

# Desenvolvimento Web

Trilha JavaScript com Angular e Node

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



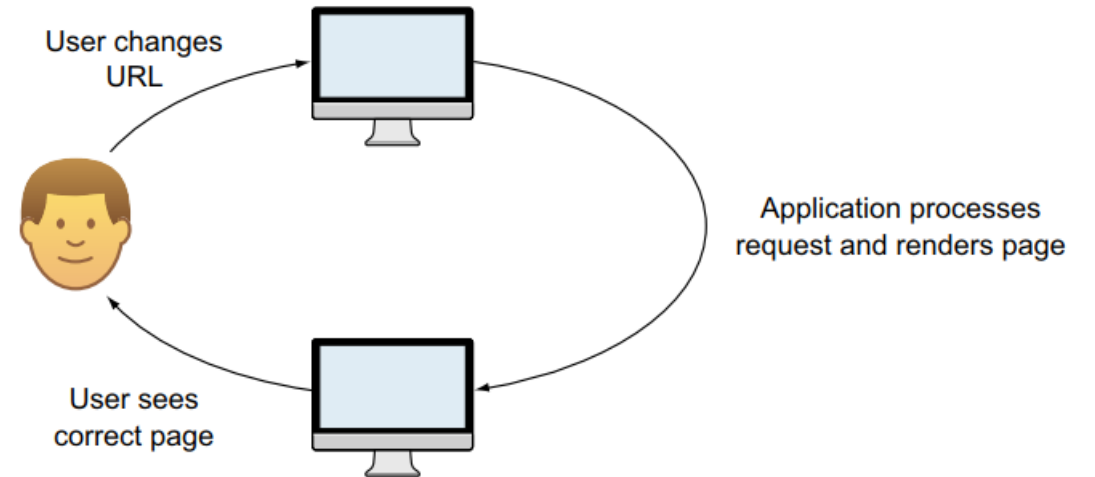
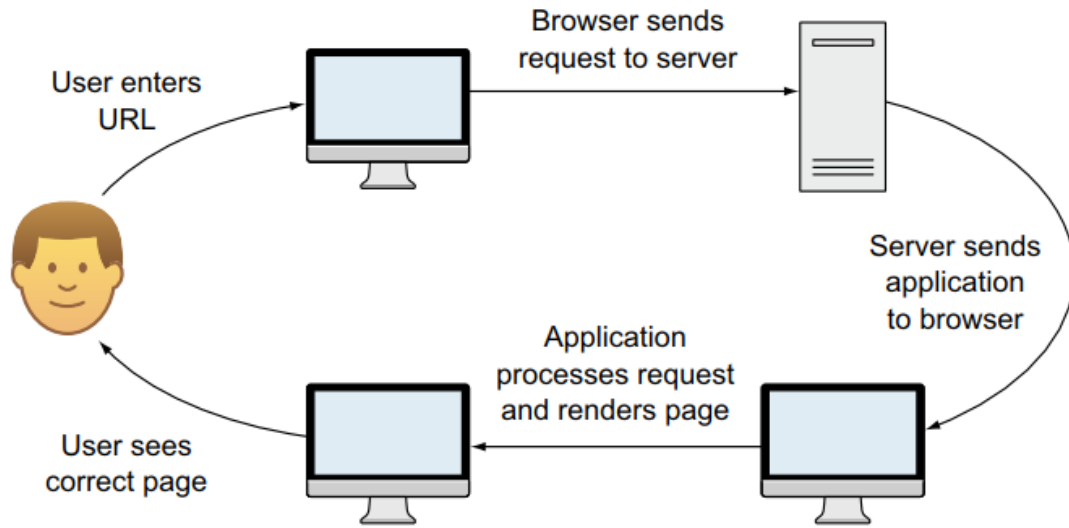
# Single Page Applications



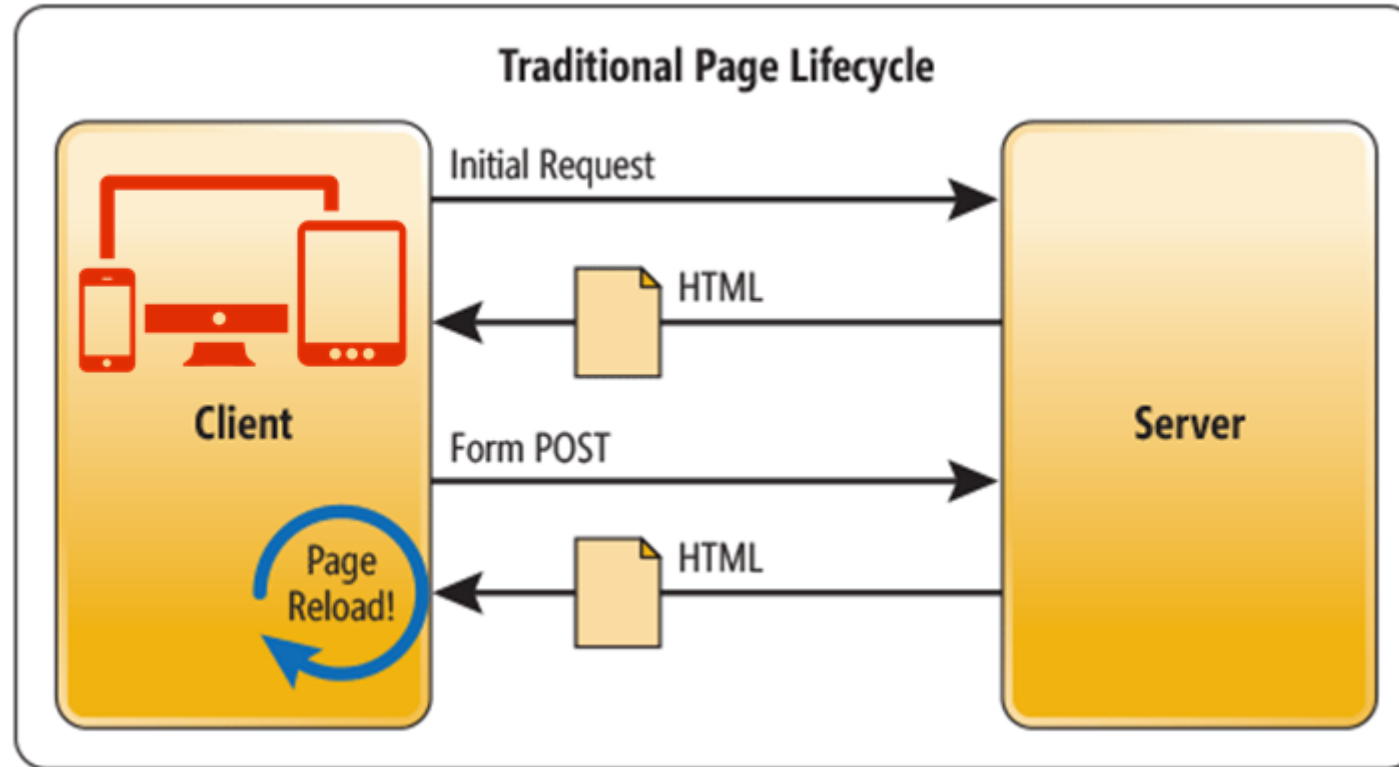
# Single Page Applications

- 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

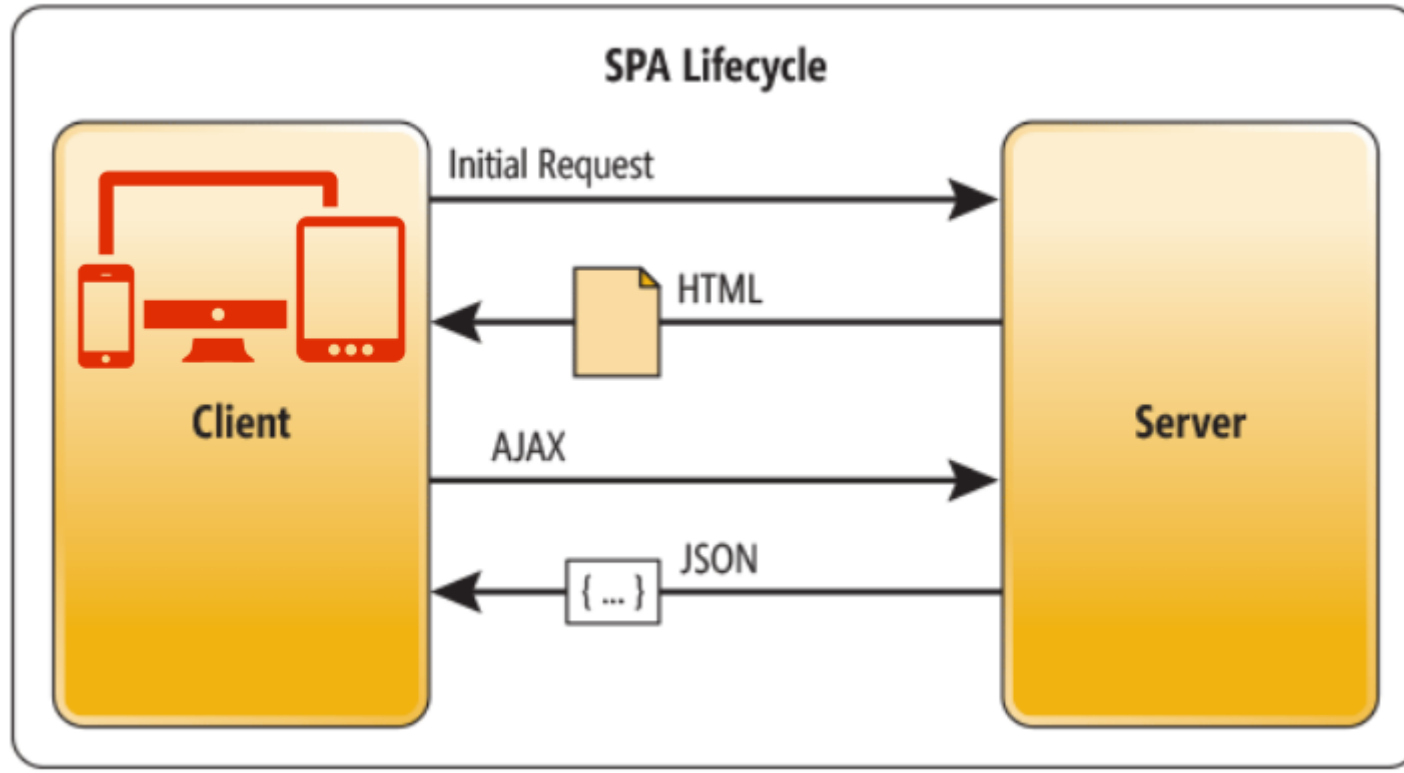
# Single Page Applications



# Single Page Applications



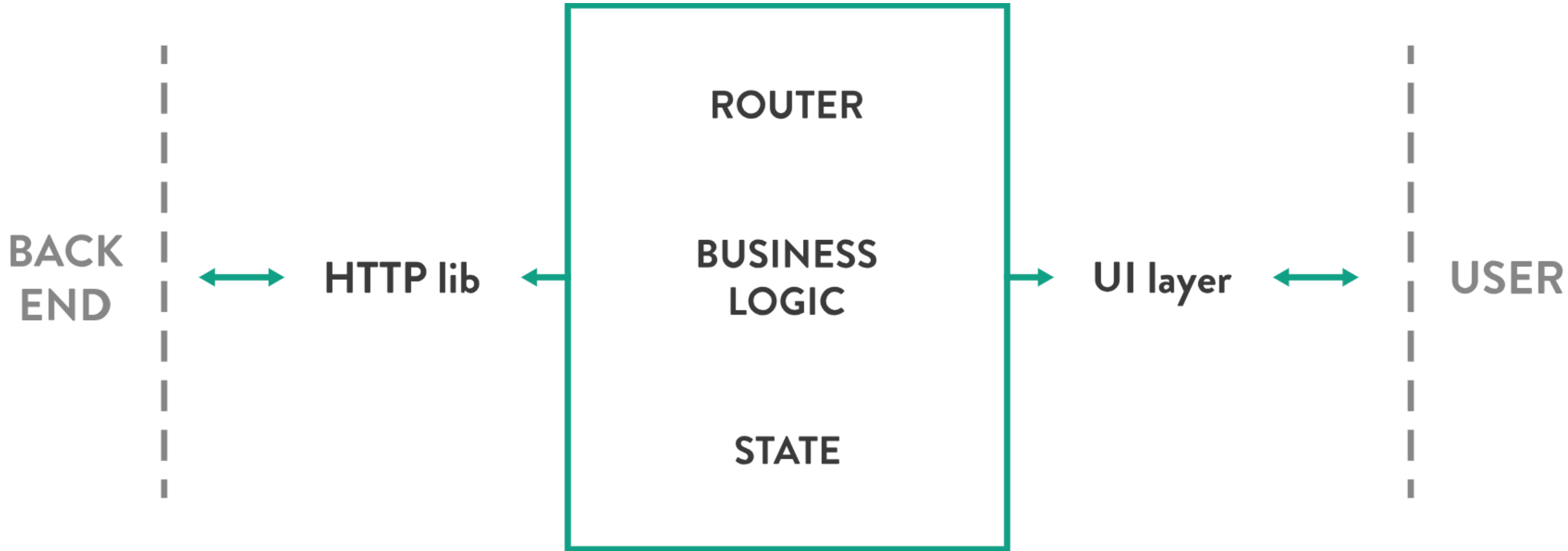
# Single Page Applications



# Single Page Applications

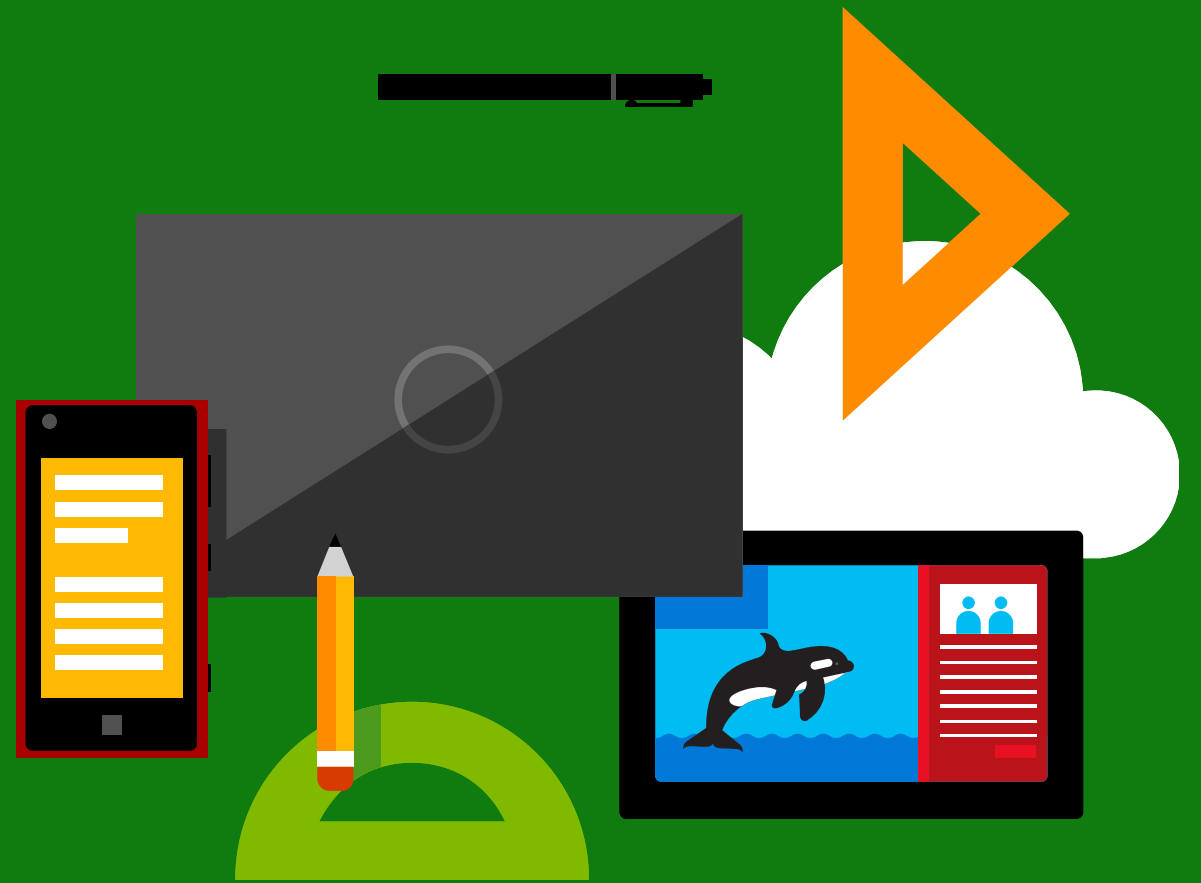
- Recursos usualmente utilizados por frameworks para criação de SPAs:
  - Views baseadas em template
  - Roteamento entre views
  - Gerenciamento de estado
  - Requisições assíncronas para o backend

# Single Page Applications





# Angular



# Angular

- Framework para o lado cliente de aplicações Web
  - Também suporta o desenvolvimento de aplicações desktop e mobile
- Código aberto

<https://angular.io/>



# Angular

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

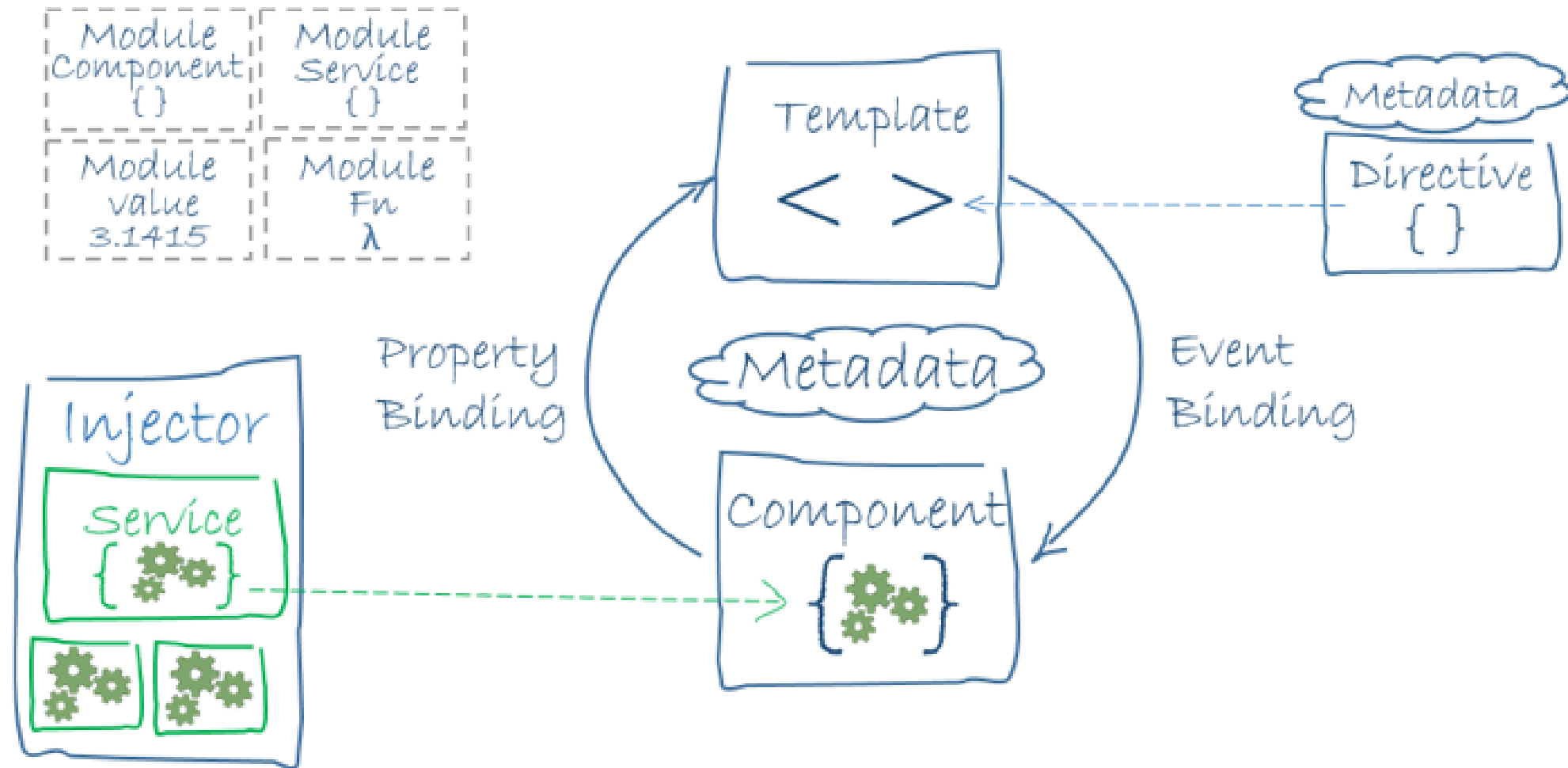
# Angular

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

# 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.io/cli>
  - <https://cli.angular.io/>
- Instalação:
  - `npm install -g @angular/cli`

# Angular – Blocos de Construção



# Angular – Principais Conceitos

## MODULES

Estrutura as diferentes partes da aplicação. Módulo para cada elemento da solução.

## COMPONENTS

Classe que representa a junção da lógica para manipulação das views e dos templates que representam a view.

## DIRECTIVES

Extensões ao HTML, utilizadas para manipular o DOM.

## SERVICES

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>



# Angular - Módulos

- Container para diferentes partes da aplicação
  - Components, services, directives, pipes, ...
  - São chamados de *NgModules*
  - Define um um "contexto de compilação" para componentes
- Cria-se módulos para:
  - Definição do módulo raiz da aplicação Angular (nome convencional de AppModule no arquivo app.module.ts)
  - Organização coesa de blocos de funcionalidades do sistema
  - Agrupamento de componentes reutilizáveis
- Biblioteca do Angular é composta de NgModules
  - <https://angular.io/guide/frequent-ngmodules>
- Cuidado!
  - Sistema de módulos do Angular é complementar ao sistema de módulos do EcmaScript
- Documentação:
  - <https://angular.io/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 { }
```

# 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.io/guide/architecture-components>

# Angular - Componentes

- Para criar um novo componente:
  - Classe decorada com `@Component()` especificando as propriedades do módulo
  - 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
  - `templateUrl` – define o arquivo de template a ser utilizado para a construção da view
  - `styleUrls` – definem os arquivos de estilos CSS locais a serem utilizados no template da view
  - `providers` – informações dos provedores de serviços que devem ser utilizados durante a injeção de dependências

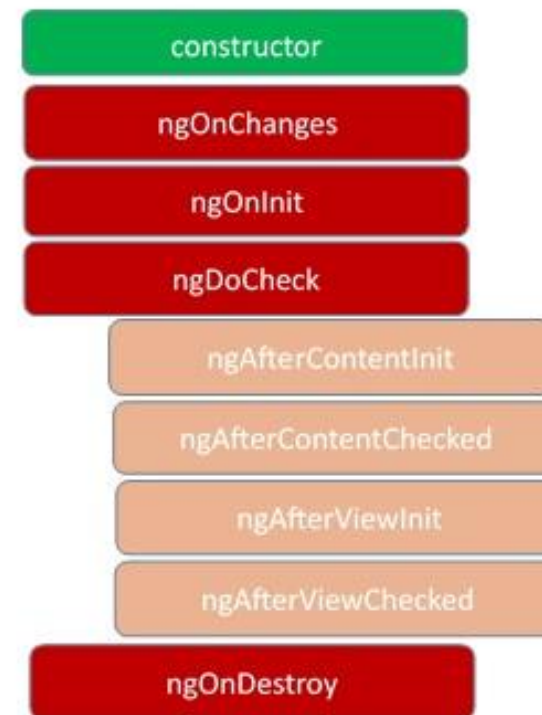
# Angular - Componentes

```
import { Component, OnInit }    from '@angular/core';
import { Hero }                 from './hero';
import { HeroService }          from './hero.service';

@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  constructor(private service: HeroService) { }
  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }
  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

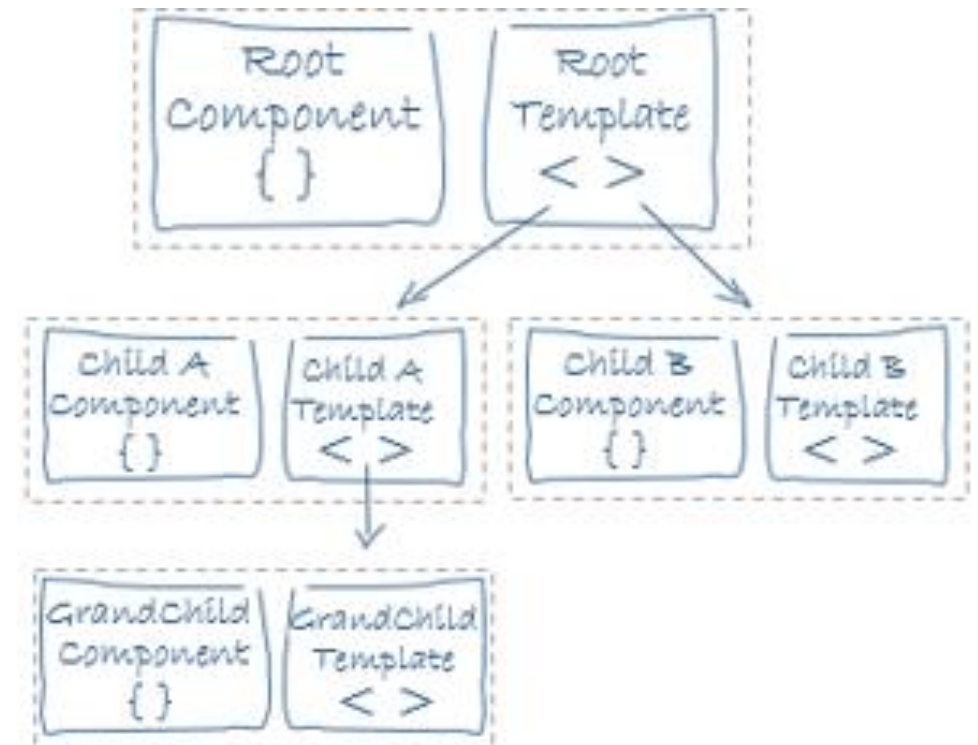
# Angular - Componentes

- Para customizar o ciclo de vida de um componente:
  - Classe deve implementar a interface adequada
  - Cada interface contem um único método correspondente a uma determinada fase do ciclo de vida (são métodos chamados automaticamente pelo Angular após o construtor)
  - Exemplos:
    - Interface OnInit e método ngOnInit()
    - Interface OnDestroy e método ngOnDestroy()
- Documentação:
  - <https://angular.io/guide/lifecycle-hooks>



# 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
  - Sintaxe: <https://angular.io/guide/template-syntax>
- Formam uma estrutura hierárquica



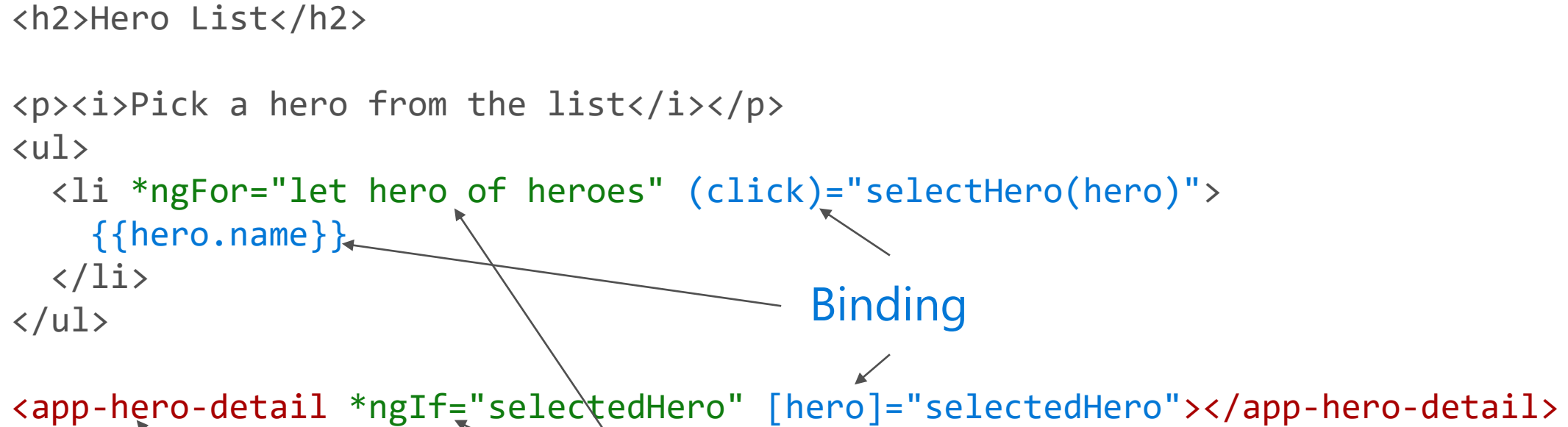


# Angular - Views

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>
```



Binding

Component

Directive

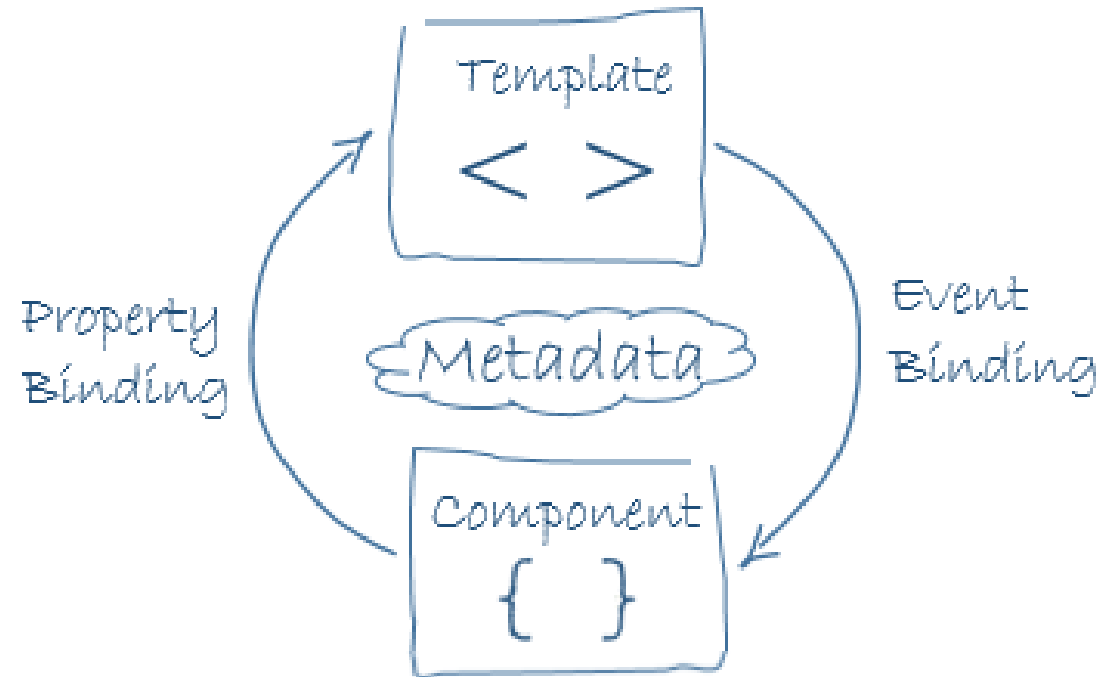
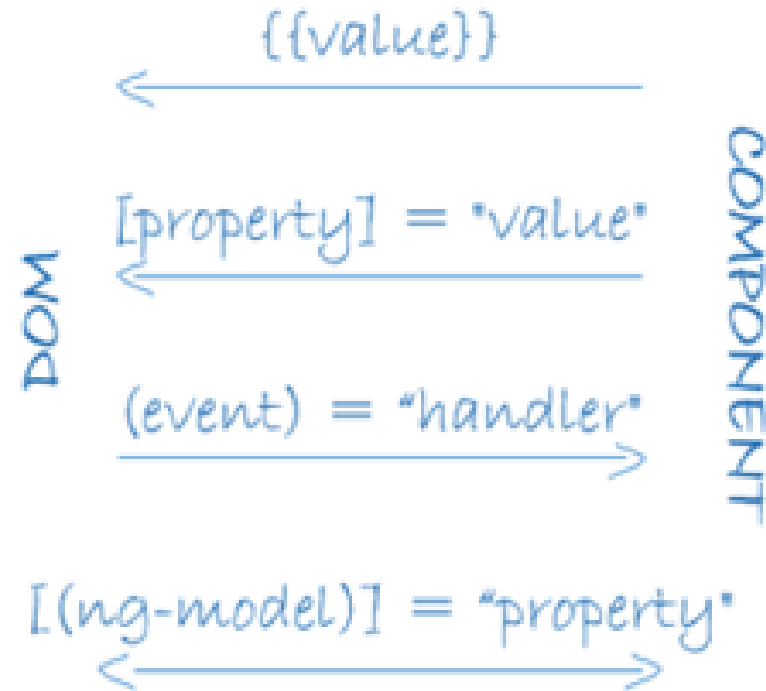
# Angular – Data Binding

- Mecanismo para coordenar comunicação entre o template e o componente
  - Angular suporta vinculação de mão dupla (*two-way data binding*) e também de mão única (*one-way data binding*)
- Dois tipos de *bindings*:
  - Eventos – permite associar tratadores a eventos do DOM
  - Propriedades – permite obter valores que são computados via código e associá-los a propriedades de elementos do HTML
- Documentação:
  - Interpolação <https://angular.io/guide/displaying-data#interpolation>
  - Propriedade <https://angular.io/guide/template-syntax#property-binding>
  - Evento <https://angular.io/guide/template-syntax#event-binding---event->
  - Two-way <https://angular.io/guide/template-syntax#two-way-binding--->

# Angular – Data Binding

- CUIDADO!
  - A vinculação de dados permite definir valores para propriedades de elementos DOM, componentes e diretivas
  - No template, não estamos associando valores a atributos do HTML
  - Ver <https://angular.io/guide/template-syntax#html-attribute-vs-dom-property>

# Angular – Data Binding



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

# Angular - Pipes

- Pipes realizam transformações e são uteis na formatação de valores de uma expressão em *data binding*
- Conjunto de pipes pré-definidos
  - date, currency, number, etc
  - <https://angular.io/api?type=pipe>
- Pipes são classes decoradas com **@Pipe**
- Aplicados através da sintaxe **expressão | pipe** na interpolação no template
- Documentação:
  - <https://angular.io/guide/pipes>

# 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
  - 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.io/guide/architecture-components#directives>
  - <https://angular.io/guide/attribute-directives>
  - <https://angular.io/guide/structural-directives>

# Angular - Diretiva Estrutural

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

Use a sintaxe  
abreviada!

```
<ng-template [ngIf]="hero">  
  <div class="name">{{hero.name}}</div>  
</ng-template>
```



# Angular - Diretiva de Atributo

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'" >  
  This div is x-large or smaller.  
</div>
```

```
<input [(ngModel)]="hero.name">
```

# 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 web, logging, lógica e validação, etc
- Documentação:
  - <https://angular.io/guide/architecture-services>
  - <https://angular.io/guide/providers>
  - <https://angular.io/guide/singleton-services>
  - <https://angular.io/guide/dependency-injection>

# Angular - Services

- Para criar um novo serviço:
  - Classe decorada com `@Injectable()` especificando as propriedades do módulo
  - Via Angular CLI usar `ng generate service nomeServico`
- 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 o módulo que contem o *provider* (caso não seja um módulo raiz)
  - Usualmente definir o ponto de injeção no construtor via um parâmetro do tipo do serviço desejado

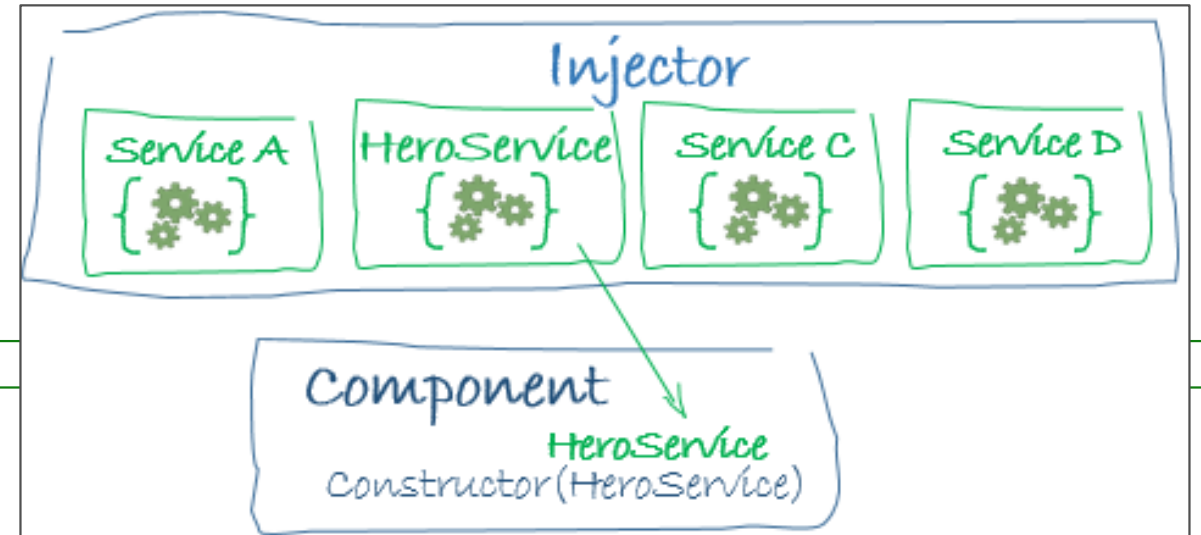
# Angular - Services

- Exemplo:

```
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 – 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.io/guide/dependency-injection>

# Angular – Injeção de Dependências

- Diferente formas de registrar um *provider*:
  - Na decoração `@Injectable()` e na propriedade `providedIn` do próprio objeto alvo da injeção
  - Na decoração `@NgModule()` e na propriedade `providers` de um módulo
  - Na decoração `@Component()` e na propriedade `providers` de um componente

# 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 – Injeção de Dependências

- Exemplos:

```
import { Injectable } from '@angular/core';
import { UserModule } from '../user.module';

@Injectable({
  providedIn: UserModule,
})
export class UserService {
}
```



# Angular – Injeção de Dependências

- Exemplos:

```
import { NgModule } from '@angular/core';  
import { UserService } from '../user.service';
```

```
@NgModule({  
  providers: [UserService],  
})  
export class UserModule {  
}
```

```
@Component({  
  selector: 'app-hero-list',  
  templateUrl: './hero-list.component.html',  
  providers: [ HeroService ]  
})
```