Semana 17 - Aula 1

Tópico Principal da Aula: Frameworks Back-end e Integração com Serviços Externos

Subtítulo/Tema Específico: Introdução aos Frameworks Back-end

Código da aula: [SIS]ANO2C2B3S17A1

Objetivos da Aula:

- Compreender e configurar frameworks back-end populares (ex.: Express, Django).
- Trabalhar a curiosidade ao explorar diferentes frameworks back-end.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.
- Recurso audiovisual para exibição de vídeos e imagens.
- 1 caneta.

Exposição do Conteúdo:

Referência do Slide: Slide 2 - Frameworks back-end e integração com serviços externos

- **Definição: Frameworks back-end são ferramentas que fornecem uma estrutura organizada e um conjunto de bibliotecas e convenções para facilitar o desenvolvimento de aplicações do lado do servidor. Eles abstraem muitas das complexidades de baixo nível, permitindo que os desenvolvedores foquem na lógica de negócio e na criação de funcionalidades.**
- **Aprofundamento/Complemento (se necessário): A integração com serviços externos refere-se à capacidade de uma aplicação back-end se comunicar e utilizar funcionalidades oferecidas por outras aplicações ou plataformas através de APIs (Application Programming Interfaces) ou SDKs (Software Development Kits).** Isso é fundamental para construir sistemas modernos e interconectados, como aqueles que utilizam pagamentos online, redes sociais, ou serviços de armazenamento em nuvem.
- **Exemplo Prático: Ao criar um e-commerce, um framework back-end como Django ou Express pode gerenciar a base de dados de produtos, autenticação de usuários e processamento de pedidos. A integração com um serviço externo, como um gateway de pagamento (ex: Stripe, PagSeguro), permitiria que o e-commerce processasse transações financeiras de forma segura e eficiente, sem a necessidade de construir toda a infraestrutura de pagamento do zero.**
- **Link de Vídeo:**
  - O que são Frameworks? Para que servem? -https://youtu.be/2zqzzTnfa0E?si=_1TPIxhhnRJUdCAs

Referência do Slide: Slide 5 - Objetivos da aula

- **Definição:** Os objetivos desta aula são capacitar os estudantes a compreenderem e configurarem frameworks back-end amplamente utilizados, como Express (para Node.js) e Django (para Python). Além disso, visa desenvolver a curiosidade em explorar diferentes ferramentas disponíveis no ecossistema de desenvolvimento back-end.
- **Aprofundamento/Complemento (se necessário):** Compreender e configurar frameworks populares significa não apenas saber como utilizá-los, mas também entender a filosofia por trás de cada um, seus pontos fortes e fracos, e como eles se encaixam em diferentes cenários de projeto. Express é conhecido por sua flexibilidade e minimalismo, ideal para construir APIs RESTful e aplicações de página única (SPAs), enquanto Django é um framework "batteries-included" que oferece uma solução completa para o desenvolvimento web, com recursos como ORM, sistema de autenticação e painel administrativo prontos para uso.
- **Exemplo Prático:**
    - **Express:** Para configurar um servidor básico com Express, você instalaria o pacote `express` via npm e criaria um arquivo `app.js` com poucas linhas de código para iniciar o servidor e definir uma rota simples, como:
    - JavaScript

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Olá, mundo com Express!');
});

app.listen(port, () => {
  console.log(`Servidor Express rodando em http://localhost:${port}`);
});
```

    -
    -
    - **Django:** Para iniciar um projeto Django, você instalaria o Django via pip e usaria comandos como `django-admin startproject meuprojeto` e `python manage.py startapp minhaapp` para configurar a estrutura inicial.
- **Link de Vídeo:**
    - O que são APIs, REST e RESTful? - APIs, REST e RESTful (Pesquisado)

Referência do Slide: Slide 7 - O que nós aprendemos hoje?

- **Definição:** Ao final da aula, os alunos deverão ter compreendido as diferenças fundamentais entre Express e Django, como esses frameworks aceleram o desenvolvimento e facilitam a escalabilidade de aplicações, e como avaliar o framework mais adequado com base nos requisitos de um projeto.[13]
- **Aprofundamento/Complemento (se necessário):**
    - **Diferenças entre Express e Django:** Express é um framework minimalista e flexível para Node.js, ideal para APIs REST e microserviços, exigindo mais configurações manuais. Django é um framework web completo em Python, que segue o padrão MVT (Model-View-Template) e oferece muitos recursos integrados (ORM, admin, autenticação), sendo mais adequado para aplicações complexas e de rápido desenvolvimento.[14]
    - **Aceleração do Desenvolvimento e Escalabilidade:** Frameworks fornecem componentes reutilizáveis, padrões de design e ferramentas que automatizam tarefas comuns (como roteamento, ORM, autenticação), reduzindo significativamente o tempo de codificação. Para a escalabilidade, eles oferecem estruturas que permitem a modularização do código e a fácil integração com bancos de dados e serviços externos, facilitando a expansão da aplicação conforme a demanda cresce.
    - **Avaliação do Framework:** A escolha do framework depende de diversos fatores, como a complexidade do projeto, o prazo de entrega, as funcionalidades específicas que serão implementadas (ex: APIs complexas, painel administrativo robusto, etc.), e a expertise da equipe de desenvolvimento nas linguagens e tecnologias associadas a cada framework.
- **Exemplo Prático:** Se o projeto exige um MVP (Minimum Viable Product) rápido com uma API simples, Express pode ser a melhor escolha devido à sua leveza. Se a aplicação precisa de um sistema de administração complexo, autenticação de usuário e manipulação de banco de dados robusta, Django pode ser mais vantajoso pela sua vasta gama de funcionalidades integradas.
- **Link de Vídeo:**
    - Node.js e Express.js: O Guia Completo - Node.js e Express.js (Pesquisado)

Semana 17 - Aula 2

Tópico Principal da Aula: Frameworks Back-end e Integração com Serviços Externos

Subtítulo/Tema Específico: integração com serviços de terceiros

Código da aula: [SIS]ANO2C2B3S17A2

Objetivos da Aula:

- Compreender e configurar frameworks back-end populares (ex.: Express, Django).
- Trabalhar a curiosidade ao explorar diferentes frameworks back-end.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.
- Recurso audiovisual para exibição de vídeos e imagens.
- 1 caneta.

Exposição do Conteúdo:

API de Terceiros (Interface de Programação de Aplicações)

Uma API é um conjunto de regras e definições que permite que diferentes aplicações "conversem" entre si. Ao integrar-se a uma API, você faz requisições a URLs específicas (endpoints) e recebe dados como resposta, geralmente em um formato padronizado.

Informações complementares e exemplos:

- **Autenticação:** É o processo que garante que você tem permissão para usar a API. Os métodos mais comuns são:
  - **API Key (Chave de API):** Uma chave única é enviada na requisição, geralmente no cabeçalho (*header*) ou como um parâmetro na URL. É simples e direto.
    - **Exemplo (Google Maps):** `https://maps.googleapis.com/maps/api/geocode/json?address=1600+Amphitheatre+Parkway,+Mountain+View,+CA&key=SUA_CHAVE_API`
  - **OAuth 2.0:** Um protocolo mais seguro e complexo, usado quando uma aplicação precisa acessar dados de um usuário em outro serviço, sem expor as credenciais do usuário. Pense no "Login com Google" ou "Login com Facebook". A aplicação obtém um *token* de acesso temporário após a autorização do usuário.
    - **Exemplo (Spotify):** Sua aplicação redireciona o usuário para a página do Spotify para autorizar o acesso aos seus playlists. Após a autorização, o Spotify fornece um *token* para sua aplicação usar nas chamadas de API.
  - **Bearer Token (JWT):** Um *token* de acesso (geralmente um JSON Web Token) é enviado no cabeçalho da requisição para provar a identidade e permissão. É muito usado em sistemas que têm seu próprio login.
    - **Exemplo:** `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...`
- **Endpoints e Métodos HTTP:** Um endpoint é a URL específica onde a API pode ser acessada. Cada endpoint está associado a um ou mais métodos HTTP que definem a ação a ser realizada.
  - **GET:** Para buscar dados. (Ex: `GET /users/123` para buscar o usuário com ID 123).
  - **POST:** Para criar um novo recurso. (Ex: `POST /users` para criar um novo usuário).
  - **PUT/PATCH:** Para atualizar um recurso existente.
  - **DELETE:** Para remover um recurso.

        - **Exemplo (API do GitHub):** Para buscar os repositórios de um usuário, você faria uma requisição `GET` para o endpoint `https://api.github.com/users/NOME_DO_USUARIO/repos`.
- **Formato de Resposta:** É a estrutura como os dados são devolvidos pela API.
    - **JSON (JavaScript Object Notation):** É o formato mais comum hoje por ser leve e fácil de interpretar por diversas linguagens.
        - **Exemplo de resposta (JSON):**
        - JSON

```
{
  "id": 1,
  "nome": "João Silva",
  "email": "joao.silva@example.com",
  "ativo": true
}
```

        -
        -
    - **XML (eXtensible Markup Language):** Um formato mais antigo, mas ainda utilizado, principalmente em sistemas legados e serviços SOAP.
        - **Exemplo de resposta (XML):**
        - XML

```
<usuario>
  <id>1</id>
  <nome>João Silva</nome>
  <email>joao.silva@example.com</email>
  <ativo>true</ativo>
</usuario>
```

        -
        -

SDK (Software Development Kit - Kit de Desenvolvimento de Software)

Um SDK é um conjunto de ferramentas, bibliotecas, documentação e exemplos de código fornecido por um provedor para facilitar a integração com sua API em uma linguagem de programação específica.

Informações complementares e exemplos:

- **Abstração de Complexidade:** O SDK "esconde" a complexidade das chamadas de API diretas. Você não precisa montar requisições HTTP manualmente, tratar da autenticação em baixo nível ou interpretar a resposta bruta.
    - **Exemplo (AWS S3 - Armazenamento de Arquivos):**
        - **Usando API (conceitual):** Você teria que: 1) Gerar uma assinatura de autenticação complexa (AWS Signature V4). 2) Montar uma requisição HTTP `PUT` com os cabeçalhos corretos (`Authorization`, `x-amz-date`, `Content-Type`, etc.). 3) Enviar o corpo da requisição com os dados do arquivo. 4) Interpretar o código de status da resposta HTTP para saber se o upload foi bem-sucedido.
        - **Usando o SDK da AWS para Python (Boto3):**
        - Python

```
import boto3


# O SDK cuida da autenticação a partir das suas credenciais configuradas
s3 = boto3.client('s3')


# O SDK encapsula toda a complexidade da requisição em um método simples
s3.upload_file('meu-arquivo.txt', 'meu-bucket', 'arquivo-na-nuvem.txt')
```

        -
        -
- **Componentes Típicos de um SDK:** Além da biblioteca de código, um bom SDK geralmente inclui:
    - **Documentação detalhada:** Explicando cada função e classe.
    - **Exemplos de código:** Receitas prontas para as operações mais comuns.
    - **Ferramentas de depuração:** Para ajudar a identificar problemas.
    - **Bibliotecas de apoio:** Funções utilitárias que simplificam tarefas comuns.

---

3. Principais Diferenças: API vs. SDK

A tabela abaixo resume as principais diferenças, vantagens e desvantagens de cada abordagem.

Característica	**Integração via API Direta**	**Integração via SDK**
Definição	Um conjunto de regras e endpoints para comunicação direta entre sistemas.	Um kit de ferramentas e bibliotecas para uma linguagem específica que facilita o uso de uma API.
Nível de Abstração	Baixo. O desenvolvedor é responsável por tudo (requisições HTTP, autenticação, etc.).	Alto. A complexidade da comunicação é "escondida" em funções e objetos fáceis de usar.
Curva de Aprendizado	Mais alta. Requer conhecimento de protocolos HTTP, formatos de dados e do fluxo da API.	Mais baixa. O desenvolvedor foca na lógica de negócio, usando os métodos fornecidos pelo kit.
Controle e Flexibilidade	**Vantagem:** Total. Você tem controle granular sobre cada aspecto da requisição e pode usar qualquer biblioteca HTTP.	**Desvantagem:** Menor. Você fica limitado às funcionalidades e ao modo de operação implementados pelo SDK.
Velocidade de Desenvolvimento	**Desvantagem:** Mais lento. Exige mais código para realizar tarefas simples e maior tratamento de erros.	**Vantagem:** Mais rápido. Menos código é necessário, acelerando a implementação e reduzindo a chance de erros.

Manutenção	**Desvantagem:** Mais complexo. Se a API mudar (ex: um endpoint), você precisa atualizar seu código manualmente.	**Vantagem:** Mais simples. Geralmente, basta atualizar a versão da biblioteca do SDK para incorporar mudanças da API.
Dependências	**Vantagem:** Mínimas. Você pode precisar apenas de uma biblioteca para fazer chamadas HTTP.	**Desvantagem:** Adiciona uma nova dependência ao seu projeto, que precisa ser gerenciada e atualizada.
Quando usar?	Quando não há um SDK para sua linguagem, quando você precisa de máximo controle ou quer minimizar dependências.	Na maioria dos casos. É a forma recomendada e mais produtiva de integrar com um serviço que o oferece.

Semana 17 - Aula 3

Tópico Principal da Aula: Frameworks Back-end e Integração com Serviços Externos

Subtítulo/Tema Específico: integração com serviços de terceiros

Código da aula: [SIS]ANO2C2B3S17A3

Objetivos da Aula:

- Compreender e configurar frameworks back-end populares (ex.: Express, Django).
- Trabalhar a curiosidade ao explorar diferentes frameworks back-end.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.
- Recurso audiovisual para exibição de vídeos e imagens.
- 1 caneta.

Exposição do Conteúdo:

O gerenciamento de dependências é um pilar do desenvolvimento de software moderno. Além de instalar e remover pacotes, essas ferramentas resolvem um problema crucial: o versionamento. Elas garantem que todos os desenvolvedores de um time (e também o servidor de produção) usem exatamente as mesmas versões das mesmas bibliotecas, evitando o clássico problema de "funciona na minha máquina, mas não na sua".

---

npm (Node Package Manager)

O npm é o gerenciador de pacotes padrão para o ecossistema JavaScript e Node.js. Ele vem instalado junto com o Node.js.

Informações Adicionais:

- **package.json**: Este é o arquivo de manifesto do projeto. Ele é o coração do gerenciamento de dependências com npm. Ao iniciar um novo projeto, o comando `npm init` cria esse arquivo. Ele lista todas as dependências do projeto, scripts (para automação de tarefas como iniciar o servidor ou rodar testes), a versão do projeto, e outras metadados importantes.
- **node_modules**: É a pasta onde o npm armazena localmente todos os pacotes instalados para o projeto.
- **package-lock.json**: Este arquivo é gerado automaticamente e grava a versão exata de cada dependência instalada. Isso garante que, ao instalar as dependências em outra máquina com o comando `npm install`, as mesmas versões sejam baixadas, garantindo a consistência do ambiente.
- **Dependências de Desenvolvimento (devDependencies)**: São pacotes necessários apenas durante o desenvolvimento e teste, como `nodemon` ou `jest`, mas que não são necessários para a aplicação em produção. Eles são instalados com o comando `npm install <pacote> --save-dev`.

Exemplo Prático Detalhado (Usando Express)

Cenário: Criar um servidor web simples com Node.js e a biblioteca Express.

1. Iniciar o projeto:

Abra o terminal na pasta do seu projeto e execute:

```
Bash

npm init -y
```

Este comando cria um arquivo `package.json` com as configurações padrão.

2. Instalar o Express:

Execute o seguinte comando para instalar o Express e salvá-lo como uma dependência no package.json:

```
Bash

npm install express
```

Após a instalação, seu `package.json` terá uma seção de `dependencies` semelhante a esta:

```
JSON

"dependencies": {
  "express": "^4.17.1"
}
```

3. Criar o servidor:

Crie um arquivo chamado index.js e adicione o seguinte código:

```
JavaScript

const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Olá, mundo com Express!');
});

app.listen(port, () => {
  console.log(`Servidor rodando em http://localhost:${port}`);
});
```

4. Rodar a aplicação:

No terminal, execute:

```
Bash

node index.js
```

Ao acessar http://localhost:3000 no seu navegador, você verá a mensagem "Olá, mundo com Express!".

---

pip (Python Package Installer)

O pip é o gerenciador de pacotes padrão para Python. Ele permite instalar e gerenciar bibliotecas do Python Package Index (PyPI).

Informações Adicionais:

- **Ambientes Virtuais (virtualenv, venv)**: Esta é uma prática fundamental no desenvolvimento Python. Um ambiente virtual é uma cópia isolada do interpretador Python, que permite instalar pacotes específicos para um projeto sem afetar outros projetos ou a instalação global do Python. A ferramenta venv já vem inclusa no Python 3.
- **requirements.txt**: Este é o arquivo de manifesto padrão para projetos Python. Ele lista todas as dependências e suas versões. Ao contrário do package.json, ele não é gerado automaticamente no início, mas é criado a partir dos pacotes instalados no ambiente. O comando pip freeze > requirements.txt gera o arquivo.
- **Instalação a partir de um arquivo**: Para instalar todas as dependências listadas em um requirements.txt em outra máquina, utiliza-se o comando pip install -r requirements.txt.

Exemplo Prático Detalhado (Usando Flask)

Cenário: Criar uma aplicação web simples com Python e a biblioteca Flask.

1. Criar e ativar um ambiente virtual:

No terminal, na pasta do seu projeto, execute:

```
Bash

# Para criar o ambiente virtual

python -m venv venv



# Para ativar o ambiente (Windows)

.\venv\Scripts\activate
```

```
# Para ativar o ambiente (Linux/macOS)
source venv/bin/activate
```

Você saberá que o ambiente está ativo pois o nome dele (venv) aparecerá no início da linha do seu terminal.

2. Instalar o Flask:

Com o ambiente ativo, instale o Flask:

```
Bash
pip install Flask
```

3. Criar a aplicação:

Crie um arquivo chamado app.py e adicione o seguinte código:

```
Python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Olá, mundo com Flask!'

if __name__ == '__main__':
    app.run(debug=True)
```

4. Gerar o requirements.txt:

Para registrar a dependência, execute:

```bash
pip freeze > requirements.txt
```

O arquivo `requirements.txt` conterá o Flask e suas dependências com suas versões exatas.

5. Rodar a aplicação:

No terminal, execute:

```bash
flask run
```

Ao acessar `http://127.0.0.1:5000` no seu navegador, você verá a mensagem "Olá, mundo com Flask!".