

Dell IT Academy

TypeScript e Angular

Instrutor: Júlio Pereira Machado (julio.machado@pucrs.br)



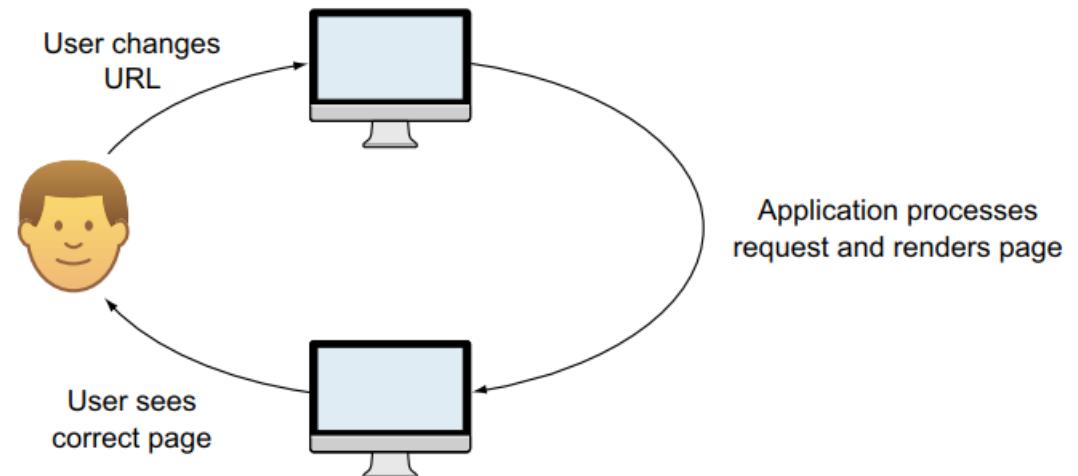
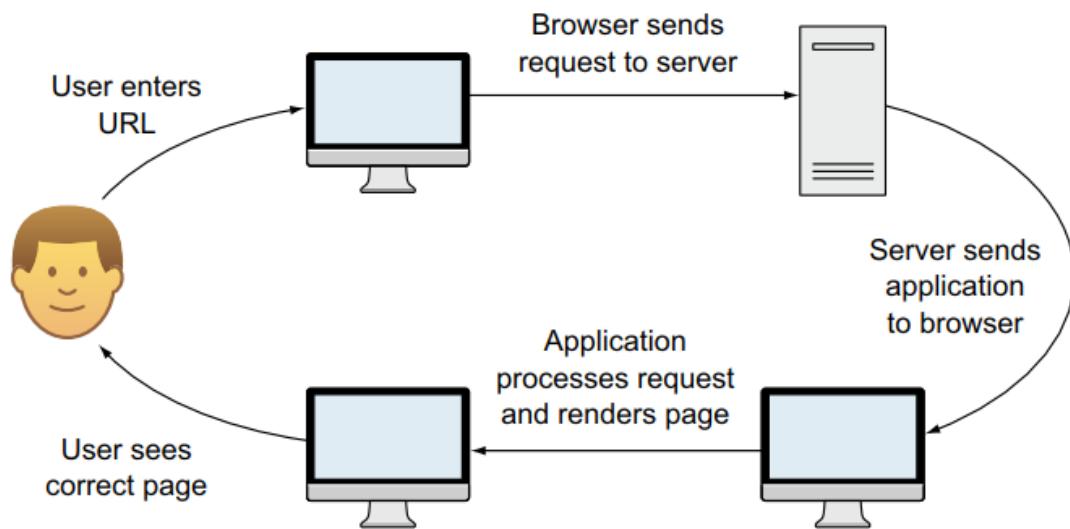
Single Page Applications



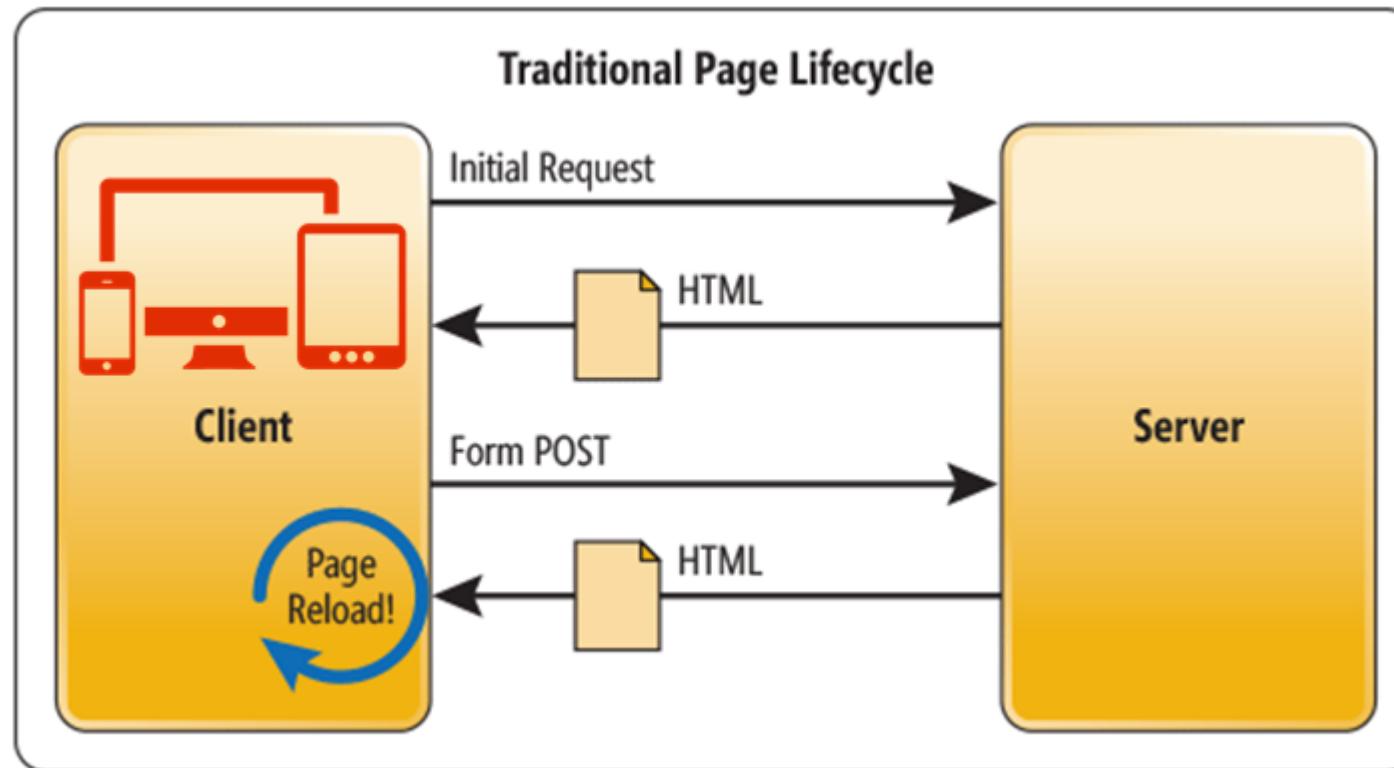
Single Page Applications

- Uma SPA ou Single Page Application é uma aplicação web que se baseia em uma única página
- Página inicial a ser renderizada funciona como uma “casca”, um pedaço de HTML que irá conter as diferentes views parciais
- Uma SPA se utiliza dos mecanismos de renderização parcial de páginas de forma assíncrona sem a necessidade de uma nova requisição completa a um servidor
- https://2023.stateofjs.com/en-US/usage/#js_app_patterns

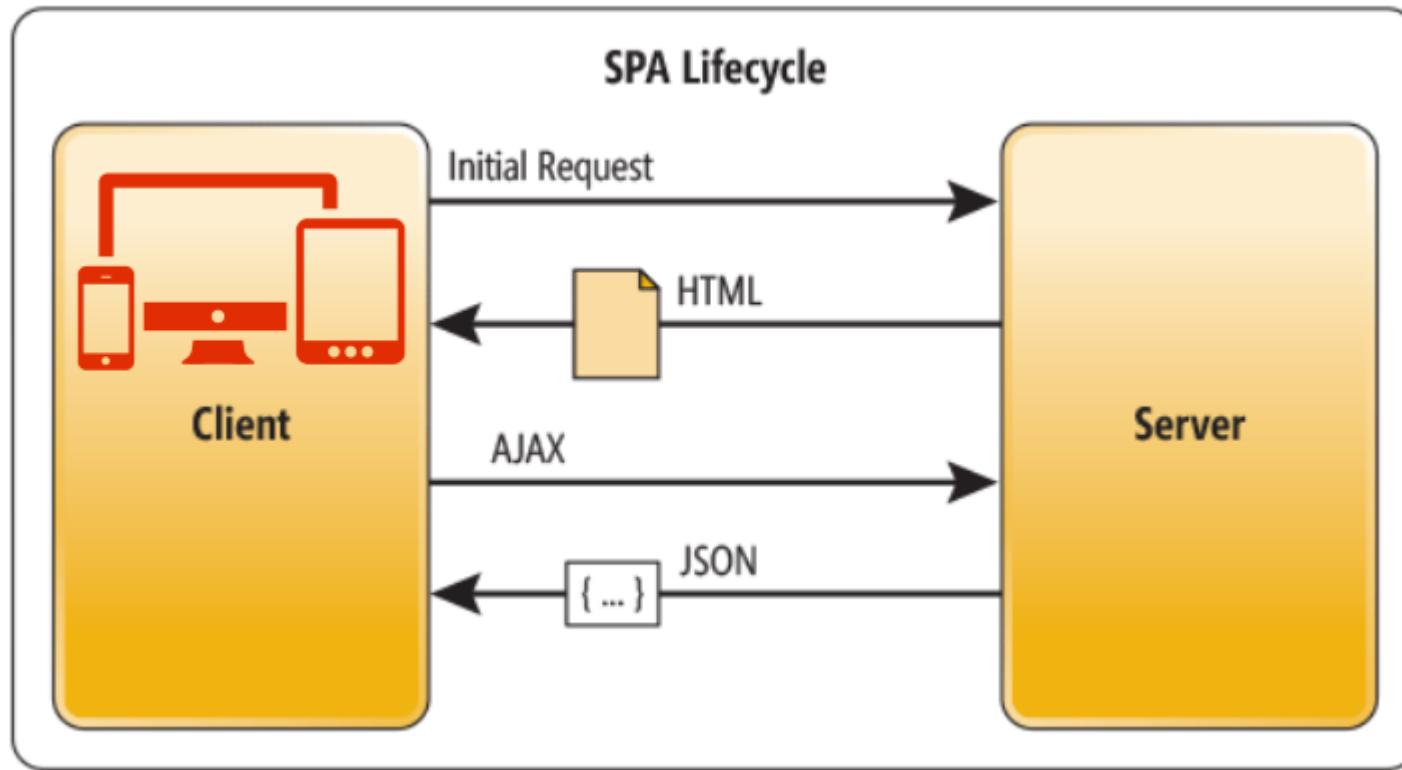
Single Page Applications



Single Page Applications



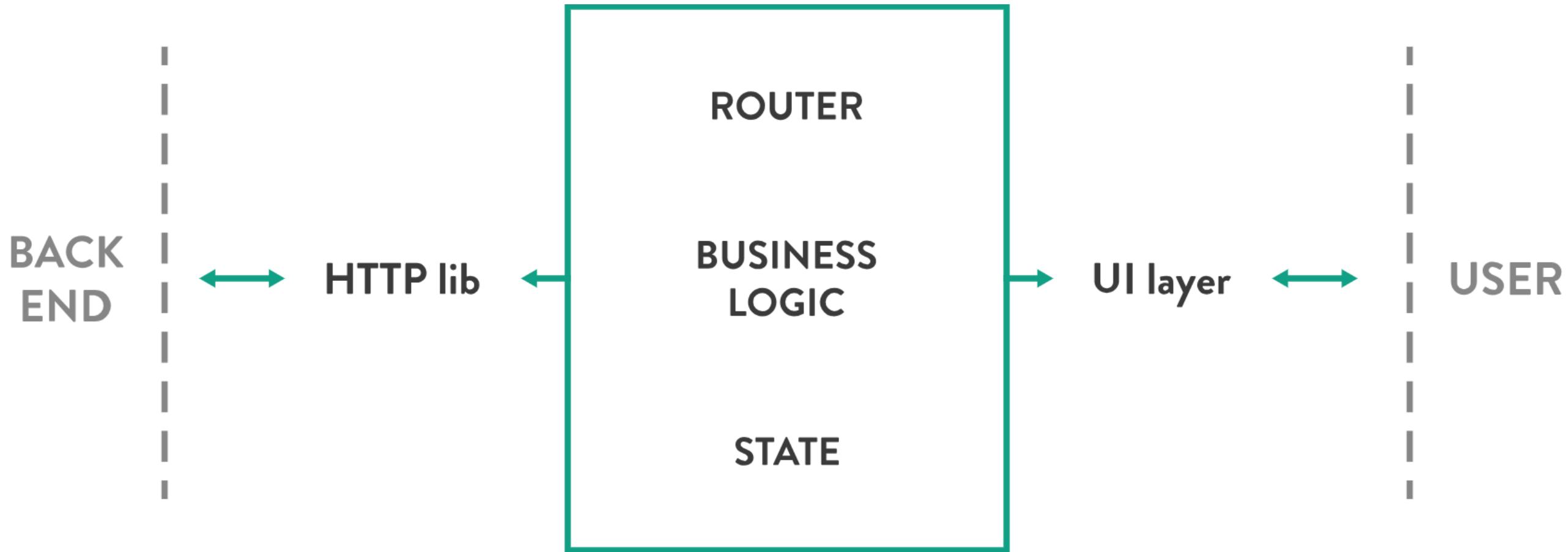
Single Page Applications



Single Page Applications

- Recursos usualmente utilizados por frameworks para criação de SPAs:
 - Views baseadas em template
 - Roteamento entre views
 - Gerenciamento de estado
 - Requisições assíncronas para o backend

Single Page Applications



Fonte: <https://marcobotto.com/blog/frontend-javascript-single-page-application-architecture/>

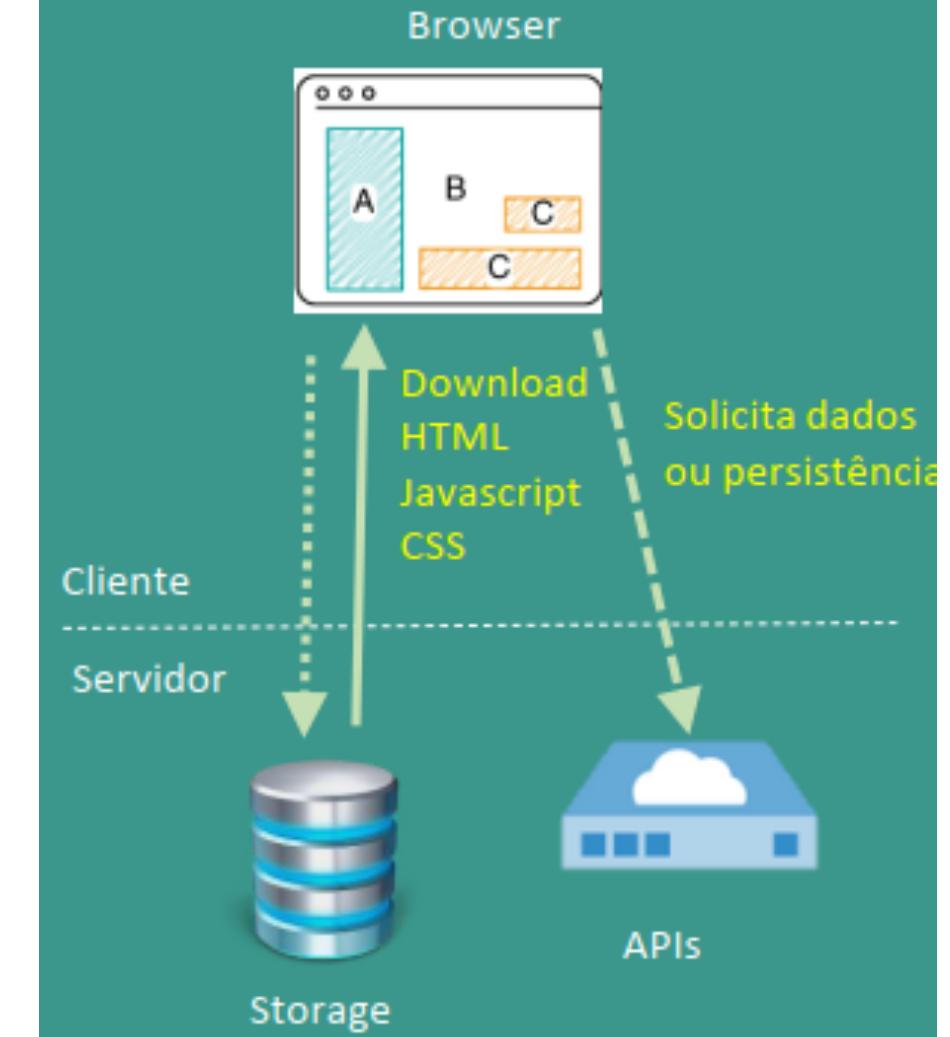
Renderização

- A renderização em tecnologias frontend refere-se ao processo de exibir e atualizar visualmente os elementos de uma interface de usuário em um dispositivo, utilizando um navegador da web ou em um aplicativo móvel. É a maneira como os dados são convertidos em elementos visuais que os usuários podem ver e interagir.
- Basicamente dois tipos de renderização:
 - CSR – cliente-side rendering
 - SSR – server-side rendering

Renderização

- Client Side Rendering (CSR)
 - Renderização do HTML é realizado pela engine do Angular dentro do browser
 - <https://www.youtube.com/watch?v=4-Lel1oaV7M>

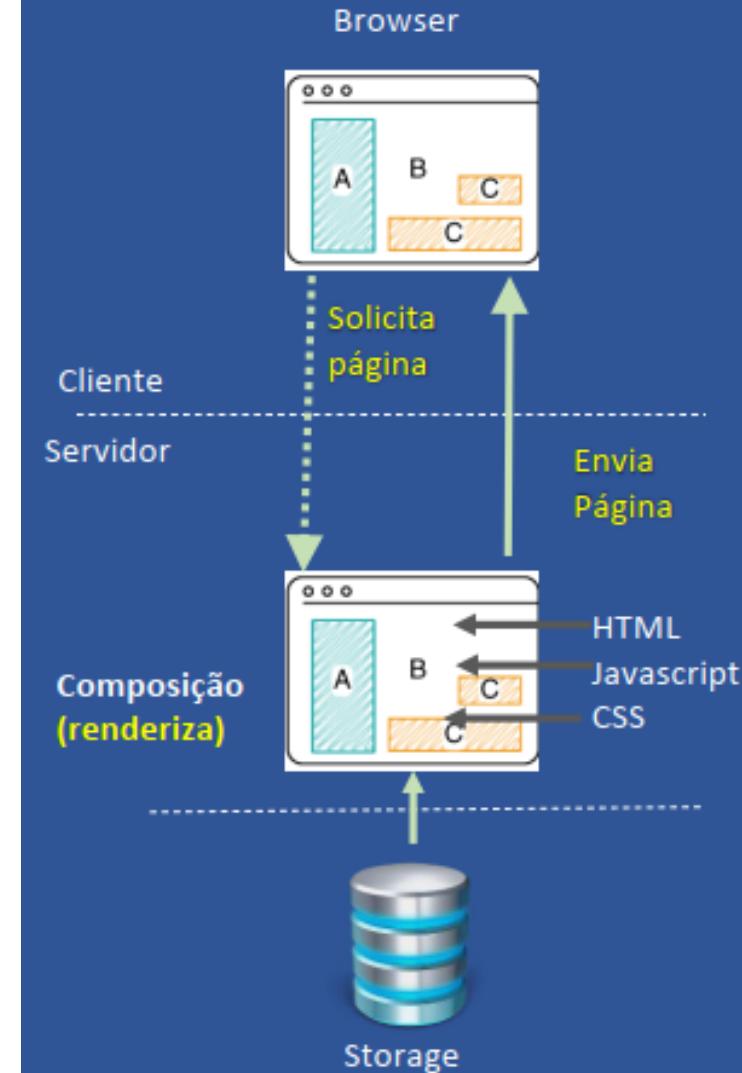
CSR
Renderização lado cliente



Renderização

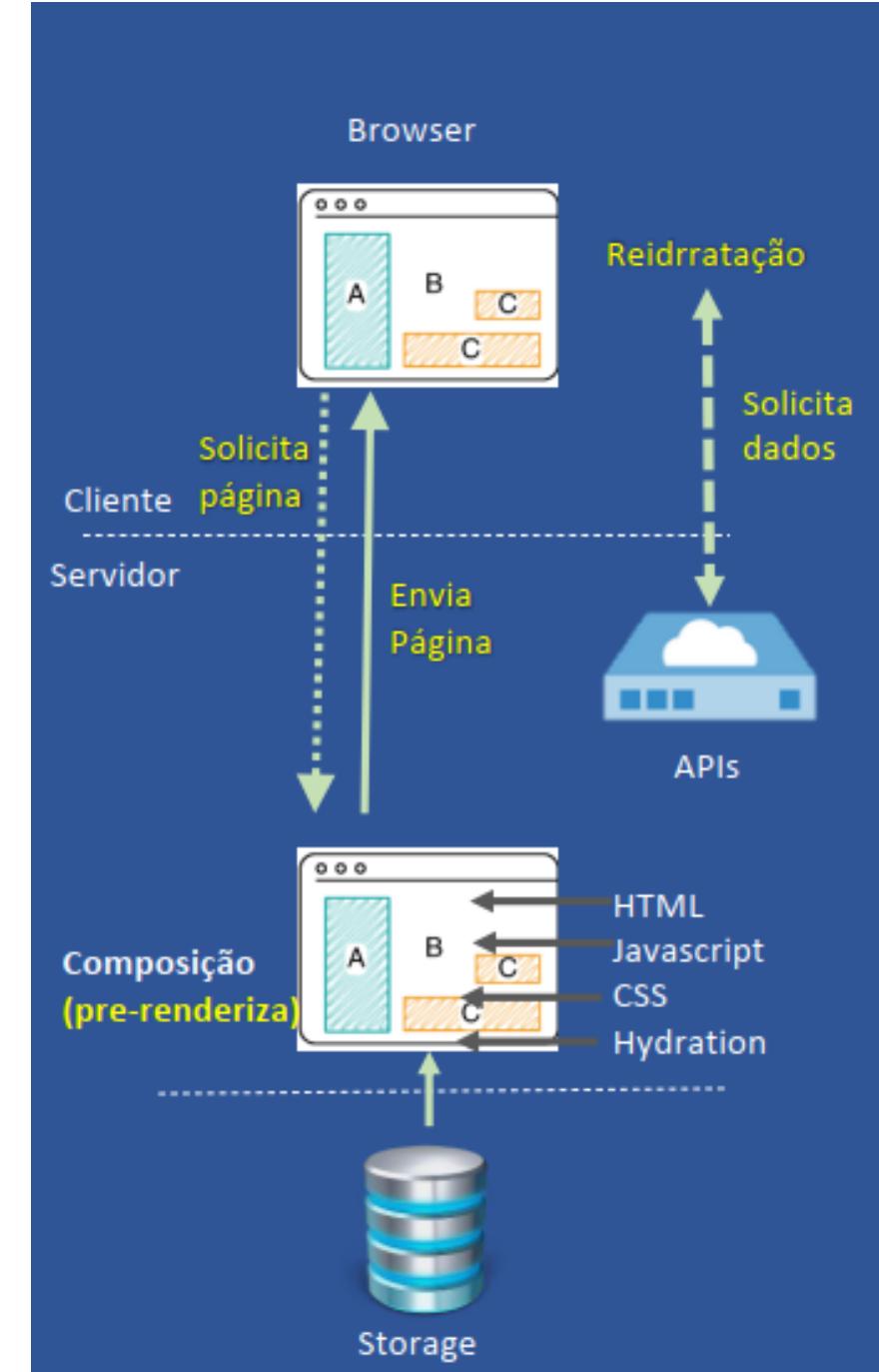
- Server Side Rendering (SSR)
 - Renderização do HTML é realizado pela engine do Angular no servidor
 - <https://www.youtube.com/watch?v=0bvo6UKkNDA>

SSR
Renderização lado servidor



Renderização

- Server Side Rendering (SSR)
 - Renderização do HTML é realizado pela engine do Angular no servidor
 - Pode envolver processo de reidratação de páginas
 - <https://www.youtube.com/watch?v=y5CpKiH-3J8>



Renderização

- Static Site Generation (SSG)
 - Renderização do HTML é realizado em tempo de construção (build) da aplicação
 - <https://www.youtube.com/watch?v=1zhT23VDVDc>

Angular



Angular

- Framework para o lado cliente de aplicações Web
- Código aberto
- <https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/>

<https://angular.dev/> 

Angular

- O aspecto mais importante do Angular é Component Driven Development
- Componentes são um dos elementos centrais do desenvolvimento de uma aplicação Angular
- Suporte à injeção de dependência

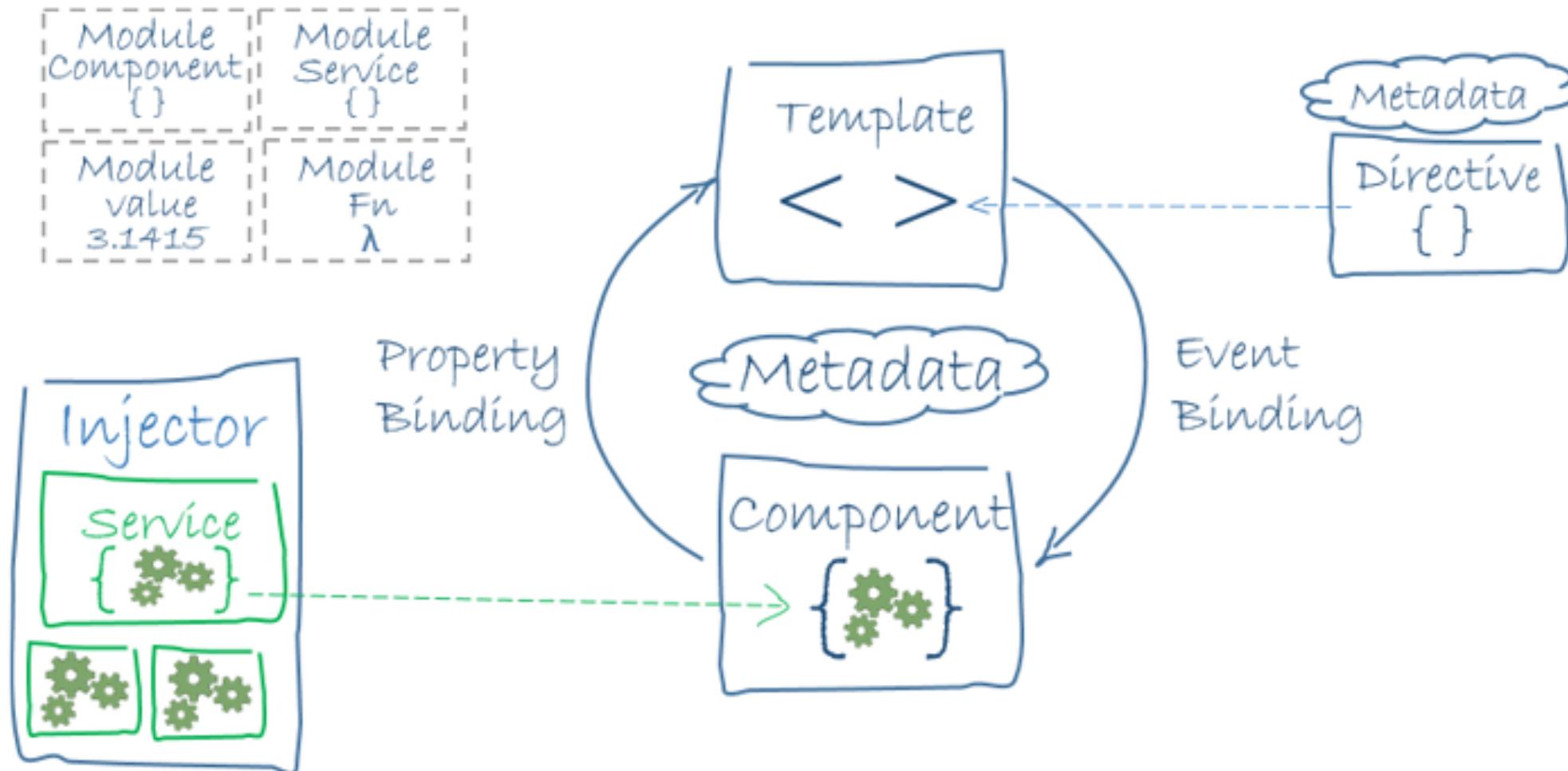
Angular

- IMPORTANTE:
- Angular possui um guia de estilo com dicas de como organizar um bom projeto
- <https://angular.dev/style-guide>

Angular CLI

- Angular CLI é uma ferramenta de linha de comando para gerenciar um projeto Angular
 - Criar e configurar novos projetos
 - Gerar partes da aplicação através de templates
 - Executar a aplicação em modo de desenvolvedor
 - Empacotar a aplicação para distribuição
- Documentação:
 - <https://angular.dev/tools/cli>
 - <https://angular.dev/cli>

Angular – Blocos de Construção



Angular – Principais Conceitos

MODULE	COMPONENT	DIRECTIVES	SERVICE
Estrutura as diferentes partes da aplicação. Módulo para cada elemento da solução.	Classe que representa a junção da lógica para manipulação das views e dos templates que representam a view.	Extensões ao HTML, utilizadas para manipular o DOM.	Classe com regras de negócio reutilizáveis independentes dos componentes.

Angular – Conceitos Adicionais

CONCEITO	DESCRIÇÃO
Template	Página HTML com marcações de diretivas que definem a parte renderizável de um componente; definem o aspecto visual de uma <i>view</i>
Pipes	Objetos de transformação de dados entre <i>model</i> e <i>view</i>
Data Binding	Elementos de ligação entre <i>model</i> e <i>view</i> tanto para propriedades quanto eventos
Injeção de Dependência	Mecanismo de controle do gerenciamento das dependências entre os diferentes módulos, componentes e serviços
Decorators	Mecanismo do JavaScript (inicia por @) que permite encapsular um elemento dentro de outro (tal como composição de funções); muito utilizado para definir opções de comportamento e metadados sobre um objeto; implementação do padrão <i>Decorator</i>
Router	Implementa um serviço para o controle de navegação entre a estrutura de <i>views</i> da aplicação; mapeia caminhos de URLs em <i>views</i>
Signal	É um wrapper em torno de um valor que habilita notificações de alterações nesse valor; implementação do padrão <i>Observer</i>

Angular - Componentes

- Classe que representa a junção da lógica para manipulação das views e dos templates que representam as views
 - Propriedades do estado do componente para vincular às views
 - Funções para associar como tratadores de eventos das views
 - Funções de gerenciamento do ciclo de vida dos componentes
- Toda aplicação tem pelo menos um componente, o chamado componente raiz
- Documentação:
 - <https://angular.dev/guide/components>

Angular - Componentes

- Para criar um novo componente:
 - Classe decorada com `@Component()` especificando as propriedades do componente
 - Via Angular CLI usar `ng generate component nomeComponente`
- Principais propriedade para `@Component`
 - `selector` – seletor CSS que define a forma de consumo do componente em um template
 - `template` – define de forma inline o template a ser utilizado para a construção da view
 - `templateUrl` – define o arquivo de template a ser utilizado para a construção da view
 - `styles` – define de forma inline os estilos CSS locais a serem utilizados no template da view
 - `styleUrls` – definem os arquivos de estilos CSS locais a serem utilizados no template da view

Angular - Componentes

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-root',
  template: `Hello`,
  styles: `:host { color: blue; }`,
  standalone: true,
})
export class AppComponent {}
```

Angular - Componentes

```
import {Component} from '@angular/core';
@Component({
  standalone: true,
  selector: 'todo-list-item',
  templateUrl: './todo-list-item.component.html',
  styleUrls: ['./todo-list-item.component.css'],
})
export class TodoListIem {}
```

Angular - Componentes

- Componentes possuem estado
 - Estado é representado por propriedades de valor na classe que implementa o componente

```
import {Component} from '@angular/core';
@Component({ ... })
export class TodoListItem {
  taskTitle = '';
  isComplete = false;
  completeTask() {
    this.isComplete = true;
  }
  updateTitle(newTitle: string) {
    this.taskTitle = newTitle;
  }
}
```

Angular - Componentes

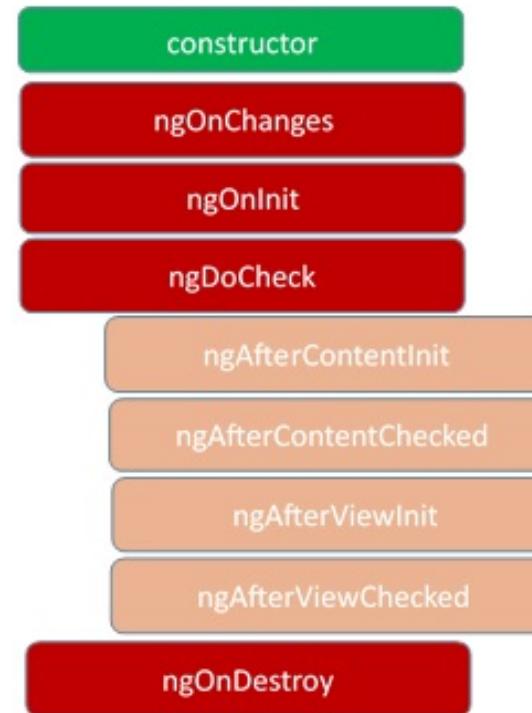
- Dois mecanismos para importar componentes:
 - Componentes *standalone* (versões “atuais”)
 - Componentes definidos em módulos *NgModule* (versões “antigas”)
- Componentes *standalone*:
 - Possuem a propriedade *standalone* com valor *true*
 - Esse tipo de componente pode importar diretamente outros componentes através da propriedade *imports*

Angular - Componentes

```
import {Component} from '@angular/core';
import {TodoListIteм} from './todo-list-item.component.ts';
@Component({
  standalone: true,
  imports: [TodoListIteм],
  template: `
    <ul>
      <todo-list-item></todo-list-item>
    </ul>
  `,
})
export class TodoList {}
```

Angular - Componentes

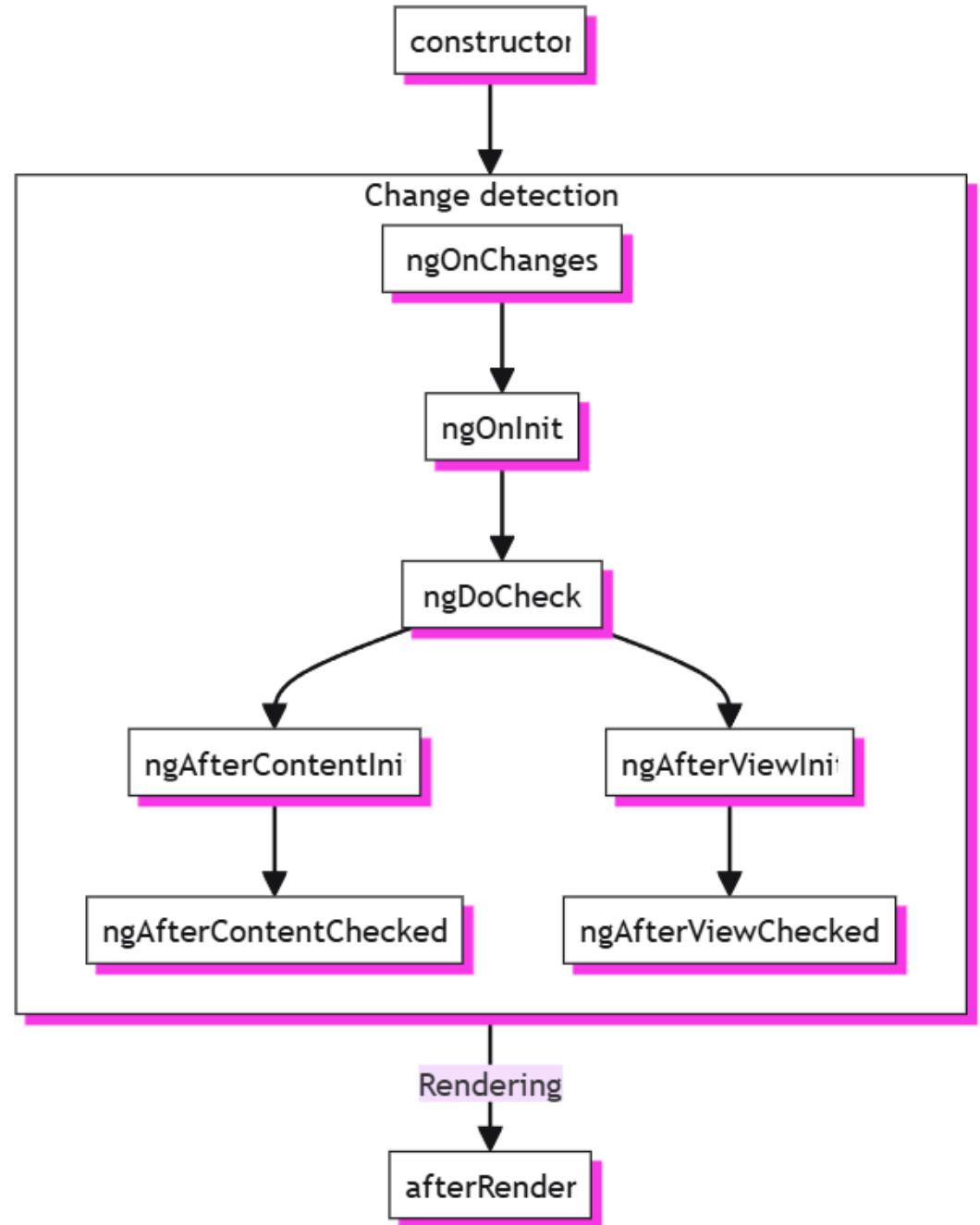
- Para customizar o ciclo de vida de um componente:
 - Classe deve implementar métodos específicos correspondentes a uma determinada fase do ciclo de vida (são métodos chamados automaticamente pelo Angular após o construtor)
 - TypeScript fornece interfaces associadas a cada método para garantir tipagem
- Exemplos:
 - método `ngOnInit()`
 - método `ngOnDestroy()`
- Documentação:
 - <https://angular.dev/guide/components/lifecycle>



Angular - Componentes

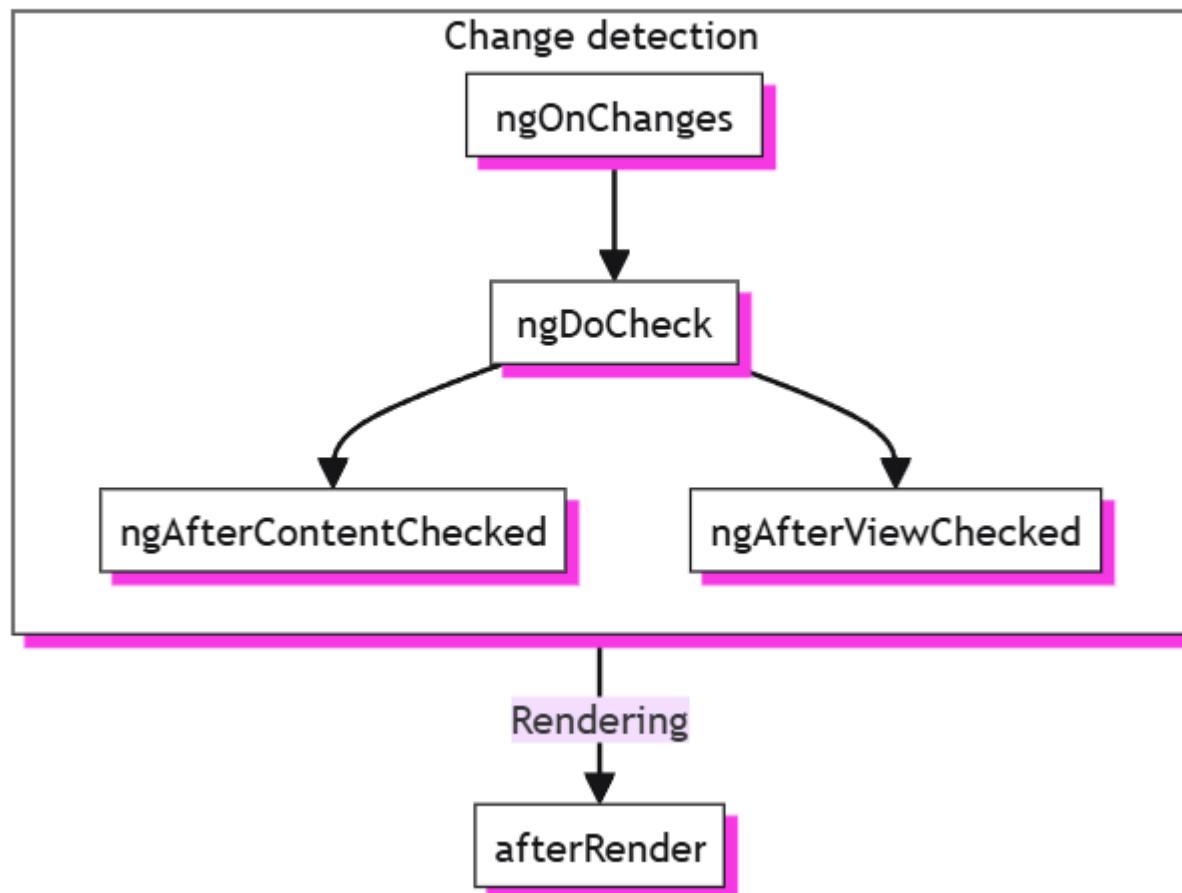
```
@Component({
  /* ... */
})
export class UserProfile implements OnInit {
  ngOnInit() {
    /* ... */
  }
}
```

Angular - Componentes



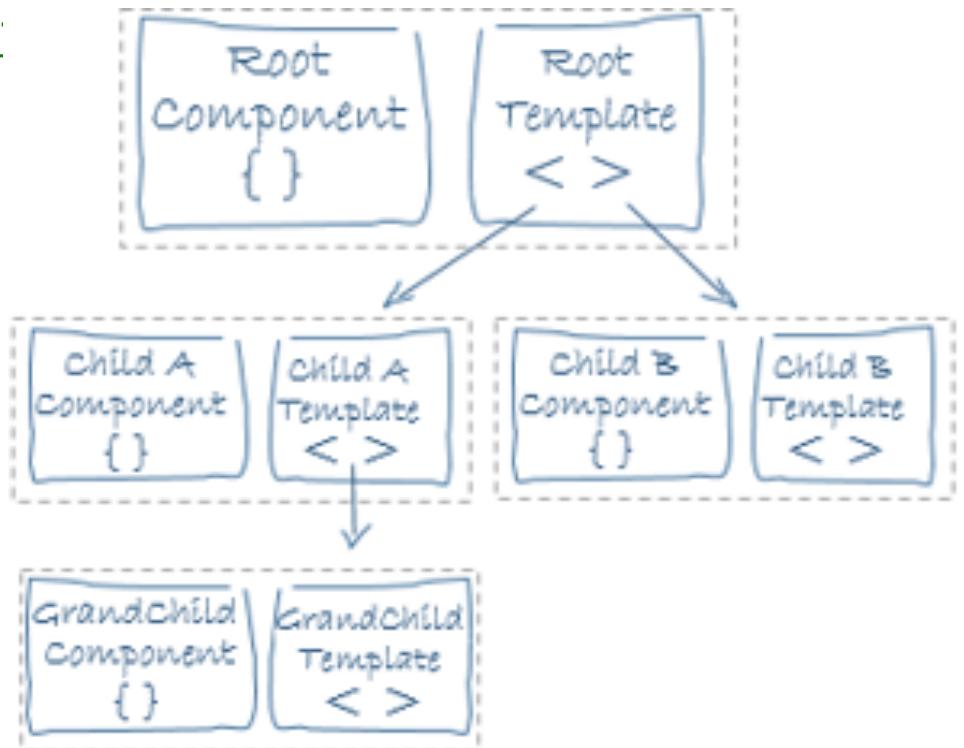
Angular - Componentes

Subsequent updates



Angular - Views

- Views são definidas através de templates do Angular associados ao componentes
- Templates são código HTML enriquecidos com diretivas, componentes e pipes do Angular
 - Documentação: <https://angular.dev/guide/templates>



Angular – Interpolation

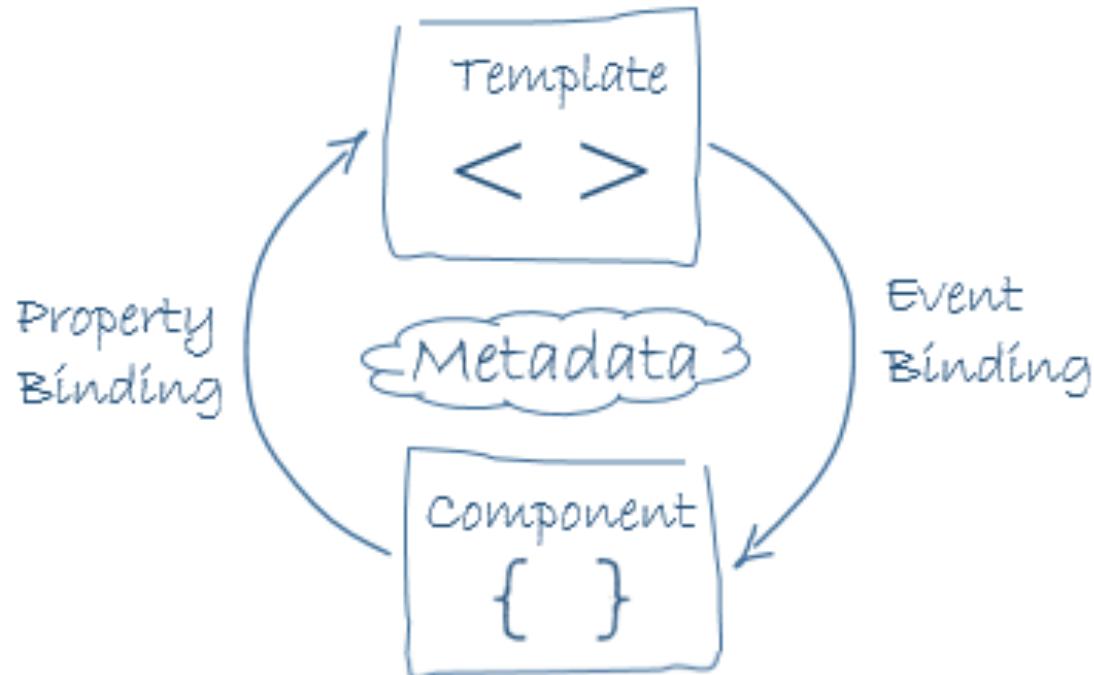
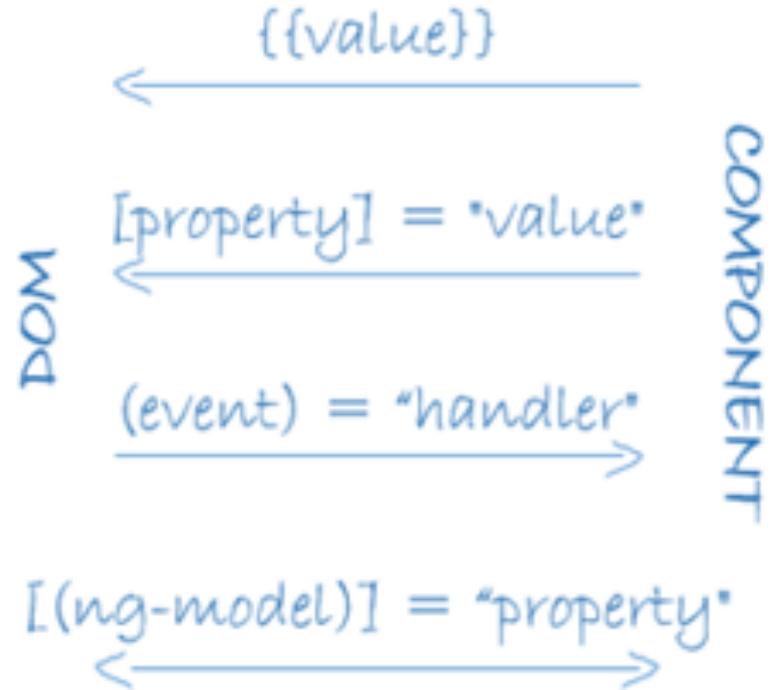
- Interpolação é o mecanismo que permite embutir e avaliar expressões diretamente no template de uma view
 - Por padrão é representada pelos delimitadores {{ }}
- Documentação: <https://angular.dev/guide/templates/interpolation>

```
import {Component} from '@angular/core';
@Component({
  selector: 'todo-list-item',
  template: `
    <p>Title: {{ taskTitle }}</p>
  `,
})
export class TodoListItem {
  taskTitle = 'Read cup of coffee';
}
```

Angular – Binding

- Mecanismo para coordenar comunicação entre o template e o componente
 - É o mecanismo que permite sincronizar a *view* com o *model*
 - Angular suporta vinculação de mão dupla (*two-way data binding*) e também de mão única (*one-way data binding*)
- Documentação: <https://angular.dev/guide/templates/binding>
- Diversos tipos de bindings:
 - Interpolação
 - Propriedade <https://angular.dev/guide/templates/property-binding>
 - Atributo <https://angular.dev/guide/templates/attribute-binding>
 - Classes de estilo <https://angular.dev/guide/templates/class-binding>
 - Evento <https://angular.dev/guide/templates/event-binding>
 - Two-way <https://angular.dev/guide/templates/two-way-binding>

Angular – Binding



```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
<input [(ngModel)]="hero.name">
```

Angular – Binding (propriedade)

```
import {Component} from '@angular/core';
@Component({
  selector: 'sign-up-form',
  template: `
    <button type="submit" [disabled]="formIsValid">Submit</button>
  `,
})
export class SignUpForm {
  formIsValid = true;
}
```

Angular – Binding (atributo)

```
import {Component} from '@angular/core';
@Component({
  standalone: true,
  template: `
    <button [attr.data-test-id]="testId">Primary CTA</button>
  `,
})
export class AppBanner {
  testId = 'main-cta';
}
```

Angular – Binding (classe e estilos)

```
import {Component} from '@angular/core';
@Component({
  standalone: true,
  selector: 'app-nav-bar',
  template: `<nav [style]="navStyle">
    <a [style.text-decoration]="activeLinkStyle">Home Page</a>
    <a [style.text-decoration]="linkStyle">Login</a>
  </nav>`,
})
export class NavBarComponent {
  navStyle = 'font-size: 1.2rem; color: cornflowerblue;';
  linkStyle = 'underline';
  activeLinkStyle = 'overline';
}
```

Angular – Binding (evento)

```
import {Component} from '@angular/core';
@Component({
  standalone: true,
  selector: 'text-transformer',
  template: `
    <p>{{ announcement }}</p>
    <button (click)="transformText()">Abracadabra!</button>
  `,
})
export class TextTransformer {
  announcement = 'Hello again Angular!';
  transformText() {
    this.announcement = this.announcement.toUpperCase();
  }
}
```

Angular – Binding (two-way)

```
import {Component, Input, Output, EventEmitter} from '@angular/core';
@Component({
  standalone: true,
  selector: 'app-sizer',
  templateUrl: './sizer.component.html',
  styleUrls: ['./sizer.component.css'],
})
export class SizerComponent {
  @Input() size!: number | string;
  @Output() sizeChange = new EventEmitter<number>();
  dec() {
    this.resize(-1);
  }
  inc() {
    this.resize(+1);
  }
  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

Angular – Binding (two-way)

```
import {Component} from '@angular/core';
import {SizerComponent} from './sizer/sizer.component';
import {FormsModule} from '@angular/forms';

@Component({
  standalone: true,
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  imports: [FormsModule, SizerComponent],
})
export class AppComponent {
  fontSizePx = 16;
}
```

Angular – Binding (two-way)

```
<h1 id="two-way">Two-way Binding</h1>
<div id="two-way-1">
  <app-sizer [(size)]="fontSizePx"></app-sizer>
  <div [style.fontSize.px]="fontSizePx">Resizable Text</div>
  <label for="font-size">FontSize (px):
    <input id="font-size" [(ngModel)]="fontSizePx">
  </label>
</div>
```

Angular – Fluxo de controle

- Templates para views no Angular suportam blocos de fluxo de controle:
 - @if, @else
 - @switch, @case, @default
 - @for, @empty

Angular – Fluxo de controle

```
@if (a > b) {  
  {{a}} is greater than  
  {{b}}  
} @else if (b > a) {  
  {{a}} is less than  
  {{b}}  
} @else {  
  {{a}} is equal to  
  {{b}}  
}
```

```
@switch (condition) {  
  @case (caseA) {  
    Case A.  
  }  
  @case (caseB) {  
    Case B.  
  }  
  @default {  
    Default case.  
  }  
}
```

```
@for (item of items; track item.id; let idx = $index) {  
  Item #{{ idx }}: {{ item.name }}  
}
```

```
@for (item of items; track item.name) {  
  <li> {{ item.name }}</li>  
} @empty {  
  <li> There are no items.</li>  
}
```

Angular - Pipes

- Pipes são funções que realizam transformações e são úteis na formatação de valores de uma expressão em binding
- Aplicados através da sintaxe **expressão | pipe** na interpolação no template
- Conjunto de pipes pré-definidos
 - date, currency, number, etc
 - <https://angular.dev/api/common#pipes>
- Pipes são classes decoradas com `@Pipe`
- Documentação: <https://angular.dev/guide/pipes>

Angular - Pipes

```
import { Component } from '@angular/core';
import { DatePipe } from '@angular/common';
@Component({
  standalone: true,
  templateUrl: './app.component.html',
  imports: [DatePipe],
})
export class AppComponent {
  today = new Date();
}
```

Angular - Pipes

```
<!-- Default format: output 'Jun 15, 2015'-->
<p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->
<p>The time is {{today | date:'shortTime'}}</p>
```

Angular - Diretivas

- Diretivas indicam ao Angular para anexar comportamento ou transformar o elemento associado ao DOM
 - Componentes são um tipo de diretiva especializada
- Diretivas são de dois tipos (além de componentes):
 - Diretivas estruturais – manipulam a estrutura hierárquica do DOM, são aplicadas através de um *nomediretiva no template
 - Diretivas de atributos – manipulam aparência e comportamento de um elemento, componente ou outra diretiva
- Diretivas pré-definidas tipicamente prefixados por `ng`
 - Exemplos: `ngFor`, `ngIf`, `ngSwitch`, `ngModel`, `ngStyles`, `ngClass`, etc
- Diretivas são classes decoradas com `@Directive`
- Documentação: <https://angular.dev/guide/directives>

Angular - Services

- Serviços são o mecanismo modular para prover funcionalidade independente de uma view
 - Componentes consomem serviços via injeção de dependências
 - Utiliza-se serviços para desacoplar componentes de responsabilidades de negócio tais como acesso a serviços web, logging, lógica e validação, etc.
- Documentação: <https://angular.dev/guide/di/creating-injectable-service>

Angular - Services

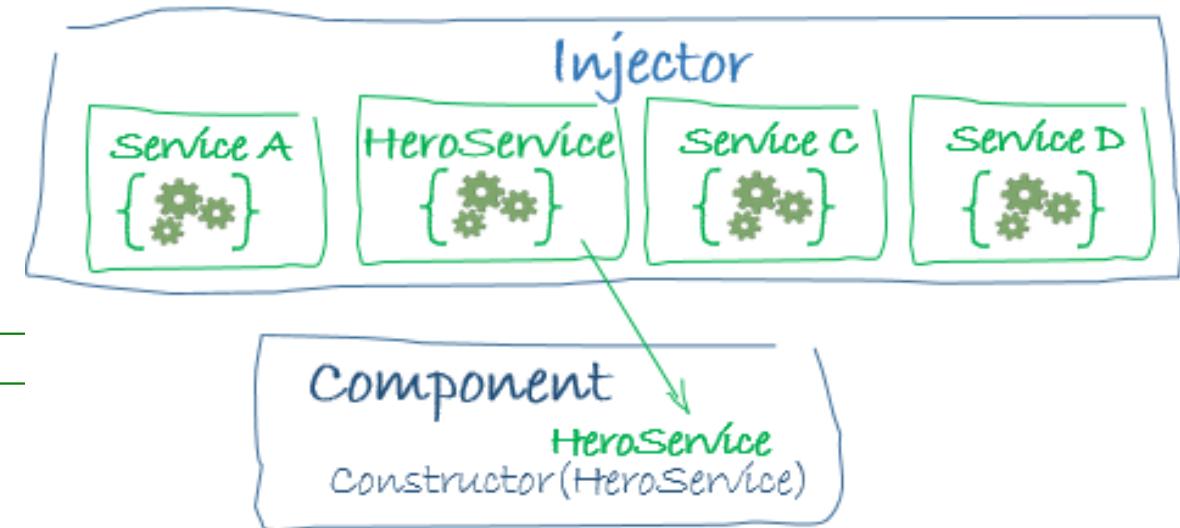
- Para criar um novo serviço:
 - Classe decorada com `@Injectable()` especificando as propriedades do módulo
 - Via Angular CLI usar `ng generate service nomeServiço`
- Para permitir o uso de um serviço via injeção de dependências:
 - Registrar um *provider* (um *provider* é algo capaz de criar e disponibilizar algo para o injetor de dependências)
 - Usualmente a própria classe do serviço é o próprio *provider* registrado no módulo raiz
- Para consumir um serviço:
 - Importar a classe do serviço ou módulo que contém o *provider* (caso não seja um módulo raiz)
 - Definir o ponto de injeção no construtor via um parâmetro do tipo do serviço desejado ou via a função `inject()`

Angular - Services

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

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

```
export class HeroListComponent {  
  heroes: Hero[];  
  
  constructor(private heroService: HeroService) {  
    this.heroes = heroService.getHeroes();  
  }  
}
```



Angular - Services

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

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

```
export class HeroListComponent {
  heroes: Hero[ ];
  heroService = inject(HeroService);
}
```

Angular – Injeção de Dependências

- Injeção de dependências é um padrão de projeto que visa resolver a quebra de dependências entre múltiplos objetos
 - Controla quem instancia os objetos e os passa (“injeta”) prontos para o uso para quem precisa deles
- Angular utiliza a seguinte estrutura:
 - **Injetor** é o objeto que gerencia o processo de injeção de dependências; são criados automaticamente pelo Angular
 - Injetor gerencia um **contêiner de instâncias** de dependências em memória
 - Um **provedor** é um objeto que informa ao injetor como obter/criar dependências
 - Uma classe solicita ao injetor as dependências via metadados e os recebe via construtor
- Documentação: <https://angular.dev/guide/di>

Angular – Injeção de Dependências

- Diferente formas de registrar um provider:
 - No nível da aplicação raiz via decoração `@Injectable()` e na propriedade `providedIn` do próprio objeto alvo da injeção (modo preferencial)
 - No nível do componente via decoração `@Component()` e na propriedade `providers` de um componente
 - No nível da aplicação raiz via objeto `ApplicationConfig`
 - Para aplicações baseadas em módulos via decoração `@NgModule()` e na propriedade `providers` de um módulo

Angular – Injeção de Dependências

- Características importantes:
 - Objetos sob controle do sistema de DI seguem o padrão *Singleton* (uma única instância em memória é compartilhada) dentro do escopo de cada injetor
 - Existe somente um único injetor raiz na aplicação
 - IMPORTANTE! Registrar um objeto via `@Injectable` com `providedIn=root` ou no módulo raiz `AppModule` via `@NgModule` com providers resulta em um único objeto compartilhado entre todos que requisitarem esse objeto via injeção de dependência
 - Cada módulo ou componente define uma hierarquia de injetores, cada qual com seu escopo

Angular - Módulos

- Conteiner para diferentes partes da aplicação
 - Components, services, directives, pipes, ...
 - São chamados de *NgModules*
 - Define um um “contexto de compilação” para componentes
- Criam-se módulos para:
 - Organização coesa de blocos de funcionalidades do sistema
 - Agrupamento de componentes reutilizáveis
 - Bibliotecas (ex.: FormsModule, HttpClientModule, RouterModule, etc.)
- Cuidado!
 - Sistema de módulos do Angular é complementar ao sistema de módulos do Ecmascript
 - Documentação: <https://angular.dev/guide/ngmodules>

Angular - Módulos

- Para criar um novo módulo:
 - Classe decorada com `@NgModule()` especificando as propriedades do módulo
 - Via Angular CLI usar `ng generate module nomeModulo`
- Principais propriedades:
 - `declarations` – estruturas (componentes, diretivas, pipes) que pertencem ao módulo
 - `exports` – subconjunto de `declarations` visíveis para quem importar o módulo
 - `imports` – módulos importados que serão utilizados por estruturas do módulo atual
 - `providers` – criadores de serviços definidos no módulo atual que são exportados para a aplicação
 - `bootstrap` – somente o módulo raiz apresenta essa propriedade especificando o componente raiz da aplicação

Angular - Módulos

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```