Roteiro de Atividade Prática

Nome: Ezequiel, Gabriel, Vitor, kevyn

Turma: .

**Título da atividade: Design patterns para back-end**

**1. Introdução aos conceitos**

**Objetivo:** apresentar os conceitos básicos de arquitetura monolítica e microsserviços, destacando suas características, prós e contras.

* **Arquitetura monolítica:**
  + **Definição:** uma arquitetura em que todos os componentes de uma aplicação são construídos em um único bloco de código.
  + **Características:** simplicidade inicial, facilidade de desenvolvimento e *deploy*, mas com problemas de escalabilidade e manutenção.
  + **Exemplo conceitual:** um sistema de *e-commerce* em que todas as funcionalidades (usuários, produtos, pedidos) estão em um único projeto.
* **Arquitetura de microsserviços:**
  + **Definição:** uma arquitetura que divide a aplicação em serviços menores e independentes, cada um responsável por uma funcionalidade específica.
  + **Características:** escalabilidade, resiliência, facilidade de manutenção, mas com maior complexidade de gerenciamento.
  + **Exemplo conceitual:** um sistema de *e-commerce* em que cada funcionalidade (usuários, produtos, pedidos) é um serviço separado, podendo ser desenvolvido e implantado de forma independente.

**2. Explorando Design Patterns**

**Objetivo:** introduzir os participantes aos principais padrões de design que podem ser aplicados em arquiteturas monolíticas e de microsserviços.

* **Padrões para arquitetura monolítica:**
  + **Singleton:** garantir que uma classe tenha apenas uma instância e fornecer um ponto global de acesso a ela. Usado para gerenciar recursos compartilhados em uma aplicação monolítica.
  + **Factory Method:** um padrão para criar objetos sem especificar a classe exata do objeto que será criado. Útil para desacoplar a lógica de criação de objetos do código de negócio.
* **Padrões para arquitetura de microsserviços:**
  + **Service Registry:** um padrão para gerenciar os serviços disponíveis e suas localizações, facilitando a descoberta e comunicação entre microsserviços.
  + **Circuit Breaker:** um padrão que ajuda a garantir a resiliência da aplicação, monitorando as interações entre serviços e "quebrando" o circuito para evitar falhas em cascata.

**3. Preparação para a atividade prática**

**Objetivo:** preparar os participantes para a implementação prática, explicando o que será construído e como cada parte do código se relaciona com os conceitos aprendidos.

* **Cenário da atividade:**
  + Você vai construir uma pequena aplicação de *e-commerce* utilizando os conceitos de arquitetura monolítica e, em seguida, refatorar o código para uma arquitetura de microsserviços.
* **Explicação dos passos:**
  + **Passo 1:** implementar uma versão monolítica simples do sistema de *e-commerce*;
  + **Passo 2:** refatorar o código para separar as funcionalidades em microsserviços, aplicando os padrões de design discutidos;
  + **Passo 3:** testar a aplicação para ver como a refatoração afeta a escalabilidade e manutenção.

**4. Atividade Prática em Python**

**Objetivo:** colocar em prática os conceitos e padrões aprendidos, implementando um sistema de *e-commerce* em duas etapas: primeiro como uma arquitetura monolítica e depois refatorando para microsserviços.

**Passo 1: Implementação monolítica**

class EcommerceSystem:

def \_\_init\_\_(self):

self.users = []

self.products = []

self.orders = []

def add\_user(self, user):

self.users.append(user)

def add\_product(self, product):

self.products.append(product)

def place\_order(self, user, product):

if product in self.products:

self.orders.append((user, product))

return "Order placed successfully"

return "Product not available"

# Exemplo de uso

system = EcommerceSystem()

system.add\_user("Alice")

system.add\_product("Laptop")

print(system.place\_order("Alice", "Laptop"))

**Passo 2: Refatoração para microsserviços**

class UserService:

def \_\_init\_\_(self):

self.users = []

def add\_user(self, user):

self.users.append(user)

class ProductService:

def \_\_init\_\_(self):

self.products = []

def add\_product(self, product):

self.products.append(product)

def check\_availability(self, product):

return product in self.products

class OrderService:

def \_\_init\_\_(self):

self.orders = []

def place\_order(self, user, product, product\_service):

if product\_service.check\_availability(product):

self.orders.append((user, product))

return "Order placed successfully"

return "Product not available"

# Exemplo de uso com microsserviços

user\_service = UserService()

product\_service = ProductService()

order\_service = OrderService()

user\_service.add\_user("Alice")

product\_service.add\_product("Laptop")

print(order\_service.place\_order("Alice", "Laptop", product\_service))

**Passo 3: Testes e análise**

* **Testes:** execute a versão monolítica e a versão baseada em microsserviços, comparando a facilidade de manutenção, a flexibilidade e o potencial para escalabilidade.
* **Análise:** discuta com os participantes como a aplicação dos padrões de design afetou o desenvolvimento, a manutenção e a escalabilidade do sistema.

**5. Discussão e reflexão**

**Objetivo:** refletir sobre o que foi aprendido e discutir como os padrões de design e as diferentes arquiteturas afetam o desenvolvimento de software.

* **Perguntas para conclusão da atividade:**
  + Como a implementação de microsserviços mudou a maneira como você pensou sobre a divisão do código?

Foi possível aprender na prática a diferença entre a arquitetura monolítica e a arquitetura baseada em microsserviços. A atividade mostrou como um sistema unificado (monolito) pode ser simples de construir, mas difícil de escalar ou manter. Já os microsserviços dividem responsabilidades em módulos independentes, facilitando manutenção, escalabilidade e até o uso de linguagens diferentes. A atividade também desenvolveu habilidades de organização de código, modularização e boas práticas de programação.

* + Quais desafios surgiram ao tentar refatorar o sistema para microsserviços?

**Arquitetura Monolítica:**

**Vantagens:**

* Simplicidade de desenvolvimento inicial.
* Implantação única (tudo é lançado junto).
* Ideal para aplicações pequenas.

**Desvantagens:**

* Dificuldade para escalar partes específicas.
* Acoplamento alto: alterações em uma parte afetam o todo.
* Dificuldade na manutenção à medida que o sistema cresce.

**Arquitetura de Microsserviços:**

**Vantagens:**

* Cada serviço pode ser escalado e mantido separadamente.
* Facilita o uso de diferentes tecnologias.
* Melhor divisão de responsabilidades.

**Desvantagens:**

* Mais complexidade na comunicação entre serviços.
* Maior esforço na configuração e implantação.
* Exige mais conhecimentos técnicos para gerenciar.
  + Como os padrões de design ajudaram a estruturar melhor o código em ambas as arquiteturas?

Sim, uma das principais dificuldades foi organizar corretamente a estrutura de pastas e arquivos para que o Python reconhecesse os módulos. Além disso, erros de digitação (como nomes de pastas ou arquivos) e problemas com os caminhos de importação foram desafios comuns. Porém, com ajustes e testes, foi possível resolver os problemas e entender melhor como o Python lida com pacotes e módulos.

**Lista de materiais**

* Computador com acesso à internet;
* Caderno para anotações;
* 1 caneta.

|  |
| --- |
|  |
|  |
|  |
|  |