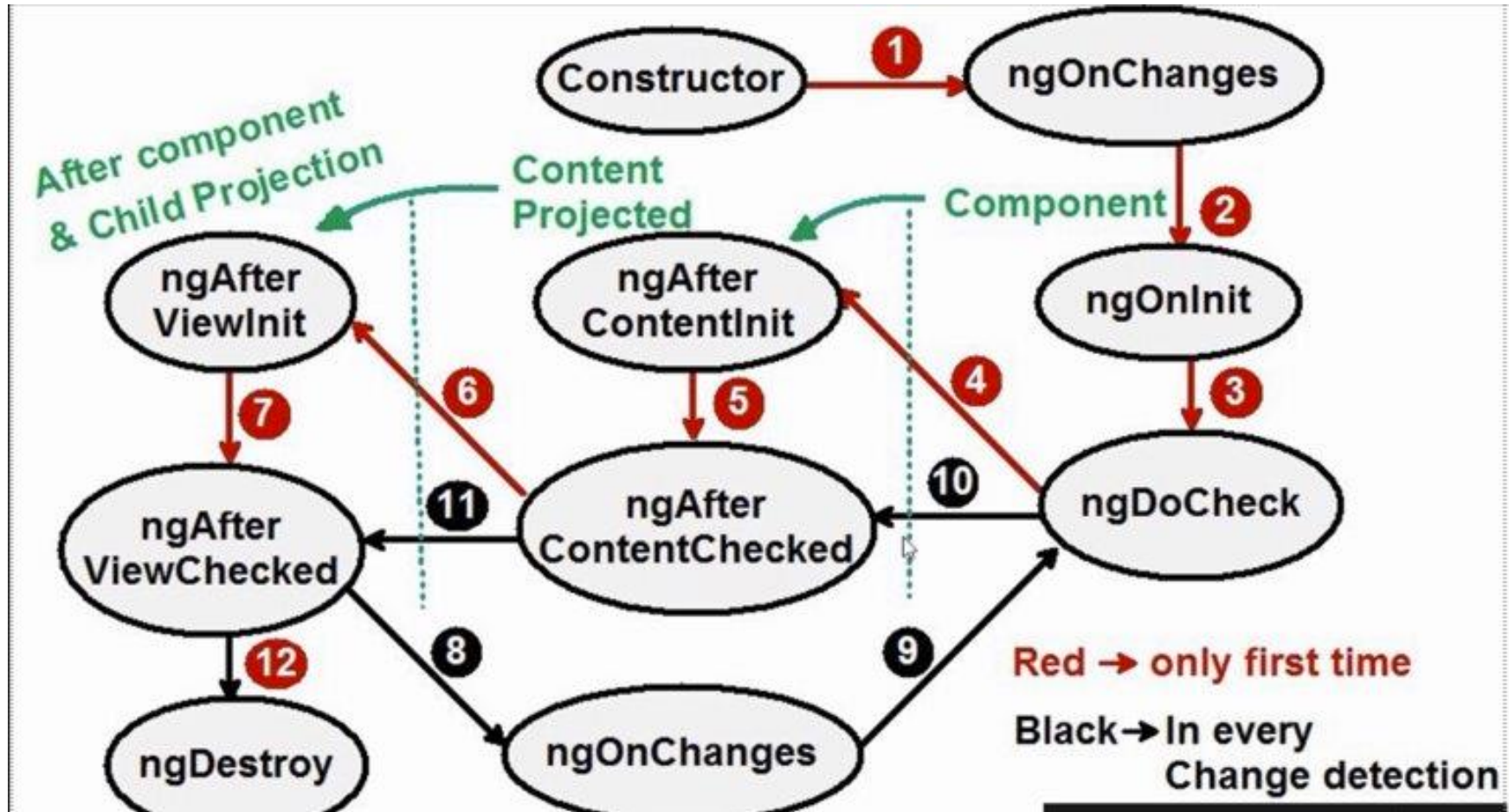
Data binding



Component Communication



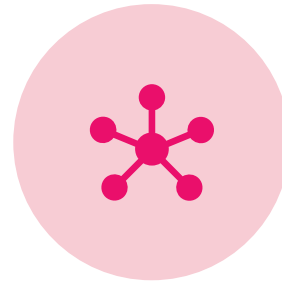
Component Lifecycle



Modules

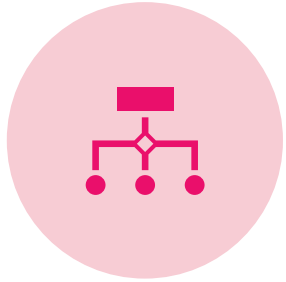


Understanding
NgModules



Core and Feature
Modules

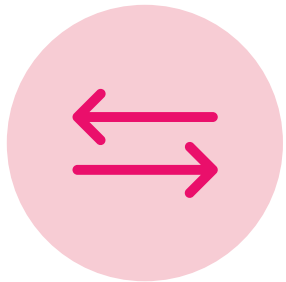
Introduction to Modules



They provide a compilation context for components, directives pipe, and service providers



Organizing an application into cohesive blocks of functionality



They can import/export functionality from/for use by other modules.



However, in the new Angular version, this approach is no longer preferred.

Core Module and Feature Modules

BrowserModule

- Includes platform-specific services that are essential for interacting with the DOM and other browser APIs
- Sets up the dependency injection system and application initialization process.
- Provides services related to change detection and rendering of components.
- Includes the CommonModule.
- For root module

CommonModule

- Provides common directives and pipes such as **ngIf**, **ngFor**, **DatePipe**, and **CurrencyPipe**
- For feature modules

Directives



Components: Used with a template. It's the most common directive type



Attribute directive: Change the appearance or behavior of an element, component, or another directive.



Structural directive: Change the DOM layout by adding and removing DOM elements

Pipes

Built-in pipe

date

number

currency

titlecase

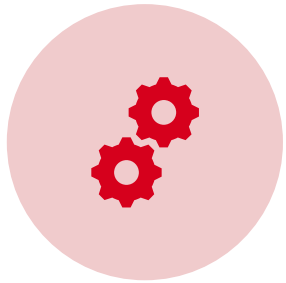
lowercase

async

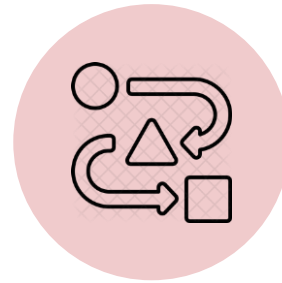
Custom pipe

- **name:** to use in template bindings, cannot contain hyphens
- **standalone:** Do not need to be declared in a Module
- **pure:**
 - ***True by default*** - transform function is invoked only when its input arguments change
 - **False** - function is invoked on each ***change-detection cycle***

Services and Dependency Injection



Creating and using
services

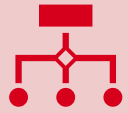


Dependency Injection
(DI)

Services



Is a class that provides specific functionality that can be shared across multiple components



Use dependency injection to provide an instance of the service to the components that require it



Service is typically use for

Encapsulating Business Logic
Data Management
Reusability

Dependency Injection



DI is a **design pattern** used to implement IoC (Inversion of Control).
Allowing a class to **receive its dependencies** from an external source rather than creating them itself.

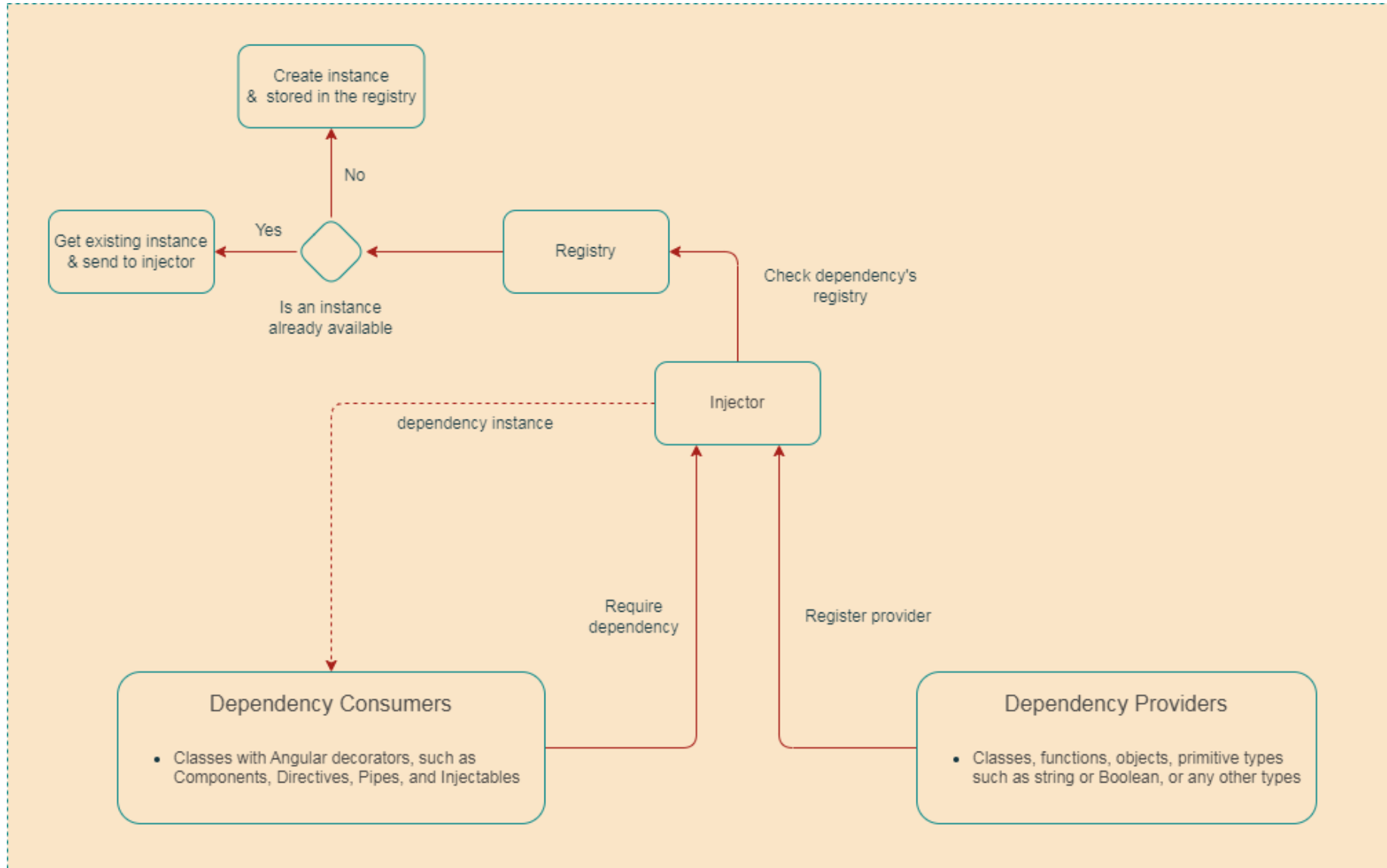


@Injectable decorator to make a class as available to be injected.

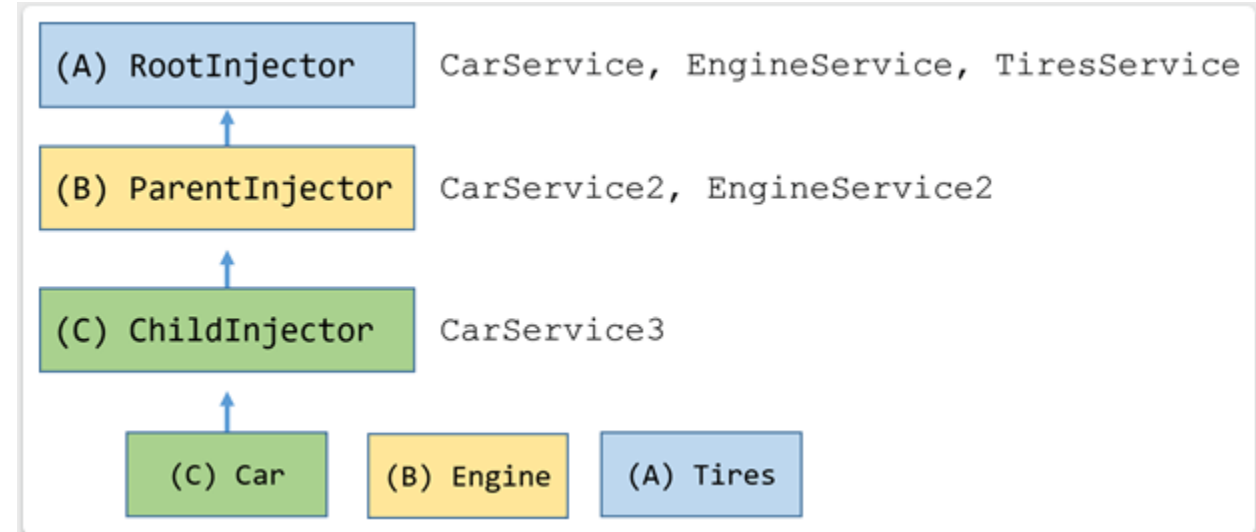
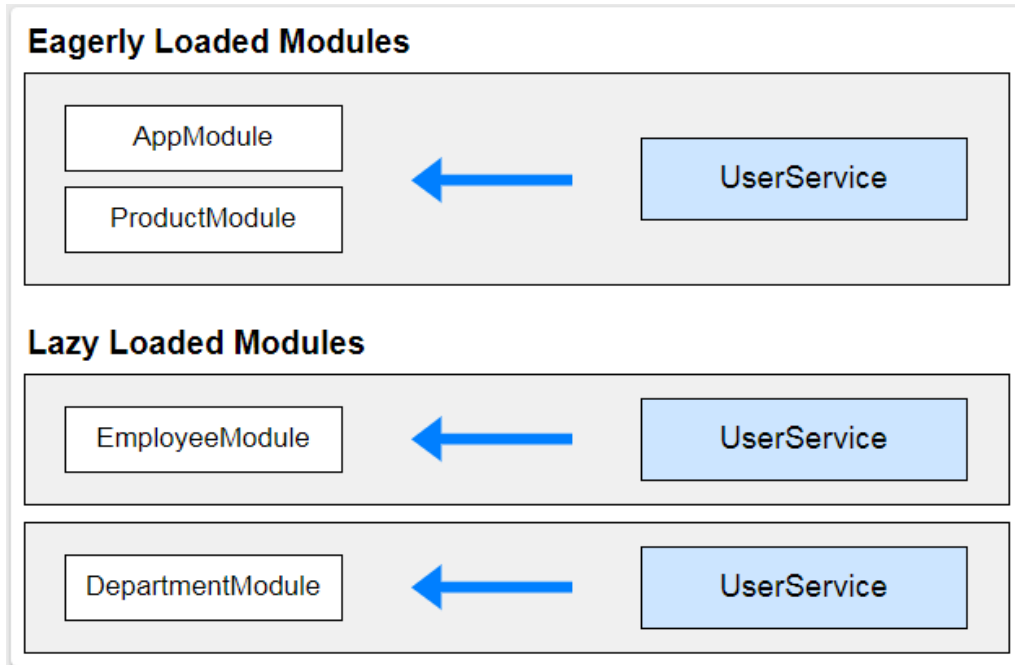


Registering service in a **module's providers** array or using **providedIn** syntax for tree-shakeable services.

Dependency Injection



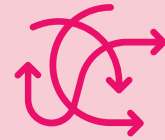
Dependency Injection



Routing and Navigation



Introduction to Angular Routing



Configuring routes



Route Guards



Lazy loading modules

Introduction to Angular Router



Angular is a platform and framework for building single-page client applications (SPA) using HTML and TypeScript.

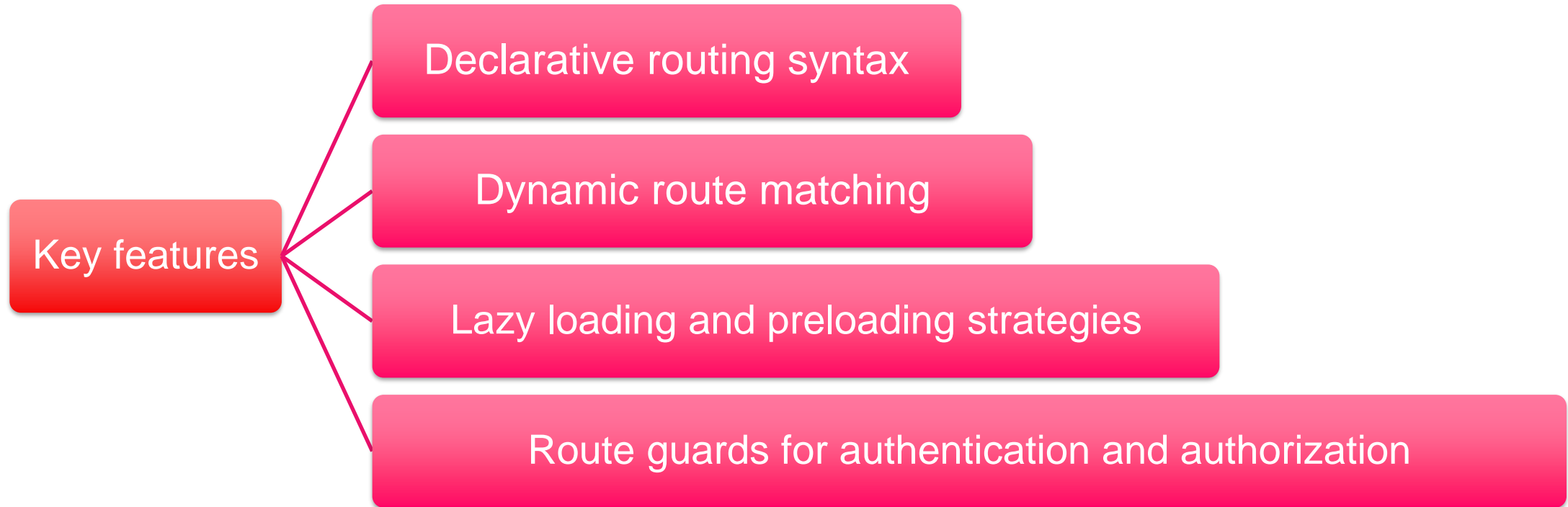


The process of navigating through different views or pages in a web application



Manages navigation and ensures the correct view is displayed for a given URL

Key Features of Angular Router



Configuring routes



<router-outlet> is a directive that acts as a placeholder in the DOM where the matched component will be displayed



routerLink is a directive used in Angular templates to define navigation paths




routerLinkActive is used to apply a CSS class to the active link



Programmatic Navigation with **Router** service

Route Guards

Route guards are Angular interfaces that allow you to **control access** to routes in your application



They help in protecting routes from **unauthorized access** and ensuring proper conditions are met before navigating



Types of Route Guards:

CanActivate

CanActivateChild

CanDeactivate

Resolve

CanLoad

Lazy Loading

Lazy loading is a design pattern used to delay the initialization of a resource until it is needed.

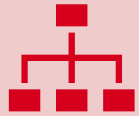


Improves application performance by loading only the necessary modules.



Reduces initial load time and enhances user experience.

Nested Routes



Nested routes allow for a hierarchical structure of routes, where routes can have child routes



Define child routes within the parent route using the **children** property.



Child components are displayed within the parent component's template where the `<router-outlet>` is placed

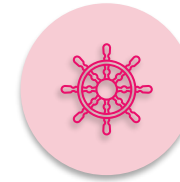
Angular Forms



What are Forms?



Template-driven
forms



Reactive
forms

What are Forms?



Forms are essential for collecting user input in web applications.



They allow users to enter data that can be sent to a server for processing.



It offers two main approaches: Template-driven forms and Reactive forms

Template-driven Forms



Built using Angular's
template syntax



Suitable for simple
forms with minimal
logic

Key Features of Template-Driven Forms

Two-Way Data Binding: ngModel Directive

Built-in Form Validation: required, minlength, maxlength, pattern

Form Grouping: ngForm Directive

Form Submission Handling: ngSubmit Directive

Dynamic Form Control Classes: Angular automatically adds classes like ng-valid, ng-invalid, ng-touched, ng-dirty to form controls based on their states

Reactive Forms



Built using Angular's reactive programming approach



Suitable for complex forms with dynamic form controls and extensive validation logic

Key features of Reactive Form

Each form input is represented by an instance of the **FormControl** class

Form controls can be grouped together using the **FormGroup** class.

Adding and Removing Controls Dynamically

FormArray is used to manage an array of form controls, useful for forms with repeated sections

Custom Validators: Define custom validation logic using functions

FormControl and **FormGroup** provide **value** and **status** change *observables*.

The **FormBuilder** service simplifies the creation of form controls and groups

Angular Change Detection

Angular automatically checks the application state for changes and updates the DOM accordingly.



Zone.js

To intercept asynchronous operations
(e.g., events, HTTP requests)

When an async operation completes,
Zone.js triggers change detection

Change Detection Strategies

Default Strategy

- Runs change detection for every component in the component tree.
- Suitable for simple applications.
- **Can lead to performance issues in large applications**

OnPush Strategy

- Runs change detection only when the component's inputs change or when an event originates from within the component.
- Improves performance by reducing unnecessary checks
- **Requires a good understanding of how data flows in the application.**

Detecting Changes Manually

ChangeDetectorRef is a service provided by Angular that allows you to control change detection manually

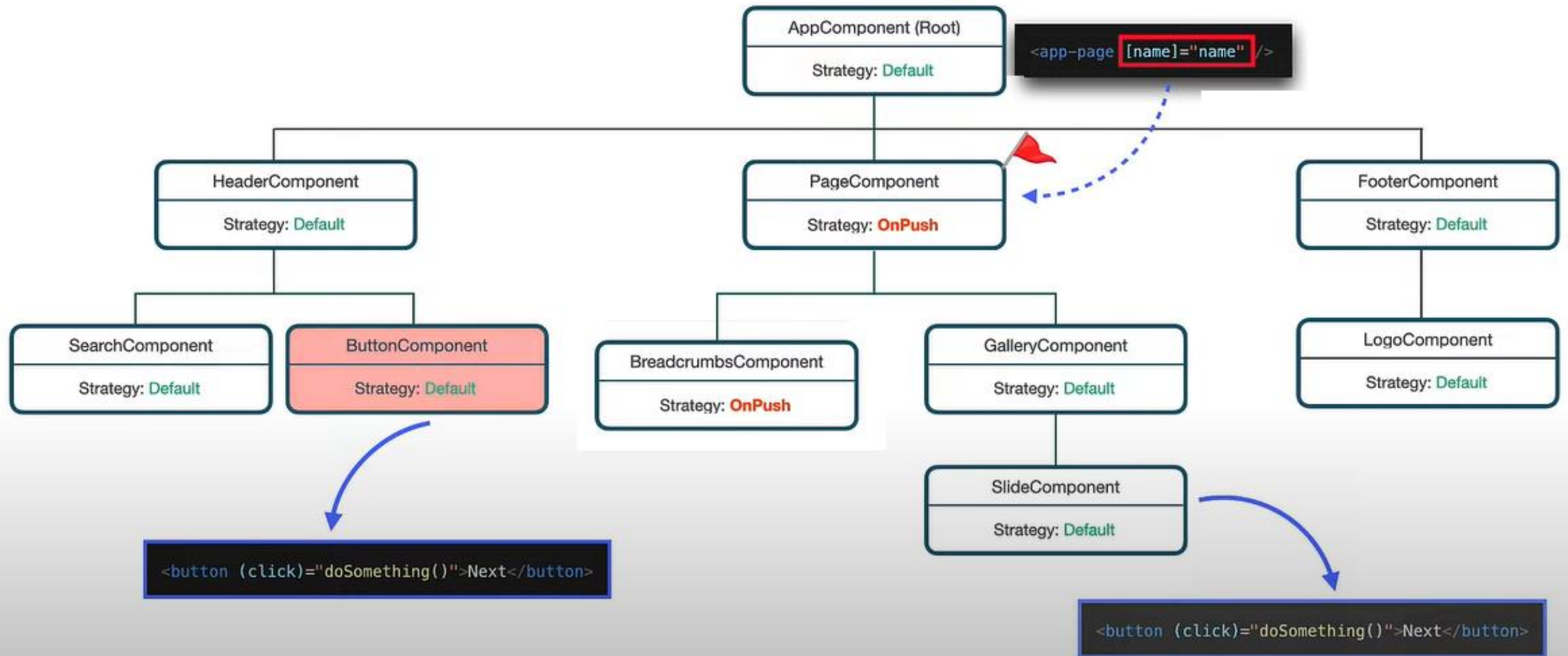
markForCheck(): Marks the component and its children to be checked during the next change detection cycle.

detectChanges(): Triggers change detection for the component and its children immediately.

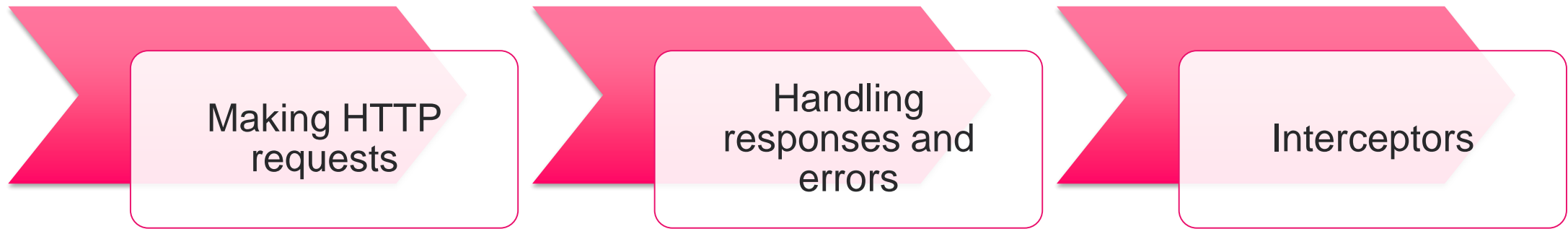
detach(): Detaches the component from the change detection tree

reattach(): Reattaches the component to the change detection tree.

ZoneJS



HttpClient and APIs



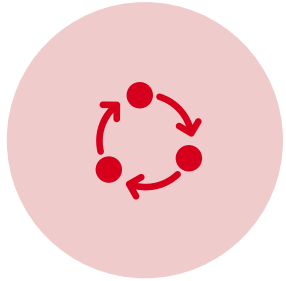
Interceptors

Interceptors are a way to inspect and transform HTTP requests and responses

They are implemented as Angular services that intercept HTTP calls before they reach the server or the application

Useful for tasks such as logging, authentication, and setting headers

State Management

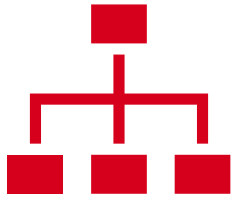


Introduction

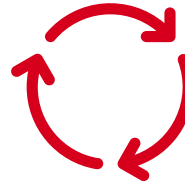


The RxJS library

State Management in Angular



State management refers to the practice of managing the state (data) of an application in a structured and predictable way



Ensures that the application state is consistent and predictable.



In Angular, state refers to the data that determines the behavior and appearance of the application at any given time.

Types of State

Local State

- Managed within individual components
- **Pros:** Simplicity and ease of use for small, isolated pieces of state
- **Cons:** Difficult to share state between components

Global State

- Shared across multiple components and services
- **Pros:** Centralized management, easy state sharing
- **Cons:** More complex to implement and manage

RxJs in State Management

Managing Component State

- Use observables to manage and react to local component state changes.

Global State Management

- Use subjects and observables to manage application-wide state.

Handling Side Effects

- Use RxJs operators (**tap**, **switchMap**, **mergeMap**, **catchError**) to handle side effects like HTTP requests.

Core Concepts of RxJs

1

Observables: An Observable is a representation of any set of values over any amount of time. They are the foundation of RxJs

2

Subscription: Observables are lazy; they do not emit values until they are subscribed to

3

Operators: are functions that enable complex manipulation of observable streams. There are various operators like creation operators, transformation operators, filtering operators, etc

4

Subjects: A Subject is a special type of Observable that allows values to be multicasted to multiple observers

Testing Angular Applications

Ensuring	Ensuring the quality, reliability, and maintainability of the codebase
Catching	Catching bugs early in the development process, reducing costs and time for fixes
Providing	Providing confidence to developers and stakeholders that the application works as expected

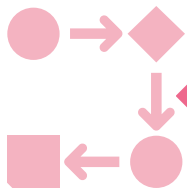
Types of Testing



Unit testing



Integration Testing



End-to-end (E2E) testing

Tools and Frameworks

Jasmine	A popular framework for writing unit and integration tests
Karma	A test runner that works seamlessly with Jasmine
Protractor	An end-to-end test framework specifically designed for Angular applications
Angular Testing Utilities	Built-in tools provided by Angular to facilitate testing

Angular Testing Utilities

TestBed

- A powerful utility for setting up and configuring the environment for unit tests.

HttpClientTestingModule

- Provides a mock HTTP client for testing HTTP services

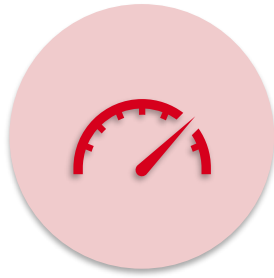
Additional Tools

Cypress	<p>Overview: A powerful end-to-end testing framework not limited to Angular.</p> <p>Features: Fast, reliable tests that run directly in the browser</p>
Jest	<p>Overview: A delightful JavaScript testing framework with a focus on simplicity</p> <p>Features: Fast and reliable with a zero-config setup.</p>
Storybook	<p>Overview: A tool for UI component development and testing.</p> <p>Features: Allows you to build, test, and showcase components in isolation</p>

Best Practices



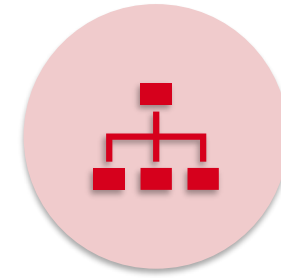
Scalability &
Maintainability



Performance
optimization



Security
considerations



Code organization
and style guides

Scalability & Maintainability

Project Structure

- Module Structure
- Core and Shared Modules
- Lazy Loading

Component Design

- Single Responsibility Principle
- Smart and Dumb Components
- Use OnPush Change Detection

Service & Dependency Injection

- Singleton Service
- Scoped Service
- Use @Injectable Decorator

Scalability & Maintainability

State Management

- Reactive State Management
- Avoid Heavy Logic in Components

Template Practices

- Avoid Complex Logic in Templates
- Use Safe Navigation Operator (?.)
- Leverage Angular Directives

Forms

- Reactive Form
- Form Validation
- Custom Validators

Performance Optimization

Lazy Loading and
Preloading

Optimize load times using lazy loading and preloading strategies

AOT Compilation

Use Ahead-of-Time (AOT) compilation to improve performance

Track by in ngFor

Use trackBy function in ngFor to optimize rendering of list items

Error Handling

Global Error Handling

Implement a global error handler using ErrorHandler.

HTTP Error Handling

Handle HTTP errors with interceptors

User-Friendly Messages

Display user-friendly error messages and logs

Security Best Practices



Sanitize Inputs

Always sanitize user inputs to prevent XSS attacks.



Use Angular's DomSanitizer

Use DomSanitizer to safely inject HTML content.



Content Security Policy (CSP)

Implement CSP headers to protect against

Thanks for joining us