Injeção de Dependência e Testes

Conceitos de Injeção de Dependência

- Cliente
  - Objeto que precisa utilizar outros objetos
- Dependência
  - Objetos que são utilizados por outro objeto
- Interface
  - Contrato que uma dependência precisa atender para ser utilizado pelo Cliente
- Injector
  - O responsável a passar as dependências para o cliente
- Inversão de controle
- Evita acoplamento entre objetos
- Parte do conceito de SOLID
  - D = Dependency Injection Principle
  - *“Dependa de abstrações, não de implementações concretas”*

Sistemas de Injeção de Dependência

- Sistema que auxilia na injeção das dependências corretas de cada cliente dada uma determinada configuração
- Responsável por instanciar as dependências
- Podem suportar diferentes tipos de resolver as dependências e diferentes tipos de injeção.

Injeção de Dependência no Angular

- Duas hierarquias de Injectors
  - **ModuleInjector**
  - **ElementInjector**
- Para buscar uma dependência, o Angular navega primeiro na hierarquia de **ElementInjectors** e caso não encontre a dependência, começa a navegar na hierarquia de **ModuleInjectors**
- Suporta injeção via construtor somente
- Dependências são representadas por **InjectionTokens**
- @Directive / @Component / @Injectable registram as classes no sistema de DI do Angular como **InjectionTokens**

Principais APIs de DI do Angular

- @Injectable, @Component, @Directive
- @Inject
- InjectionToken<T>
- Injector
- providers

Modos de registrar um InjectionToken como provider

- useValue - Usa uma constante
- useClass - Uma classe registrada no DI que será instânciada
- useFactory - Uma função que retornará a dependência
- useExisting - Referencia um token existente

Modificadores

- deps - As dependências necessárias para a função de useFactory
- multi - Indica que o token deve ser retornado em forma de array, de forma que multiplos providers podem contribuir com o registro de novos tokens (ex: Interceptadores HTTP)

Modificadores de resolução de dependência

- Categorias:
  - O que fazer quando o Angular não encontra a dependência? **@Optional**
  - Onde devo começar a buscar a dependência? **@SkipSelf**
  - Onde devo parar de procurar a dependência? **@Host** / **@Self**

@Optional: Por padrão, se o Angular não consegue resolver a dependência, ele vai acusar um erro. Caso a dependência não seja obrigatória, declare ela no construtor como opcional (?) e decore ela com @Optional()

@SkipSelf: Por padrão o Angular procura a dependência direto no ElementInjector do próprio componente/diretiva. Caso você queira que ele pule direto para o ElementInjector pai, decore a dependência com @SkipSelf()

@Self: Faz com que não se procure a dependência em outros Injectors pais.

@Host: Faz com que a busca pela dependência se restrinja a procurar a dependência somente na view que contém o componente.

Dependências

- Tipos de dependência
  - Transitiva
  - Singleton
- Quando o InjectionToken é registrado nos providers de um módulo, ele é registrado como um **Singleton** e a mesma instância é compartilhada todos os clientes. Ele pode ser sobrescrito por outro provider do mesmo InjectionToken.
- Quando um InjectionToken é registrado por um componente ou directiva, ele é instanciado junto com o mesmo, gerando novas instâncias.

Testes

Tipos de Testes

- Unitário
- Integração
- End-to-End

Testes unitários

- Testa a menor unidade possível, sem envolver outras dependências
- Dependências podem ser “simuladas” (mocks) para atenderem o comportamento adequado para o teste
- Funções puras (que tem somente inputs e outputs bem definidos, sem dependências ou interações externas) são extremamente fáceis de se testar.
- Devido ao ponto acima, tente sempre escrever funções puras para facilitar os testes, principalmente funções que isolam lógica de negócio.
- Testes unitários são altamente performáticos devido a simplicidade.

Testes de integração

- Testam mais de uma unidade ao mesmo tempo, mas com um escopo bem limitado somente para testar a integração entre os componentes.
- Testes de integração tem uma performance mediana devido a necessidade de se instanciar algumas peças para realizar os testes.
- Ex: Classe de componente + template do angular do componente

Testes end-to-end (e2e)

- Testam a aplicação simulando um usuário interagindo com a aplicação.
- Necessitam da aplicação rodando como um todo, por isso tem performance baixa e devem se limitar a testar os caminhos mais críticos da aplicação.

Testes no Angular

- Classes podem ser testadas sem tratamento especial
- Componentes e diretivas onde é precisa simular o template acabam tendo a dependência do Angular em si, então podemos considerar que são testes de integração junto ao framework.
- Ao gerar objetos com a CLI, o Angular automaticamente já cria um arquivo de testes nome.spec.ts contendo um scaffold básico da estrutura de testes.
- Por padrão o Angular utiliza a framework de testes **Jasmine** e o **Karma,** mas é recomendado utilizar o **Jest** devido a melhor performance e comunidade. A API do Jest é muito similar ao Jasmine, então é fácil migrar/aprender.
- Testes end-to-end era utilizado o framework **Protractor**, mas foi descontinuado. Recomenda-se usar o framework **Cypress** para esses testes atualmente.

Testes no Angular

- Para se escrever testes de integração no Angular, ele provê uma biblioteca chamada de **TestBed,** que nos ajuda a configurar um módulo de teste para testarmos os componentes.
- Podemos **mockar** as dependências utilizando os providers do módulo para sobrescrever as dependências com novos valores.