



# Webservices em Java

Você vai compreender a interface para integração de aplicações (API), web services e as tecnologias SOAP e REST, através do consumo e fornecimento de web services em Java.

Prof.Alexandre de Oliveira Paixão

### Preparação

Para implementar web services em Java, é necessário configurar o ambiente, instalando o Java JDK e ajustando as variáveis de ambiente. A escolha da IDE é flexível, podendo utilizar Apache NetBeans, IntelliJ, Eclipse, entre outras. Os exemplos serão codificados tanto na Spring Tool Suite quanto na Netbeans, utilizando o framework Spring Boot. Este último inclui um servidor web embarcado (Tomcat), eliminando a necessidade de configurar servidores web adicionais, como o GlassFish. A combinação dessas ferramentas proporciona um ambiente eficiente e simplificado para o desenvolvimento de web services em Java.

### Objetivos

- Definir os conceitos de web services.
- Descrever o consumo de web services através do protocolo SOAP em Java.
- Descrever o consumo de web services RESTful em Java.
- Reconhecer o conceito de API e sua implementação com base em web services.

### Introdução

Neste estudo, exploraremos os conceitos de web services, incluindo as abordagens SOAP e REST. Também demonstraremos o uso prático dessas tecnologias, utilizando a linguagem de programação Java.

Para começar, assista ao vídeo!



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

### O que são web services?

Trata-se de um conjunto de padrões que visam garantir a comunicação, integração e interoperabilidade entre serviços – normalmente software –, utilizando protocolos do ambiente web, como SOAP e HTTP.

Assista ao vídeo e conheça os conceitos e a arquitetura dos web services, e compreenda seu papel na comunicação e integração entre aplicações distribuídas. Explore também alguns protocolos importantes, como SOAP e HTTP, e algumas tecnologias, como WSDL e UDDI. Por fim, confira a distinção entre os protocolos SOAP e REST na integração de sistemas.



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

### Como os web services surgiram?

No início da computação distribuída, a comunicação entre cliente e servidor era limitada a uma rede interna: o servidor era responsável por todo o processamento. Com o tempo, o processamento passou a ser distribuído entre vários servidores, e surgiram middlewares para facilitar a comunicação em sistemas distribuídos, como:

- Common Object Request Broker Architecture (CORBA)
- Distributed Component Object Model (DCOM)
- Remote Method Invocation (RMI)

Mais recentemente, as aplicações cliente *versus* servidor migraram para a internet, resultando no surgimento de web services: uma extensão dos conceitos de chamada remota de métodos presentes nos middlewares mencionados, mas adaptados para a web.

Web services são aplicações distribuídas que se comunicam por meio de mensagens, ou seja, são interfaces que descrevem um conjunto de operações acessíveis pela rede através de mensagens. Isso permite que transações e regras de negócio de uma aplicação sejam expostas por meio de protocolos compreensíveis por outras aplicações, independentemente da linguagem de programação ou sistema operacional utilizados.

### Componentes dos web services

A arquitetura dos web services é baseada em três componentes. Vamos conhecê-los!

1

#### Provedor de serviços

É responsável pela criação e descrição do web service – em um formato padrão, compreensível para quem precise utilizar o serviço –, assim como pela sua disponibilização, a fim de que possa ser utilizado.

2

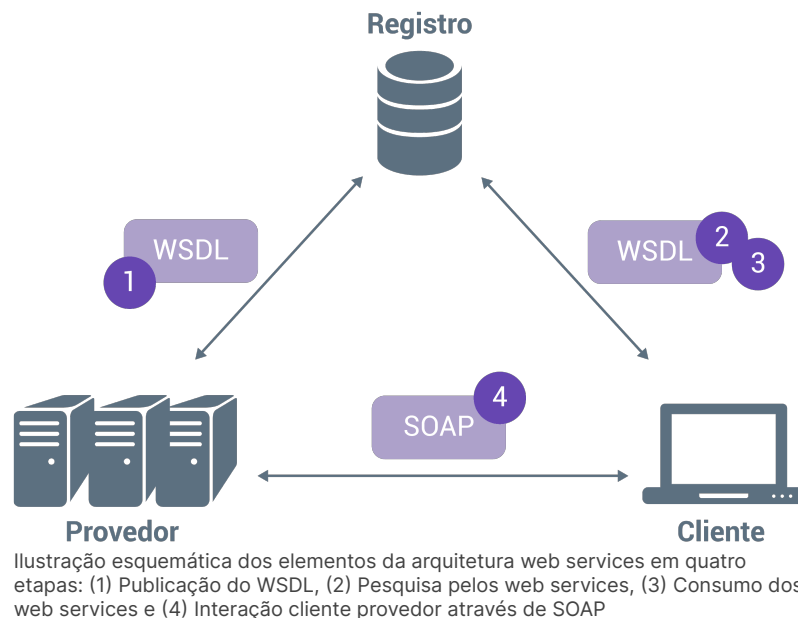
#### Consumidor de serviços

É representado por quem utiliza um web service disponibilizado em um provedor de serviços.

### 3 Registro dos serviços

É um repositório a partir do qual o provedor de serviços pode disponibilizar seus web services e no qual o consumidor de serviços pode utilizá-los. Em termos técnicos, o registro dos serviços contém informações, como os detalhes da empresa, os serviços por ela oferecidos e a descrição técnica dos web services.

Agora, observe a representação do funcionamento desses componentes!



## Outros elementos da arquitetura dos web services

Além dos elementos mencionados, há outros componentes importantes nos web services, como WSDL, SOAP, XML e UDDI. O SOAP será abordado posteriormente junto com o REST.

Agora, vamos conhecer um pouco mais sobre as tecnologias WSDL (*Web services description language*) e UDDI (*Universal description, discovery and integration*). Vamos lá!

### WSDL

É uma linguagem XML que descreve os serviços de um web service de forma automatizada. Permite que os clientes tenham acesso às informações necessárias para utilizar o web service, incluindo as operações disponíveis e suas assinaturas.

### UDDI

Fornece um mecanismo para a descoberta e publicação de web services. Contém informações categorizadas sobre funcionalidades e serviços disponíveis no web service, permitindo a associação de informações técnicas, normalmente definidas usando WSDL.

## SOAP e REST

No início da computação distribuída, tecnologias como RMI, DCOM e CORBA foram usadas para integrar aplicações em redes locais e homogêneas. Com a internet, surgiram outras soluções, como aplicações web escritas em **ASP, PHP e Java** (JSP), que utilizavam **XML** para integração.

Embora o XML fosse um formato padronizado para transmissão de dados, faltava padronização nas empresas em termos de desenvolvimento, protocolos, transações e segurança. Para resolver isso, o W3C desenvolveu o padrão WS-\*, que consiste em especificações para criar web services baseados no protocolo SOAP. O padrão WS-\* inclui várias especificações, como WS-Addressing (mecanismos de transporte para troca de mensagens em web services) e WS-Security (protocolo de segurança para web services).

Já o REST não é uma especificação nem foi criado pelo W3C. É uma arquitetura web alternativa para consumir web services, baseada no uso dos recursos fornecidos pelo HTTP. Agora, vamos explorar mais profundamente essas tecnologias!

## SOAP

*Simple object access protocol* (SOAP) é um protocolo baseado em XML para troca de informações em ambientes distribuídos. Ele encapsula chamadas e retornos de métodos de web services, principalmente usando o protocolo HTTP. Com o uso do SOAP, é possível invocar aplicativos remotamente usando RPC ou troca de mensagens, independentemente do sistema operacional, plataforma ou linguagem de programação envolvida.

## Comunicação em SOAP

Web services que usam o protocolo SOAP podem empregar dois modelos distintos de comunicação. Confira!

### RCP

Neste modelo, é possível modelar chamadas de métodos com parâmetros, assim como receber valores de retorno. Com ele, o corpo (body) da mensagem SOAP contém o nome do método a ser executado e os parâmetros. Já a mensagem de resposta contém um valor de retorno ou falha.

### Document

Neste modelo, o body contém um fragmento XML que é enviado ao serviço requisitado no lugar do conjunto de valores e parâmetros presente no RPC.

## Formato da mensagem

Uma mensagem SOAP é composta por três elementos. Vamos conhecê-los!

### Envelope

Elemento principal (raiz do documento) do XML, responsável por definir o conteúdo da mensagem. É obrigatório.

### Header

Elemento genérico que torna possível a adição de características e informações adicionais à mensagem. É opcional, mas, quando utilizado, deve ser o primeiro elemento do envelope.

### Body

Elemento caracterizado como corpo da mensagem. Contém a informação a ser transportada. Assim como o envelope, é obrigatório.

Observe o fragmento XML, que contém os seguintes elementos:

plain-text

Como vimos no código, o elemento body pode conter um elemento opcional (Fault), que é usado para transportar mensagens de status e/ou erros.

## Exemplo de requisição e resposta utilizando SOAP

Vamos dar um exemplo prático de como uma requisição e uma resposta de um web service são feitas usando o protocolo SOAP.

A partir disso, iremos invocar o método GetModulosTema, que recebe o nome do tema como parâmetro, representado pela variável TemaNome. A resposta será uma lista com os nomes dos módulos relacionados ao tema informado. Veja o XML com o envelope da requisição.

plain-text

Webservices

A seguir, demonstramos o XML do envelope contendo a resposta do método invocado.

plain-text

SOAP e REST

Utilização de SOAP XML em JAVA

Utilização de REST JSON em JAVA

## REST

Proposto pelo cientista de computação Roy Fielding em 2000, o REST utiliza os recursos do protocolo HTTP. É uma arquitetura web mais simples que o SOAP e não é um protocolo em si. É composto pelos seguintes elementos:

- Cliente do web service
- Provedor do web service
- Protocolo HTTP

No consumo de um web service baseado em REST, o ciclo de vida começa quando o cliente envia uma solicitação para o provedor. Após processar a requisição, o provedor responde ao cliente. O HTTP define formato e transporte das mensagens.



### Comentário

Assim como o SOAP possui o WSDL para descrever serviços, o REST possui o WADL (Web application description language) para descrever serviços web ou serviços REST.

## Estrutura dos recursos REST

Na arquitetura REST, os recursos são representados por URIs únicas. Por exemplo, o método GetModulosTema é relacionado ao recurso Tema, cuja URI de consumo seria: `www.dominio.com.br/tema/GetModulosTema/{nome-do-tema}`.

Além disso, podemos ter outros métodos disponíveis para o recurso Tema, como listar todos os temas ou inserir um novo tema. Cada serviço possui sua própria URI.

Na sequência, vejamos alguns exemplos:

- Listagem de todos os temas – `www.dominio.com.br/tema`
- Inserção de tema – `www.dominio.com.br/tema/CreateTema/{nome-do-tema}`

O exemplo de inserção de tema via URL é mais ilustrativo do que prático. Em uma API REST real, é mais comum e recomendado enviar as informações no corpo da requisição em vez de incluí-las na URL.

Os recursos REST seguem a estrutura dos métodos e códigos de retorno HTTP.



### Exemplo

O método GET é utilizado quando você deseja recuperar ou listar recursos, o POST é usado para inclusão, o PUT é empregado para edição e, por fim, o DELETE é utilizado para exclusão.

Além disso, os serviços REST utilizam os códigos de retorno HTTP para indicar o status da operação. Por exemplo, um código 200 indica que o recurso foi atualizado com sucesso, enquanto os códigos 400 e 404 indicam erros.

## Exemplo de requisição e resposta utilizando REST

O consumo de um recurso REST é feito através de uma URL. Normalmente, desenvolvemos um cliente em uma linguagem de programação para acessar o recurso, enviar parâmetros, quando necessário, e tratar a resposta. Vamos usar o mesmo exemplo de listar módulos disponíveis para um tema, como vimos em SOAP.

Para consumir o serviço, utilizaremos a seguinte URL: [www.dominio.com.br/tema/GetModulosTema/Webservices](http://www.dominio.com.br/tema/GetModulosTema/Webservices).

A partir disso, confira um possível retorno para essa requisição.

```
plain-text
{
  "Modulos": [{
    "Nome": "SOAP e REST"
  }, {
    "Nome": "Utilização de SOAP XML em JAVA"
  }, {
    "Nome": "Utilização de REST JSON em JAVA"
  }
]
```

Como vimos, as informações retornadas pelo web service consumido estão no formato JSON. Embora não seja o único tipo de dado disponível para o transporte de informações em REST – ou melhor, em mensagens HTTP –, é o mais utilizado nessa arquitetura.

Como curiosidade, em requisições REST, podemos usar qualquer um destes tipos de conteúdo (Content-Type):

- Application/xml
- Application/json
- Text/plain
- Text/xml
- Text/html

## Atividade 1



Estudamos algumas definições aplicáveis aos web services. Assinale a alternativa que corresponde a uma dessas definições.

A

Os web services são serviços para integração de aplicações que, para se comunicarem, precisam ser escritos em uma mesma linguagem de programação.

B

Embora não seja necessário que estejam codificados na mesma linguagem de programação, é essencial que os web services, como facilitadores da integração entre diversas aplicações, estejam alojados no mesmo servidor web e sistema operacional, a fim de estabelecer uma comunicação eficaz.

C

Os web services são uma solução utilizada na integração e comunicação entre diferentes aplicações.

D

A integração entre aplicações é uma novidade que só se tornou possível com o advento da internet.

E

A partir do advento da internet, a integração através de web services tornou-se restrita ao ambiente da internet, ou seja, não é mais possível integrar aplicações hospedadas em uma mesma rede interna.



A alternativa C está correta.

Os web services são uma solução tecnológica para a integração de diferentes aplicações e que independe de fatores como localização das aplicações – se estão em uma mesma rede ou na internet –, linguagens utilizadas em sua escrita, sistemas operacionais ou servidores web utilizados em sua hospedagem.

## Manipulando requisições e respostas SOAP

Assista ao vídeo e entenda como realizar uma atividade prática para exemplificar como criar uma solicitação e receber uma resposta de um serviço que utiliza o protocolo SOAP.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

### Roteiro de prática

Construiremos agora uma requisição e uma resposta de um serviço que utiliza o protocolo SOAP. Para isso, além do conteúdo visto, você precisará de uma ferramenta que facilite a confecção das requisições. Você pode utilizar um editor de textos como o próprio bloco de notas do Windows, o Nano Editor do Linux ou algo mais avançado, como o software (livre/gratuito) Notepad.

Vamos ao roteiro da atividade!

## Método a ser consumido e retorno

- O serviço que será invocado tem o nome de ConsultaCEP.
- Os parâmetros requeridos por esse serviço são:
  - Credenciais – esse nó é composto por dois parâmetros (usuário e senha), ambos campos do tipo string.
  - CEP – string com o formato 00000-000, onde os zeros deverão ser substituídos por um código de CEP válido.
- A resposta do serviço Consulta CEP deverá estar contida no nó ConsultaCEPResult, composto pelos seguintes elementos:
  - CEP – string contendo o CEP pesquisado.
  - TipoLogradouro – string contendo o tipo do logradouro (Rua|Avenida|etc.).
  - Logradouro – string contendo o descritivo do endereço.
  - Bairro – string.
  - UF – string com dois caracteres contendo a unidade da federação.
  - Estado – string contendo o nome do estado por extenso.
  - Cidade – string contendo o nome da cidade.

## Edição e entregáveis

- Utilize o editor de sua escolha para produzir os arquivos XML referentes à requisição e ao retorno do serviço SOAP descrito.
- Como já mencionado, deverão ser codificados dois arquivos: um contendo a requisição do serviço e outro contendo sua resposta.

A resolução desse serviço pode ter mais de uma resposta. Cabe avaliar a sintaxe dos XMLs produzidos, que deve seguir os critérios mencionados ao longo do conteúdo.

Veja um exemplo de resolução nos códigos a seguir.

## Requisição

Confira um exemplo de resolução nos códigos!

```
plain-text
```

```
string
string

string
```

## Resposta

Agora, observe a resposta nos códigos a seguir.

plain-text

```
string
string
string
string
string
string
string
```

## Faça você mesmo!

A arquitetura REST possui alguns elementos e características. Quanto às requisições, temos a URI do recurso a ser consumido e utilizamos um método/verbo HTTP específico para cada tipo de solicitação realizada ao recurso em questão. Considerando o cenário onde se deseja alterar os dados de uma entidade, assinale a alternativa que representa a solicitação em questão.

A

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"nome_tema":"API REST"}' http://www.dominio.com.br/temas/10
```

B

```
$ curl -v http://www.dominio.com.br/temas/10
```

C

```
$ curl -H "Content-Type: application/json" -X GET -d '{"nome_tema":"API REST"}' http://www.dominio.com.br/temas/10
```

D

```
$ curl -H "Content-Type: application/json" -X POST http://www.dominio.com.br/temas/10
```

E

```
$ curl -X DELETE -H "Content-Type: application/json" -d '{"nome_tema":"API REST"}' http://www.dominio.com.br/temas/10
```



A alternativa D está correta.

Uma requisição REST para alterar uma entidade deve ser feita com o uso do método/verbo PUT. Além disso, na requisição, deve ser enviado, em formato JSON, o dado a ser alterado e seu identificador através da URI. Tal sintaxe pode ser vista na alternativa A. Cada verbo HTTP deve ser utilizado com a devida semântica. Por exemplo, quando queremos requisitar dados, devemos usar o GET. Para persistência/envio

de dados, o POST. A alteração de dados também poderia ser feita com o uso do POST. Entretanto, em termos de semântica, é recomendável priorizar o uso do verbo PUT para essa finalidade.

## Construindo web service SOAP em Java

Descubra no vídeo como criar e consumir um web service SOAP em Java com Spring Boot. Isso inclui codificar o web service e o cliente, e explorar conceitos fundamentais, como fluxo de dados, estrutura da aplicação e configurações essenciais. Isso fornecerá o conhecimento mínimo necessário para trabalhar com web services simples e consumir serviços de terceiros.

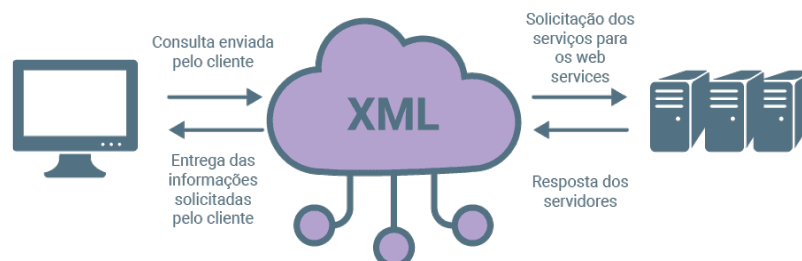


### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Estrutura da aplicação

A troca de dados em um web service é composta por dois elementos principais: a requisição e a resposta do serviço.



Fluxo de requisição e resposta em um web service SOAP

Nossa aplicação consistirá em uma aplicação Java usando o framework Spring Boot. Essa aplicação atuará como provedor de serviços, construindo os recursos que serão disponibilizados via SOAP. Para testar o serviço, usaremos o Postman API Platform.

A estrutura da aplicação, composta por alguns packages, será descrita a seguir. Confira!

1

### Package principal

É o pacote, cujo nome poderá ser definido por você, conterá o script Java da aplicação Spring Boot e será responsável pela inicialização do serviço SOAP. Dentro da estrutura do Spring Boot, tal classe Java recebe a anotação `@SpringBootApplication` e é composta por um método `main`, responsável por inicializar a aplicação.

2

### Package config

É o pacote onde está a classe de configuração, responsável pela definição dos recursos do web service.

### 3 Package endpoint

É o pacote que contém os métodos de cada recurso disponibilizado e as regras de negócio, se houver.

4

### Package entities

É o pacote das entidades. Dentro dos papéis existentes no Spring Boot, as entidades são classes que representam a informação a ser compartilhada na aplicação. Tais classes são compostas pelos atributos de uma entidade e servirão para o mapeamento objeto relacional com o SGBD utilizado no projeto.

5

### Package repositories

É o pacote onde ficam os scripts referentes aos repositórios. Um repository, na arquitetura do Spring Boot, é uma interface (que estende uma interface JPA, contida em tal lib) e tem como finalidade manipular dados no SGBD.

6

### Resources

É a pasta onde contém os arquivos importantes para a aplicação, como o application.properties, para configuração, e o arquivo xsd, que define o esquema XML do recurso do web service SOAP.

## Provedor SOAP

Nosso provedor SOAP será criado com a utilização do framework Spring Boot. Antes de vermos o passo a passo, vamos conhecer um pouco mais sobre esse framework.

## Spring Boot

É um framework open source que simplifica a criação de diferentes tipos de aplicação, como APIs REST, web services SOAP e aplicações web. Ele fornece configurações prontas e permite personalizações, garantindo flexibilidade.

A elaboração de um novo projeto pode ser realizada por meio da Spring Initializr, na qual é possível configurar o gerenciador de dependências, a versão do Java, o empacotamento e escolher as dependências a serem utilizadas.

Agora, veja a configuração inicial do projeto.

The screenshot shows the Spring Initializr interface. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.0.6' is selected. The 'Project Metadata' section has 'Group' as 'com.organization', 'Artifact' as 'spring-soap-ws', 'Name' as 'spring-soap-ws', 'Description' as 'Web Service SOAP com Spring Boot', 'Package name' as 'com.organization.spring-soap-ws', 'Packaging' as 'Jar', and 'Java' version as '17'. On the right, under 'Dependencies', 'Spring Web', 'Spring Web Services', 'Spring Data JPA', and 'H2 Database' are all selected. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE'.

Configurações iniciais do projeto

Essa tela mostra a ferramenta Spring Initializr. Do lado esquerdo, temos as seguintes configurações do projeto.

- Tipo de gerenciador de dependências – no nosso projeto, utilizaremos o Maven.
- Linguagem de programação – usaremos Java.
- Versão do Spring Boot – utilizaremos a 3.0.6.
- Metadados do projeto – o Group e o Artifact podem ser preenchidos conforme suas preferências. Normalmente, no primeiro, preenchemos com o domínio da organização/empresa que está desenvolvendo a aplicação. No segundo, com o nome do projeto em si.
- Empacotamento – usaremos o Jar.
- Versão do Java – usaremos a versão 17.

As demais configurações do projeto, vistas do lado direito, no Spring Initializr, correspondem às dependências a serem utilizadas no projeto. Inicialmente, a partir das opções disponíveis, as seguintes foram selecionadas: Spring Web, Spring Web Services, Spring Data JPA e H2 Database.



#### Dica

Na própria tela de adição de dependências, há um pequeno resumo sobre cada uma das dependências disponíveis.

Outras dependências serão necessárias no projeto, mas falaremos sobre isso mais adiante. Por ora, cabe ressaltar que as duas últimas dependências da lista não são obrigatórias. Vamos utilizá-las para armazenarmos/recuperarmos os dados que serão disponibilizados pelo provedor SOAP a partir do banco de dados – nesse caso, um banco de dados embarcado, o H2.

## Banco de dados embarcado

O H2 é um banco de dados embarcado que armazena os dados diretamente no sistema de arquivos. É ideal para aplicações de demonstração. Em nossa aplicação, a estrutura do banco de dados será criada a partir das entidades, incluindo o próprio banco de dados, tabela e colunas.

Para realizar uma carga inicial de dados, utilizamos o recurso do Spring chamado data.sql (armazenado na pasta resources), onde inserimos as instruções SQL para inclusão de dados. A seguir, veja o exemplo do arquivo data.sql utilizado em nossa aplicação.

sql

```
INSERT INTO country (name, capital, currency, population) VALUES ('Spain', 'Madrid', 'EUR', 46704314);
INSERT INTO country (name, capital, currency, population) VALUES ('Poland', 'Warsaw', 'PLN', 38186860);
INSERT INTO country (name, capital, currency, population) VALUES ('United Kingdom', 'London', 'GBP', 63705000);
```

Para configurar o banco de dados embarcado — nome do banco, usuário e senha —, assim como executar o script SQL durante a inicialização da aplicação e após a criação da estrutura do banco, incluímos as seguintes propriedades no arquivo `application.properties`:

plain-text

```
#Configuracoes de acesso ao banco embarcado
spring.datasource.url=jdbc:h2:file:/data/webservice
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
#spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.ddl-auto=create-drop

#Faz com que o hibernate crie as tabelas no BD antes de inserir os dados (data.sql)
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

## Outras dependências

Além das dependências já mencionadas, mais uma será necessária em nosso projeto. O arquivo `pom.xml` é o local onde gerenciamos as dependências do projeto – em projetos que utilizam o Maven. Tal arquivo possui uma série de seções, entre elas a `dependencies`. Dentro desse nó, as seguintes linhas deverão ser adicionadas.

plain-text

```
wsdl4j
wsdl4j
```

## Arquivo XSD

Existem duas formas de criar um web service: **Contract-Last** e **Contract-First**. Na abordagem **Contract-Last**, o contrato do web service (WSDL) é criado a partir das classes Java. Já na abordagem **Contract-First**, o código Java é gerado a partir de um contrato WSDL previamente criado.

No nosso projeto, usando o Spring, adotaremos a abordagem **Contract-First**. O arquivo XSD contém o domínio do nosso serviço, incluindo métodos e parâmetros. Com base no XSD, o Spring gerará o contrato WSDL. Após isso, observe o conteúdo do XSD do projeto.



plain-text

Perceba que no XSD está definido um método apenas: **getCountryRequest**, que recebe um parâmetro, do tipo string, contendo o nome do país a ser pesquisado. Há ainda o elemento referente à resposta do serviço: **getCountryResponse**. Em sua definição, consta que o retorno do método será composto por um objeto do tipo country, o qual também é definido posteriormente no próprio arquivo.

## Classes de domínio

As classes Java do domínio da aplicação são geradas automaticamente pelo Spring durante o processo de compilação do projeto, utilizando o plugin jaxb2. Para habilitar esse processo, é necessário adicionar algumas configurações no arquivo pom.xml, na seção build e na subseção plugins. Essas configurações especificam o diretório onde o esquema (arquivo XSD) está armazenado e o local onde as classes Java serão geradas.

As linhas que devem ser adicionadas são as seguintes.

java

```
org.codehaus.mojo
jaxb2-maven-plugin
3.1.0

xjc

xjc

com.groupid.wsprovider.schemas
src/main/java

src/main/resources/countries.xsd

false
false
```

Agora, confira a função de algumas configurações contidas nesse fragmento de código.

<packageName>

Nome do pacote, dentro do seu projeto, onde as classes geradas serão armazenadas.

<source>

Diretório e nome do arquivo xsd que será utilizado para geração das classes Java.

A tarefa de gerar o código é realizada durante o build da aplicação. Entretanto, também é possível realizar tal processo utilizando o Maven. Para isso, basta executar o seguinte comando, a partir da raiz do projeto (mesmo nível de onde está o pom.xml).

plain-text

```
mvn clean install (Sistemas *nix)
mvnw clean install (Sistema Windows)
```

## Entidades

Uma entidade em projetos Spring Boot é responsável pelo mapeamento objeto-relacional com o banco de dados utilizado na aplicação. No nosso projeto, utilizaremos um banco de dados embarcado para dar um caráter mais real ao projeto, mas é possível substituí-lo por qualquer SGBD de preferência. A partir da entidade, toda a estrutura do banco de dados será criada, incluindo tabelas e colunas.

Como nosso projeto possui apenas um método que retorna os dados de um país, teremos apenas uma entidade, chamada **CountryEntity**. A seguir, vamos observar o código dessa classe!

java

```
@Entity
@Table(name = "country")
public class CountryEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_country")
    private Integer idCountry;

    @Column(name = "name")
    private String name;

    @Column(name = "capital")
    private String capital;

    @Column(name = "currency")
    private String currency;

    @Column(name = "population")
    private Integer population;

    public Integer getIdCountry() {
        return idCountry;
    }

    public void setIdCountry(Integer idCountry) {
        this.idCountry = idCountry;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCapital() {
        return capital;
    }

    public void setCapital(String capital) {
        this.capital = capital;
    }

    public String getCurrency() {
        return currency;
    }

    public void setCurrency(String currency) {
        this.currency = currency;
    }

    public Integer getPopulation() {
        return population;
    }

    public void setPopulation(Integer population) {
        this.population = population;
    }
}
```

Nesse código, observamos a utilização de algumas anotações. Confira!

### @Entity

Define a função da classe.

### @Table

É através dela que podemos definir o nome da tabela a ser criada no banco.

### @Column

É utilizada em atributos para definição dos nomes das colunas no bd.

Além disso, o atributo **idCountry** tem duas outras anotações especiais: a **@Id**, que lhe confere o papel de chave primária, e a **@GeneratedValue**, que define de quem será a responsabilidade pela geração dos valores do atributo em questão (em nosso caso, essa anotação foi configurada para deixar tal responsabilidade com o SGBD).

## Repositório

Após criar a entidade, responsável pelo mapeamento objeto-relacional com o banco de dados, precisamos de mecanismos que nos permitam realizar operações no banco de dados em questão. Tal papel é feito pela interface de repositório, que estende as funcionalidades da interface `JpaRepository`, tendo acesso, assim, a inúmeros métodos de recuperação e persistência de dados.

Na sequência, veja o código de nossa interface `CountryRepository`!

```
java

public interface CountryRepository extends JpaRepository {
    Optional findByName(String name);
}
```

Nessa interface, declaramos, usando as facilidades da `JpaRepository`, um método que nos permite realizar buscas pelos atributos de nossa entidade. Tal método recebe a string contendo o nome do país a ser buscado e, encontrando-o, devolve tal dado como instância da entidade `CountryEntity`.

## Endpoint

É responsável por responder a todas as requisições feitas para o serviço no provedor de web service. Ele recebe a requisição, interage com a aplicação, coleta/processa os dados e os retorna para o cliente.

No nosso projeto, o endpoint chamado `CountryEndpoint` possui um método para receber a requisição de busca pelo nome do país. Ele aciona o repositório para buscar os dados no banco de dados e os converte em uma instância da classe `Country`, que é uma das classes geradas pelo `jaxb2`. Em seguida, os dados são retornados para o cliente.

Agora, observe o código do endpoint!

```

java
@Endpoint
public class CountryEndpoint {
    private static final String NAMESPACE_URI = "http://groupid.com/wsprovider";

    private CountryRepository countryRepository;

    @Autowired
    public CountryEndpoint(CountryRepository countryRepository) {
        this.countryRepository = countryRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getCountryRequest")
    @ResponsePayload
    public GetCountryResponse getCountry(@RequestPayload GetCountryRequest request) {
        Map countryMap = new HashMap<>();
        Country country = new Country();

        GetCountryResponse response = new GetCountryResponse();
        CountryEntity countryEntity =
countryRepository.findByName(request.getName()).get();

        country.setName(countryEntity.getName());
        country.setCapital(countryEntity.getCapital());
        country.setCurrency(countryEntity.getCurrency());
        country.setPopulation(countryEntity.getPopulation());
        countryMap.put(country.getName(), country);

        response.setCountry(countryMap.get(request.getName()));

        return response;
    }
}

```

## Classe de configuração do WS

Para que nosso provedor de web service seja capaz de receber as requisições, direcioná-las para o Endpoint e retornar os resultados, é necessário criar uma classe de configuração. Essa classe, que estende a classe `WsConfigureAdapter`, utiliza o modelo de programação orientado a anotação do Spring WS.

Dentro dessa classe, temos o método `"messageDispatcherServlet"`, responsável por receber as requisições SOAP. Além disso, existem outros Beans na classe responsáveis por expor um esquema WSDL padrão. Aqui está a estrutura da nossa classe de configuração, veja!

```

java
@EnableWs
@Configuration
public class WebServiceConfig extends WsConfigurerAdapter {
    @Bean
    public ServletRegistrationBean messageDispatcherServlet(ApplicationContext
applicationContext) {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(true);
        return new ServletRegistrationBean<>(servlet, "/ws/*");
    }

    @Bean(name = "countries")
    public DefaultWsd11Definition defaultWsd11Definition(XsdSchema countriesSchema)
{
        DefaultWsd11Definition wsdl11Definition = new DefaultWsd11Definition();
        wsdl11Definition.setPortTypeName("CountriesPort");
        wsdl11Definition.setLocationUri("/ws");
        wsdl11Definition.setTargetNamespace("http://groupid.com/wsprovider");
        wsdl11Definition.setSchema(countriesSchema);
        return wsdl11Definition;
    }

    @Bean
    public XsdSchema countriesSchema() {
        return new SimpleXsdSchema(new ClassPathResource("countries.xsd"));
    }
}

```

## Atividade 1

A classe contém dados vistos/definidos em outros locais de nossa aplicação, como o namespace e o arquivo xsd, além das informações necessárias para o funcionamento da aplicação. Como vimos, há duas abordagens para a criação de um web service: a Contract-Last e a Contract-First. Considerando a Contract-First, é correto afirmar que:

A

nessa abordagem, todo o código Java é codificado e, a partir dele, é gerado o WSDL.

B

essa abordagem tem como característica a definição de endpoints REST para o fornecimento de serviços.

C

nessa abordagem, definimos inicialmente um arquivo de Schema, o XSD, a partir do qual as classes de domínio Java são geradas.

D

essa abordagem não é suportada pela linguagem Java, onde apenas a Contract-Last está disponível.

E

nessa abordagem, os códigos da aplicação (classes de domínio, entidades, repositórios, endpoints etc.) são todos gerados automaticamente a partir do Schema XSD.



A alternativa C está correta.

As duas abordagens de criação de web-service, Contract-Last e Contract-First, possuem uma importante diferença: enquanto na primeira, codificamos nossas classes Java e, a partir dela, geramos o WSDL; na segunda, precisamos gerar o XSD e, a partir dele, as classes Java são geradas. Embora com a linguagem Java possamos construir web services com ambas as abordagens, o framework Spring WS só dá suporte à Contract-First.

## Testando nosso web service com o Postman

Agora, vamos falar sobre a estrutura final da aplicação de um web service, apresentando pacotes e classes envolvidos, bem como as configurações necessárias.

Assista ao vídeo e aprenda a configurar e executar requisições para testar seu web service utilizando o Postman. Além disso, veja como criar um workspace, configurar os dados da requisição e analisar os resultados obtidos.


































Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Estrutura final da aplicação

Ao final da codificação, temos alguns pacotes e classes Java codificadas, responsáveis por prover os serviços do web service. A estrutura da aplicação pode ser visualizada na IDE (Spring Tool Suite – STS). Vamos conferir!

- ▼  spring-soap-ws [boot]
  - ▼  src/main/java
    - ▼  com.groupid.wsprovider
      - >  ProducingWebServiceApplication.java
    - ▼  com.groupid.wsprovider.config
      - >  WebServiceConfig.java
    - ▼  com.groupid.wsprovider.endpoints
      - >  CountryEndpoint.java
    - ▼  com.groupid.wsprovider.entities
      - >  CountryEntity.java
    - ▼  com.groupid.wsprovider.repositories
      - >  CountryRepository.java
    - ▼  com.groupid.wsprovider.schemas
      - >  Country.java
      - >  GetCountryRequest.java
      - >  GetCountryResponse.java
      - >  ObjectFactory.java
      - >  package-info.java
    - >  META-INF
  - ▼  src/main/resources
    -  application.properties
    -  countries.xsd
    -  data.sql
  - >  JRE System Library [JavaSE-17]
  - >  Maven Dependencies
  - >  src
  - >  target
  -  mvnw
  -  mvnw.cmd
  -  pom.xml
  -  request.xml

Estrutura da aplicação na IDE Spring Tool Suite

---

Na sequência, vamos entender alguns itens dessa estrutura.



1 Com.groupid.wsprovider  
Package principal da aplicação.

- **ProducingWebServiceApplication.java:** Classe de inicialização da aplicação no framework Spring Boot. Possui a anotação `SpringBootApplication`, que lhe confere tal papel.

2  
Com.groupid.wsprovider.config  
Package que contém a classe de configuração (e outras, quando necessário) da aplicação.

- **WebServiceConfig.java:** Essa classe contém a configuração do serviço disponibilizado, como seu nome, URI, Namespace e Schema. Além disso, essa classe recebe a anotação `@Configuration`, que lhe concede, dentro do Spring Boot, o papel de classe responsável pelas configurações necessárias para o funcionamento da aplicação e que serão lidas durante a inicialização do serviço.

3  
Com.groupid.wsprovider.endpoints  
Package que contém a definição e regras de negócio dos recursos disponibilizados no WS.

- **CountryEndpoint.java:** Essa classe recebe a anotação `@Endpoint` e é responsável pelo processamento das requisições recebidas pelo web service. Nela é definido cada recurso disponibilizado e suas regras de negócio.

4  
Com.groupid.wsprovider.entities  
Package que armazena as entidades do projeto, responsáveis pelo mapeamento objeto-relacional.

- **CountryEntity.java:** Essa classe contém a estrutura – atributos e métodos que representam a entidade respectiva no banco de dados (tabela e colunas). É usada para transferência dos dados na aplicação.

5  
Com.groupid.wsprovider.repositories  
Package que armazena as interfaces que fornecem métodos de acesso ao banco de dados.

- **CountryRepository.java:** Essa interface estende a interface `JpaRepository` e é responsável pela disponibilização de métodos de acesso ao banco de dados, referentes à Entidade Country/Tabela country.

## 6 Com.groupid.wsprovider.schemas

**Package que contém as classes de domínio e outros arquivos gerados a partir do xsd.**

Essas classes são responsáveis pela representação dos dados, métodos, requisição e resposta no formato **XML** – formato usado para a transmissão de dados em web services SOAP. Veja!

- Contry.java
- GetCountryRequest.java
- GetCountryResponse.java
- ObjecFactory.java
- Package-info.java

## 7

### Src/main/resources

**Package resources é aquela pasta importante que armazena os arquivos o xsd e o sql.**

Nessa pasta são armazenados o xsd, o arquivo de configurações do projeto – application.properties – e o sql responsável pela inserção inicial de dados em nosso database. Veja!

- o **application.properties**: Nesse arquivo, são definidas várias propriedades do projeto, como os dados de acesso ao SGBD, controle de logs etc.
- o **countries.xsd**: Schema no formato XML, responsável pela definição do domínio – métodos e parâmetros – de nosso serviço.
- o **data.sql**: Script SQL contendo queries de inserção de dados. Tal script, conforme configuração realizada no application.properties, será executado na inicialização do web service.

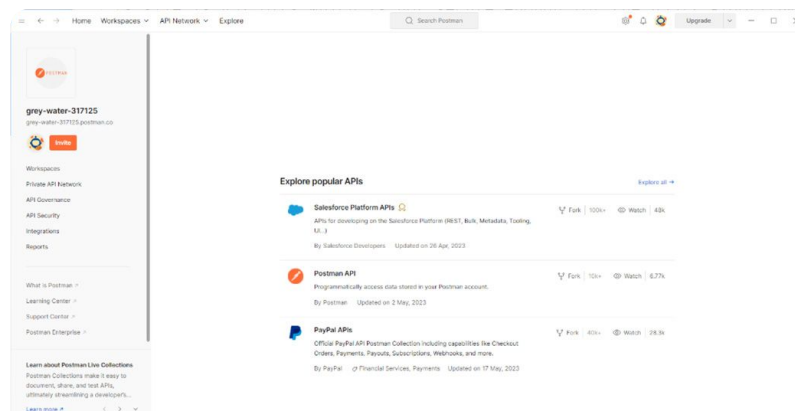
O arquivo pom.xml na raiz do projeto é usado pelo Maven para gerenciar dependências e configurações.

## Postman

Para testar o provedor de serviços, podemos utilizar o Postman, uma ferramenta popular para testes de APIs e web services.

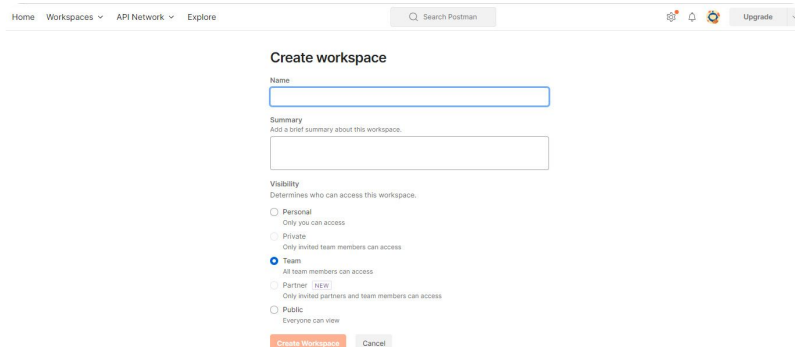
## Visão inicial

Após a instalação e execução do Postman, você verá uma tela inicial semelhante a esta:



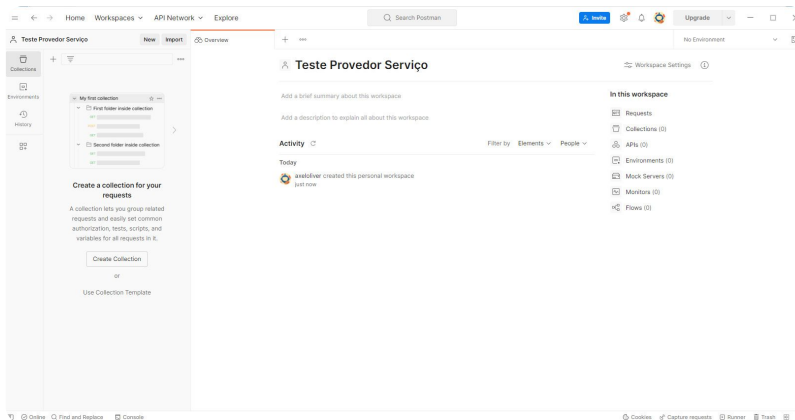
## Workspaces

O Postman organiza projetos em Workspaces. Para criar um Workspace, clique no respectivo menu e em Create Workspace.



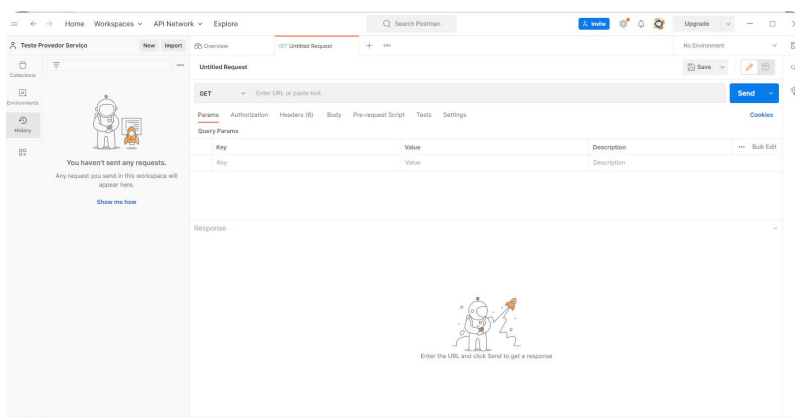
The screenshot shows the 'Create workspace' dialog box in Postman. It has a title bar with 'Create workspace'. Below the title, there are three sections: 'Name' with a text input field, 'Summary' with a text area, and 'Visibility' with radio button options. The 'Visibility' section includes descriptions for each option: 'Personal' (Only you can access), 'Private' (Only invited team members can access), 'Team' (All team members can access, selected), 'Partner (NEW)' (Only invited partners and team members can access), and 'Public' (Everyone can view). At the bottom, there are two buttons: 'Create Workspace' and 'Cancel'.

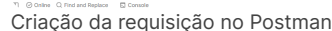
Defina o nome do Workspace e uma breve descrição, caso queira. Na seção visibilidade, selecione a opção de sua preferência – ao lado de cada uma delas, há uma explicação. Ao final, após clicar no botão para criação do espaço de trabalho, você será levado para sua página principal. Confira!

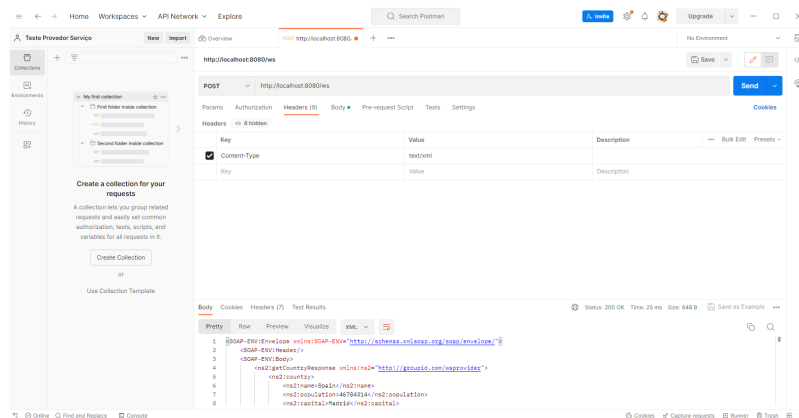


## Criando a primeira requisição

No Workspace, clique em Requests e, depois, no sinal de "+" para criar uma requisição. Será exibida a tela de configuração dos dados da requisição (do request).



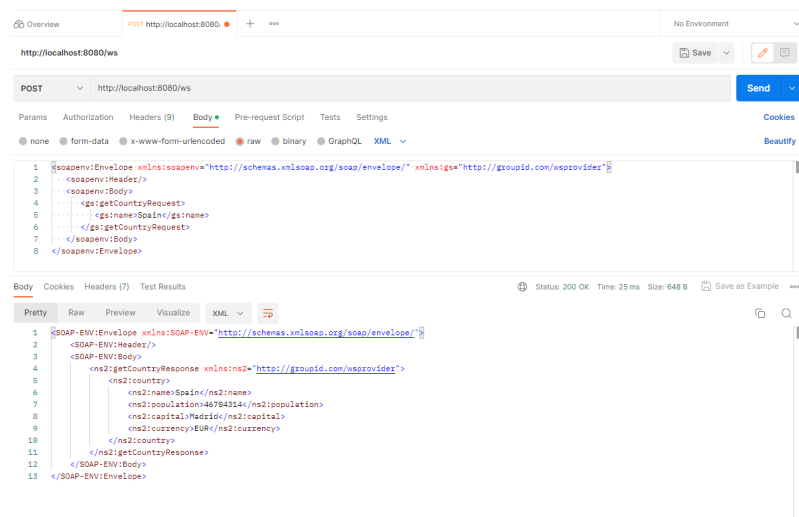




Criação da requisição no Postman

## Executando a requisição

Tendo configurado a requisição, chegou a hora de executá-la. Certifique-se de ter iniciado a aplicação a partir da IDE (e que ela subiu sem nenhum erro). A seguir, no Postman, clique no botão Send. Após o processamento da requisição, na aba Response, será exibido seu resultado.



Execução da requisição no Postman

Veja que foram retornados os dados do país buscado, em nosso teste, Spain. Tal parâmetro foi passado no body da requisição. Os dados foram recuperados do banco de dados, onde foram previamente inseridas durante a inicialização da aplicação, conforme querye inserida no data.sql.

Faça outros testes, como a inclusão de outro país existente no bd ou até mesmo um não existente. Ao final, você pode salvar a requisição no Postman, clicando no botão Save.

## Atividade 2

Qual dos componentes do provedor de web service é responsável pelo processamento das requisições recebidas pelo web service e define os recursos disponibilizados e suas regras de negócio?

A

com.groupid.wsprovider.config.WebServiceConfig.java

B

com.groupid.wsprovider.endpoints.CountryEndpoint.java

C

com.groupid.wsprovider.entities.CountryEntity.java

D

com.groupid.wsprovider.repositories.CountryRepository.java

E

com.groupid.wsprovider.schemas.Contry.java



A alternativa B está correta.

A classe CountryEndpoint.java, localizada no pacote com.groupid.wsprovider.endpoints, é responsável pelo processamento das requisições recebidas pelo web service. Nela são definidos cada recurso disponibilizado e suas regras de negócio. As demais alternativas referem-se a outros componentes do provedor de web service, como configuração (A), entidades (C), repositórios (D) e classes de domínio (E), mas não estão diretamente relacionadas ao processamento das requisições.

## Adicionando novos recursos ao provedor de serviços

Após construir a primeira versão de nosso provedor de serviços, onde um único recurso foi disponibilizado, vamos adicionar um novo serviço. Nessa atividade, você utilizará os conceitos estudados para adicionar um endpoint que retorne a lista de todos os países cadastrados em nosso banco de dados.

Descubra como integrar novos recursos ao provedor de serviços, expandindo as funcionalidades do seu web service em um guia passo a passo.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Roteiro de prática

Nessa atividade, utilizaremos a aplicação codificada anteriormente. A partir de algumas classes e interfaces já criadas, você deverá inserir novos códigos com o objetivo de adicionar mais um recurso ao nosso provedor getAllCountries. Tal serviço deverá ser criado conforme roteiro a seguir. Como ferramenta, utilize a IDE e o código desenvolvido até aqui.

Agora, vamos ao passo a passo!

- Altere o arquivo countries.xsd adicionando o Request e o Response do novo serviço.
- Em relação ao Request, você pode usar como base o existente. Entretanto, esse novo não receberá parâmetros. Logo, copie a estrutura do anterior e retire a tag .

- Certifique-se de que o conteúdo do package com.groupid.wsprovider.schemas foi atualizado com a inclusão do novo Request e também do Response.
- Altere o CountryEndpoint, criando o nome método, responsável por processar o Request criado. Lembre-se de modificar o método do repositório para o findAll, que retornará uma lista da Entidade CountryEntity.
- A partir da lista de CountryEntity obtida do repositório, percorra seus valores e crie uma lista da classe de domínio Country.
- Crie uma requisição nos Postman para testar as alterações realizadas.

Como resposta, você deverá ver algo semelhante a isto:

plain-text

Spain  
46704314  
Madrid  
EUR

Poland  
38186860  
Warsaw  
PLN

United Kingdom  
63705000  
London  
GBP

A resolução desse serviço pode ter mais de uma resposta. Cabe avaliar o resultado do teste do serviço no Postman, onde uma lista de países, e seus dados, deverá ser retornada – conforme exemplo anterior.

Observe um exemplo de resolução nos códigos a seguir.

## Countries.xsd

Agora, vejamos um exemplo de countries.xsd.

plain-text

## CountryEndpoint.java

Agora, vejamos um exemplo de código java.



```

java
@Endpoint
public class CountryEndpoint {
    private static final String NAMESPACE_URI = "http://groupid.com/wsprovider";

    private CountryRepository countryRepository;

    @Autowired
    public CountryEndpoint(CountryRepository countryRepository) {
        this.countryRepository = countryRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getCountryRequest")
    @ResponsePayload
    public GetCountryResponse getCountry(@RequestPayload GetCountryRequest request) {
        Map countryMap = new HashMap<>();
        Country country = new Country();

        GetCountryResponse response = new GetCountryResponse();
        CountryEntity countryEntity =
countryRepository.findByName(request.getName()).get();

        country.setName(countryEntity.getName());
        country.setCapital(countryEntity.getCapital());
        country.setCurrency(countryEntity.getCurrency());
        country.setPopulation(countryEntity.getPopulation());
        countryMap.put(country.getName(), country);

        response.setCountry(countryMap.get(request.getName()));

        return response;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getAllCountriesRequest")
    @ResponsePayload
    public GetAllCountriesResponse getCountries(@RequestPayload
GetAllCountriesRequest request) {
        List countries = new ArrayList<>();

        GetAllCountriesResponse response = new GetAllCountriesResponse();
        List listCountryEntity = countryRepository.findAll();

        for(CountryEntity countryEntity:listCountryEntity) {
            Country country = new Country();
            country.setName(countryEntity.getName());
            country.setCapital(countryEntity.getCapital());
            country.setCurrency(countryEntity.getCurrency());
            country.setPopulation(countryEntity.getPopulation());
            countries.add(country);
        }

        response.getCountry().addAll(countries);

        return response;
    }
}

```

Faça você mesmo!

Na geração das classes de domínio da atividade anterior, foram geradas duas novas classes Java: uma referente ao request e outra à response do serviço adicionado (um método para listar todos os países cadastrados) ao provedor.

Observando a classe Response existente anteriormente, é possível perceber um atributo do tipo Country e métodos de acesso a ele: `getCountry` e `setCountry`. Entretanto, é possível que, durante a geração das classes de domínio, não seja criado o método `set` para o Response, mas apenas o método `get`. Nesse contexto, considere o seguinte código:

java

```
/**  
*
```

Classe Java de anonymous complex type.

```
*  
*
```

O seguinte fragmento do esquema especifica o conteúdo esperado contido dentro desta classe.

```
*  
*
```

```
*  
*  
*/  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlType(name = "", propOrder = {  
    "country"  
})  
@XmlRootElement(name = "getAllCountriesResponse")  
public class GetAllCountriesResponse {
```

```
    @XmlElement(required = true)  
    protected List country;
```

```
/**  
 * Gets the value of the country property.  
 *  
 *
```

```
* This accessor method returns a reference to the live list,  
 * not a snapshot. Therefore any modification you make to the  
 * returned list will be present inside the Jakarta XML Binding object.  
 * This is why there is not a set method for the country property.  
 *  
 *
```

```
* For example, to add a new item, do as follows:  
*
```

```
*  
*  
*
```

```
* Objects of the following type(s) are allowed in the list  
 * {@link Country }  
 *  
 *  
 */
```

```
public List getCountry() {  
    if (country == null) {  
        country = new ArrayList();  
    }  
}
```

Para recuperar a lista de objetos do tipo Country, você poderia utilizar o método `getCountry`. Para adicionar uma lista de objetos do mesmo tipo, qual das alternativas poderia ser utilizada?

A

A classe `Response` pode ser editada diretamente e, com isso, ser adicionado o método `setCountry`, que receberia uma listagem do objeto em questão. Além disso, a partir da edição diretamente da classe `Response` gerada a partir do XSD, a estrutura do XSD também será atualizada para refletir tal mudança.

B

A partir do método `getCountry()`, pode ser utilizado, para a finalidade em questão, o método `push()`, ao qual seria passado a lista de objetos `Country`.

C

Por ser uma classe de `Response`, ou seja, usada para tratar o resultado de uma requisição, não é possível incluir métodos de adição de dados.

D

A partir do método `getCountry()`, pode ser utilizado o método `addAll()`, que recebe uma lista completa contendo objetos do tipo `Country`.

E

Um novo método chamado `setListCountry` deverá ser criado no `.xsd`. Tal método receberia como parâmetro uma lista de objetos do tipo `Country` e, após geração das classes de domínio, ficaria disponível na classe `Response` para ser utilizado.



A alternativa D está correta.

A partir do `xsd`, nossas classes de domínio são criadas em nossa aplicação. Após criadas por esse processo, tais classes não devem ser editadas diretamente, pois perderíamos suas alterações a cada nova edição do `xsd` e respectivas atualizações nas classes de domínio.

Em contrapartida, ao analisar o código apresentado, podemos ver, no comentário automaticamente gerado para o método `getCountry`, que há ao menos um método disponível para a adição de itens: `add`. Tal método, como acontece com as listas, adiciona apenas um novo elemento a uma lista. Além dele, há ainda o `addAll`, que adiciona todos os elementos de uma lista à outra lista. Tal método seria o indicado para a finalidade pretendida.

## API Rest com Spring Boot

O Spring Boot é um framework Java que fornece uma série de componentes que facilitam o desenvolvimento de aplicações. Entre os tipos de aplicação que podemos criar com tal ferramenta, temos as API Rest.

Aprenda no vídeo a configurar um projeto usando o Spring Boot para disponibilizar serviços REST. Explore a configuração, as camadas, o consumo de APIs e o banco de dados H2, ajudando a aprimorar suas habilidades em desenvolvimento REST com Java.



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

### Estrutura da aplicação

Nossa API REST será construída com a utilização do framework Spring Boot. Através dela, disponibilizaremos um serviço de recuperação de dados de países a partir da pesquisa pelo seu nome. Seguindo a separação em módulos proposta pelo framework, a estrutura de nossa API terá a seguinte organização. Confira!

1

#### Camada controller

Onde são expostos os endpoints disponíveis em nossa API. É a camada de relacionamento com os clientes que realizam requisições para nossa aplicação.

2

#### Camada service

Onde ficam os métodos responsáveis pelas regras de negócio e pela interação entre as camadas controller e repository.

3

#### Camada repository

Contém os métodos de acesso ao banco de dados, os quais são estendidos a partir da JpaRepository, que fornece diversas funções de recuperação e persistência de dados.

4

#### Camada entity

Onde temos as entidades de nossa aplicação, que têm o papel de representar, em código Java, a estrutura existente no banco de dados associado à nossa API. Para isso, elas implementam os conceitos de Mapeamento Objeto Relacional, refletindo através de código Java as tabelas, colunas e relacionamentos existentes no database.

### Estrutura do banco de dados

Em nosso projeto, utilizaremos um banco de dados embarcado: H2. Para isso, basta inserirmos a respectiva dependência em nosso projeto, além de configurarmos os dados de acesso no arquivo de propriedades. A seguir, confira a dependência e as propriedades mencionadas.

## Dependência

Vamos conferir o código relacionado a esse componente específico.

java

```
com.h2database
h2
runtime
```

## Configuração

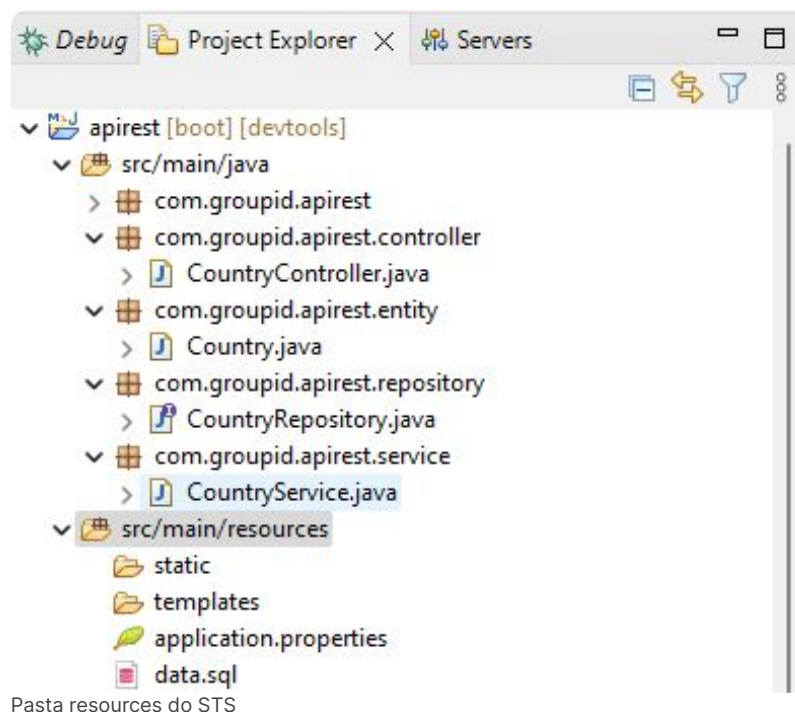
Vamos explorar o código relacionado a esse componente específico.

java

```
spring.datasource.url=jdbc:h2:file:/data/apirest
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
#spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.ddl-auto=create-drop

#Faz com que o hibernate crie as tabelas no BD antes de inserir os dados (data.sql)
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

Sendo assim, as linhas anteriores deverão ser inseridas no application.properties, disponível na pasta resources. Observe!



Sobre as propriedades que configuram o banco embarcado, cabe destacar alguns pontos. Veja!

Na primeira linha, temos, ao final da url de conexão, o nome do banco de dados a ser criado. Nesse caso, apirest. Nas penúltima e última linhas, temos a propriedade responsável por gerenciar a criação da estrutura do banco de dados (o próprio database, suas tabelas, colunas etc.). Perceba que a penúltima linha está comentada. Além disso, a propriedade nela contida é semelhante à da última linha, mudando apenas, entre as duas, os valores definidos.

O valor update configura a aplicação para não recriar, a cada inicialização da API, toda a estrutura do banco, mas apenas atualizá-la, se necessário. Já o valor create-drop faz com que, a cada inicialização, toda a estrutura do banco seja deletada e recriada. Logo, atente-se para modificar essas linhas. Na primeira vez que subir a API, deixe habilitado o valor create-drop.

Ao final do arquivo de propriedades, temos duas linhas, responsáveis por informar ao Spring que as queries existentes no arquivo data.sql (ver imagem anterior, dentro da pasta resources) deverão ser executadas na inicialização da API. Através desse recurso, podemos fazer uma carga inicial em nosso banco de dados.

Agora, confira um exemplo de query que você poderá modificar, acrescentando nos registros para serem inseridos.

plain-text

```
INSERT INTO country (name, capital, currency, population) VALUES ('Spain', 'Madrid', 'EUR', 46704314);
INSERT INTO country (name, capital, currency, population) VALUES ('Poland', 'Warsaw', 'PLN', 38186860);
INSERT INTO country (name, capital, currency, population) VALUES ('United Kingdom', 'London', 'GBP', 63705000);
```

## Spring Boot Initializr: criando a base da aplicação

O Spring Boot disponibiliza a ferramenta Initializr, através da qual podemos configurar nossa API e baixar toda sua estrutura inicial. Observe a configuração sugerida para nossa API, vista a partir da Initializr.

The screenshot shows the Spring Initializr web application interface. It is configured for a REST API. The 'Project' section shows 'Maven' selected as the build tool. The 'Language' section shows 'Java' selected. The 'Spring Boot' version is set to '3.1.0'. The 'Project Metadata' section shows 'Group' as 'com.groupid', 'Artifact' as 'apirest', 'Name' as 'apirest', 'Description' as 'API Rest implementada com Spring Boot', and 'Package name' as 'com.groupid.apirest'. The 'Packaging' section shows 'Jar' selected. The 'Java' version is set to '17'. The 'Dependencies' section shows 'Spring Boot DevTools', 'Spring Web', 'Spring Data JPA', and 'H2 Database' selected. A button 'ADD DEPENDENCIES... CTRL + B' is visible.

Configuração para API REST

Nesta configuração, teremos:

1

Project – Seleção do gerenciador de dependências/tipo de projeto  
Foi escolhido o Maven.

2 Language – Linguagem de programação a ser utilizada  
Foi selecionada a linguagem Java.

3  
Spring Boot – versão do framework  
Foi selecionada a versão 3.1.0.

4  
Project Metadata – metadados do projeto  
Foram definidos o Group, Artifact, Description e, conseqüentemente, o Package name. Aqui, você pode definir os valores de acordo com sua preferência.

5  
Packaging – tipo de empacotamento  
Foi selecionado o Jar.

6  
Java – versão do Java  
Foi selecionada a versão 17.

7  
Dependencies – dependências/bibliotecas a serem utilizadas no projeto  
Foram selecionados o Spring Boot Dev Tools, o Spring Web, o Spring Data JPA e o H2 Database. Sobre tais libs, na própria Initializr, é possível ler uma descrição sobre cada uma delas. Além disso, há outras dependências disponíveis. Para ver a lista completa, clique em Add Dependencies.

Após configurar o projeto, é possível realizar o download da estrutura inicial da aplicação, em formato zip. Para isso, clique, ao final da página, em Generate. Após o download, copie o arquivo compactado para uma pasta de sua preferência. Recomendamos que não utilize nenhuma pasta de sistema, como Downloads, Documentos, Desktop etc. Prefira utilizar uma pasta diretamente na raiz – unidade C:\ ou outra, caso disponível. Descompacte o arquivo. A seguir, veremos como importá-lo na IDE.

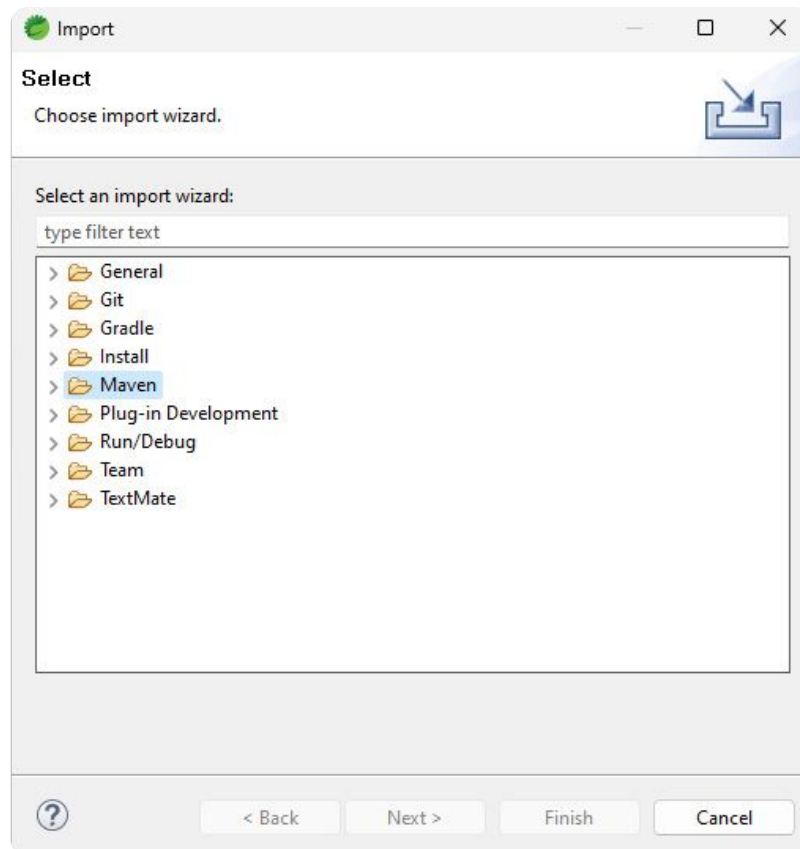
## Configurando o projeto na IDE

Os próximos passos, nos quais abordaremos a configuração e posterior codificação do projeto, serão apresentados usando a IDE Spring Tool Suite, uma ferramenta gratuita baseada no Eclipse. No entanto, é possível empregar qualquer IDE Java de sua escolha.

## Importando o projeto

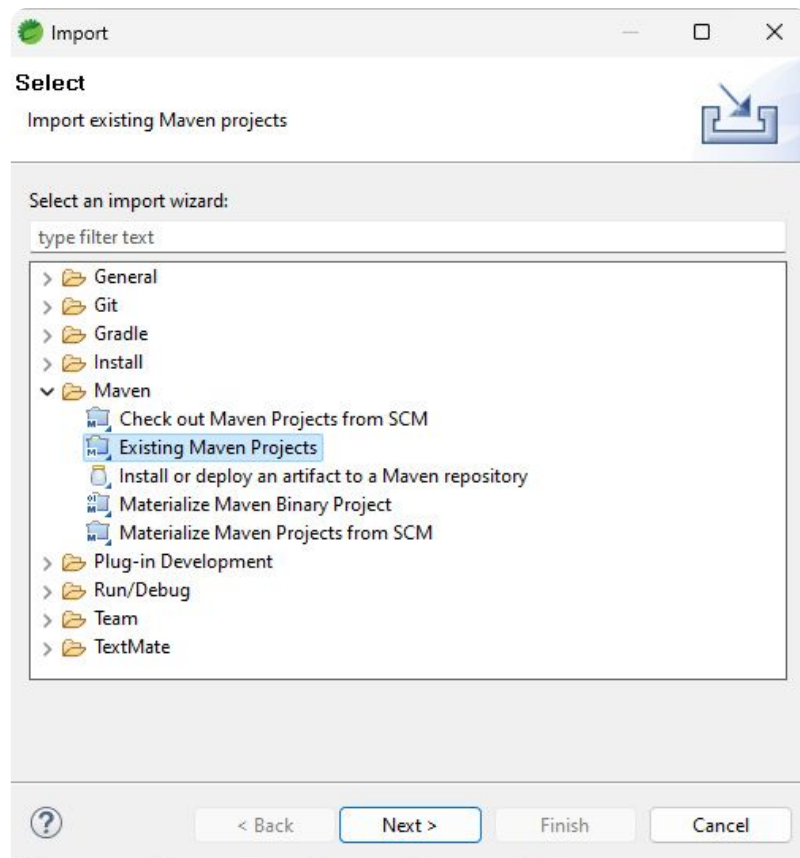
Com a IDE Spring Tool Suite (STS) aberta, clique com o botão direito do mouse na área/janela Project Explorer – localizada à esquerda – e, a seguir, na opção Import. Nesse ponto, a seguinte janela será exibida:





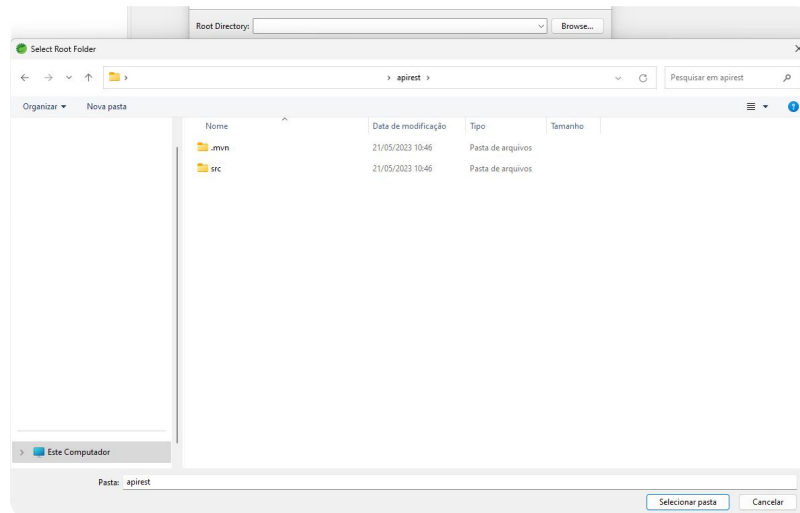
Janela de importação de projeto na STS

Dentre as opções de tipo de projeto, há uma chamada Maven, mostrada em destaque na imagem. Expanda tal item, clicando no símbolo > à sua esquerda. Dentre as novas opções apresentadas, há uma chamada Existing Maven Projects. Selecione essa opção, conforme mostrado a seguir.



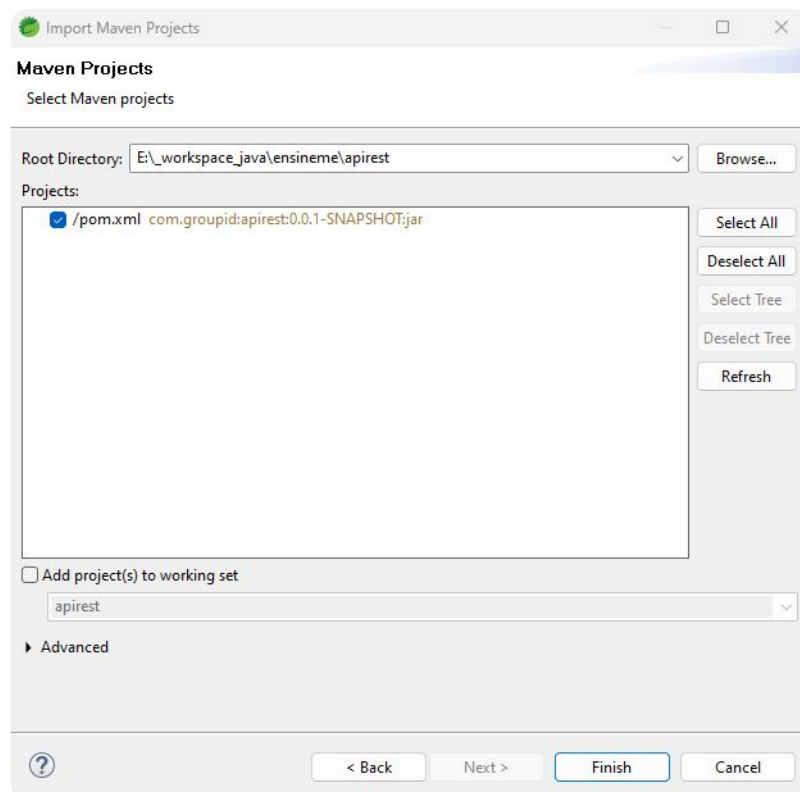
## Janela de importação de projeto na STS

Na sequência, clique no botão Next. Na próxima tela, a partir do botão Browse, selecione a pasta que descompactou anteriormente e que contém a base da aplicação.



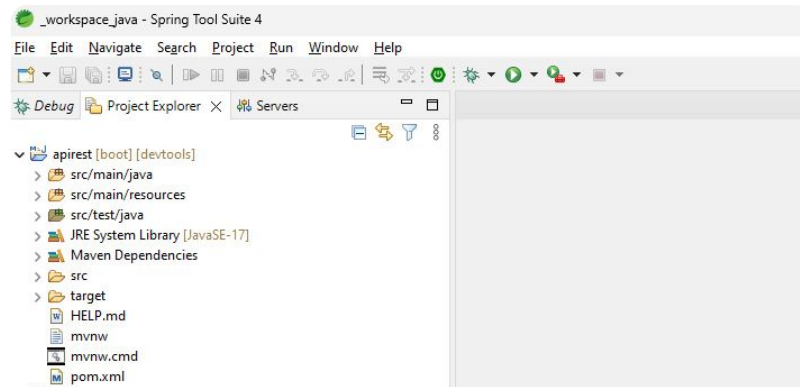
Seleção da pasta contendo a estrutura inicial do projeto na STS

Vendo a tela semelhante a essa, onde são visíveis as pastas **.mvn** e **src**, clique no botão Selecionar pasta. Feito isso, na tela de importação, será mostrado o arquivo **pom.xml** do projeto já selecionado. Confira!



Tela com projeto a ser importado selecionado na STS

Agora, clique no botão Finish. Com isso, a IDE importará o projeto, que ficará disponível na janela Project Explorer.



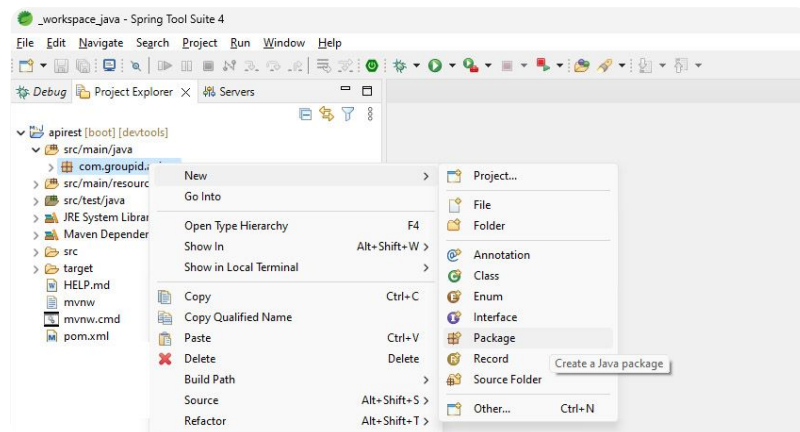
Visão do projeto importado/criado na IDE STS

## Componentes da aplicação

Nesse projeto, utilizaremos a divisão em camadas proposta pelo framework: **Entity, Repository, Service e Controller**. Tais camadas serão criadas como packages em nossa aplicação. A seguir, em cada uma delas, criaremos os códigos responsáveis pela disponibilização dos recursos REST – em nosso caso, um endpoint que permitirá a listagem dos dados dos países cadastrados em nosso banco de dados.

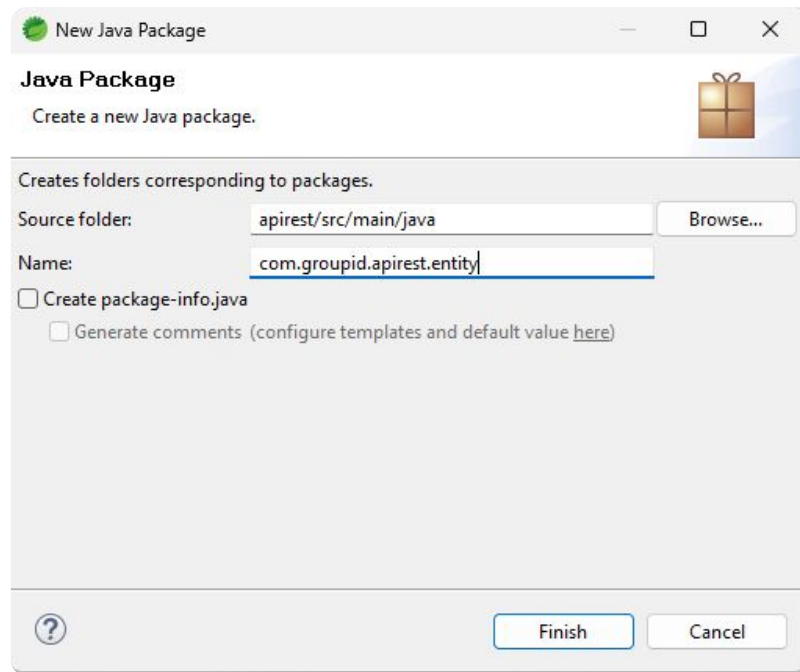
## Package Entity

A partir do pacote principal da aplicação, o `com.groupid.apirest`, que pode ser visto dentro da pasta `src/main/java`, crie um novo package chamado `entity`. Para isso, clique com o botão direito do mouse sobre o pack principal. A seguir, escolha a opção `new` e, depois, a package.



Criação de novo package na STS

Na tela seguinte, digite o nome do pacote: **Entity**. Para finalizar a criação, clique em Finish.



Criação do package na STS

Ao final do processo, você verá, na janela Project Explorer, o novo pacote criado. Dando continuidade, criaremos nossa primeira entidade, que chamaremos de Country. Isso pode ser feito clicando com o botão direito do mouse sobre o pacote que acabamos de criar, a seguir, em New e, depois, em Class. Na janela que abrirá, escreva o nome da entidade e deixe as demais opções como estão. Clique em Finish para concluir essa etapa.

Após criada, abra a classe Entity, clicando duas vezes sobre ela. Substitua o código da classe existente (cuidado para não alterar a primeira linha, que contém a informação do package definido por você, que poderá ser diferente do utilizado aqui) por este.

java

```
import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@JsonIdentityInfo(
    scope = Country.class,
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "idCountry"
)
@Entity
@Table(name = "country")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_country")
    private Integer idCountry;

    @Column(name = "name")
    private String name;

    @Column(name = "capital")
    private String capital;

    @Column(name = "currency")
    private String currency;

    @Column(name = "population")
    private Integer population;

    public Integer getIdCountry() {
        return idCountry;
    }

    public void setIdCountry(Integer idCountry) {
        this.idCountry = idCountry;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCapital() {
        return capital;
    }

    public void setCapital(String capital) {
        this.capital = capital;
    }

    public String getCurrency() {
        return currency;
    }

    public void setCurrency(String currency) {
        this.currency = currency;
    }

    public Integer getPopulation() {
        return population;
    }

    public void setPopulation(Integer population) {
        this.population = population;
    }
}
```

Sobre esse código, destacamos alguns pontos que correspondem à forma como o Spring Boot define os papéis dos códigos que compõem a aplicação. Vamos conferi-los!

#### @JsonIdentityInfo

---

Essa anotação é utilizada para evitar que o Spring Boot faça o aninhamento recursivo, de forma infinita, entre as entidades relacionadas. Embora nossa API tenha, inicialmente, apenas uma entidade, é interessante já anotá-la com essa configuração. Faça o mesmo nas demais entidades que venha a criar na aplicação.

#### @Entity

---

Essa anotação concede à classe Country o papel de entidade – objeto com o qual os dados serão transferidos na API.

#### @Table(name = "country")

---

Essa anotação é utilizada para definir o nome da tabela a ser gerada no banco de dados – em nosso caso, a tabela country.

#### @Column(name = "...")

---

Essa anotação é inserida em cada atributo da classe Country e serve para definir o nome da coluna a ser criada no banco de dados, referente ao atributo em questão.

#### @Id e @GeneratedValue

---

Essas anotações são inseridas apenas no atributo que terá o papel de chave-primária. A primeira concede ao atributo o papel em questão, e a segunda serve para definir de quem será a responsabilidade por gerar os valores desse atributo. No código apresentado, foi definido que esse papel será do próprio SGBD (o banco embarcado H2 que utilizaremos).

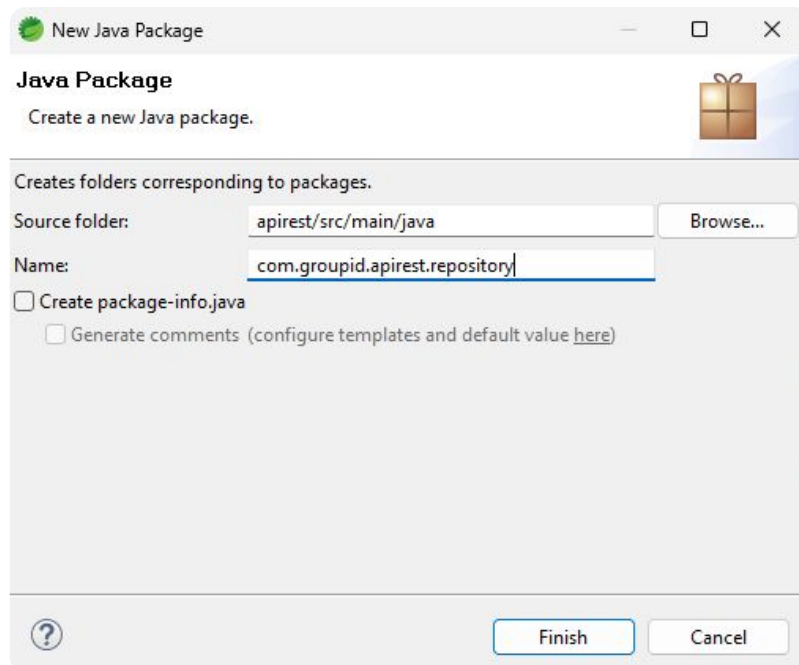
#### Métodos Get e Set

---

Como padrão, os atributos de nossa classe são private e, portanto, precisam de métodos para serem acessados: Get e Set.

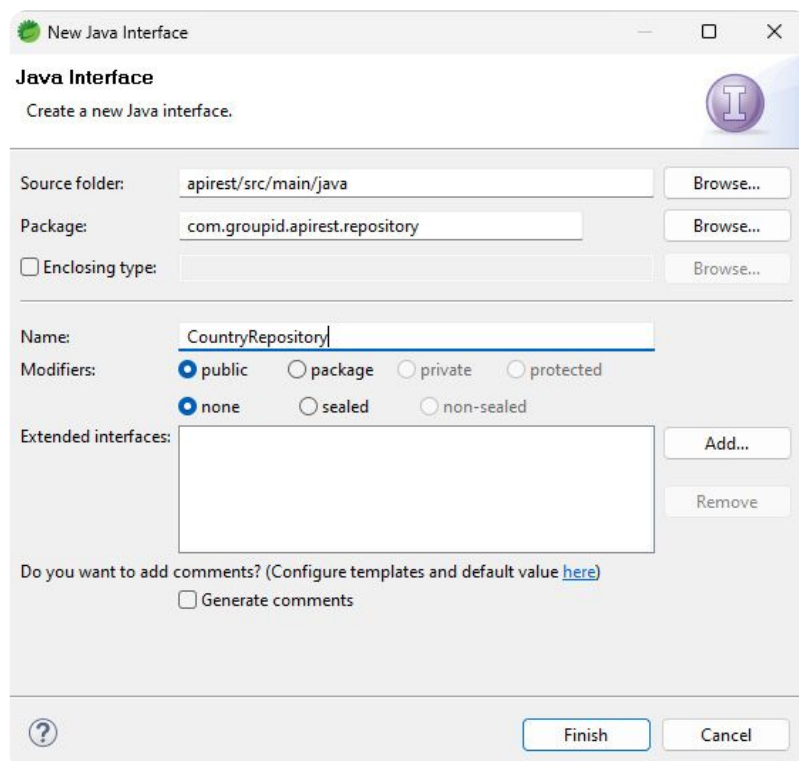
## Package Repository

Após a criação da entidade, podemos criar o repositório. Diferente dos demais scripts criados na API, o repository será uma interface Java. Além disso, ela herdará, como já mencionado, os métodos de acesso a dados disponibilizados pela JpaRepository. Para criação da interface em questão, crie primeiro um package, seguindo os passos já vistos. Dê a esse novo pacote o nome de repositório.



Criando o package repository na STS

Com o package criado, proceda à criação de um novo script dentro dele, configurado como uma interface, e atribua a ele o nome de CountryRepository.



Criando a interface CountryRepository na STS

Em relação ao seu código, a implementação inicial do CountryRepository é bastante simples. Como herdaremos os métodos do JpaRepository, não precisaremos, inicialmente, definir nenhum método em tal interface. Veja como deverá ficar seu conteúdo (lembre-se de manter em seu script a linha inicial, contendo o endereço do package).

```

java

import org.springframework.data.jpa.repository.JpaRepository;
import com.groupid.apirest.entity.Country;

public interface CountryRepository
    extends JpaRepository {

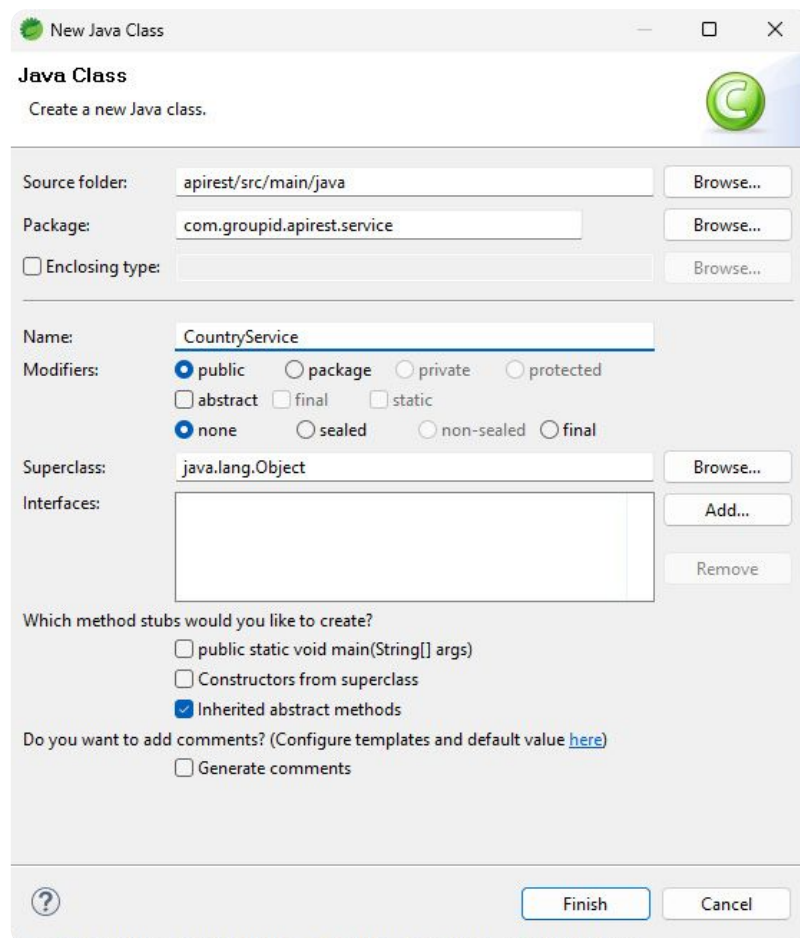
}

```

Sobre o escopo da interface, perceba que foi feito o extends da JpaRepository e que ela deve receber dois parâmetros: o nome da entidade a que se refere – em nosso caso, Country – e o tipo de dado do atributo com papel de chave primária (anotado com o @Id) da entidade relacionada – em nosso caso, Integer.

## Package Service

Depois de criados os packages Entity e Repository, criaremos o package Service. Siga os passos utilizados anteriormente e crie o novo package. Dentro dele, criaremos uma classe Java chamada CountryService. Veja!



Criando o CountryService na STS

Repare que criamos o Repository e o Service usando o padrão NomeDaEntidade + Papel: CountryRepository, CountryService. Isso facilita a identificação de cada script dentro da API. Logo, recomendamos manter esse padrão por toda a aplicação.

Na classe Service, são definidos os métodos responsáveis pelas regras de negócio, quando existentes, e os métodos responsáveis pela intermediação entre a camada Controller e a camada Repository. Em nossa API, disponibilizaremos, inicialmente, apenas um serviço, responsável por recuperar a listagem de países existentes. A seguir, vejamos o código do CountryService.



```

java

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.groupid.apirest.entity.Country;
import com.groupid.apirest.repository.CountryRepository;

@Service
public class CountryService {
    @Autowired
    CountryRepository countryRepository;

    public List getAllCountries(){
        return countryRepository.findAll();
    }
}

```

Agora, vamos aos comentários sobre esse código!

## @Service

Essa anotação define o papel da classe em questão dentro do Spring Boot, ou seja, trata-se de uma classe de serviço. Veja a definição desse tipo de classe, retirada da documentação do Spring (Spring Framework, 2023).

“A anotação @Service indica que a classe anotada é um ‘Service’, originalmente definido pelo Design Orientado a Domínio (Domain-Driven Design – Evans, 2003) como uma operação fornecida como uma interface que fica sozinha no modelo, sem nenhum estado encapsulado. Além disso, também pode indicar que a classe anotada é um ‘Business Service Facade’ (dentro do core dos padrões J2EE), ou algo similar. Além disso, essa anotação fornece uma especialização de outra anotação, a @Component, que permite que classes com essas anotações sejam autodetectadas no escaneamento de classes da aplicação”.

## @Autowired

Essa anotação, utilizada em cima de instâncias de outras classes ou interfaces, permite, pelo próprio Spring Boot, o autogerenciamento (através de injeção de dependências e inversão de controle) da criação e manuseio de instâncias de classes ou interfaces por ela anotadas.

Veja ainda, no código, que o retorno do método criado contém apenas uma chamada a um método do CountryRepository: findAll(). Esse método, disponibilizado pela interface JpaRepository, implementa a query no banco de dados responsável pela seleção de todos os dados de determinada tabela, devolvendo uma coleção (Lista Java) da entidade em questão. Além desse método, há vários outros, como findById, save, delete etc.

## Package Controller

Após a codificação das demais camadas que fazem parte de nossa API, chegou a hora de codificar a camada que expõe os recursos disponibilizados em nossa aplicação. O package Controller conterá uma classe Controller para cada entidade de nossa API, ficando a nosso critério definir quais os recursos desejamos oferecer – como a recuperação dos dados dos países, a recuperação do país com base no seu nome, a persistência de um novo país etc.

Seguindo os passos já vistos, crie um package chamado controller e, dentro dele, uma classe chamada CountryController. Depois, substitua o código pelo exibido a seguir, lembrando sempre de manter a primeira linha do seu script, que contém o endereço do package, confira!

```

java

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.groupid.apirest.entity.Country;
import com.groupid.apirest.service.CountryService;

@RestController
@RequestMapping("/countries")
public class CountryController {
    @Autowired
    CountryService countryService;

    public ResponseEntity<
        getAllCountries(){
            return new ResponseEntity<>(countryService.getAllCountries(),
                                      HttpStatus.OK);
        }
}

```

Agora, vamos aos comentários sobre esse código!

## @RestController

Essa anotação define, dentro do Spring Boot, o papel da Classe com ela anotada: um provedor de recursos REST.

## @RequestMapping("/countries")

Essa anotação é empregada para estabelecer a identificação da URI do recurso. Nesse contexto, a URI countries será adicionada à URL de nossa API, permitindo o acesso aos serviços oferecidos por meio do recurso denominado countries.

## @Autowired

Como já vimos no Service, aqui também utilizamos esse recurso para instanciarmos o CountryService em nosso Controller.

## ResponseEntity

Perceba que, diferente do que fizemos no Service, aqui, para retornar os dados, estamos usando a classe ResponseEntity, que nos permite, além de retornar valores, manipular o código HTTP da resposta a ser enviada. Em nosso caso, definimos que o código HTTP a ser retornado é o 200, ou seja, informamos que a requisição foi executada com sucesso.

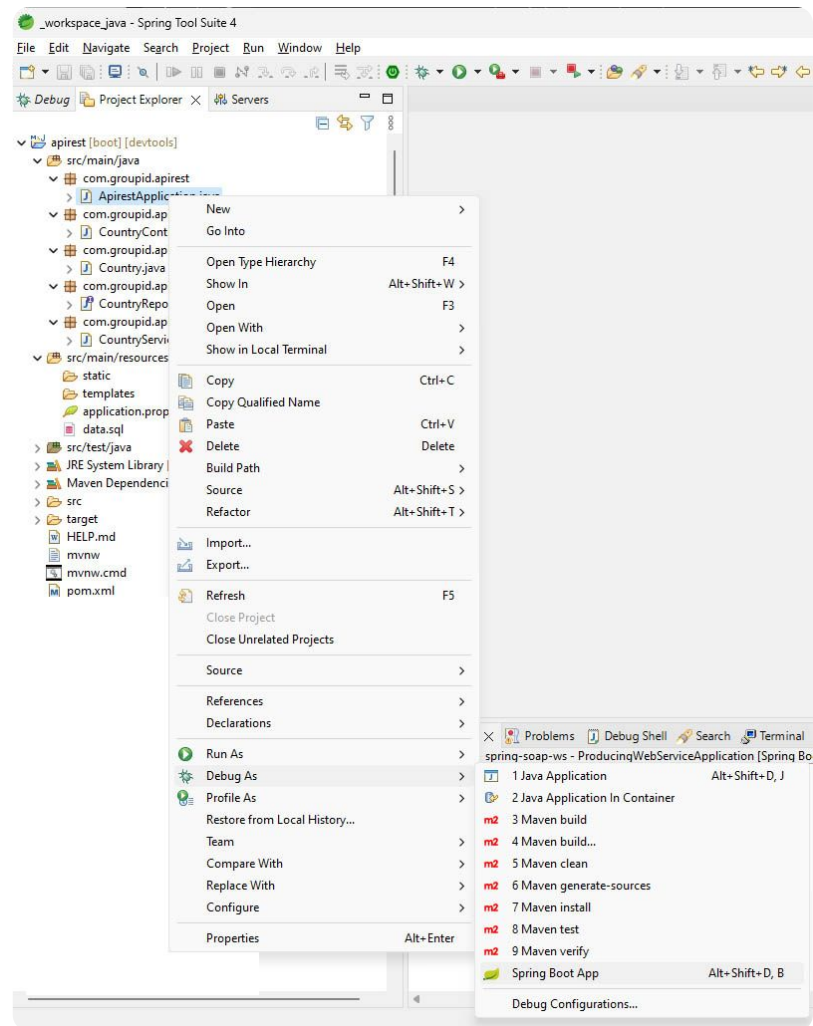
Sobre os códigos HTTP, é importante tratarmos as possíveis exceções em cada método implementado, a fim de retornarmos os códigos corretos para cada situação. Por exemplo, se implementarmos um método que recupera os dados de um país pelo seu nome, e se tal país não existe em nosso banco de dados, o correto será retornar um código 404, de registro não encontrado (Not Found).

Nesse ponto, após codificado o Controller, nossa API está pronta. A seguir, veremos como testar o recurso disponibilizado a partir de nossa aplicação.

## Consumindo a API REST

Após concluída a codificação da API, chegou a hora de verificar se tudo está funcionando e de testar a API, utilizando um cliente para consumir o recurso disponibilizado por ela. Para começar, precisamos “subir” a API. Essa tarefa pode ser feita na própria IDE.

No caso da STS, na primeira vez que for iniciar a API, clique com o botão direito sobre a classe principal: `ApirestApplication.java` (ou, caso tenha definido outro nome para o projeto, a classe cujo nome termina com `Application.java`, localizada na raiz do package principal da aplicação). A seguir, selecione a opção **run as** ou **debug as** e, por fim, clique em **Spring Boot App**.



Inicializando a API na STS

## Atividade 1

O framework Spring Boot usa anotações para gerenciar a injeção de dependências e a inversão de controles entre as classes que compõem uma aplicação, permitindo que papéis específicos sejam atribuídos a classes, métodos, atributos etc. Nesse contexto, na camada Controller, algumas anotações são usadas para mapear as requisições realizadas para a API em relação ao verbo HTTP utilizado em seu acesso. Assinale a alternativa que contém exemplos de tais anotações.

A

@GET, @POST, @PUT e @DELETE

B

@RestController e @RequestMapping

C

@Entity, @Repository e @Service

D

@GetMapping, @PostMapping, @PutMapping e @DeleteMapping

E

@Id, @Column e @JsonIdentityInfo



A alternativa D está correta.

O Spring Boot possui anotações específicas para mapear as requisições recebidas pela API, de acordo com o verbo HTTP utilizado nelas. Na alternativa D, podemos ver exemplos dessas anotações.

## Como testar uma API REST

O consumo ou testes de uma API REST pode ser realizado de diferentes formas: utilizando aplicações criadas em distintas linguagens de programação, com softwares de teste específicos, com plugins em navegadores ou diretamente nos navegadores, através de comandos em terminais de sistemas operacionais etc. Independentemente da ferramenta utilizada, os testes da API são uma tarefa indispensável para quem os desenvolveu. Através desse processo, todos os endpoints e recursos disponibilizados deverão ser checados, a fim de verificar se atendem ao seu propósito.

Assista ao vídeo e conheça diferentes formas de testar uma API REST. Além disso, compreenda também a significativa importância desses testes para garantir a qualidade e o correto funcionamento dos endpoints e recursos da API.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

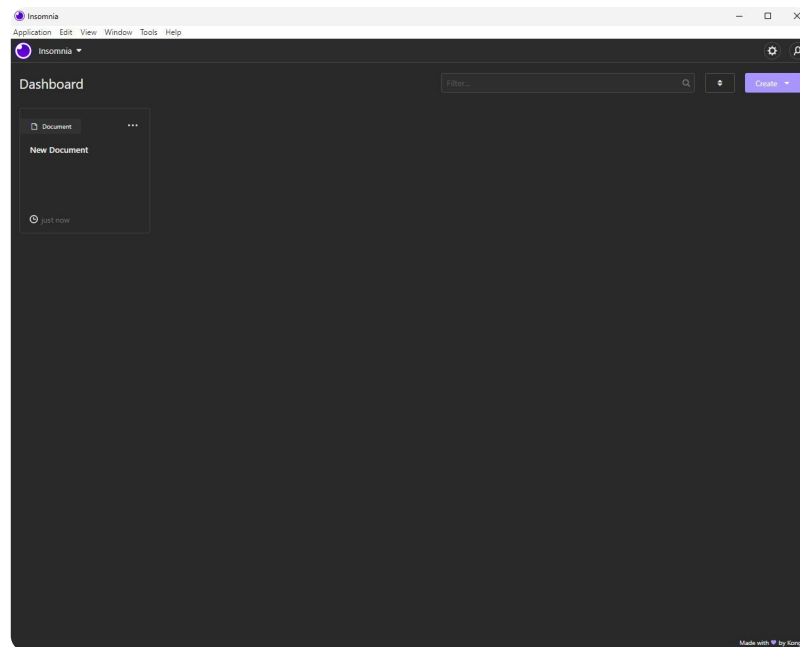
## Utilizando o Insomnia

Uma das formas mais simples para testar uma API REST é usar softwares específicos. Há várias ferramentas disponíveis, com versões gratuitas ou pagas, como o Postman ou o Insomnia, por exemplo. Agora, vamos conferir como utilizar o Insomnia para testar uma API REST.

## Download e instalação

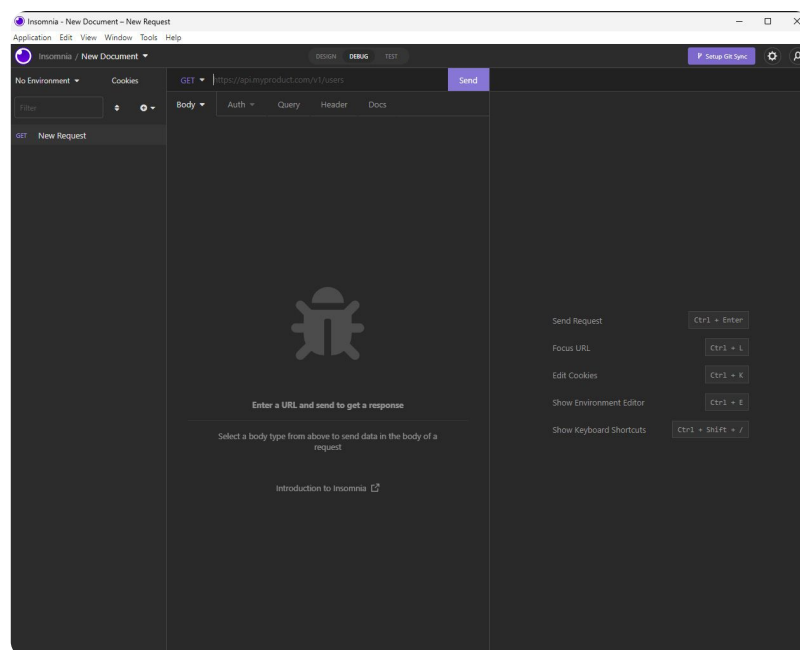
O download do Insomnia pode ser feito através de sua página oficial. Esse software é gratuito e possui versões pagas, onde funcionalidades adicionais, como sincronização, projetos compartilhados ou projetos em cloud, estão disponíveis. Entretanto, a versão gratuita atende perfeitamente ao processo de testes de API.

Após instalar e executar o Insomnia pela primeira vez, somos levados à sua página inicial.



Tela inicial do Insomnia

É comum, também, que o aplicativo seja iniciado dentro de um Document. Veja!

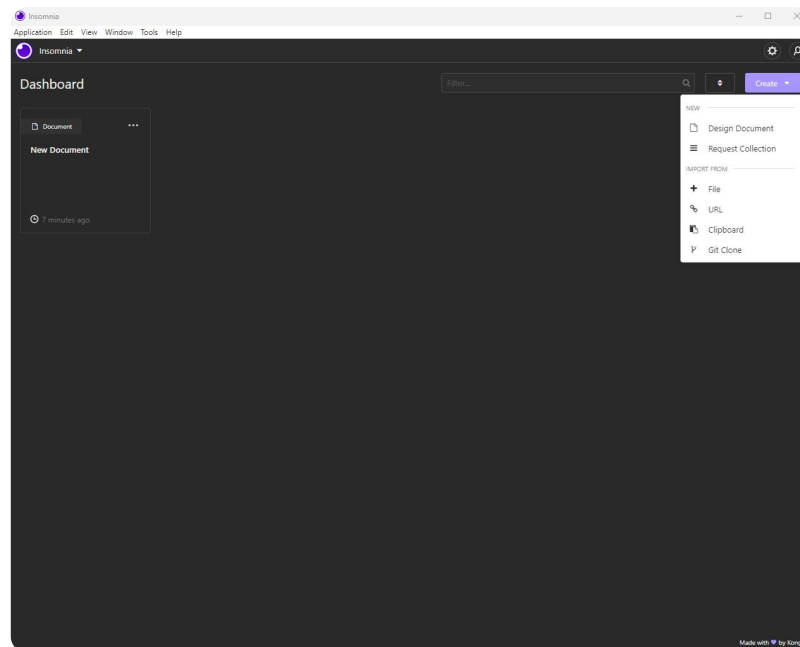


Tela inicial do new document do Insomnia

Antes de prosseguirmos, cabe destacar a forma como o Insomnia nos permite organizar nossas requisições, projetos, coleções etc. Um Document é uma forma de organizar um projeto, assim como uma Request Collection. Logo, podemos ter diferentes grupos de requisições, o que, inclusive, é uma boa prática. Para navegar entre a página de um Document ou Request Collection para a página inicial, basta usar os links (em estilo breadcrumb) disponíveis logo na parte inferior do menu principal.

## Criação de uma Request Collection

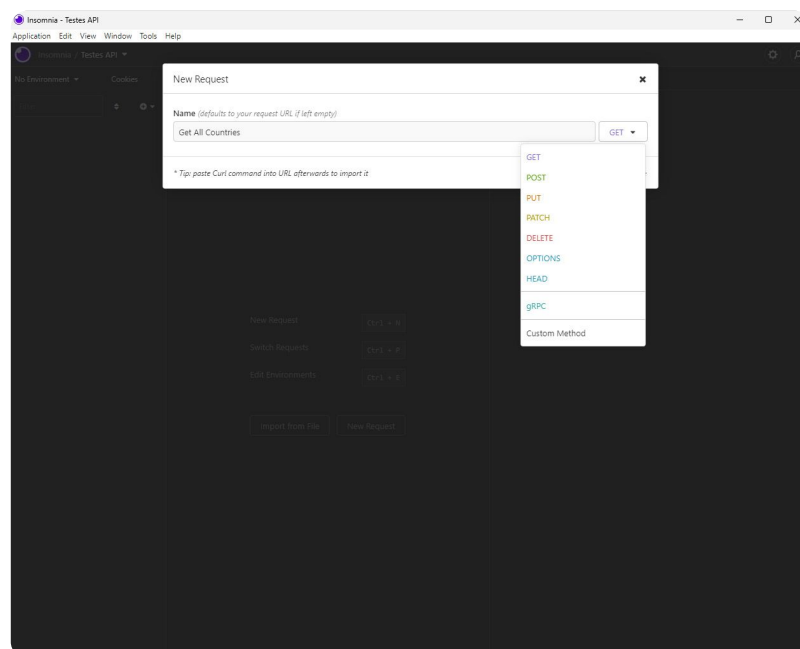
Para organizar melhor nossos testes, vamos criar uma Request Collection. Para isso, você deverá estar na página inicial. No canto superior direito, há um botão chamado Create. Ao clicar nele, algumas opções são apresentadas.



Criando uma coleção do Insomnia

A partir das opções, clique em Request Collection. A seguir, dê um nome para a coleção. Clique em Create para finalizar esse processo. Na sequência, você será levado para a tela inicial da coleção. Nessa tela, temos, tanto do lado direito quanto no centro, atalhos para criarmos as requisições. Nosso foco, agora, será este: criar uma requisição HTTP para testar um recurso REST.

O primeiro passo, ao selecionarmos a opção de criação de uma nova requisição, é darmos um nome a ela. Essa opção nos ajuda a descrever/identificar a requisição. Além do nome, você também deve selecionar o método/verbo HTTP, dentre as opções disponíveis no combobox à direita. Veja!



Criação de requisição do Insomnia

Digite um nome para a requisição – por exemplo Get All Countries. Vamos testar o recurso que criamos anteriormente em nossa API REST. No método, deixe como GET. Finalize clicando em Create. Agora, precisamos inserir o endereço do recurso que desejamos consumir.

Na parte central superior da tela, há a barra de endereços. Preencha com os seguintes dados: `http://localhost:8080/countries`. Nesse endereço, teremos:

### 1 URL

http://localhost (um endereço que se refere ao seu próprio computador).

### 2

Porta onde o servidor está rodando

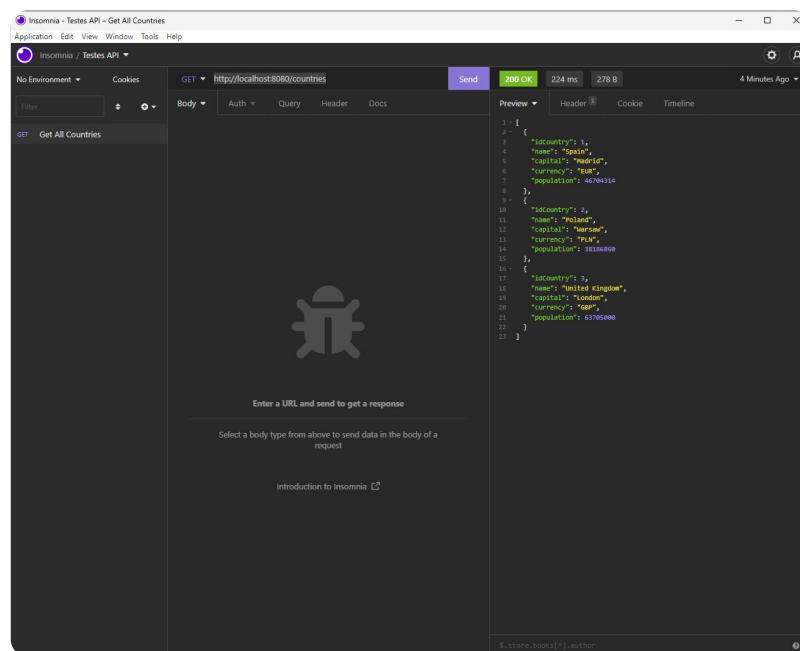
8080 (normalmente, o Tomcat embutido no Spring Boot roda nessa porta. Caso necessário, é possível alterar tal configuração).

### 3

URI

/countries (definimos essa configuração em nosso Controller através da anotação @RequestMapping).

Após inserir a URL, você terá completado o processo de configuração da requisição. Agora, basta clicar em SEND. Tendo sucesso ou erro na requisição, você poderá ver seu resultado na parte direita da tela do aplicativo.



Requisição e resposta do consumo de API REST do Insomnia

Sobre o resultado da requisição, cabem alguns comentários. Vamos conferi-los!

### Primeiro

---

A resposta à requisição HTTP é exibida (por padrão em APIs REST) no formato de string JSON.

### Segundo

---

Além do conteúdo da resposta – em nosso caso, uma lista dos países cadastrados no banco de dados –, também é exibido, logo acima da resposta, algumas informações, como o tempo de execução e o código de status HTTP – aqui, o código “200 OK”.

Após vermos o resultado, chegamos ao final do processo de configuração e execução de nossa primeira requisição. Além das opções aqui demonstradas, há outras, complementares, disponíveis no Insomnia, como definição de propriedades específicas no header da requisição, definição de opções de autenticação etc.

## Atividade 2

Sobre a utilização de ferramentas para testes em API REST, é correto afirmar que:

A

através do uso de ferramentas específicas de teste, é possível ter acesso, ao fazer uma requisição para a URL principal da API, a todos os recursos disponíveis nela.

B

apenas as ferramentas comerciais/pagas permitem a realização de testes em APIs REST.

C

uma ferramenta de testes de API REST permite apenas a realização de requisições do tipo GET.

D

para realizar requisições do tipo POST, onde dados serão persistidos a partir da API, é obrigatório informar credenciais (usuário e senha), por questões de segurança.

E

o uso de ferramentas de teste permite realizar, de forma fácil e prática, testes em todos os endpoints e recursos disponibilizados em uma API REST, inclusive recursos protegidos, mediante a obtenção prévia das credenciais de acesso.



A alternativa E está correta.

As ferramentas de testes permitem o acesso aos endpoints e recursos disponibilizados a partir de uma API REST. Através delas, podemos realizar requisições de qualquer tipo, usando qualquer método HTTP (considerando, claro, os que foram disponibilizados pelas APIs). Nesse contexto, há várias ferramentas disponíveis, tanto gratuitas quanto pagas.



# Adicionando novos recursos à API REST

Tendo codificado nossa primeira API REST, na qual disponibilizamos um recurso, o de consulta de todos os países existentes no banco de dados associado à aplicação, chegou a hora de colocarmos em prática tudo o que vimos até aqui.

Assista ao vídeo e aprenda a incorporar um novo recurso à sua API REST, desde a codificação até a integração com o banco de dados. Aprimore suas habilidades em desenvolvimento de APIs REST para facilitar a busca de países.



## Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Roteiro de prática

Para essa atividade, o ponto de partida é a API REST criada anteriormente. A partir dela, e usando as camadas já existentes (Entity, Repository, Service e Controller), você deve criar um recurso que permitirá a busca pelos dados de um país a partir do seu nome.

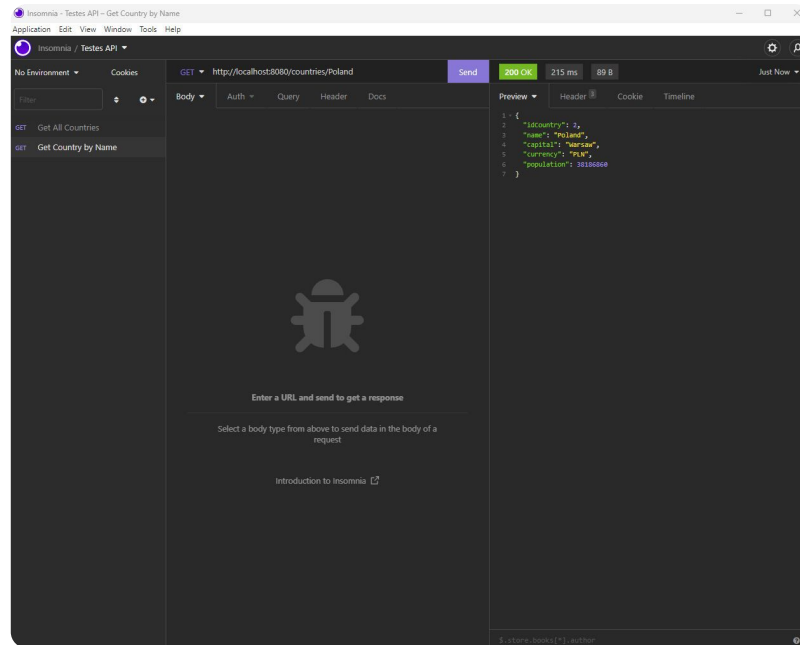
Agora, vamos ao passo a passo!

- Em relação à entidade, você não precisará modificar nada.
- Quanto ao repositório – interface `CountryRepository` –, você deverá criar um método dentro dele. Isso será necessário, porque, nos métodos disponibilizados pela `JpaRepository`, não há um método predefinido que possibilite a busca por atributos da entidade, com exceção do Id. Entretanto, é possível codificar métodos do tipo `find` para qualquer atributo, usando o padrão `findNomeDoAtributo(parâmetro)`. Por exemplo, se você tem um atributo na entidade relacionada ao repositório chamado `cidade`, você poderá criar, dentro do repositório, o método `findByCidade(String nomeCidade)`.
- Sobre o Service, você precisará criar um método, que responderá à requisição vinda do Controller e chamará o novo método criado na etapa anterior.
- Sobre o Controller, você também precisará elaborar um novo método. Nesse caso, por estarmos criando um recurso de recuperação de dados, o método deverá ser anotado com `GetMapping`. Aqui é importante saber que não pode haver dois métodos com exatamente o mesmo mapeamento. Considerando já haver um método no Controller anotado com `GetMapping`, esse novo método deverá receber essa anotação acrescida de um atributo adicional, como `@GetMapping("/{nome})`.
- Ainda sobre o Controller, ao definir o mapeamento conforme foi sugerido, você precisará adicionar uma anotação ao parâmetro que será recebido pelo método `@PathVariable`. Nesse caso, imaginando que você tenha um método que recebe como parâmetro uma String chamada `cidade`, e que o método tenha sido anotado como `@GetMapping("/{cidade}")`, o parâmetro do método ficaria assim: `nomeDoMetodo(@PathVariable String cidade)`. Repare, aqui, que o nome do parâmetro deverá ser igual à palavra usada no `@GetMapping`.
- Ao final da codificação, você deverá testar o novo recurso, usando o `Insomnia`.

A resolução desse serviço pode ter mais de uma resposta. É importante analisar o resultado do teste do recurso no Insomnia, onde os dados do país procurado devem ser exibidos, como exemplificado a seguir. A resolução do problema é ilustrada nestes códigos. Veja!

## Resultado do teste do recurso no Insomnia

Agora, vejamos o resultado do teste do recurso no Insomnia.



Resultado do teste do recurso no Insomnia

## CountryRepository.java

A seguir, veja como o código se encontra no CountryRepository.java.

```
java

public interface CountryRepository
    extends JpaRepository {
    Country findByName(String name);
}
```

## CountryService.java

Agora, acompanhe como o código se comporta no CountryService.java.

```

java

@Service
public class CountryService {
    @Autowired
    CountryRepository countryRepository;

    public List getAllCountries(){
        return countryRepository.findAll();
    }

    public Country getCountryByName(String name) {
        return countryRepository.findByName(name);
    }
}

```

## CountryController.java

Veja como o código se comporta no CountryController.java.

```

java

@RestController
@RequestMapping("/countries")
public class CountryController {
    @Autowired
    CountryService countryService;

    @GetMapping
    public ResponseEntity<
    getAllCountries(){
        return new ResponseEntity<>(countryService.getAllCountries(),
                                    HttpStatus.OK);
    }

    @GetMapping("/{name}")
    public ResponseEntity
    getCountryByName(@PathVariable String name){
        return new ResponseEntity<>(countryService.getCountryByName(name),
                                    HttpStatus.OK);
    }
}

```

## Faça você mesmo!

Na abordagem sobre o Controller da API, foram mencionadas a importância e a necessidade de tratar tanto o conteúdo quanto o correto código de status HTTP da resposta a uma requisição. Logo, devemos ter conhecimento dos status HTTP, a fim de entender quando utilizá-los.

Considere os itens abaixo:

1

```
java
@GetMapping("/{name}")
public ResponseEntity
    getCountryByName(@PathVariable String name){
    return new ResponseEntity<>(countryService.getCountryByName(name),
        HttpStatus.OK);
}
```

2

```
java
@GetMapping("/{name}")
public ResponseEntity
    getCountryByName(@PathVariable String name){

        Country country = countryService.getCountryByName(name);

        if(null == country)
            return new ResponseEntity<>(null,
                HttpStatus.NOT_FOUND);

        else
            return new ResponseEntity<>(country,
                HttpStatus.OK);

}
```

3

```
java
@GetMapping("/{name}")
public ResponseEntity
    getCountryByName(@PathVariable String name){

        Country country = countryService.getCountryByName(name);

        if(null == country)
            return new ResponseEntity<>(null,
                HttpStatus.BAD_REQUEST);

        else
            return new ResponseEntity<>(country,
                HttpStatus.OK);

}
```

4

```
java
@GetMapping("/{name}")
public ResponseEntity
    getCountryByName(@PathVariable String name){

    Country country = countryService.getCountryByName(name);

    if(null == country)
        return new ResponseEntity<>(null,
            HttpStatus.OK);
    else
        return new ResponseEntity<>(country,
            HttpStatus.OK);

}
```

5

```
java
@GetMapping("/{name}")
public ResponseEntity
    getCountryByName(@PathVariable String name){
        return new ResponseEntity<>(countryService.getCountryByName(name),
            HttpStatus.NOT_FOUND);
    }
}
```

Nesse contexto, considerando o cenário onde é realizada uma requisição com o objetivo de recuperar uma entidade em um recurso, qual das alternativas acima representa correta para implementação desse método?

A

1

B

2

C

3

D

4

E

5



A alternativa B está correta.

O código HTTP recomendado para requisições em que o recurso solicitado não pode ser encontrado é o 404 - NOT\_FOUND. Logo, o código indicado é o da alternativa B, onde temos o retorno da entidade buscada, o código http 200 – OK em caso de sucesso e o http 404 – NOT\_FOUND, sem valor no body, em caso de não localização do recurso buscado.

## Diferenciando APIs, bibliotecas e frameworks

Na engenharia de software, o conceito de API é utilizado/aplicado quando desejamos integrar diferentes softwares, que podem ser escritos em distintas linguagens, possuir diferentes tecnologias ou bancos de dados. É um recurso muito importante, assim como as bibliotecas e frameworks (códigos desenvolvidos por terceiros) – ferramentas que podemos incluir em nosso próprio código, ganhando, assim, agilidade e facilitando o processo de desenvolvimento.

Neste vídeo, você aprenderá a construir uma API REST utilizando o framework Spring Boot. Entenda a diferença entre APIs, bibliotecas e frameworks, e descubra como essas ferramentas podem aumentar sua produtividade no desenvolvimento de sistemas. Aprenda a configurar um projeto Maven, adicionar dependências e criar um web service RESTful. Não perca!



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Bibliotecas

Em sistemas computacionais, temos um grande conjunto de estruturas de dados, funções e procedimentos que podem ser organizados em módulos, disponibilizando operações para algum domínio de utilização específico. Esses módulos, denominados **bibliotecas**, constituem o ferramental de reuso mais comum no mundo da programação.



### Exemplo

Existem muitas bibliotecas no mercado para as mais diversas plataformas, como a JQuery, amplamente utilizada na construção de front-end para web; a Retrofit, que facilita a construção de clientes Java ou Kotlin para web services; e a Panda, usada para análise de dados no ambiente do Python.

## Frameworks

Definem soluções completas, customizáveis para determinados domínios de utilização. Ao contrário das bibliotecas, que apenas disponibilizam os recursos de forma passiva, os frameworks controlam a maioria das operações necessárias, permitindo que o programador se preocupe apenas com os aspectos específicos do negócio.



### Exemplo

Com a utilização do angular, temos a construção de front-end do tipo SPA (single page application) em Type Script, com ótimos recursos de vinculação de dados, enquanto o Spring Web permite criar web services baseados em classes Java comuns, configuradas por meio de anotações.

## API

Application program interface pode ser definida como a exposição de um conjunto de funcionalidades de determinado sistema para ferramentas externas. Para ilustrar, a API de um sistema operacional viabiliza a programação com múltiplas threads em programas criados na linguagem C, bem como a automatização do uso de Word e Excel, ambos produtos da Microsoft, com base em Java Script.



### Exemplo

Os web services, como os oferecidos por empresas de pagamento online, são um exemplo prático de APIs. Eles fornecem conjuntos de serviços interconectados que permitem que aplicativos realizem transações financeiras com segurança, como pagamentos e transferências de fundos. Esses web services atuam como interfaces, permitindo que os aplicativos acionem processos internos para concluir as operações desejadas.

## Aumentando a produtividade

Tanto as APIs quanto as bibliotecas e os frameworks são ferramentas que aumentam a produtividade, trazendo uma série de benefícios ao processo de desenvolvimento. Cada uma dessas opções possui características e recursos específicos. Vamos conferir!

## Injeção de dependências e inversão de controle

Na definição de um framework, injeção de dependências e inversão de controle são recursos que permitem grande aumento da produtividade na construção de sistemas. Ambos fornecem um meio simples para configurar o relacionamento das classes com o framework e delegar atividades para execução no núcleo comum do ambiente.

Quando trabalhamos com bibliotecas, nós as importamos e efetuamos as chamadas para as funções necessárias no fluxo de execução do programa. Entretanto, quando usamos um framework, temos um container que se responsabiliza pela execução de boa parte do processo, o que é definido como inversão de controle.



### Comentário

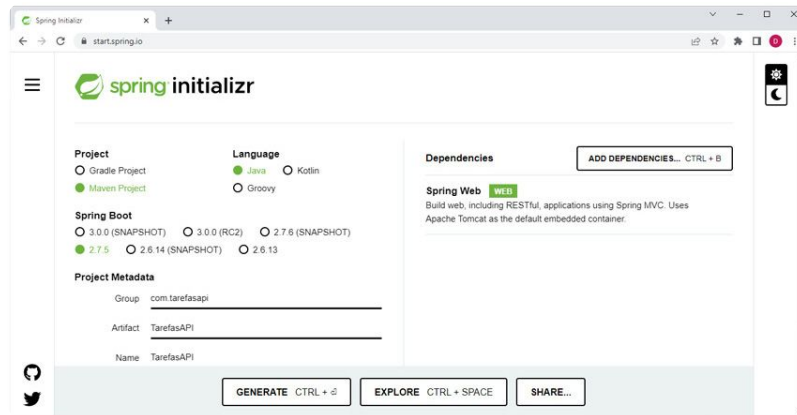
Uma das vantagens do uso de um framework é o baixo acoplamento, já que o container pode ser substituído facilmente, sem causar impacto no código do sistema, além de permitir que o programador se preocupe exclusivamente com a lógica do negócio.

Outra funcionalidade oferecida pelos frameworks é a inclusão de recursos do container, como serviços e conexões com bancos de dados, nos objetos do sistema, com base em metodologias simples, o que é conhecido como **injeção de dependência**.

Exemplos incluem a injeção de serviços em classes TypeScript no Angular, utilizando parâmetros no construtor, ou a utilização de objetos do tipo Model nos controladores Web do Spring para capturar automaticamente dados de um formulário através de uma chamada HTTP.

Agora, vamos construir uma API no estilo REST, com base no framework Spring, executando no modo Spring Boot, que não requer um servidor pré-instalado. Nosso passo inicial será a criação do projeto Maven, utilizando o Spring Initializr. Observe!

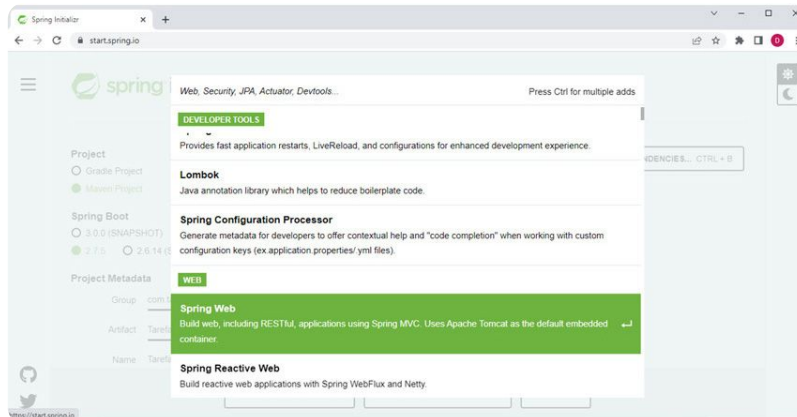




Configuração inicial do projeto no Spring Initializr

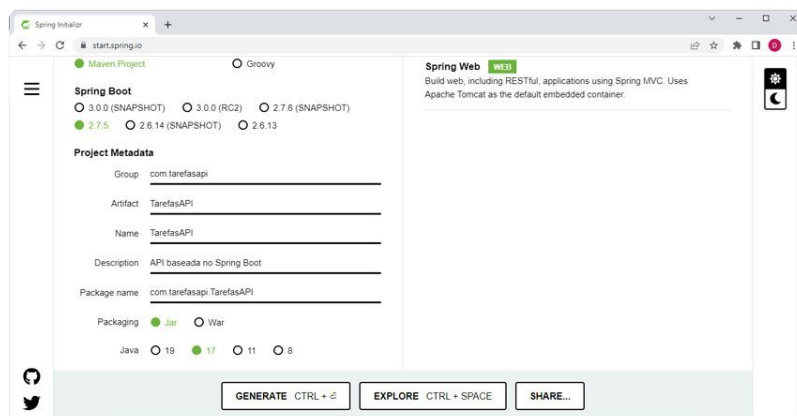
Utilizaremos projeto do tipo Maven, linguagem Java e versão 2.7.5 do Spring. Também vamos definir o grupo como com.tarefasapi, artefato TarefasAPI e nome com mesmo valor.

Após definirmos as características principais do projeto, vamos adicionar a dependência para criar nosso web service RESTful. Para isso, devemos clicar em ADD DEPENDENCIES e, em seguida, escolher o módulo Spring Web no diálogo que será aberto.



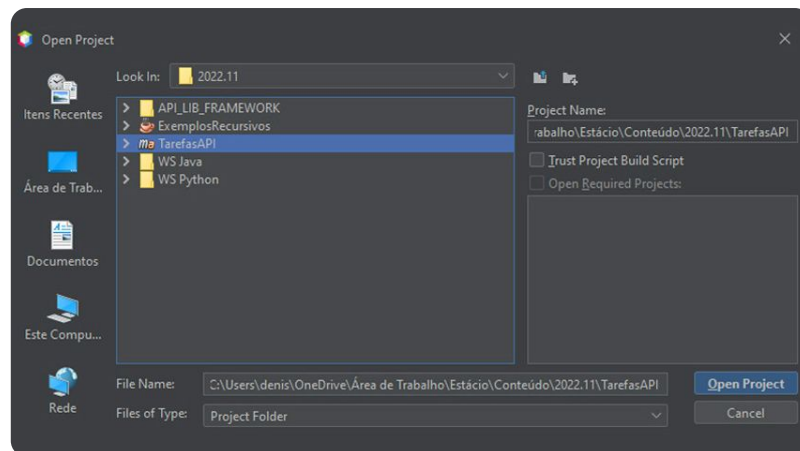
Acréscimo do módulo Spring Web ao projeto

Por fim, podemos efetuar algumas configurações opcionais, como a descrição do sistema e o pacote principal, onde será criada a classe de inicialização do Spring Boot, além de escolher o empacotamento como JAR e a versão do Java. Concluídas as configurações, precisamos apenas clicar em GENERATE para efetuar o download do projeto, compactado no formato ZIP.



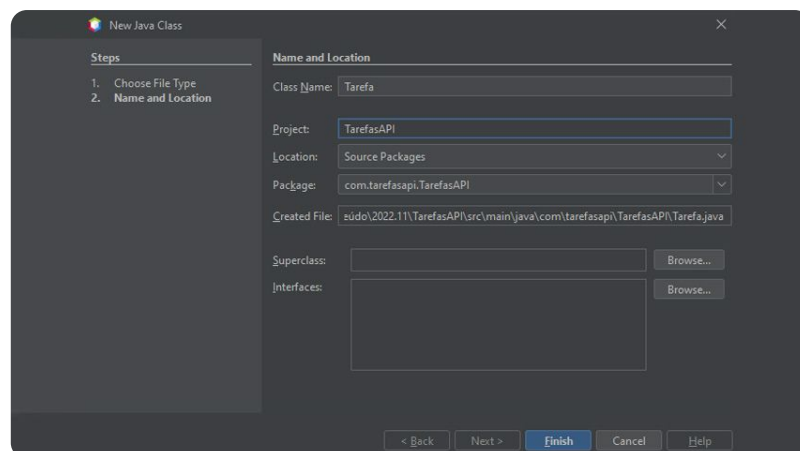
Configurações finais e geração do projeto

Agora, vamos descompactar nosso projeto e efetuar sua abertura no NetBeans, escolhendo a opção Open Project e selecionando o diretório extraído.



Abertura do projeto no NetBeans

Precisaremos de uma classe para representar o formato de dados, a qual será gerada com o nome **Tarefa**, definindo uma tarefa genérica, no pacote principal do projeto. Para criar a classe, clique com o botão direito sobre o pacote e escolha a opção New, seguida de Java Class.



Criação da classe Tarefa

Nossa classe terá apenas os atributos para definição de **código**, **título** e **descrição**, como pode ser observado na listagem a seguir.

```
java

public class Tarefa {
    public String codigo;
    public String titulo;
    public String descricao;
}
```

Com nossa unidade de informação definida, vamos acrescentar uma nova classe, com o nome **TarefaController**, ao pacote principal, utilizando o código da listagem seguinte. Essa classe será responsável pela definição da API REST de nosso sistema.

```

java
@RestController
@RequestMapping("tarefa")
@CrossOrigin("")
public class TarefaController {
    private static final HashMap tarefas =
        new HashMap<>();

    @GetMapping
    public List obterTarefas(){
        return new ArrayList<>(tarefas.values());
    }
    @PostMapping
    public void incluirTarefa(@RequestBody Tarefa tarefa){
        tarefas.put(tarefa.codigo, tarefa);
    }
    @DeleteMapping("{codigo}")
    public void excluirTarefa(@PathVariable String codigo ){
        tarefas.remove(codigo);
    }
}

```

Em termos práticos, temos apenas uma classe comum, com o repositório estático do sistema, denominado tarefas, baseado em um HashMap. Para gerenciar o repositório, a classe fornece os métodos obterTarefas, que retornam as tarefas em um ArrayList: incluirTarefa, para adicionar uma tarefa, e excluirTarefa, para remoção a partir do código.

A inversão de controle é configurada por meio de anotações, a começar por RestController, que transforma a classe em um web service RESTful; Requestmapping, que define a rota principal de acesso; e CrossOrigin, que viabiliza o acesso a partir de outros domínios.

A recepção das chamadas via **HTTP** é configurada nos métodos da classe, por meio das seguintes anotações. Confira!

### GetMapping

Essa anotação está associada ao método GET do protocolo.

### PostMapping

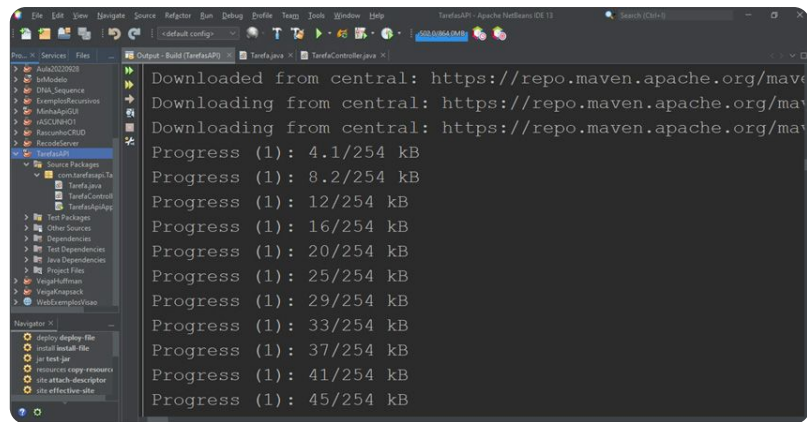
Essa anotação está associada ao método POST do protocolo.

### DeleteMapping

Essa anotação está associada ao método DELETE do protocolo.

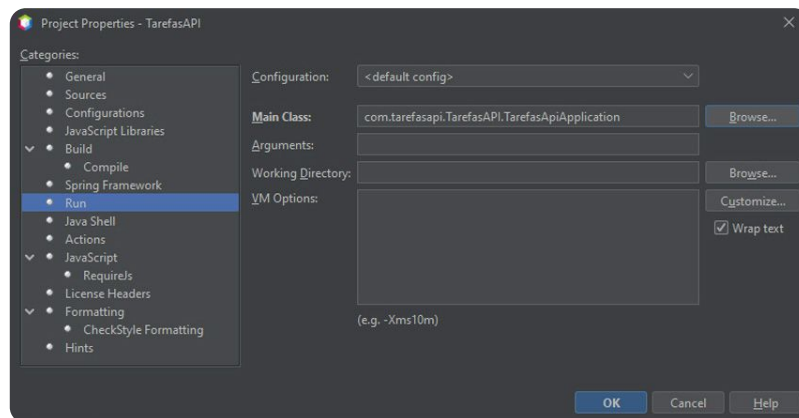
Temos ainda a anotação RequestBody no parâmetro tarefa de incluirTarefa, visando à recepção dos dados que serão enviados no corpo da requisição. A anotação PathVariable, no parâmetro código de excluirTarefa, obtém o código da tarefa que será removida a partir do segmento da rota.

Por se tratar de um projeto do tipo Maven, é necessário baixar as dependências definidas no arquivo pom.xml. Para fazer isso, clique com o botão direito sobre o projeto e escolha a opção Build with Dependencies. Veja!



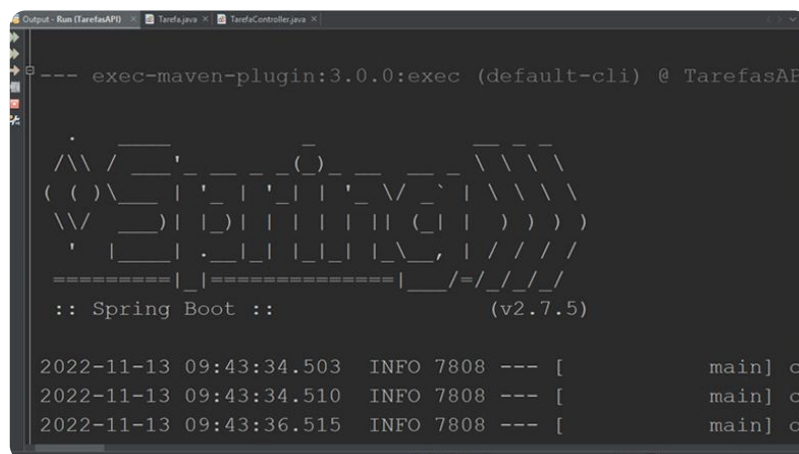
Compilação com download de dependências

No passo seguinte, escolha a classe principal do aplicativo servidor: clique com o botão direito sobre o projeto e escolha a opção Properties. Na janela aberta, navegue para a divisão Run, clique em Browse e selecione TarefasApiApplication.



Escolha da classe principal do projeto no NetBeans

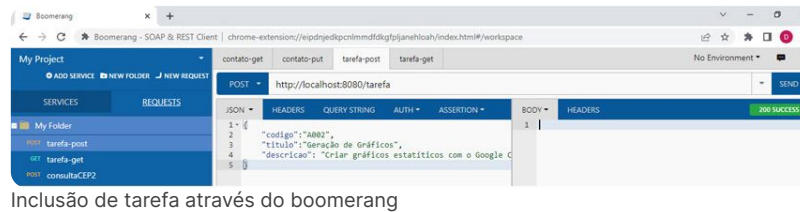
Ao trabalhar com Spring Boot, não precisamos de um servidor, como GlassFish ou JBoss, pois temos o Tomcat embarcado no próprio sistema, atuando como um aplicativo Java de linha de comando. Com tudo pronto, vamos executar o projeto, por meio da opção **Run**, e observar a saída do console na divisão Output.



Execução do aplicativo servidor

Ao trabalhar com uma API, não temos uma interface de usuário constituída, apenas um conjunto de serviços disponibilizados para outras plataformas. Para testar as APIs do tipo REST, sem a criação de um aplicativo cliente, podemos utilizar o Postman ou o plugin Boomerang, do navegador Chrome, devido à sua praticidade.

Inicialmente, devemos efetuar a inclusão de algumas tarefas no repositório, criando uma requisição pela opção New Request, escolhendo o endereço `http://localhost:8080/tarefa`, selecionando o modo Post e modificando o formato de Body para JSON. Em seguida, os dados da nova tarefa devem ser digitados. Ao clicar na opção Send, o código 200 do HTTP indicará que a operação foi bem-sucedida. Veja!

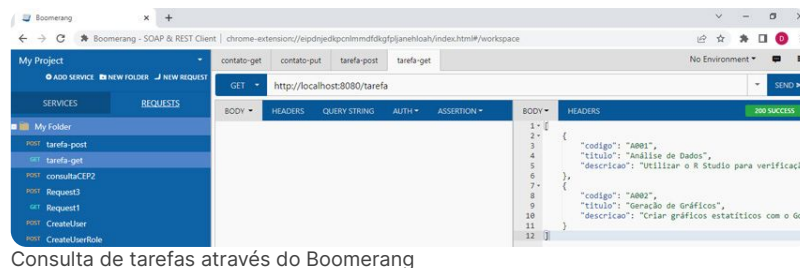


Os dados de cada tarefa devem ser fornecidos no formato JSON, como no exemplo da listagem seguinte. Nesse caso, é necessário fornecer os valores para todos os atributos da classe.

```
plain-text

{
  "codigo": "A002",
  "titulo": "Geração de Gráficos",
  "descricao": "Criar gráficos estatísticos com o Google Charts"
}
```

Após inserir algumas tarefas, repetindo o processo anterior com a modificação dos dados que serão enviados, vamos criar uma requisição para o mesmo endereço, em modo GET, e corpo vazio. Ao clicar em Send, devemos ter, além do indicador de sucesso, o retorno de todas as tarefas enviadas, na forma de um vetor JSON.



## Interoperabilidade

Em meados da década de 1990, com a expansão dos sistemas computacionais, começaram a ser oferecidas diferentes formas padronizadas para armazenar, recuperar e processar dados. Com base em conversores de dados, os arquivos de um fabricante eram convertidos para determinado formato, de modo que outro fabricante pudesse ler. Esse processo surgiu da necessidade de compartilhamento de dados entre aplicativos distintos.

Atualmente, a **interoperabilidade** é garantida pela utilização de formatos de dados padronizados, como XML e JSON, porque eles adotam modo texto, podendo ser processados facilmente em qualquer plataforma cliente. Nossos web services, tanto SOAP quanto RESTful, definem APIs com serviços baseados nessa estratégia de interoperabilidade, muito adequada ao ambiente de rede, em que temos grande heterogeneidade e necessidade de transparência frente aos firewalls.

## Interoperabilidade

Ao longo do tempo, diferentes formas de compartilhamento de informação surgiram, tendo como elementos relevantes os bancos de dados e protocolos de rede padronizados. Bancos de dados, como MySQL e Oracle, podem ser acessados por plataformas distintas simultaneamente, e protocolos, como RPC (remote procedure call), viabilizaram a execução de procedimentos em máquinas remotas, não importando a plataforma do cliente. Surgiu, então, o conceito de interoperabilidade, que é capacidade de um sistema de oferecer recursos e serviços para outros sistemas, com independência de plataforma, baseada em meios padronizados de comunicação. A criação de uma API deve ser guiada pela interoperabilidade, já que o objetivo é oferecer funcionalidades para cliente distintos a partir do sistema existente.

## Clientes Retrofit para web services

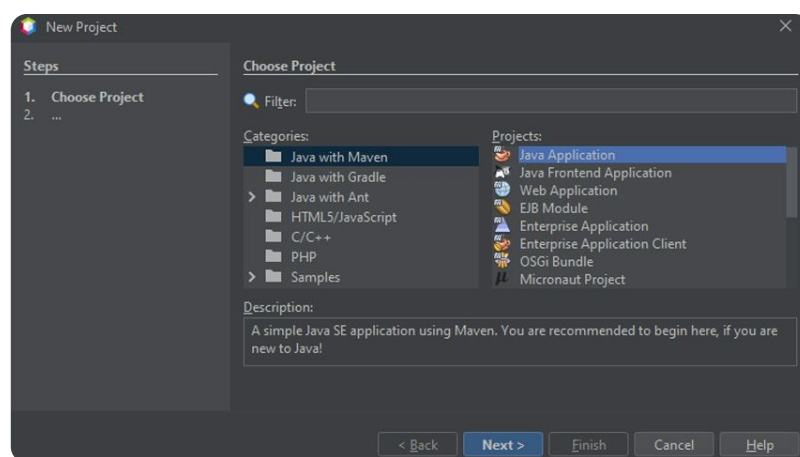
O uso de métodos como PUT e DELETE, além da adoção de JSON para os dados, pode dificultar testes tradicionais, baseados em formulários HTML. Entretanto, existem muitas soluções viáveis, como o uso do aplicativo Postman ou de chamadas AJAX a partir de bibliotecas JQuery. Ainda assim, a forma mais eficaz para verificar a funcionalidade da solução é a criação de um aplicativo Java, com a transformação dos dados recebidos em objetos.



### Dica

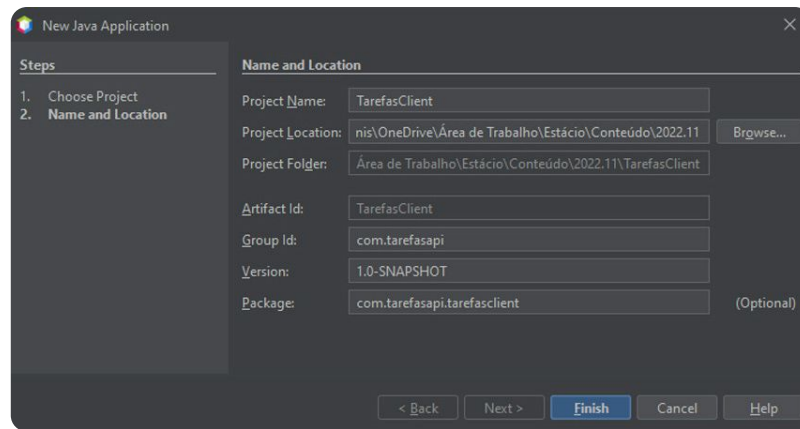
As bibliotecas como Axios, no ambiente NodeJS, e Retrofit, para a plataforma Java, permitem o uso de todos os métodos do HTTP de forma simplificada. Mais que isso, o Retrofit trabalha com diversos conversores, alguns deles capazes de efetuar o mapeamento automático dos dados no formato JSON para objetos Java.

Vamos iniciar a definição de nosso cliente com a criação de um projeto no NetBeans, com base no modelo Java Application, a partir do grupo Java with Maven, como pode ser observado na imagem seguinte. Lembrando que esse tipo de projeto tem todas as configurações no arquivo pom.xml.



Escolha de projeto baseado no Maven

Utilizaremos o nome TarefasClient e grupo com.tarefasapi, como pode ser observado a seguir.



Configuração do projeto TarefasClient

O próximo passo será o acréscimo das dependências do Retrofit e do Jackson Converter no arquivo pom.xml, que pode ser encontrado na divisão Project Files. Vamos observar agora o conteúdo do arquivo com as inclusões necessárias.

plain-text

```
4.0.0
com.tarefasapi.tarefasclient
TarefasClient
1.0-SNAPSHOT

UTF-8
17
17
com.tarefasapi.tarefasclient.TarefasClient

com.squareup.retrofit2
retrofit
2.9.0

com.squareup.retrofit2
converter-jackson
2.9.0
```

Utilizando o Jackson Converter, é possível realizar a conversão dos dados do formato JSON para objetos Java. Como estamos no ambiente Java, podemos copiar a classe Tarefa, definida no projeto de nosso servidor, para o pacote com.tarefasapi.tarefasclient do novo projeto.

Se utilizássemos Kotlin, definiríamos uma estrutura do tipo data class. No Java Script, poderíamos usar diretamente o formato JSON, ou seja, os dados são capturados em uma estrutura do cliente.

Após copiar a classe de entidade, vamos definir uma interface, com o nome TarefaAPIClient, de acordo com a seguinte listagem:

```

java

public interface TarefaAPIClient {
    @GET("tarefa")
    Call< obterTarefas();

    @POST("tarefa")
    Call incluirTarefa(@Body Tarefa tarefa);

    @DELETE("tarefa/{codigo}")
    Call excluirTarefa(@Path("codigo") String codigo);
}

```

As anotações GET, POST, PUT e DELETE definirão o tipo de método HTTP utilizado, além da rota de acesso ao recurso, colocada entre parênteses. No caso de caminhos com identificação do recurso, como na obtenção de uma tarefa pelo código, o trecho identificador é colocado entre chaves, sendo relacionado a um parâmetro do método através da anotação Path, algo que pode ser observado no método excluirTarefa.

Além das características citadas, para enviar dados JSON no corpo da requisição, basta definir um parâmetro do tipo da entidade, no caso Tarefa, anotado com Body, e todos os métodos encapsulam o retorno em um objeto genérico Call. O encapsulamento do tipo de retorno serve para trabalhar tanto no modelo síncrono, com a invocação feita via execute, quanto de forma assíncrona, por meio de uma chamada enqueue e a criação de um objeto Callback.

Para que a interface de serviços seja implementada de forma automática, deve ser instanciado um cliente Retrofit, onde o conversor de JSON será especificado. A criação do cliente segue o padrão de desenvolvimento Builder, permitindo efetuar diversas configurações de forma muito dinâmica e simples.

Agora, vamos modificar o conteúdo da classe principal, com o nome TarefasClient, para este:

```

plain-text

public class TarefasClient {
    public static void main(String[] args) throws IOException {
        Retrofit cliente = new Retrofit.Builder()
            .baseUrl("http://localhost:8080")
            .addConverterFactory(JacksonConverterFactory.create())
            .build();
        TarefaAPIClient tarefaCli = cliente.create(TarefaAPIClient.class);
        for(Tarefa t: tarefaCli.obterTarefas().execute().body()){
            System.out.println(t.codigo+" :: "+t.titulo);
            System.out.println("==>"+t.descricao);
            System.out.println();
        }
    }
}

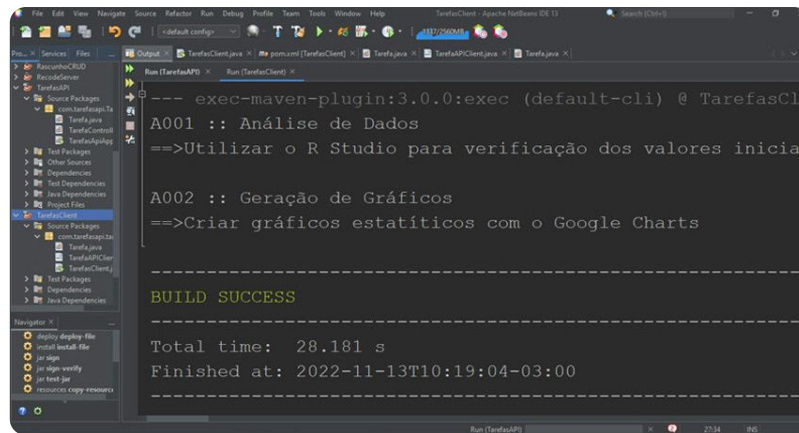
```

Nossas configurações para o Retrofit incluem o endereço de base das chamadas REST e a classe para conversão dos dados, que, no caso, é a JacksonConverterFactory. Com o cliente instanciado, a interface é implementada a partir de uma chamada do tipo create.

As chamadas efetuadas para o serviço são muito semelhantes à execução padrão de métodos Java, mas todas seguidas de execute, pois estamos utilizando o modo síncrono dos objetos do tipo Call. Temos ainda a captura do retorno, já na forma de objetos e coleções Java, encadeando uma chamada para body.

Considerando que nosso servidor ainda esteja em execução, com os dados adicionados por meio do plugin Boomerang, podemos executar o novo projeto, alcançando um resultado semelhante ao apresentado a seguir.





Execução do cliente no NetBeans

Para incluir uma tarefa, teríamos um trecho de código similar a este:

```
java

Tarefa t1 = new Tarefa();
t1.codigo = "A001";
t1.titulo = "Análise de Dados";
t1.descricao = "Utilizar o R Studio para verificação dos valores";
tarefaCli.incluirTarefa(t1).execute();
```

Nesse caso, a exclusão seria ainda mais simples, observe!

```
java

tarefaCli.excluirTarefa("A001").execute();
```

Analisando os exemplos, chegamos à conclusão de que o uso de Retrofit permite a criação de clientes REST de forma simples, abstraindo completamente de detalhes como o conhecimento acerca do protocolo HTTP, bem como do processo de conversão entre os formatos JSON e Java.

## Atividade 1

Com o Spring Boot, podemos definir uma API REST de forma simples, sem a necessidade de um servidor, como GlassFish ou JBoss. Por meio de anotações, a implementação de toda a funcionalidade relacionada à exposição de serviços e captura de requisições é delegada para o framework Spring, segundo uma técnica denominada:

A

interoperabilidade.

B

inversão de controle.

C

polimorfismo.

D

sobrecarga.

E

injeção de dependência.



A alternativa B está correta.

Ao trabalhar com um framework, temos um container que se responsabiliza pela execução de boa parte do processo, o que é denominado inversão de controle. Podemos ainda utilizar algum recurso do container na programação através da injeção de dependência. Interoperabilidade é a capacidade que um sistema tem de oferecer serviços para plataformas externas. Polimorfismo representa a habilidade de modificar a programação de um método herdado. E sobrecarga é a possibilidade de criar funções e método com o mesmo nome, desde que os parâmetros sejam diferentes.

## Consumindo nossa API REST utilizando Retrofit

Vimos anteriormente, e de forma separada, como codificarmos uma API REST e como construirmos uma aplicação cliente, em Java, para o consumo desse tipo de API. Nessa atividade, aplicaremos essas duas tarefas em conjunto.

Assista ao vídeo e coloque em prática o conhecimento adquirido na programação de uma API REST e na criação de um aplicativo cliente em Java. Por meio de um exercício prático, exploraremos a integração dessas atividades ao desenvolver uma API e um aplicativo cliente para seu consumo.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Roteiro de prática

Nessa prática, utilizaremos a API REST desenvolvida anteriormente, além dos conceitos e códigos demonstrados para a criação de uma aplicação Java utilizando a biblioteca Retrofit. Em termos de ferramenta, você precisará utilizar a IDE Spring Tool Suite, para rodar a API REST, e a NetBeans, para criação do cliente.

Agora, vamos ao passo a passo!

- Elabore um novo projeto no NetBeans com base no modelo Java Application.
- Insira os dados da nova aplicação conforme sua preferência.
- Adicione, no pom.xml, as dependências relativas à biblioteca Retrofit.
- Crie uma classe chamada Country, com os mesmos atributos e métodos (get e set) existentes na API REST. Lembre-se de não incluir nessa nova classe nenhuma das anotações utilizadas na API.
- Crie uma interface chamada CountryClient e, dentro dela, o método obterPaíses(), que deverá ser anotado com GET("countries") – onde countries é o nome do recurso configurado no Controller, na API REST.
- Na classe principal, método main, crie a instância da classe Retrofit e faça a chamada ao método obterPaíses().
- Faça um laço for que percorra os dados recuperados a partir da API REST (dados dos países) e os imprima no console utilizando System.out.println.

A resolução desse serviço pode ter mais de uma resposta. Cabe avaliar o resultado do teste no console, onde deverão ser impressas as informações de todos os países, conforme retornado pela API REST. Veja um exemplo de resolução nos próximos códigos.

## Resultado da execução da aplicação no console

Agora, veja o resultado da execução da aplicação.

```
java  
  
name: Spain  
capital: Madrid  
currency: EUR  
population: 46704314  
  
name: Poland  
capital: Warsaw  
currency: PLN  
population: 38186860  
  
name: United Kingdom  
capital: London  
currency: GBP  
population: 63705000
```

## Classe Country.java

Para explorar o código da classe 'Country.java', siga adiante.

java

```
public class Country {
    private Integer idCountry;
    private String name;
    private String capital;
    private String currency;
    private Integer population;

    public Integer getIdCountry() {
        return idCountry;
    }

    public void setIdCountry(Integer idCountry) {
        this.idCountry = idCountry;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCapital() {
        return capital;
    }

    public void setCapital(String capital) {
        this.capital = capital;
    }

    public String getCurrency() {
        return currency;
    }

    public void setCurrency(String currency) {
        this.currency = currency;
    }

    public Integer getPopulation() {
        return population;
    }

    public void setPopulation(Integer population) {
        this.population = population;
    }
}
```

## Interface CountryClient.java

Para explorar o código da classe 'Interface CountryClient.java', prossiga.

java

```
public interface CountryClient {
    @GET("countries")
    Call<> obterPaises();
}
```

## Método main

Agora, acompanhe o código da classe 'Método main'.

```
java

public static void main(String[] args) throws IOException {
    Retrofit cliente = new Retrofit.Builder()
        .baseUrl("http://localhost:8080")
        .addConverterFactory(JacksonConverterFactory.create())
        .build();
    CountryClient countryCli = cliente.create(CountryClient.class);
    for(Country c: countryCli.obterPaises().execute().body()){
        System.out.println("name: " + c.getName());
        System.out.println("capital: " + c.getCapital());
        System.out.println("currency: " + c.getCurrency());
        System.out.println("population: " + c.getPopulation());

        System.out.println();
    }
}
```

## Faça você mesmo!

O uso da biblioteca Retrofit facilita o trabalho de consumo de APIs REST e demais tarefas relacionadas a esse processo. Qual é a função das linhas de código a seguir?

```
java

Retrofit cliente = new Retrofit.Builder()
    .baseUrl("http://localhost:8080")
    .addConverterFactory(JacksonConverterFactory.create())
    .build();
CountryClient countryCli = cliente.create(CountryClient.class);
List listaPaises = countryCli.obterPaises().execute().body();
```

A

Realizar a injeção de dependências da entidade CountryClient.

B

Fazer a inversão de controle, permitindo à classe onde os códigos acima foram inseridos tomar conta da aplicação como um todo.

C

Converter os dados da classe Country na classe JacksonConverterFactory.

D

Instanciar um objeto da biblioteca Retrofit, utilizá-lo na chamada à API REST e converter os dados retornados por ela, através do JacksonConverterFactory, em instâncias da classe Country.

E

Imprimir no console os dados de um array.



A alternativa D está correta.

A biblioteca Retrofit permite a realização de requisições a APIs REST e a conversão dos dados retornados em instâncias de classes Java. Tal processo pode ser visto no código apresentado.

## Considerações finais

### O que você aprendeu neste conteúdo?

- Como desenvolver um web service SOAP utilizando Java.
- Como testar um web service SOAP.
- Como desenvolver uma API REST utilizando Java.
- Como testar uma API REST usando Java e a ferramenta Insomnia.

### Explore +

Confira as indicações que separamos especialmente para você!

Para saber mais sobre os assuntos tratados neste conteúdo, pesquise sobre especificações dos padrões WS-\*, no site do **W3C**.

Pesquise **REST with Spring tutorial** (Tutorial REST com Spring), que fornece um guia completo, que poderá servir como material de apoio, de criação de uma API REST com Spring.

Para conhecer outras anotações do Spring, pesquise **Hibernate Many to Many: annotation tutorial** (Tutorial e anotações de muitos para muitos).

### Referências

ACOBSON, D.; BRAIL, G.; WOODS, D. **APIs: A Strategy Guide**. California: O'Reilly, 2012.

KALIN, M. **Java Web Services: up and running**. 2. ed. California: O'Reilly, 2013.

PRESSMAN, R.; MAXIM, B. **Engenharia de software: uma abordagem profissional**. Porto Alegre: McGraw-Hill: Artmed, 2016.

SPRING FRAMEWORK. **Spring Web Services**. Consultado na internet em: 4 out. 2023.

W3C. **W3C Working Group Note 11**. In: W3C – Web Services Architecture. Publicado na internet em: fev. 2014.