CAPACITAÇÕES

JAVA PARA A PDPJ-Br

JAVA AVANÇADO

E-BOOK 2

Ronaldo Pinheiro Gonçalves Junior

Unifor | JUSTIÇA DO TRABALHO Conselho Superior da Justiça do Trabalho | Tribunal Superior Eleitoral | STJ SUPERIOR TRIBUNAL DE JUSTIÇA | JUSTIÇA 40 | PNUD | JUSTIÇA FEDERAL Conselho da Justiça Federal | CNJ CONSELHO NACIONAL DE JUSTIÇA

INTRODUÇÃO

OBJETIVO DA TRILHA

Instalar e configurar um servidor WildFly para uso de injeção de dependência com Spring Boot, introduzir o conceito de Web Services com o protocolo HTTP, abordar autenticação com o protocolo OAuth2 e Spring Security, e efetuar operações com arquivos.

CONTEÚDOS DA TRILHA

4. Servidor de aplicações WildFly e injeção de dependência;
5. Desenvolvimento de APIs REST, utilizando Spring Boot e Spring Cloud;
6. Autenticação e operações com arquivos.

INDICAÇÃO DE MATERIAL COMPLEMENTAR

INDICAÇÃO 1
Tipo: Tutorial
Título: Construindo serviços REST com Spring
Link: **https://spring.io/guides/tutorials/rest**

INDICAÇÃO 2
Tipo: Tutorial
Título: Spring Boot e OAuth2
Link: **https://spring.io/guides/tutorials/spring-boot-oauth2**

4 SERVIDOR DE APLICAÇÕES WILDFLY E INJEÇÃO DE DEPENDÊNCIA

4.1 Instalação e configuração

Na perspectiva de um usuário final, um sistema de *software* pode ser visto como um conjunto de telas que contêm recursos distintos. Essas telas, ou interfaces gráficas do usuário, atuam como uma camada de abstração, que escondem a complexidade arquitetural do *software*. Independentemente das tecnologias envolvidas, uma das arquiteturas mais populares, neste contexto, é a arquitetura cliente-servidor através da Internet, onde um usuário atua como cliente e faz requisições aos recursos disponibilizados por um servidor, que contém uma aplicação que foi desenvolvida e está em execução.

Uma das opções mais populares de servidor para aplicações Java, se não a mais popular, é o servidor de aplicação WildFly. Anteriormente conhecido como JBoss, o WildFly é gratuito e *open-source* e foi desenvolvido em Java para ser executado em múltiplas plataformas. Por ser compatível com as especificações Java EE, este servidor provê suporte aos *frameworks* e extensões que estudamos na trilha de aprendizagem anterior. Utilizaremos o WildFly para disponibilizar e executar uma aplicação Java. Para isso, precisamos primeiramente instalar nosso servidor WildFly, por meio dos seguintes passos:

1. Acesse o *site* oficial:
   • https://www.wildfly.org/downloads/
2. Selecione a opção para *download* do arquivo ZIP.
   • Nesta instalação, utilizaremos a versão “30.0.0.Final”.
3. Extraia o conteúdo do arquivo ZIP, em um diretório de trabalho.
   • Neste exemplo, o nome do diretório de instalação será o padrão, “wildfly-30.0.0.Final”. Lembre-se do local e do nome do diretório de instalação.

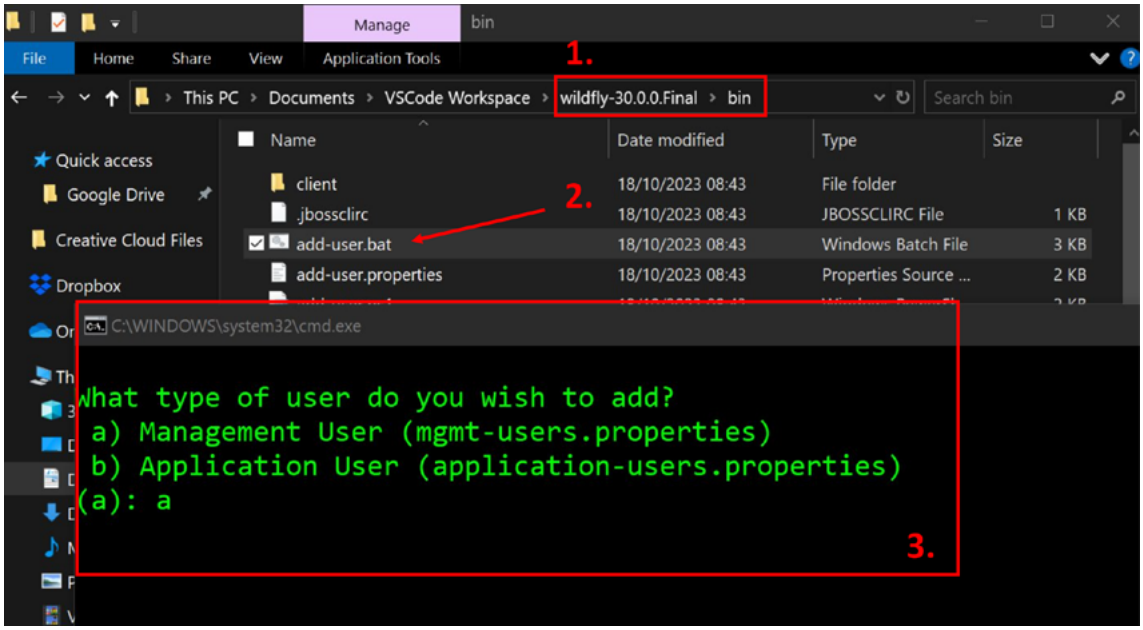
Agora que o servidor está instalado, vamos abrir o diretório de instalação para configurar o nosso servidor WildFly. Dentro da pasta de instalação, você encontrará diversas pastas e arquivos.

Para gerenciar nosso servidor, precisamos adicionar um novo usuário administrador, por meio dos passos a seguir:

1. Acesse a pasta “bin”, dentro do diretório de instalação;

2. Execute o arquivo “add-user.bat” (para sistemas Windows) ou “add-user.sh” (para sistemas Linux ou MacOS);

3. Preencha as informações na tela do terminal.

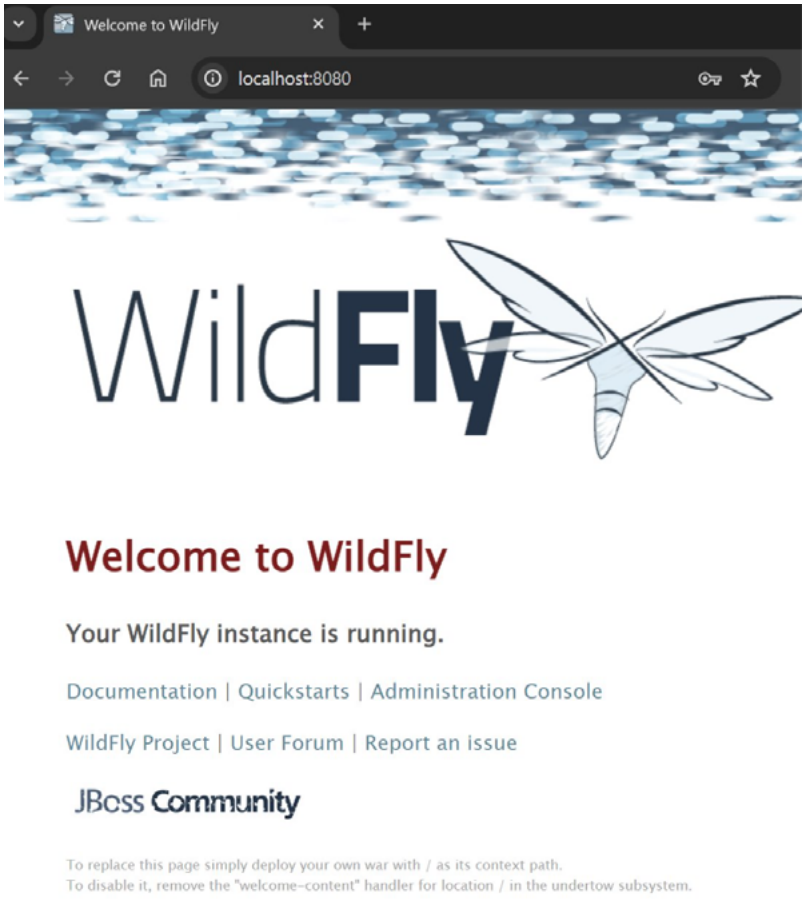


Elaborada pelo autor, 2024.

Primeiro, precisamos selecionar a opção de usuário administrador. Em seguida, precisamos definir um nome e uma senha. Por padrão, valores como “root”, “admin” ou “administrator” não podem ser usados, por questões de segurança. Além disso, a senha deve ser diferente do nome de usuário e deve conter um mínimo de oito caracteres, dos quais um deve ser uma letra, um deve ser um número e um deve ser um símbolo especial. Após definir o nome de usuário e a senha, você deverá entrar novamente com a mesma senha. Por exemplo, neste caso, os valores “ronaldo” e “seguro10!” foram usados como nome de usuário e senha, respectivamente.

Uma vez que a senha tenha sido definida, precisamos agora decidir a qual(is) grupo(s) o usuário irá pertencer. Podemos utilizar o grupo “full” neste momento, pois possui todos os privilégios de gerenciamento e administração. Todavia, dentre os demais grupos existentes, poderíamos utilizar também o grupo “read-only”, que confere ao usuário um acesso de apenas leitura, ou o grupo “read-write”, que permite um acesso para leitura e escrita.
Um último pedido de confirmação, escrevendo “yes”, encerra o processo de criação do usuário administrador. Antes de a janela do terminal fechar, você verá uma mensagem de sucesso. Com o administrador criado, podemos agora executar o servidor para iniciar o passo de configuração.

Para iniciar o seu servidor, navegue até a pasta “bin”, dentro do diretório de instalação. Você encontrará os arquivos “standalone.bat”, para sistemas Windows, e “standalone.sh”, para sistemas Linux. Basta executar este arquivo e uma janela de terminal mostrará o servidor em execução.



Elaborada pelo autor, 2024.

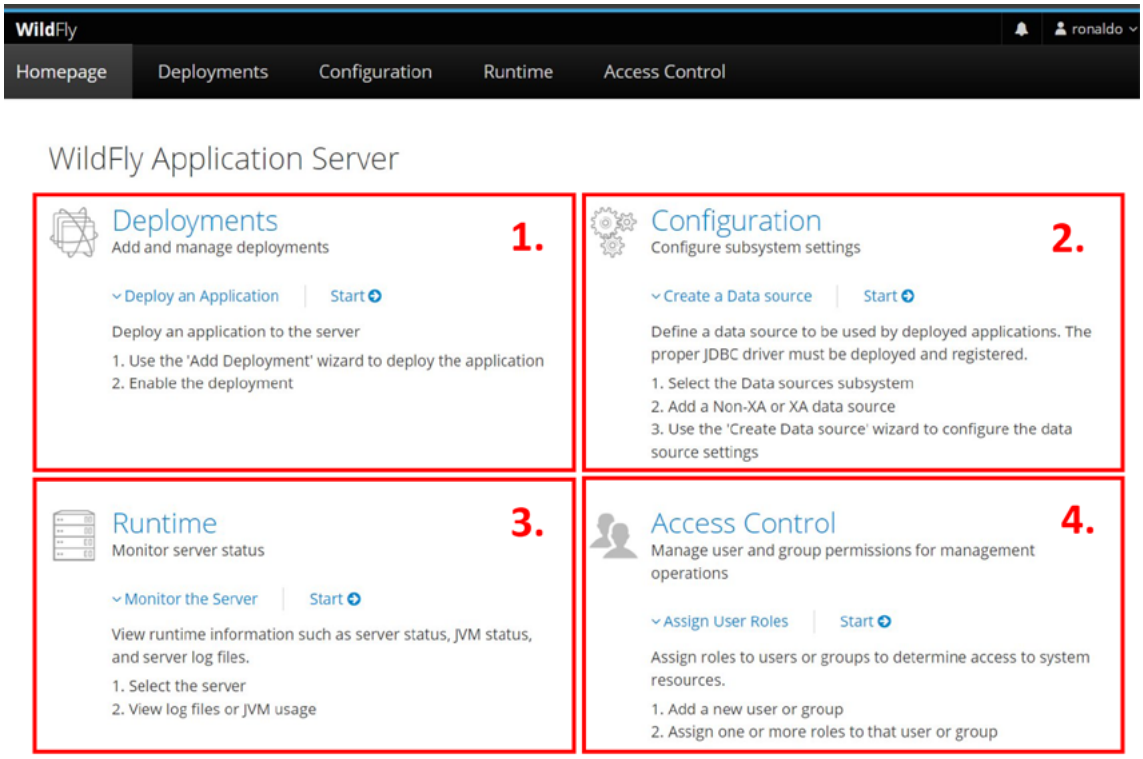
Abra seu navegador preferido (por exemplo, Google Chrome) e escreva o endereço do servidor, na barra de busca: localhost:8080. Você verá uma página de boas-vindas do WildFly, que indica que o servidor está funcionando apropriadamente. Para acessar a tela de configurações, escreva o endereço do console de administração: localhost:8080/console. Neste momento, o servidor irá requisitar suas credenciais de administrador. Coloque as informações, conforme os passos de criação de usuário que acabamos de efetuar. Ao abrir a página de administração, você verá quatro seções principais:

1. Deployments;

2. Configuration;

3. Runtime; e

4. Access Control.



Elaborada pelo autor, 2024.

Na primeira seção, podemos fazer a implantação de nossas aplicações no servidor. Após produzir o código-fonte de um projeto, podemos gerar um arquivo JAR, WAR, entre outros, como fizemos na trilha de aprendizagem anterior. Esses arquivos podem ser adicionados por meio dessa página e o projeto estará em execução no servidor. Em outras palavras, podemos tornar o projeto disponível no servidor WildFly.

Na seção de configuração, podemos adicionar ou modificar detalhes de configuração do sistema. Ao clicar em “Start”, você será redirecionado para a aba “Configuration”, onde muitos detalhes de configuração poderão ser efetuados, como, por exemplo, configuração de acesso ao banco de dados. Faremos isso na próxima trilha de aprendizagem.

A terceira seção serve para monitoramento de estatísticas de desempenho, como utilização de CPU, memória e demais recursos da máquina. Por fim, a seção Access Control visa aumentar a segurança do servidor, através do gerenciamento de usuários, grupos e papéis do servidor WildFly, sendo possível definir restrições de acesso ou modificar permissões de acesso.

4.2 Configuração de projeto para execução no WildFly

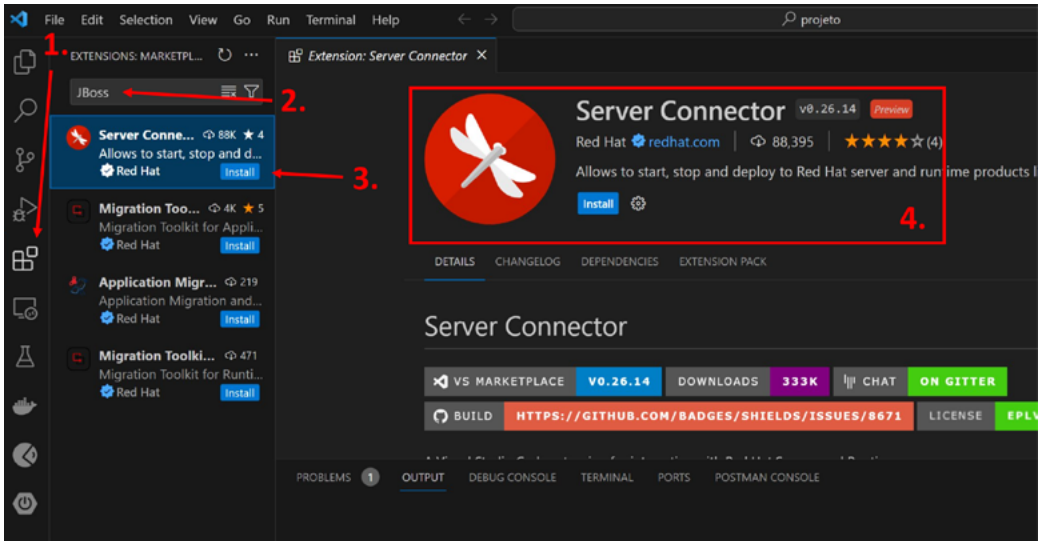
Algumas etapas devem ser seguidas, para garantir que um projeto seja implantado e executado corretamente no servidor WildFly que acabamos de instalar e configurar. Primeiro, precisamos garantir que todos os arquivos de dependência estejam configurados. Ao realizar os passos de criação de um projeto Maven, como vimos na trilha de aprendizagem anterior, isso pode ser feito por meio da definição das dependências no arquivo pom.xml.

Em seguida, precisamos garantir que o projeto seja empacotado apropriadamente, como fizemos, utilizando o comando "java- jar target/projeto". Além disso, podemos realizar configurações adicionais, para configurarmos um banco de dados ou outros recursos externos, editando diretamente os arquivos "standalone.xml", "domain.xml" ou "jboss-web. xml". Uma cuidadosa configuração garante uma implantação com sucesso. Vale ressaltar que estes passos manuais de configuração podem levar a erros, que poderão resultar em uma implantação malsucedida.

Alternativamente, podemos instalar e configurar, em nossos IDEs, um pacote de suporte para a configuração, implantação e execução de projetos no WildFly.

Veremos agora como efetuar essa configuração no VSCode:

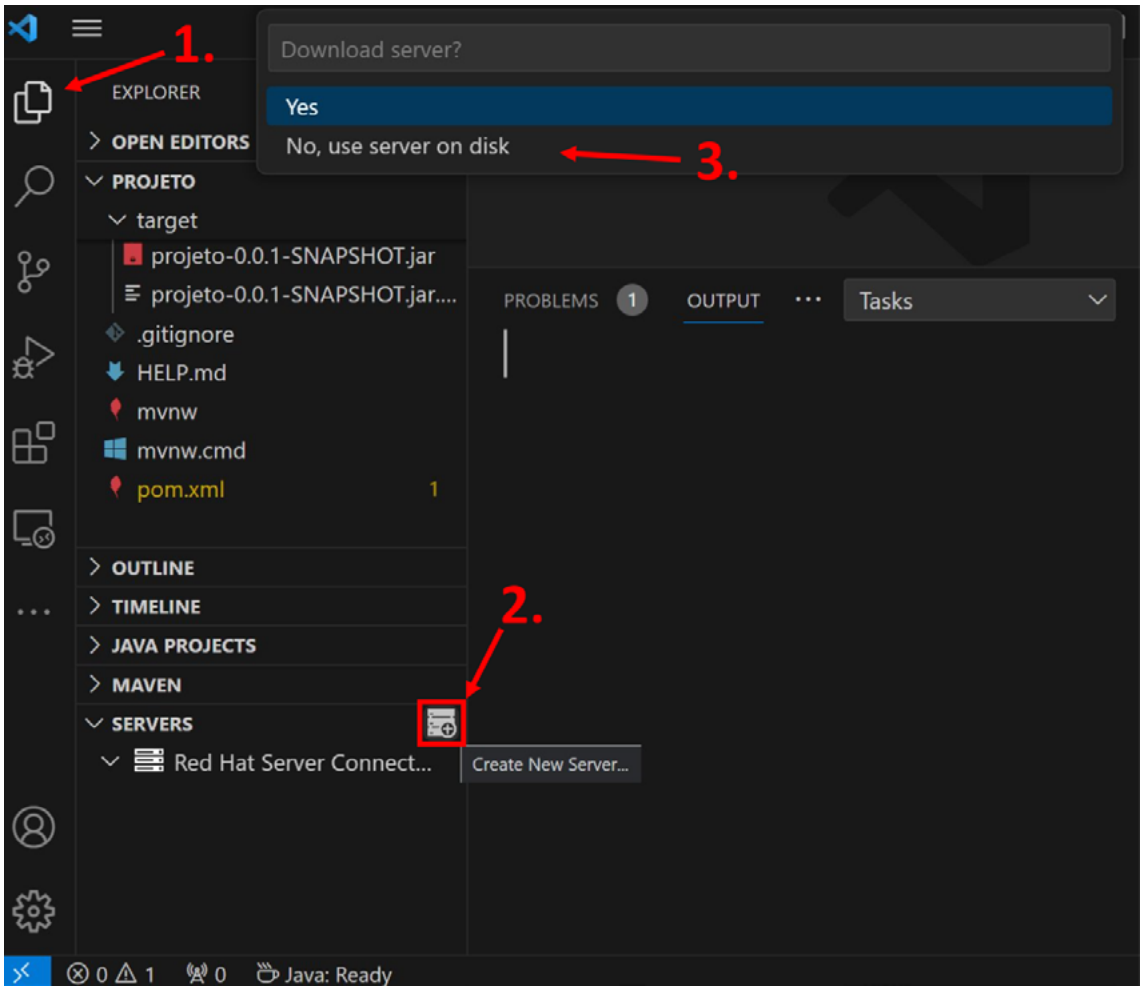
1. Acesse a aba de extensões do VSCode;
2. Pesquise por "JBoss", no campo de busca;
3. Identifique e selecione a extensão "Serve Conector", da Red Hat;
4. Confirme a extensão e selecione a opção de instalar.



Elaborada pelo autor, 2024.

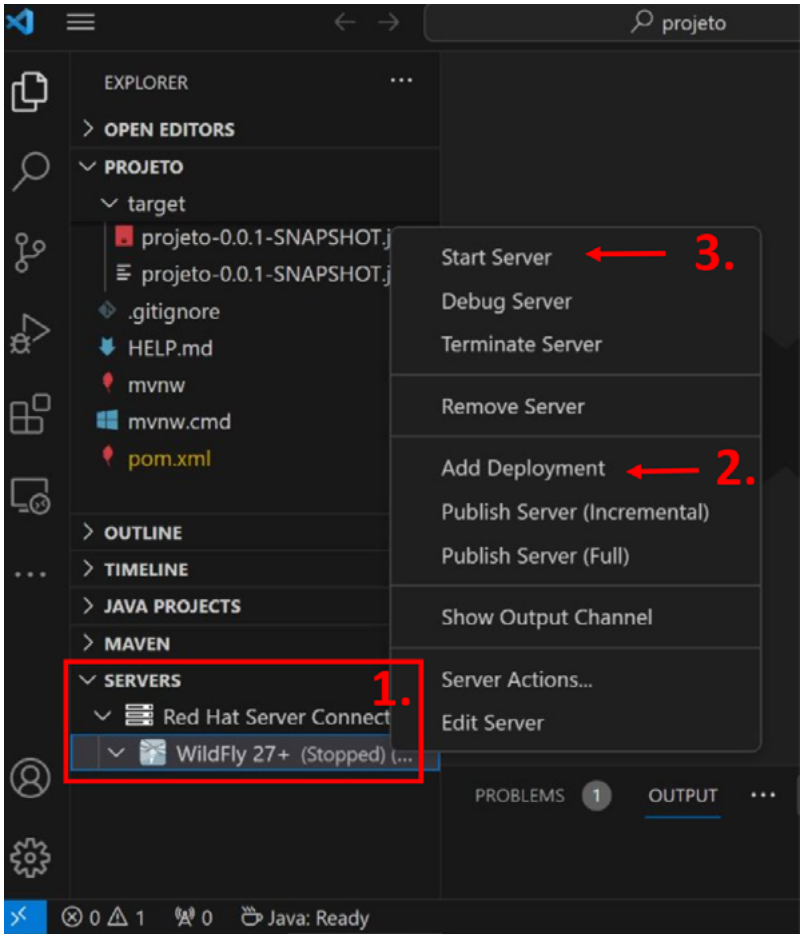
Uma vez que a extensão for instalada, realize os seguintes passos, para configurar o servidor no VSCode:

1. Retorne para a aba de explorador de arquivos;
2. Selecione a opção de criar um novo servidor;
3. Como fizemos o *download* e a instalação do WildFly anteriormente, escolha a opção de utilizar um servidor instalado em seu disco;
4. Navegue até o diretório de instalação do WildFly e confirme a escolha;
5. Finalize a criação do servidor, com as configurações padrão.



Elaborada pelo autor, 2024.

Após a configuração do WildFly, podemos implantar e executar nosso projeto com maior facilidade. Conforme mostra a imagem a seguir, note que, na área (1), encontra-se o servidor de aplicação WildFly. Com um clique do botão direito, podemos ver várias opções.



Elaborada pelo autor, 2024.

Para implantar nosso projeto no servidor WildFly, vamos utilizar agora a opção (2), “Add Deployment”. Ao selecionar essa opção, devemos escolher um arquivo para implantar no servidor. Neste caso, iremos escolher o arquivo JAR, que criamos na trilha de aprendizagem anterior, “projeto-0.0.1-SNAPSHOT.jar”. Em seguida, executaremos o servidor através da opção (3). Acesse o endereço “localhost:8080” e veja que o servidor está em execução. Em breve, iremos adicionar ao nosso projeto funcionalidades que exigirão passos adicionais de configuração, utilizando Spring Boot, Maven e WildFly, conforme aprendemos até o momento.

4.3 Conceito e aplicações da injeção de dependência

Em um projeto Java, encontramos componentes como interfaces, classes e classes abstratas. Em adição, é normal que esses componentes estejam associados entre si, em uma relação de dependência natural e necessária. Todavia, é possível que um componente esteja conectado de maneira excessiva. Nesse caso, dizemos que o componente está altamente acoplado. Um importante conceito avançado de programação, que visa aliviar o problema de alto acoplamento, é a injeção de dependência.

Para entender esse conceito, vamos analisar o seguinte exemplo:

Suponha que um sistema de um tribunal possui uma classe chamada *CasoJudicial*, responsável por calcular os gastos de um caso judicial. Para calcular o custo final, será necessário utilizar a classe *CustoJudicial*, que verifica o ano do julgamento. Se o ano for 2024 ou superior, adiciona-se um valor igual a R$ 20,00. Caso contrário, adiciona-se um valor igual a R$ 10,00. Além disso, temos também uma classe *TaxaJudicial,* que adiciona um valor igual a R$ 10,00, se for o Distrito Federal. No caso dos Estados, a taxa é de R$ 20,00. Note, abaixo, que a classe *CasoJudicial* depende dessas duas outras classes, pois realiza uma instanciação delas, nas linhas 2 e 3.

```
1  public class CasoJudicial {
2      private CustoJudicial custoJudicial = new CustoJudicial();
3      private TaxaJudicial taxaJudicial = new TaxaJudicial();
4
5      public double finalizarCusto(double custo, String estado, int anoJulgamento) {
6          custo += custoJudicial.adicionarCusto(anoJulgamento);
7          custo += taxaJudicial.adicionarTaxa(estado);
8
9          return custo;
10     }
11 }
```

Em vez disso, podemos utilizar um construtor e passar as classes concretas, ou até mesmo interfaces, para injetar essa dependência nos atributos. Note que não temos mais uma instanciação nas linhas 2 e 3, pois transferimos essa responsabilidade para a classe que faz uso da classe *CasoJudicial*.

```
1  public class CasoJudicial {
2      private CustoJudicial custoJudicial;
3      private TaxaJudicial taxaJudicial;
4
5      public CasoJudicial(CustoJudicial custo, TaxaJudicial taxa) {
6          this.custoJudicial = custo;
7          this.taxaJudicial = taxa;
8      }
9
10     public double finalizarCusto(double custo, String estado, int anoJulgamento) {
11         //...
12     }
13 }
```

4.4 Como utilizar injeção de dependência com Spring Boot

Na seção anterior, abordamos o conceito de injeção de dependência e realizamos sua aplicação em um exemplo. Um aspecto importante, que merece destaque, é que, apesar de termos realizado injeção de dependência através de um construtor, ainda é necessário existir uma classe com a responsabilidade de instanciar uma classe concreta a ser utilizada. Com Spring Boot, podemos ir além da facilidade de injeção de dependência tradicional. Como o *framework* realiza um controle do fluxo, ele também armazena os componentes necessários, chamados *Bean*. Dessa forma, não é necessário nos preocuparmos com a criação de componentes, apenas com suas definições. Podemos usar anotações do Spring Boot, para marcar quais classes são componentes.

O primeiro passo é indicar, para o Spring Boot, quais classes e interfaces deverão ser tratadas automaticamente como componentes. Podemos fazer isso escrevendo a *annotation* *@Component*, logo acima da linha de criação da classe, conforme o exemplo a seguir. Alternativamente, você pode obter os mesmos resultados e registrar os componentes, através da *annotation* *@Service*.

```
16  @Component
17  public class CasoJudicial {
18      private CustoJudicial custoJudicial;
19      private TaxaJudicial taxaJudicial;
20
21      public CasoJudicial(CustoJudicial custo, TaxaJudicial taxa) {
22          this.custoJudicial = custo;
23          this.taxaJudicial = taxa;
24      }
25      //...
26  }
27
28  @Component
29  public class TaxaJudicial {
30      public double adicionarTaxa(String estado) {
31          //...
32      }
33  }
34
35  @Component
36  public class CustoJudicial {
37      public double adicionarCusto(int ano) {
38          //...
39      }
40  }
```

O segundo passo é utilizar a *annotation* @*Autowired* nos atributos em que gostaríamos de realizar injeção de dependência. Ao marcar um atributo com essa anotação, o Spring realizará a injeção de dependência automaticamente. Dessa forma, não existe necessidade de instanciar o atributo, nem de implementar a injeção de dependência, pois será efetuada pelo Spring Boot. Utilizaremos, aqui, a interface *CommandLineRunner*, para visualizar o resultado no console do terminal.

```
1  @SpringBootApplication
2  public class ProjetoApplication implements CommandLineRunner{
3      @Autowired
4      private CasoJudicial caso;
5
6      public static void main(String[] args) {
7          SpringApplication.run(ProjetoApplication.class, args);
8      }
9
10     @Override
11     public void run(String... args) throws Exception {
12         caso.finalizarCusto(100, "MS", 2017);
13     }
14 }
```

Em breve, veremos outras *annotations* do Spring Boot, como @*Service*, @*Controller*, @*Repository* e assim por diante. Antes disso, precisamos voltar ao conceito de servidor e entender o que são serviços web.

5 DESENVOLVIMENTO DE APIs REST, UTILIZANDO SPRING BOOT E SPRING CLOUD

5.1 O que são web services e APIs?

Ao implantar um sistema de *software* em um servidor e executá-lo, como, por exemplo, no WildFly, um conjunto de serviços torna-se acessível, por meio de uma rede para usuários do *software* e para outros sistemas de *software* interessados em acessar seus recursos, sejam estes recursos dados ou funcionalidades. Quando esses serviços são disponibilizados através da Internet, mais especificamente da *World Wide Web*, chamamo-los de *web services* ou serviços web.

Uma das principais características que acompanha o uso de *web services* é a utilização do protocolo HTTP. Esse protocolo possibilita realizar uma comunicação entre diferentes sistemas através da Internet, mesmo quando esses serviços são construídos em linguagens de programação distintas. O uso do protocolo HTTP garante que *web services* sejam independentes de plataformas específicas.

Para evitar que um cliente saiba detalhes de programação de outro cliente, utilizamos APIs – *Application Programming Interface*, ou, em português, Interface de Programação de Aplicações – para expor as operações disponíveis para consumo externo. Dessa forma, um cliente sabe que tipos de serviços estão disponíveis e como consumir os recursos de seu interesse, sem que haja um alto acoplamento, entre cliente e serviço. Nas próximas seções, estudaremos protocolos e padrões para *web services*.

5.2 Web services SOAP x REST

Um importante passo no planejamento de um *web service* é a definição do estilo arquitetural a ser utilizado. Dentre as opções mais populares, temos o protocolo SOAP – *Simple Object Access Protocol*, ou Protocolo Simples de Acesso a Objetos – que auxilia desenvolvedores a expor objetos e estruturar mensagens, para comunicação em sistemas distribuídos, através de um conjunto de regras. Essas mensagens enviadas pela rede são formatadas em XML e seguem uma popular interface contratual (o WSDL, *Web Services Description Language*).

Alternativamente, temos também o estilo arquitetural REST – *Representational State Transfer*, ou Transferência de Estado Representacional. Nesse estilo, não existe um protocolo específico, pois é possível utilizar, além do XML, protocolos web como o HTTP, HTTPS, JSON etc. Em geral, consideramos o protocolo SOAP uma melhor opção para serviços web complexos ou para serviços com transações seguras, pois possui maior complexidade e oferece uma camada de segurança mais robusta. Todavia, isso implica em um desempenho inferior, quando

comparado ao REST. Além de um melhor desempenho devido a uma estrutura simples, outros motivos que tornam o REST um estilo arquitetural amplamente utilizado é a simplicidade de implementação e manutenção de operações CRUD (*Create*, *Read*, *Update*, *Delete*).

5.3 Protocolo HTTP (HyperText Transfer Protocol)

Vimos anteriormente que clientes podem gerar requisições, através da *World Wide Web*, que serão atendidas por um servidor. O protocolo de comunicação utilizado nessa comunicação é o HTTP ou *HyperText Transfer Protocol* (em português, Protocolo de Transferência Hipertexto). Dizemos que esse protocolo é *stateless*, sem estado, pois em uma cadeia de requisição gerada por um cliente, uma requisição arbitrária não depende das requisições que a antecederam. Em outras palavras, não existe um registro de estado entre requisições distintas, o que torna este protocolo apropriado, em cenários com requisições simples, ou até mesmo em sistemas distribuídos com microsserviços que necessitam de escalabilidade. As próximas seções irão descrever os principais detalhes do protocolo HTTP.

5.3.1 Request e Response

Os dois principais conceitos na comunicação via HTTP são as requisições e as respostas. Para melhor entender esses conceitos através de um exemplo, suponha que uma parte interessada está acessando uma página web de um tribunal, para consultar informações de um caso judicial. Enquanto a parte interessada utilizar a página, é possível que várias requisições sejam geradas. Cada uma dessas requisições é enviada ao servidor do tribunal, para que seja processada.

Uma vez que uma requisição é recebida, o servidor consultará o cabeçalho da requisição, local onde informações como autenticação, tipo de código e assim por diante estão contidas. Além do cabeçalho, uma requisição possui um método de solicitação, que usa um verbo HTTP: GET, POST, PUT, DELETE etc. Optativamente, a requisição pode conter um corpo, onde dados adicionais serão informados. Uma vez que uma requisição seja processada, o servidor irá construir uma resposta. Essa resposta é uma mensagem que será retornada ao cliente, contendo os dados ou funcionalidades solicitadas.

5.3.2 Verbos HTTP

Requisições que utilizam o protocolo HTTP fazem uso de um de seus verbos para indicar o tipo de operação que desejam realizar em um recurso.

Os principais verbos HTTP são:

- GET: utilizado para acessar recursos;
- POST: utilizado para criar recursos;
- PUT: utilizado para atualizar recursos;
- PATCH: utilizado para atualizar partes de um recurso;
- DELETE: utilizado para remover recursos.

Temos também outros verbos HTTP que são utilizados com menor frequência, como o HEAD e o OPTIONS. A seguir, estudaremos exemplos de aplicação para os principais verbos HTTP.

5.3.3 HTTP Status Codes

O protocolo HTTP utiliza um sistema de códigos para indicar o tipo de resultado de uma solicitação. Esses códigos são representados por três dígitos, sendo o primeiro dígito aquele que categoriza o tipo de resultado.

Seguem os principais tipos de resultados e alguns códigos de exemplo:

- 1xx: indica que a requisição foi recebida e traz informações adicionais.
  - 100: Continue
  - 102: Processing
- 2xx: indica que a requisição foi recebida e aceita com sucesso.
  - 200: OK
  - 201: Created
- 3xx: indica a necessidade de ação, por parte do cliente.
  - 301: Moved Permanently
  - 302: Found
- 4xx: indica que a requisição não pode ser atendida, devido a um erro do cliente.
  - 400: Bad Request
  - 401: Unauthorized
  - 404: Not Found
- 5xx: indica que a requisição não pode ser atendida, devido a um erro do servidor.
  - 500: Internal Server Error
  - 503: Service Unavailable
  - 504: Gateway Timeout

5.3.4 HTTP Headers

No protocolo HTTP, tanto requisições, quanto respostas possuem um cabeçalho, parte integrante dessas mensagens, que fornece informações sobre a mensagem. De uma forma geral, podemos dizer que o cabeçalho é utilizado por cliente e servidor para compreender e processar solicitações ou gerar respostas de uma forma apropriada.

O cabeçalho de uma requisição contém:

- Host: nome "host" do servidor ao qual a requisição está sendo enviada.
- User-Agent: informação da ferramenta utilizada pelo cliente para gerar a requisição, como um navegador web, sistema operacional, entre outros.
- Accept: indica o formato que o cliente espera receber, como "text/html", "application/json" etc.
- Authorization: credenciais do cliente, a fim de autenticação no servidor.
- Content-Type: descreve o formato do que foi enviado no corpo da requisição, como "application/json", "application/xml" e assim por diante.

Um cabeçalho de resposta tipicamente contém:

- Server: identificação do sistema de *software* que gerou a resposta.
- Content-Length: descreve o tamanho do corpo da resposta, em bytes.
- Content-Type: descreve o formato do que foi enviado no corpo da resposta, semelhante ao que temos no cabeçalho de uma requisição.
- Location: indica o endereço de um recurso criado.

Além das principais informações listadas acima, muitas outras informações podem ser encontradas em cabeçalhos, como: "Date", "Cookie", "Cache-Control".

5.4 Publicação de endpoints

Web services implantados em um servidor como o WildFly podem ser acessados via uma rede, como a Internet. Para tal, é necessário definir o endereço de cada serviço, para que possam ser acessados por clientes externos. Uma URL (*Uniform Resource Locator*) aponta para um recurso único na Internet. Logo, precisamos definir um conjunto de URLs, para os recursos e as operações do nosso sistema de *software*. Chamamos esse processo de publicação de endpoints, onde disponibilizamos todos os recursos e as funcionalidades de uma aplicação.

Com o Spring Boot, nossos endpoints são publicados automaticamente, durante o passo de inicialização dos serviços. Uma vez que executamos nosso projeto, o *framework* irá mapear cada controlador, definindo, através da *annotation* @*Controller*, uma URL específica de acesso. Na próxima seção, aprenderemos a definir controladores e serviços.

5.5 Controllers e Services

Anteriormente, estudamos alguns exemplos de uso de @*Component* no Spring Boot e mencionamos que a annotation @*Service* também poderia ser usada como componente, mas com um significado diferente. Um *service* possui a responsabilidade pela parte de lógica de negócios. Alguns exemplos de serviço incluem busca, atualização ou exclusão de dados de uma entidade específica. Veja o exemplo abaixo para um serviço de caso judicial:

```
@Service
public class CasoJudicialService {

    private final CasoJudicialRepository repository;

    @Autowired
    public CasoJudicialService(CaseRepository repository) {
        this.repository = repository;
    }

    public List<CasoJudicial> getTodosOsCasos() {
        return repository.findAll();
    }
}
```

Controllers possuem a responsabilidade de receber requisições e gerar respostas, servindo como uma camada de abstração que delega tarefas específicas para outros componentes, como *services*. Veja, a seguir, como seria a definição de um *controller*, utilizando REST.

```
@RestController
@RequestMapping("/api/casos")
public class CasoJudicialController {

    private final CasoJudicialService service;

    @Autowired
    public CasoJudicialController(CasoJudicialService service) {
        this.service = service;
    }
}
```

Na próxima trilha de aprendizagem, aprenderemos anotações adicionais, como: @*Entity*, para definir nossas entidades básicas; e @*Repository*, para definir um serviço de persistência de dados.

5.6 Verbos HTTP e Status Codes

Nesta seção, iremos utilizar os verbos HTTP e os códigos de status que aprendemos anteriormente, para compor nosso exemplo de controlador de casos judiciais.

5.6.1 Recuperação de dados (GET)

Vamos, agora, criar um método, dentro da nossa classe CasoJudicialController, para recuperação de todos os casos judiciais. Note abaixo que delegamos ao *service* a tarefa de recuperar todos os casos. Em seguida, adicionamos o status code 200, que significa que a requisição foi recebida e processada com sucesso.

```
@GetMapping
public ResponseEntity<List<CasoJudicial>> pegarTodosOsCasos() {
    List<CasoJudicial> casos = service.pegarTodosOsCasos();
    return ResponseEntity.ok(casos);
}
```

5.6.2 Inserção de dados (POST)

Código que recebe um novo caso judicial para ser inserido na lista de casos judiciais, como pode ser visto abaixo. Note que utilizamos o código 201 (Created), ao gerar nossa resposta.

```
@PostMapping
public ResponseEntity<CasoJudicial> criarCaso(@RequestBody CasoJudicial novoCaso) {
    CasoJudicial caso = service.criarCaso(novoCaso);
    return ResponseEntity.status(HttpStatusCode.valueOf(201)).body(caso);
}
```

5.6.3 Atualização de dados (PUT e PATCH)

Os métodos para atualizar um caso judicial por completo e para ajustar apenas algumas partes de um caso, utilizando PUT e PATCH, respectivamente, podem ser exemplificados conforme a seguir:

```
@PutMapping("/{id}")
public ResponseEntity<CasoJudicial> atualizarCaso(@PathVariable Long id,
    @RequestBody CasoJudicial casoAtualizado) {
    CasoJudicial caso = service.atualizarCaso(id, casoAtualizado);
    return ResponseEntity.ok(caso);
}

@PatchMapping("/{id}")
public ResponseEntity<CasoJudicial> ajustarCaso(@PathVariable Long id, @RequestBody
    Map<String, Object> atualizacoes) {
    CasoJudicial caso = service.atualizarCaso(id, atualizacoes);
    return ResponseEntity.ok(caso);
}
```

5.6.4 Exclusão de dados (DELETE)

Para realizar a exclusão de um caso judicial, podemos utilizar uma chamada ao método abaixo. Note que o código apropriado para mensagem de remoção é o 204 (No Content), em que nossa resposta deve ser enviada sem conteúdo.

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> deletarCaso(@PathVariable Long id) {
    service.deletarCaso(id);
    return ResponseEntity.noContent().build();
}
```

5.7 Configuração do CORS (Cross-Origin Resource Sharing)

Em geral, usuários geram requisições através de navegadores, que utilizam uma política de que recursos devem ter uma mesma origem, por motivos de segurança. Todavia, em muitos cenários, desejamos flexibilizar essa política, utilizando um mecanismo de segurança que permite que os recursos sejam acessados por mais de uma origem ou domínio. O nome desse mecanismo é CORS (*Cross-Origin Resource Sharing*) e podemos realizar sua configuração por meio de dois passos básicos.

Primeiramente, precisamos adicionar a dependência do “spring-boot-starter-web”, que já fizemos no momento de criação do projeto. Em seguida, criamos uma classe de configuração do CORS:

```
1  @Configuration
2  public class CorsConfig implements WebMvcConfigurer {
3
4      @Override
5      public void addCorsMappings(CorsRegistry registry) {
6          registry.addMapping("/**")
7                  .allowedOrigins("*")
8                  .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE")
9                  .allowedHeaders("*");
10     }
11 }
```

Observe que nas linhas 7 e 9, através do “*”, estamos permitindo todas as origens e todos os cabeçalhos, respectivamente. Dependendo do contexto, é recomendado, por questões de segurança, que essa configuração seja limitada e inclua apenas o necessário. Na linha 8, estamos permitindo os métodos que implementamos anteriormente. Alternativamente, podemos, quando necessário, realizar uma configuração mais específica e adicionar a seguinte anotação em um ou mais de nossos controladores: @CrossOrigin(origins = “”). Dentro das aspas, devemos colocar o endereço de domínio permitido.

5.8 Tratamento de exceções

O Spring possui as anotações @ExceptionHandler e @ControllerAdvice, para tratar de exceções, no momento de processamento de requisições. Da mesma forma que realizamos tratamento de exceções básico em Java, em um contexto de APIs, podemos tratar exceções relacionadas a erros, como tentar criar um recurso duplicado diretamente nos *controllers* da nossa aplicação.

```
@PostMapping
public ResponseEntity<CasoJudicial> criarCaso(@RequestBody CasoJudicial novoCaso) {
    if (service.existe(novoCaso))
        throw new CasoDuplicadoException("Caso duplicado!");

    CasoJudicial caso = service.criarCaso(novoCaso);
    return ResponseEntity.status(HttpStatusCode.valueOf(201)).body(caso);
}
```

Observe que, no exemplo apresentado, fizemos uma atualização em nosso método POST, para a criação de um novo caso judicial. Agora, temos tratamento de exceção, através de uma verificação se o caso judicial, com as informações recebidas, já existe. Se este for o caso, lançaremos uma exceção que será tratada pelo @*ControllerAdvice*:

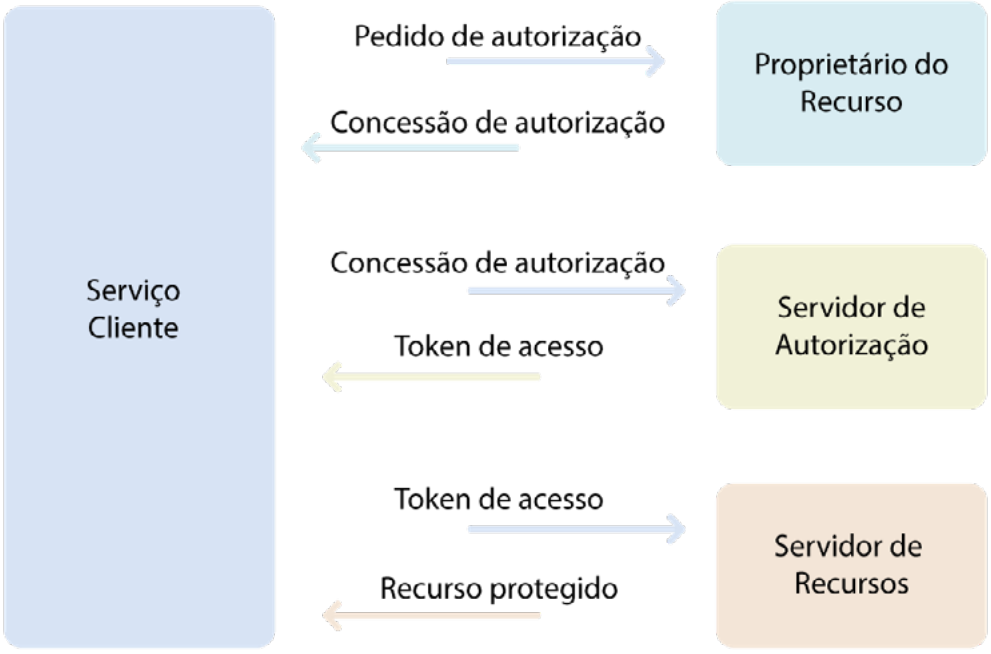
```
1  @ControllerAdvice
2  public class TratadorExcecoes {
3
4      @ExceptionHandler(CasoDuplicadoException.class)
5      public ResponseEntity<String> tratarCasoDuplicado(CasoDuplicadoException ex) {
6          return ResponseEntity.status(HttpStatus.CONFLICT).body(ex.getMessage());
7      }
8  }
```

6 AUTENTICAÇÃO E OPERAÇÕES COM ARQUIVOS

6.1 O protocolo OAUTH2

No modelo tradicional cliente-servidor, é necessário que o usuário final forneça suas credenciais, caso deseje acessar um recurso protegido. Este processo de fornecimento de credenciais também ocorre quando serviços de terceiros são utilizados. Em outras palavras, as credenciais do usuário são armazenadas por aplicações de terceiros. Além desta possível vulnerabilidade de segurança, cliente e servidor devem participar diretamente do processo de autorização de acesso do usuário final.

Através do protocolo OAuth2, um protocolo que introduz uma camada de autorização no modelo de requisições, um usuário final não precisa utilizar suas credenciais. Um serviço cliente, como uma aplicação web ou uma aplicação móvel, realiza um pedido de autorização ao usuário final, que é o proprietário do recurso a ser requisitado. A partir desse momento, o serviço cliente não precisa das credenciais do usuário e passa a utilizar a concessão de autorização conferida pelo usuário, para ganhar um *token* de acesso da nova camada de autorização, representada por um servidor de autorização. Uma vez que o serviço cliente tenha um *token* de acesso, este pode ser utilizado para requisitar o desejado recurso protegido, que está armazenado no serviço de recursos.



Fonte: Adaptado de Hardt (2012).

6.2 Autenticação com Spring Security

A fim de proteger nossa aplicação de diversas ameaças de segurança – como ataques de CSRF, injeção de SQL etc. – podemos optar por utilizar o Spring Security. Além de possuir integração com Spring Boot e Spring Cloud, o Spring Security pode ser integrado ao OAuth2. Uma forma fácil de configurar o Spring Security é escolhendo-o como dependência, no momento de criação de um projeto Spring. Alternativamente, adicione manualmente a dependência do Spring Security, no arquivo "pom.xml". No caderno de atividades, temos uma prática para adicionar e configurar o Spring Security, através da criação de classes de segurança.

6.3 Autenticação com Spring Security e Keycloak

O Keycloak é uma solução que exerce o papel de servidor de autorização e autenticação. No contexto de Spring Security com OAuth2, esta solução permite a popular autenticação única, o SSO ou *Single Sign On*. Em outras palavras, o Keycloak serve de camada de abstração no processo de autorização, evitando que o usuário final forneça suas credenciais para aplicações de terceiros.

Para realizar as configurações básicas do Keycloak, precisamos inicialmente adicionar as dependências do Keycloak. De maneira semelhante ao Spring Security, podemos configurar o Keycloak, em nosso projeto, adicionando a respectiva dependência no momento de criação do projeto Spring. Alternativamente, podemos adicionar a dependência no "pom.xml". Nosso caderno de atividades traz uma prática com os passos para a configuração manual do Keycloak.

6.4 Upload de arquivos

Uma importante funcionalidade que comumente é esperada de servidores é a capacidade de manter arquivos, como um documento PDF ou DOC. Em Java básico, temos o uso de APIs do java.io e java.nio, como as classes Files, Path etc. No contexto de *upload* de arquivos, em uma aplicação Spring Boot com REST, contamos com esses recursos básicos e outros adicionais do *framework* Spring, como a classe MultipartFile.

De modo geral, podemos definir o processo de *upload* como uma requisição que contém o documento ou arquivo a ser persistido. Ao incluir a dependência "spring-boot-starter-web", já temos a configuração básica. Resta, agora, configurar algumas restrições para o componente do Spring Multipart.

```
1  spring.servlet.multipart.max-file-size=10MB
2  spring.servlet.multipart.max-request-size=10MB
```

Para isso, podemos escrever duas linhas acima, dentro do arquivo “application.properties” que, por padrão, fica dentro do diretório “resources”. A primeira linha irá restringir o tamanho máximo do arquivo e a segunda restringe o tamanho máximo da requisição. O próximo passo é a construção de um método POST, para tratar o *upload* de um arquivo. Um exemplo desse método pode ser encontrado em nosso caderno de atividades, como proposta de prática.

6.5 Upload de múltiplos arquivos

Por meio de alguns simples ajustes ao método de *upload* de um arquivo, podemos modificar nosso serviço para receber mais de um arquivo, de uma única vez. Alternativamente, podemos possuir dois métodos distintos, um para tratar o *upload* de um único arquivo e outro para múltiplos arquivos. A principal diferença é que, para um único arquivo, o parâmetro do método será do tipo MultipartFile, enquanto, para múltiplos arquivos, teremos um parâmetro coleção de elementos do tipo MultipartFile, como uma lista.

6.6 Download de arquivos

Para prover um serviço que possibilita o *download* de arquivos, precisamos realizar a construção de um método GET, que recebe, por exemplo, um atributo identificador do arquivo por parâmetro e busca através do método “get” da classe Paths, pelo arquivo desejado. Caso a recuperação do arquivo seja bem-sucedida, podemos agora construir uma resposta e incluir o arquivo no corpo da mensagem. Um último passo é informar, no cabeçalho, que temos um arquivo anexo à mensagem, adicionando “Content-Disposition” para “attachment; filename=”””” e informando o nome do arquivo.

6.7 Tratamento de exceções

É possível que ocorram exceções ao manter arquivos em um serviço. Logo, os mesmos conceitos que utilizamos, para tratamento de exceções em nossos métodos de serviços REST, também podem ser empregados para *upload* e *download* de arquivos.

O caderno de atividades conta com práticas de *upload* de arquivos, *upload* de múltiplos arquivos e *download* de arquivos, acompanhado de código exemplo, para tratamento de exceções nesse contexto. Dessa forma, a realização das práticas do caderno de atividades é uma tarefa essencial para a absorção de todo o conteúdo apresentado nesta trilha de aprendizagem. Na próxima trilha, veremos Sistemas de Gerência Banco de Dados (SGBD), acesso e versionamento de banco de dados e arquitetura microsserviços.

REFERÊNCIAS

CARNELL, J. **Spring Microservices in Action**. 1ª edição. Editora Manning, 2017.

FUGARO, L. **WildFly Cookbook**. 1ª edição. Editora Packt Pub Ltd, 2015.

GUTIERREZ, F. **Pro Spring Boot 2**: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices. 2ª edição. Editora Apress, 2018.

HARDT, D. **The OAuth 2.0 Authorization Framework**. Internet Engineering Task Force (IETF), Microsoft, 2012. Disponível em: https://tools.ietf.org/html/rfc6749. Acesso em: 28 fev. 2024.

SPILCA, L. **Spring Security in Action**. 1ª edição. Editora Manning, 2020.

TURNQUIST, G.; et al. **Learning Spring Boot 3.0**: Simplify the development of production-grade applications using Java and Spring. 3ª edição. Editora Packt Publishing, 2022.