

Java revenge

ATIVIDADES DE HOJE

Horário das atividades

- 13h30 Faísca
- 13h45 Passo a Passo
- 15h00 Intervalo
- 15h30 Continuação do Exercícios



Passo a Passo

Prestem atenção!

Passo 1:

O que vamos retornar?

Devemos retornar uma lista de missões!



Passo 2:

Criar a estrutura do serviço

- Vocês já possuem uma aplicação em Java com Spring que possui as duas dependências necessárias (supostamente...) e elas são:
- **Spring Web** (conjunto de ferramentas que ajuda os desenvolvedores a criar sites e aplicativos)
- **Spring JPA** (facilita a manipulação dos dados, para que seja mais simples armazenar e recuperar essas informações no banco de dados.)
- Caso vocês não tenham acessem o spring initializer e criem um projeto maven/gradle com essas dependências.

Passo 3:

Spring precisa conversar com o Banco

- Com as dependências necessárias, AGORA precisamos criar representações do banco de dados dentro da nossa aplicação.
- Lembrem-se: Representação é um MODELO, logo vamos implementar classes que “traduzam” as colunas de uma tabela do banco.
- Precisamos de modelos das entidades!
- Mas como? Ora, ora...



Passo 3:

Spring precisa conversar com o Banco

```
@Entity
public class Ninja {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String vila;
    private String status;
    private String nivelExperiencia;

    // Getters e setters ou usa o Lombok Genin
}
```

Passo 3:

Spring precisa conversar com o Banco

```
@Entity
public class Missao {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String classificacao;
    private String tipoMissao;
    private String status;

    @OneToOne
    @JoinColumn(name = "ninja_id")
    private Ninja ninjaResponsavel;

    // Getters e setters ou vocês já sabem o que usar né?
}
```


Passo 3:

Spring precisa conversar com o Banco

- Com as nossas entidades criadas, precisamos implementar a consulta ao banco para finalizar esse passo 3.
- Com os exemplos das aulas anteriores é trivial (ou deveria ser kkkk):

```
public interface NinjaRepository extends JpaRepository<Ninja, Long> {  
    List<Ninja> findByNivelExperiencia(String nivelExperiencia);  
}
```

```
public interface MissaoRepository extends JpaRepository<Missao, Long> {  
    List<Missao> findByTipoMissaoAndNinjaResponsavel_NivelExperiencia  
        (String tipoMissao, String nivelExperiencia);  
}
```

Passo 4:

Criar o endpoint!

- Como vimos anteriormente, a gente manipula os endpoints numa classe chamada Controller;
- Controller de que? De Missões, logo o nome da classe é MissaoController;
- Nossa API é Rest? Se sim, usamos a anotação @RestController.
- Tão sacando né? Pensou, associou e GG.
- Vamos ao código:



Passo 4:

Criar o endpoint!

```
@RestController
@RequestMapping("/api/missao")
public class MissaoController {

    @Autowired
    private MissaoRepository missaoRepository;

    @Autowired
    private NinjaRepository ninjaRepository;

    @GetMapping("/resgate/srank")
    public ResponseEntity<List<Missao>> getMissaoResgateSRank() {
        List<Missao> missaoList = missaoRepository.
            findByTipoMissaoAndNinjaResponsavel_NivelExperiencia("Resgate", "S-Rank");
        return new ResponseEntity<>(missaoList, HttpStatus.OK);
    }
}
```

Pronto!

Ou será que não?



Resposta: Não!

Passo 5: Tá na hora do padrão

- Jovens, vocês precisam lembrar que padronização de código é um item que sempre vai nos ajudar, então vamos ir atrás disso...



Passo 6:

Separação de responsabilidades

- Como já falamos anteriormente, precisamos dividir as responsabilidades nas soluções que criamos;
- Além de que é MELHOR que a Controller não tenha lógica em si e seja uma porta de entrada e saída para a nossa API;
- Maravilha e como faz?
- A wild Service Class appears...



Passo 6:

Separação de responsabilidades

- Precisamos de uma classe Service que contenha a logica em si da funcionalidade:

```
@Service
public class MissaoService {

    @Autowired
    private MissaoRepository missaoRepository;

    public List<Missao> getMissaoExploracaoSRank() {
        return missaoRepository.findByTipoMissaoAndNinjaResponsavel_NivelExperiencia("Exploração", "S-Rank");
    }
}
```

Passo 6:

Separação de responsabilidades

- Agora “refatoramos” a nossa Controller para ficar assim:

```
@RestController
@RequestMapping("/api/missao")
public class MissaoController {

    @Autowired
    private MissaoService missaoService;

    @GetMapping("/exploracao/srank")
    public ResponseEntity<List<Missao>> getMissaoExploracaoSRank() {
        List<Missao> missaoList = missaoService.getMissaoExploracaoSRank();
        return new ResponseEntity<>(missaoList, HttpStatus.OK);
    }
}
```

Passo 7:

Teste Unitários

- Agora chegamos na parte em poucos adoram, muitos odeiam, mas quem faz se destaca nesse mundão veio de devs safados.
- Nesse caso precisamos de um exemplo maior, se preparem para os prints:



Passo 7:

Teste Unitários

- Primeiro precisamos criar a classe de teste e arrumar as anotações:

```
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

@SpringBootTest
class MissaoServiceTest {

    @Mock
    private MissaoRepository missaoRepository;

    @InjectMocks
    private MissaoService missaoService;
```

Passo 7:

Teste Unitários

```
@Test
void getMissaoExploracaoSRank() {
    // Criar dados de exemplo
    Ninja ninjaSRank = new Ninja();
    ninjaSRank.setNivelExperiencia("S-Rank");

    Missao missao1 = new Missao();
    missao1.setTipoMissao("Resgate");
    missao1.setNinjaResponsavel(ninjaSRank);

    Missao missao2 = new Missao();
    missao2.setTipoMissao("Resgate");
    missao2.setNinjaResponsavel(ninjaSRank);

    List<Missao> missaoList = Arrays.asList(missao1, missao2);

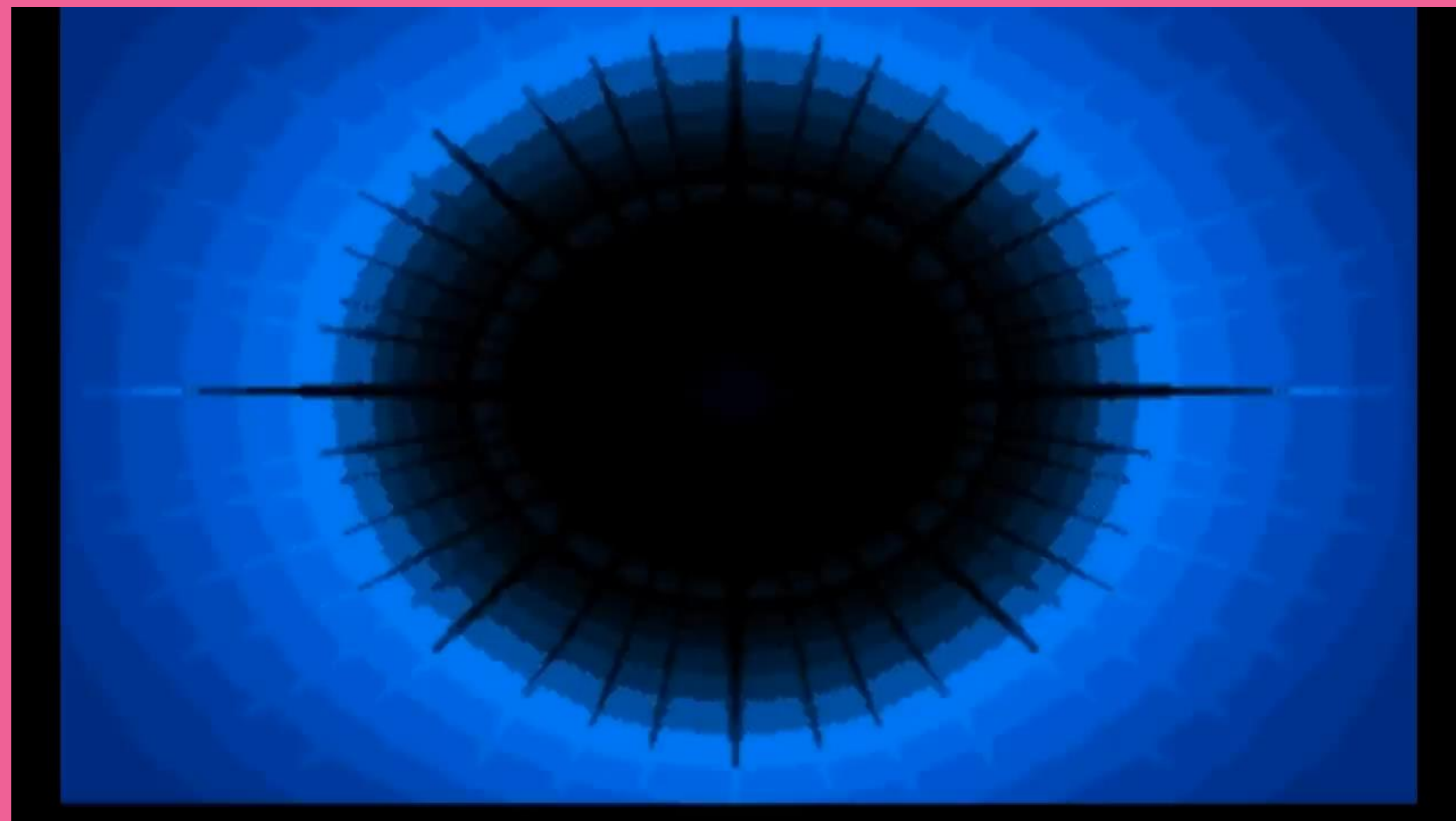
    // Configurar comportamento do mock
    Mockito.when(missaoRepository.findByTipoMissaoAndNinjaResponsavel_NivelExperiencia("Resgate", "S-Rank"))
        .thenReturn(missaoList);

    // Chamar o método do serviço
    List<Missao> result = missaoService.getMissaoExploracaoSRank();

    // Verificar se o resultado é o esperado
    assertEquals(missaoList.size(), result.size());
    assertEquals(missaoList, result);
}
```

Agradecemos a sua atenção!

***Ou será
que não?***



Novo desafio

Se arrumem nos seus grupos para:

- Finalizarem o desafio da aula passada;
- Seu código tem exceções? **tá na hora de fazer nééééééé;**
- Implementar essa consulta da apresentação, só que precisa utilizar streams! Na prática vocês precisam trazer todas as missões de exploração do banco e filtrar pelos ninjas de rank alto;
- Final challenge, atualizar a api de vocês para usar o Swagger.
<https://swagger.io/>
<https://swagger.io/solutions/api-documentation/>
<https://swagger.io/tools/swagger-ui/>



Mockito

Test and Spy

O que é o Mockito

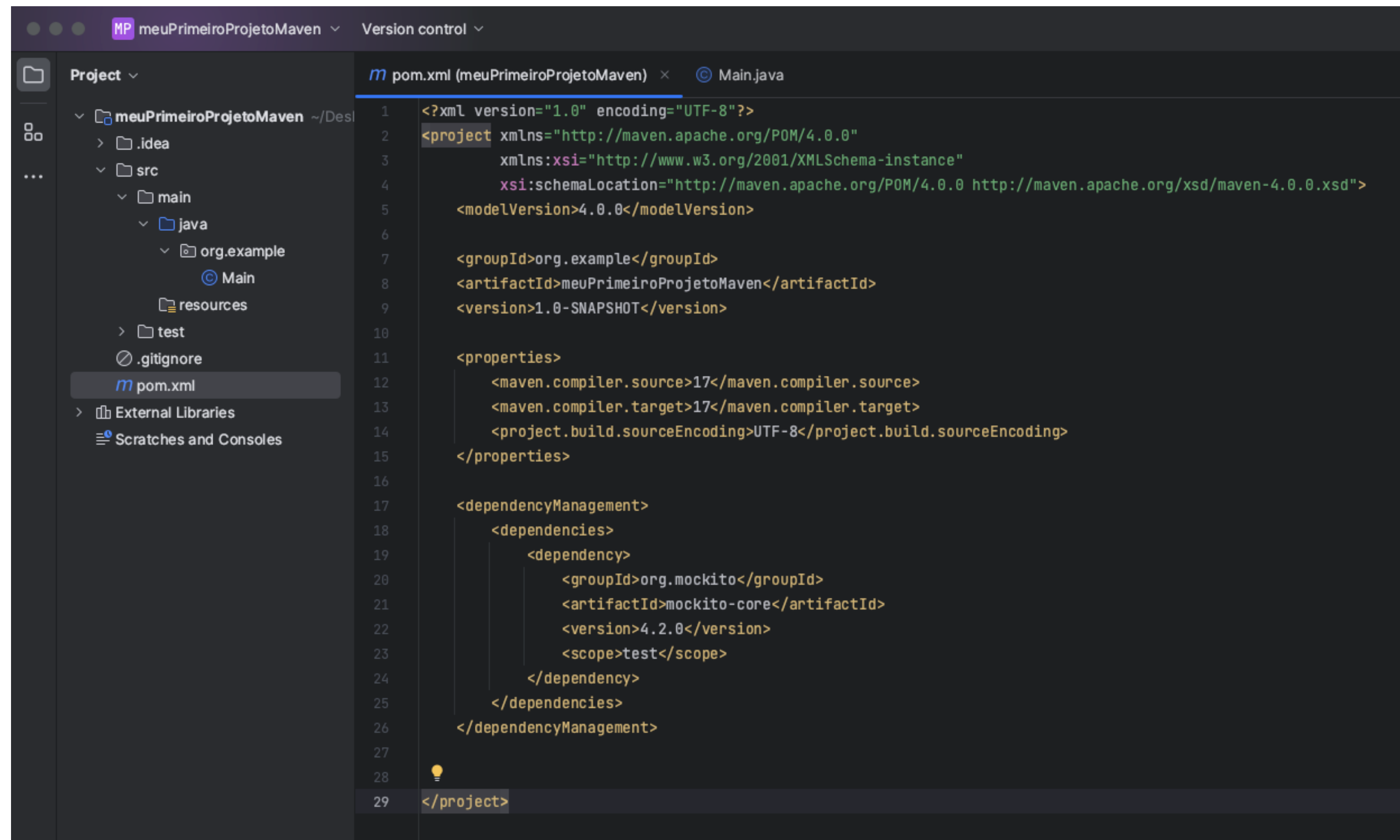
O Mockito é um framework de Test and Spy e o seu principal objetivo é simular a instancia de classes e comportamento de métodos. Ao mockar uma dependência com o mockito, podemos fazer com que a classe que vai ser testada simule o método testado e suas dependências. Durante o mock podemos configurar o retorno e ações de acordo com o necessidade do teste.

Principais Funções

- **Mock:** cria uma instancia de uma classe, porém Mockada. Se você chamar um metodo ele não irá chamar o metodo real, a não ser que você queira.
- **Spy:** cria uma instancia de uma classe, que você pode mockar ou chamar os metodos reais. É uma alternativa ao InjectMocks, quando é preciso mockar metodos da propria classe que esta sendo testada.
- **InjectMocks:** criar uma intancia e injeta as dependências necessárias que estão anotadas com @Mock.
- **Verify:** verifica a quantidade de vezes e quais parametros utilizados para acessar um determinado metodo.
- **When:** Após um mock ser criado, você pode configurar ações na chamada e o retorno.
- **Matchers:** permite a verificação por meio de matchers de argumentos (anyObject(), anyString() ...)

Utilizando Mockito no IntelliJ IDEA

Crie um novo projeto Java no IntelliJ IDEA. Em seguida, adicione a dependência do Mockito ao arquivo pom.xml do seu projeto. *Obs.: Projeto Java Maven



The screenshot shows the IntelliJ IDEA interface with a Maven project named 'meuPrimeiroProjetoMaven'. The left sidebar displays the project structure, including 'src/main/java/org.example/Main.java' and 'pom.xml'. The main editor window shows the 'pom.xml' file with the following XML content:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>meuPrimeiroProjetoMaven</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>4.2.0</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

</project>
```


Utilizando Mockito para simular dependências

Vamos imaginar uma classe chamada “***UserService***” que depende da classe “***UserRepository***”:

```
public class UserService {  
    private UserRepository userRepository;  
  
    public UserService(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
  
    public boolean isActive(int userId) {  
        User user = userRepository.findById(userId);  
        return user != null && user.isActive();  
    }  
}
```

Para testar o método “isActive”, nós podemos usar o Mockito para criar um mock do “UserRepository” e definir o seu comportamento. Criar uma nova classe chamada “UserServiceTest” e adicionar o código no slide a seguir:



```

import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

public class UserServiceTest {

    @Test
    public void testIsUserActive() {
        // Create a mock UserRepository
        UserRepository userRepository = mock(UserRepository.class);

        // Define the behavior of the mock UserRepository
        User activeUser = new User(1, "John Doe", true);
        when(userRepository.findById(1)).thenReturn(activeUser);

        // Instantiate UserService with the mock UserRepository
        UserService userService = new UserService(userRepository);

        // Test the isUserActive method
        assertTrue(userService.isUserActive(1), "User with ID 1 should be active");

        // Verify the mock UserRepository's findById method was called with the
        verify(userRepository, times(1)).findById(1);
    }
}

```

```

@Test
public void testIsUserInactive() {
    // Create a mock UserRepository
    UserRepository userRepository = mock(UserRepository.class);

    // Define the behavior of the mock UserRepository
    User inactiveUser = new User(2, "Jane Doe", false);
    when(userRepository.findById(2)).thenReturn(inactiveUser);

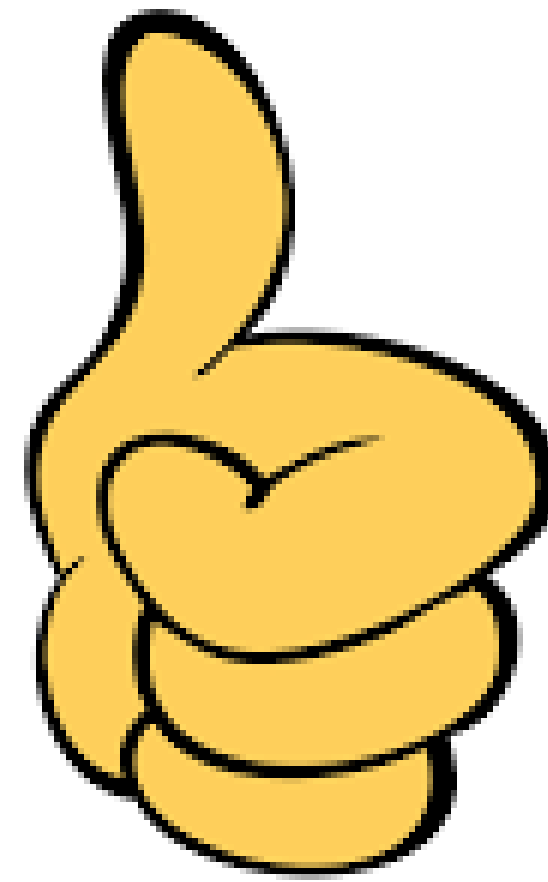
    // Instantiate UserService with the mock UserRepository
    UserService userService = new UserService(userRepository);

    // Test the isUserActive method
    assertFalse(userService.isUserActive(2), "User with ID 2 should be inactive");

    // Verify the mock UserRepository's findById method was called with the
    verify(userRepository, times(1)).findById(2);
}
}

```

Com esses testes implementados, usamos efetivamente o Mockito para simular a dependência UserRepository e testar a classe UserService isoladamente. Isso nos permite focar no comportamento específico da classe UserService sem nos preocupar com os detalhes de implementação do UserRepository.



Swagger

definir, criar, documentar e consumir APIs REST;

O que é o Swagger?

O Swagger é um framework composto por diversas ferramentas que, independente da linguagem, auxilia a descrição, consumo e visualização de serviços de uma API REST.



O que é o Swagger?

No framework Swagger, existem ferramentas para os seguintes tipos de **tarefas** a serem realizadas para o completo desenvolvimento da API de um serviço WEB:

O que é o Swagger?

1) A **especificação** da API consiste em determinar os modelos de dados que serão entendidos pela API e as funcionalidades presentes na mesma. Para cada funcionalidade, é preciso especificar o seu nome, os parâmetros que devem ser passados no momento de sua invocação e os valores que irão ser retornados aos usuários da API. Entre esta ferramentas, podemos citar o *OpenAPI Specification*.

O que é o Swagger?

2) Após especificar a API, o framework facilita sua **implementação**, com a ferramenta Swagger Codegen é possível montar o código inicial automaticamente nas principais linguagem de programação.

O que é o Swagger?

3) Os **testes** de API são extremamente importantes, pois ajudam a garantir o funcionamento, o desempenho e a confiabilidade da sua aplicação. O Swagger oferece ferramentas para teste manuais, automatizados e de desempenho

O que é o Swagger?

4) Para auxiliar na utilização da API, o Swagger dispõe de ferramenta para deixar a visualização mais intuitiva, permitindo também que interajam com a API.

Swagger - documentação API:

O Swagger permite criar a documentação da API de 3 formas:

- 1- **Automaticamente:** Simultaneamente ao desenvolvimento da API é gerada a documentação.
- 2- **Manualmente:** Permite ao desenvolvedor escrever livremente as especificações da API e as publicar posteriormente em seu próprio servidor.
- 3- **Codegen:** Converte todas as anotações contidas no código fonte das APIs REST em documentação.

Swagger - Ferramentas:

Swagger Editor

O Swagger Editor é uma ferramenta online que permite criar manualmente a documentação da API. Ao utilizar YAML, faz com que o desenvolvedor não tenha dificuldades em descrever os seus serviços. Outro benefício do Editor é possuir um conjunto de templates de documentos que servem como base para quem não deseja iniciar a documentação do “zero”.

Swagger - Ferramentas:

Swagger Editor

The image shows the Swagger Editor interface. On the left, the Swagger 2.0 definition is displayed in a code editor. On the right, the visual representation of the API is shown.

Swagger Editor Code:

```
1 swagger: "2.0"
2 info:
3   description: "API para informações sobre a Orquestra de Ouro Preto"
4   version: "1.0.0"
5   title: "Swagger Orquestra OP"
6
7 host: ""
8 basePath: "/"
9 tags:
10  - name: "News"
11    description: "Everything about your News"
12  - name: "MainPost"
13    description: "Everything about your MainPost"
14  - name: "Event"
15    description: "Everything about your Event"
16 schemes:
17  - "https"
18  - "http"
19 paths:
20  /news:
21    post:
22      tags:
23        - "News"
24      summary: "Adiciona news"
25      description: ""
26
27      consumes:
28        - "application/json"
29      produces:
30        - "application/json"
31      parameters:
32        - in: "body"
33          name: "body"
34          required: true
35          schema:
36            example:
37              title: "Academia em OP"
38              headline: "academia_7_ultimas_palavras"
39              author: "admin"
40              date: "13/04/2020"
41              photoUrl: "http://www.orquestraouropreto.com.br/site/wp-content/uploads/2020/03/ooop-20-Academia-Semana-Santa-OP-Cover-Facebook-1"
```

Swagger Orquestra OP 1.0.0

[Base URL:]

API para informações sobre a Orquestra de Ouro Preto

Schemes: HTTPS

News Everything about your News

- POST** /news Adiciona news
- GET** /news Buscar todas News
- GET** /news/{id} Busca por id
- DELETE** /news/{id} Deleta uma News por ID

MainPost Everything about your MainPost

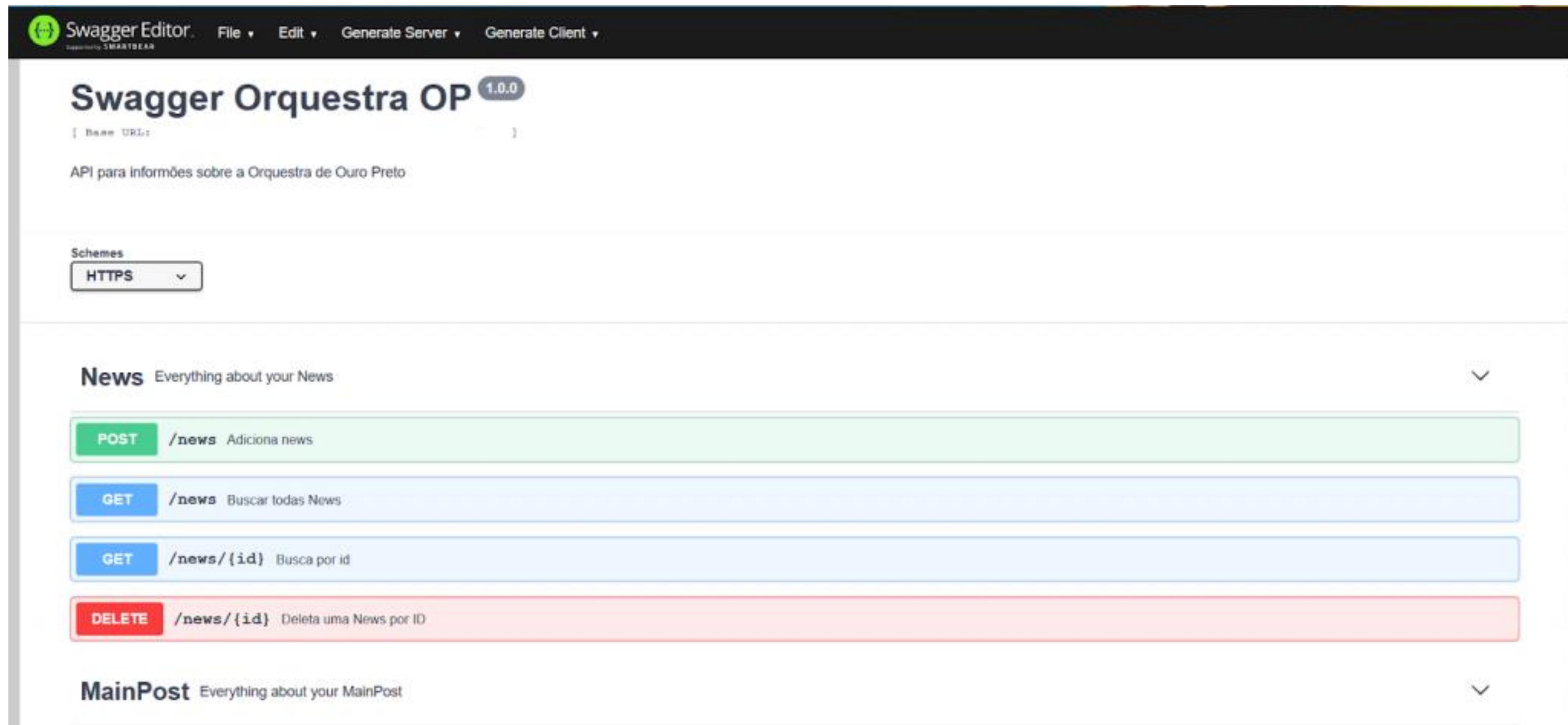
Swagger - Ferramentas:

Swagger UI

Com o Swagger UI, a partir da especificação da API, podemos criar documentações elegantes e acessíveis ao usuário, permitindo assim uma compreensão maior da API, pois além de poder ver os endpoints e modelos das entidades com seus atributos e respectivos tipos, o módulo de UI possibilita que os usuários da API interajam intuitivamente com a API usando uma sandbox. A sandbox é uma plataforma de testes onde as aplicações podem ser alteradas sem interferir no meio de produção. Nela, os desenvolvedores podem executar todas as operações de mudanças experimentais que vão garantir o bom funcionamento da solução, evitando danos que possam prejudicar o sistema.

Swagger - Ferramentas:

Swagger UI



Swagger - Ferramentas:

Swagger Codegen

O Swagger Codegen é um projeto muito interessante, a partir da especificação em YAML gera automaticamente o “esqueleto” da API em diferentes linguagens, como Java, Python, Kotlin, Lua, Haskell, C++, entre outras. Isso mesmo, com algumas linhas de comando você cria todo o código inicial da sua API na linguagem que desejar. Se você deseja utilizar o Codegen é recomendável que primeiro verifique quais versões do OpenAPI Specification ele suporta.

Você pode conferir no link:

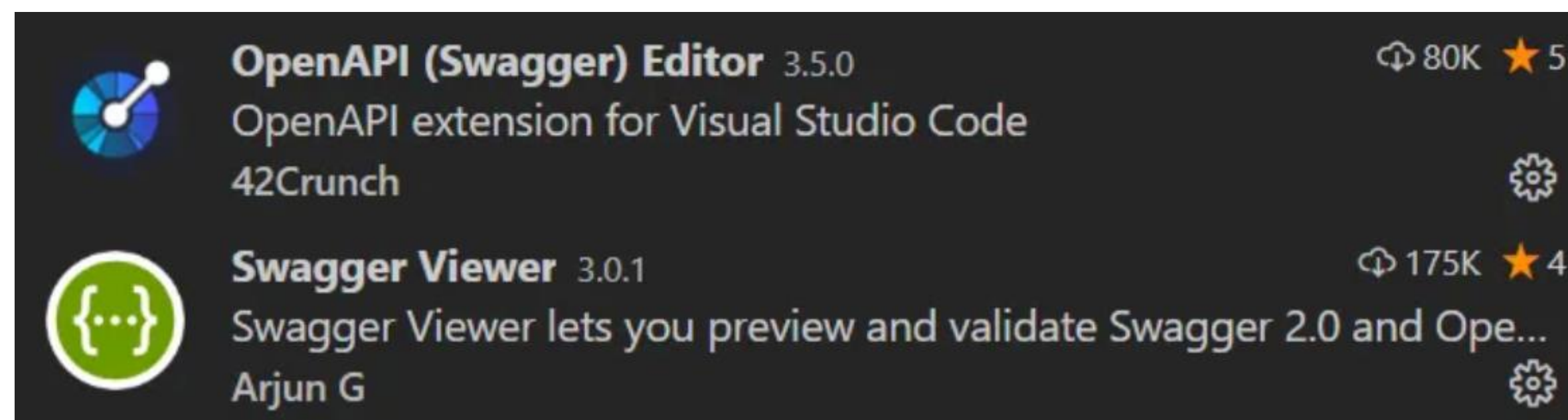
<https://github.com/swagger-api/swagger-codegen#compatibility>.

<https://www.jetbrains.com/help/idea/openapi.html#remote-spec>

Swagger no VSCode

Editor Online: <https://editor.swagger.io/>

- Ele possui um exemplo e uma pré-visualização imediatos.
- Se você está preocupado que o Editor Online possa perder suas alterações, assim como eu, você pode simplesmente criar um arquivo .yaml em sua máquina local. Instale as seguintes 2 Extensões do Visual Studio Code. Você está pronto para começar.



- O Editor OpenAPI (Swagger) realiza destaque de sintaxe, verificação de erros, preenchimento automático, etc.
- O Swagger Viewer fornece uma prévia enquanto você edita o arquivo yaml.

Swagger

Swagger
pronto para usar

The image shows a Visual Studio Code window with two panes. The left pane displays a YAML file named `example.yaml` with the following content:

```
1 swagger: "2.0"
2 info:
3   -description: "This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc://freenode.net. #swagger. For this sample, you can use the api key special-key to test the authorization filters.
4   -version: "1.0.0"
5   -title: "Swagger Petstore"
6   -termsOfService: "http://swagger.io/terms/"
7   -contact:
8     -email: "apiteam@swagger.io"
9   -license:
10    -name: "Apache 2.0"
11    -url: "http://www.apache.org/licenses/LICENSE-2.0.html"
12 host: "petstore.swagger.io"
13 basePath: "/v2"
14 tags:
15   - name: "pet"
16     -description: "Everything about your Pets"
17     -externalDocs:
18       -description: "Find out more"
19       -url: "http://swagger.io"
20   - name: "store"
21     -description: "Access to Petstore orders"
22   - name: "user"
23     -description: "Operations about user"
24   -externalDocs:
25     -description: "Find out more about our store"
26     -url: "http://swagger.io"
27 schemes:
28   - "https"
29   - "http"
30 paths:
31   - /pet:
32     - post:
33       - tags:
34         - "pet"
35       - summary: "Add a new pet to the store"
36       - description: ""
37       - operationId: "addPet"
38       - consumes:
39         - "application/json"
40         - "application/xml"
```

The right pane shows the Swagger Preview for the `Swagger Petstore 1.0.0` API. It includes the base URL `petstore.swagger.io/v2`, a description of the sample server, and links for terms of service, contact, and more information. Below this, there is a section for Schemes with a dropdown menu set to `HTTPS` and an `Authorize` button. A `Filter by tag` input field is also present. The `pet` tag is selected, showing a list of endpoints:

- POST** `/pet`: Add a new pet to the store
- PUT** `/pet`: Update an existing pet
- GET** `/pet/findByStatus`: Finds Pets by status

A large, stylized pink star with five points, positioned on the left side of the slide. It is partially obscured by the text.

Agradecemos a sua atenção!