

EBOOK DIGITAL
POR WENDERSON ANJOS

Guia de API RESTful com **SPRING BOOT** EM JAVA



SUMÁRIO

INTRODUÇÃO	3
1. O QUE É O SPRING FRAMEWORK?	5
1.1. Inversão de Controle	7
1.2. Injeção de Dependências	8
1.3. Componentes e Suas Anotações	9
1.4. Scopes e Suas Anotações	10
1.5. Starters do Spring Boot	10
2. INSTALAÇÃO DAS FERRAMENTAS	12
2.1. Instalando a JRE e o JDK	12
2.2. Instalando o Editor VSCode	17
2.3. Instalando o Banco de Dados PostgreSQL	20
3. DESENVOLVENDO UMA API USANDO SPRING MVC	33
3.1. Projeto Maven com Spring Initializr	33
3.2. Projeto Spring sem Dependências	38
3.3. Projeto Spring com Dependências	50
3.4. Aplicação de Cadastro de Usuários.....	63
3.5. Aplicação RESTful com Spring MVC	72
CONCLUSÃO	89
REFERÊNCIAS	90

INTRODUÇÃO

Nos dias atuais, com a alta demanda no mercado por softwares de maior qualidade, se tornou cada vez mais recorrente as práticas mais modernizadas na construção de software. Até um certo tempo atrás, o desenvolvimento de softwares era algo mais “monolítico” e “imperativo”, isto é, você apenas dava ordens ao computador com uma sequência de comandos sem uma separação clara de responsabilidades, mesmo que tudo poderia ser dividido em rotinas/funções, ainda sim era uma tarefa árdua dar manutenção em um código onde tudo ficava em um mesmo lugar.

Este tipo de aplicação usava o paradigma estruturado, que consistia em dividir o programa em estruturas, como: Sequência, Decisão e Repetição. Com o avanço da tecnologia junto as exigências de clientes para maior produtividade, nasceu um outro paradigma chamado “Orientação a Objetos”, abreviado para “OO”. A orientação a objetos resolve boa parte dos problemas quando se trata de *aproximar* o desenvolvimento de aplicações mais da realidade humana, usando conceitos de proteção, divisão de responsabilidades e melhor organização do código.

Linguagens como Java e C#, que utilizam em sua essência a orientação a objetos, vieram pra ficar e são frequentemente utilizadas para aplicações comerciais, jogos e demais outros softwares. Mesmo assim, sabemos que na tecnologia tudo se evolui para prover maior agilidade e qualidade em se construir código, desta forma, desenvolvedores criam novos “Frameworks” que são ambientes com uma vasta de coleção de bibliotecas que permite gerenciar e automatizar o processo de desenvolvimento, permitindo que outros programadores reutilizem o que já está pronto, sem precisar reinventar a roda, trazendo melhor custo-benefício para os clientes a longo prazo.

É possível também separar uma só aplicação em camadas como fazemos no modelo MVC (Model-View-Controller), dividindo responsabilidades da interface com o cliente da lógica de negócios, no entanto, não para por aí, podemos ainda criar várias aplicações completamente separadas e independentes mas que se comunicam entre si. Este é um conceito muito trabalhado na área de micros serviços. Esta comunicação se dá pelo uso de Web APIs (Interface de Programação de Aplicações), que são sistemas

acessíveis que definem serviços em uma aplicação servidora recebendo requisições e fornecendo dados para aplicações que requisitam.

As APIs desempenham um papel crucial nas aplicações de software e através delas os chamados “EndPoints” podem ser disponibilizados pelos usuários que são URIs customizadas para cada serviço e acessado pelo usuário, podendo realizar cadastros, atualizações, entrega de dados e recursos úteis para o front-end. É visando este conceito que utilizaremos neste E-book o Spring Web na linguagem Java para automatizar o processo de criação de uma API. Nossa API seguirá os conceitos REST que é um padrão muito utilizado para APIs que recebem e enviam dados em JSON.

A missão deste E-book é ensinar da maneira mais didática possível a criação de uma API de um pequeno game, onde consiste em cadastrar e recuperar informações de magos, bruxas e monstros, aproveitando o dia de hoje – Halloween. Para isto, começaremos no primeiro capítulo apresentando os conceitos do Spring Framework, suas vantagens de utilização e como isto pode ajudar na produtividade de um software.

Após isto, mostraremos o passo a passo de instalação das ferramentas no segundo capítulo, ferramentas estas necessárias para a construção da nossa API RESTful, como o JDK (Java Development Kit), O banco de dados PostgreSQL e a IDE VSCode. E por fim, apresentaremos no terceiro capítulo a construção por partes da nossa aplicação, começando por gerar o projeto Maven por um site que facilita esta criação e começar a filtrar algumas das dependências iniciais.

Bons estudos!

1. O QUE É O SPRING FRAMEWORK?

O Spring Framework é um ecossistema de anotações do JAVA e bibliotecas que fornecem a otimização no desenvolvimento de software para maior produtividade. Ele contém alguns fundamentos que é importante conhecer, como o *IoC (Inversion of Control)* que é o núcleo do Spring, A *injeção de Dependências* utilizado para automação das instâncias de objetos e anotações importantes como Beans, Autowired e Scopes que visam aplicar de maneira eficaz os fundamentos citados.

Este framework open source é baseado nos principais padrões de projetos que ajudam a organizar o código e o desenvolvimento de aplicações com as boas práticas de acordo com as necessidades. Estes padrões de projetos é a inversão de controle e a injeção de dependências que abordaremos mais adiante. O Spring é composto por módulos que reduzem a complexidade de desenvolvimento de aplicações simples e corporativas, veja a figura abaixo apresentando os módulos do Spring:

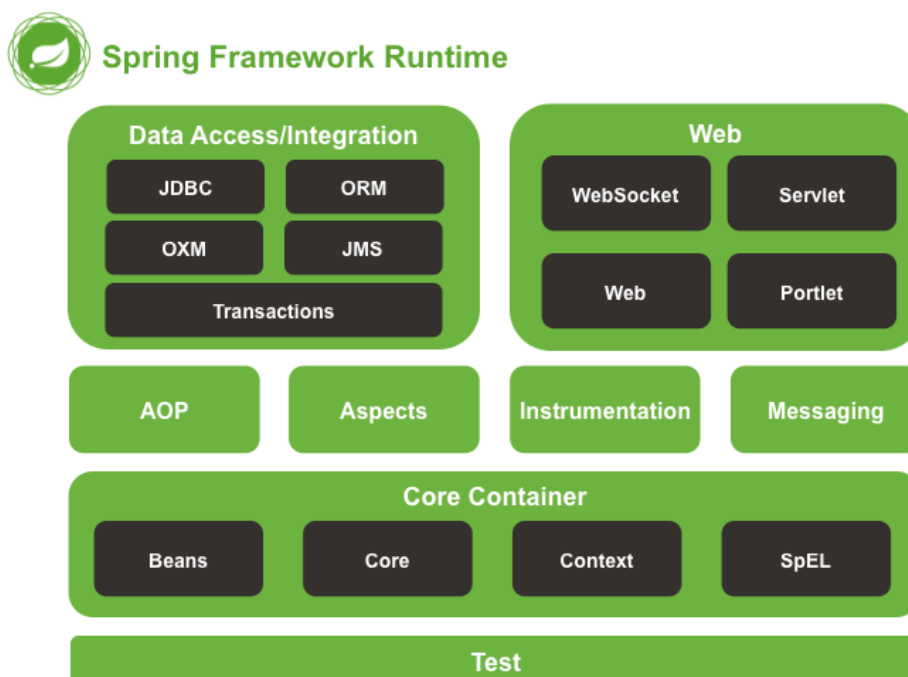


Figura 1 - Overview do Spring framework

Na imagem citada são apresentados os módulos do Spring framework. Podemos ver que existem um gama de funcionalidades que são utilizadas por este framework, uma delas é na parte de dados que integra o JDBC que é a forma padrão JAVA para

comunicação com o banco de dados, o ORM que mapeia objetos para tabelas relacionais e demais outras bibliotecas.

Na parte Web encontramos recursos como WebSocket, Servlet, Web e Portlet, recursos estes muito utilizados para softwares que usam navegadores de internet, ou seja, aplicações web. Dentre estas aplicações, podemos criar as Web APIs e Servidores de Back-End para as Web UI (User Interface para Web). Também temos os módulos AOP e Aspects que permitem trabalhar com *programação orientada a aspectos* possibilitando desacoplar de maneira limpa funcionalidades que precisam ser separadas, além de instrumentações e messagings que integra o Spring em servidores, ex.: Tomcat. E mapeia mensagens de métodos e abstração de chaves, respectivamente.

O módulo principal do Spring é o seu “Core” sendo o Núcleo base do framework. É através deste Core que os padrões de projeto são aplicados de maneira automatizada. O Núcleo é um container do Spring que é inicializado pelo seu contexto e através do BeanFactory gerencia objetos Beans ou Components que são tipo de objetos que não precisam ser instanciados explicitamente, isto é, ocorre uma injeção de dependências. O Container núcleo do Spring possibilita trabalhar com uma implementação flexível e dinâmica, uma vez que o Core elimina as altas dependências, criando baixo acoplamento e alta coesão na aplicação.

A coesão em aplicações é um fundamento onde diversos contextos contém um significado semântico de conexão, isto é, todos os contextos podem se comunicar e se “interligar” sem um danificar o outro, possibilitando máxima interoperabilidade. Já o acoplamento, se refere as dependências de cada contexto, sendo a influência que um objeto ou parte da aplicação interfere em outra parte. Isto significa que um software com baixo acoplamento contém divisões independentes muito bem definidas em suas responsabilidades, garantindo que um contexto da aplicação não quebre as demais.

Por último, o módulo de testes ajuda o desenvolvedor a aplicar testes no sistema, como testes unitários e de integração seguindo todos estes princípios, uma vez que certos tipos de testes dependem de certos contextos, o Spring gerencia estes contextos.

1.1. Inversão de Controle

Inversion of Control ou IoC, redireciona o fluxo de execução de um código retirando o controle sobre ele de forma parcial, isto é, ainda é possível controlar o código, no entanto, para aspectos mais essenciais da solução. O controle é transferido para um container, que intermediará todas as configurações, instanciações e alocação de recursos para a nossa aplicação. O principal objetivo da IoC é minimizar o acoplamento de código.

Antes desse conceito, quando criávamos uma aplicação do zero, precisaríamos configurar cada propriedade, instanciar cada objeto e depois utilizá-los, o que consumia bastante tempo apenas para “iniciar” o desenvolvimento de um software. Com o IoC do Spring, não mais isto é necessário, pois o Core Container mencionado anteriormente utiliza este fundamento para gerenciar todas estas instanciações, resultando em um maior foco na solução do negócio do cliente.

O principal aspecto do IoC é delegar o controle parcial para o container, isto significa que cada módulo Spring para diversos contextos como Dados, Web, etc. Terá seu container de configuração e instanciação, no qual o núcleo controlará, tanto na inicialização, quanto na execução do projeto. O Spring também utiliza muito o conceito de “Interfaces” que são contratos da orientação a objetos que não tem implementação, dizendo ao desenvolvedor **Quais** são os métodos obrigatórios que precisam ser implementados.

Normalmente, interfaces são criadas quando já existe alguma implementação interna, no entanto, possibilitando o “Override” (Sobre escrita) destas implementações ao longo das necessidades do projeto. As interfaces servem para encapsular as funcionalidades de uma aplicação, fornecendo ao programador apenas “O que” fazer, e não “Como” fazer, isto é, um contrato. É através das Interfaces que o Spring consegue manter seu baixo acoplamento de código pois o desenvolvedor não chama diretamente os métodos das implementações, e sim das interfaces, o que minimiza a “dependência” desta implementação.

1.2. Injeção de Dependências

Também conhecido como *DI (Dependency Injection)*, é um padrão de desenvolvimento que serve para “injetar” um recurso, por exemplo: Uma instanciação de um objeto. Quem injeta estes recursos é o próprio container, no entanto, precisamos dizer ao container quais são os objetos pra ser injetados, desta forma, o container procurará qual é a sua dependência.

No tópico anterior foi dito sobre a IoC – Inversão de Controle – que utiliza as interfaces para encapsular as implementações e também criar os objetos na inicialização, como configurações e instanciações, com o objetivo de apenas usarmos. Isto nos diz que a aplicação não sabe de fato qual é a implementação de um método, o que significa a baixa dependência (baixo acoplamento). No entanto, quando vamos criar um objeto, primeiro precisamos definir o seu tipo e depois alocar a memória para este objeto com a cláusula **new**, com a injeção de dependências isto é automatizado.

A IoC está fortemente ligado ao DI, uma vez que a IoC gerencia seus recursos usando a DI, entretanto, a DI pode ser iniciada por nós desenvolvedores, uma vez que se um objeto é criado na aplicação, o desenvolvedor não mais precisa instanciá-lo, já que a injeção de dependência fará isto de forma automática. Para tornar possível a DI em uma classe específica, uma vez que a classe se tornará um objeto, é preciso informar ao Spring se a classe será um componente auto-injetável.

Esta maneira de informar ao Spring é através de anotações dizendo que uma classe será um Component ou um Bean. E assim, o Spring saberá que aquela classe será gerenciada pelo container através da IoC, no entanto, pode ocorrer de quisermos gerenciar nós mesmos em alguns cenários, então o Spring possibilita outra anotação para dizer em qual momento quisermos a injeção e isto é feito usando os Autowireds.

Tanto o Bean, quanto o Component são componentes auto-gerenciáveis pelo container da IoC, porém eles contêm uma diferença simples que será visto no próximo tópico. Também precisamos conhecer os Scopes, que se dividem em “Scopes de Beans” e “Scopes HTTP”, no qual eles referenciam alguns dos principais padrões de projeto, veremos sobre isto no próximo subcapítulo.

1.3. Componentes e Suas Anotações

O Spring trabalha com anotações JAVA específicas para informar ao container quais classes serão componentes auto injetáveis da aplicação. Um componente é um objeto que será gerenciado pela IoC seguindo os atributos da injeção de dependência. Eles podem ser de dois tipos: Bean & Component. Para criar um Bean é preciso inserir a anotação **@Bean** em cima de um método que retorna um tipo de uma Classe, enquanto que um Component é preciso adicionar a anotação **@Component** em cima de uma classe. Qual é a diferença dos dois?

A diferença do Bean com o Component é que o Component é usado quando o desenvolvedor tem acesso ao código-fonte da classe, isto é, quando ele conhece a implementação. Já o Bean é quando não se tem este acesso ao código, como em uma biblioteca externa. Vamos colocar como exemplo uma classe que precisamos converter um dado em JSON chamado *JSONConversor*, como é uma classe que estamos implementando na aplicação, ela precisa ter a anotação **@Component** para ser auto-instanciada usando os Autowireds.

O Bean é utilizado em classes que vem de bibliotecas que não conhecemos o seu código-fonte, um outro exemplo disso é a classe *CommandLineRunner* que é utilizada em uma aplicação Spring Boot para inicializar um código que é gerenciado pelo Spring. Desta forma, é criado um método que será inicializado (como um *Main* da aplicação) que receberá o tipo *CommandLineRunner*, portanto, este método receberá a anotação **@Bean**, permitindo que o Spring gerencie a injeção de dependências tanto no próprio método, quando no corpo do método. Isto significa que quando passamos um componente como argumento desse método, no corpo desse método este componente será auto injetável (instanciado automaticamente) sem o uso da cláusula **new**.

Após criado seus componentes, é necessário indicar onde deverá ocorrer esta injeção, então o Spring disponibiliza a anotação **@Autowired** para colocarmos antes de um atributo daquele tipo, indicando que aquele atributo quando for utilizado nesta classe, será instanciado automaticamente, sem precisarmos inserir **new**. O processo de injeção pelo container é feito *byName* que é buscar um método set que corresponde ao nome do Bean, *byType* que procura o tipo da classe para a inclusão e *byConstructor* que utiliza o construtor como forma de incluir a dependência.

1.4. Scopes e Suas Anotações

Os escopos dizem em qual contexto um objeto da aplicação pode estar, aplicando os padrões de projeto do Spring, como Singleton e Prototype. No conceito standalone, o singleton é um padrão no qual a aplicação só utiliza uma instância de um dado objeto, isto significa que no início do programa, é criado um novo objeto pelo *new* caso não exista uma instância dele, mas se existir, a instância anterior é retornada, garantindo que o objeto só tenha uma instância.

No caso do Prototype, um novo objeto é criado a cada utilização, sendo uma instância a cada vez que o uso do objeto é exigido. Em variados contextos, podemos utilizar tanto o Singleton, quanto o Prototype, de acordo com as necessidades do projeto. O Spring também administra este escopo de objetos através da anotação **@Scope**, passando como parâmetro o valor “prototype” ou “singleton”, como **@Scope(“prototype”)**.

A anotação Scope é passado em cima do método que utiliza a classe, informando ao Spring que se for do tipo *singleton*, a instância do objeto será injetada apenas na primeira vez, usando uma mesma instância toda vez que o objeto for solicitado e se for *prototype*, a cada solicitação do objeto, o Spring irá criar um novo objeto na memória.

Já no escopo do tipo *HTTP*, temos o **Request** onde um Bean é criado a cada requisição HTTP, fazendo com que os objetos existam apenas durante o ciclo de requisição; O **Session** que cria um Bean para a sessão de usuário, armazenando estados do usuário durante esta sessão; E o **Global** que gera um Bean para todo o contexto da aplicação, ou seja, também chamado de *Application Scope*, permite compartilhar o objeto em toda a aplicação.

1.5. Starters do Spring Boot

Um mecanismo utilizado frequentemente no gerenciamento de dependências Maven/Gradle são os **Starters**. Os starters são inicializadores de um determinado módulo ou biblioteca externa gerenciado pelo Spring. Por exemplo, para trabalhar com banco de dados, utiliza-se o JPA como dependência de biblioteca, então incluímos no arquivo XML

o starter Spring Boot do JPA; se vamos utilizar testes unitários, utilizamos o starter da dependência “Test” do Spring boot, e assim por diante.

Inicialmente, focaremos no starter do JPA no 3ª capítulo e depois veremos sobre Spring Web. O JPA (Java Persistence API) é uma coleção de interfaces para acesso de funções de banco de dados. O uso de interfaces em orientação a objetos advém do conceito mencionado sobre inversão de controle e injeção de dependências, no qual o Spring gerencia todos os objetos e encapsula as implementações.

As implementações dos métodos nas interfaces do JPA utilizam o hibernate, que é uma biblioteca que foi muito utilizada para gerenciamento de consultas ao banco. No Java tradicional, quando não usamos JPA e Hibernate, a forma de persistência de dados é utilizando statements e prepareStatements que são métodos para fornecer as strings SQL para consulta e atualização de tabelas do banco de dados.

As formas de conexões também eram por meio de classes robustas e vários passos, passando a URL do JDBC, que é a biblioteca padrão do Java para se comunicar com o SGBD (Gerenciador de Banco de Dados). Com a adoção do Spring, estas classes de conexão não mais são necessárias, pois bastaríamos colocar as strings de conexão diretamente no arquivo .properties e o Spring irá automaticamente gerenciar esta conexão.

Com o Spring JPA é possível mapear as tabelas do banco através do ORM, que captura as classes e suas instâncias como uma forma de “atualização” de uma tabela, sem mesmo precisar utilizar o SQL de forma manual. Este é o mapeamento objeto-relacional, no qual um objeto se transforma em tabelas relacionais. O JPA fornece apenas um contrato de métodos que deveríamos implementar, para salvar, deletar ou buscar dados, no entanto, esta implementação já é feita pelo Hibernate. Neste contexto, nos preocuparíamos apenas em modelar o domínio de negócio, que são as “entidades” da aplicação.

O Spring MVC já é um outro starter que veremos com mais detalhes no terceiro capítulo. Ele é responsável por gerenciar as requisições HTTP de aplicações RESTful, que é onde expomos nossos EndPoints acessíveis de forma externa. Trabalhamos com anotações do REST para definir quais métodos serão acessados na API.

2. INSTALAÇÃO DAS FERRAMENTAS

Neste capítulo iremos apresentar a instalação das ferramentas necessárias para iniciar nossa aplicação. Começaremos com a instalação do Java 8, que é uma versão recomendada para compatibilidade da maioria das aplicações. Também utilizaremos versões do Spring e do Maven (gerenciador de dependências) que compatibilizem com esta versão do JAVA, possibilitando o trabalho de forma mais fluída e simples.

Em seguida, iremos instalar o editor VSCode, que também é uma IDE de programação que conta com novos recursos integrados, como sugestão de código inteligência, auto complete e desenvolvimento Java. O VSCode é uma alternativa leve e eficaz para desenvolvimento fluído de aplicações, que proporciona maior produtividade. É recomendado a instalação do **Extension Pack For Java** no gerenciador de extensões do VSCode, veremos isto mais adiante.

Por último, mostraremos a instalação passo a passo do banco de dados PostgreSQL, além do PgAdmin de forma integrada que é uma interface intuitiva para criar tabelas do banco de dados e visualizar dashboard interativos e estatísticas de consultas. Uma outra alternativa que será mostrada é o DBeaver, onde baixaremos a versão Community que é gratuita e que também será possível trabalhar com o PostgreSQL e o MySQL.

2.1. Instalando a JRE e o JDK

Para desenvolvimento de aplicações em Java, é necessário instalar o JDK (Java Development Kit), que é um kit de desenvolvimento de softwares em JAVA, contendo uma vasta bibliotecas e pacotes de classes. Instalaremos a JRE 8 para motivos de compatibilidade com algumas versões do Spring.

Primeiro digite **Java Download** no google e clique nas primeiras pesquisas que diz “Fazer download Java para Windows”, se caso você for usuário do Windows, mas se for de Linux, clique no link “Download Java for Linux”. Neste E-book trabalharemos em cima do ambiente Windows. Também é possível acessar este link diretamente: https://www.java.com/pt-BR/download/ie_manual.jsp?locale=pt_BR.

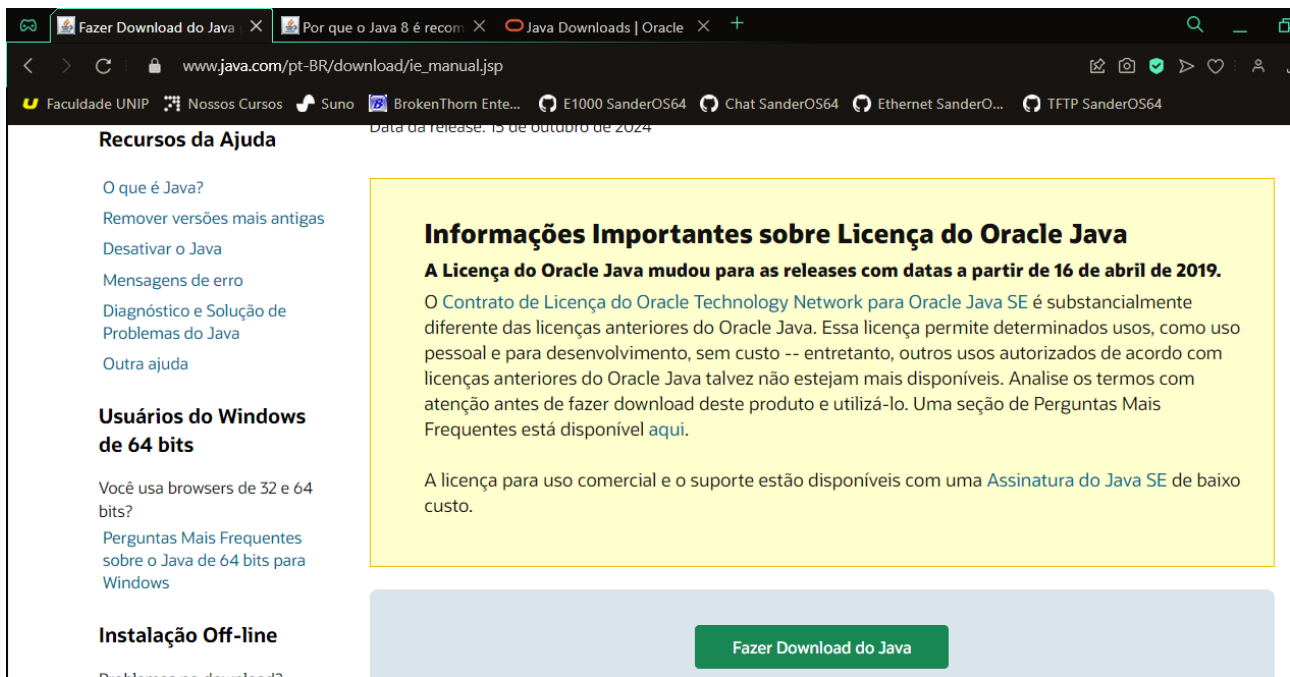


Figura 2 - Tela Inicial do Web Site de Download

Role a página para baixo e visualize o botão azul **Fazer Download do Java**. Clique neste botão e abrirá uma tela para baixar o arquivo de instalação **JavaSetup8u431.exe**.

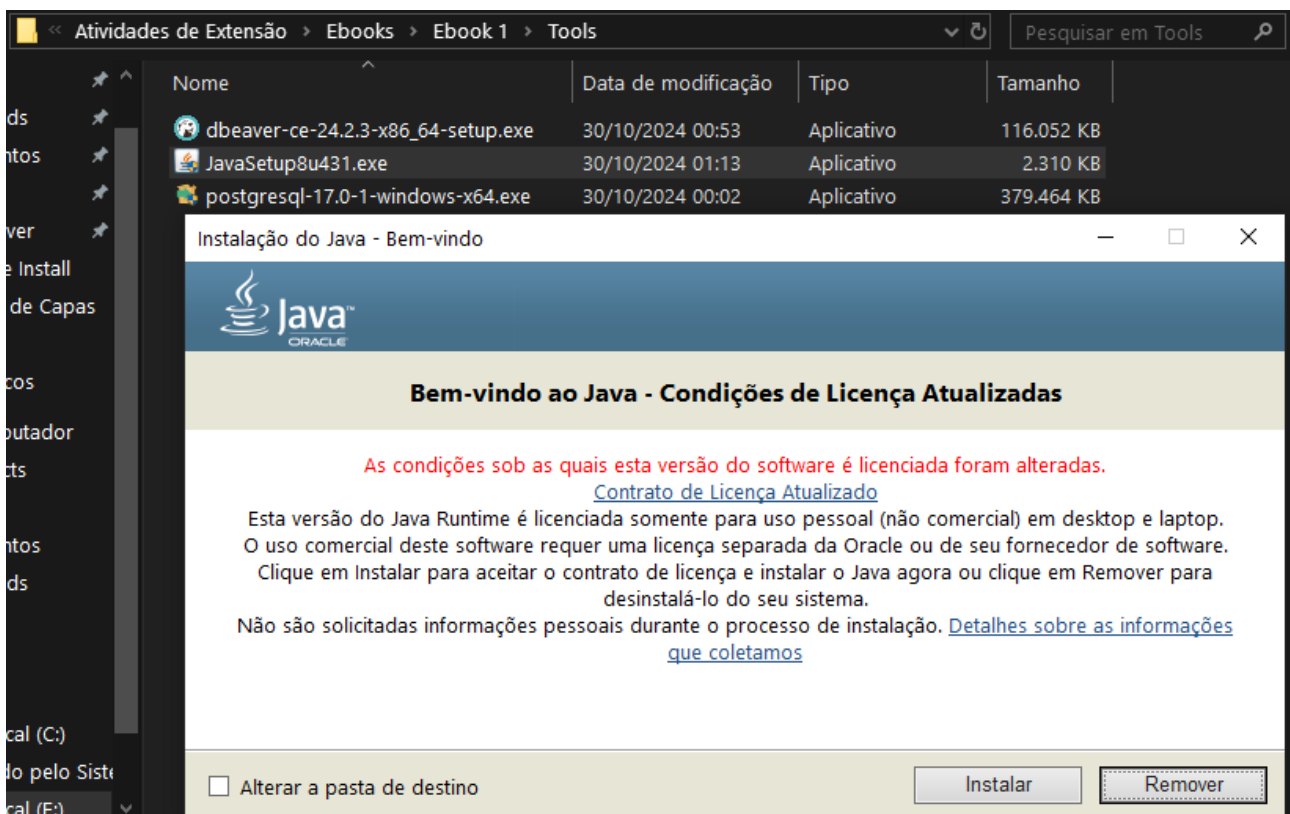


Figura 3 - Tela de instalação do Java

Após baixar, clique no executável e abrirá esta tela. Na tela de instalação, leio os termos e clique em **Instalar**.

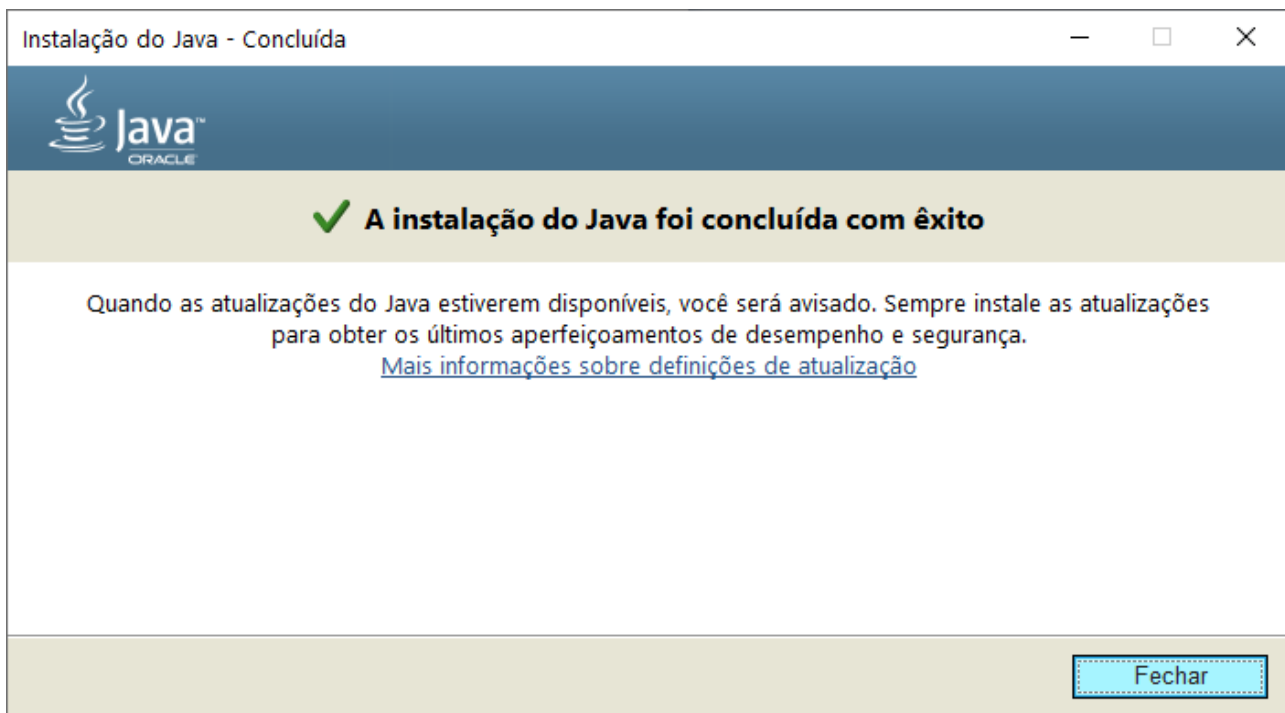


Figura 4 - Instalação do Java Concluída

Após finalizar a instalação, abrirá a tela de instalação concluída e agora basta fechar. Para verificar a versão instalada, abra o **Prompt de Comando** no menu iniciar e digite **java -version**.

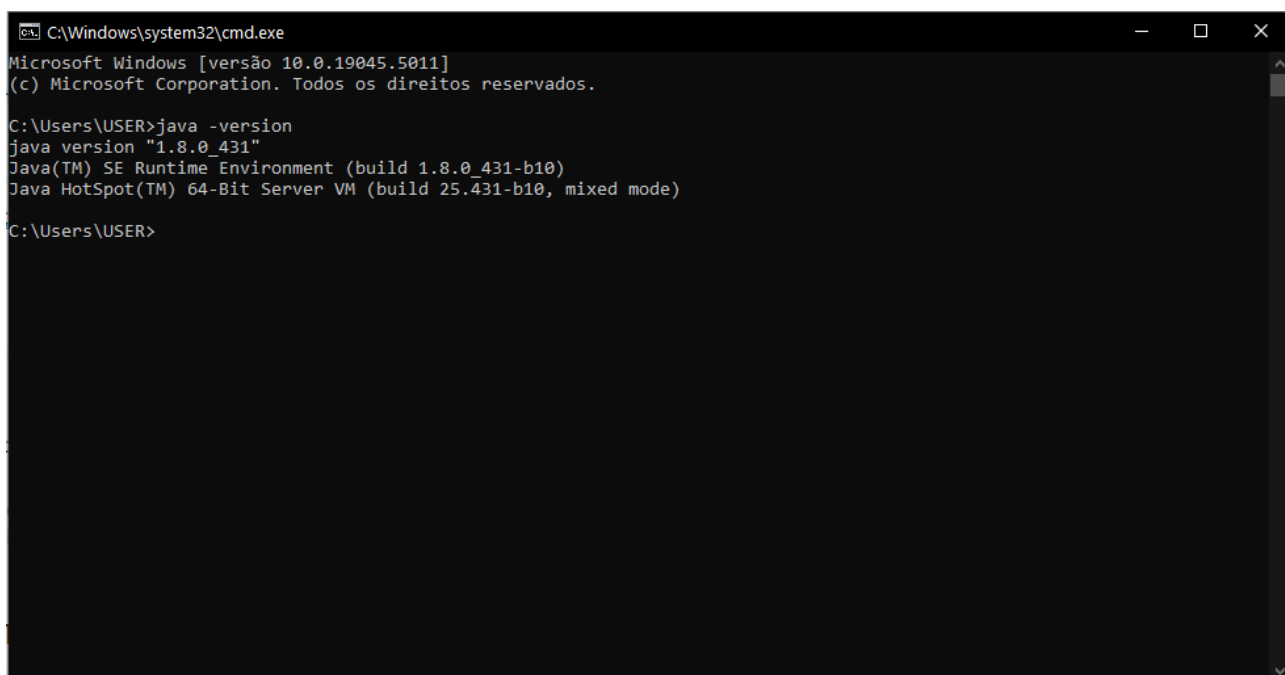


Figura 5 - Verificando a versão da JRE

Perceba que instalamos a versão **1.8.0_431** que é a versão recomendada que mencionamos onde várias aplicações ainda utilizam. No caso do JDK, baixaremos o Kit de desenvolvimento na versão 21. Para isto, poderá digitar no Google **JDK 21** e clicar no link que aparece **Java SE 21 Archive Downloads**.

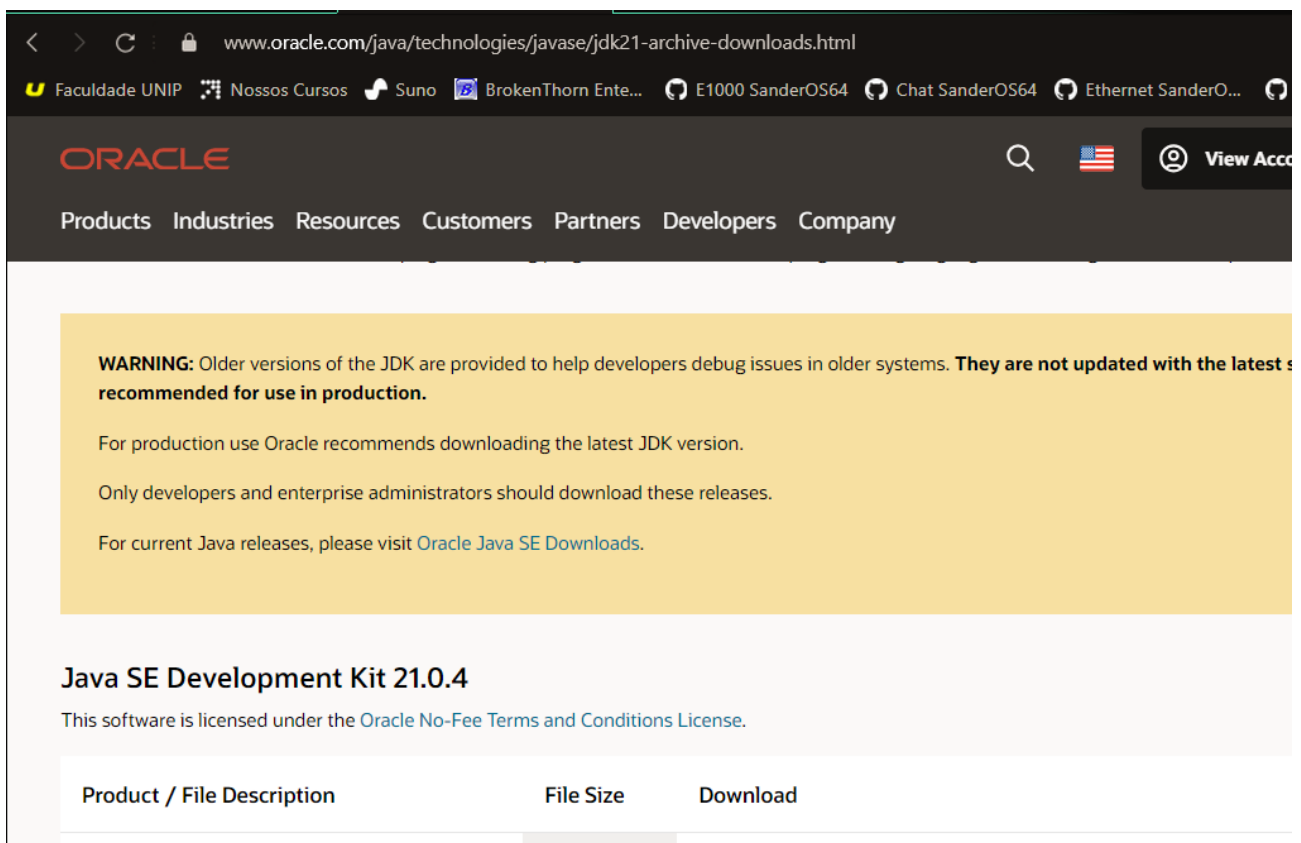


Figura 6 - Página inicial da Oracle

Abrirá esta página onde identificar o texto Java SE Development Kit 21.0.4. Se rolar a tela, encontrará outras versões mais antigas. Role a página neste mesmo título até encontrar o nome *Windows x64 msi Installer* e veja o link a frente dele, clique neste link e ele fará o download do arquivo **jdk-21.0.4_windows-x64_bin.msi**.

Windows x64 Compressed Archive	185.84 MB	https://download.oracle.com/java/21/archive/jdk-21.0.4_windows-x64_bin.zip (sha256)
Windows x64 Installer	164.23 MB	https://download.oracle.com/java/21/archive/jdk-21.0.4_windows-x64_bin.exe (sha256)
Windows x64 msi Installer	162.97 MB	https://download.oracle.com/java/21/archive/jdk-21.0.4_windows-x64_bin.msi (sha256)

Figura 7 - Link que realiza o download do arquivo de instalação

O link direto para este download é https://download.oracle.com/java/21/archive/jdk-21.0.4_windows-x64_bin.msi e poderá realizar o download por outra URL também, como <https://www.oracle.com/java/technologies/downloads/?er=221886#java21>. Nesta última URL inclusive encontra-se o JDK 8, 11, 17, 21 e 23, que são as principais versões.

JDK 23 **JDK 21** GraalVM for JDK 23 GraalVM for JDK 21

JDK Development Kit 21.0.5 downloads

JDK 21 binaries are free to use in production and free to redistribute, at no cost, under the [Oracle No-Fee Terms and Conditions](#) (NFTC).

JDK 21 will receive updates under the NFTC, until September 2026, a year after the release of the next LTS. Subsequent JDK 21 updates will be licensed under [SE OTN License](#) (OTN) and production use beyond the [limited free grants](#) of the OTN license will [require a fee](#).

Linux macOS **Windows**

Product/file description	File size	Download
x64 Compressed Archive	185.91 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.zip (sha256)
x64 Installer	164.28 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)
x64 MSI Installer	163.03 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.msi (sha256)

Figura 8 - Link para download da versão 21.0.5

Java 17 GraalVM for JDK 17 Java 11 **Java 8** Java 8 Enterprise Performance Pack

Java SE Development Kit 8u431

Java SE subscribers will receive JDK 8 updates until at least **December 2030**.

The Oracle JDK 8 license changed in April 2019

The [Oracle Technology Network License Agreement](#) for Oracle Java SE is substantially different from previous versions, such as personal use and development use, at no cost -- but other uses authorized under prior Oracle terms carefully before downloading and using this product. FAQs are available [here](#).

Commercial license and support are available for a low cost with [Java SE Universal Subscription](#).

JDK 8 software is licensed under the [Oracle Technology Network License Agreement](#) for Oracle Java SE.

Java SE 8u431 [checksums](#) and [OL 8 GPG Keys](#) for RPMs

Linux macOS Solaris **Windows**

Figura 9 - Download das versões 8, 11 e 17

Pode-se perceber que na Figura 8, é mostrado a versão 21.0.5, que é a versão instalada na próxima imagem. Na Figura 9, temos as 3 versões do Java, incluindo a informação de que a versão 8 terá suporte até o final de 2030.

O próximo de instalação do JDK é quase a mesma da JRE, com a diferença que é só preciso clicar em **Next**, **Next...** e **instalar**. Não é preciso configurar nenhuma opção especial, desta forma, a instalação vai criar uma pasta **jdk-21** em C:\Program Files\Java.

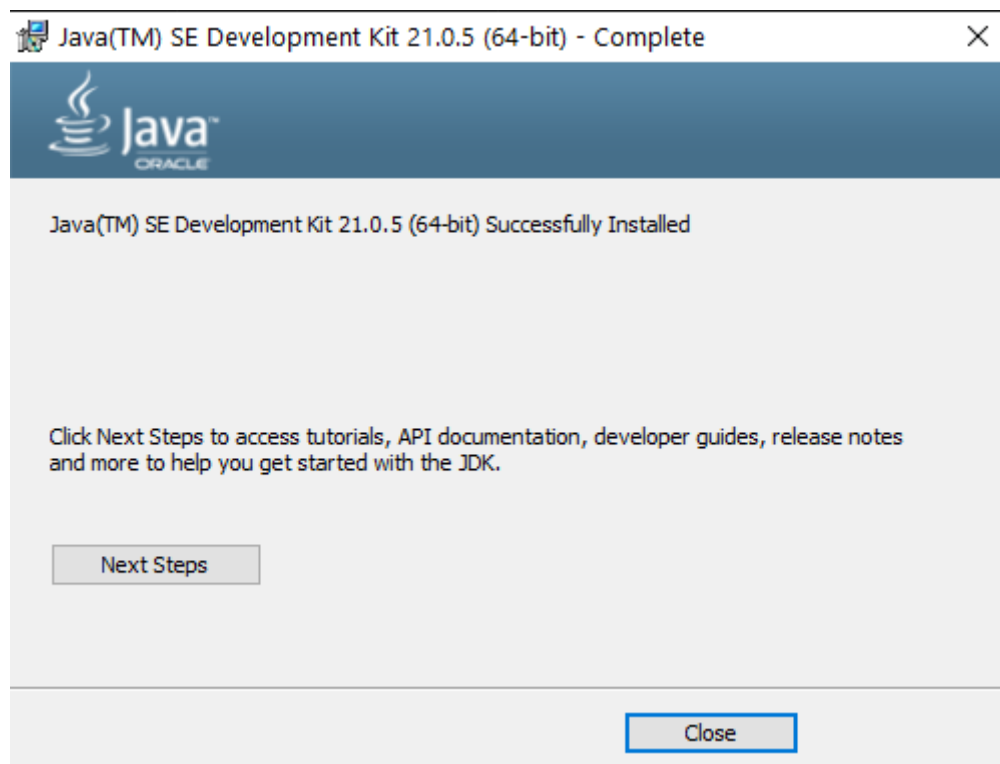


Figura 10 – Tela do JDK 21.0.5 concluído

Agora você já pode trabalhar com aplicações em Java! Sinta-se à vontade para explorar outras versões e instalá-las, o instalador vai garantir de criar uma nova pasta para você de cada JRE e JDK. Desta forma, é possível configurar seus projetos para utilizar versões específicas previamente instaladas e testar alguns limites.

2.2. Instalando o Editor VSCode

O VSCode ou Visual Studio Code é uma alternativa do Visual Studio Community da Microsoft, sendo um editor mais leve e flexível para desenvolvimento de aplicações em inúmeras linguagens de programação. Os desenvolvedores Web normalmente preferem este Editor devido a sua compatibilidade com ambientes Linux e integração de inteligência artificial como o Github Copilot para aumentar a produtividade.

Para instalar o VSCode, basta entrar no link <https://code.visualstudio.com> ou simplesmente digitar **VSCode** no Google e entrar no primeiro link de pesquisa. O site que se abrirá terá o botão **Download for Windows** para realizar o download do instalador.

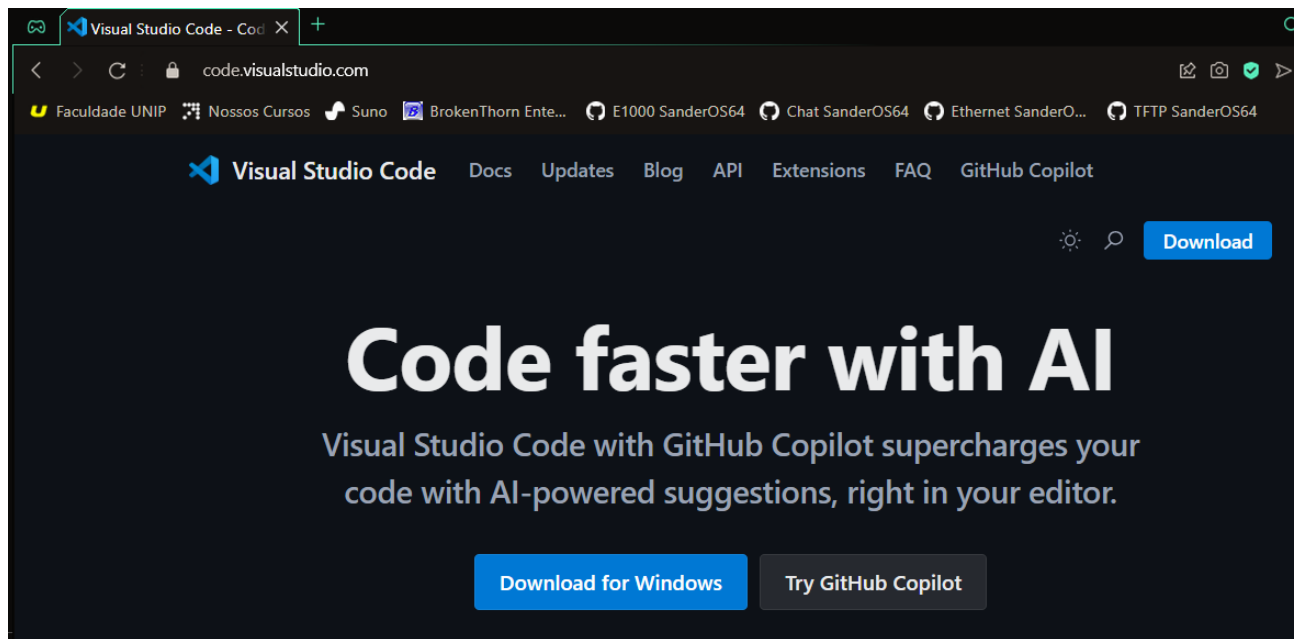


Figura 11 - Página inicial pra Download do VSCode

Após clicar no botão, será redirecionado para uma página que vai baixar o arquivo VSCodeUserSetup-x64-1.95.1.exe. A janela que se abre pergunta em qual local quer salvar o instalador, durante estas aulas estou escolhendo a pasta Tools no desenvolvimento do E-book para facilitar, recomendo organizar uma pasta para administrar seus instaladores caso precise reinstalar em outros computadores.

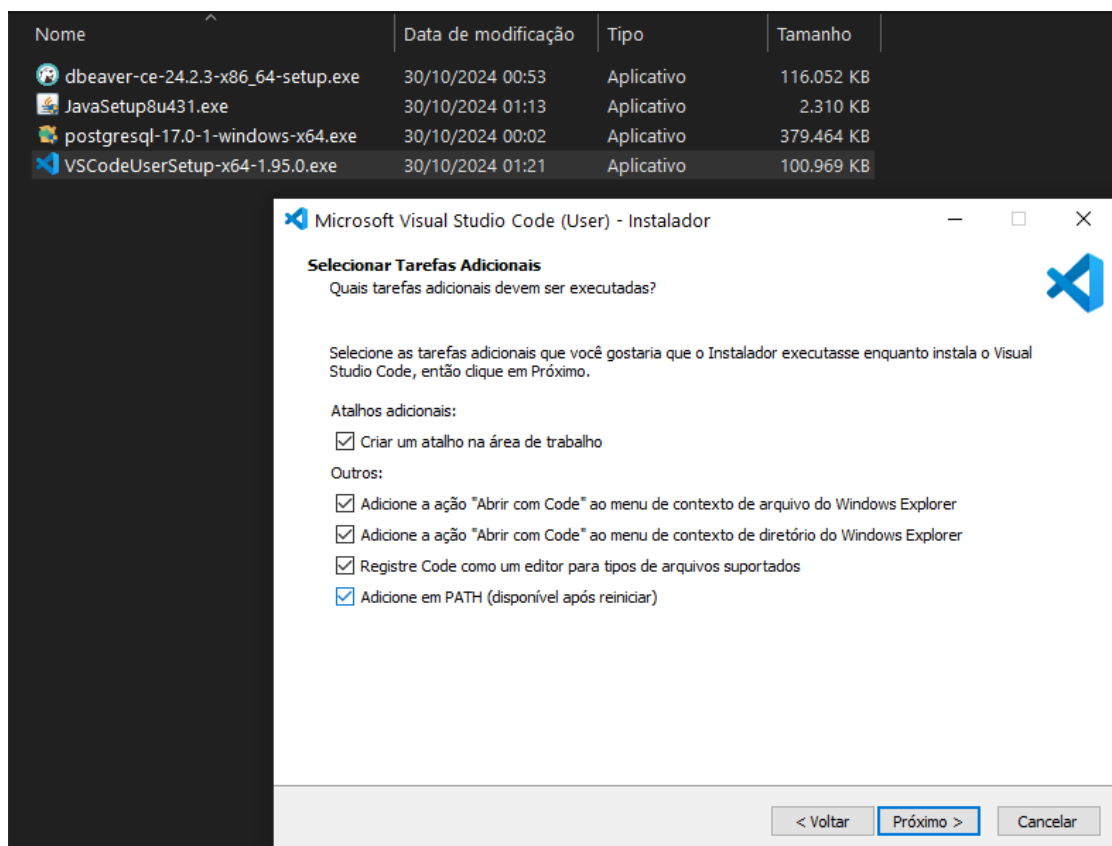


Figura 12 - Instalando o VSCode - Opções de Marcação

O processo de instalação é bem simples, apenas clique em próximo até chegar nesta tela da Figura 12. Nesta tela, as 4 últimas opções em “Outros” já vão estar marcadas, deixe elas como estão e marque a 1ª caixa de **Criar um atalho na área de trabalho**, isto facilita na identificação do ícone para execução do VSCode. Clique em próximo e instale, após isto, é só inicializar o editor.

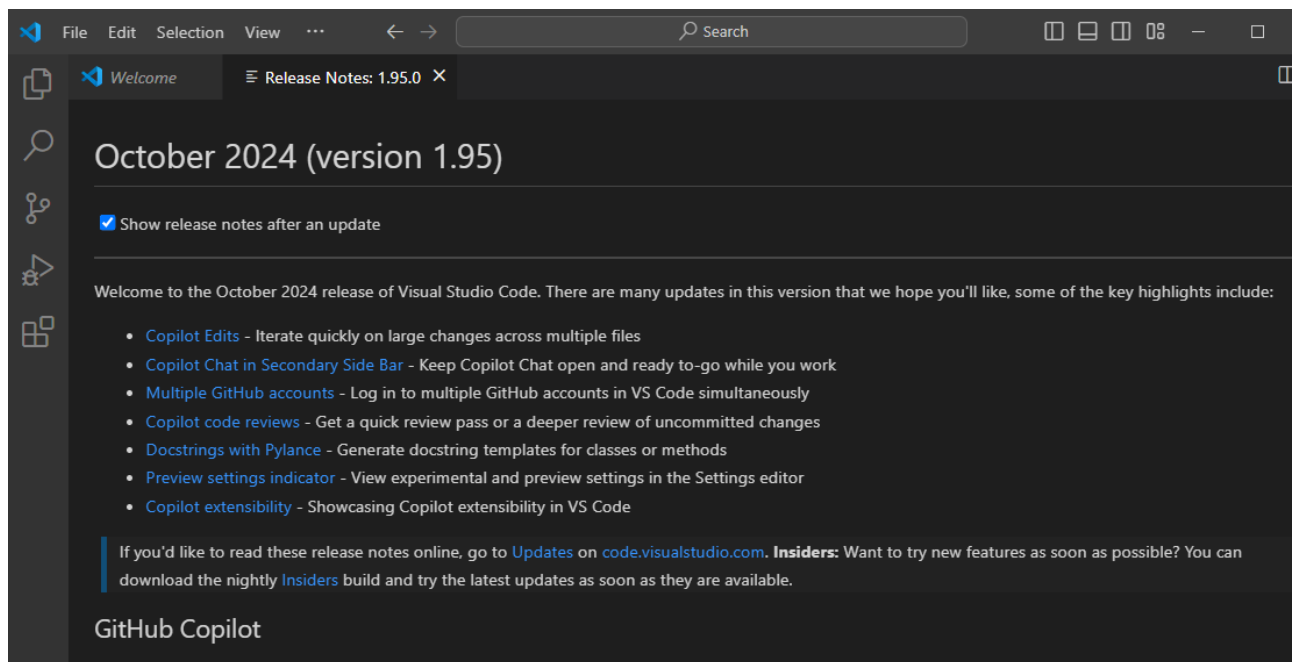


Figura 13 - Tela inicial do VSCode

No momento que foi feita esta instalação, o VSCode está na versão 1.95, que foi atualizada em outubro de 2024. Você poderá ler todos os Updates geradas desta versão nesta tela e explorar mais o que foi adicionado clicando sobre os links. Por exemplo, edições com a I.A Copilot, Chat com o Copilot, Revisões de código com o Copilot, além das múltiplas contas do Github e diversas outras atualizações.

Se você prefere utilizar uma IDE como Eclipse ou IntelliJ, fica de sua escolha, use aquilo que mais te satisfazer, mas o intuito de utilizar o VSCode é demonstrar o quão simples é programar nesta ferramenta, incluindo as vantagens de importar pacotes de forma automática ao utilizar uma anotação do Spring. Certifique-se de também instalar o pacote de extensão Java.

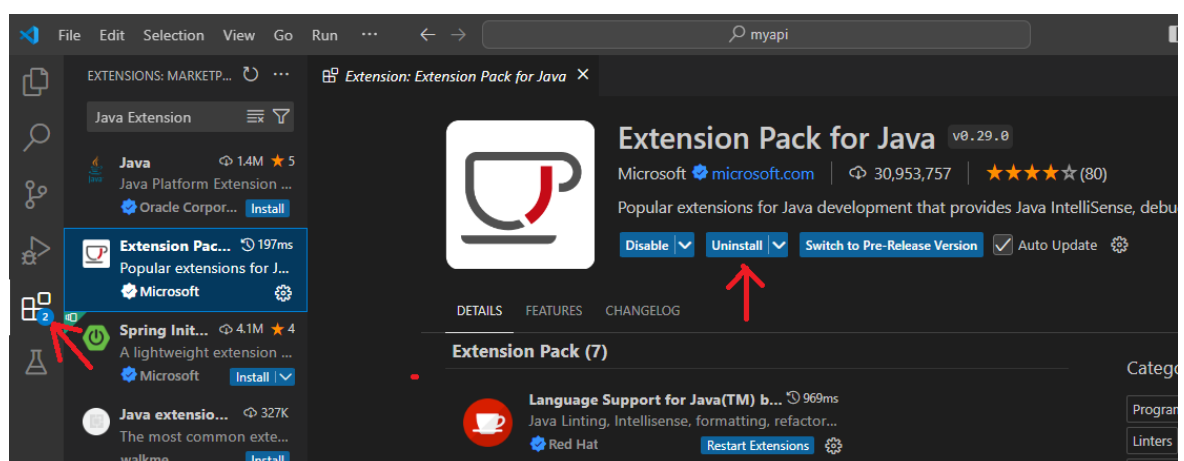


Figura 14 - Extensão VSCode para Java

Na figura 14 é possível instalar diversas extensões para várias linguagens de programação e recursos de frameworks. Isto é feito clicando no 5ª ícone da barra lateral e pesquisando com alguma palavra-chave. O VSCode já sugere as melhores extensões para ser utilizadas. Para instalar, basta clicar em **Install**, no entanto, na imagem mostra-se **Uninstall** pelo fato de já ter sido instalado. Ao iniciar um projeto em Java, durante o gerenciamento de dependências Maven, o VSCode já sugere automaticamente para instalar esta extensão. Explore outras extensões possíveis e veja o que elas podem fazer!

2.3. Instalando o Banco de Dados PostgreSQL

Primeiramente, precisamos instalar um banco de dados, com o intuito da nossa aplicação cadastrar e buscar dados pela API usando o Spring JPA. Instalaremos o PostgreSQL que já vem integrado algumas ferramentas úteis para o controle das tabelas, como o PgAdmin, uma interface intuitiva com dashboard e estatísticas.

Digite PostgreSQL no Google ou click no link <https://www.postgresql.org> e será redirecionado para esta página da imagem abaixo. Também é possível acessar pelo link direto <https://www.postgresql.org/download/>.



Figura 15 - Site inicial do PostgreSQL

Após clicar no botão **Download ->**, será redirecionado para uma nova página que especifica os sistemas operacionais no qual vai realizar a instalação. O ambiente que estamos trabalhando é o Windows, mas se você estiver utilizando outro ambiente, apenas clique no ícone do seu sistema operacional para instalar.

No caso da imagem abaixo, clicaremos no ícone **Windows** e baixaremos o PostgreSQL para esta plataforma:

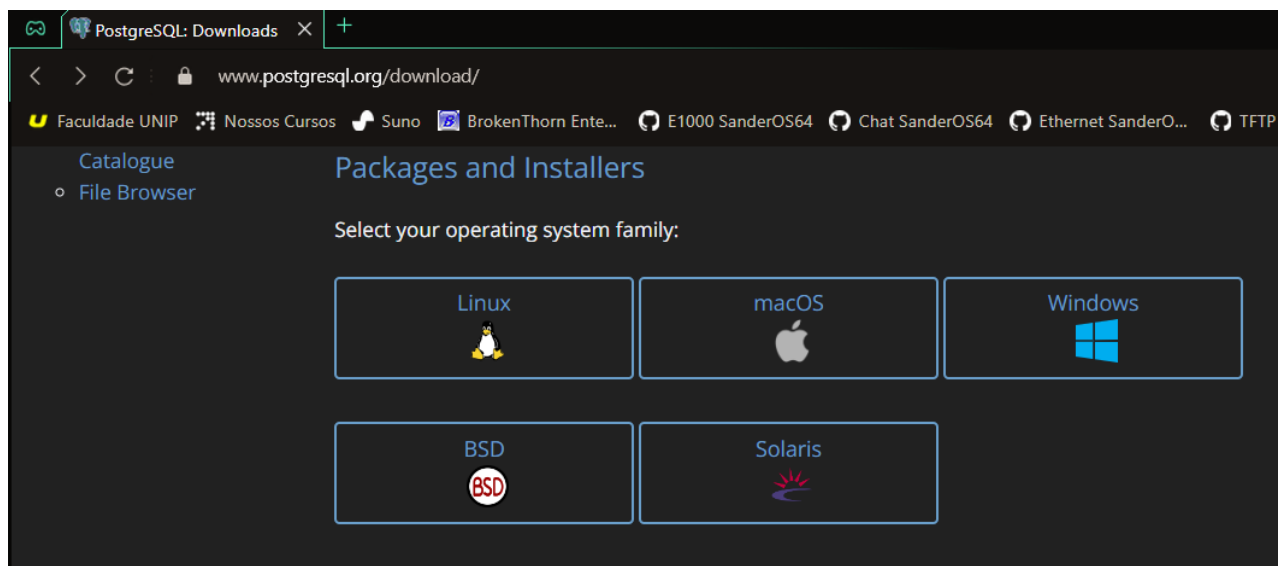


Figura 16 - Página de Download do PostgreSQL para várias plataformas

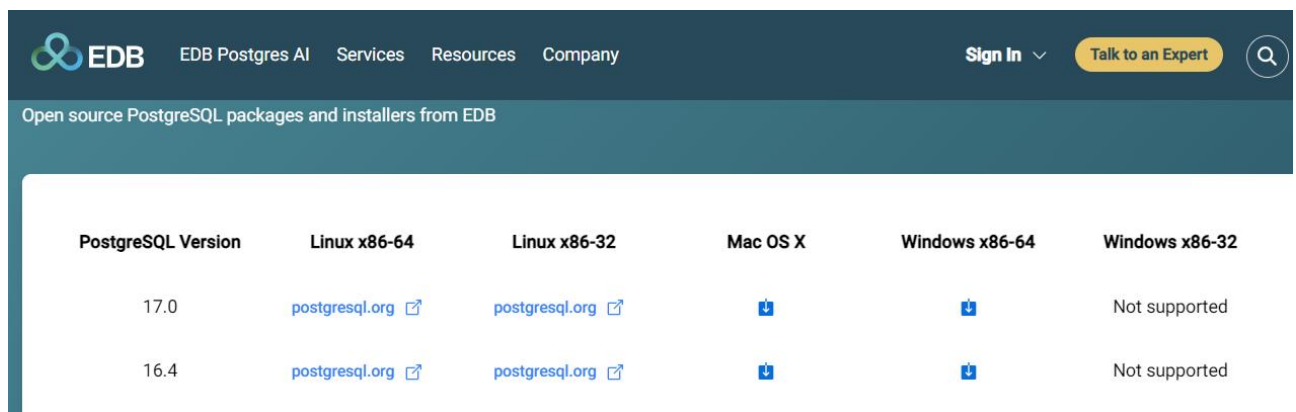
Clicando sobre o ícone do sistema, o site abre esta próxima página, apresentando os instaladores possíveis, apenas clique em **Download the installer**.



Figura 17 - Página para baixar o instalador do Windows

Os links de downloads estão hospedados pelo EDB, portanto, serão redirecionados para a página do EDB que contém as versões e arquiteturas do PostgreSQL. Nesta próxima tela, encontra-se a versão 17.0 e 16.4, além das arquiteturas de 32 e 64 bits.

No entanto, a arquitetura 32-bit só contém para sistemas Linux, se for escolher baixar para Mac OS X ou Windows, 32-bit já não é suportado. Clique sobre o botão de seta verde abaixo de **Windows x86-64**, na mesma linha da versão 17.0.



PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
17.0	postgresql.org	postgresql.org			Not supported
16.4	postgresql.org	postgresql.org			Not supported

Figura 18 - Página do EDB para instaladores de versões

No momento que é clicado, uma janela é aberta para escolher o local de download do instalador. Foi escolhido na pasta Tools onde contém o arquivo **postgresql-17.0-1-windows-x64.exe**. Após executar este programa, ele irá abrir a tela de instalação abaixo.

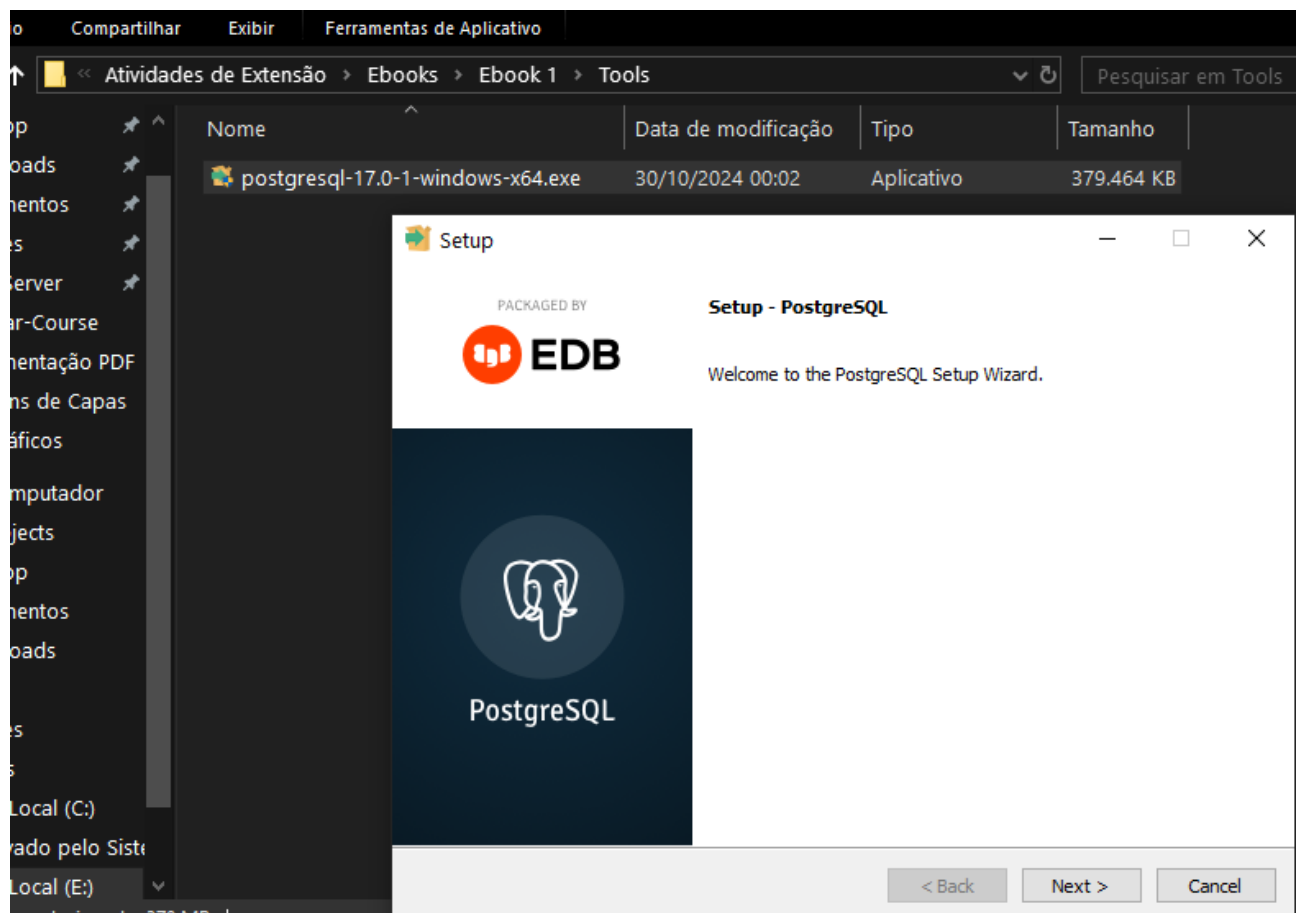


Figura 19 - Tela de instalação inicial do PostgreSQL

Clicando em **Next** (próximo), é mostrado uma nova tela para informar o local de instalação no computador. Vamos deixar exatamente o diretório que já está: C:\Program Files\PostgreSQL\17. Após clicar em Next, uma outra tela vai apresentar as caixas de marcação, como o **PgAdmin** que é a interface gráfica, o servidor PostgreSQL, as ferramentas de linha de comando e o StackBuilder para instalar outras ferramentas.

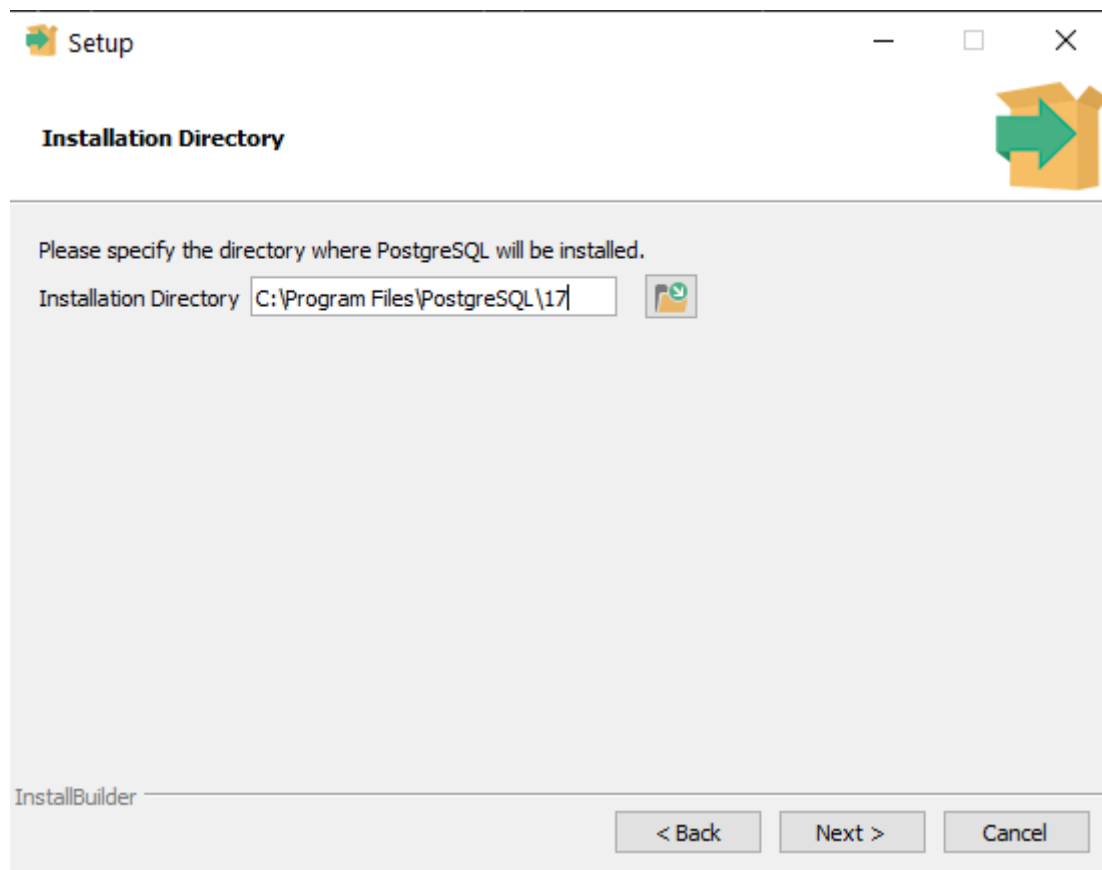


Figura 20 - Configura Diretório de instalação do PostgreSQL

Após clicar em Next, todas as caixinhas já são selecionadas, deixe como está e Next.

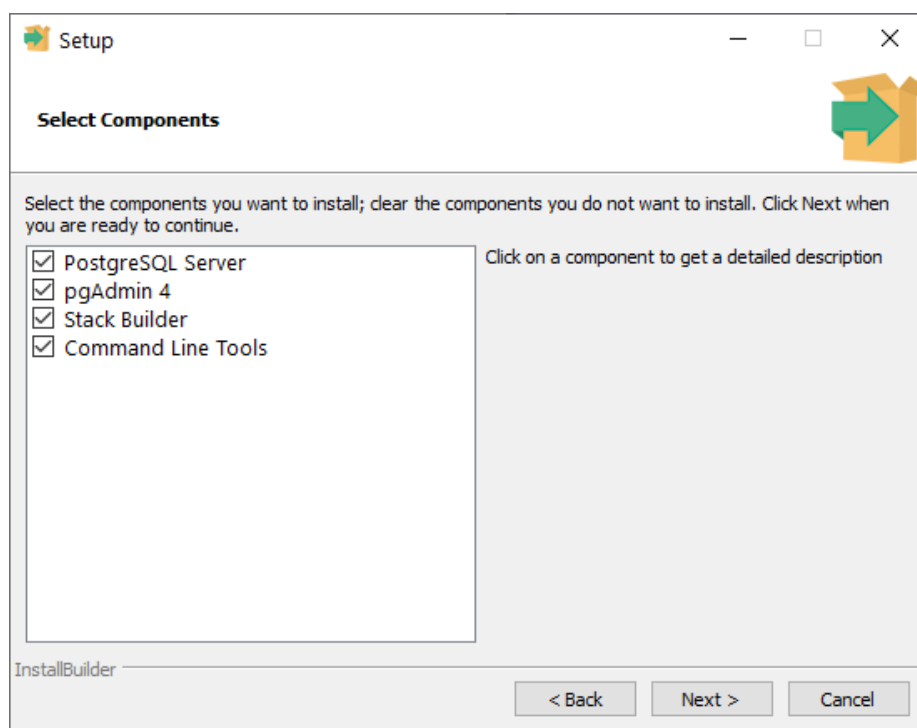


Figura 21 - Ferramentas integradas no PostgreSQL: pgAdmin, Command Tools e Stack Builder

Coloque um diretório para dados do banco, no entanto, ele já cria por padrão. Então apenas clique em Next.

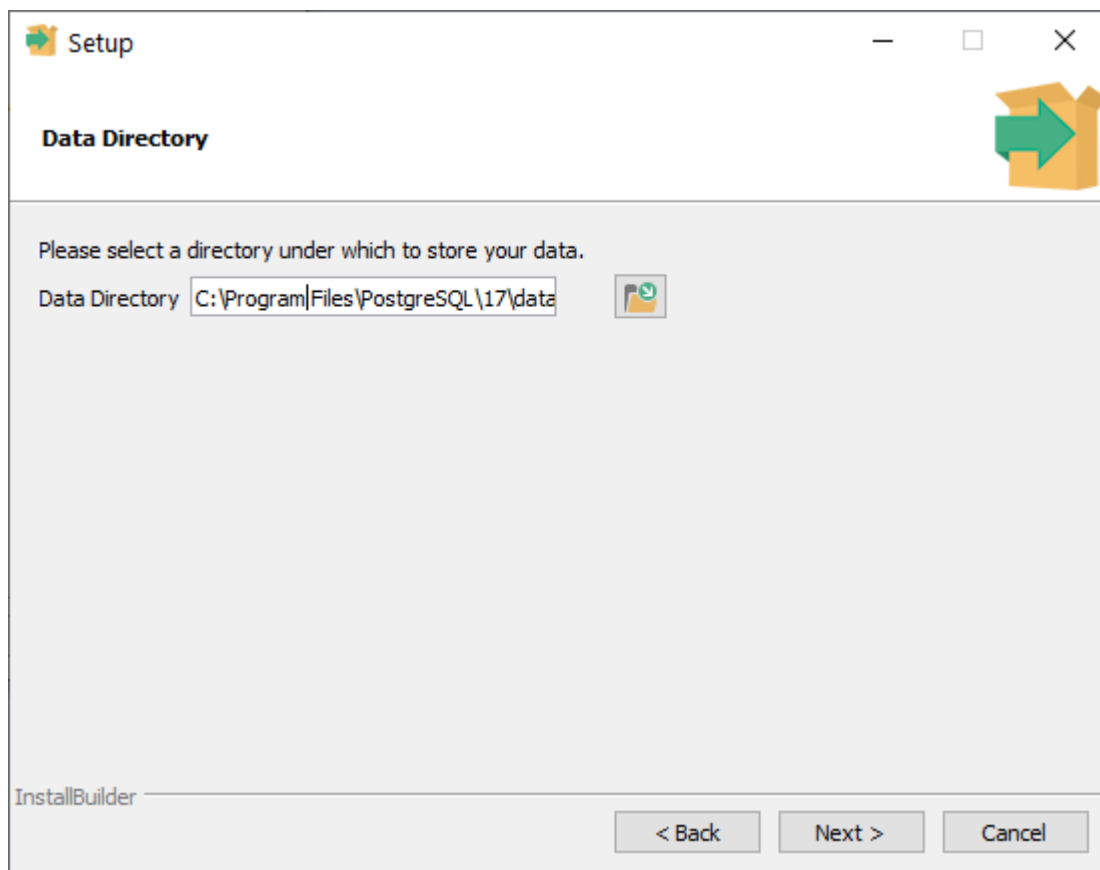


Figura 22 - Diretório padrão de dados do banco PostgreSQL

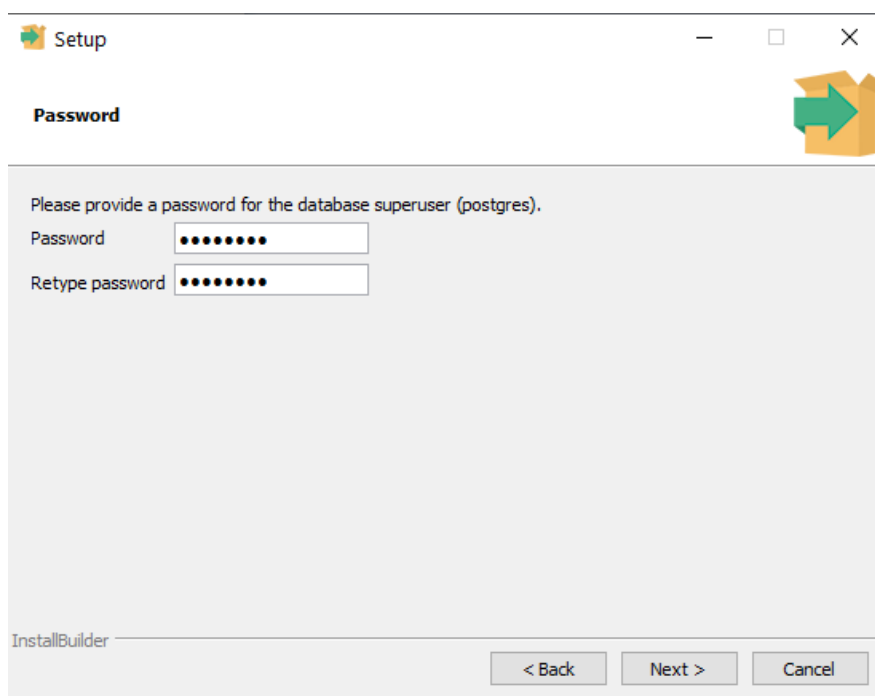


Figura 23 - Inserir a senha do banco de dados para usuário "postgres"

Na imagem acima é solicitado para criar uma senha de acesso ao PostgreSQL. Por padrão, ele criará um banco de dados com o mesmo nome, e você só terá acesso a este banco se fornecer a senha. Porém, a cada novo banco criado, para ter acesso, é preciso fornecer esta mesma senha de acesso.

O usuário padrão é “postgres”. Normalmente, escolhemos a senha como “postgres” também para facilitar nos testes de conexão. Logo abaixo é apresentado o número da porta de conexão onde as aplicações irão se comunicar com o PostgreSQL. Por padrão é a porta 5432.

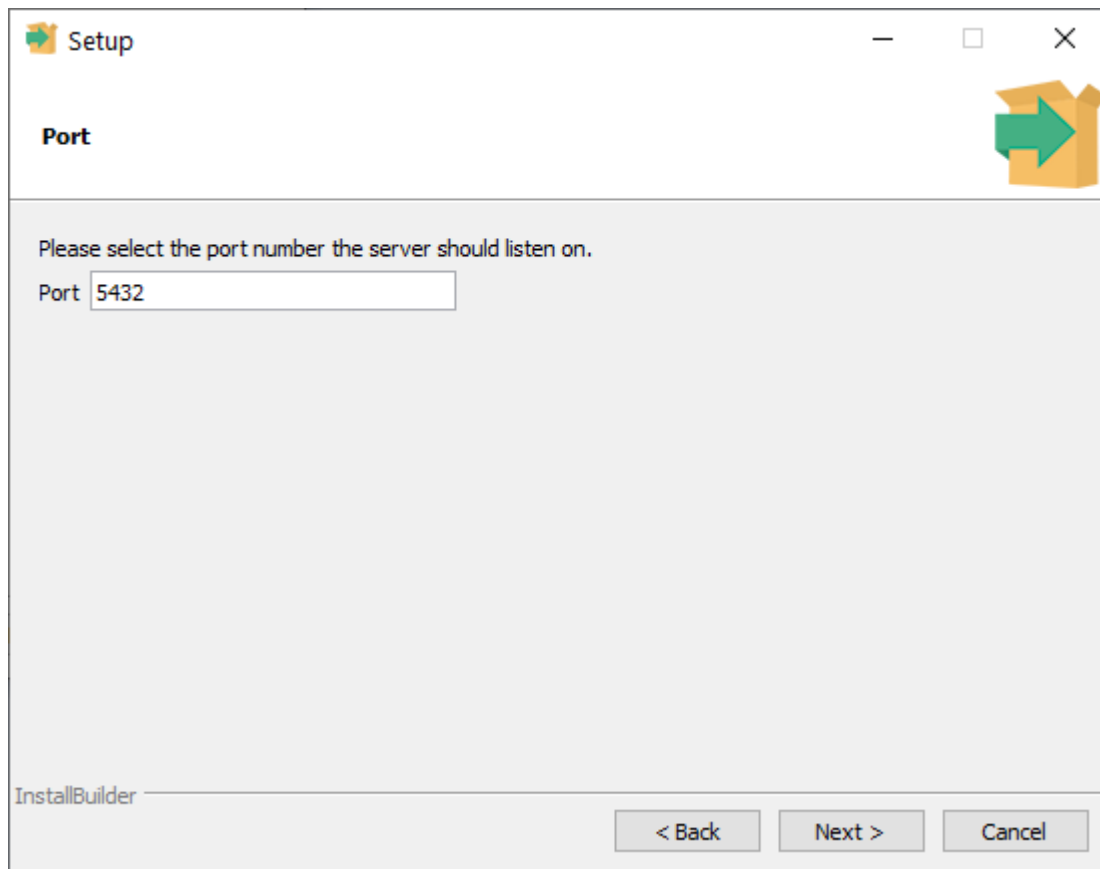


Figura 24 - A porta padrão 5432 para acesso do banco

Cada servidor contém sua própria porta de comunicação na rede, o banco de dados também é um servidor, que através de sua porta é possível receber solicitações de queries (consultas) e atualizações, seja de forma interna ou externa.

No momento da conexão ao banco pela aplicação, o desenvolvedor precisa fornecer uma simples URL especificando o Driver de banco de dados Java, que é o *JDBC*, o tipo de banco de dados, que em nosso caso é o *postgresql*, o número IP do servidor ou se for servidor local, ao invés de colocar o ip, basta inserir *localhost*, e por fim, a porta de acesso a este servidor, além do nome do banco de dados.

Esta porta de acesso mencionada é a **5432**. Podemos ter várias aplicações em portas diferentes rodando em um mesmo tipo de servidor – O localhost. Por exemplo: A API e o PostgreSQL.

As duas imagens abaixo finalizam a nossa instalação, escolhendo o local do idioma que é **Portuguese, Brasil** e por fim, a instalação é feita.

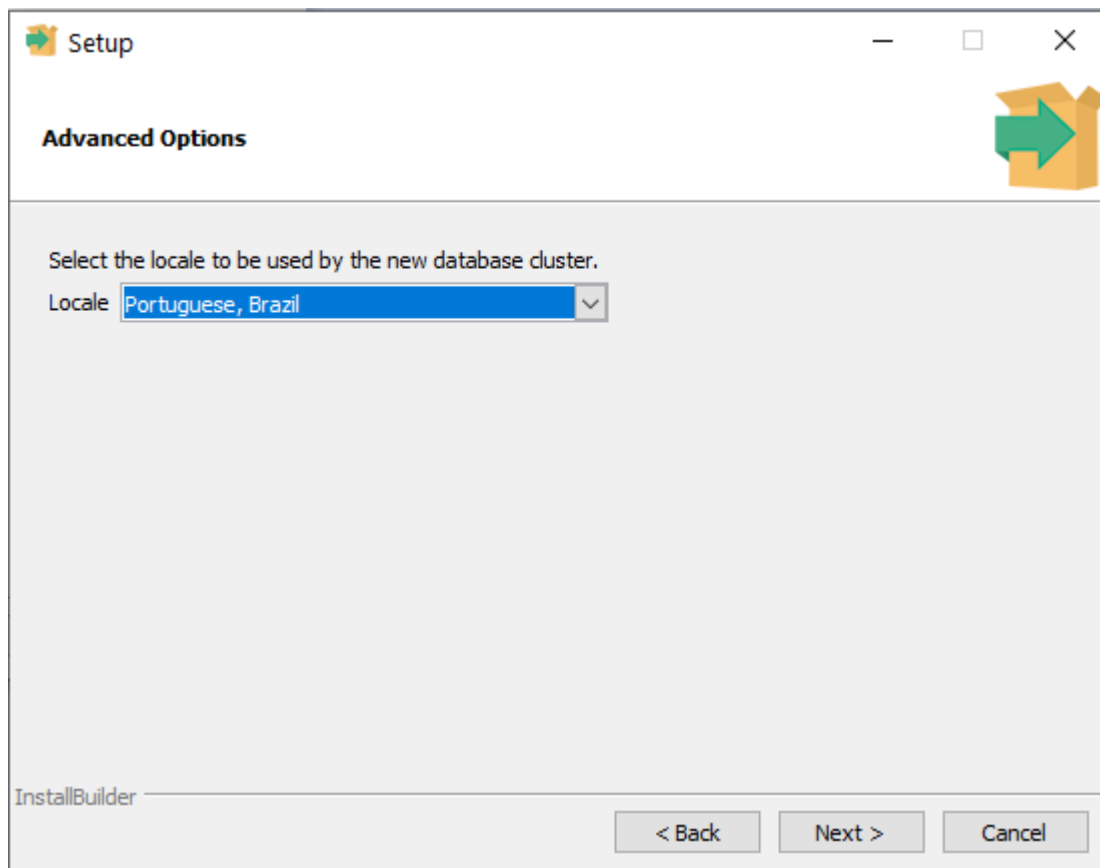


Figura 25 - Escolhendo o local de idioma

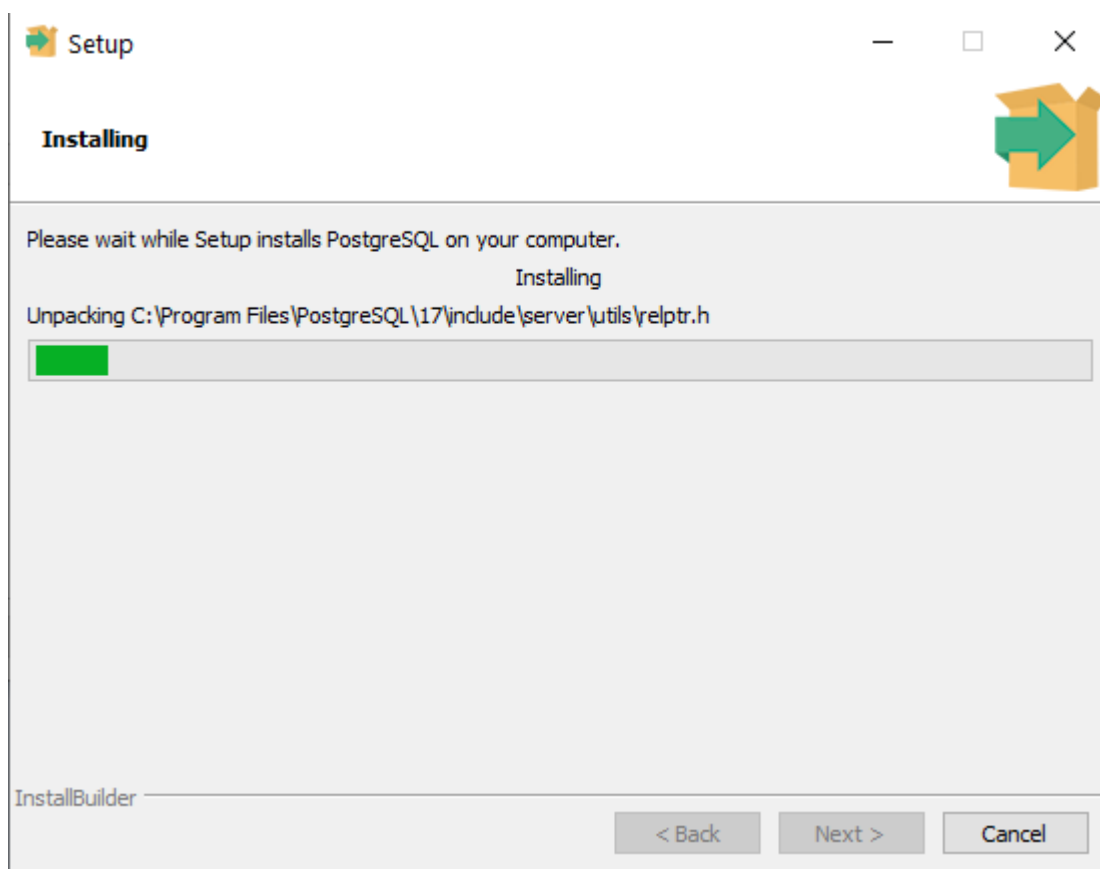


Figura 26 - Processo de instalação dos arquivos

Após a instalação do Postgres ser concluída, é perguntado se o usuário quer instalar ferramentas adicionais, como drivers e complementos do PostgreSQL.

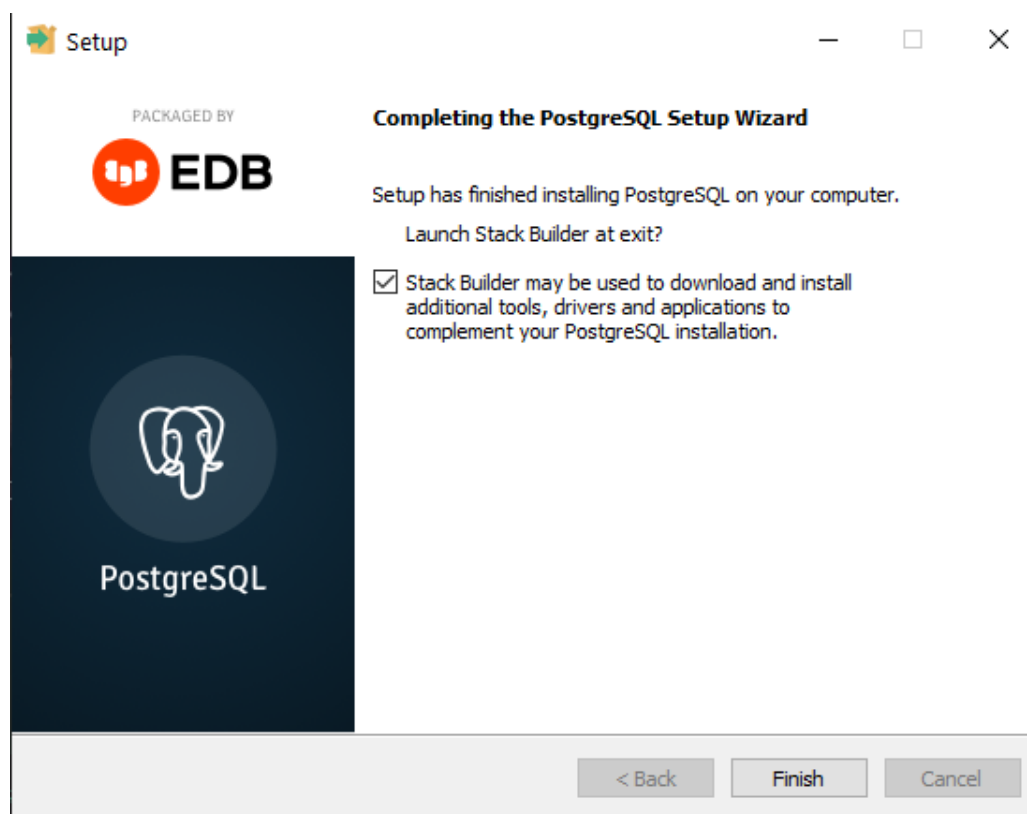


Figura 27 - Marcação do Stack Builder para instalações adicionais

Foi marcado o Stack builder para instalarmos uma ferramenta que pode ser útil no futuro, que é o Toolkit de Migrations. **Nota:** Esta etapa é opcional.

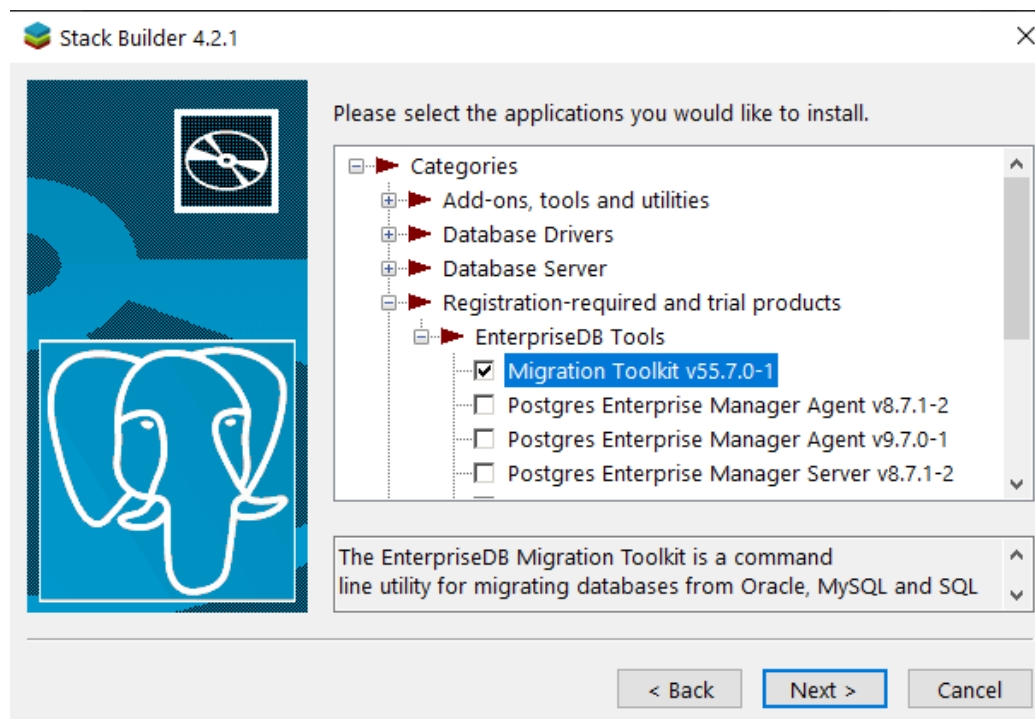


Figura 28 - Marcando a instalação do Toolkit Migration

Na etapa anterior, será instalado o Migration, pois ele pode desempenhar um papel fundamental na migração de banco de dados e gerenciamento de Scripts DDL. No entanto, fica a seu critério instalar, pois não vai influenciar na aplicação desse E-book. No menu iniciar, digite **pgAdmin** ou apenas as iniciais, será sugerido o aplicativo a ser aberto.

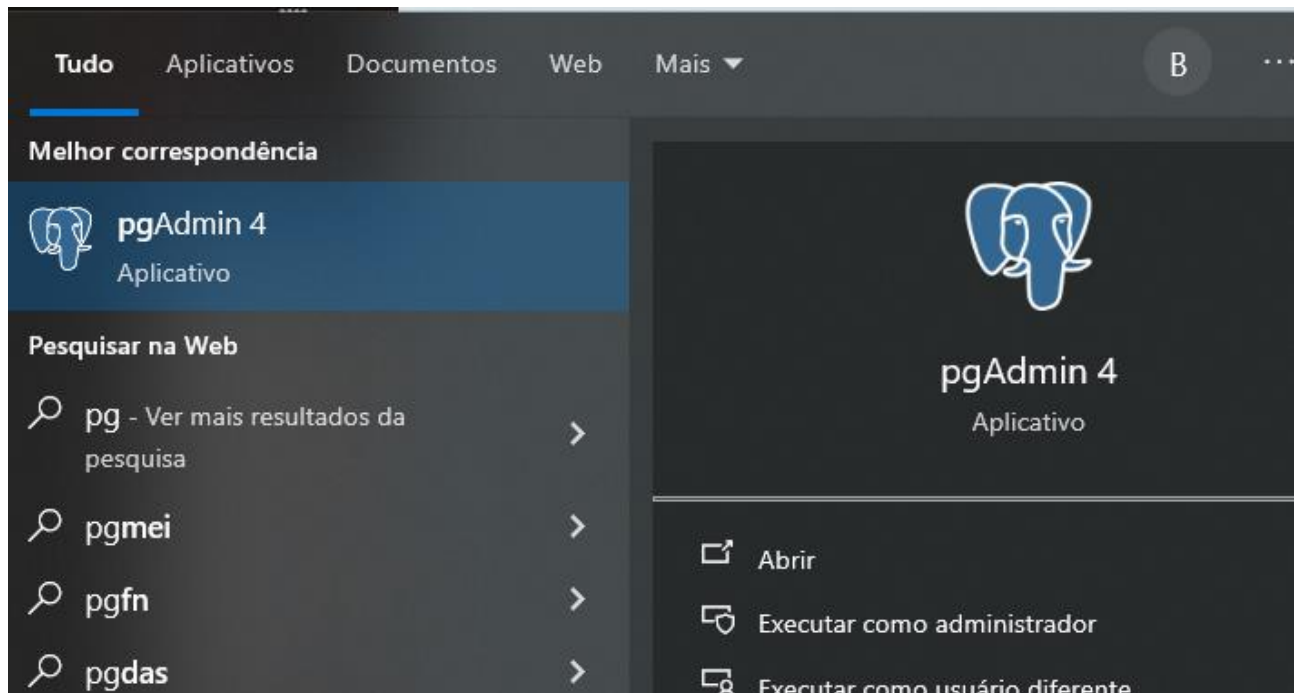


Figura 29 - Pesquisa do pgAdmin no menu iniciar

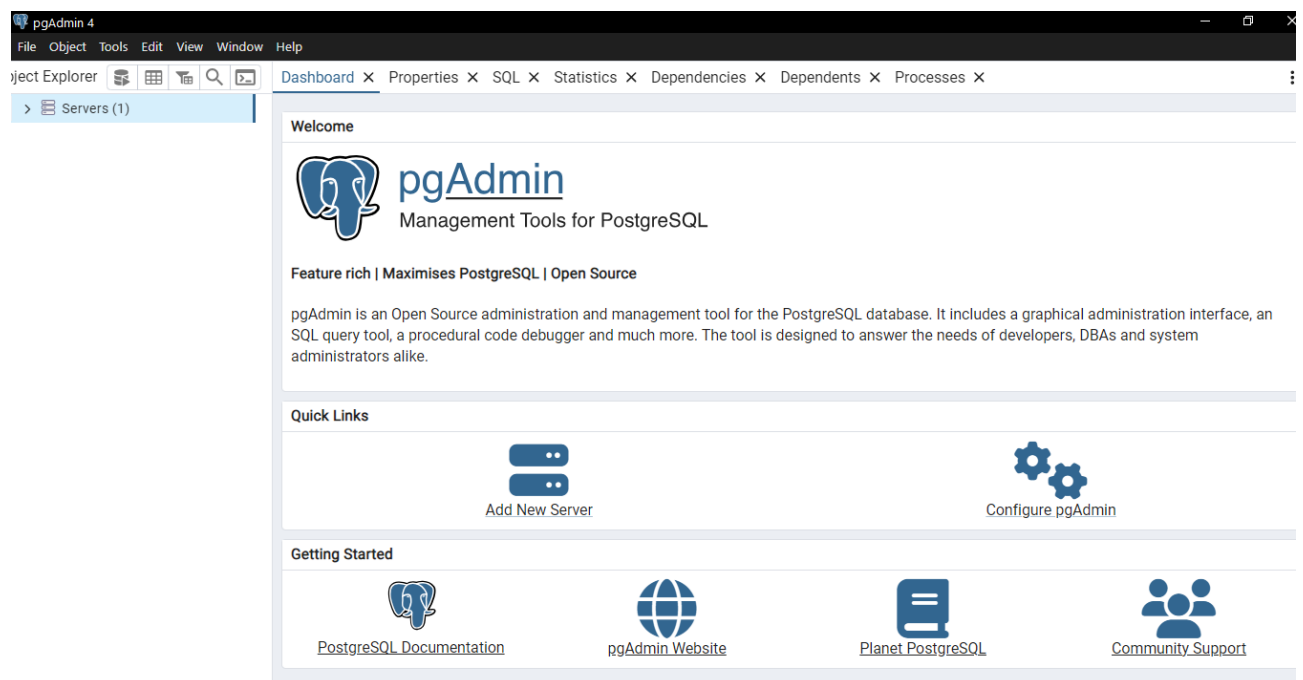


Figura 30 - Tela inicial do pgAdmin 4

Esta é a tela inicial do pgAdmin, onde poderá ser feito a administração dos dados. Ele conta com algumas abas na parte de cima e na lateral que se encontra os servidores.

Clicando em **Servers (1)** onde diz que tem 1 servidor, é aberto uma tela para inserir a senha de conexão. Coloque a mesma senha que foi criado na instalação: *postgres*.

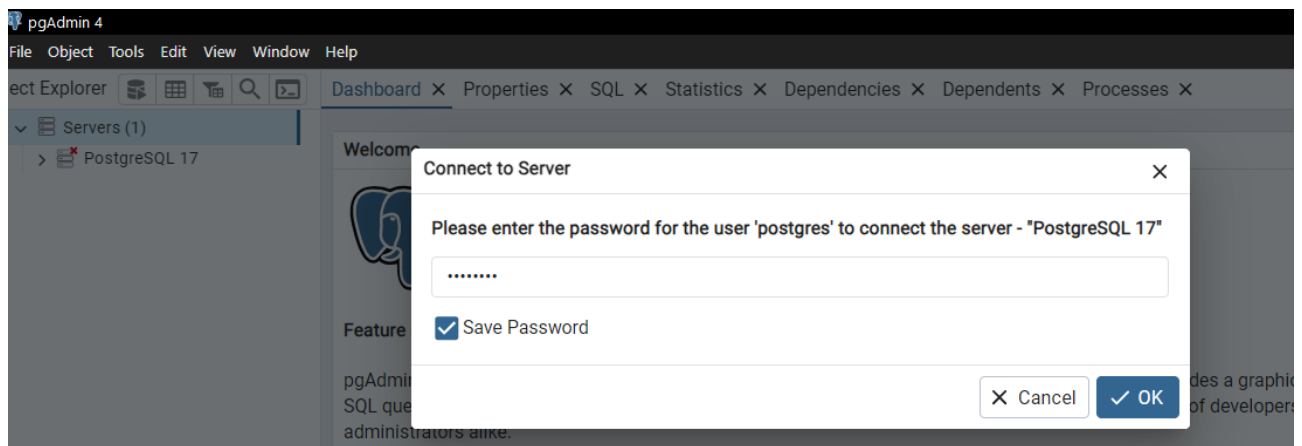


Figura 31 - Inserindo a senha para conexão com o servidor

Abaixo é apresentado todos os elementos fundamentais para trabalharmos com o banco de dados. Após se conectar ao servidor, será aberto a lista de banco de dados em **Databases (1)**, dentro dele contém o banco **postgres** que é criado por padrão na instalação. É possível perceber uma série de elementos que compõem um banco de dados, no entanto, a parte principal fica em **Schemas (1)** e **public** onde ficará nossas tabelas.

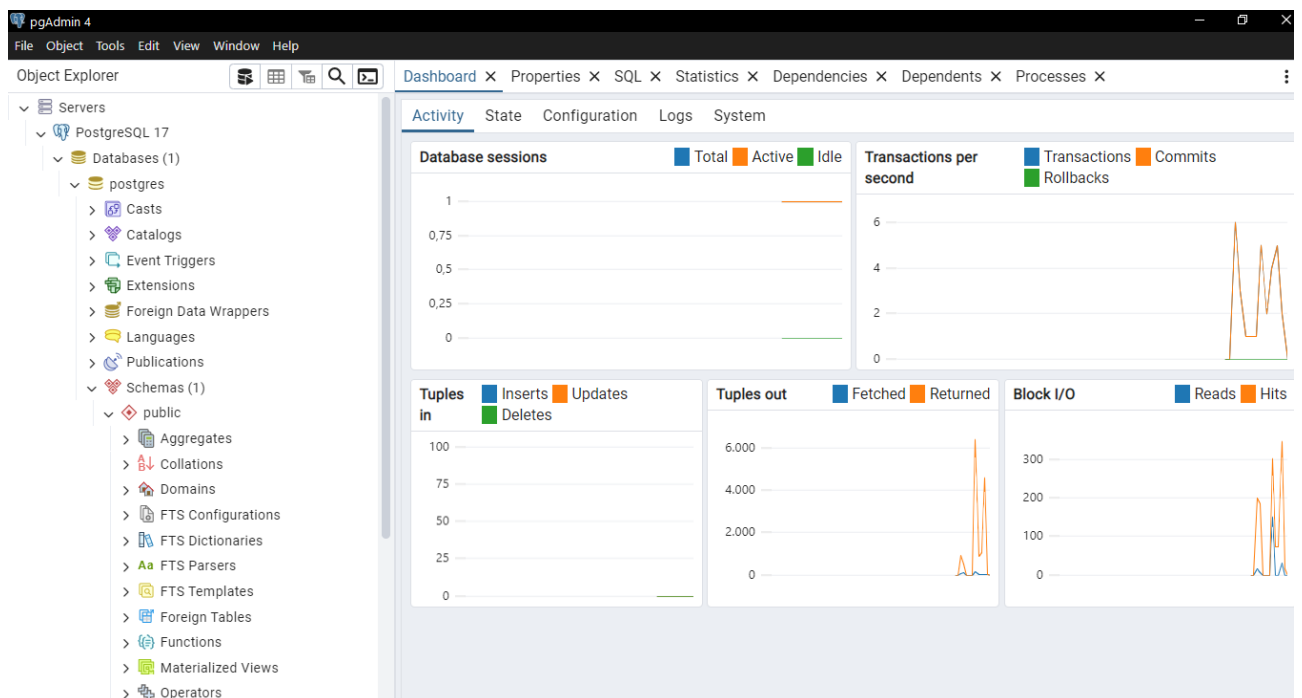


Figura 32 - Servidor conectado com banco de dados e dashboard

Por enquanto, ainda não temos nenhuma tabela criada, apenas alguns atributos essenciais no gerenciamento. Veja na parte central que a aba **Dashboard** está aberta, mostrando dados estatísticos em tempo real do gerenciamento do banco, como: Transações, commits, entradas de tuplas (Inserts, Updates), entre outras coisas. Na aba acima, também podemos clicar em **SQL** para realizar nossas consultas e criações de tabelas usando a linguagem SQL.

Como uma alternativa amigável, também baixaremos o DBeaver Community no site <https://dbeaver.io>. Apenas role a tela e clique em **Download** na parte do DBeaver Community que é uma versão gratuita. Ou vá direto para o link <https://dbeaver.io/download/> que será redirecionado para a Figura 35.

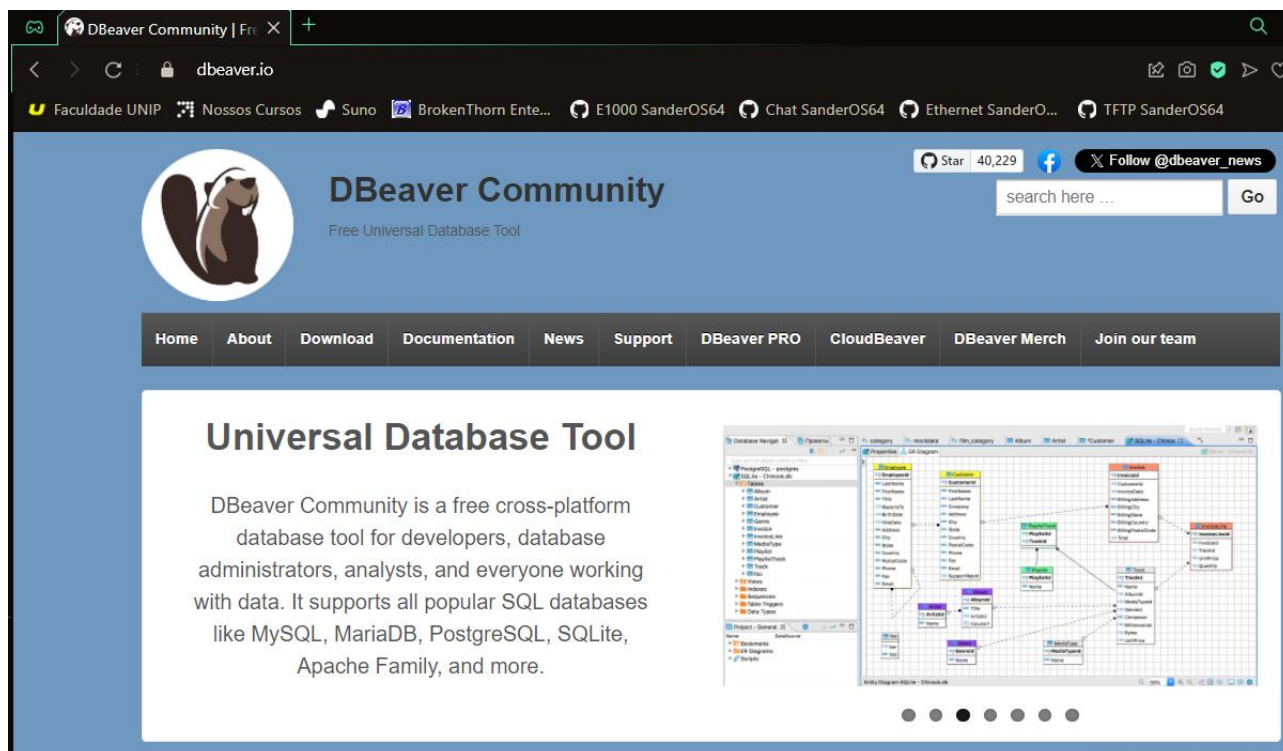


Figura 33 - Página inicial do software DBeaver

DBeaver Community	DBeaver PRO
Open-source version	Commercial versions
<ul style="list-style-type: none"> Basic support for relational databases: MySQL, SQL Server, PostgreSQL and others Data Editor SQL Editor Database schema editor DDL Basic ER Diagrams Basic charts Data export/import Task management Database maintenance tools 	<ul style="list-style-type: none"> All DBeaver Community features Advanced security Advanced support for relational databases Connection through ODBC drivers NoSQL databases support: MongoDB, Cassandra, Redis, CouchDB and others Cloud databases support: Redshift, Google BigQuery, Oracle Cloud and others Native support for AWS, Google Cloud, and Azure Cloud storage support Metadata management tools Database performance visual tools AI assistant in SQL Multi-component task management Task Scheduler Visual Query Builder Ongoing technical support
Download	Learn more

Figura 34 - Download do DBeaver Community e PRO

Abaixo é apresentada a Figura 35 com a tela de Download do **Windows (Installer)**.

Released on October 20th 2024 ([Milestones](#)).

It is free and open source ([license](#)).

Also you can get it from the [GitHub mirror](#).

[System requirements](#).

Windows

- **Windows (installer)**
- [Windows \(zip\)](#)
- [Chocolatey](#) (`choco install dbeaver`)
- [Install from Microsoft Store](#)

Mac OS X

- [MacOS for Apple Silicon \(dmg\)](#)
- [MacOS for Intel \(dmg\)](#)
- [Brew Cask](#) (`brew install --cask dbeaver-community`)

Figura 35 - Tela de download de instaladores do DBeaver

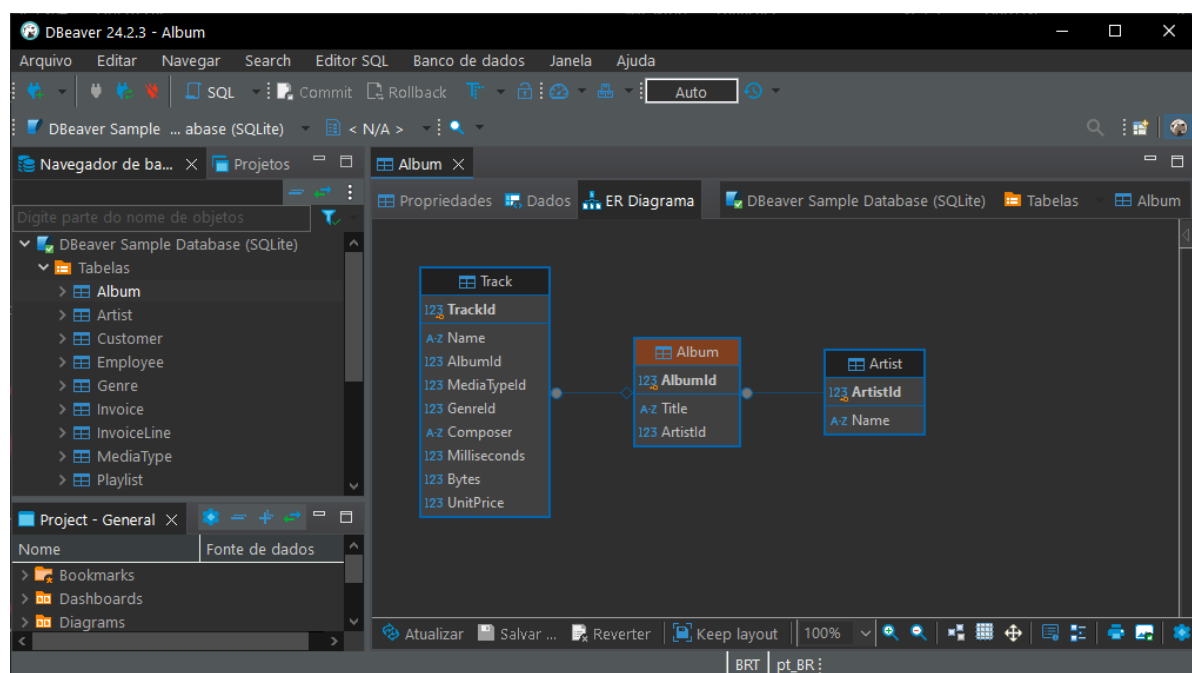


Figura 36 - Tela inicial do DBeaver após instalado

O processo de instalação segue o padrão anterior, clicar em **Next** até concluir a instalação. Após o aplicativo ser aberto, o DBeaver apresentará uma tela vazia e no painel lateral terá o banco de dados **DBeaver Sample Database** que é criado por padrão na instalação. Isto te ajuda a explorar o formato de elementos que são gerados na interface.

Clicando neste banco de dados, é aberto uma árvore de elementos pré-criados, como **tabelas**, **visualização**, **índices**, **gatilhos**, dentre outros. Para explorar as tabelas, clique em **tabelas** e uma nova árvore vai se abrir, apresentando as tabelas que já são criadas na instalação, para fins de estudos do DBeaver.

Clique duas vezes em uma das tabelas, como **Album** e na parte central teremos 3 abas principais: **Propriedades** – que apresenta o formato de colunas na tabela e seus tipos; **Dados** – Que são os dados previamente armazenados; E **ER – Diagrama** – sendo o diagrama Entidade-Relacionamento desta tabela Album relacionada com outras tabelas;

Quando queremos trabalhar com SQL, clicamos no menu **Editor SQL** na parte superior e **Abrir Console SQL**.

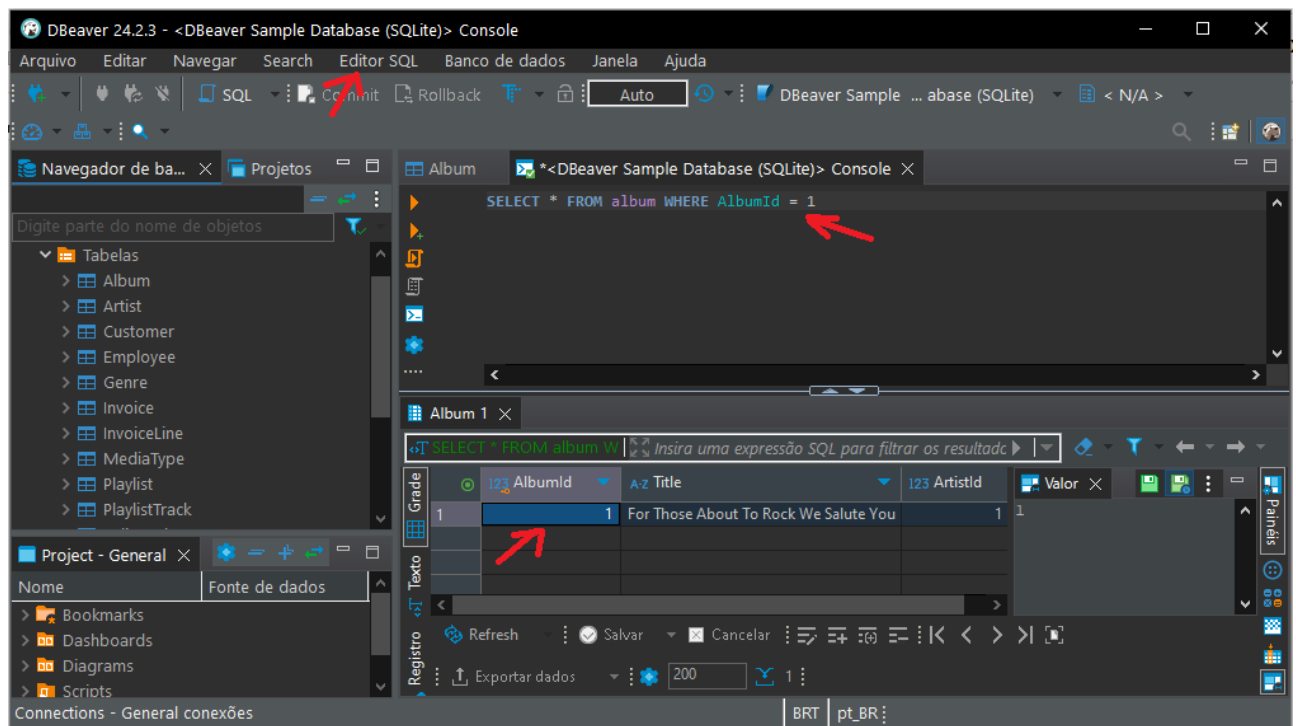


Figura 37 - Consulta SQL de SELECT no DBeaver para tabela Album

É aberto um editor central, onde podemos editar o código SQL para consultas e atualizações. Pedimos para selecionar da tabela Album onde o id do álbum é igual a 1, desta forma, no painel inferior é apresentada apenas a tupla do dado que contém o mesmo ID mencionado. Clicando no menu superior **Banco de dados** e depois em **Nova conexão**, é possível configurar conexões com novos bancos de dados, por exemplo: O PostgreSQL, SQLite, dentre outros. Se escolher o PostgreSQL, será solicitado um novo driver a ser instalado, mas é uma tarefa simples, pois a própria interface já te sugere o Driver e permite que você instale por ela mesma apenas clicando em **Download**. Feito isto, é só configurar a URL de conexão mencionada (que vai ter uma por padrão) e insira a senha "postgres", teste a conexão antes de finalizar e clique em **finish**. Isto demonstra o quão flexível é a ferramenta para se conectar a várias bases de dados e modelagem dos dados via ER.

3. DESENVOLVENDO UMA API USANDO SPRING MVC

Neste capítulo veremos a criação de uma API para cadastro de personagens usando o Spring MVC, simulando os dados de um game. Estes personagens serão bruxas e magos em homenagem ao dia de ontem, o dia das bruxas – 31 de outubro. Cada personagem terá um nome, sua skill e sua capacidade de danos, para isto criaremos uma entidade (modelo de domínio) que será mapeada para a tabela do banco de dados.

Antes de começarmos, iremos no ambientar no desenvolvimento de aplicações simples usando um projeto Maven, para entender como o Maven gerencia os pacotes de bibliotecas de forma automática. Aproveitaremos para apresentar os conceitos do Spring Boot na prática, como IoC e DI. Após isto, desenvolveremos uma aplicação simples para cadastro de um usuário para nos habituar na simplicidade da comunicação com o banco de dados.

3.1. Projeto Maven com Spring Initializr

Primeiramente, pesquisaremos o site **Spring Initializr** que facilita no processo de criação de um projeto Maven do Spring Boot. O Maven é um gerenciador de pacotes de bibliotecas Java, que permite automatizar o processo de inclusão de dependências em nossos projetos, tornando simples a tarefa de adicionar novas dependências.

Antigamente, ao se criar um novo projeto sem Maven ou Gradle (Um outro gerenciador), toda dependência era adicionada manualmente, isto é, para se instalar uma biblioteca, era preciso realizar inúmeras configurações, como fazer a pesquisa do JAR para download, baixar o arquivo, adicionar a biblioteca nas configurações da IDE e realizar o refresh (atualização) do projeto. Esta tarefa era árdua, ainda mais quando tínhamos inúmeros JARs para configurar.

O Maven utiliza uma sintaxe XML básica no qual a tarefa de adicionar bibliotecas é muito mais rápida, desta forma, existem sites como **mvn repository** que nos permite filtrar a biblioteca que queremos adicionar e ele já nos fornece o bloco XML pronto para adicionarmos no arquivo **pom.xml**. Toda vez que alteramos e salvamos este arquivo, o plugin do Maven automaticamente realizar o novo build do projeto, realizando o download e as devidas configurações com as novas bibliotecas.

Algumas IDEs fornecem por padrão a opção de gerar um projeto Maven direto pela IDE, como **Eclipse** e **IntelliJ**, no entanto, no VSCode seria preciso instalar extensões adicionais para ele proporcionar esta geração. Por motivos de facilidade, iremos utilizar um site chamado **Spring Initializr** que além de gerar todo o projeto com o arquivo pom.xml, ele também nos ajuda a pesquisar e adicionar as dependências muito utilizadas no Spring. Para isto, pesquise no google como na imagem abaixo:

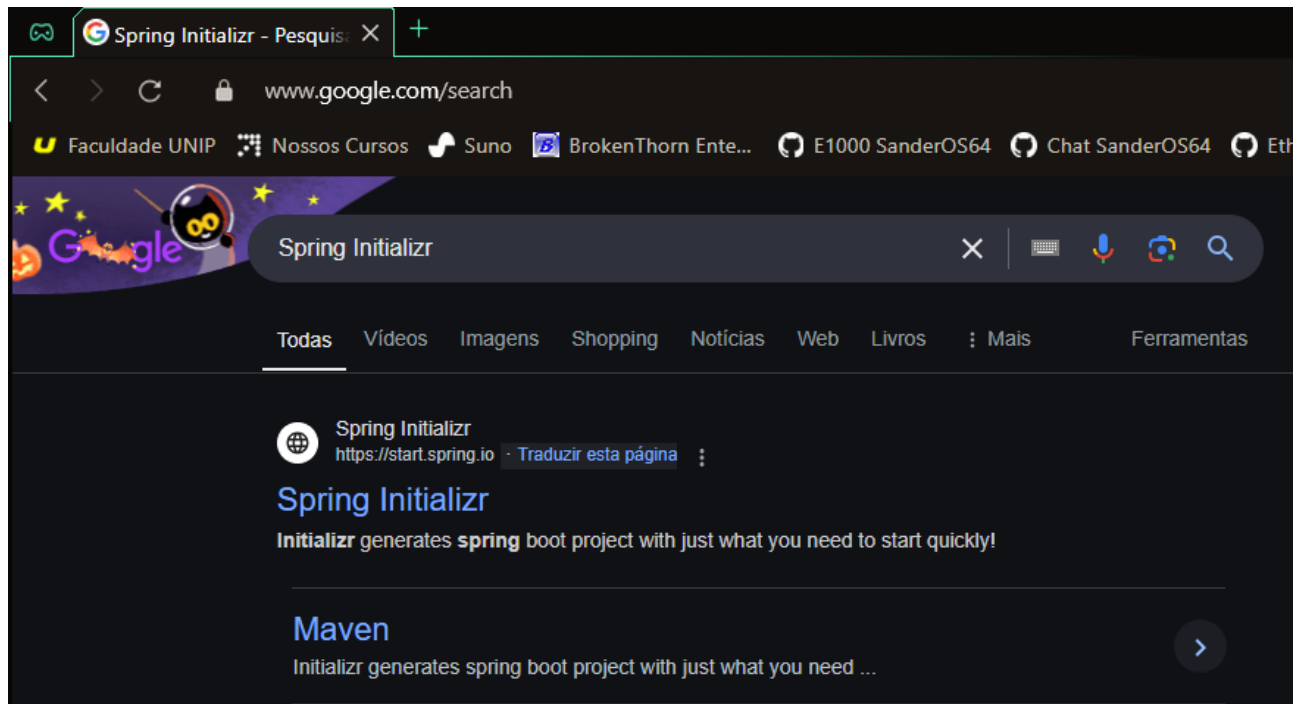


Figura 38 - Pesquisa do Spring Initializr no Google

Ao clicar no primeiro link da pesquisa, será redirecionado para uma nova página. Também poderá entrar diretamente pelo link <https://start.spring.io>.

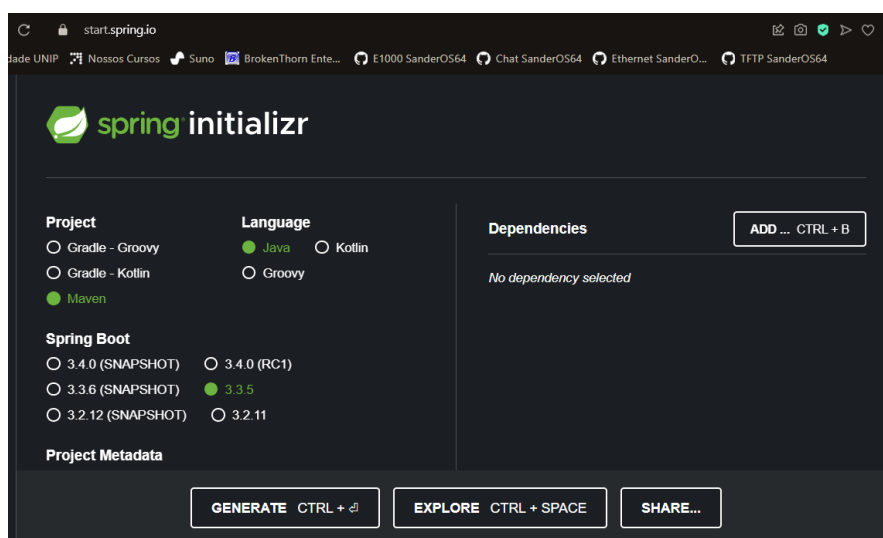


Figura 39 - Configuração do Projeto Maven no Spring Initializr

Ao entrar no site, é possível ver todas as configurações de uma maneira prática. O projeto é do tipo **Maven**, mas poderá ser criado outros tipos como **Gradle em Groovy** ou **Gradle em Kotlin**. O Gradle opera com uma linguagem mais estruturada, este tipo de projeto é muito utilizado em aplicativos mobile. Podemos trabalhar com as linguagens **Java**, **Groovy** ou **Kotlin**. Marcamos Java pois é a linguagem que vamos programar na API.

Utilizaremos a versão **3.3.5** do Spring Boot de forma inicial, no entanto, quando criarmos nossa API, vamos alterar esta versão no arquivo pom.xml, demonstrando a praticidade de trabalhar com o Maven e utilizando alguns processos compatíveis das versões anteriores. Perceba na lateral direita *no dependency selected* pois nenhuma dependência foi adicionada no botão “**ADD... CTRL + B**”. Podemos simplesmente clicar em **GENERATE** para gerar o projeto, **EXPLORE** para verificar o XML do arquivo pom.xml ou **SHARE** para compartilhar o nosso projeto.

Antes de gerar o nosso projeto, rolaremos um pouco para baixo para identificar como os metadados do projeto é configurado em **Project Metadata**:

Figura 40 - Metadados do projeto Maven do Spring Boot

Nos metadados temos alguns atributos essenciais para o gerenciamento de dependências. Estes atributos ficarão embaixo do bloco **<parent>** no início do arquivo pom.xml. Vamos compreender cada um destes atributos.

O **Group** é a base de um pacote Java, isto significa que se você alterar o Group, vai alterar automaticamente o campo **Package Name** que é o nome completo do pacote. Uma relação destes campos que temos no gerenciamento de dependências é que se você pretende criar uma biblioteca própria (dependência) para um outro projeto e querer disponibilizar para os desenvolvedores no site **mvn repository**, Os desenvolvedores terão que adicionar este mesmo nome de Group na tag **<groupId>** dentro do bloco **<dependency>** no arquivo pom.xml para utilizar sua biblioteca.

Neste caso eles colocariam **<groupId>com.example</groupId>**, pois o nosso Group é **com.example**. Já o **Package Name** é o nome completo que os desenvolvedores teriam que adicionar em seus códigos para utilizar nossa biblioteca após o build do Maven. Exemplo: **import com.example.demo.*;** neste exemplo, com o símbolo **'.'**, todas as classes que tiverem no pacote **"com.example.demo"** poderão ser importadas. No entanto, você poderá deixar seu Package Name = Group sem problemas, ou seja, **com.example**.

Já o Artifact é o **ID do Artefato**, que tem o nome de **"demo"**, significa que no exemplo da sua biblioteca, os desenvolvedores além de inserir o **groupId**, teria de adicionar abaixo uma nova tag chamada **artifactId**, com o mesmo nome que você definiu no artefato, exemplo: **<artifactId>demo</artifactId>**. O ID do artefato é uma subcategoria, especificando a identificação exata da sua biblioteca. Desta forma, poderão existir várias dependências com o mesmo **groupId**, mas com **artifactId** diferentes.

Veremos um exemplo de como duas dependências são adicionadas usando estes atributos, sendo elas o starter do Spring Test (para testes unitários) e do Lombok (para automação de getters e setters, dentre outros aspectos):

```
<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Figura 41 - Dependências do Spring Test e Lombok com groups e artifacts

Assim como nossos exemplos, o ID de grupo para a biblioteca Lombok é **org.projectlombok**, assim como colocamos **com.example** no projeto. O ID do artefato é apenas **lombok**, que em nosso caso é o ID **demo**.

E aqui entra um fator interessante, é que se você altera o valor “demo” no Artifact do seu projeto Maven no Spring Initializr, automaticamente ele altera o valor “demo” no atributo **Name**. No entanto, o oposto não acontece, isto é, você poderá colocar outro nome no Name, mas o Artifact se manterá intacto. Isto significa que o artifactId pode ter o mesmo valor do Name e por este motivo, os desenvolvedores poderão omitir a referência do name da tag **<scope>**, assim como vemos na dependência Lombok que só tem o artifactId.

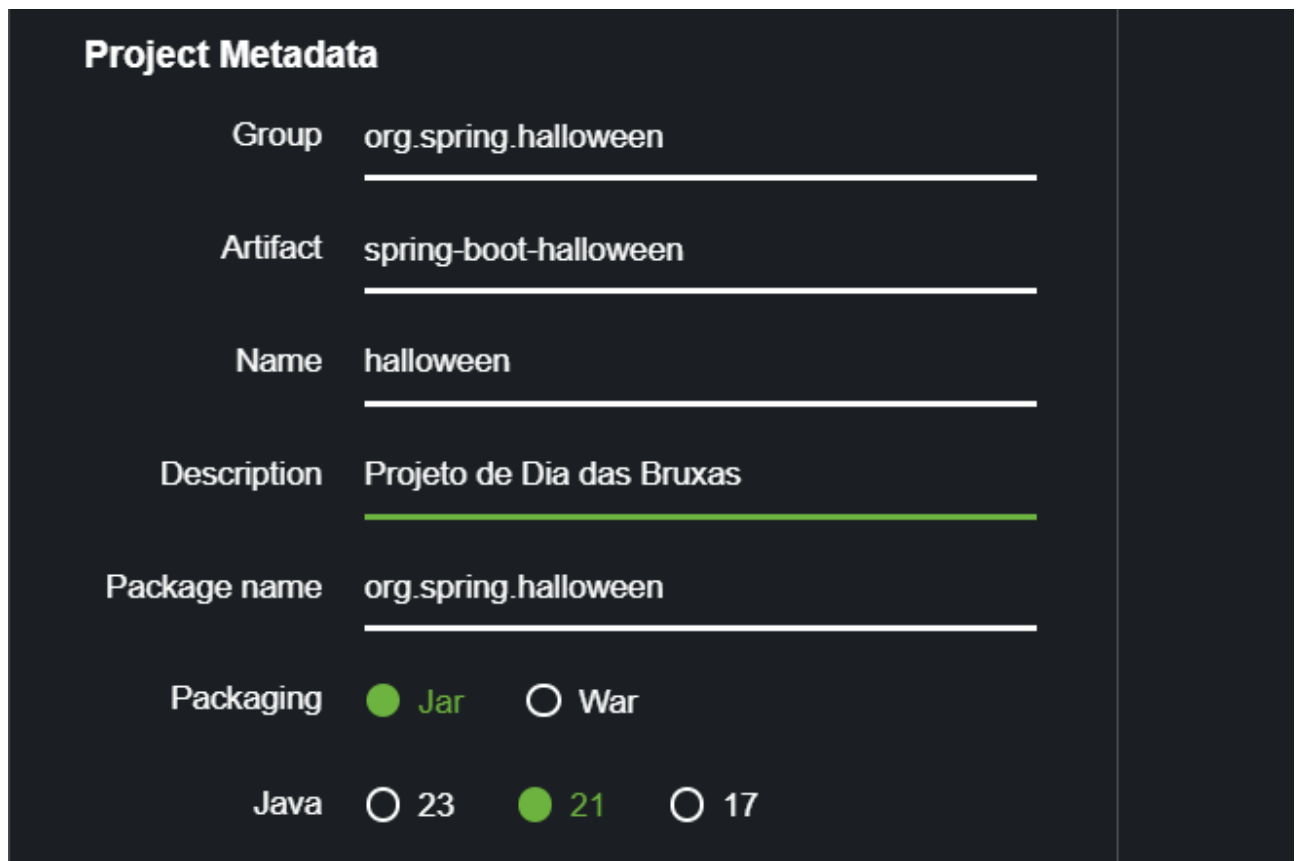
Já na dependência do Spring Test, usamos a tag **<scope>** com o valor **test**, isto é, o Name configurado nos metadados do projeto foi “test”, mas o Artifact configurado foi **spring-boot-starter-test**, sendo nomes diferentes. Em nosso caso, como o valor do Artifact é igual ao valor do Name (demo), os desenvolvedores ao adicionar nossa biblioteca fariam como apresentamos no Lombok, que é omitir o scope “Demo”. Já o atributo **Description** é uma breve descrição do que se trata sua biblioteca e os desenvolvedores poderão analisar nas configurações do projeto.

Ainda nos metadados temos o **Packaging** e a **Java**, no Packaging escolhemos em qual tipo de empacotamento a nossa JDK irá gerar, que no caso é o tipo **.JAR**, mas também poderá escolher o tipo **.WAR**. E no **Java** escolhemos a versão do Java que será utilizado, que poderá ser **23**, **21** ou **17**. Podemos alterar livremente esta versão no arquivo pom.xml após a geração do projeto, na tag **<java.version>** dentro do bloco **<properties>**. O properties é onde inserimos algumas variáveis de versões para inserir na tag **<version>** em algumas dependências, caso quisermos trabalhar com versões específicas.

Iremos alterar esta versão no java.version para **1.8** (Java 8) a fim de trabalharmos de forma fluída com a versão **2.5.4** do Spring Boot pois as novas versões incluíram algumas mudanças que iriam complicar no ensinamento deste E-book. Pra finalizarmos a explicação sobre versões, no arquivo pom.xml contém a tag **<version>** inserida junto com os metadados, porém esta versão não é configurada pelo Spring Initializr. Inicialmente ela vem com o valor **0.0.1-SNAPSHOT**. É possível alterar esta versão à medida que sua biblioteca evolui, assim os desenvolvedores poderão escolher qual versão usar da sua biblioteca através da tag **<version>** em **<dependency>**.

3.2. Projeto Spring sem Dependências

Para ilustrar tudo que já vimos até aqui, iremos gerar um projeto Maven com Spring Boot, no entanto, não utilizaremos nenhuma dependência/biblioteca por enquanto, apenas exploraremos o código inicial de um projeto Spring Boot e seus conceitos de injeção de dependências.



The image shows a 'Project Metadata' form with the following fields and values:

Field	Value
Group	org.spring.halloween
Artifact	spring-boot-halloween
Name	halloween
Description	Projeto de Dia das Bruxas
Package name	org.spring.halloween
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 23 <input checked="" type="radio"/> 21 <input type="radio"/> 17

Figura 42 - Metadados do novo projeto Maven sem dependências

As configurações iniciais como **Project**, **Language** e **Spring Boot** vamos deixar como está, que é o tipo **Maven**, **Java** e versão **3.3.5** do Spring, os valores padrões que o Spring gera como mencionado em figuras anteriores. Vamos apenas nos atentar as configurações de metadados, que terá o grupo **org.spring.halloween**, que será o mesmo nome do pacote (**Package Name**), o nosso artefato será **spring-boot-halloween** e o campo nome será diferente, sendo o valor **halloween**.

Também inserimos uma descrição simples “Projeto de Dia das Bruxas” com o empacotamento do tipo **Jar** e a versão do java **21**. Verifique no lado direito do Spring Initializr que nenhuma dependência foi adicionada. Portanto, agora podemos gerar o nosso projeto clicando sobre o botão **GENERATE...** abaixo dos metadados. Na próxima imagem, mostraremos o arquivo baixado como .ZIP.

Nome	Data de modificação	Tipo	Tamanho
spring-boot-halloween	01/11/2024 15:08	Pasta de arquivos	
spring-boot-halloween.zip	01/11/2024 15:08	Arquivo ZIP do ...	15 KB

Figura 43 - Arquivo baixado do Spring Initializr

O arquivo **spring-boot-halloween.zip** foi baixado automaticamente após gerar o projeto com o botão generate. Este arquivo foi baixado na pasta Downloads, no entanto, movi para as pastas de desenvolvimento do Ebook e extrai os arquivos zip, gerando uma pasta de mesmo nome.

Nome	Data de modificação	Tipo	Tamanho
.mvn	01/11/2024 15:08	Pasta de arquivos	
src	01/11/2024 15:08	Pasta de arquivos	
.gitattributes	01/11/2024 15:08	Documento de T...	1 KB
.gitignore	01/11/2024 15:08	Documento de T...	1 KB
HELP.md	01/11/2024 15:08	Arquivo Fonte M...	1 KB
mvnw	01/11/2024 15:08	Arquivo	11 KB
mvnw.cmd	01/11/2024 15:08	Script de Coman...	7 KB
pom.xml	01/11/2024 15:08	Microsoft Edge ...	2 KB

Figura 44 - Arquivos dentro da pasta do projeto Maven

Após abrir a pasta, verificamos que existem alguns arquivos essenciais para o gerenciamento do Maven, um deles é a pasta “.mvn” com valores de configuração de versão e distribuição de URL, a pasta “src” onde ficará nossos códigos fontes Java, arquivos do Git como .gitignore, .gitattributes e um arquivo Markdown (.MD), o próprio pom.xml que é onde ficam nossas dependências de bibliotecas e configuração do projeto, e por último, os Scripts CMD e Bash do Maven para o processo de configuração.

A seguir abriremos esta pasta “spring-boot-halloween” no VSCode e começaremos a explorar o nosso código Spring.

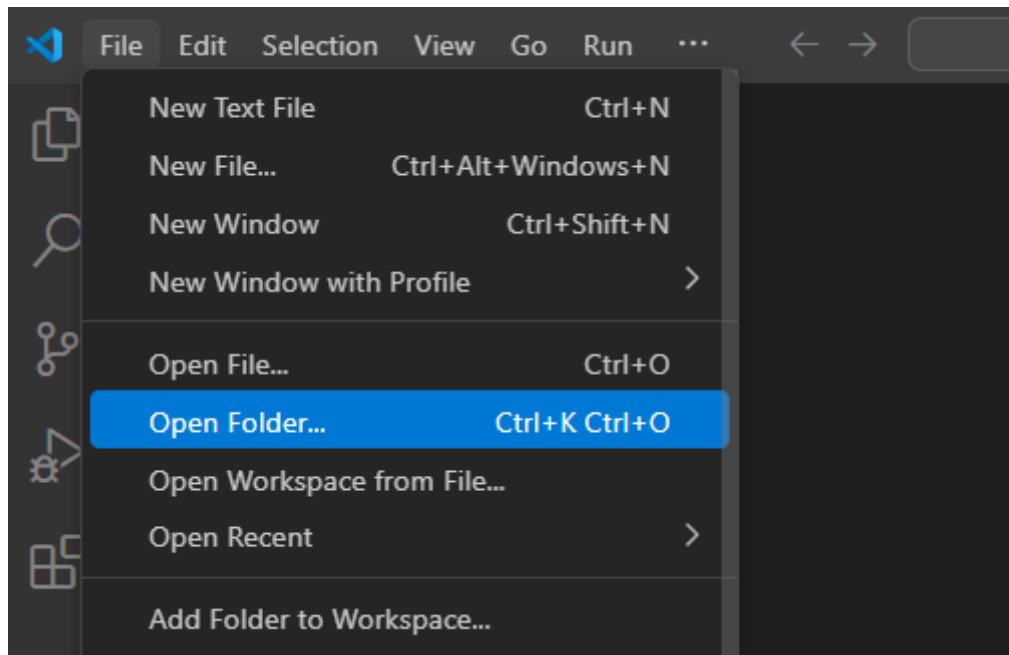


Figura 45 - Abrindo uma pasta de projeto no VSCode

Abra o VSCode e selecione o menu **File**, clique em **Open Folder**, após isto, localize e selecione a pasta **spring-boot-halloween**.

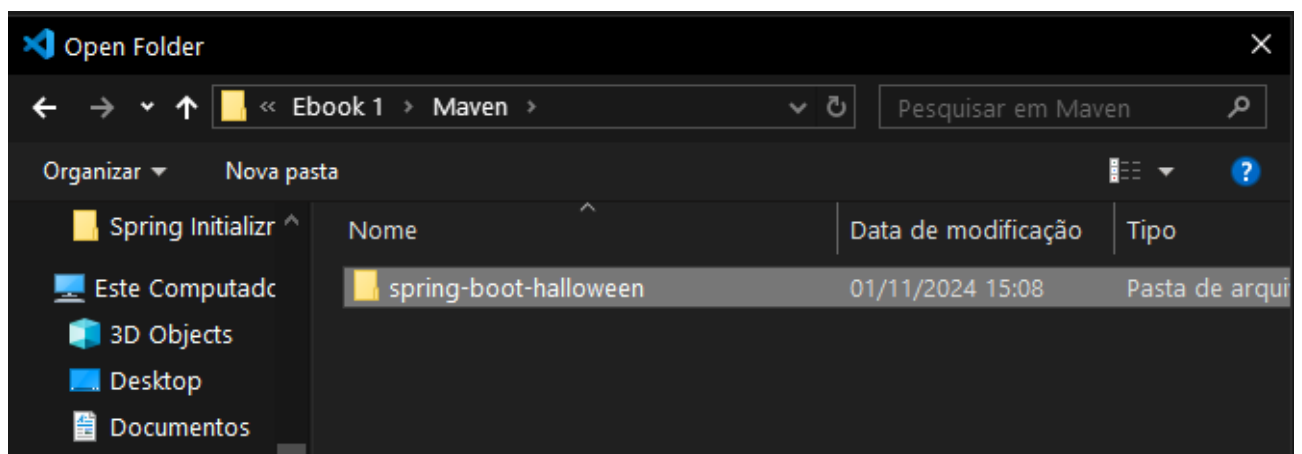


Figura 46 - Selecionando a pasta de projeto para abertura

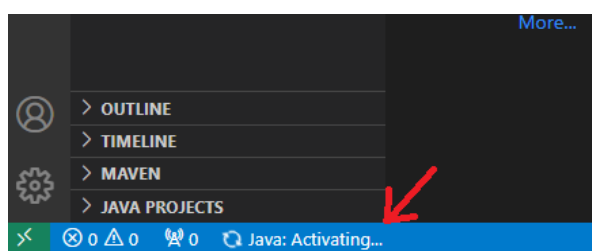
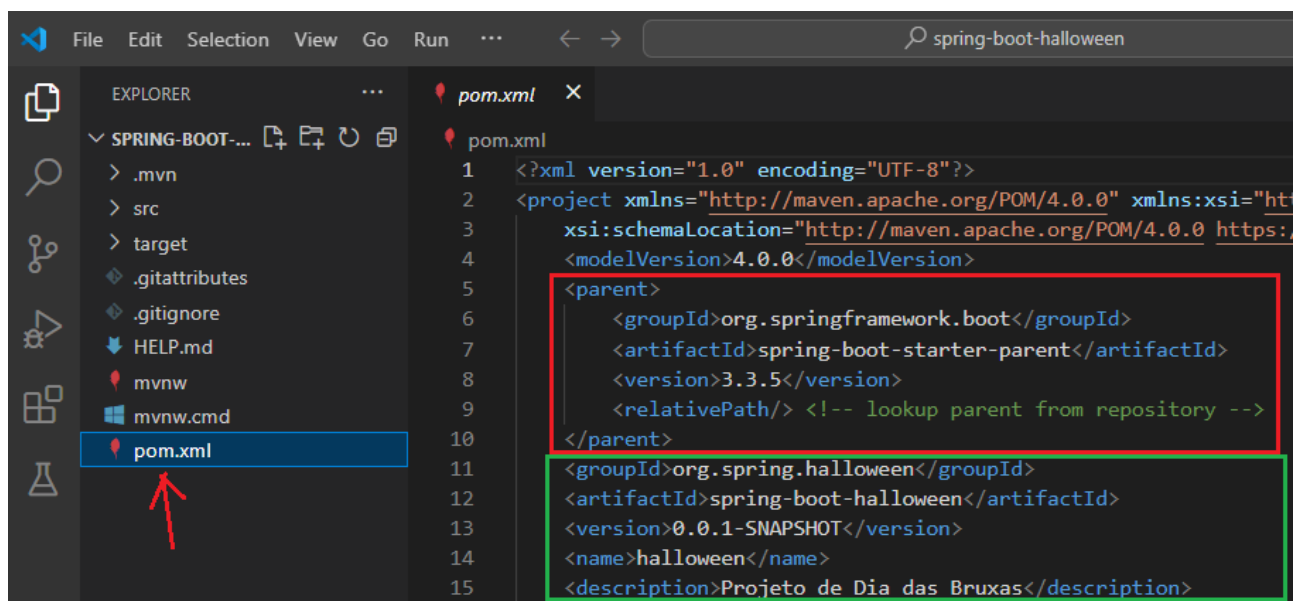


Figura 47 - Ativando as configurações do Java no projeto

Na imagem acima (Figura 47), analise as mensagens que aparecem de configuração do projeto, como **Java: Activating...** que ativa o Java nas configurações. E nesta região que apresenta o processo de configuração e builds do Maven. A cada mudança no arquivo pom.xml, é ocorrido um refresh no Build do Maven, apresentando neste canto inferior este processo de build.

Para demonstrarmos esta operação, verifique na imagem abaixo a árvore de diretórios aberta no VSCode (a mesma que mostramos na Figura 44) e o arquivo **pom.xml** aberto apresentando as primeiras propriedades.



Acima é apresentado a estrutura do diretório do projeto e também pode ser visto o arquivo pom.xml, com as primeiras propriedades. No destaque em vermelho, é o bloco **<parent>** ou “Pai” que define as propriedades do starter principal do Spring Boot, assim como sua versão. No destaque em verde, temos as propriedades do nosso próprio projeto (Repare nas semelhanças). O que vamos fazer é alterar a nossa versão **0.0.1-SNAPSHOT** para **0.0.2** e excluir tags que não são utilizadas, apresentadas logo abaixo:

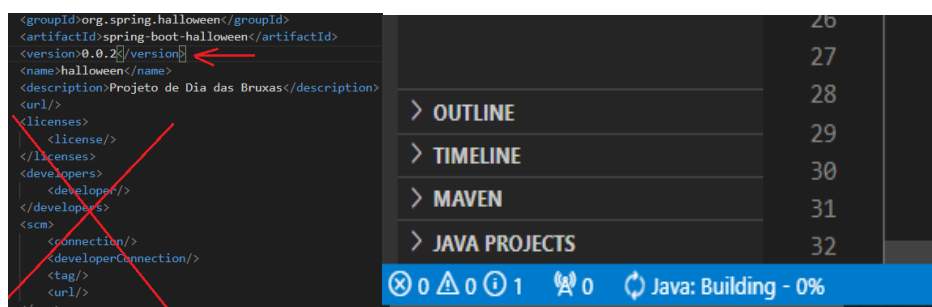


Figura 48 – Alterando a versão do projeto e excluindo Tags não utilizadas – Processo de Rebuild

Note que nós teremos um XML mais enxuto e limpo, apenas com as configurações necessárias e que também o VSCode ao identificar uma nova mudança (Após salvar com CTRL + S no arquivo pom.xml), o editor identifica estas mudanças e já faz o Rebuild. Da mesma forma quando adicionamos uma nova dependência Maven, o editor executa os Scripts do Maven deste projeto (mvnw.cmd) apresentando ao desenvolvedor o processo de download, instalação e configuração, porém de uma forma bastante rápida.

```
<properties>
|   <java.version>21</java.version>
</properties>
<dependencies>
|   <dependency>
|       <groupId>org.springframework.boot</groupId>
|       <artifactId>spring-boot-starter</artifactId>
|   </dependency>
|
|   <dependency>
|       <groupId>org.springframework.boot</groupId>
|       <artifactId>spring-boot-starter-test</artifactId>
|       <scope>test</scope>
|   </dependency>
</dependencies>
<build>
|   <plugins>
|       <plugin>
|           <groupId>org.springframework.boot</groupId>
|           <artifactId>spring-boot-maven-plugin</artifactId>
|       </plugin>
|   </plugins>
</build>
</project>
```

Figura 49 - Dependências e Plugin do Maven no POM.xml

Eis as configurações do pom.xml, no qual o maven vai gerenciar. O primeiro bloco é o **<properties>** que são variáveis que utilizamos para referenciar em outras partes do XML, útil para definir versões repetidas que serão utilizadas em várias dependências. O bloco **<dependencies>** é onde insere cada dependência com a tag **<dependency>**, seguindo as propriedades padrões. No bloco **<build>** serão ferramentas de construção, sendo plugins do Maven na IDE para gerir as dependências durante os Builds.

Abaixo veremos a árvore de diretórios do projeto, focada na aplicação Java Spring Boot, divididos em Sources, Resources, Tests & Targets.

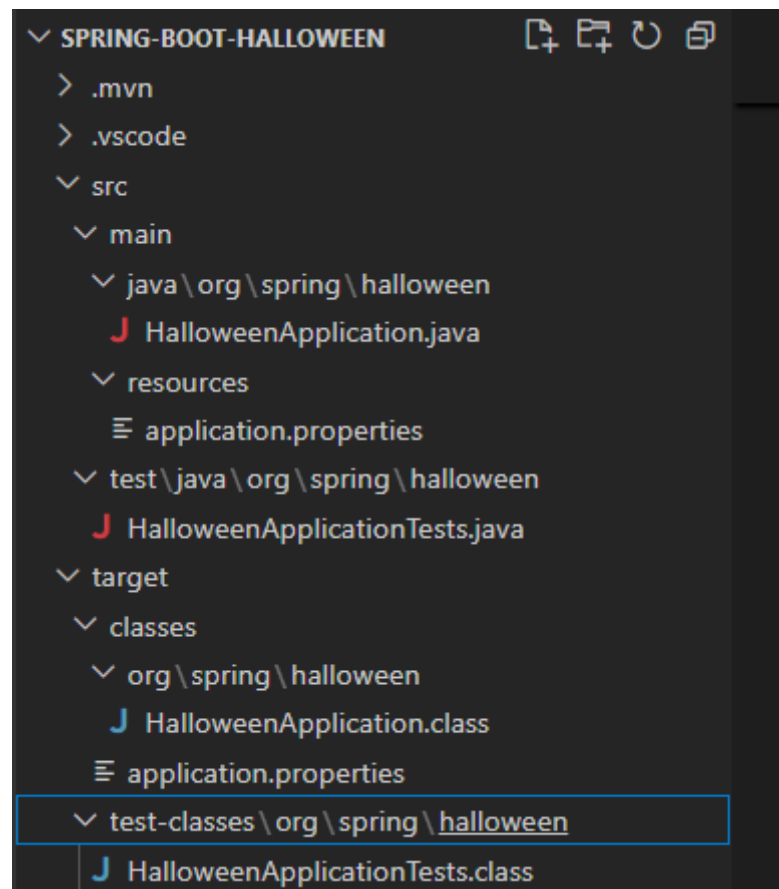


Figura 50 - Árvore do Projeto Spring Boot

A pasta **Src** (Source) é onde fica os códigos-fonte em Java, que é o código principal (main) e o código de testes unitários (test). No diretório “Main\Java” identificamos o diretório do pacote principal “org\spring\halloween”, que é o mesmo nome que definimos para o pacote no Spring Initializr. Voltando ao exemplo da biblioteca, se esta aplicação fosse uma biblioteca no qual os desenvolvedores iriam importar, eles teriam que utilizar o pacote **org.spring.halloween** onde todas as classes Java ficariam neste mesmo diretório.

No entanto, percebemos que existe uma pasta **target** com esta mesma estrutura de diretórios. O Target (alvo) é onde vai todo o código alvo, isto é, as compilações finais do código-fonte para arquivos `.class`, portanto, a nossa biblioteca que seria importada pelos desenvolvedores na verdade utilizaria o código das Classes em Target (os bytecodes) e não os códigos `.Java`.

Também temos a pasta **resources** que terão arquivos `.properties`, que são arquivos para configuração de propriedades da aplicação, como Strings de conexão ao banco,

constantes de variáveis, dentre outras configurações. Por último, nossos arquivos de testes unitários ficarão na mesma organização do pacote, porém na pasta **test**, tanto na pasta **src** quanto na pasta **target**.

O código-fonte principal da aplicação (Que vai se iniciar primeiro), é o **HalloweenApplication.java**, perceba que o nome da aplicação + “Application” se refere ao **Name** que definimos no Spring Initializr dos metadados. Da mesma forma que a aplicação de testes **HalloweenApplicationTests.java** contém o mesmo padrão de nome, adicionando + “Tests” no final do nome. Todo projeto Spring que for iniciado, terá esta mesma estrutura de diretório e padrões de nomes vindo dos Metadados.

Veremos abaixo qual é o conteúdo da aplicação principal do Spring **HalloweenApplication.java** e da aplicação de testes **HalloweenApplicationTests.java**:

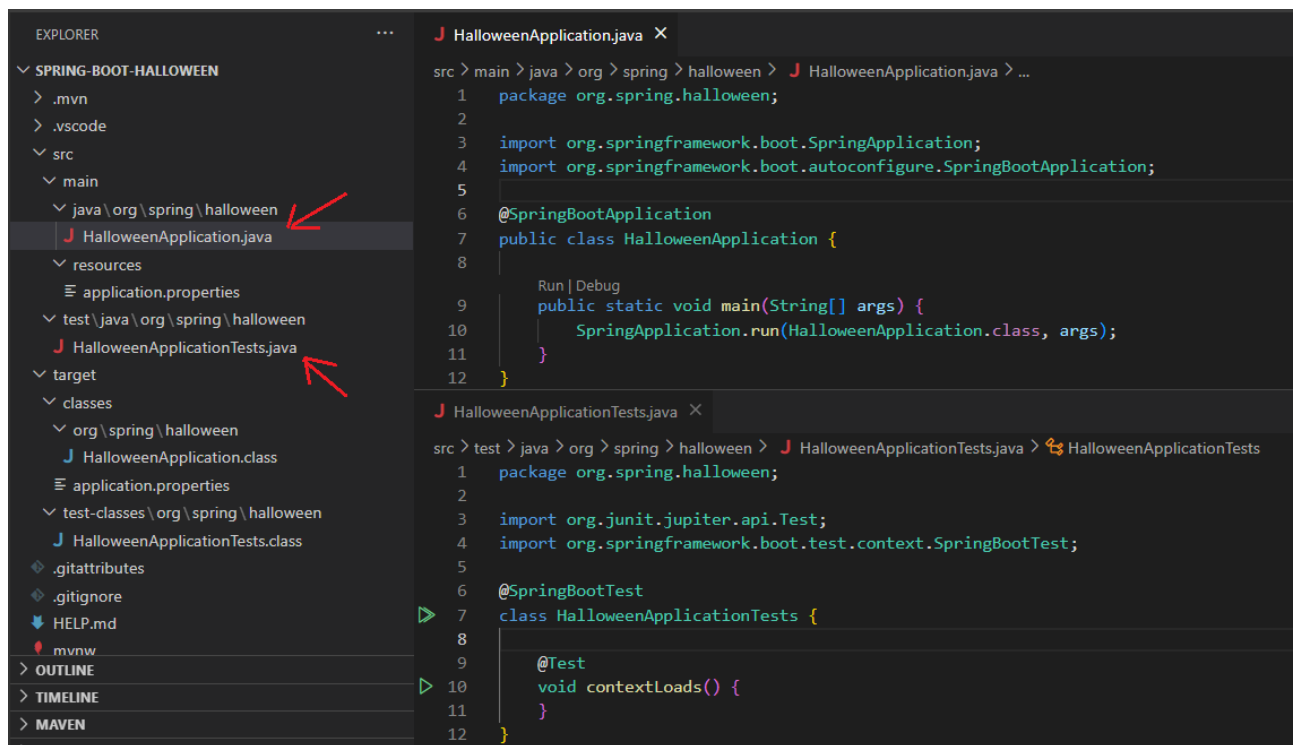


Figura 51 - Código Spring Boot Principal e Código Spring para Testes

O arquivo **HalloweenApplication.java** foi aberto clicando duas vezes sobre ele, mas o **HalloweenApplicationTests.java** foi arrastado para a parte inferior, desta forma, é possível segurar o mouse sobre o arquivo e arrastando para a posição onde quer deixar. Isto possibilita que você trabalhe com dois códigos ao mesmo tempo, um para testes e outro para implementação.

Primeiramente, definimos o pacote da aplicação **org.spring.halloween** com o comando **package**. Depois importamos os pacotes do Spring com o comando **import**, como pode ser visto, tanto a aplicação principal, quanto o de testes, contém o mesmo pacote **org.springframework.boot** que também é o groupId inserido nas dependências do Maven.

Na aplicação principal, na linha 3 vemos a importação direta da classe **SpringApplication**, no qual é utilizada dentro do método Main como uma classe estática, isto é, através do SpringApplication o método **run()** é chamado diretamente passando como parâmetros a classe da aplicação principal (HalloweenApplication) e o vetor de Strings **args** (do tipo `String[]`) no qual todos os argumentos de linha de comando são passados.

Como sabemos, o método Main é o primeiro a ser inicializado no programa, portanto, quando o programa iniciar, o método Run da classe SpringApplication vai executar um método interno do tipo **CommandLineRunner** que é uma interface que pega o contexto da aplicação em linha de comando, passa os argumentos CLI e administra todas as injeções de dependências (como instanciações automáticas de objeto) através da inversão de controle.

O método do Run() tipo de interface CommandLineRunner é um método que aceita sobreescrita (Override), quando qualquer subclasse “estender” e “implementar” a interface CommandLineRunner usando o comando **implements** (Estender pelo motivo da interface herdar de outra classe). Este é o famoso conceito de “Polimorfismo” em orientação a objetos, onde você pode sobrescrever com seu próprio algoritmo um método de uma classe herdada, sem utilizar a implementação original do método.

No entanto, para ocorrer de fato esta injeção de dependências pelo CommandLineRunner, é necessário especificar que sua classe principal é uma aplicação do tipo Spring Boot “autoconfigurável”, isto é feito importando a interface **SpringBootApplication** na linha 4 do pacote **autoconfigure** da mesma base de pacotes e utilizar esta interface como uma “Anotação” **@SpringBootApplication** em cima da classe principal.

O SpringBootApplication contém outras sub-anotações de configuração automática, configuração do Spring Boot, dentre outras características. Para estudar, experimente pressionando CTRL e clicando com o mouse em cima do nome “SpringBootApplication”

na linha 4, a IDE vai abrir esta classe e mostrar o seu código, você pode explorar e passar o mouse em cima das anotações encontradas nesta classe para verificar as informações de cada anotação de configuração.

Já na aplicação de testes na parte inferior, pode-se perceber que temos a importação da interface **SpringBootTest** que é utilizada como anotação acima da classe de testes, onde esta interface fica na mesma base de pacotes do springframework e nos subpacotes **test.context**. A outra importação é do JUnit para testes unitários em Java, através do pacote **org.junit.jupiter.api** no qual utilizamos a anotação **@Test** deste pacote.

Temos um método base chamado **contextLoads()** com a anotação de teste unitário, no entanto, não é preciso utilizar este método pois é possível criar um método com nome ultra descritivo para cada tipo de teste, especificando valores esperados e valores retornados usando **Assertions** e **Assumptions**. No entanto, como não é o foco deste E-book explicar sobre testes unitários, vamos apenas entender o porquê utilizamos uma anotação do Spring para testes e outra do JUnit.

Pense no seguinte cenário: Você precisa testar uma API para ver se ela está retornando dados JSON corretos dado valores esperados, no entanto, para este teste ser possível, é preciso instanciar uma série de objetos, realizar métodos Post HTTP (Ou Get), passar o corpo da requisição com dados em JSON e autenticação, dentre outras coisas. O SpringBootTest vai automatizar todas estas instâncias com a injeção de dependências e te fornecer métodos para realizar a requisição de maneira mais direta e fácil, enquanto que o JUnit vai proporcionar os métodos que validam os retornos esperados em JSON.

Sem o SpringBootTest, você precisaria implementar toda uma aplicação apenas para realizar um pequeno teste, algo que você já vai fazer de fato na implementação após os testes, logo o Spring automatiza esse processo. E sem o JUnit, você precisaria utilizar Logs estáticos no console dentro de condicionais IF e ELSE, tornando o processo de teste mais árduo. Outra vantagem é a execução dos métodos de testes de forma individual, sem a necessidade de subir toda uma aplicação para o servidor, que poderia deixar o processo de teste mais lento.

Portanto, o Spring tanto para a aplicação principal quanto para a aplicação de testes, te ajuda a gerenciar processos de execução que sem ele seria feito manualmente, aumentando na produtividade do desenvolvimento.

Vamos analisar a nossa aplicação, criando um simples Hello World usando o conceito de Beans. Também faremos outras análises recorrentes.

Adicione no seu HalloweenApplication.java o seguinte código:

```
8  @SpringBootApplication
9  public class HalloweenApplication {
10
11      Run | Debug
12      public static void main(String[] args) {
13          SpringApplication.run(HalloweenApplication.class, args);
14      }
15
16      @Bean
17      CommandLineRunner runner(){
18          return args -> {
19              System.out.println(x:"Hello World ");
20          };
21      }
22  }
```

Figura 52 - Aplicação Hello World usando Spring Boot

Falamos anteriormente que o CommandLineRunner é uma interface no qual contém um método Run() que pode ser sobrescrito quando usamos implements em uma Classe, mas isto é quando você opta por criar uma nova classe, que é o que faremos mais tarde. No exemplo da imagem acima, não precisamos sobrescrever o método Run, uma vez que o Spring já faz isto para gente através da anotação **@Bean**.

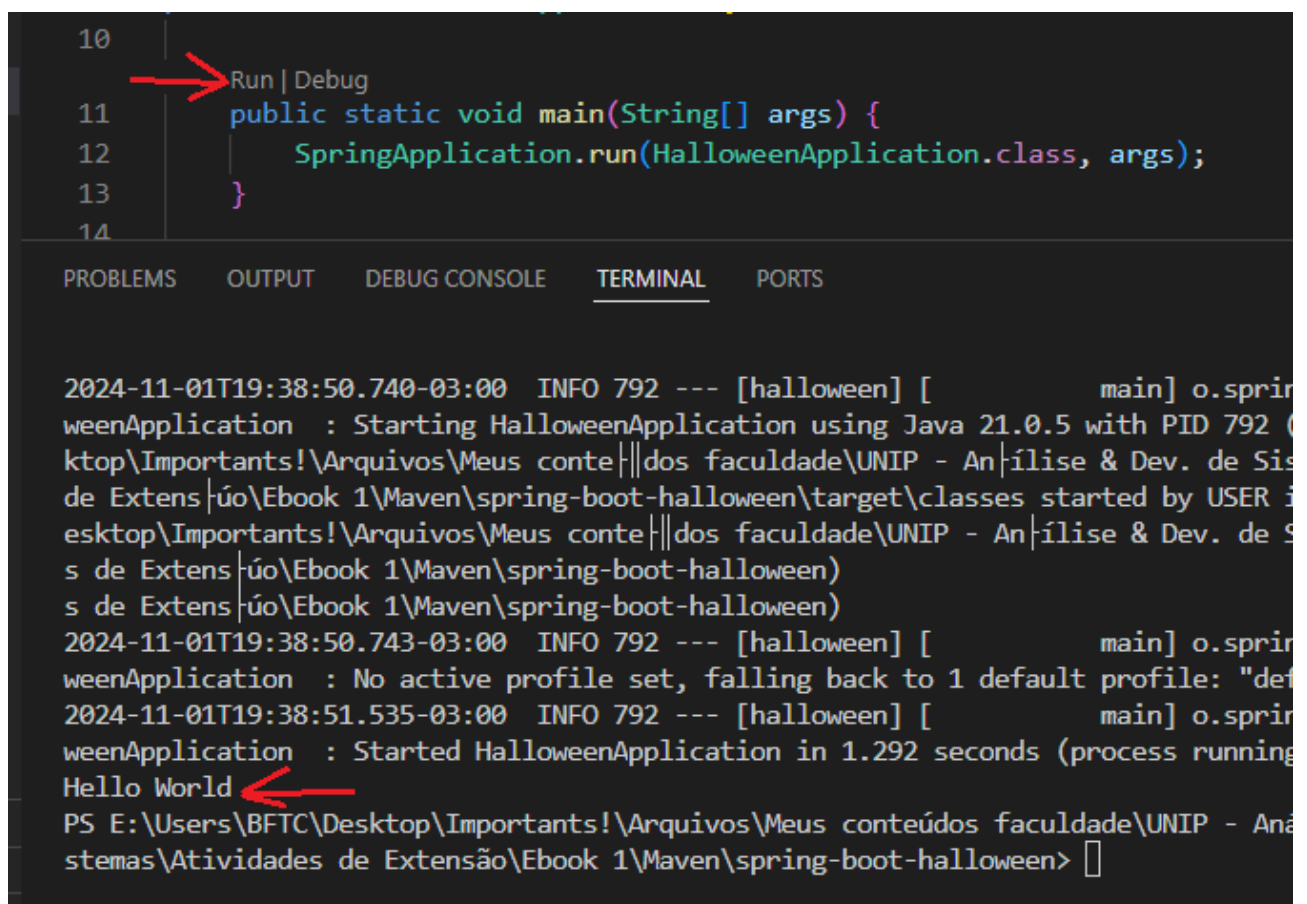
A aplicação Spring executaria sem esta anotação, no entanto, o método run não seria executado, já que não estaríamos especificando que o CommandLineRunner seria um componente gerenciado automaticamente pelo Spring. Definimos outro nome do método que é o “runner” pelo fato do Spring identificar o método Run associado ao tipo que ele retorna, ou seja, o próprio CommandLineRunner. Este retorno é mostrado na linha 17 a 19 e podemos perceber que uma função lambda está sendo chamada.

As funções lambdas contém o formato **() -> corpo_da_função**; onde o que fica entre parênteses é o argumento desta função e o que fica no **corpo_da_função**; é o código que irá ser executado pelo método Run (). Como o CommandLineRunner é apenas um contrato que especifica o método a ser executado, logo a real implementação

de chamada deste método está dentro do Spring Boot. No exemplo acima, não inserimos parênteses `()`, uma vez que ela é opcional quando a função lambda tem argumentos.

Dentro da chamada lambda, nós temos um print de um “Hello World” que irá ser executado. O método `runner ()` neste caso não contém argumentos, mas é possível passar qualquer tipo de variável e quantas variáveis quiser, incluindo objetos, faremos isto mais adiante. Qualquer objeto passado no argumento do `runner ()` seria automaticamente instanciado no corpo do método pelo Spring pelo fato deste método ser um Bean, na condição de que este objeto também seja um componente.

O **args** é um argumento da função lambda que vem do método `Run()` do `CommandLineRunner`, o `args` é do tipo `String[]` – Um vetor de Strings – que identifica as Strings de linha de comando. Para ilustrar isto, primeiramente vamos executar a nossa primeira aplicação `HelloWorld` pelo Spring Boot, basta clicar em “run” nas palavras **Run | Debug** que ficam entre a linha 10 e 11 do editor. Isto permite que o VSCode chame o executável Java para compilar e executar a nossa aplicação.



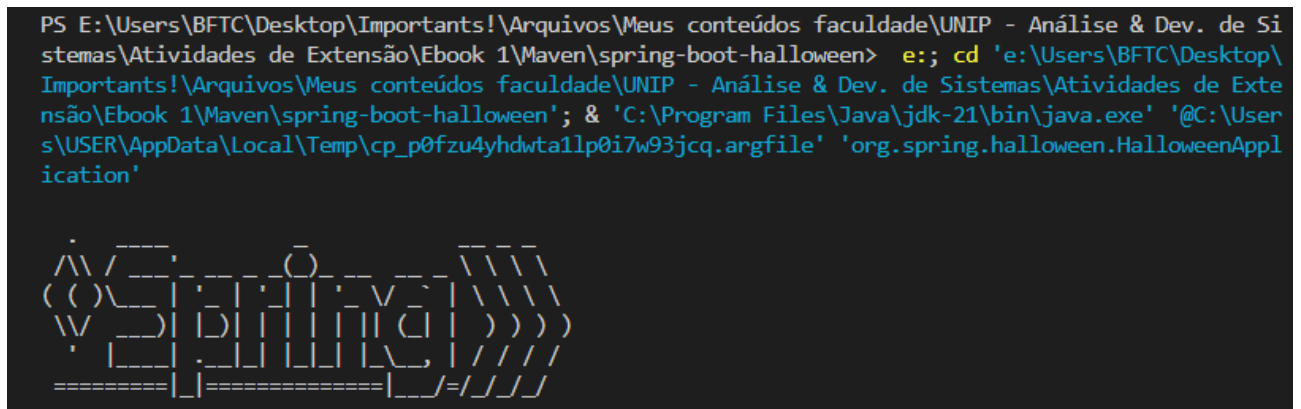
```
10
11  Run | Debug
12  public static void main(String[] args) {
13      SpringApplication.run(HalloweenApplication.class, args);
14  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
2024-11-01T19:38:50.740-03:00 INFO 792 --- [halloween] [           main] o.s.pri
weenApplication : Starting HalloweenApplication using Java 21.0.5 with PID 792 (
ktop\Importants!\Arquivos\Meus conte||dos faculdade\UNIP - An|ílise & Dev. de Sis
de Extens|úo\Ebook 1\Maven\spring-boot-halloween\target\classes started by USER i
esktop\Importants!\Arquivos\Meus conte||dos faculdade\UNIP - An|ílise & Dev. de S
s de Extens|úo\Ebook 1\Maven\spring-boot-halloween)
s de Extens|úo\Ebook 1\Maven\spring-boot-halloween)
2024-11-01T19:38:50.743-03:00 INFO 792 --- [halloween] [           main] o.s.pri
weenApplication : No active profile set, falling back to 1 default profile: "def
2024-11-01T19:38:51.535-03:00 INFO 792 --- [halloween] [           main] o.s.pri
weenApplication : Started HalloweenApplication in 1.292 seconds (process running
Hello World
PS E:\Users\BFTC\Desktop\Importants!\Arquivos\Meus conteúdos faculdade\UNIP - Aná
stemas\Atividades de Extensão\Ebook 1\Maven\spring-boot-halloween> □
```

Figura 53 - Executando a aplicação Hello World

A aplicação apresenta todos os logs de execução do Spring e no final a nossa string “Hello World”. Role o terminal para cima e observe o comando gerado em azul para compilar a aplicação acima do logotipo “Spring”:



```
PS E:\Users\BFTC\Desktop\Importants!\Arquivos\Meus conteúdos faculdade\UNIP - Análise & Dev. de Si
stemas\Atividades de Extensão\Ebook 1\Maven\spring-boot-halloween> e;; cd 'e:\Users\BFTC\Desktop\
Importants!\Arquivos\Meus conteúdos faculdade\UNIP - Análise & Dev. de Sistemas\Atividades de Exte
nsão\Ebook 1\Maven\spring-boot-halloween'; & 'C:\Program Files\Java\jdk-21\bin\java.exe' '@C:\User
s\USER\AppData\Local\Temp\cp_p0fzu4yhdwta1lp0i7w93jcq.argfile' 'org.spring.halloween.HalloweenApp
lication'
```

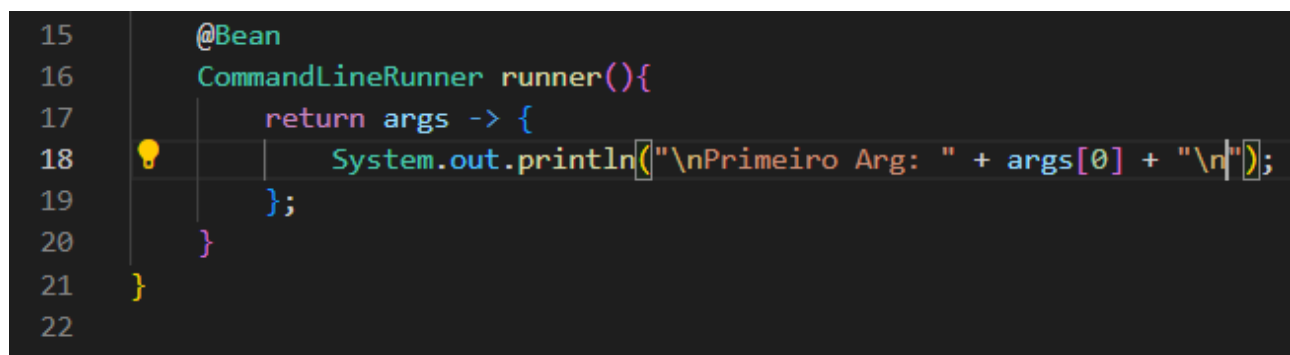
The Spring logo is displayed in a stylized, blocky font.

Figura 54 - Comando de compilação java da aplicação Spring

Note que ele executou o comando **cd** para navegar até o diretório que contém a pasta principal do projeto **spring-boot-halloween**, depois ele realizou o “&” que é uma conjunção de comandos, ou seja, “mais outro comando será executado”. Este outro comando é o executável **java.exe** que está na pasta padrão do jdk-21. Este executável está utilizando 2 parâmetros: Um arquivo temporário do tipo **.argfile** e a nossa classe principal **HalloweenApplication** que está no pacote **org.spring.halloween**.

Procure este arquivo temporário com o nome que foi dado e no diretório especificado e verifique seu conteúdo. Ele contém um parâmetro adicional chamado **-cp** que vai copiar para o projeto todos os jar e dependências utilizadas pelo Spring, incluindo o próprio Maven (Bibliotecas estas geradas pelas dependências do pom.xml). Para recuperar este comando, basta clicar na seta direcional **up** (para cima) dentro do terminal.

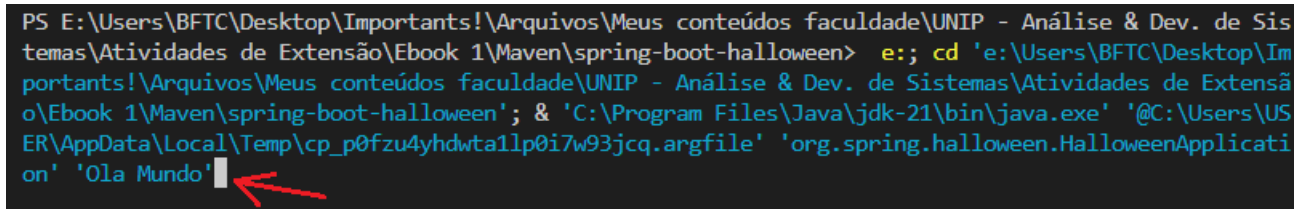
Vamos alterar o nosso código inserindo o argumento **args** no **println()**:



```
15     @Bean
16     CommandLineRunner runner(){
17         return args -> {
18             System.out.println("\nPrimeiro Arg: " + args[0] + "\n");
19         };
20     }
21 }
22
```

Figura 55 - Primeiro argumento apresentado no terminal

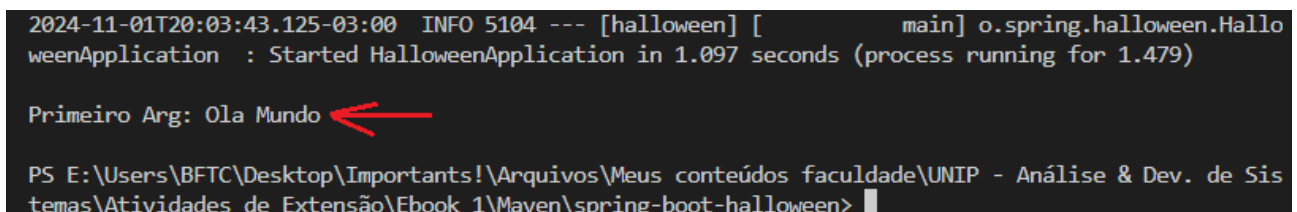
No código acima, alteramos o “Hello World” para “Primeiro Arg” com uma quebra de linha antes e concatenamos com o valor do índice 0 do vetor de String **args**. Também adicionamos uma quebra de linha após para facilitar na visualizar. Para executar, basta pressionar a seta para cima no terminal (se já não tiver clicado) e recuperar o comando anterior de compilação, então, adicione a String ‘Ola Mundo’ como na imagem abaixo:



```
PS E:\Users\BFTC\Desktop\Importants!\Arquivos\Meus conteúdos faculdade\UNIP - Análise & Dev. de Sis-
temas\Atividades de Extensão\Ebook 1\Maven\spring-boot-halloween> e;; cd 'e:\Users\BFTC\Desktop\Im-
portants!\Arquivos\Meus conteúdos faculdade\UNIP - Análise & Dev. de Sistemas\Atividades de Extensã-
o\Ebook 1\Maven\spring-boot-halloween'; & 'C:\Program Files\Java\jdk-21\bin\java.exe' '@C:\Users\US-
ER\AppData\Local\Temp\cp_p0fzu4yhdwta1lp0i7w93jcq.argfile' 'org.springframework.halloween.HalloweenApplicati-
on' 'Ola Mundo'
```

Figura 56 - Comando com o argumento da CLI

Pronto, agora basta pressionar Enter e terá o seguinte resultado:



```
2024-11-01T20:03:43.125-03:00 INFO 5104 --- [halloween] [main] o.springframework.halloween.Hallo-
weenApplication : Started HalloweenApplication in 1.097 seconds (process running for 1.479)

Primeiro Arg: Ola Mundo

PS E:\Users\BFTC\Desktop\Importants!\Arquivos\Meus conteúdos faculdade\UNIP - Análise & Dev. de Sis-
temas\Atividades de Extensão\Ebook 1\Maven\spring-boot-halloween>
```

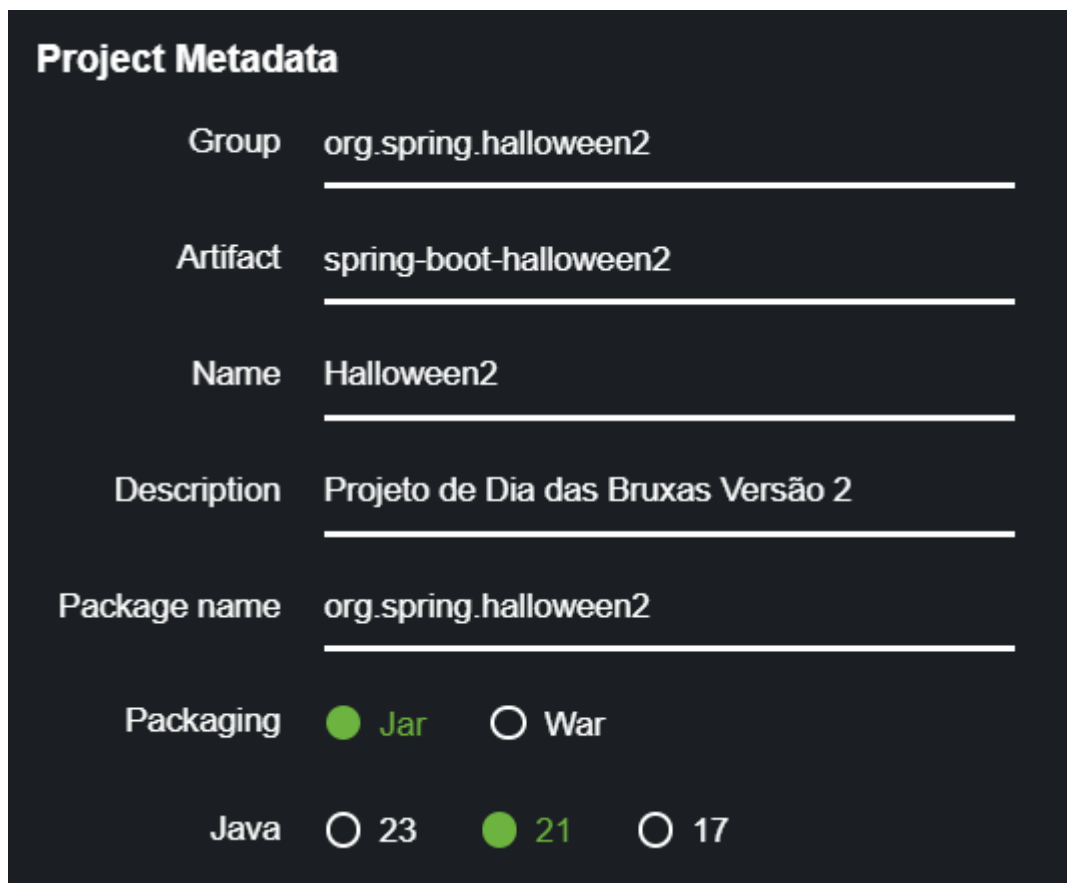
Figura 57 - Executando o Ola Mundo via argumentos

Desta forma, é possível visualizar facilmente nossas Strings de argumentos e assim conseguiríamos criar uma aplicação que explora cada argumento na linha de comando a fim de realizar uma tarefa para cada tipo de argumento. Esta é uma das missões do CommandLineRunner, pois ele é um executor de linha de comando.

3.3. Projeto Spring com Dependências

Neste tópico focaremos em algumas dependências principais como Lombok e H2 DataBase. O H2 é um banco de dados em memória, com os dados só existindo enquanto a aplicação estiver executando, facilitando no manejo de testes antes de trabalhar com um banco de dados real. O mesmo código se aplicará tanto para o H2 quanto para o PostgreSQL.

Também utilizaremos o CommandLineRunner em uma classe separada, para demonstrar os exemplos citados anteriormente do polimorfismo. Primeiramente, criaremos um outro projeto no Spring Initializr, no entanto, adicionando as dependências do Lombok e do H2 na imagem a seguir:



Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 23 ☒ 21 ☐ 17

Figura 58 - Metadados do novo projeto com dependências

Estes são os novos metadados, apenas coloque o número 2 na frente dos nomes, depois verifique que nenhuma dependência foi selecionada na imagem abaixo, então clique em “**ADD...**”.

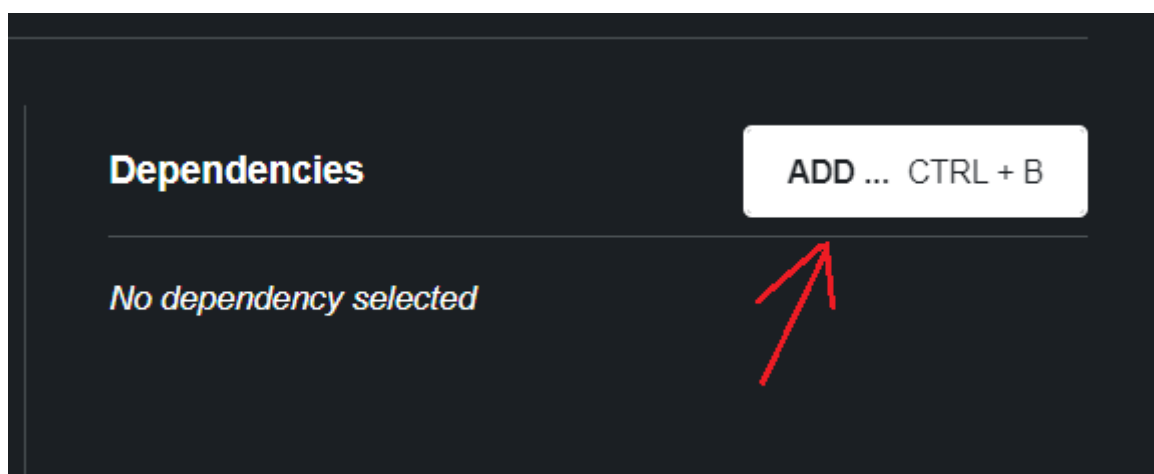


Figura 59 - Clicar no botão de adicionar dependências

Após clicar no botão, uma nova janela vai se abrir com algumas dependências padrões, apenas digite no campo de pesquisa a palavra **Lombok**:

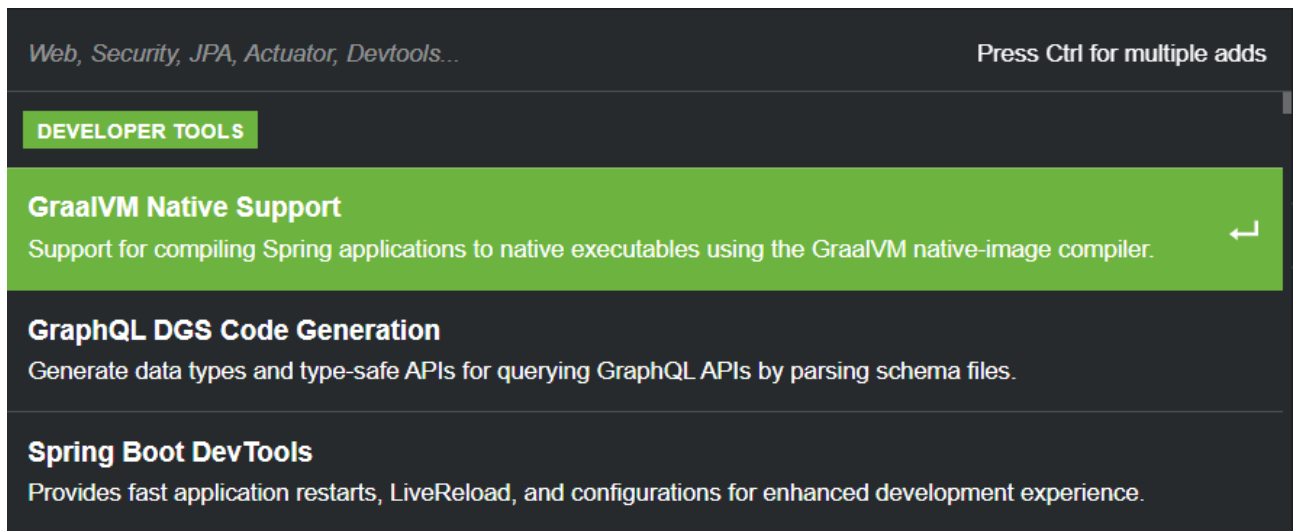


Figura 60 - Lista de dependências podem ser adicionadas

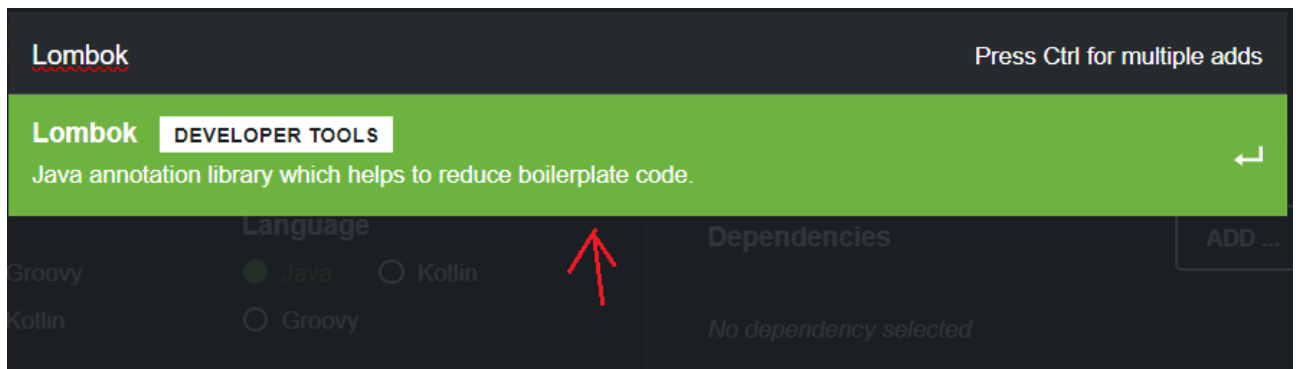


Figura 61 - Adicionando a dependência Lombok

Após clicar no retângulo verde da dependência, ela será automaticamente adicionada no pom.xml, então clique novamente sobre o botão **ADD...** e digite **H2**, adicionando a próxima dependência.

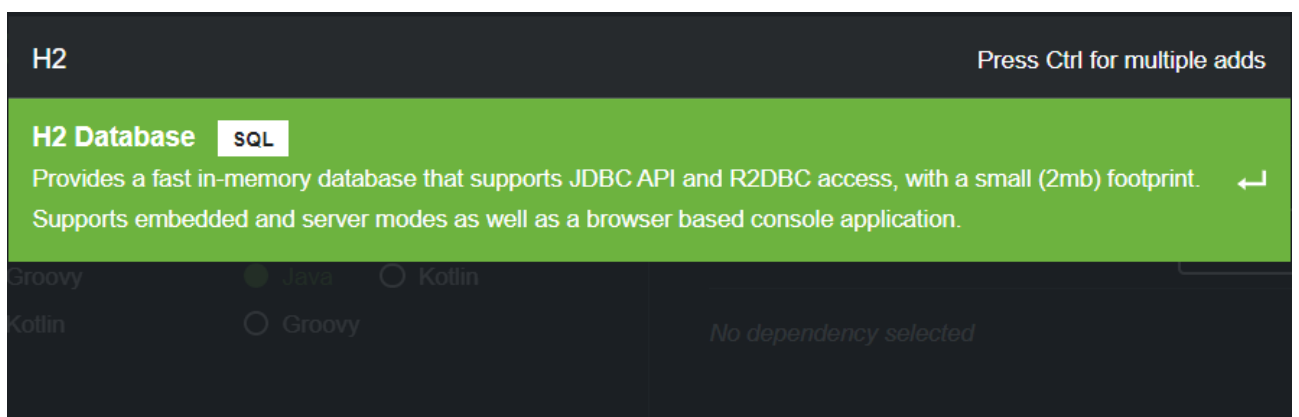


Figura 62 - Adicionando a dependência H2

Após clicar sobre o retângulo para adicionar a dependência, veja elas adicionada na lista:

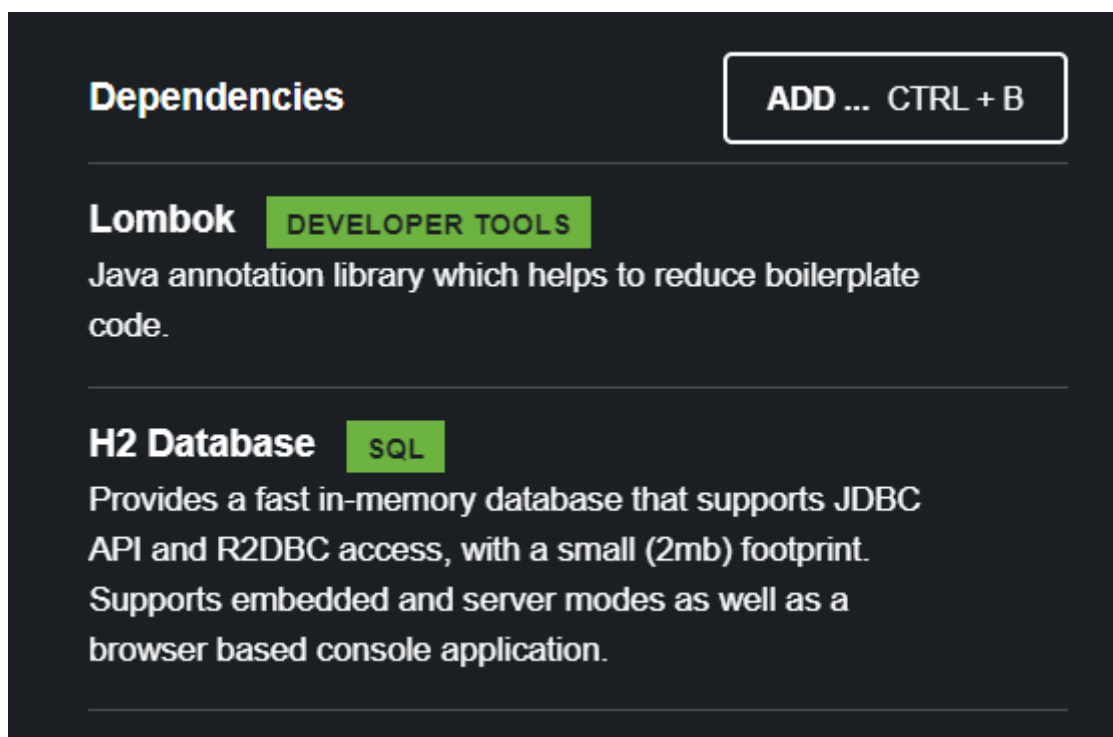


Figura 63 - Dependências adicionadas do Lombok e H2

Agora apenas clique no botão **GENERATE...** que vai gerar o seu projeto com estas dependências. Vamos analisar na próxima imagem o arquivo pom.xml no bloco **<dependencies>**.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Figura 64 - Dependências do Lombok e H2 no XML

Identificamos as dependências do banco de dados H2 no bloco **<dependency>** com o groupId **com.h2database**, da mesma forma o Lombok com o groupId **org.projectlombok**, as outras dependências são adicionadas por padrão na construção do projeto Maven pelo Spring Initializr.

No entanto, podemos perceber algo interessante dentro do bloco **<build>** na próxima imagem:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Figura 65 – Plugins do maven e exclusão do lombok na compilação do projeto

Aqui é demonstrado a lista de plugins, incluindo o primeiro plugin relacionado ao Maven (**spring-boot-maven-plugin**) e sua configuração que mostra as “exclusões” da biblioteca Lombok durante a construção (Build). O bloco **<exclude>** é para especificar que durante o Build da aplicação, o arquivo JAR do Lombok não será copiado para a aplicação, uma vez que o Lombok é uma biblioteca para agir apenas em tempo de desenvolvimento, e não de execução.

A essência do Lombok é gerar certos dados na aplicação que costumamos realizar de forma manual, no entanto, esta geração ocorre apenas do código-fonte Java para os códigos alvo .class, isto significa o projeto final de compilação será o mesmo código se não tivéssemos utilizado o Lombok. A missão desta biblioteca é agilizar no desenvolvimento, gerando dados como **Getters**, **Setters**, **toString ()** e demais outros métodos de entidades que criamos manualmente, tornando o código mais limpo.

Dentre outras funcionalidades, o Lombok também pode gerar logs customizados, mas no exemplo desta aplicação, vamos apenas utilizar uma anotação específica para mostrar a praticidade de se criar entidades. Antes disso, criaremos nossa primeira entidade dentro do pacote **org.spring.halloween**. Neste pacote, criaremos um novo pacote chamado **Model**.

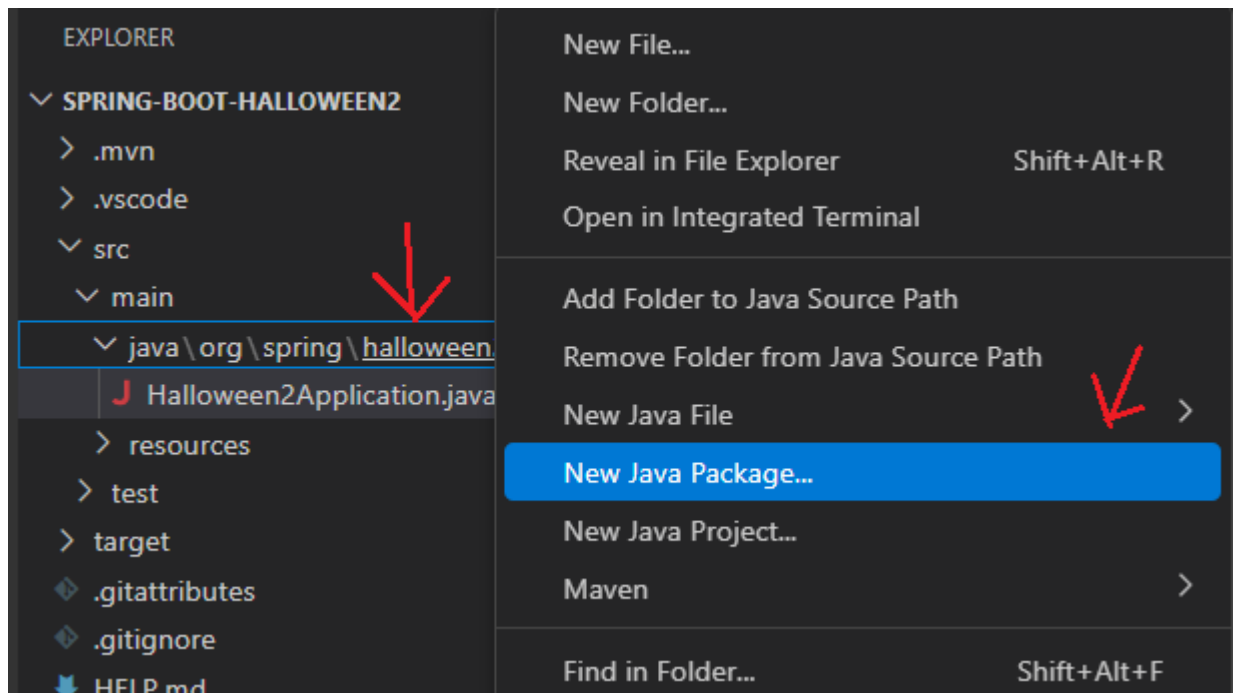


Figura 66 - Criando um novo pacote na pasta Halloween

Clique com o botão direito do mouse sobre a pasta **Halloween** e em **New Java Package**. Um novo campo é aberto para digitar o nome do pacote, digite **model**.

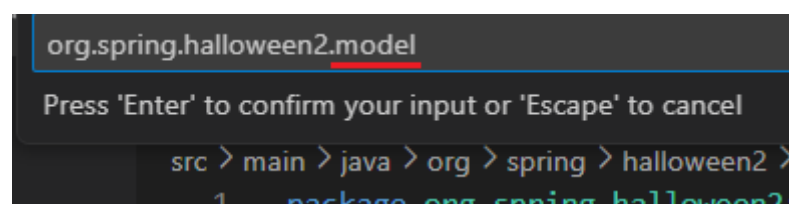


Figura 67 - Nomeando o novo pacote

Após dar enter, uma nova pasta “model” será adicionada no projeto, clique com o botão direito sobre ela, depois **New Java File** e por fim, **class....**

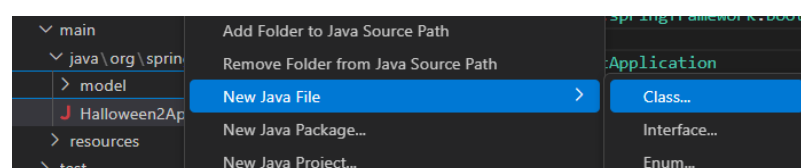


Figura 68 - Criando uma nova classe no pacote model

Após clicar em class, nomeie a nova classe de **User** (sempre se lembrar da primeira letra em maiúsculo) e confirme pressionando Enter.

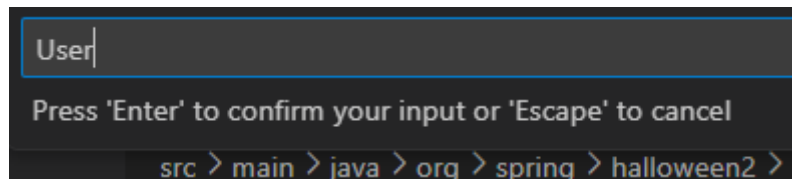


Figura 69 - Nomeando a classe para User

Abaixo está nossa primeira classe criada que está no pacote **org.springframework.halloween2.model**, da mesma forma que configuramos.

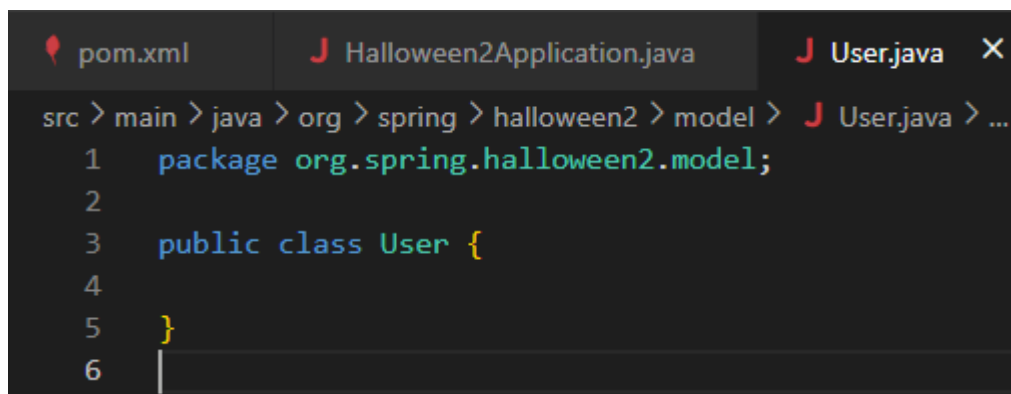


Figura 70 - Código vazio da classe User

Esta classe poderá ser do tipo **Entidade** que serve para modelar os dados de um domínio de negócio, mas vamos focar apenas em criar os atributos, seus getters e setters.

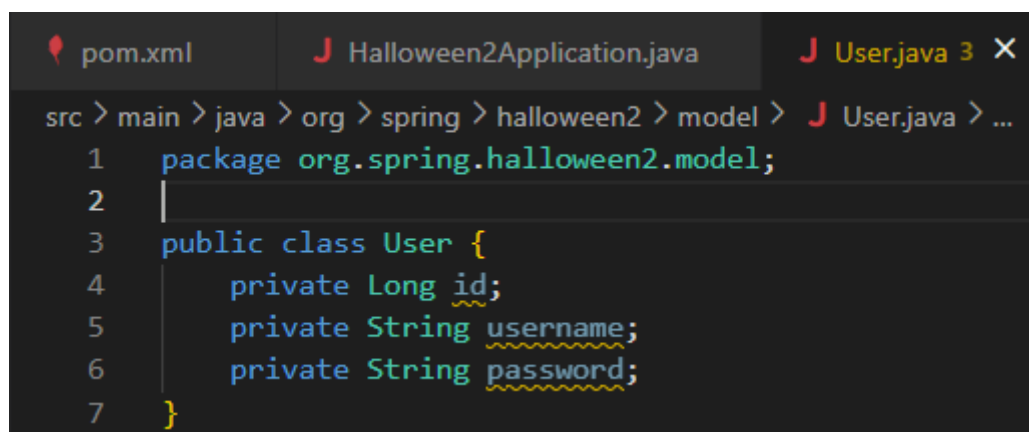
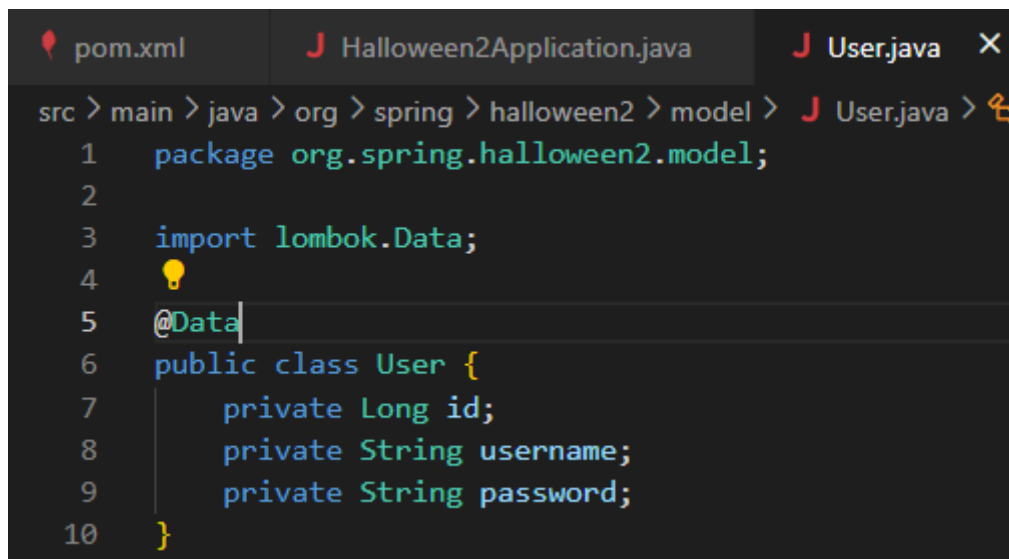


Figura 71 - Alerta da IDE com sublinhado amarelo

Após criarmos os atributos privados **id**, **username** e **password**, a IDE nos apresenta um alerta através do sublinhado amarelo. Não se preocupe, isto não é necessariamente um erro, no entanto, a IDE nos informa que estes atributos não estão sendo utilizados. Para observar isto, basta passar o mouse em cima dos atributos e a IDE te mostrará a informação.

A partir do momento que criamos os getters para retornar os valores como `getUsername()` e `getPassword()` ou os setters para definir os valores como `setUsername(string)` ou `setPassword(string)`, o código passa a utilizar os atributos de alguma forma, fazendo com que o alerta desapareça. Mas imagina que temos 10 atributos, precisaríamos criar 10 getters + 10 setters, ou seja, 20 métodos apenas para especificar um modelo de dados.

Neste quesito, a fim de reduzir exponencialmente um código que seria extremamente grande no cenário, bastaríamos utilizar a anotação **@Data** do Lombok, que além dele gerar os Getters e Setters, ele geraria outros métodos fundamentais como: `toString()` para retornar os valores do objeto em formato de String, `hashCode()` com o código HASH do objeto e até mesmo o `equals()` pra comparar objetos.

A screenshot of an IDE window showing the code for User.java. The file is located at src > main > java > org > spring > halloween2 > model > User.java. The code is as follows:

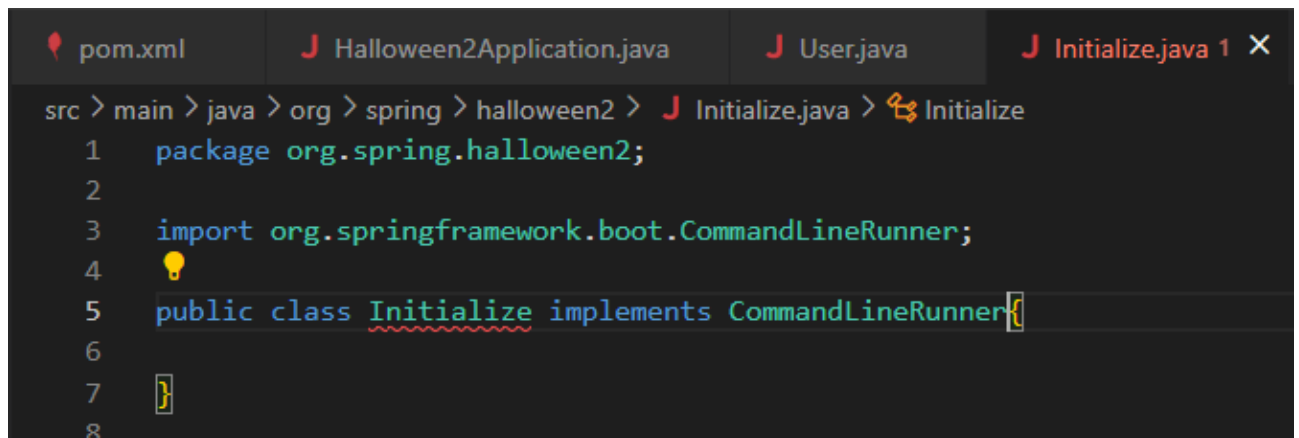
```
1 package org.spring.halloween2.model;
2
3 import lombok.Data;
4
5 @Data
6 public class User {
7     private Long id;
8     private String username;
9     private String password;
10 }
```

Figura 72 - Classe User com Getters e Setters gerados por Lombok

Desta forma, a IDE já importa para gente a anotação **Data** do **lombok**, e percebeba que o alerta amarelo sumiu, isto significa que os atributos estão sendo utilizados. Mas cadê os Getters? Pois então, estes métodos são gerados por detrás da cortina. Ou seja, o **Lombok** avisa a IDE que aqueles atributos conterão métodos que vão utilizá-los e na

compilação, gera estes métodos para o código final. Também é possível gerar métodos individuais com as anotações **@Getter**, **@Setter**, **@ToString** e **@EqualsAndHashCode**.

Agora iremos criar a classe **Initialize** dentro do pacote **org.springframework.halloween**:



```

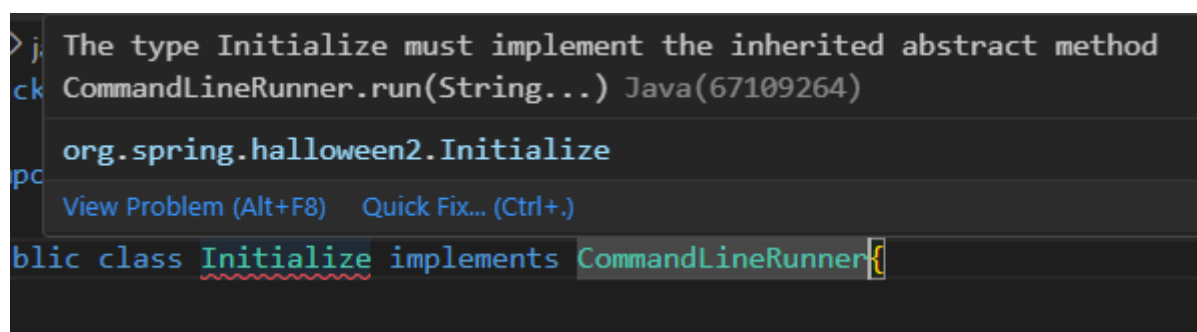
src > main > java > org > spring > halloween2 > Initialize.java > Initialize
1  package org.springframework.halloween2;
2
3  import org.springframework.boot.CommandLineRunner;
4
5  public class Initialize implements CommandLineRunner{
6
7  }
8

```

Figura 73 - Classe Initialize com um método não implementado

Apenas clique sobre o pacote **halloween** com o botão direito do mouse e clique em **New Java File** e depois **class...**, o mesmo procedimento que fizemos anteriormente, no entanto, no pacote base. A partir de agora, veremos os conceitos de injeção de dependência e os conceitos do polimorfismo que foi explicado antes.

Perceba que ao implementar a interface **CommandLineRunner**, a IDE apresenta um erro (sublinhado vermelho sobre a classe **Initialize**). Ao passar o mouse em cima, ele apresenta esta informação:



```

> j The type Initialize must implement the inherited abstract method
ck CommandLineRunner.run(String...) Java(67109264)
    org.springframework.halloween2.Initialize
    View Problem (Alt+F8) Quick Fix... (Ctrl+.)
blic class Initialize implements CommandLineRunner{

```

Figura 74 - Forma de resolver a adição do método

O erro diz que devemos implementar um método abstrato **...run (String...)**. Basta clicarmos em **Quick Fix...** e depois em **add unimplemented methods**, que a IDE irá rapidamente resolver este problema, adicionando o método não implementado.

```

1 package org.spring.halloween2;
2
3 import org.springframework.boot.CommandLineRunner;
4
5 public class Initialize implements CommandLineRunner {
6
7     @Override
8     public void run(String... args) throws Exception {
9         // TODO Auto-generated method stub
10        throw new UnsupportedOperationException(message:"Unimplemented method 'run'");
11    }
12
13 }
14

```

Figura 75 - Método run implementado do CommandLineRunner

Após a adição do método de implementação `run()`, percebemos algumas características que destacamos em momentos anteriores. Uma delas é o **@Override**, que especifica que este método poderá ser sobrescrito. Uma prova disso é o lançamento de uma exceção **UnsupportedOperationException**, com uma String de mensagem de método não implementado. Podemos substituir este código pelo nosso próprio.

A outra característica é o argumento **args** do tipo String, desta forma, podemos fazer igual fizemos na função lambda, de ler argumento por argumento da linha de comando, no entanto, utilizando outra forma que é diretamente pelo método. A notação **String...** indica que os argumentos serão recebidos em um Array de Strings. Vamos adicionar um código instanciando a classe User e armazenando seus dados.

```

@Override
public void run(String... args) throws Exception {
    User user = new User();
    user.setId(id:1);
    user.setUsername(username:"Wenderson");
    user.setPassword(password:"1234");
}

```

Figura 76 - Erro ao definir um número para o tipo Long

Perceba que nós temos um erro apontado por **setId**, se passarmos o mouse, o erro especifica que o argumento 1 não é aplicável para objeto **Long** que é uma classe para números longos. Existem duas formas de resolvermos isto: Ou realizamos um cast do número 1 para **long** (o tipo nativo da linguagem e não a classe **Long**), ou trocamos o tipo do atributo **id** de **Long** para **long**. Abaixo apresentamos as duas formas:

```
@Override
public void run(String... args) {
    User user = new User();
    user.setId((long) 1);
    user.setUsername(username: "Wenderson");
    user.setPassword(password: "1234");
}

@Data
public class User {
    private long id;
    private String username;
    private String password;
}
```

Figura 77 - Alterando para o tipo primitivo para long

Desta forma, é possível ver a resolução do problema usando uma das duas soluções, não é preciso utilizar as duas ao mesmo tempo. Mas e se quisermos que o Spring gerencie a dependência de instânciação? A maneira de se fazer isso é transformando a classe **User** em um componente, adicionando a anotação **@Component**. E após isto, informar ao Spring que na classe Initialize, vamos injetar a dependência toda vez que o tipo User for referenciado, isto é feito pela anotação **@Autowired**. Veja a seguir:

```
import org.springframework.stereotype.Component;
import lombok.Data;

@Component
@Data
public class User {
    private long id;
    private String username;
    private String password;
}

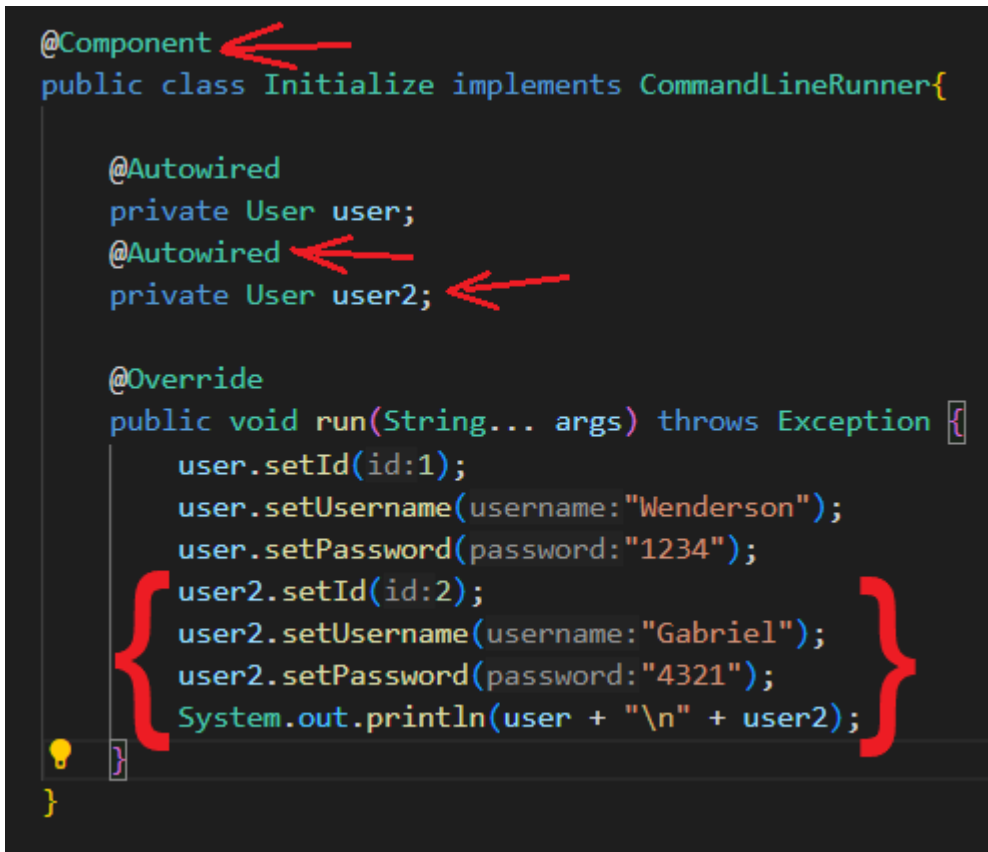
@Autowired
private User user;

@Override
public void run(String... args) throws Exception {
    user.setId(id:1);
    user.setUsername(username:"Wenderson");
    user.setPassword(password:"1234");
}
```

Figura 78 - Componente User auto injetável pelo Spring

Agora nossa classe User é um componente auto injetável, onde toda vez que formos utilizar, não precisaremos mais instanciar, o Spring cria este objeto na memória de forma automática. Para compreendermos o gerenciamento destes objetos, vamos apresentar mais um conceito que é o Scope, antes de partirmos para a persistência dos dados.

Em tese nosso programa já funciona, no entanto, para funcionar completamente, precisaríamos adicionar a anotação **@Component** em cima da classe Initialize. Feito isto, para vermos as informações do objeto, apenas adicione um **System.out.println()** para visualizar os dados. Desta forma, será mostrado as mudanças que fizemos na classe Initialize, a fim de apresentar o conceito de Scopes.



```

@Component
public class Initialize implements CommandLineRunner{

    @Autowired
    private User user;
    @Autowired
    private User user2;

    @Override
    public void run(String... args) throws Exception {
        user.setId(id:1);
        user.setUsername(username:"Wenderson");
        user.setPassword(password:"1234");
        user2.setId(id:2);
        user2.setUsername(username:"Gabriel");
        user2.setPassword(password:"4321");
        System.out.println(user + "\n" + user2);
    }
}

```

Figura 79 - Dois objetos Singleton auto injetáveis definindo dois dados

Após adicionar a anotação `Component` dizendo que `Initialize` será gerenciado pelo Spring e também o novo `Autowired`. Vamos pensar o seguinte: Olhando para este código, fica a parecer que estamos criando dois objetos, certo? Então, isto depende do tipo de escopo. Na primeira vez, armazenamos o `id=1`, o nome e a senha, em um objeto chamado **user**, e na segunda vez, armazenamos o `id=2`, com nome e senhas diferentes do primeiro, mas desta vez, no objeto chamado **user2**.

Estamos imprimindo os dados de **user** e **user2** separados por uma quebra de linha (o Lombok faz com que os objetos sejam retornados como `String` no console pelo método `toString()`). Ambos os objetos privados, são auto injetáveis pelo `Autowired`, mas será que eles são objetos separados em memória? Bom, se a classe `User` for do Escopo **prototype**, sim! Mas se for do escopo **singleton**, não!

Por padrão, toda classe é do escopo `Singleton`, pois quando você cria mais de uma variável com o tipo da classe, todas estas variáveis apontaram para um mesmo objeto em memória, devido a instanciação ser feita apenas uma vez no padrão `Singleton`. Se alterar o dado de um objeto, alterará de todos. Vamos executar nossa aplicação e ver os resultados sem definir o `Scope`.

```

. Started halloweenApplication in 1.719 seconds (process
User(id=2, username=Gabriel, password=4321)
User(id=2, username=Gabriel, password=4321)
PS E:\Users\BFTC\Desktop\Importants!\Arquivos\Meus conteúdos
s de Extensão\Ebook 1\Maven\spring-boot-halloween2>

```

Figura 80 - Informações de um mesmo objeto, porém duas variáveis

Note que o segundo dado substituiu o primeiro e quando exibimos o objeto user e user2, na verdade estamos exibindo apenas um objeto em memória, pelo fato de ser do tipo Singleton. No entanto, se precisarmos criar uma nova instância, para cada vez que referenciamos esta classe, basta adicionar a anotação **@Scope** com o tipo **prototype** na classe onde deseja este comportamento.

```

@Component
@Data
@Scope("prototype")
public class User {
    private long id;
    private String username;
    private String password;
}

```

Figura 81 - Transformando a classe em escopo prototype

Sendo a classe do tipo prototype, agora veremos o comportamento no console:

```

User(id=1, username=Wenderson, password=1234)
User(id=2, username=Gabriel, password=4321)
PS E:\Users\BFTC\Desktop\Importants!\Arquivos\Meus con
s de Extensão\Ebook 1\Maven\spring-boot-halloween2>

```

Figura 82 - Apresentando informações individuais do escopo prototype

Agora as informações são completamente individuais, pois a classe sendo do escopo prototype, permite que a cada referência da classe, seja uma nova injeção de uma instância individual em memória.

Agora veremos como nossa aplicação vai persistir estes dados, para isto, precisamos adicionar uma nova dependência – Spring Data JPA. É através da coleção de interfaces do JPA, que iremos utilizar a implementação pronta do Hibernate, ainda usando o banco de dados em memória (H2 database).

3.4. Aplicação de Cadastro de Usuários

Nos tópicos anteriores, vimos a vantagem de se trabalhar com o Spring Boot e o Lombok, tendo uma auto configuração e gestão de dependências, incluindo a produtividade de declarar classes de domínio sem inserir códigos extensos. Neste tópico, veremos a adoção do JPA, aproveitando a dependência do H2 que já adicionamos no projeto anterior.

Desta vez, não criaremos um novo projeto, utilizaremos o **mvn repository** para buscar a dependência do JPA e adicionar ao nosso projeto. Para encontrar a dependência do JPA, busque no Google por “mvn repository” ou entre diretamente pelo link <https://mvnrepository.com>.

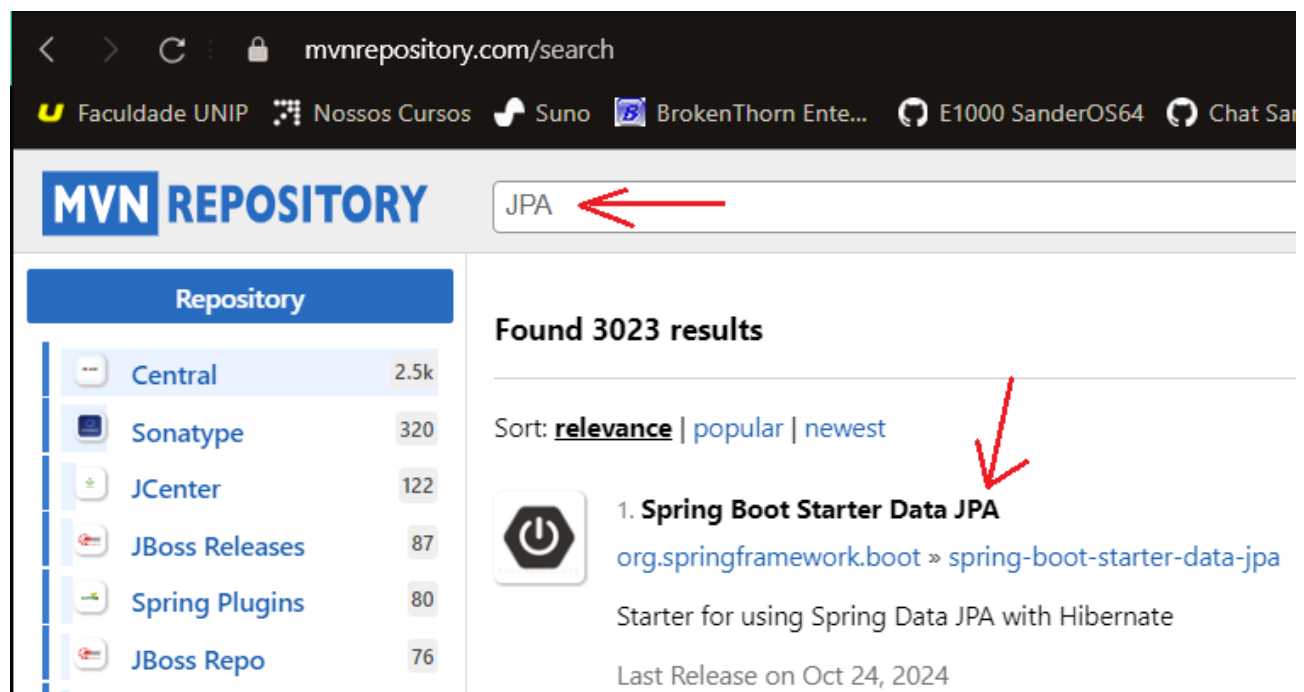


Figura 83 - Site inicial do Mvn Repository buscando por JPA

Digite no campo superior o nome “JPA” e escolha o primeiro link, que é o Starter do Spring Boot para o JPA. Na próxima página que se abrir, escolha a versão 3.3.5 e clique nela.



Spring Boot Starter Data JPA

Starter for using Spring Data JPA with Hibernate

License	Apache 2.0
Tags	persistence data spring jpa framework starter
Ranking	#192 in MvnRepository (See Top Artifacts)
Used By	2,732 artifacts

Central (231)	Spring Releases (1)	Spring Milestones (96)	Redhat GA (1)
Alfresco (2)	Evolveum (1)	Kyligence Public (2)	ICM (3)

Version	Vulnerabilities	Repository	Usages	Date
3.3.5		Central	9	Oct 24, 2024

Figura 84 - Versão 3.3.5 para adição da dependência

Após clicar na versão, será aberto uma outra página onde vamos copiar o bloco XML da dependência e adicionar em nosso projeto:

Maven
Gradle
Gradle (Short)
Gradle (Kotlin)
SBT
Ivy
Grape
Leiningen

Buildr

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <version>3.3.5</version>
</dependency>

```

☐ Include comment with link to declaration

Copied to clipboard!

Figura 85 - Copiando o bloco de dependência do JPA

Agora adicionaremos ao projeto Maven colocando dentro do bloco Dependencies o novo bloco que copiamos e após isto, CTRL + S para salvar. Assim o projeto Maven irá baixar todas as dependências relacionadas a este Starter do JPA no Spring Boot.


```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <version>3.3.5</version>
</dependency>
</dependencies>

```

Figura 86 - Adicionando a dependência do Starter JPA

Adicionada a dependência, vamos ao código! A primeira parte que vamos fazer é criar um repositório do JPA relacionada a nossa classe User. Fazemos isto criando um novo pacote chamado **repository** dentro do pacote **halloween2** e no novo pacote, criamos uma interface com o nome **UserRepository**.

```

java\org\spring\halloween2
├── model
│   └── User.java
├── repository
│   └── UserRepository.java
├── Halloween2Application.java
├── Initialize.java
└── resources

```

Figura 87 - Criando o pacote repository com a interface UserRepository

No pacote repository é onde terão todos os nossos repositórios, ou seja, a cada classe de dados que for criado, é necessário criar um repository desta classe. O repository da classe será na verdade uma *Interface* que vai herdar os métodos da interface **JpaRepository**, neste quesito, é preciso estender o JpaRepository a fim de firmar um “contrato” de métodos que serão utilizados.

Este contrato diz entrelinhas que “Todos os meus métodos contêm uma implementação do Hibernate, portanto, herde os meus métodos a fim de utilizar estas implementações, sem a necessidade de conhecê-los”. Se quisermos criar nossas próprias implementações ou extensões destes métodos, bastaria usar o **implements** ao invés do **extends**, numa classe ao invés de uma interface, do JpaRepository.

```
package org.spring.halloween2.repository;

import org.spring.halloween2.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long>{

}
```

Figura 88 - Herdando os métodos do JpaRepository em nosso repositório

Desta vez, estamos utilizando uma **interface** e não uma **class**. Para isto, é preciso escolher em **New Java File** a opção **interface**. Também adicionamos a anotação **@Repository** que vem do package **org.springframework.stereotype**. Esta anotação transforma o UserRepository automaticamente em um Bean.

Estendemos também o JpaRepository com os tipos genéricos User e Long. Esta interface contém o tipo genérico T que pode ser qualquer classe de dados, como nosso caso, o User. O S herda de T e alguns métodos de persistência irá retornar uma lista de S ou o próprio S, isto é, se usarmos um findById por exemplo, ele retornar S -> User. Mas se utilizarmos o findAll, como são vários dados, então vai retornar uma lista, que é o List<S> -> List<User>.

Já o outro tipo genérico onde vai o Long é o tipo ID, no qual será passado como tipo de argumento para alguns métodos, como: existsById que é um método que verifica a existência de um dado pelo id, esperando o argumento de identificação ID, isto é, se o tipo for INT, este método receberá INT, se for Long, como nosso caso, ele receberá Long. Contanto que, o dado de ID armazenado na tabela, seja do tipo originalmente persistido.

Será demonstrado agora quais foram as modificações feitas na classe User, que agora é mais do que uma simples classe, mas sim uma **Entidade**. Uma entidade é uma classe que é persistida no banco de dados que modela o domínio do negócio através de um modelo de dados, ou seja, já temos este modelo de dados que são os atributos de um usuário, só precisamos mapear este modelo com uma tabela relacional do banco.

Este mapeamento é feito inserindo a anotação **@Entity** e **@Table** que especifica o nome da tabela relacionada com esta entidade. Veremos o código na imagem abaixo.

```

import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;

@Component
@Data
@Scope("prototype")
@Entity
@Table(name = "user_tb")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String username;
    private String password;
}

```

Figura 89 - Entidade de Usuário mapeando com o banco

Na anotação **@Entity** especificamos que nossa classe `User` agora é uma Entidade que pode ser mapeada, a anotação **@Table** com seu nome `"user_tb"` especifica o nome dessa tabela mapeada relacionada ao `User`. Feito isto, precisamos definir os nossos atributos: A anotação **@Id** define que o atributo `"id"` do tipo `long` é uma coluna `id` da nossa tabela `User` no banco de dados, enquanto que a anotação **@GeneratedValue** especifica o tipo de geração deste `id`, isto é, a estratégia é do tipo **IDENTITY** (Identidade).

Uma estratégia do tipo `identidade` fará com que todo dado inserido nesta tabela, terá um `id` único (identificável) e auto incrementável (auto increment), adicionando uma unidade no último `id` a cada registro (começando por 1). A coluna `id` no banco também será uma **primary key** (chave primária). Se quisermos alterar o nome de alguns destes atributos na tabela do banco, apenas insira a anotação **@Column** com o `name = "novo_nome"` em cima do atributo.

No cenário atual, como não especificamos o `Column`, então as colunas geradas na tabela terão o mesmo nome que o atributo, exemplo: `id`, `username` e `password`. Esta é a mágica do ORM, que consegue mapear um objeto para o modelo relacional quase que de forma direta. Veremos esta mágica acontecendo nas próximas imagens, no qual usaremos o nosso `UserRepository` para salvar o objeto `User` no banco de dados em memória. Uma observação é que todas estas anotações vêm do pacote Jakarta.

```

@Component
public class Initialize implements CommandLineRunner{

    private User user;
    @Autowired
    UserRepository repository;

    @Override
    public void run(String... args) throws Exception {
        Random gen = new Random();
        for(int i = 0; i < 5; i++){
            user = new User();
            user.setUsername("Wend" + gen.nextLong(bound:1000));
            user.setPassword("pass" + gen.nextLong(bound:1000));
            repository.save(user);
        }

        for(User u : repository.findAll())
            System.out.println(u);
    }
}

```

Figura 90 - Registrando 5 usuários no banco H2 usando o repository

Vamos compreender o que acontece com o nosso Component Initialize. Removemos a anotação Autowired do objeto user, pois agora nós vamos instanciar-lo manualmente em um laço de repetição. Adicionamos um novo objeto que será Autowired, ele é o repository do tipo UserRepository. Portanto, através do repository poderemos chamar métodos de persistência do JPA.

No método run, temos um gerador de números aleatórios chamado **gen** que instancia a classe **Random**, o gen vai gerar um próximo número aleatório do tipo long que está entre o intervalo 0 e 1000, através do método nextLong. O número aleatório retornado será concatenado com a String de username e a String de password. A cada vez que o novo objeto é criado na memória e seus dados definidos, o repository irá salvar no banco de dados em memória.

Armazenaremos 5 registros neste banco, logo inserimos este código em um laço for com 5 repetições. Após finalizar, teremos outro laço for, porém retornando pelo método findAll do repository uma lista de dados, isto é, um **SELECT * FROM** de todos os dados registrados. Esta lista será iterada armazenando em um objeto User e apresentando na tela. Veremos na próxima imagem o resultado deste cadastro.

```
User(id=1, username=Wend706, password=pass952)
User(id=2, username=Wend786, password=pass254)
User(id=3, username=Wend637, password=pass822)
User(id=4, username=Wend923, password=pass43)
User(id=5, username=Wend282, password=pass471)
```

Figura 91 - Visualizando dados de registros do banco H2

Os dados são retornados pelo `findAll` para o objeto `User` através da iteração do laço `for`, como podemos observar na tela. Cada registro terá um dado de usuário e senha com números aleatórios na `String`. Considerando que o banco H2 simula um banco de dados real, então sabemos que este mesmo código irá funcionar para outros bancos, então desta vez vamos configurar o projeto para aceitar o PostgreSQL.

Para isto o processo é bastante simples, apenas adicionaremos a dependência do driver do PostgreSQL em nosso `pom.xml`, comentando o bloco do banco H2 e definiremos as Strings de conexão no arquivo **application.properties** da pasta **resources**.

```
<!--
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
-->
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

Figura 92 - Dependência do driver PostgreSQL

```
src > main > resources > application.properties
1  spring.application.name=Halloween2
2
3  spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
4  spring.datasource.username=postgres
5  spring.datasource.password=postgres
6  spring.jpa.hibernate.ddl-auto=update
7
```

Figura 93 - Strings de conexão no `application.properties`

O **org.postgresql** é que vai possibilitar que nossa aplicação se comunique com o banco do PostgreSQL, ao invés do banco em memória, neste quesito, o código da aplicação não muda, uma vez que o SQL gerado é o mesmo. Comentamos o bloco do H2 justamente para não termos conflitos de 2 banco de dados distintos compartilhar o mesmo código.

Já as Strings de conexão do properties facilita o gerenciamento pelas classes de auto configuração do Spring. Em **datasource**, utilizamos a **url** do banco, no qual utiliza o JDBC (Conector do banco de dados Java), o driver do PostgreSQL, o servidor local na porta 5432 e o próprio banco **postgres** que foi criado na instalação no capítulo anterior. Também temos o **username** e o **password** que é o usuário e a senha, respectivamente.

Na propriedade **spring.jpa.hibernate.ddl-auto** é do tipo **update**, isto significa que na primeira execução é criada a tabela, mas se ela existir, apenas a atualização dos dados serão feitas. Após abrir o PgAdmin, veremos na parte lateral as tabelas do antes e do depois da aplicação executar:

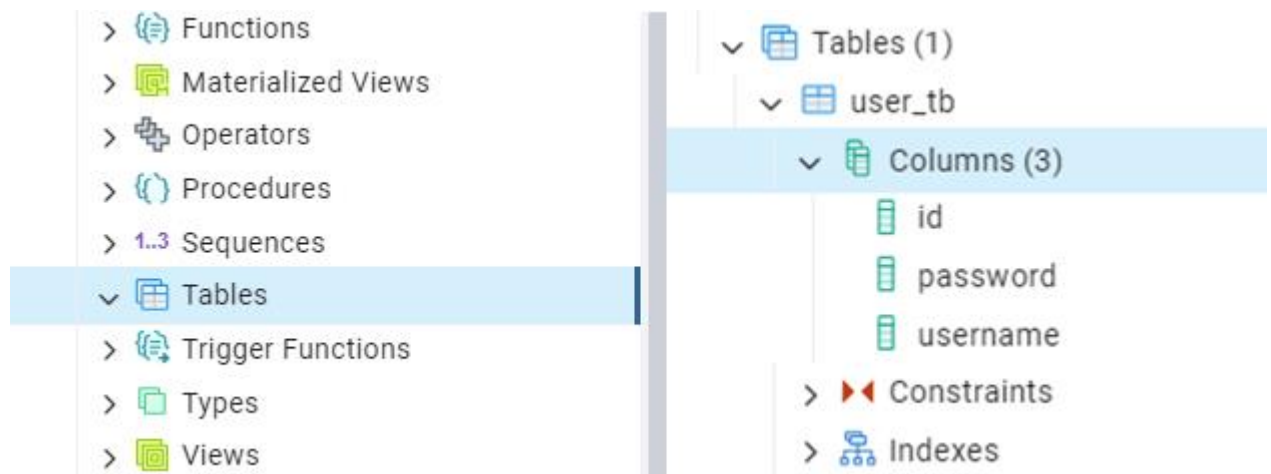


Figura 94 - Tabela antes e depois da aplicação executar

Na parte direita são as tabelas vazias, antes da aplicação executar e na parte direita, as tabelas preenchidas, após a execução, com o nome que definimos na entidade e suas respectivas colunas. Faremos uma consulta SQL clicando com o botão direito sobre **Tables (1)** e clicando na opção **Query Tools**, um novo editor será aberto, onde poderemos escrever o Script de consulta dos dados.

Query

Query History










1

SELECT * FROM user_tb

Data Output

Messages

Notifications



SQL

	id [PK] bigint	password character varying (255)	username character varying (255)
1	1	pass567	Wend964
2	2	pass785	Wend338
3	3	pass228	Wend641
4	4	pass484	Wend58
5	5	pass377	Wend679

Figura 95 - Realizando uma query dos dados de user_tb em PgAdmin

Após retornar todos os dados registrados, note que o ID é auto incrementado de 1 a 5, o mesmo valor que definimos em nosso loop de repetição FOR. A sigla **PK** refere-se ao Primary Key, sendo uma chave primária. Bigint é o tipo correlacionado ao long do Java. Vamos verificar no log do VSCode para ver se estes são os mesmos dados retornados.

```
User(id=1, username=Wend964, password=pass567)
User(id=2, username=Wend338, password=pass785)
User(id=3, username=Wend641, password=pass228)
User(id=4, username=Wend58, password=pass484)
User(id=5, username=Wend679, password=pass377)
```

Figura 96 - Comparando os dados retornados no console

Os dados exibidos no console são os mesmos que vimos no PgAdmin, isto demonstra a praticidade de criar aplicações de persistência de dados, sem nem mesmo criar implementações custosas.

A única diferença que podemos perceber é a ordem de criação das colunas, pois criamos os atributos username e password, no entanto, a criação das colunas foram feitas de forma inversa: password e username.

Também iremos ver os mesmos dados no software DBeaver. Certifique-se de criar uma nova conexão do PostgreSQL no menu **Banco de dados** e **Nova conexão**.

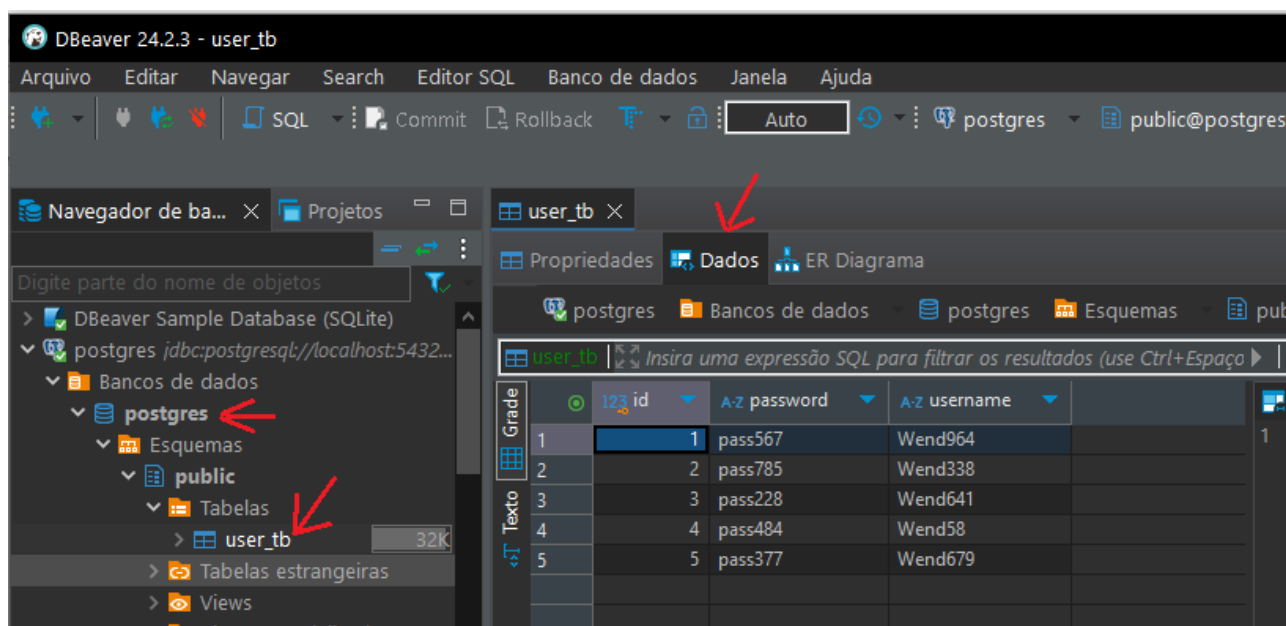


Figura 97 - Visualizando os dados de user_tb no DBeaver

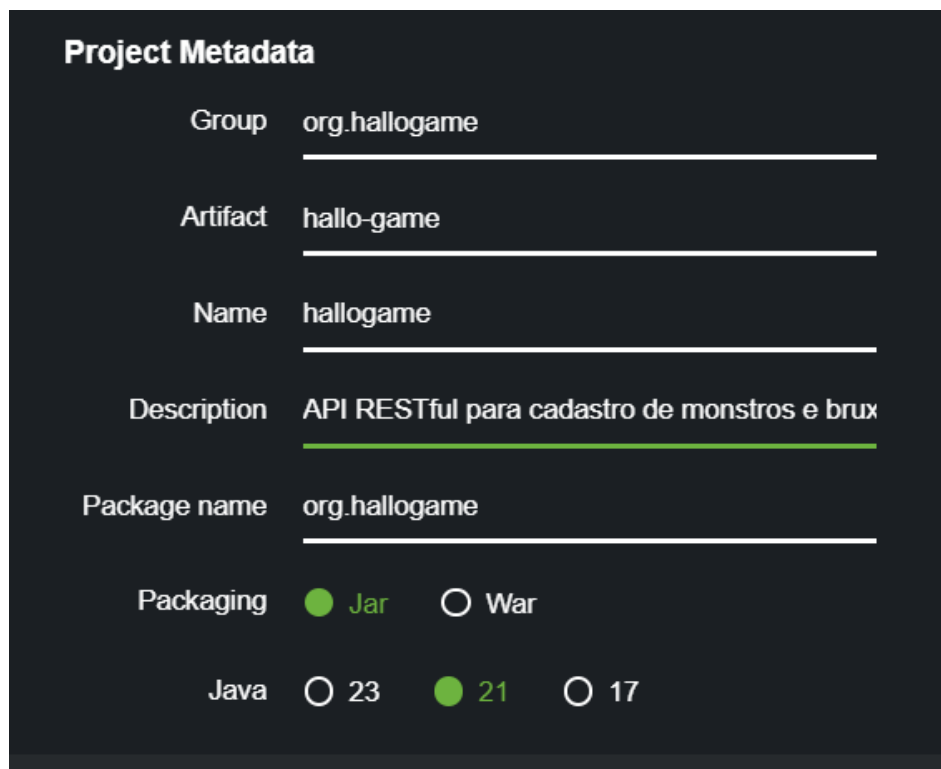
No próximo e último subcapítulo, aproveitaremos de todos os conceitos abordados até aqui adicionando novos fundamentos para a criação da nossa API.

3.5. Aplicação RESTful com Spring MVC

Uma aplicação RESTful é aquela que segue os conceitos REST (**R**epresentational **S**tate **T**ransfer), ou transferência de estado representacional. Esta é uma forma de transferência de dados (estados) com uma certa representação, como dados em JSON ou XML. É muito comum a adoção do JSON por ser um tipo de dado mais leve de se transferir na rede, entretanto, é possível enviar XML para aplicações que exigem maior declaratividade dos dados.

O REST é composto por “verbos” que podem ser: Post, Get, Put, Patch e Delete. Cada um destes verbos compõe um papel fundamental na administração de dados, muito similar a um CRUD de banco de dados (Create, Read, Update e Delete). O Post está ligado ao Create, enquanto que o Read ao Get, da mesma forma que Put para Update. Os verbos são formas de requisições web que é feita a uma API RESTful, assim como numa arquitetura cliente-servidor, onde o cliente (A Interface de Usuário – UI – Front-end) solicita dados do servidor (Aplicação funcional – Server – Back-end).

Visando estes conceitos, focaremos em importar as dependências do Spring MVC, que é composto por anotações para gerenciar estes verbos e possibilitar a criação de uma API Web de maneira prática e ágil. Veja abaixo as novas configurações Maven.



Project Metadata

Group

Artifact

Name

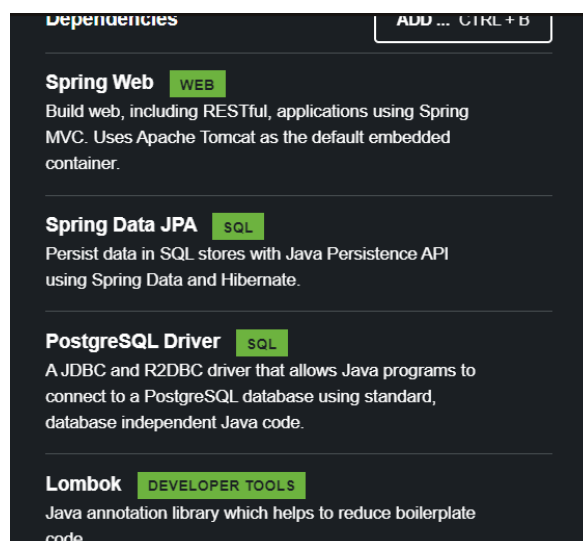
Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 23 ☒ 21 ☐ 17

Figura 98 - Nomes de pacotes definidos no metadados



Dependencies ADD ... CTRL+B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

PostgreSQL Driver SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Figura 99 - Dependências do projeto de API

Definimos outros metadados e adicionamos as dependências que já trabalhamos.

A diferença é que iremos trabalhar com uma nova dependência chamada **Spring Web**. Esta dependência faz parte do módulo Web do Spring e ele contém recursos de requisições http com os verbos que foram especificados. Além disso, a última versão do Spring Web integra juntamente com o Tomcat, que é um servidor de aplicações Web que antes era adicionado separadamente, mas agora é embedado no Spring.

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

Figura 100 - Adição de dependências para a API Web

Inicialmente, temos as mesmas dependências de antes como o **JPA**, o driver do **PostgreSQL**, o **Lombok** e o **Test** que é um módulo padrão do Spring. Para encontrar o Spring Web, basta digitar o mesmo nome na adição de dependências no Spring Initializr. Faremos uma última modificação no XML, que é determinar a versão do JPA, adicionando o bloco **<version>** como nas aulas anteriores, usando a versão **3.3.5**.

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <version>3.3.5</version>
</dependency>

```

Figura 101 - Adicionando a versão no JPA

O Spring Web também contém integrado o Spring MVC, que são pacotes que facilitam na criação de um software seguindo o padrão MVC (Model-View-Controller), onde criaríamos um pacote de modelo (Model) para organizar os modelos de dados/domínio do negócio como as nossas entidades, as visualizações (View) que são as interfaces de interação do usuário e os controladores (Controller) para inserir a lógica da camada de negócios.

No entanto, neste desenvolvimento não adicionaremos a camada View, já que vamos criar apenas o Back-end da aplicação, que são as funcionalidades da API. Também adicionaremos uma nova camada, que é a **camada de serviços**, onde é possível definir quais os serviços da nossa API, renomeando métodos que executam os métodos de persistência dos nossos repositórios.

Vamos deixar já pré-configurado o nosso application.properties, com as Strings de conexão do banco de dados PostgreSQL.

```
src > main > resources > application.properties
1  spring.application.name=hallogame
2
3  spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
4  spring.datasource.username=postgres
5  spring.datasource.password=postgres
6  spring.jpa.hibernate.ddl-auto=update
7
```

Figura 102 - Strings de conexão do postgresql

Após esta configuração, é hora de desenvolvermos as funcionalidades da nossa API. Primeiramente, vamos definir a camada de dados, depois de repositório e após isto de serviços, além de criar a classe de inicialização.

Seguiremos criando a classe de dados que é a nossa entidade. A nossa entidade terá os campos **name** que determina o nome do personagem, **skill** que é a sua habilidade, **damage** que é o dano causado, **weakness** que é a sua fraqueza e o **type** que é o tipo de habilidade. Também terá mais dois atributos: **strength** sendo o nível de força e **intelligence** que é o nível de inteligência. Deixaremos um campo opcional **description** para descrição do personagem.

Crie o pacote **model** dentro do pacote **hallogame** e crie a classe **Character** dentro do pacote model. Iremos ver a seguir:

```

@Component
@Data
@Scope("prototype")
@Entity
@Table(name = "character")
public class Character {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column(length = 30, nullable = false)
    private String name;
    @Column(length = 100, nullable = false)
    private String skill;
    @Column(nullable = false)
    private int damage;
    @Column(length = 100, nullable = false)
    private String weakness;
    @Column(length = 20, nullable = false)
    private String type;
    @Column(nullable = false)
    private int strength;
    @Column(nullable = false)
    private int intelligence;
    private String description;
}

```

Figura 103 - Modelo de dados da classe Character

Esta é uma classe do tipo entidade, através da anotação **@Entity** que possibilitará a persistência destes dados, a tabela relacionada no banco será o “character”, pode ser um componente auto injetável e seus getters/setters serão gerados automaticamente. Nós temos um id do tipo **long**, que pode armazenar uma grande quantidade de dados, será auto incrementado e uma chave primária.

Todas as colunas relacionadas terão os mesmos nomes destes atributos, já que não definimos o parâmetro **name =**, todos os campos também não deverão ser nulos (NOT NULL), através do parâmetro **nullable = false**, exceto o campo description que será opcional. Os campos name, skill, weakness e type serão do tipo **String**, no entanto, as colunas strength e intelligence serão **int** pelo fato de ser níveis pontuáveis. Pela anotação **@Column** usamos o tamanho máximo do campo através do parâmetro **length =**.

Abaixo iremos criar a interface repository do Character e a classe de serviços para acessar o repository.

```
package org.hallogame.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CharacterRepository extends JpaRepository<Character, Long>{

}
```

Figura 104 - O repository da classe Character

Criamos a interface **CharacterRepository** no pacote **repository** que também foi criado. O tipo genérico será um **Character** que é a nossa entidade e o **Long** que é o tipo de dados do ID. Veremos como foi criado a camada de serviço.

```
@Service
public class CharacterService {
    @Autowired
    private CharacterRepository repository;

    public Character salvar(Character character) {
        return repository.save(character);
    }

    public Character buscar(Long id) {
        return repository.findById(id).orElse(other:null);
    }

    public List<Character> buscarTudo() {
        return repository.findAll();
    }

    public void deletar(Long id){
        repository.deleteById(id);
    }
}
```

Figura 105 - A classe de serviço que define o CRUD do banco

A classe **CharacterService** passa a ser um serviço a partir da anotação **@Service**, isto significa que automaticamente ela se torna um Bean. Esta classe define o repository usado do **CharacterRepository** afim de utilizar os métodos do JPA, no entanto, inserimos outros nomes, como: **salvar** – que espera um objeto para ser salvo e retorno o mesmo objeto. **Buscar** – que encontra um dado pelo seu ID e retorna o objeto. **buscarTudo** – que encontra todos os dados do banco e retorna uma lista do objeto. E **deletar** – que deleta um registro da tabela pelo seu ID.

Outro fator importante é que através do objeto **repository** podemos acessar inúmeros métodos para consulta e atualização de dados no banco, sendo que cada método pode ou não retornar os dados em formato do próprio objeto. O método **findById** é necessário incluir o método **orElse(null)** que define um valor padrão de retorno para o objeto quando o dado não é encontrado. Será visto a classe Initialize e o cadastro dos dados do modelo.

```
@Component
public class Initialize implements CommandLineRunner{

    @Autowired
    CharacterService service;
    @Autowired
    Character character;

    @Override
    public void run(String... args) throws Exception {
        Character character = new Character();
        character.setName(name:"Benedita");
        character.setSkill(skill:"Fazia magia negra");
        character.setWeakness(weakness:"Água benta");
        character.setStrength(strength:30);
        character.setIntelligence(intelligence:70);
        character.setDamage(damage:60);
        character.setType(type:"magia");
        service.salvar(character);
    }
}
```

Figura 106 - Salvando os dados na classe Initialize

Nesta classe, injetamos a dependência na entidade **Character** e utilizamos os setters definidos pelo lombok para gerar os dados no banco. Então, salvamos usando o método do serviço que instancia o repository.

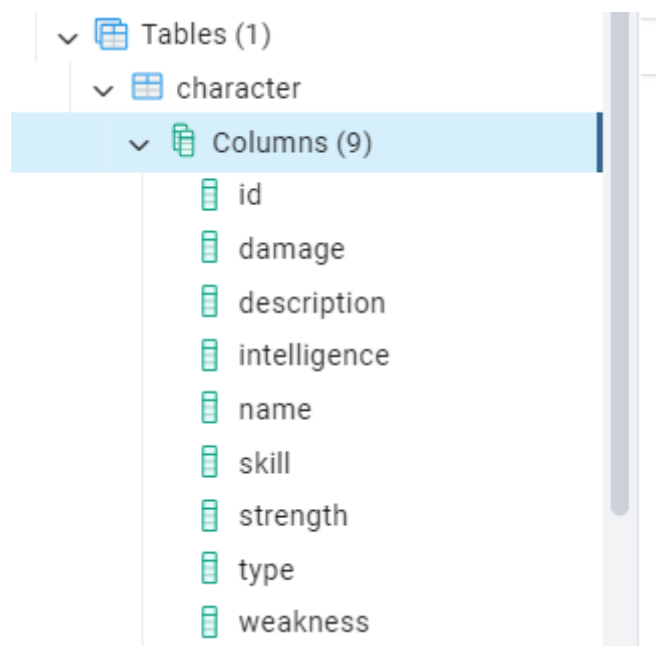


Figura 107 - Visualizando as tabelas criadas no PgAdmin

Após executar a aplicação, a tabela é criada com suas colunas, da mesma forma que definimos os dados na classe de entidade. Também é possível ver os dados com a consulta SQL.

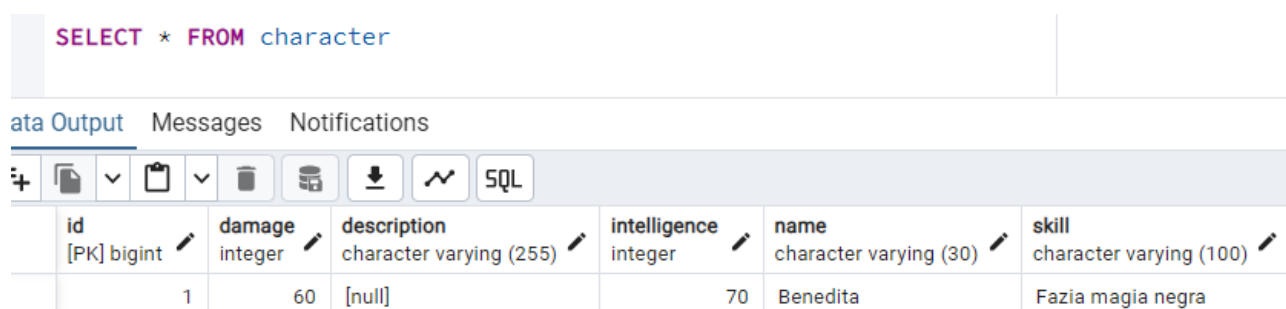


Figura 108 - Dados registrados na tabela Character

Após realizarmos os nossos testes com o cadastro de dados da nossa entidade, garantimos que tudo está funcionando da maneira correta. Desta forma, iremos explorar o Spring Web com suas anotações REST, criando nossos controllers para requisições externas do usuário e demais regras de negócio.

Primeiro, crie uma classe chamada **CharacterController** no pacote **controller** e assim iremos definir nossos primeiros Endpoints para acesso dos nossos serviços.

```
package org.hallogame.controller;

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

@RestController
@RequestMapping(value = "/character")
public class CharacterController {

}
```

Figura 109 - Criando endpoint base pelo REST

Vamos criar o nosso controlador em partes: A primeira característica é a anotação **@RestController** que determina a nossa classe **CharacterController** como uma base de endpoints, tornando esta classe acessível via URL.

A anotação **@RequestMapping** é o mapeamento de requisição, que determina qual é a URL base dos nossos endpoints, ou seja, todo endpoint configurado nesta classe, será utilizado junto com o endpoint base. Um endpoint é um ponto de acesso, uma parte da URL (caminho) que serve para disponibilizar os serviços da nossa API.

No nosso caso, a base “/character” seria fornecida junto com a URL <http://localhost:8080>, que é o link de acesso ao servidor (Um servidor local na porta 8080 – Porta padrão do servidor Tomcat do Spring Web).

```
@RestController
@RequestMapping(value = "/character")
public class CharacterController {

    @Autowired
    CharacterService service;

    public ResponseEntity<Character> saveCharacter(Character chars){
        return ResponseEntity.ok().body(chars);
    }

}
```

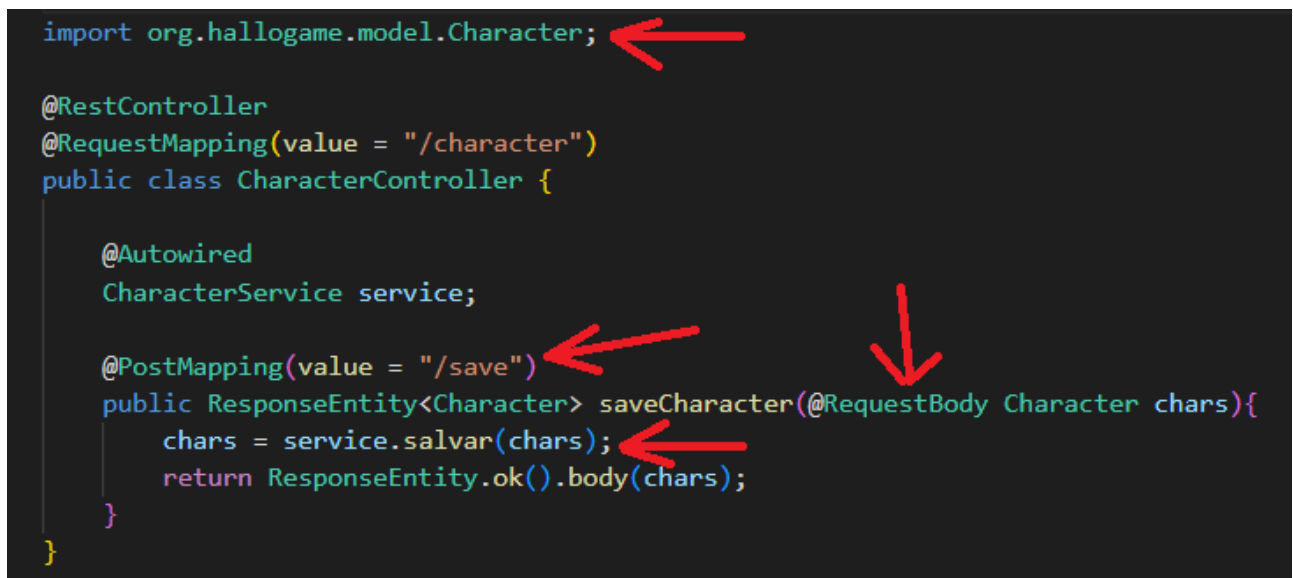
Figura 110 - Criando o método que retorna o corpo de resposta

Agora instanciamos a classe de serviço (CharacterService) usando injeção de dependência, no entanto, ainda não utilizamos nenhum dos serviços. Estamos apenas

criando um método **saveCharacter** que vai retornar a classe **ResponseEntity**. Esta classe é uma entidade de resposta em JSON com dados de cabeçalhos das requisições e corpos destas requisições.

O **ResponseEntity** espera um tipo genérico, que é um objeto que ele irá retornar para a página após a requisição. Este objeto será a nossa entidade **Character** que será retornada junto com o status de “ok” (200) e no corpo da resposta – o objeto **Character**. No entanto, podemos perceber que se quisermos salvar um novo registro, precisamos passar estes dados no corpo da requisição e isto não especificamos neste código.

Os dados que serão passados é um JSON com os mesmos atributos da entidade **Character** e estes dados não se recomenda passar via URL (que é usando o método **Get**), mas normalmente é utilizando o método **Post**, que envia estes dados de forma oculta na solicitação. Desta forma, precisamos informar ao Spring que o parâmetro **chars** do tipo **Character** será o “corpo da requisição” e precisamos informar que o método **saveCharacter** será acessado via método **Post**, acessando uma URI básica. Veremos como isto é feito:



```
import org.hallogame.model.Character;

@RestController
@RequestMapping(value = "/character")
public class CharacterController {

    @Autowired
    CharacterService service;

    @PostMapping(value = "/save")
    public ResponseEntity<Character> saveCharacter(@RequestBody Character chars){
        chars = service.salvar(chars);
        return ResponseEntity.ok().body(chars);
    }
}
```

The image shows a code editor with Java code for a REST controller. Four red arrows point to specific parts of the code: the first arrow points to the import statement `import org.hallogame.model.Character;`; the second arrow points to the `@PostMapping(value = "/save")` annotation; the third arrow points to the `@RequestBody` annotation on the `Character chars` parameter; and the fourth arrow points to the `chars` parameter in the `saveCharacter` method signature.

Figura 111 - Transforma o método em um método Post

Uma pequena observação é que na primeira seta da 1ª linha, mostra que precisamos importar explicitamente o pacote **org.hallogame.model.Character** e isto deve ser feito em todas as classes do projeto que utilizar esta entidade, pois no contrário, o java iria entender que esta classe se trataria de um dos pacotes internos com o mesmo nome da classe **Character**, resultando em erro.

Na segunda seta, a anotação **@PostMapping** transforma o nosso método `saveCharacter` em um método acessível via URL, isto significa que a partir de agora, nós temos um EndPoint. Este EndPoint será acessado pela string “/save”, junto com as outras partes da URL, como <http://localhost:8080/character/save>. A segunda seta, transforma o parâmetro deste método em um corpo de requisição através da anotação **@RequestBody**. Considerando que o parâmetro é a entrada de dados de uma função, a entrada de dados do nosso Endpoint fornecida pelo usuário, é um objeto JSON de dados a ser registrados.

A última seta, se refere à utilização do serviço **salvar**, que irá chamar o método `save` do `CharacterRepository`, estendido do JPA. Retornamos o resultado ao próprio objeto, para assim retornar este objeto ao corpo da solicitação pelo `ResponseEntity`.

Antes de prosseguirmos com a implementação da API, testaremos o registro dos dados da solicitação usando uma ferramenta específica para testes de APIs – O Postman. Entre no link <https://www.postman.com/downloads/>, e clique sobre o botão laranja **Windows 64-bit**. O arquivo **Postman-win64-setup.exe** será baixado, execute-o e será solicitado que crie um cadastro numa nova página web. Crie este cadastro e a página irá solicitar que abra o aplicativo.

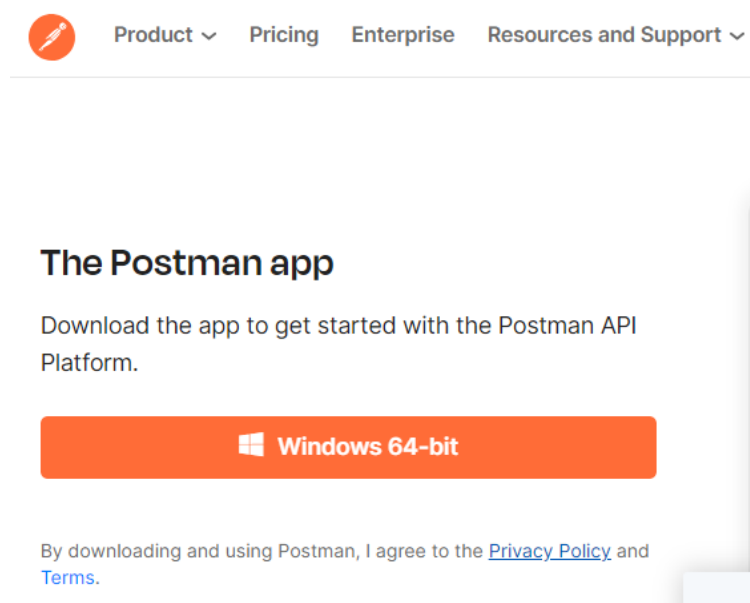


Figura 112 - Tela de download do Postman

Após abrir a interface do Postman, clique em **home** e **REST API Basic**, irá abrir a próxima tela da imagem. Na imagem abaixo terá todas os números em vermelho na mesma ordem de clique.

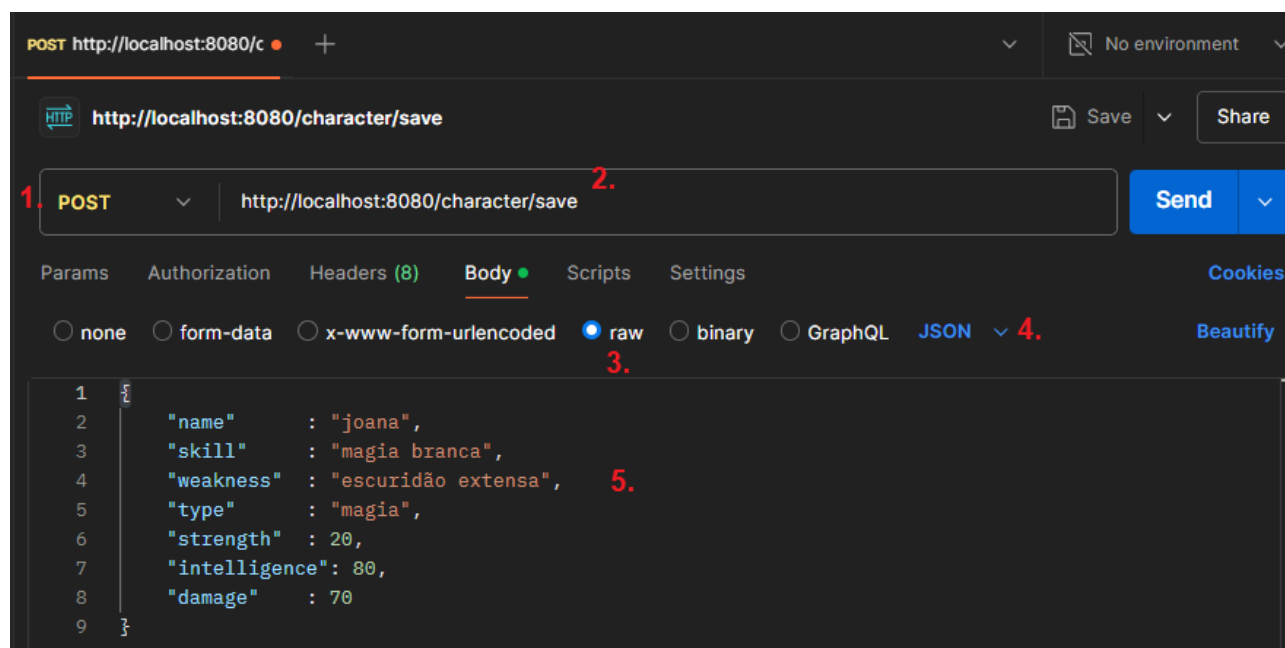


Figura 113 - Solicitando dados em JSON para endpoint

Primeiro, configure na região onde está o número 1. de GET para POST. Após isto, na região do 2. defina a URL do endpoint, que é o <http://localhost:8080/character/save>, o mesmo que definimos no controller. O botão **none** estará marcado inicialmente, altere para **raw**. Na região 4. Escolha de text para **JSON**, que é o tipo de dado que será enviado e no corpo do editor (região 5.) insira o objeto JSON de solicitação obedecendo a estrutura dos seus atributos da entidade Character. Após isto, clique no botão **SEND** e receba a seguinte resposta:

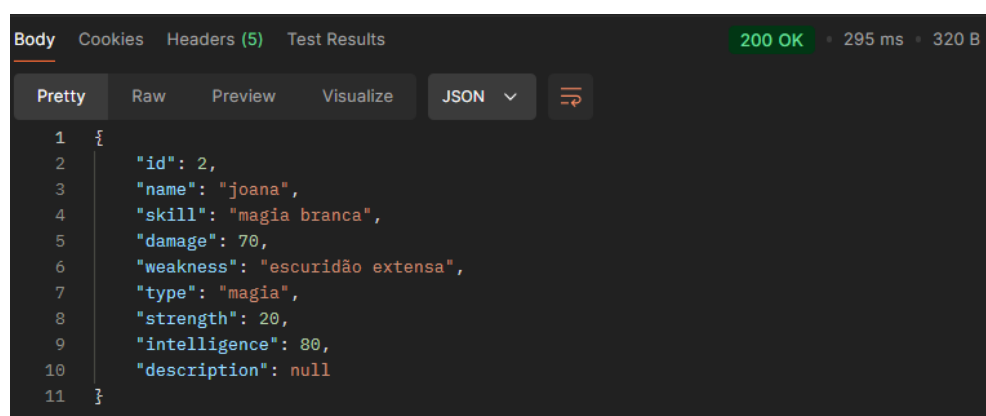


Figura 114 - Recebendo resposta em JSON da API

Note que na resposta recebemos o status **200 OK** no canto superior que é o mesmo status que definimos no método **ok()** do ResponseEntity. Em cima está marcado **body**, que é o corpo da resposta, e vemos o nosso objeto JSON, assim como colocamos

no método **body()** o objeto Character. Também poderemos ver a resposta em outros formatos e também em outros contextos como Headers ou Cookies.

O nosso dado retornado é o id = 2 pois na execução da aplicação, a nossa classe Initialize já registra um conjunto de dados fixos. Veremos os dados armazenados no PgAdmin:

1 `SELECT * FROM character`

Data Output Messages Notifications

	damage integer	description character varying (255)	intelligence integer	name character varying (30)	skill character varying (100)	strength integer	type character
1	60	[null]	70	Benedita	Fazia magia negra	30	magia
2	70	[null]	80	joana	magia branca	20	magia

Figura 115 - Dados cadastrados pela API

Podemos ver que os mesmos dados retornados no corpo da resposta, são os que foram registradores em uma nova linha da tabela. Será visto como criamos os outros métodos de endpoints para compor o CRUD da nossa API.

```
@GetMapping(value =("/{id}")
public ResponseEntity<Character> searchCharacter(@PathVariable Long id){
    Character character = service.buscar(id);
    return ResponseEntity.ok().body(character);
}

@GetMapping(value = "/search-all")
public ResponseEntity<List<Character>> searchAllCharacter(){
    List<Character> characters = service.buscarTudo();
    return ResponseEntity.ok().body(characters);
}
```

Figura 116 - Novos métodos GET para retornar dados

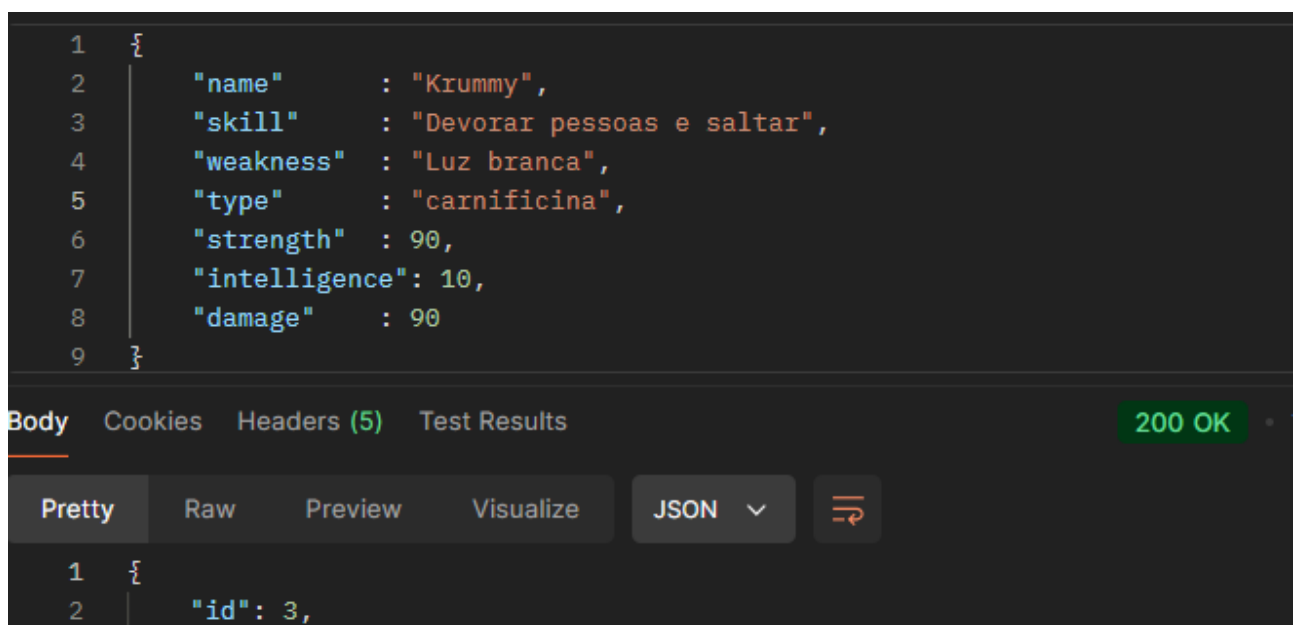
Aqui está a verdadeira praticidade de se criar uma API REST com Spring e utilizar os verbos HTTP. Nos dois métodos acima, ainda no mesmo arquivo fonte (do controlador), utilizamos a anotação **@GetMapping** especificando que os dois métodos agora serão acessados via método GET, a diferença é que o primeiro método retornar um objeto pelo ID e o outro retornará uma lista de objetos pois será “todos” os dados.

O primeiro terá o endpoint “/{id}” pois o que está entre chaves será um valor da URL passado para o parâmetro do método. O nosso parâmetro é o Long id, então

precisamos informar ao Spring que este Long id é uma **@PathVariable**, pois ela será acessada pela URL. Então retornamos o objeto **Character** pelo método **buscar()** e enviamos como corpo de resposta.

Já o segundo retornará todos os dados, desta forma, não é preciso passar nenhum parâmetro, nem no corpo da solicitação via Post (RequestBody) e nem na URL via Get (PathVariable) e como será retornado todos, é necessário de uma lista, pois é o que o método **buscarTudo()** retorna. Definimos também o tipo de objeto que estará nesta lista e o próprio Character. Portanto, enviamos de volta como resposta para a solicitação esta lista de dados.

Iremos testar nossos novos endpoints configurados, então cadastraremos os mesmos dados anteriores + novos personagens, buscaremos todos eles e depois buscaremos um em específico pelo seu id.



```
1 {
2   "name"      : "Krummy",
3   "skill"     : "Devorar pessoas e saltar",
4   "weakness"  : "Luz branca",
5   "type"      : "carnificina",
6   "strength"  : 90,
7   "intelligence": 10,
8   "damage"    : 90
9 }
```


Body Cookies Headers (5) Test Results 200 OK

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 3,
```

Figura 117 - Cadastrando um monstro

Cadastramos um 3ª personagem chamado Krummy, um monstro que devora pessoas, com força e danos altos, mas com inteligência baixa.



```

1  {
2    "name"      : "Baltazar",
3    "skill"     : "Mover objetos com a mente",
4    "weakness"  : "barulho e ruídos",
5    "type"      : "telecinesia",
6    "strength"  : 87,
7    "intelligence": 91,
8    "damage"    : 92
9  }

```

Body Cookies Headers (5) Test Results 200 OK

Pretty Raw Preview Visualize JSON ↕

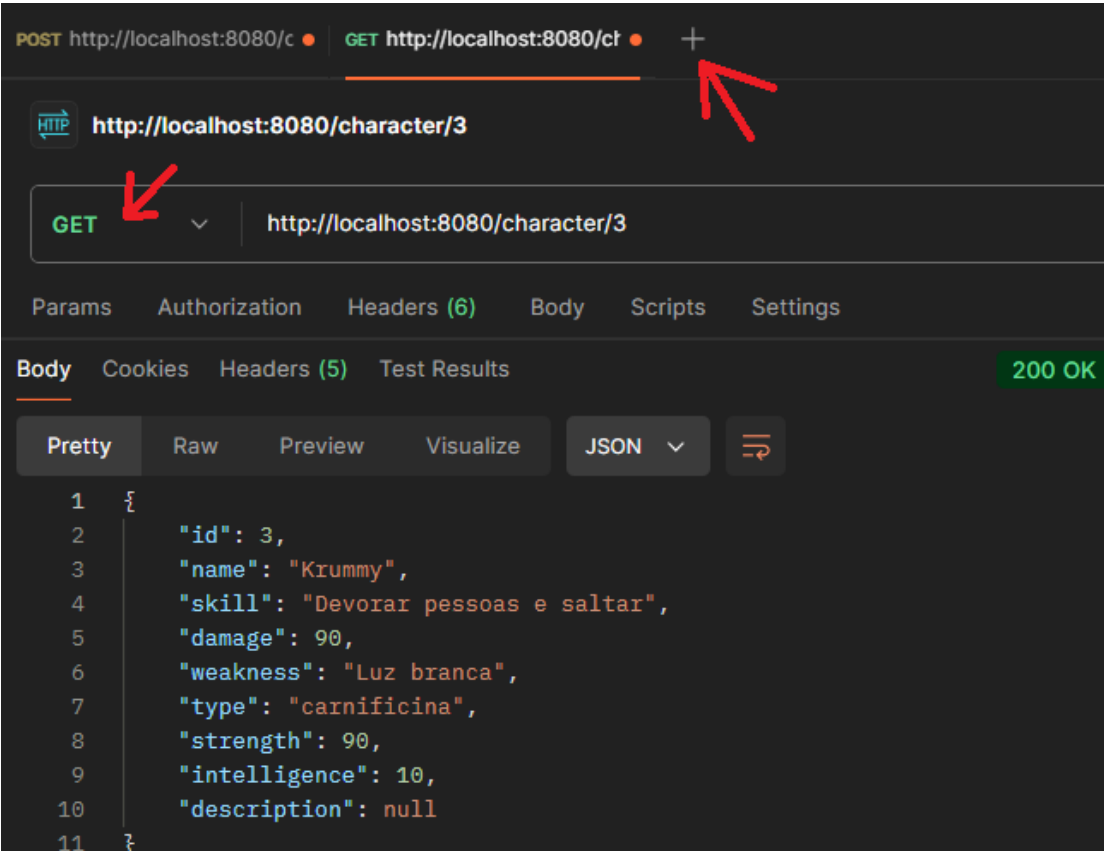
```

1  {
2    "id": 4,

```

Figura 118 - Cadastrando um personagem tele cinético

Cadastramos um 4ª personagem, chamado Baltazar, que tem um poder tele cinético, contendo uma força um pouco menor que a do monstro anterior, mas sua capacidade de danos é mais alta, assim como sua inteligência.



POST http://localhost:8080/c GET http://localhost:8080/cl +

HTTP http://localhost:8080/character/3

GET http://localhost:8080/character/3

Params Authorization Headers (6) Body Scripts Settings

Body Cookies Headers (5) Test Results 200 OK

Pretty Raw Preview Visualize JSON ↕

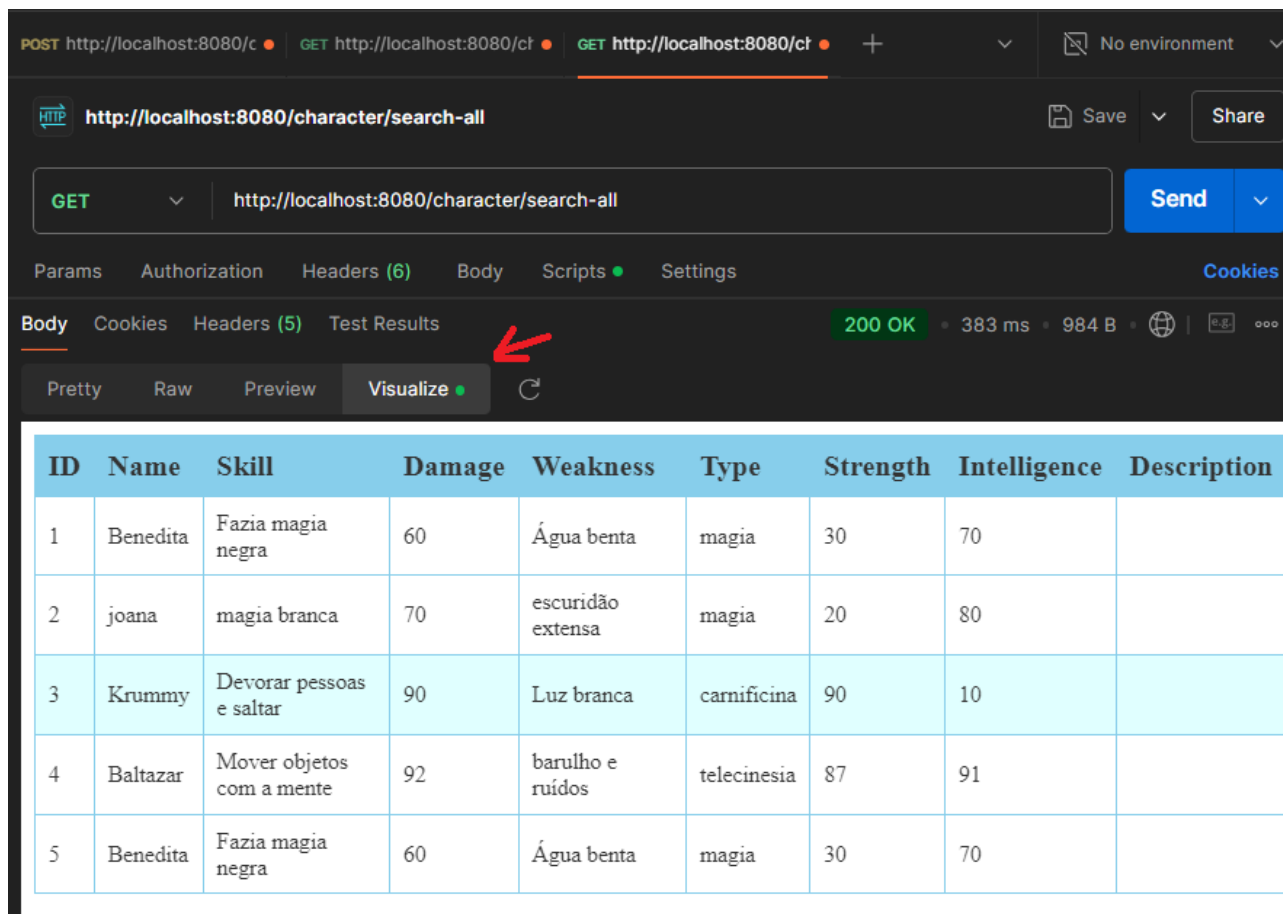
```

1  {
2    "id": 3,
3    "name": "Krummy",
4    "skill": "Devorar pessoas e saltar",
5    "damage": 90,
6    "weakness": "Luz branca",
7    "type": "carnificina",
8    "strength": 90,
9    "intelligence": 10,
10   "description": null
11 }

```

Figura 119 - Buscando o monstro Krummy pelo ID = 3

Em seguida, buscamos o nostro Krummy pelo seu ID, sendo o valor 3, passado pela URL, através do nosso endpoint “/{id}”. Foi feito usando o método GET em uma nova aba.



ID	Name	Skill	Damage	Weakness	Type	Strength	Intelligence	Description
1	Benedita	Fazia magia negra	60	Água benta	magia	30	70	
2	joana	magia branca	70	escuridão extensa	magia	20	80	
3	Krummy	Devorar pessoas e saltar	90	Luz branca	carnificina	90	10	
4	Baltazar	Mover objetos com a mente	92	barulho e ruídos	telecinesia	87	91	
5	Benedita	Fazia magia negra	60	Água benta	magia	30	70	

Figura 120 - Buscando todos os dados em formato de tabela

E por fim, retornamos todos os dados cadastrados pelo endpoint “search-all”, usando o método GET. Escolhendo o menu **Visualize** é possível abrir uma assistência de I.A que nos ajuda a visualizar os dados em vários formatos, incluindo como uma tabela.

Para finalizar este capítulo, criaremos um último método com o intuito de apagar os dados repetidos da “benedita”, pois toda vez que a aplicação se inicia, a tabela é gerada com este novo dado.

```
@DeleteMapping(value =("/{id}")
public ResponseEntity<String> deleteCharacter(@PathVariable Long id){
    Character character = service.buscar(id);
    service.deletar(id);
    return ResponseEntity.ok().body(
        "O personagem " + character.getName() + " foi excluído com sucesso!"
    );
}
```

Figura 121 - Método para deleção de personagens

Neste código usamos o verbo Delete através da anotação **@DeleteMapping**, isto permite que utilizemos um mesmo Endpoint que o de busca apenas passando o id, no entanto, alterando o método de requisição. O comportamento da requisição é alterado de acordo com o método. Buscamos o personagem antes do serviço de deleção, pois vamos retornar como uma String no corpo da resposta.

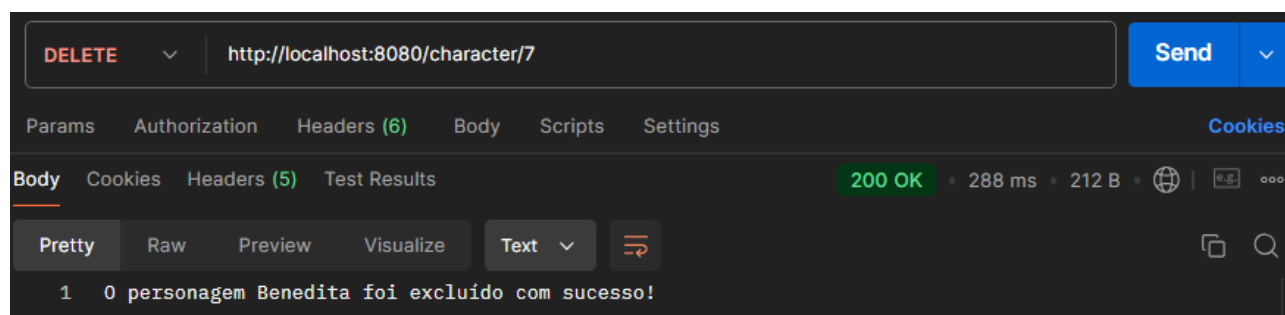


Figura 122 - Deletando o personagem com ID 7

Quando testamos o delete pelo Postman, alteramos o verbo para **delete** e passamos o ID = 7, retornando que a Benedita foi excluída com sucesso. Perceba que agora não retornamos um JSON e sim uma String. Na verdade, os Ids 5 e 6 já foram deletados antes por meio dos testes e no início da aplicação, ele gerou a nova benedita no ID = 7, então utilizamos este ID para deletar.

ID	Name	Skill	Damage	Weakness	Type	Strength	Intelligence	Description
1	Benedita	Fazia magia negra	60	Água benta	magia	30	70	
2	joana	magia branca	70	escuridão extensa	magia	20	80	
3	Krummy	Devorar pessoas e saltar	90	Luz branca	carnificina	90	10	
4	Baltazar	Mover objetos com a mente	92	barulho e ruídos	telecinesia	87	91	

Figura 123 - Visualizando os dados atualizados

Quando visualizamos, os últimos dados da Benedita já foram apagados. Note que utilizamos a mesma tabela da aba que estava ao lado do delete, ou seja, a aba do search-all, sendo assim, é possível definir várias abas, uma para cada tipo de requisição, a fim de agilizar o processo de testes. Outros recursos que podem ser adicionados, são: métodos **@PutMapping** para atualização de dados, monitorar a aplicação com **mediator** e definir a segurança de acesso via **JWT**. Deixaremos isto para um próximo momento.

CONCLUSÃO

Neste E-book apresentamos vários conceitos teóricos e práticos das aplicações em Java usando Spring Boot e Spring Web, iniciando pelos conceitos Beans, Components, Scopes e os módulos internos do Spring, que favorece a adoção dos principais padrões de projetos, como: Singleton, Prototype, Injeção de dependências e Inversão de Controle.

Também falamos um pouco sobre os “Starters” que são inicializadores de cada tipo de dependência, a fim de automatizar o processo de build, importação de bibliotecas e execução. Logo em seguida, instalamos as principais ferramentas para se começar a trabalhar com uma API que se comunica com o banco de dados, baixando o PostgreSQL, DBeaver, VSCode e a JDK/JRE.

Demonstramos a visualização de dados de nossos testes através das ferramentas de interface PgAdmin, que já vem integrado na instalação do PostgreSQL e o DBeaver podendo se comunicar com vários bancos. Instalamos a versão 1.8 da JRE e a versão 21 do JDK, mas deixando um leque de opções para o programador Java escolher suas próprias versões.

Quanto as versões, não reduzimos o número de versão nas dependências do Maven equivalente ao java (de 21 para 1.8) e do Spring Boot (de 3.3.5 para 2.5.4) como mencionamos no início, pois de alguma forma durante a criação deste E-book, foi possível executar as aplicações desenvolvidas de maneira esperada utilizando as versões mais atualizadas do Spring e das dependências.

Após as instalações do 2ª capítulo, tivemos uma boa base de prática no 3ª capítulo, explorando desde o passo a passo na criação de projetos Maven usando o Spring Initializr e suas principais características até a demonstração da DI (Dependencies Injection) e IoC (Inversion Of Control) no Spring usando CommandLineRunner, em um primeiro momento, sem utilizar dependências e em um segundo momento, introduzindo as dependências.

Nos projetos com dependências, começamos adicionando uma forma de banco de dados em memória chamado H2 para intuito de testes e depois gradualmente evoluímos para o banco de dados PostgreSQL, criando cadastros usando uma entidade. É através disso que concluímos a API REST, usando todos estes recursos + O Spring Web, seguindo o padrão de desenvolvimento Model-View-Controller e conceitos de EndPoints e verbos HTTP na prática, além de testar nossos serviços na ferramenta Postman.

REFERÊNCIAS

BROADCOM INC. Spring Data JPA – Getting Started. Disponível em: <<https://docs.spring.io/spring-data/jpa/reference/jpa/getting-started.html>>. Acesso em: 31 Nov. 2024.

ORACLE. Java Documentation. Disponível em: <<https://docs.oracle.com/en/java/>>. Acesso em: 31. Nov. 2024.

HASHTAG PROGRAMAÇÃO. Instalando o PostgreSQL e Criando o Primeiro Banco de Dados. Disponível em: <https://www.youtube.com/watch?v=L_2l8XTCPAE>. Acesso em: 31. Nov. 2024.

BROADCOM INC. Spring Framework Documentation. Disponível em: <<https://docs.spring.io/spring-framework/reference/index.html>>. Acesso em: 31. Nov. 2024.

DIGITAL INNOVATION ONE. Deal – Spring Boot e Angular (17+). Disponível em: <<https://web.dio.me/track/coding-the-future-spring-boot-angular-17>>. Acesso em: 20. Set. 2024.