Criação de API Rest usando Java, VSCode da Microsoft, Maven, OpenAPI (Swagger) e Spring Boot.

Autor: Rodrigo Pimenta Carvalho - 13/10/2023.

Conforme [2], o Spring Boot facilita a criação de APIs Rest, com Java. E isso pode ser feito no VSCode, conforme mostrado nesse tutorial. Para tal, é necessário ter instalado na máquina o VSCode, o Java e também o Maven. A metodologia mostrada abaixo leva em consideração o uso também do Maven.

**Maven:**

**O que é o Maven?**

O Maven é uma ferramenta de gerenciamento de *build* e dependências para projetos Java. Ele simplifica o processo de construção, distribuição e manutenção de aplicativos Java, garantindo que as bibliotecas e dependências corretas sejam incorporadas ao projeto.

**Quando usar o Maven:**

1. Gerenciamento de Dependências: Use o Maven quando você precisar gerenciar as dependências do seu projeto Java. Ele automatiza o download e a inclusão de bibliotecas de terceiros em seu projeto.
2. Construção de Projetos: Maven é ideal para automatizar o processo de compilação, empacotamento e distribuição de projetos Java.
3. Padronização de Estrutura: O Maven incentiva a padronização na estrutura de diretórios e no ciclo de vida do projeto, o que é útil em projetos com múltiplos colaboradores.
4. Relatórios e Documentação: Ele fornece recursos para gerar relatórios de projeto e documentação automaticamente.

**Spring:**

**O que é o Spring?**

O Spring é um framework de desenvolvimento de aplicativos Java que simplifica o desenvolvimento de aplicativos empresariais. Ele oferece um conjunto de módulos e funcionalidades que abordam diversos aspectos do desenvolvimento, como injeção de dependência, controle transacional, segurança e acesso a dados.

**Quando usar o Spring:**

1. Desenvolvimento de Aplicativos Empresariais: Use o Spring quando estiver desenvolvendo aplicativos Java empresariais que requerem uma arquitetura modular e escalável.
2. Injeção de Dependência: O Spring é especialmente valioso para gerenciar a injeção de dependência, permitindo a criação de componentes facilmente testáveis e altamente acoplados.
3. Acesso a Dados: O Spring fornece suporte eficaz para acesso a bancos de dados, tornando mais simples a integração com sistemas de armazenamento de dados.
4. Segurança: Ele oferece recursos robustos de segurança para proteger seu aplicativo contra ameaças.
5. Controle Transacional: O Spring facilita a implementação de transações em aplicativos que requerem consistência de dados.

Em resumo, o Maven é uma ferramenta de construção e gerenciamento de dependências usada principalmente para gerenciar bibliotecas e simplificar o processo de construção, enquanto o Spring é um framework que fornece um conjunto abrangente de funcionalidades para desenvolver aplicativos Java empresariais, incluindo injeção de dependência, segurança e acesso a dados. Em muitos casos, eles são usados em conjunto, com o Maven gerenciando as dependências do projeto e o Spring facilitando o desenvolvimento de aplicativos Java robustos e escaláveis.

1. Criando o projeto no VSCode

Conforme dito no web site Treinaweb [1]: “até pouco tempo o Java possuia um ciclo de atualizações grande, entre o lançamento da versão 5 (07/2004) até a versão 9 (07/2017) foram treze anos. A partir desta nona versão o ciclo foi alterado e agora há lançamento de novas versões duas vezes ao ano, nos meses de março e setembro, por isso que entre a versão 9 e a 12, o intervalo foi de 1 ano e meio.

Este grande intervalo entre as antigas versões era necessário devido ao seu complexo processo de atualização. Visando estabilidade e segurança, cada versão passava por vários testes e interações até serem liberados para os usuários. É preciso dizer que infelizmente nem sempre a versão lançada fornecia estabilidade e segurança. Qualquer usuário que já tenha instalado o Java no seu computador sabe que mensalmente havia alguma atualização de segurança.

Tudo isso dificultava a adição de novos recursos na linguagem, o que permitiu o destaque de alguns frameworks. Dentre vários, o que mais ganhou os holofotes foi o Spring. Criado para aplicações Java EE, hoje o Spring é um projeto que contém várias bibliotecas, de segurança (Spring Security) à big data (Spring XD).

Dentre todos os projetos abaixo da asa do Spring, o que mais se destaca é o Spring Boot.” “...O maior benefício deste *framework* é que permite o desenvolvedor se preocupar com o ponto mais importante da aplicação, as regras de negócio.”

Existem algumas formas de criar uma aplicação Spring Boot. Ela pode ser criada no Eclipse, utilizando a IDE [Spring Tool Suíte](https://spring.io/tools" \t "/home/rodrigopimenta/Documentos\\x/_blank), que também é fornecida como plug-in para o [Eclipse](https://marketplace.eclipse.org/content/spring-tools-4-spring-boot-aka-spring-tool-suite-4" \t "/home/rodrigopimenta/Documentos\\x/_blank), [Visual Studio Code](https://marketplace.visualstudio.com/items?itemName=Pivotal.vscode-boot-dev-pack" \t "/home/rodrigopimenta/Documentos\\x/_blank) e [Atom](https://atom.io/packages/spring-boot" \t "/home/rodrigopimenta/Documentos\\x/_blank). Ou mesmo no site [Spring Initializr](https://start.spring.io/" \t "/home/rodrigopimenta/Documentos\\x/_blank).

1. Abra o VSCode a partir de um diretório onde você deseja criar o seu projeto de API Rest. Instale nele as seguintes duas extensões:

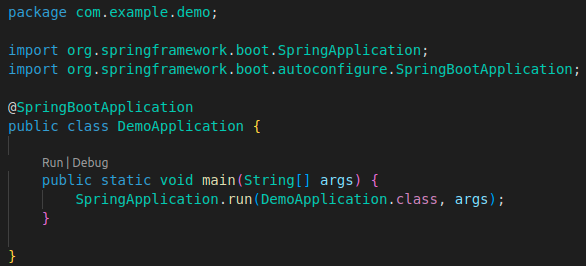
**Spring Boot Tools** e

**Spring Initializr Java Support**

1. Para criar o projeto, digite Ctrl + Shift + p. Na janela que se abre, escolha a opção de Spring Initializr. Em seguida escolha a versão. Ex: 2.7.5. Escolha Java. Digite o pacote do seu projeto. Ex: br.com.inatel, ou br.aluno.inatel, etc. Tecle <enter>. Digite o nome do seu projeto. Ex: Demo. Tecle <enter>.Escolha criar um Jar, não um War. Escolha a versão do Java. Ex: 11. Escolha as dependências: Spring web e Spring Boot DevTools. Tecle <enter>. Na janela que se abre, escolha gerar o projeto no diretório corrente.

Aqui usa-se escolher jar e não war, porque esse exemplo de projeto não será implantando num web server. Na verdade, o próprio jar conterá em si um web server embutido. Então, ao executar esse jar, o web server estará executando junto, para suportar o web service sendo criado. Esse web server será o Tomcat.

1. Na pasta main do seu projeto, foi criada automaticamente uma classe como essa abaixo:



1. Em seguida, na pasta raiz do projeto, crie um script para executar o jar e então deixar o web server rodando com o web service ativo. Por exemplo,um script .sh para o Linux pode ter o seguinte código:

#!/bin/bash

export JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-amd64

exec mvn "$@"

Caso precise de uma versão desse script para rodar no Windows, como um .bat, verifique no ChatGPT como seria o correspondente então. O nome do script pode ser, por exemplo, rodaProjeto.sh. Isso garantirá que o Java 11 será usado, mesmo se a variável de ambiente JAVA\_HOME estiver apontando para outro Java de outra versão, também instalado na máquina. Altere as permissões desse arquivo. Pode usar o comando ***chmod 777*** nele. Para rodar o projeto nesse exemplo, pode-se executar o seguinte na linha de comando:

**./rodaProjeto.sh spring-boot:run**

1. Ao criar um projeto com Spring para API Rest, como visto, o Maven também será usado. Ou seja, o VSCode aqui providencia o uso do Maven de forma correta. Nesse caso, é criado automaticamente o arquivo pom.xml. Substitua todo o conteúdo do arquivo pom.xml pelo XML dado abaixo:

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.15</version> <!-- Versão do Spring Boot. Roda com JAva 11, por exemplo -->

<relativePath/>

</parent>

<groupId>br.com.inatel</groupId>

<artifactId>restexample</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>restexample</name>

<description>Demo project for Spring Boot</description>

<!--properties>

<java.version>11</java.version>

</properties-->

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId> <!-- Starter para criar uma aplicação web -->

</dependency>

<!--dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId> <!-- Starter para testes -->

<scope>test</scope>

</dependency>

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

<version>2.0.1.Final</version> <!-- Ou uma versão compatível -->

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId> <!-- Plugin para empacotar o projeto em um jar -->

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>exec-maven-plugin</artifactId>

<version>3.0.0</version>

<executions>

<execution>

<goals>

<goal>exec</goal>

</goals>

<configuration>

<executable>java</executable>

<arguments>

<argument>-Dmaven.compiler.source=11</argument> <!-- Versão do Java para o Tomcat usar -->

<argument>-Dmaven.compiler.target=11</argument>

<argument>-jar</argument>

<argument>${project.build.directory}/${project.build.finalName}.jar</argument>

</arguments>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.10.1</version>

<configuration>

<source>11</source>

<target>11</target>

</configuration>

</plugin>

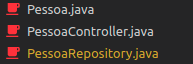
</plugins>

</build>

</project>

Esse arquivo XML contem as configurações necessárias para o Maven instalar as dependências que o projeto precisa. Também estão configuradas versões de software compatíveis entre si.

1. A classe criada automaticamente é encarregada de iniciar tudo o que o projeto vai precisar, como a execução das dependências do Maven, execução do web server, etc. Qualquer outra classe que seja criada no mesmo diretório da classe principal ou em subdiretórios, poderão ser consideradas também como componentes pelo Spring. Assin sendo, crie agora mais 3 classes no mesmo diretório da classe principal, com os seguintes nomes e conteúdos:



public class Pessoa {

private int id;

private String nome;

private int idade;

public int getId() {

return id;

}

public int getIdade() {

return idade;

}

public void setIdade(int idade) {

this.idade = idade;

}

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

public void setId(int id) {

this.id = id;

}

}

//----------------------------------------

import java.util.List;

import java.util.Optional;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import br.com.inatel.restexample.Pessoa;

import br.com.inatel.restexample.PessoaRepository;

@RestController

public class PessoaController {

@Autowired

private PessoaRepository \_pessoaRepository;

@RequestMapping(value = "/pessoa", method = RequestMethod.GET)

public List<Pessoa> Get() {

return \_pessoaRepository.GetAll();

}

@RequestMapping(value = "/pessoa/{id}", method = RequestMethod.GET)

public ResponseEntity<Pessoa> GetById(@PathVariable(value = "id") int id) {

Pessoa pessoa = \_pessoaRepository.Get(id);

if (pessoa != null)

return new ResponseEntity<Pessoa>(pessoa, HttpStatus.OK);

else

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

@RequestMapping(value = "/pessoa", method = RequestMethod.POST)

public Pessoa Post(@Valid @RequestBody Pessoa pessoa) {

return \_pessoaRepository.Add(pessoa);

}

@RequestMapping(value = "/pessoa/{id}", method = RequestMethod.PUT)

public ResponseEntity<Pessoa> Put(@PathVariable(value = "id") int id, @Valid @RequestBody Pessoa newPessoa) {

Pessoa oldPessoa = \_pessoaRepository.Get(id);

if (oldPessoa != null) {

Pessoa pessoa = oldPessoa;

pessoa.setNome(newPessoa.getNome());

\_pessoaRepository.Edit(pessoa);

return new ResponseEntity<Pessoa>(pessoa, HttpStatus.OK);

} else

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

@RequestMapping(value = "/pessoa/{id}", method = RequestMethod.DELETE)

public ResponseEntity<Object> Delete(@PathVariable(value = "id") int id) {

Pessoa pessoa = \_pessoaRepository.Get(id);

if (pessoa != null) {

\_pessoaRepository.Delete(pessoa.getId());

return new ResponseEntity<>(HttpStatus.OK);

} else

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

Na classe acima, é importante destacar alguns detalhes:

A anotação **@RestController** permite definir um controller com características REST;

A anotação **@Autowired** delega ao Spring Boot a inicialização do objeto;

A anotação **@RequestMapping** permite definir uma rota. Caso não seja informado o método HTTP da rota, ela será definida para todos os métodos.

A anotação **@PathVariable** indica que o valor da variável virá de uma informação da rota;

A anotação **@RequestBody** indica que o valor do objeto virá do corpo da requisição;

E a anotação **@Valid** indica que os dados recebidos devem ser validados.

**//--------------------------------------------**

import br.com.inatel.restexample.Pessoa;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import org.springframework.stereotype.Repository;

@Repository

public class PessoaRepository {

private final static HashMap<Integer, Pessoa> pessoas = new HashMap<>();

public List<Pessoa> GetAll() {

return new ArrayList<Pessoa>(pessoas.values());

}

public Pessoa Get(final int id) {

return pessoas.get(id);

}

public Pessoa Add(final Pessoa pessoa) {

if (pessoa.getId() == 0)

pessoa.setId(generateId(pessoas.size() + 1));

pessoas.put(pessoa.getId(), pessoa);

return pessoa;

}

public void Edit(final Pessoa pessoa) {

pessoas.remove(pessoa.getId());

pessoas.put(pessoa.getId(), pessoa);

}

public void Delete(final int id) {

pessoas.remove(id);

}

private int generateId(final int possible) {

if (pessoas.containsKey(possible))

return generateId(possible + 1);

return possible;

}

}

//----------------------------

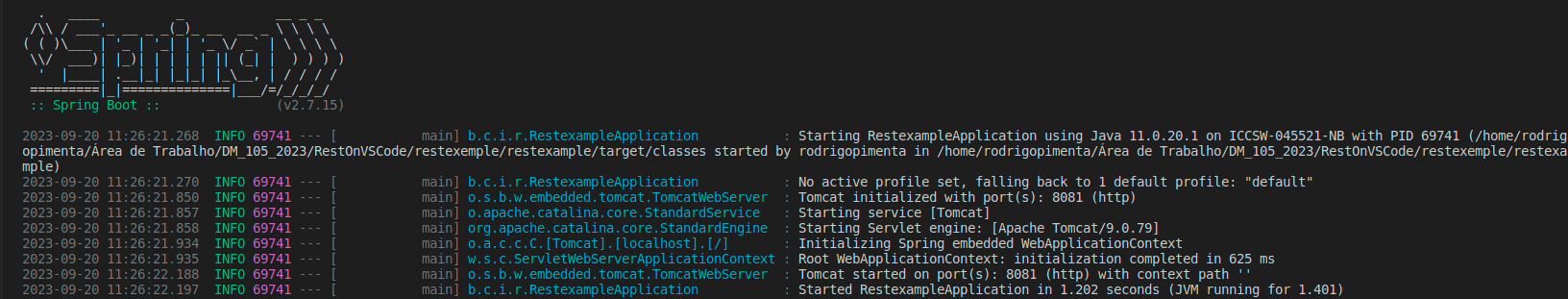
1. Essas classes não usam banco de dados, para facilitar. Elas usam uma tabela hash a conter os dados que serão manipulados em memória principal.
2. No arquivo de properties (...src/main/resources/application.properties), defina:

server.port=8081

1. Agora rode o seu projeto com o seguinte comando:

**./rodaProjeto.sh spring-boot:run**

Correndo tudo bem, será mostrado no log algo como:



Isso significa que o projeto já pode ser testado no SoapUI, por exemplo. Experimente com a URL <http://localhost:8081/pessoa.> Por exemplo, já no navegador, se usar essa URL, um get será invocado no seu web service, mas como ainda não há pessoas cadastradas lá, um array vazio [ ] será retornado. Mas, isso comprovará que o web service está realmente disponível localmente na porta 8081.

1. Para faze testes direto na linha de comando do linux, pode ser feito o seguinte:

curl <http://localhost:8081/pessoa> (comando para enviar o get para o end point)

Para cadastrar um registro pode ser feito o seguinte:

curl -X POST -H "Content-Type: application/json" -d '{"chave1":"valor1", "chave2":"valor2"}' http://localhost:8081/pessoa

Ex:

curl -X POST -H "Content-Type: application/json" -d ‘{ "id": 1, "nome": "Maria" }’ http://localhost:8081/pessoa

P.s.: ao copiar a linha acima e colar no seu terminal, verifique se as áspas simples foram trocadas por outro character. Se sim, corrija.

Outras operações possíeis:

curl <http://localhost:8081/pessoa/3>

curl -X PUT -H "Content-Type: application/json" -d ‘{ "id": 1, "nome": "Maria Modificada" }’ <http://localhost:8081/pessoa/1>

curl -X DELETE -H "Content-Type: application/json" http://localhost:8081/pessoa/3

1. Documentando a API

Usaremos o **springdoc-openapi**.

O web service criado pode atender no seu end point, como visto. Suas capacidades atendem via requisições que ele recebe como requests HTTP, para certos verbos do HTTP. Essas característas podem estar documentadas numa especificação da interface do seviço, para então retratar a API correspondente. Tal retratação poderia ser feita em WADL, contudo o VSCode não tem extensão para a geração automática de arquivos WADL. Nesse caso, uma forma de contornar isso é usar documentação OpenAPI (anteriormente chamada de Swagger). A documentação OpenAPI não é uma tecnologia submetida ao W3C para padronização, mas já se tornou amplamente usada no mundo e agora é popular. Portanto, conhecer a OpenAPI é algo realmente conveniente.

Os arquivos da OpenAPI são geralmente escritos em Json ou em YAML. A seguir está explicado como gerar a interface em OpenAPI com o apoio do VSCode, para o web service criado aqui.

A biblioteca Java springdoc-openapi ajuda a automatizar a geração de documentação de API usando projetos Spring Boot. O springdoc-openapi funciona examinando uma aplicação em tempo de execução para inferir semântica da API com base nas configurações do Spring, na estrutura das classes e em várias anotações.

Automaticamente gera documentação em formato JSON/YAML e HTML para APIs. Esta documentação pode ser complementada por meio de comentários usando as anotações swagger-api.

1. Instale no VSCode a extensão “Spring Initializr Java Support”.

Deve ser adicionado no arquivo pom.xml o seguinte código:

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.6.4</version> <!-- Verifique a versão mais recente no repositório Maven -->

</dependency>

Será usado o Springdoc Openapi UI na versão 1.6.4. Atualmente existe versão mais nova, mas esta escolhida é suficiente para o aprendizado sobre o que se quer ensinar.

1. É necessário configurar o springdoc-openapi-ui na aplicação Spring. Isso pode ser feito adicionando uma classe de configuração. Uma classe deve ser criada chamada **SwaggerConfig** no pacote de configuração:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import org.springdoc.core.SpringDocUtils;

import org.springdoc.ui.SwaggerUiConfigParameters;

import org.springdoc.ui.SwaggerUiConfigProperties;

import org.springdoc.ui.SwaggerUiOAuthProperties;

@Configuration

@EnableWebMvc

public class SwaggerConfig implements WebMvcConfigurer {

@Bean

public SwaggerUiConfigParameters swaggerUiConfigParameters() {

return new SwaggerUiConfigParameters();

}

@Bean

public SwaggerUiConfigProperties swaggerUiConfigProperties() {

return new SwaggerUiConfigProperties();

}

@Bean

public SwaggerUiOAuthProperties swaggerUiOAuthProperties() {

return new SwaggerUiOAuthProperties();

}

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/webjars/\*\*")

.addResourceLocations("classpath:/META-INF/resources/webjars/");

}

static {

SpringDocUtils.getConfig().addAnnotationsToIgnore(

org.springframework.web.bind.annotation.RequestMapping.class

);

}

}

Essa classe pode ser criada na mesma pasta onde estão os outros arquivos Java.

1. Agora deve-se anotar os controladores com as anotações do SpringDoc para documentar as operações. Por exemplo:

import io.swagger.v3.oas.annotations.Operation;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

**@RequestMapping("/api")**

public class MyController {

**@Operation(summary = "Obter algo")**

@RequestMapping(value = "/pessoa", method = RequestMethod.GET)

public ResponseEntity<Pessoa> getSomething() {

// Seu código aqui

}

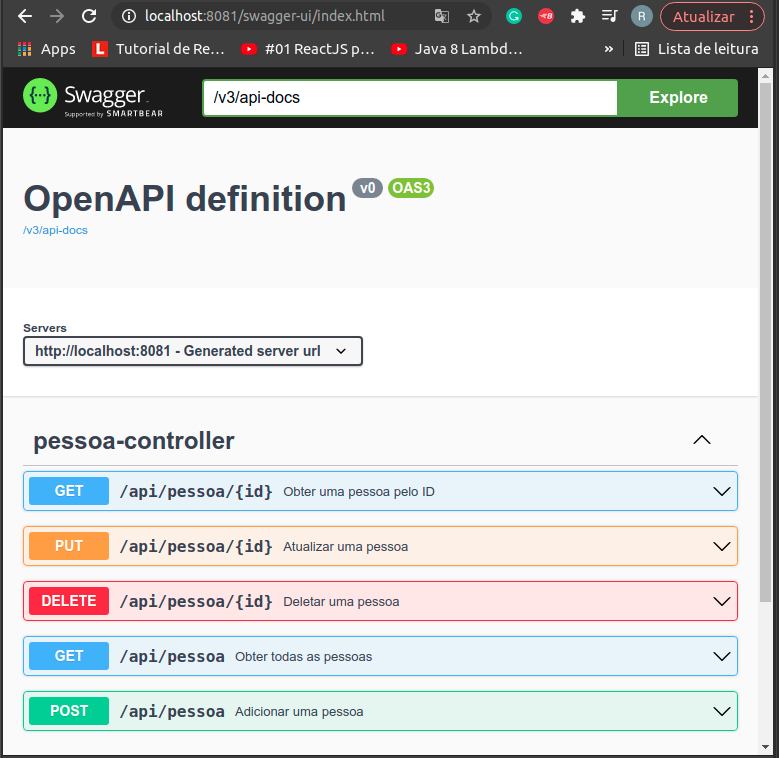
}

1. Execute a aplicação com **./rodaProjeto.sh spring-boot:run** .

1. Acesse no navegador o seguinte endereço:

**http://localhost:8081/swagger-ui.html**

1. Veja e utilize as capacidades que são mostradas na página abaixo. Isso permite testar se o web service está respondendo corretamente, assim como testado antes com o Curl.



1. Agora pode-se ver a API Swagger escrita em Json. Essa API será usada para a facilitação de criação de web services requestors (clientes do web service criado com Rest). E alguma ferramenta de software poderá ser usada para a criação automática das classes para o web service requestor, semelhantemente ao que pode ser feito no Eclipse para o uso de arquivos WSDL e geração de web services com SOAP.

Então, pode-se clicar no link [/v3/api-docs](http://localhost:8081/v3/api-docs" \t "/home/rodrigopimenta/Documentos\\x/_blank) visto na página acima. Isso abre o código Json que é a retratação da API em Swagger, ou OpenAPI. O Json a ser visto está representado abaixo:

{"openapi":"3.0.1","info":{"title":"OpenAPI definition","version":"v0"},"servers":[{"url":**"http://localhost:8081**","description":"Generated server url"}],"paths":{"/api/pessoa/{id}":{"get":{"tags":["pessoa-controller"],"summary":"Obter uma pessoa pelo ID","operationId":"GetById","parameters":[{"name":"id","in":"path","required":true,"schema":{"type":"integer","format":"int32"}}],"responses":{"200":{"description":"OK","content":{"\*/\*":{"schema":{"$ref":"#/components/schemas/Pessoa"}}}}}},"put":{"tags":["pessoa-controller"],"summary":"Atualizar uma pessoa","operationId":"Put","parameters":[{"name":"id","in":"path","required":true,"schema":{"type":"integer","format":"int32"}}],"requestBody":{"content":{"application/json":{"schema":{"$ref":"#/components/schemas/Pessoa"}}},"required":true},"responses":{"200":{"description":"OK","content":{"\*/\*":{"schema":{"$ref":"#/components/schemas/Pessoa"}}}}}},"delete":{"tags":["pessoa-controller"],"summary":"Deletar uma pessoa","operationId":"Delete","parameters":[{"name":"id","in":"path","required":true,"schema":{"type":"integer","format":"int32"}}],"responses":{"200":{"description":"OK","content":{"\*/\*":{"schema":{"type":"object"}}}}}}},"/api/pessoa":{"get":{"tags":["pessoa-controller"],"summary":"Obter todas as pessoas","operationId":"Get","responses":{"200":{"description":"OK","content":{"\*/\*":{"schema":{"type":"array","items":{"$ref":"#/components/schemas/Pessoa"}}}}}}},"post":{"tags":["pessoa-controller"],"summary":"Adicionar uma pessoa","operationId":"Post","requestBody":{"content":{"application/json":{"schema":{"$ref":"#/components/schemas/Pessoa"}}},"required":true},"responses":{"200":{"description":"OK","content":{"\*/\*":{"schema":{"$ref":"#/components/schemas/Pessoa"}}}}}}}},"components":{"schemas":{"Pessoa":{"type":"object","properties":{"id":{"type":"integer","format":"int32"},"nome":{"type":"string"},"idade":{"type":"integer","format":"int32"}}}}}}

**O código acima é semelhante a um WADL.**

É possível reparar no Swagger acima que o end point está em localhost:8081 obviamente.

1. Pacode do projeto

Como dito antes, esse projeto Maven embute em si o web server Tomcat. Portanto, um pacote jar gerado desse projeto terá a capacidade de executar já contendo o Tomcat em funcionamento. Isso facilita migrar esse web service provider para outra máquina, mesmo que lá não haja o Tomcat em execução. Ou seja, esse web service lava consigo tudo o que ele precisa para ficar online e disponível.

Para gerar o pacote jar, deve-se executar o seguinte comando no prompt:

**./rodaProjeto.sh package**

Um pacote será criado, na pasta target, por exemplo com o nome:

**restexample-0.0.1-SNAPSHOT.jar**

Para executar esse pacote na máquina local ou em outra máquina, faz-se:

**java -jar restexample-0.0.1-SNAPSHOT.jar**

Contudo, ao executar em outra máquina, o endpoint será outro. Por exemplo, não será localhost, mas sim o IP da máquina remota.

1. Criação do Web Service Requestor

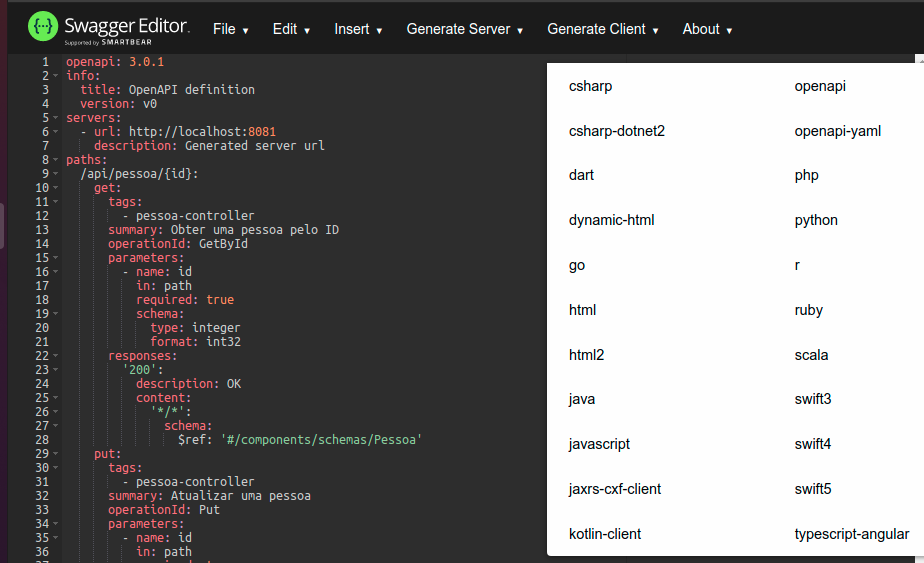
Agora pode-se criar o web service cliente do web service criado acima.

Se a API estiver documentada em um arquivo OpenAPI (anteriormente conhecido como Swagger), pode-se automatizar a geração do código Java para o cliente da API de forma mais fácil usando ferramentas específicas. O OpenAPI é mais adequado para a geração automatizada de clientes e servidores em várias linguagens de programação.

No web site <https://editor.swagger.io/> pode-se citar o documento .json ou .yml e em seguida solcilitar a geração de um SDK client em Java, por exemplo.

1. No web site <https://editor.swagger.io/> deve ser colado todo o código do arquivo .json que já contem a especificação do web service provider criado como mostrado mais acima. O Swagger Editor irá converter o conteúdo json em yml. Permitir isso ou não. É opcional e não altera a geração final do código Java.

1. Escolha a opção de gerar código cliente em Java, como mostrado na imagem abaixo:



1. O conteúdo gerado fica disponível num arquivo .zip. Ao descompactar esse arquivo na máquia local, o código fonte Java para um web service requestor estará disponível.

Vale a pena ler o conteúdo do arquivo README.md nesse momento!

1. Compilar o projeto com ‘mvn clean install’ para checar se está tudo ok.
2. O arquivo README.md sugere que o código abaixo deve ser colocado no arquivo pom.xml do projeto:

<dependency>

<groupId>io.swagger</groupId>

<artifactId>swagger-java-client</artifactId>

<version>1.0.0</version>

<scope>compile</scope>

</dependency>

Mas, na verdade, isso não é necessário.

1. Reparar que o código Java cliente gerado não contem uma classe main. Ou seja. desse ponto em diante pode-se programar a classe que fará uso desse SDK Java client gerado automaticamente. Então uma classe com método main pode ser criada agora para testar o web service provider. Seguem as instruções:
2. Crie a classe Main class na pasta ‘io.swagger.client’. Essa classe pode ter o código abaixo:

package io.swagger.client;

import io.swagger.client.model.\*;

import io.swagger.client.api.PessoaControllerApi;

import java.util.\*;

public class Main {

public static void main(String[] args) {

// Obter a lista de pessoas

PessoaControllerApi apiInstance = new PessoaControllerApi();

Pessoa body = new Pessoa(); // Pessoa |

body.setNome("Aluno do Inatel");

body.setId(11);

body.setIdade(50);

try {

List<Pessoa> result = apiInstance.get();

System.out.println(result);

} catch (ApiException e) {

System.err.println("Exception when calling PessoaControllerApi#get");

e.printStackTrace();

}

try {

apiInstance.post(body);

} catch (ApiException e) {

System.err.println("Exception when calling PessoaControllerApi#post");

e.printStackTrace();

}

try {

List<Pessoa> result = apiInstance.get();

System.out.println(result);

} catch (ApiException e) {

System.err.println("Exception when calling PessoaControllerApi#get");

e.printStackTrace();

}

body = new Pessoa(); // Pessoa |

body.setId(11);

body.setNome("Aluno do Inatel com idade modificada");

body.setIdade(20);

try {

Pessoa result = apiInstance.post(body);

System.out.println(result);

} catch (ApiException e) {

System.err.println("Exception when calling PessoaControllerApi#post");

e.printStackTrace();

}

Integer id = 11; // Integer |

try {

apiInstance.delete(id);

} catch (ApiException e) {

System.err.println("Exception when calling PessoaControllerApi#delete");

e.printStackTrace();

}

try {

List<Pessoa> result = apiInstance.get();

System.out.println(result);

} catch (ApiException e) {

System.err.println("Exception when calling PessoaControllerApi#get");

e.printStackTrace();

}

}

}

1. Execute essa classe como Java a partir do VSCode. Será possível ver que o web service requestor realmente conseguirá consumir capacidades do web service provider. O web serivce provider deverá estar em execução obviamente, para testar a classe acima.
2. Para entender o que faz o código Java gerado, é recomendável debucar o código fonte a partir da classe main acima.

Esse tutorial mostrou então como pode-se criar um web service provider em Rest, ou API Rest. E **o web service provider em rest pode ser levado para uma máquina na nuvem, para executar de lá. Caso isso seja feito, o web service requestor terá que ser modificado, para enviar suas mensagens Rest (HTTP) para a mesma máquina na nuvem. Isso pode ser feito modificando a linha 56 no arquvo ApiClient.java disponível nesse projeto.**

1. Observação importante

Conforme dito em [5], **o desenvolvimento dos web services pode ser via botton up ou top down.**

Pela via botton up, o back end é feito primeiro e depois faz-se o contrato da interface desse web service provider. Como mostrado nesse documento, quando a API foi retratada em Swagger. Nesse caso, o front end pode iniciar seu desenvolvimento mesmo antes do back end ficar todo pronto ou mesmo antes de existir uma API documentada. Isso agiliza o desenvolvimento e facilita a criação da retratação da API em Swagger, por exemplo. Mais adiante, o desenvolvimento do web service requestor (front end) pode usar a API documentada e se adequar a ela, a qualquer detalhe novo, como por exemplo uma modificação de tipo de dados usado, que passa a ser retratado no SWagger.

Pela via top down, o back and e o front end não iniciam o desenvolvimento antes que haja uma API documentada. A documentação nesse caso, em Swagger por exemplo, deve ser feita em acordo com a equipe de front end e back end. Essa é a forma mais elegante para o projeto. Porque assim todas as equipes foram um contrato entre si. Tal contrato é o resultado da análise de todas as equipes e padronizações já podem ser definidas nesse momento, como por exemplo a definição dos tipos de dados, sobre que parâmetro é obrigatório ou opcional, etc. Com a API documentada, as partes iniciam o desenvolvimeto e as equipes de teste já terão com a API o material necessário para inciar a produção dos testes antes mesmo do término do desenvolimento. E com a API pronta, pode-se gerar automaticamente um SDK client Java e um SDK server Java. Como são gerados automaticamente, todo esse código automático não precisa de testes unitários, por exemplo. É normal confiar nas ferramentas que geram o código automaticamente e considerar que tais códigos estão perfeitos. Isso agiliza muito para a equipe de testes.

**Referências bibliográficas**

1. <https://www.treinaweb.com.br/blog/criando-uma-api-rest-com-o-spring-boot> Acessado em 20/09/2023
2. Chat GPT Acessado em 20/09/2023.
3. <https://springdoc.org/#migrating-from-springfox>Acessado em 22/09/2023
4. Swagger (suas APIs documentadas sem esforço) // Dicionário do Programador <https://youtu.be/3nl9AzttzBQ> Acessado em 23/09/2023
5. Webinar: Geração Automática de Código com Swagger Codegen <https://youtu.be/kQ0eiAuFg5Y> Acessado em 24/09/2023

Outubro de 2023.