Controller

É um dos **princípios GRASP** e é uma peça fundamental no **Design Orientado a Objetos (OO Design)**.

- Ele atua como um intermediário entre a **interface do utilizador (UI)** e a **lógica de negócio (Modelo de Domínio)**.

1️⃣ O Que É um Controller?

📌 **É uma classe que recebe solicitações da interface do utilizador e decide como processá-las.**

📌 **Segue o princípio "Separation of Concerns" (Separação de Responsabilidades)** → Mantém a UI e lógica de negócio desacopoladas.

2️⃣ Para Que Serve o Controller?

✅ **Separa a UI da lógica de negócio** → A interface apenas **captura os dados** e passa para o Controller.

✅ **Reduz acopolamento** → A interface não precisa conhecer os detalhes internos do domínio.

✅ **Melhora a organização do código** → Mantém a lógica de fluxo dentro de um único ponto de controlo.

✅ **Facilita testes unitários** → Podemos testar a lógica do Controller sem precisar da interface gráfica.

3️⃣ Quando Usar um Controller?

📌 Sempre que tivermos **interações entre um utilizador e o sistema**, devemos usar um Controller para processá-las.

📌 Quando um **caso de uso do sistema envolve múltiplos passos** e **precisamos coordená-los**.

📌 Quando queremos **evitar que a UI tenha acesso direto ao modelo de domínio**, garantindo encapsulamento.

💡 **Exemplo de Aplicação:**

- Processar um **login de utilizador**.
- Gerir **criação e edição de pedidos**.
- Validar um **pagamento antes de finalizar uma compra**.

4️⃣ Como Funciona um Controller?

📌 Fluxo Básico de um Controller

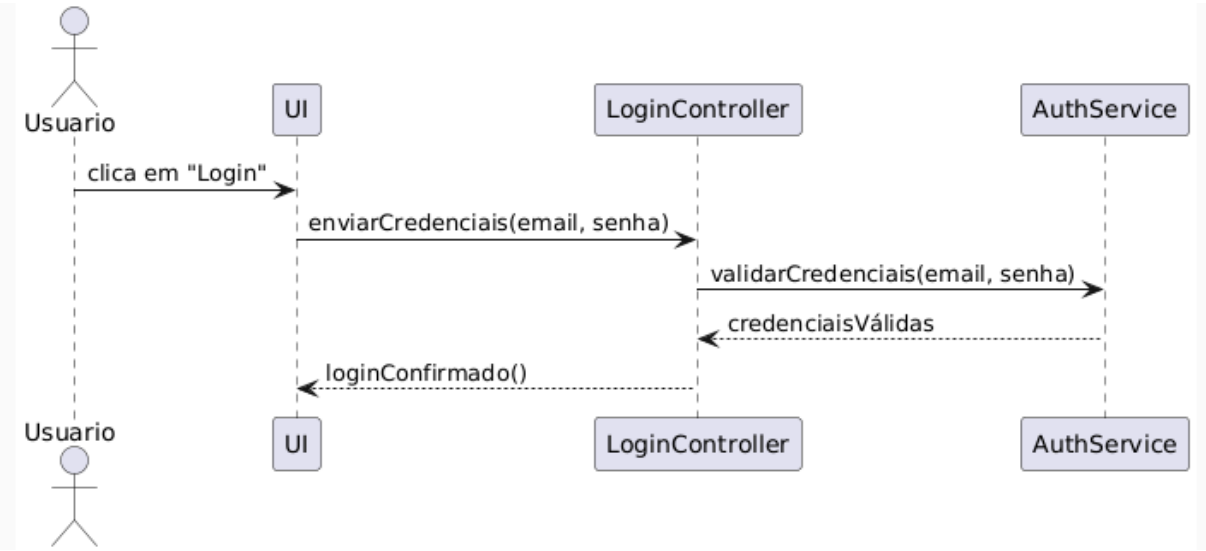
1️⃣ A UI chama um **método do Controller**.

2️⃣ O Controller **valida os dados recebidos**.

3️⃣ O Controller chama o **Modelo de Domínio** para processar a lógica de negócio.

4️⃣ O Controller retorna **uma resposta** à UI.

💡 **Exemplo Visual:**



📌 **O que acontece aqui?**

✅ A **UI não chama diretamente o serviço de autenticação**.

✅ O **Controller intermedia a lógica de negócio**, garantindo encapsulamento.

✅ Se for necessário **alterar a autenticação**, basta modificar `AuthService`, sem impactar a UI.

6️⃣ Boas Práticas ao Criar um Controller

✅ 1. Um Controller Não Deve Ter Lógica de Negócio

O Controller deve apenas **chamar o serviço de domínio**, que contém a lógica real.

✅ 2. Um Controller Deve Ser Focado Num Caso de Uso

✔️ **Correto:** Criar Controllers separados para diferentes ações (`LoginController`, `CarrinhoController`).

✅ 3. Um Controller Deve Ser Simples

📌 **O Controller deve apenas delegar ações, não processá-las diretamente.**

✔️ **Correto:** O Controller chama um **Service ou Repository** para fazer isso.

📌 Responsabilidade do Controller

O **Controller** deve ser responsável por:

1. **Receber requisições** (como HTTP GET, POST, PUT, etc.).
2. **Chamar a camada de serviço** para executar a lógica de negócios.
3. **Capturar exceções** que podem ocorrer durante a execução da lógica de negócios.
4. **Tratar exceções** de forma adequada (ex: retornando um código HTTP apropriado e uma mensagem de erro significativa).
5. **Retornar a resposta** para o cliente (normalmente em formato JSON, XML, etc.).

1️⃣ Como o Controller Retorna Erros para a UI?

📌 O **Controller não deve processar o erro diretamente**. Em vez disso, ele:

✅ **Valida os dados de entrada** (ex.: campos vazios, formato inválido).

✅ **Chama o Service para processar a lógica de negócio**.

✅ **Se ocorrer um erro, captura a exceção e retorna uma resposta apropriada para a UI**.

O fluxo correto é:

1️⃣ **O Service (Domain) lança uma exceção** quando há um erro (ex.: dado inválido, regra de negócio quebrada).

2️⃣ **O Controller captura essa exceção com** `try-catch` e retorna uma resposta apropriada para a UI.

3️⃣ **A UI recebe a resposta e exibe uma mensagem amigável ao utilizador**.

Onde validar se os repositórios são `null`?

1️⃣ **No próprio construtor do controller** (boa prática):

- Garante que o controller recebe sempre dependências válidas.
- Evita erros `NullPointerException` em tempo de execução.

2️⃣ **Na injeção de dependências (Spring, CDI, etc.)**:

- Se estiver a usar **Spring Boot**, a injeção de dependência automática (`@Autowired`) **já garante que os repositórios não serão** `null`, então essa verificação pode ser desnecessária.

❓ Perguntas:

1. **Porquê validar se um repositório passado no construtor não é** `null`**?**

   📌 Para evitar `NullPointerException` e garantir que tudo o que o controller precisa foi realmente passado para ele.

2. **Onde validar essa dependência (**`Controller`** ou** `Service`**)?**

   📌 Geralmente no próprio **construtor do controller**, mas se a lógica de validação for mais complexa, pode ser responsabilidade da **Service Layer**.

3. **Se estivermos a usar Spring Boot, ainda precisamos validar** `null` **no construtor?**

   📌 **Não necessariamente**, pois o Spring já gerencia as dependências, lançando erros caso algo esteja errado.

🎯 Conclusão

✅ **O Controller serve para intermediar interações entre a UI e o domínio.**

✅ **Ajuda a organizar o código, separando responsabilidades e reduzindo acoplamento.**

✅ **Segue boas práticas, mantendo a lógica de negócio fora da UI.**

✅ Se houver um erro, o Controller retorna uma resposta com a mensagem apropriada.