UniSENAI

Trilha 07

APIs

Instruções para a melhor prática de Estudo

1. **Leia atentamente todo o conteúdo:** Antes de iniciar qualquer atividade, faça uma leitura detalhada do material fornecido na trilha, compreendendo os conceitos e os exemplos apresentados.

2. **Não se limite ao material da trilha:** Utilize o material da trilha como base, mas busque outros materiais de apoio, como livros, artigos acadêmicos, vídeos, e blogs especializados. Isso enriquecerá o entendimento sobre o tema.

3. **Explore a literatura:** Consulte livros e publicações reconhecidas na área, buscando expandir seu conhecimento além do que foi apresentado. A literatura acadêmica oferece uma base sólida para a compreensão de temas complexos.

4. **Realize todas as atividades propostas:** Conclua cada uma das atividades práticas e teóricas, garantindo que você esteja aplicando o conhecimento adquirido de maneira ativa.

5. **Evite o uso de Inteligência Artificial para resolução de atividades:** Utilize suas próprias habilidades e conhecimentos para resolver os exercícios. O aprendizado vem do esforço e da prática.

6. **Participe de debates:** Discuta os conteúdos estudados com professores, colegas e profissionais da área. O debate enriquece o entendimento e permite a troca de diferentes pontos de vista.

7. **Pratique regularmente:** Não deixe as atividades para a última hora. Pratique diariamente e revise o conteúdo com frequência para consolidar o aprendizado.

8. **Peça feedback:** Solicite o retorno dos professores sobre suas atividades e participe de discussões sobre os erros e acertos, utilizando o feedback para aprimorar suas habilidades.

Essas instruções são fundamentais para garantir um aprendizado profundo e eficaz ao longo das trilhas.

1. SuperTest

O SuperTest é uma biblioteca para Node.js usada para testar APIs RESTful. Ele permite fazer requisições HTTP e validar as respostas de forma simples e eficaz. O SuperTest é frequentemente utilizado junto com frameworks como Express.js e bibliotecas de teste como Jest ou Mocha.

O SuperTest facilita a automação de testes de API, permitindo verificar se endpoints estão respondendo corretamente a diferentes requisições. Ele é especialmente útil para validar que um backend está retornando os códigos de status, cabeçalhos e corpo das respostas conforme o esperado.

Exemplo: Suponha que temos uma API REST para gerenciar carros, onde podemos buscar um carro pelo ID, criar novos carros e deletá-los. Abaixo está uma implementação básica do teste.

Exemplo Prático

Cenário: Suponha que temos uma API REST para gerenciar carros, onde podemos buscar um carro pelo ID, criar novos carros e deletá-los. Abaixo está uma implementação básica do teste:

Passo 1: Criando Exemplo de Teste (get.test.js)

```
const request = require('supertest');
const app = require('../server');

Add only
describe('Função para pegar um Carro', () => {
    Add only
    it('Deve encontrar o carro', async () => {
        var userId = '67991afc2a565f894cb0ee4a';

        // Enviar requisição para buscar o carro
        const res = await request(app).get(`/api/carros/${userId}`);

        // Verifica se o conteúdo é um objeto válido
        expect(res.status).toBe(200);
    });
});
```

Executar os testes

No terminal, digite o comando:

```
npm test get.test.js
```

Configurações Básicas

No arquivo **pacakge.json** coloque o seguinte comando:

```
"devDependencies": {
    "supertest": "^7.0.0"
}
```

Lembrando que é necessário já ter o arquivo **package.json**, caso ainda não tenha inicie ele em seu projeto com o comando: **npm init -y**.

---

2. ServerRest

O ServeRest é uma API REST para testes inspirada no JSON Server e projetada para simular um backend para testes de aplicações.

Para realizar a instalação use o seguinte comando no seu terminal: **npx serverest@latest**.

Exemplo: Neste caso já temos nossa API REST para gerenciar um e-commerce, onde podemos realizar vários processos, entre eles vamos usar o método de buscar usuário.

Exemplo Prático

Cenário: Antes de mais nada devemos subir nossa API REST para gerência, onde podemos realizar vários processos. Abaixo está uma implementação básica do teste que irá buscar todos os usuários e verificar se é um objeto válido:

Passo 1: Subindo a API REST

Rodar o comando**: npx serverrest@latest**, após realizar o comando no terminal irá aparecer algumas informações de que a API está rodando, segue o exemplo abaixo:

```
ServeRest v2.29.5 está em execução
Teste o funcionamento acessando http://localhost:3000/usuarios no navegador

Quer saber as rotas disponíveis e como utilizá-las? Acesse http://localhost:3000
Quer alterar porta de execução, timeout do token, etc? Execute npx serverest --help
Para outras dúvidas acesse github.com/ServeRest/ServeRest
```

Passo 2: Criando Exemplo de Teste (getusers.test.js)

```
const request = require('supertest');

const API_URL = 'http://localhost:3000';

Add only
describe('Função para a rota de Usuário', () => {
    Add only
    it('Deve encontrar um Usuário.', async () => {
        // Enviar requisição para buscar todos os usuários
        const res = await request(API_URL).get('/usuarios');

        // Verifica se o conteúdo é um objeto válido
        expect(res.status).toBe(200);
    });
});
```

Executar os testes

No terminal, digite o comando:

```
npm test getserver.test.js
```

---

3. MongoDB

MongoDB Atlas é um serviço de banco de dados na nuvem oferecido pelo MongoDB. Ele permite hospedar, gerenciar e escalar bancos de dados MongoDB sem precisar configurar servidores manualmente. O Atlas é um Database-as-a-Service (DBaaS), oferecendo alta disponibilidade, backup automático e segurança integrada.

Características Principais do MongoDB

Modelo de Dados Flexível (Orientado a Documentos)

- Armazena dados em documentos BSON (Binary JSON), que são semelhantes a objetos JSON.
- Não exige um esquema fixo, permitindo que cada documento tenha diferentes campos e estruturas.
- Ideal para aplicações que precisam de agilidade na modelagem de dados.

Escalabilidade Horizontal (Sharding)

- Permite a distribuição de dados entre vários servidores (sharding), garantindo escalabilidade horizontal.
- Isso significa que o banco pode crescer conforme a demanda, sem perder desempenho.
- Ótimo para aplicações de grande volume de dados e alta disponibilidade.

Alta Performance e Velocidade

- Utiliza índices e caching em memória para acelerar consultas.
- Excelente para grandes volumes de dados e aplicações em tempo real.
- Operações de leitura e escrita são otimizadas, pois os dados são armazenados como documentos JSON.

Exemplo: Imagine que você está desenvolvendo uma plataforma de e-commerce e precisa escolher um banco de dados para armazenar informações sobre produtos, usuários e pedidos. A aplicação precisa lidar com:

- Alto volume de dados (milhares/milhões de produtos e usuários).
- Flexibilidade na estrutura dos produtos, pois diferentes categorias têm atributos variados.
- Escalabilidade para suportar picos de acesso em datas sazonais (Black Friday, Natal).
- Alta performance para responder rapidamente às consultas de usuários.

Como configurar e conectar ao MongoDB Atlas

Passo 1: Criar uma conta e configurar um cluster

- Acesse o site do MongoDB Atlas e crie uma conta.
- Clique em **New Project**, coloque um nome para ele e siga o fluxo.
- Clique em **Create a Cluster**.
- Selecione a opção (**Free**).
- Escolha o provedor de nuvem (AWS, GCP ou Azure) e a região, na seção Configurações e informe o nome do Cluster, exemplo **ClusterTeste**.

Passo 2: Criar um banco de dados e um usuário

- No cluster criado, clique em **Database Access** e crie um novo usuário com permissões adequadas, coloque, não esqueça de guardar a senha gerada.
- Em **Network Access**, adicione o IP do seu ambiente de desenvolvimento ou permita acessos globais (0.0.0.0/0).
- No painel do **Clusters**, clique em "Connect" e escolha **Conect your application** em **Driver**.
- Após realizar os processos o mesmo irá aparecer o exemplo completo do que será necessário.

Passo 3: Conectar ao seu projeto Node.js

Observações

- Tenha em mãos o nome e senha do usuário criado para o banco de dados e as configurações.
- Visando que já tenha o projeto criado em NodeJS seguir os passos abaixo.

Crie o arquivo **.env** e coloque a seguinte instrução como **user_db**, **pass_db** e **name_cluster**):

```
.env
MONGODB_URI=mongodb+srv://user_db:pass_db@name_cluster.mongodb.net/
```

Crie um arquivo **server.js** para configurar a conexão:

```
server.js > ...
require('dotenv').config();

const mongoose = require('mongoose');

const uri = process.env.MONGODB_URI;

// Conectando ao MongoDB
mongoose.connect(uri)
.then(() => {
    console.log('Conectado ao MongoDB');
}).catch((err) => {
    console.error('Erro ao conectar ao MongoDB:', err.message);
});
```

Configurações básicas

No arquivo **package.json** coloque os seguintes comandos:

```
"devDependencies": {
    "dotenv": "^16.4.7",
    "mongodb": "^6.12.0",
    "mongoose": "^8.9.4"
}
```

Lembrando que é necessário já ter o arquivo **package.json**, caso ainda não tenha inicie ele em seu projeto com o comando: **npm init -y**.

O que é o Mongose?

Mongoose é uma biblioteca (ou ODM – Object Data Modeling) para Node.js que facilita a interação com o MongoDB, um banco de dados NoSQL orientado a documentos. Neste contexto, usamos essa biblioteca em nossos exemplos para facilitar o desenvolvimento dos nossos testes.

---

4. Servidor Node.js com Express e MongoDB

Este projeto utiliza **Node.js**, **Express** e **MongoDB** para construir uma API REST. Ele inclui configuração do banco de dados, middleware para tratamento de JSON e definição de rotas.

Configuração

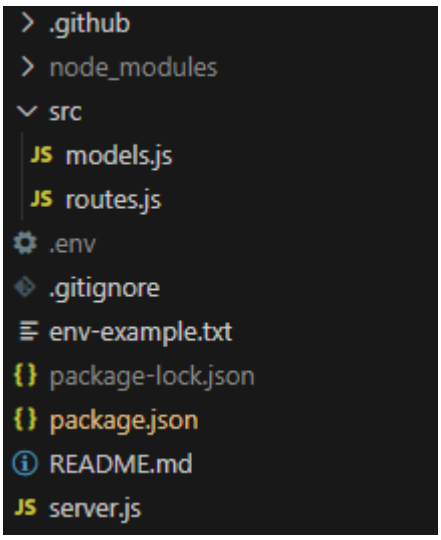
O projeto requer as seguintes bibliotecas:

- **dotenv**: Gerenciamento de variáveis de ambiente
- **express**: Framework para criação de servidores HTTP
- **mongoose**: ODM para interagir com o MongoDB

Certifique-se de criar a pasta do projeto e executar o comando dentro dela: **npm init -y**

Certifique-se de instalar as dependências com: **npm install dotenv express mongoose –save-dev**

Estrutura do Projeto



Com base na estrutura da imagem, aqui está uma breve explicação de cada pasta e arquivo:

Pastas e Arquivos

- **.github/**: Normalmente contém configurações e workflows do GitHub Actions para automação de CI/CD.
- **node_modules/**: Diretório onde o npm instala as dependências do projeto.
- **src/**: Pasta que contém os arquivos principais do código-fonte da aplicação.
- Arquivos
- **models.js** (dentro de src/): Define os modelos (schemas) do MongoDB para a aplicação usando o Mongoose.
- **routes.js** (dentro de src/): Contém as definições das rotas da API, mapeando endpoints para funções específicas.
- **.env**: Arquivo que armazena variáveis de ambiente sensíveis, como credenciais e URIs de conexão.

- **.gitignore**: Define quais arquivos e pastas devem ser ignorados pelo Git (exemplo: node_modules/ e .env).
- **env-example.txt**: Exemplo de arquivo .env, mostrando quais variáveis precisam ser definidas sem expor valores reais.
- **package-lock.json**: Arquivo gerado automaticamente pelo npm para garantir versões exatas das dependências.
- **package.json**: Arquivo de configuração do projeto Node.js, contendo metadados, dependências e scripts npm.
- **README.md**: Documento que fornece informações sobre o projeto, como instalação, configuração e uso.
- **server.js**: Arquivo principal que inicializa o servidor Express e configura a conexão com o MongoDB.

Código Fonte

Em **package.json** todas as configurações bases que é necessário que tenha no projeto depois de realizar o comando **npm install**:

```
{
    "name": "mini-api",
    "version": "1.0.0",
    "description": "Mini API em Node.js + Express + MongoDB",
    "main": "index.js",
    ▷Debug
    "scripts": {
        "test": "jest"
    },
    "keywords": [],
    "author": "Emerson Amancio",
    "license": "ISC",
    "devDependencies": {
        "dotenv": "^16.4.7",
        "express": "^4.21.2",
        "mongoose": "^8.10.0"
    }
}
```

Em **.env** ficam nossas variáveis de ambiente, neste exemplo temos somente credenciais de conexão referente a conexão com o **MongoDB**:

```
.env
1   MONGODB_URI=mongodb+srv://user_db:pass_db@name_cluster.mongodb.net/
```

Em **server.js** fica toda a nossa lógica referente a inico do servidor e conexão com nosso Banco de Dados e demais configurações:

```
require('dotenv').config();

const express = require('express');
const mongoose = require('mongoose');
const routes = require('./src/routes');

// Criando a aplicação Express
const app = express();

const uri = process.env.MONGODB_URI;

// Conectando ao MongoDB
mongoose.connect(uri)
.then(() => {
    console.log('Conectado ao MongoDB');
}).catch((err) => {
    console.error('Erro ao conectar ao MongoDB:', err.message);
});

// Middleware para permitir JSON nas requisições
app.use(express.json());

// Usando as rotas definidas em routes.js
app.use('/api', routes);

module.exports = app
```

Em **routes.js** fica toda nossa lógica de back-end (API), todas as rotas são criadas aqui:

```
const express = require('express');
const Carro = require('./models');
const router = express.Router();

// Listagem de carros
router.get('/carros', async (req, res) => {
    try {
        const carros = await Carro.find();
        res.json(carros);
    } catch (error) {
        res.status(500).json({ error: 'Erro ao listar os carros' });
    }
});
```

Em **models.js** fica nossa modelo (tabela) que irá conter na nossa coleção no MongoDB:

```
const mongoose = require('mongoose');

// Definindo o schema do carro
const CarroSchema = new mongoose.Schema({
    marca: { type: String, required: true },
    modelo: { type: String, required: true },
    ano: { type: Number, required: true }
});

// Criando o modelo do carro
const Carro = mongoose.model('Carro', CarroSchema);

module.exports = Carro;
```

Subindo a API

Para iniciar o servidor, utilize o comando no terminal: **node server.js**

Observação: Essas configurações servem somente quando for criar arquivos de testes e o mesmo deseja realizar a conexão através do **app**. Para subir a API local e utilizar alguma ferramenta para testes como por exemplo o Postman é necessário colocar o seguinte trecho de código no arquivo **server.js**:

```
// Iniciando o servidor
const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
    console.log(`Servidor rodando na porta ${PORT}`);
});
```

---

5. Wiremock

O WireMock é uma ferramenta de simulação de APIs (mocking) que permite criar servidores HTTP simulados para testar aplicações que dependem de serviços externos. Ele é útil para testar sistemas que fazem chamadas a APIs sem precisar depender de serviços reais.

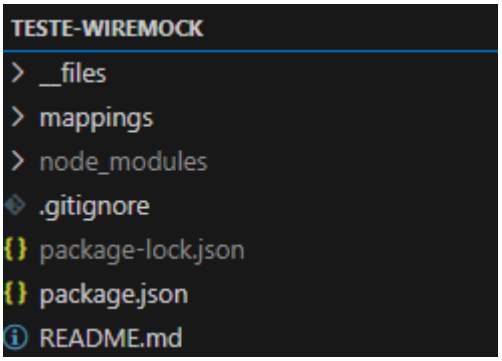
Para que serve?

O WireMock é usado principalmente para:

- **Simular APIs externas**: Ideal quando a API real não está disponível ou tem custos altos.
- **Testes automatizados confiáveis**: Reduz flutuações nos testes devido a variações nas respostas das APIs externas.
- **Desenvolvimento independente**: Permite que desenvolvedores avancem no código mesmo sem acesso à API real.

Ele pode ser usado como um servidor independente ou embutido em testes com JUnit, Cypress, Postman, entre outros.

Aqui a estrutura de pastas do projeto:



Arquivo carros.json dentro de __files:

```
{
    "carros": [
        { "id": 1, "marca": "Toyota", "modelo": "Corolla", "ano": 2022 },
        { "id": 2, "marca": "Honda", "modelo": "Civic", "ano": 2023 },
        { "id": 3, "marca": "Ford", "modelo": "Mustang", "ano": 2021 }
    ]
}
```

Neste arquivo definimos como deverá ser o nosso retorno de dados para quem realiza a solicitação do endpoint.

Arquivo carros.json dentro de mappings:

```
{
    "request": {
        "method": "GET",
        "url": "/api/carros"
    },
    "response": {
        "status": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "bodyFileName": "carros.json"
    }
}
```

Nesta etapa, realizamos todas as definições do que nosso endpoint necessita, como método, url, resposta, status da resposta, tipo do cabeçalho e seu o mesmo possui alguma informação em forma de dados, no nosso exemplo seria um arquivo no formato JSON.

Rodando nosso teste:

Com isso, nossa API ficará rodando na porta 8080,

```
npm run wiremock
```

Agora no nosso Postman ou Insomnia:

Resultado:

Preview

```
{
  "carros": [
    {
      "id": 1,
      "marca": "Toyota",
      "modelo": "Corolla",
      "ano": 2022
    },
    {
      "id": 2,
      "marca": "Honda",
      "modelo": "Civic",
      "ano": 2023
    },
    {
      "id": 3,
      "marca": "Ford",
      "modelo": "Mustang",
      "ano": 2021
    }
  ]
}
```

Nossa Coleção de Rotas:

Criamos as principais rotas para simulação de uma API em nosso projeto, basta acessar as ferramentas de testes que são o **Postman** ou o **Insomnia**.

Lista de Exercícios de Fixação

Questões Dissertativas

1. O SuperTest é uma ferramenta poderosa para testar APIs RESTful. Explique como ele pode ser utilizado para validar o funcionamento de uma API Node.js e quais são as vantagens de utilizá-lo em conjunto com Jest ou Mocha.
2. O ServeRest é uma API REST já configurada para uso em testes de software local. Explique como essa ferramenta pode ser útil para desenvolvedores e testadores ao validar funcionalidades de aplicações back-end sem a necessidade de um servidor real.
3. O MongoDB utiliza um modelo de dados baseado em documentos BSON (Binary JSON), diferindo dos bancos de dados relacionais tradicionais. Explique as vantagens desse modelo para fins de testes.
4. Imagine que você está desenvolvendo uma API RESTful para um sistema de gerenciamento de tarefas e precisa garantir que os endpoints estão funcionando corretamente. Explique quais testes seriam essenciais e como o SuperTest poderia ser utilizado nesse cenário.

Atividade Prática: Testes em APIs Públicas

Os alunos devem desenvolver e executar testes automatizados em uma API pública de sua escolha, garantindo a cobertura das principais rotas (GET, POST, PUT, DELETE etc.).

Requisitos:

Utilizar um framework ou biblioteca de testes (ex: Jest, Mocha, pytest, RestAssured).

Aplicar os conceitos abordados neste documento, incluindo:

- Configuração inicial do projeto de testes.
- Validação de status codes e respostas.
- Testes de autenticação (se aplicável).
- Tratamento de erros e edge cases.

Dicas:

- Escolha uma API com boa documentação (ex: JSONPlaceholder, ReqRes).
- Priorize testes end-to-end e integração, não apenas requisições isoladas.
- Documente os resultados (logs, relatórios de cobertura).

Os acadêmicos deverão entregar os seguintes itens:

- Scripts de testes desenvolvidos no repositório GitHub.

Boa sorte e bom aprendizado!