Fundamentos Java e Spring Boot

SENAI <LAB365>

SUMÁRIO

ORGANIZAÇÃO DO E-BOOK

Qual o objetivo deste e-book?

Ser um conteúdo introdutório sobre **Java/Spring Boot Web** abordando os fundamentos de:

- Java Web;
- Spring Boot;
- Padrão de projeto MVC com o uso de DTOs;
- Segurança de aplicações com o uso de JWT;
- Cache;
- Profiles;
- Monitoramento.

Para quem é recomendado este e-book?

Este e-book é recomendado para iniciantes no mundo **Java/Spring** que desejam ter uma visão geral do funcionamento do framework com suas principais bibliotecas.

Pré-requisitos

- Fundamentos de Java;
- Lógica de programação;
- Fundamentos de programação Web.

JAVA PARA WEB

Dos primórdios ao Servlet

Quando o **Java** foi lançado oficialmente no ano de **1995,** como consta no site oficial Java.com, talvez um dos seus fundadores, James Gosling não imaginasse o caminho que o **Java** iria percorrer na Web e os diversos desafios que estariam a surgir.

Saiba mais sobre o Java e James Gosling, um dos fundadores do Java através dos links a seguir ou dos códigos QR:

https://www.zup.com.br/blog/java

Saiba mais sobre o Java e por que precisamos dela através do link a seguir ou do código QR:

https://www.java.com/pt-BR/download/help/whatis_java.html

Saiba mais sobre James Gosling, criador do Java, através do link a seguir ou do código QR:

https://itforum.com.br/noticias/james-gosling-criador-do-java-fala-sobre-carreira-low-code-e-futuro-da-linguagem/

Sem dúvida, o mercado da Web hoje é um grande polo de oportunidades e, com certeza, o **Java** está presente nele.
Não podemos iniciar este e-book sem mencionar como o **Java** resolve alguns desafios e como a sua arquitetura está presente e organizada nesse ecossistema.
Nesse contexto, o **Java** nos apresenta a sua série de especificações, conhecida por muitos como **Java EE** ou *Java Enterprise Edition* – especificações bem detalhadas de como devem ser implementados alguns requisitos, que podemos chamar aqui de **requisitos não funcionais**.

O que seriam esses requisitos não funcionais?

Uma aplicação **Web** já tem de lidar geralmente com uma série de desafios que variam conforme o seu domínio de negócio (as chamadas regras de negócio). Porém, no enfrentamento a esses desafios, há questões comuns a todas e quaisquer aplicações **Web**, tais como persistência em banco de dados, *web services*, conexões HTTP, gerenciamento de sessão, transações, entre outros. O **Java EE** disponibiliza uma série de especificações que descrevem, em detalhes, como esses problemas comuns devem ser resolvidos.
A comunidade que gerencia essas especificações é conhecida por **JCP,** ou Java Community Process.

Você pode acessar o site da Java Community Process através do link ou do código QR a seguir:

https://www.jcp.org/en/home/index

Veremos, de forma introdutória, a especificação do **Java EE** que dá suporte ao **Java** no contexto **Web**.
Veremos, aqui nesta sessão, de forma introdutória, um pouco de **Java** para a Web nas especificações **Java Servlets**.

Servlets

O termo *Servlet* refere-se a um pequeno servidor ou, se preferir, "servidorzinho", em tradução livre a partir do inglês. A primeira especificação publicada para esse fim no **Java EE** foi no ano de **1997**.
As aplicações feitas sob essa especificação necessitam "rodar" dentro do que chamamos de servidores de aplicação **Web Container,** responsáveis por conter as implementações de infraestrutura dos **requisitos não funcionais** que mencionamos anteriormente.
As especificações do **Java EE** são muitas, e grande parte das aplicações **Web** não necessita de todas essas implementações – daí, então, veio a necessidade da criação desse **Web Container**. Um dos mais famosos hoje no mercado é o Apache Tomcat, que implementa**,** até a criação deste e-book, somente as especificações de **Servlet** e **JSP**.

Você pode acessar o site da Apache Tomcat através do link ou do código QR a seguir:

https://tomcat.apache.org/

Uma aplicação feita sobre essa especificação segue basicamente o fluxo descrito a seguir:

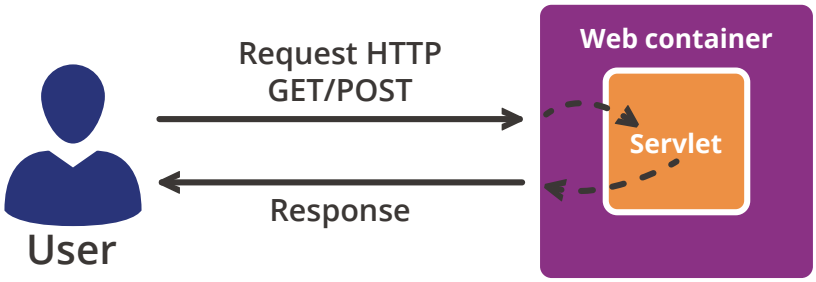


Figura 1 - Fluxo de uma requisição Servlet

Fonte: Do autor (2022)

Agora, entenderemos em mais detalhes esse fluxo:

1. Quando o usuário realiza a requisição (*request*), que pode ser através do seu navegador web, ela cai diretamente no **Web Container**;
2. O **Web Container** descobre qual **Servlet** encaminhar através do link elaborado pela requisição; agora o **Web Container** disponibiliza para o **Servlet** dois objetos – **HttpServletRequest** e **HttpServletResponse**;
3. Toda implementação de infraestrutura é disponibilizada pelo **Web Container**, como no caso de alocação de **threads** caso essa requisição necessite;
4. Já no **Servlet**, é realizado todo o processamento da requisição, e, através do objeto **HttpServletResponse**, é encaminhada a resposta ao usuário que realizou a solicitação.

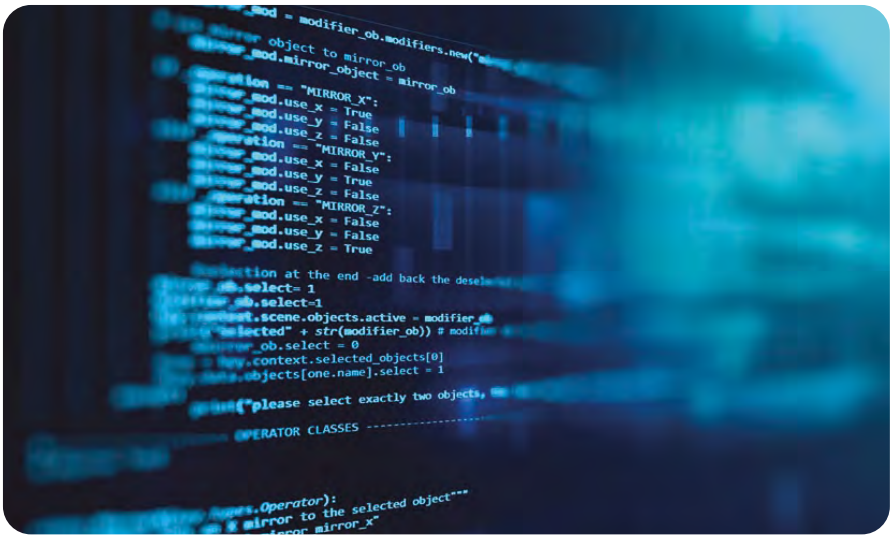
Vejamos um exemplo de um **Servlet** sendo implementado em uma classe **Java**. O exemplo a seguir está implementado na versão 3.0 dessa especificação.

```
@WebServlet("/helloWorld")
public class HelloWorldServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
   public HelloWorldServlet()
   {
       super();
   }
   protected void service(HttpServletRequest request,
                          HttpServletResponse response)
           throws ServletException, IOException
   {
       PrintWriter out = response.getWriter();
       out.print("<HTML><H1>Hello World!</H1></HTML>");
       out.close();
   }
}
```

A operação *service*() é executada sempre quando há o redirecionamento do **Web Container** para o **Servlet** através do contexto (link) informado na anotação **@WebServlet**. Como visto no exemplo, ele recebe os dois objetos, **HttpServletRequest** e **HttpServletResponse**, como parâmetro.
Neste nosso exemplo, estamos utilizando o objeto **HttpServletResponse** para escrever a frase "**Hello World!**" com o uso de html simples. Essa resposta será interpretada pelo navegador web e essa frase será impressa na tela de navegação do usuário que realizou a requisição (*request*).

Esse tipo de aplicação, em que todo o processamento é realizado no servidor de aplicação, é também conhecido como **Server Side Rendering** ou **SSR**.

Mas, escrever código html em código **Java** não é assim tão bom de se fazer, considerando os padrões modernos de hoje em dia em que a maioria das páginas web tem diversos arquivos de **JavaScript** e **CSS** para seu correto funcionamento, o que tornaria muito difícil a manutenção dessas páginas.



SPRING

(Introdução) Spring Boot, do problema à solução

Como constatamos na sessão anterior, o **Java** tem uma forma peculiar de organização com as suas especificações **Java EE,** o que, para alguns, demonstra a sua solidez no mercado e, para outros, pode parecer muito burocrático e ocasionar uma demora em termos de evolução da linguagem.
Tendo em vista esse pensamento, a **Pivotal,** que foi recentemente adquirida pela VMware, criou um ecossistema em **2003** chamado de **Spring**, com o intuito de acelerar a evolução da linguagem, sem se prender a especificações, proporcionando uma gama maior de funcionalidades disponibilizadas em suas próprias bibliotecas.

Você pode acessar o site da VMware através do link ou do código QR a seguir:

https://tanzu.vmware.com/

A aceitação do mercado foi ótima, tendo entre as empresas apoiadoras nomes como **Netflix** e **Amazon**. Porém, com o número alto de bibliotecas disponibilizadas, os projetos **Web** tiveram configurações mais complexas, em comparação a tecnologias que estavam surgindo e ganhando força na época, como o **Node.js**.

Com esse problema em mente, os engenheiros da **Pivotal** criaram o **Spring Boot**, visando acelerar o processo de desenvolvimento, trazendo uma série de configurações já importadas em sua estrutura, fazendo o processo de desenvolvimento se tornar mais *clean* para os desenvolvedores.
Até o seu lançamento, todas e quaisquer aplicações **Java Web** precisavam necessariamente rodar dentro de um servidor de **aplicação Java** ou um **Web Container**, como já foi visto aqui neste e-book, na sessão **Servlet**. Com o seu lançamento, o **Spring Boot** trouxe um conceito inovador, com a própria aplicação sendo seu servidor de **Web Container,** bastando que ela estivesse rodando sobre a estrutura de seu framework. O **Spring Boot** incorporou parte do código-fonte dos principais **Web Containers** do mercado, como o Apache Tomcat, em sua estrutura de configuração, proporcionando um *start* de projeto **Java** muito mais ágil, como até aquele momento nunca se havia constatado na comunidade.

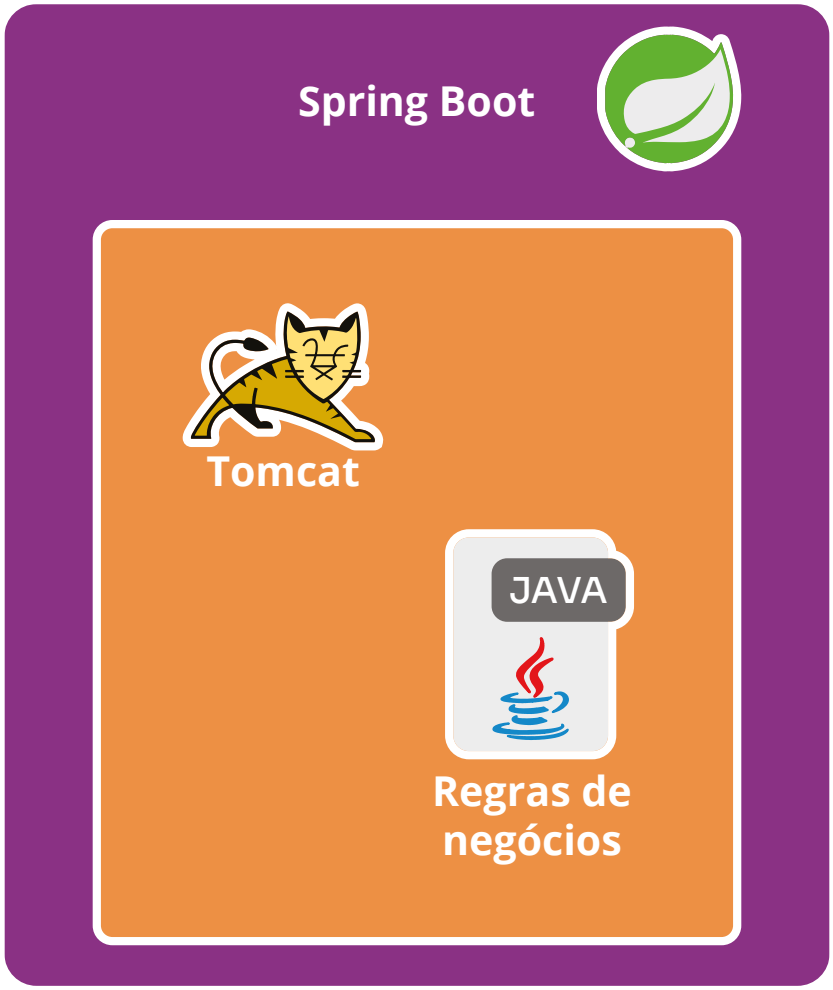


Figura 2 - Arquitetura Spring Boot

Fonte: Do autor (2022)

Para mais detalhes sobre a arquitetura desse framework, acesse a sua documentação oficial acesse o link ou o código QR a seguir:

https://spring.io/projects/spring-boot

A partir desta sessão, abordaremos os principais conceitos **Web** sobre essa estrutura de projeto, por se tratar uma estrutura bem aceita e considerada estável no mercado atualmente.

PADRÃO DE PROJETO MVC

Segundo o blog de Martin Fowler, padrões de projetos são soluções ou, se preferir, “sugestões” de como devem ser resolvidos determinados problemas que ocorrem em determinadas frequências em projetos de software. Essas “sugestões” foram catalogadas por uma série de pessoas que passaram pelos mesmos problemas, as quais, pensando no critério de manutenibilidade e extensibilidade, chegaram a tais sugestões.
O padrão MVC é um desses padrões, categorizado como um padrão arquitetural.

Para mais detalhes sobre o padrão MVC, acesse o link ou o código QR a seguir:

https://martinfowler.com/eaaDev/uiArchs.html#ModelViewController

Nele, temos a seguinte estrutura:

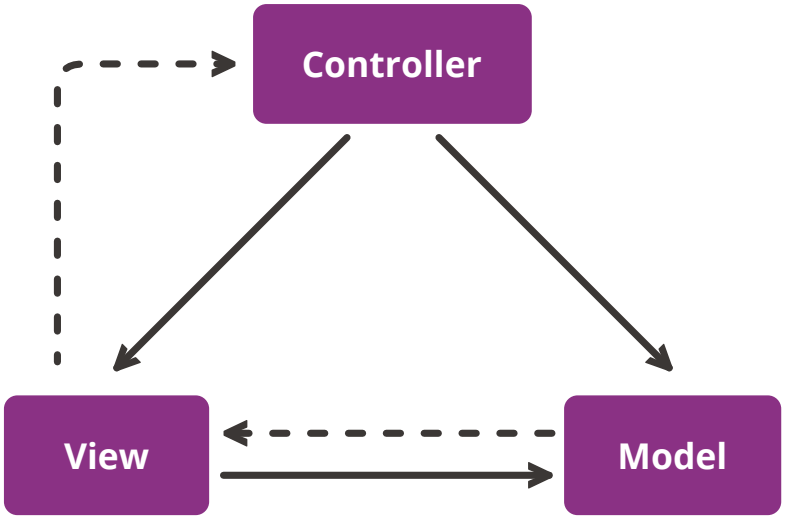


Figura 3 - Fluxo de comunicação MVC

Fonte: Do autor (2022)

Na figura, as linhas sólidas indicam associação direta e as tracejadas indicam associação indireta.

Model: é a camada onde são armazenadas todas as lógicas de negócio de uma aplicação. É através dela que são acessados, excluídos e persistidos os dados em um banco de dados.

View: é a camada de saída dos dados processados pela camada Model. Essas saídas podem ser páginas estáticas, dados formatados (json, txt etc.).

Controller: é a camada de ligação entre a View e a Model, responsável pelo armazenamento das ações disponibilizadas para um usuário ou cliente de uma aplicação.

No framework **Spring**, temos a sessão exclusiva que trata dessa camada, chamada de **Spring Web MVC**.

Spring Web MVC

O **Spring Web MVC** é construído sobre a especificação de **Servlet** do **Java EE** e, quando utilizado sobre o **Spring Boot**, não tem a necessidade das implementações de classes **DAO** *(Data Acces Object)*, como pode ser visto no fluxo a seguir:

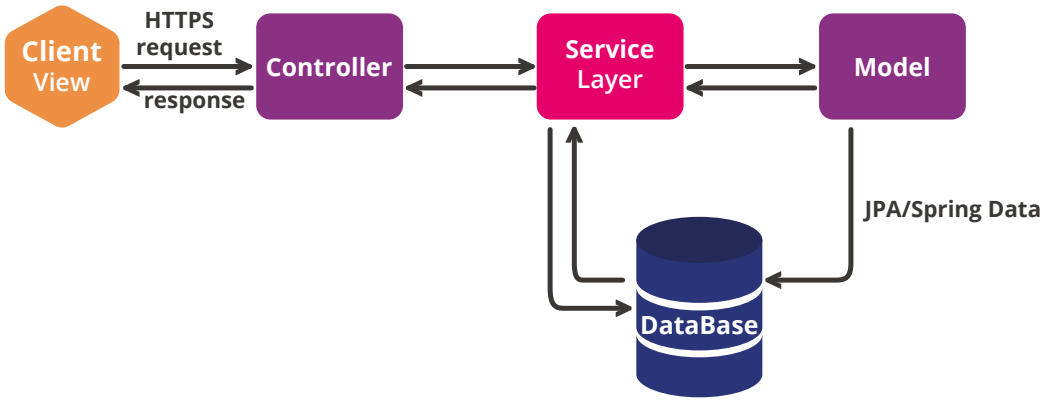


Figura 4 - Fluxo de comunicação Spring Web MVC

Fonte: Do autor (2022)

Para mais informações sobre formas de implementações no **Spring MVC**, confira a documentação oficial através do link ou do código QR a seguir:

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc

DTO

Data Transfer Object (DTO) é um padrão de projeto bastante usado para trafegar dados de uma camada a outra. É comum, em uma aplicação **Java**, que haja a necessidade, muitas vezes, de agrupar dados para fins de otimização ou, até mesmo, omitir dados para fins de segurança.
Usaremos aqui o "clássico" exemplo de cadastro de um usuário para demonstrar a aplicação desse padrão.
Observemos a entidade **User** abaixo:

```java
@Entity
public class User implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    private String password;
}
```

Trabalhando na arquitetura **MVC**, provavelmente teríamos um **Controller** para realizar algumas operações nessa entidade, por exemplo cadastrar um novo **Usuario**.

```java
@RestController
@RequestMapping("/user")
public class UserController {
    private final UserService service;
    @Autowired
    UserController(UserService service) {
        this.service = service;
    }
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<User> save(@RequestBody User obj) {
        return new ResponseEntity<> (service.create(obj));
    }

}
```

Perceba que, após salvar as informações do usuário em um banco de dados, neste exemplo estamos retornando dados *sensíveis* para a camada de visualização (*view*) como o atributo **password**.
Esse é o contexto perfeito para ser aplicado o **DTO**, em que efetuaríamos os seguintes ajustes.

1. Criaremos o objeto **UserDTO** com somente os campos que desejamos retornar para a *view*.

```
public class UserDTO implements Serializable {
    private Long id;
    private String name;
    private String email;
}
```

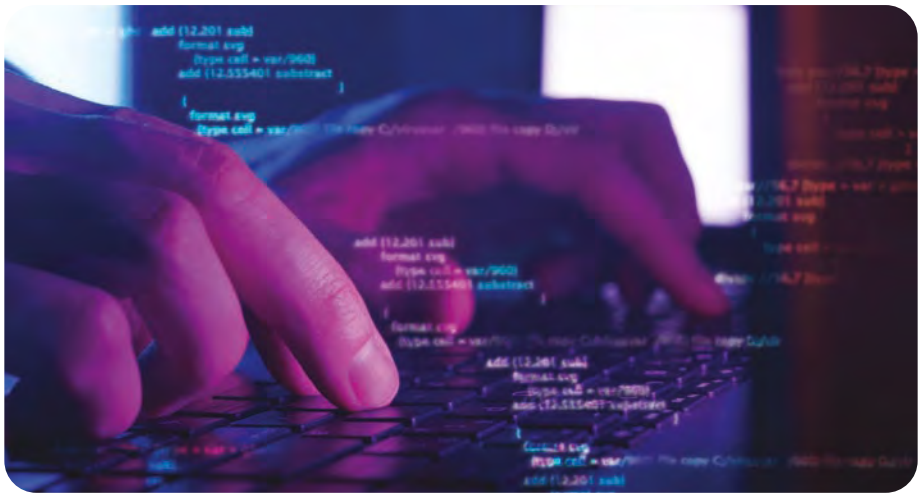
2. Alteraremos o objeto **UserController** para retornar o nosso DTO criado: **UserDTO**.

```java
@RestController
@RequestMapping("/user")
public class UserController {
    private final UserService service;
    @Autowired
    UserController(UserService service) {
        this.service = service;
    }
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<UserDTO> save(@RequestBody User obj) {
        return new ResponseEntity<> (service.create(obj));
    }
}
```

3. A implementação na camada de **Service Layer** também deve ser alterada. Procure saber um pouco mais da biblioteca *modelmapper* – ela é perfeita para esses casos.

Para mais informações sobre a biblioteca modelmapper, acesse o link ou o código QR a seguir:

http://modelmapper.org/



Segurança

É essencial saber os conceitos-base de autenticação e autorização para se ter uma aplicação **Web** segura e saber lidar com as opções do framework **Spring**.

Autenticação

No contexto geral de uma aplicação **Web**, autenticação significa validar se um usuário pode ou não acessar determinados recursos de uma aplicação. Geralmente, essa validação é feita por formulários de usuário e senha armazenados em banco de dados.

Autorização

Autorização é a extensão do contexto de autenticação. De maneira simples, um usuário pode ter acesso a uma aplicação **Web** através de seu usuário e senha, porém, a determinadas páginas do sistema o seu acesso pode ser negado.

Spring Security

O **Spring Security** é a estrutura dentro do framework **Spring** que dá suporte às questões de autenticação e autorização. Essa estrutura é desenvolvida sob as especificações do **Java EE** e oferece uma vasta extensão e personalização entre suas configurações para se adequar de acordo com a necessidade do sistema.

Algumas de suas características são:

- Suporte abrangente e extensível para autenticação e autorização;
- Proteção contra ataques, como *session fixation, clickjacking, cross-site request forgery*;
- Integração com a API do Servlet;
- Integração opcional com Spring Web MVC.

Para incluir essa estrutura em um projeto **Spring Boot**, basta incluir a dependência a seguir no **pom.xml** de sua aplicação:

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

Agora, o projeto autentica todas as requisições que chegam, e isso também inclui a classe **SecurityAutoConfiguration** que contém a configuração de segurança inicial/padrão.

A forma padrão de **Basic Authentication** é habilitada. As seguintes propriedades devem ser informadas no arquivo *application.properties* do seu sistema:

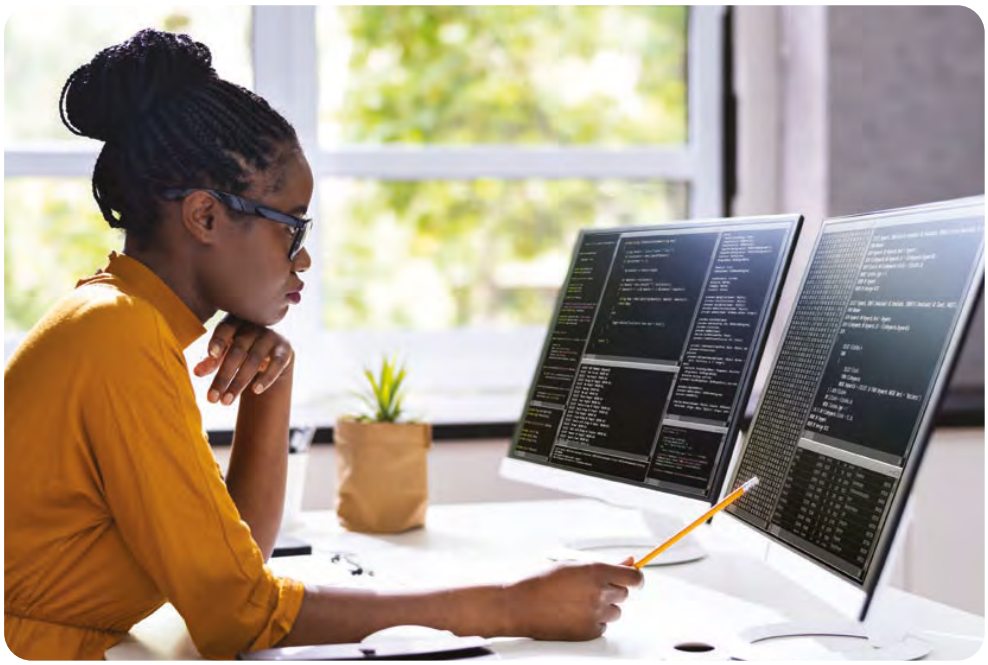
```
spring.security.user.name
spring.security.user.password
```

Caso não sejam informadas, uma senha padrão será gerada aleatoriamente e impressa no log do console.

```
Using default security password:
c8be15de-4488-4490-9dc6-fab3f91435c6
```

Porém, como vimos no início desta sessão, o **Spring Security** é uma estrutura muito flexível e nos permite desabilitar a forma de autenticação padrão, estendendo e implementando recursos conforme a necessidade da aplicação. Veja o exemplo a seguir:

```
@Configuration
@EnableWebSecurity
public class BasicConfiguration extends
        WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth)
            throws Exception {
        PasswordEncoder encoder = PasswordEncoderFactories.
                createDelegatingPasswordEncoder();
        auth.inMemoryAuthentication()
                .withUser("user")
                .password(encoder.encode("password"))
                .roles("USER")
                .and()
                .withUser("admin")
                .password(encoder.encode("admin"))
                .roles("USER", "ADMIN");
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .anyRequest()
                .authenticated()
                .and()
                .httpBasic();
    }
}
```



Outra forma muito comum em projetos utilizando o **Spring Security** é por tokens **JWT.** Vamos ver mais sobre esse assunto no próximo item de sessão.

JWT

JSON Web Token ou **JWT** são tokens de sessão que carregam dados criptografados usando um segredo privado ou uma chave pública/privada. Essa forma de autenticação é muito comum em **API Rest** e em projetos **BFF** (*Backend for Frontend*).
Esse token é dividido em 3 partes:

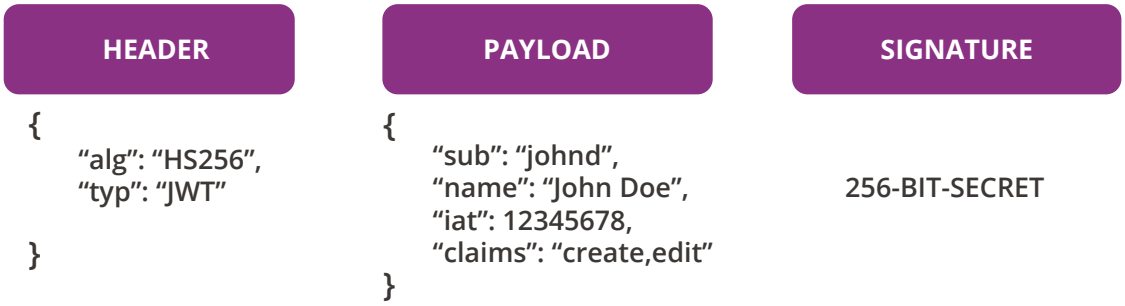


Figura 5 - Visão geral de um token JWT

Fonte: Do autor (2022)

- **Header**: especifica se o token será assinado e, caso seja, informa qual o algoritmo que será usado para a assinatura, usando a declaração obrigatória *alg* (algoritmo); além disso, pode conter as declarações opcionais *typ* (tipo de mídia) e *cty* (tipo de conteúdo).

- **Payload**: contém qualquer tipo de dado relevante para a aplicação; aqui, podemos entrar informações pessoais, grupos de acesso etc.; não existem declarações obrigatórias nesta sessão.

- **Signature**: é o formato do algoritmo usado para encriptar; é usada para verificar se a mensagem não foi alterada ao longo do seu envio e, no caso de tokens assinados com chave privada, também pode verificar se o remetente do JWT é quem diz ser.

O resultado é uma *string* em **Base64**, separada por 3 pontos, contendo as 3 partes mencionadas no quadro anterior:

```
 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
yJzdWIiOiJqb2huZCIsIm5hbWUiOiJKb2huIERvZSIsImlhdCI6MTEyMzQ1Njc4fQ.
XIbRNmUuvS6zj9E6CUaTS5TppCJzxb40fUeKNU_ZAIk
```

Caso queira visualizar essa *string* formatada, acesse o site do jwt.io e cole-a no campo *Encoded*.

Você pode acessar o site do jwt.io através do link ou do código QR a seguir:

https://jwt.io/#debugger-io

Acessando o site do jwt.io, já indicado, e colando essa *string* no campo *Encoded*, você verá, no campo *Decoded*, a seguinte saída:

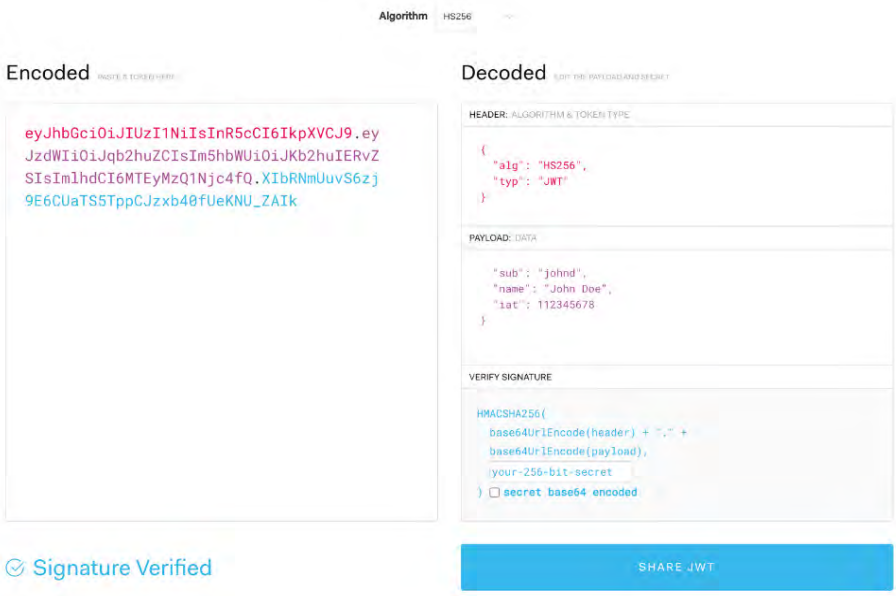


Figura 6 - Visão geral de m token JWT site jwt.io

Fonte: Do autor (2022)

No **Spring Security**, é possível usar **JWT** de algumas formas, umas mais simples e outras mais complexas, dependendo da necessidade de cada aplicação.

Em nosso contexto introdutório, vamos visualizar um exemplo em que o próprio *back-end* é responsável por gerar o token **JWT.** Essa função é conhecida como *Authorization Server,* sendo o próprio *back-end* responsável pela validação do token – essa função também pode ser conhecida como *Resource Server*.

Se você quiser aprofundar seus conhecimentos ainda mais nesses termos de segurança, pesquise por termos como “OAth 2.0” e “OpenId”. A busca por cada um desses dois termos pode ser acessada através dos links ou dos códigos QR indicados a seguir:

OAuth 2.0:

https://oauth.net/

OpenId:

https://openid.net/connect/

Vejamos o nosso exemplo na imagem a seguir.

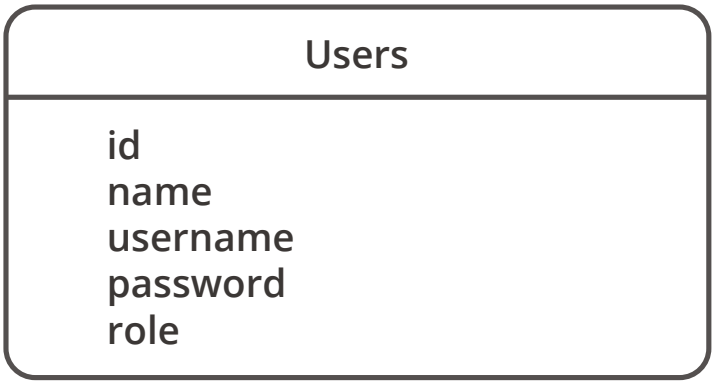


Figura 7 - Modelo de dados exemplo autenticação/autorização

Fonte: Do autor (2022)

Neste exemplo, **Users** representam os usuários cadastrados em nossa aplicação. Vamos usar aqui o H2 Database, e a coluna **role** representa as permissões de acesso dadas a cada usuário cadastrado.

Você pode acessar o site do H2 Database através do link ou do código QR a seguir:

https://www.h2database.com/html/main.html

Vamos ter dois *endpoints*:

1. findAll() irá consultar todos os usuários no banco. Essa operação será habilitada para os usuários que possuem a *role* **ROLE_USER**.
2. save() irá salvar os dados relacionados ao **Users** no banco. Essa operação será habilitada para os usuários que possuem a *role* **ROLE_ADMIN**.

Vejamos as classes usadas seguindo um padrão **MVC** para a criação do nosso exemplo:

Users

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Users implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false)
    private Long id;
    private String name;
    private String userName;
    private String password;
    private String role;
}
```

Nossa entidade que representa os usuários cadastrados no banco de dados **UserRepository**:

```
@Repository
public interface UserRepository extends JpaRepository<Users, Long> {
   Users findByUserName(String userName);
}
```

Classe que segue o padrão de **Repository** e é responsável por acessar e modificar os dados da entidade **Users**:

UserService

```
public interface UserService {
   Users saveUser(Users user);
   Users findByUsername(String userName);
   List<Users> findAll();
}
```

Interface contendo as operações básicas:

UserServiceImpl

```
@Service
public class UserServiceImpl implements UserService,
        UserDetailsService {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    @Autowired
    UserServiceImpl(UserRepository userRepository,
                    PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }
    @Override
    public Users saveUser(Users user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        return userRepository.save(user);
    }
    @Override
    public Users findByUsername(String userName) {
        return userRepository.findByUserName(userName);
    }
    @Override
    public List<Users> findAll() {
        return userRepository.findAll();
    }
    @Override
    public UserDetails loadUserByUsername(String username) throws
            UsernameNotFoundException {
        Users user = userRepository.findByUserName(username);
        if (user == null) {
            throw new UsernameNotFoundException("Usuario não
                    encontrado");
        }
        Collection<SimpleGrantedAuthority> authorities = new
                ArrayList<>();
        authorities.add(new SimpleGrantedAuthority(user.getRole()));
        return new org.springframework.security.core.userdetails.
                User(user.getUserName(), user.getPassword(), authorities);
    }
}
```

Implementação da interface **UsersService** contendo as operações de negócio deste nosso exemplo. Repare na operação loadUserByUsername() – é através dela que vamos recuperar as informações de acesso do nosso usuário; ela está sendo sobrescrita pela interface **UserDetailsService**.

UserController

```
@RestController
@RequestMapping("/user")
public class UserController {
    private final UserService userService;
    @Autowired
    UserController(UserService userService) {
        this.userService = userService;
    }
    @GetMapping
    public ResponseEntity<List<Users>> findAll() {
        return ResponseEntity.ok().body(userService.findAll());
    }
    @PostMapping
    public ResponseEntity<Users> save(@RequestBody Users user) {
        return ResponseEntity.ok().body(userService.saveUser(u-
ser));
    }
}
```

Vejamos nossa classe contendo os *endpoints* do exemplo em questão.
Para “agilizar”, vamos inserir alguns dados em nosso banco de dados para demonstração do nosso exemplo no início de nossa aplicação.
Vamos usar aqui a classe **CommandLineRunner** para inserir nossos dados.

```
@Bean
CommandLineRunner run(UserService userService) {
        return args -> {
            userService.saveUser( new
            Users(null,”Joao”,”joao”,”123”,”ROLE_USER”));
            userService.saveUser( new
            Users(null,”José”,”jose”,”123”,”ROLE_ADMIN”));
        };
}
```

Agora, vamos dar início à implementação de segurança com a classe **SecurityConfig**.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   private final UserDetailsService userDetailsService;
   private final BCryptPasswordEncoder bCryptPasswordEncoder;

   @Override
   protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
       auth.userDetailsService(userDetailsService).passwordEnco-
der(bCryptPasswordEncoder);
   }

   @Override
   protected void configure(HttpSecurity http) throws Exception {
       CustomAuthenticationFilter customAuthenticationFilter = new
CustomAuthenticationFilter(authenticationManager());
       customAuthenticationFilter.setFilterProcessesUrl('/user/lo-
gin');
       http.csrf().disable();
       http.sessionManagement().sessionCreationPolicy(SessionCrea-
tionPolicy.STATELESS);
       http.authorizeHttpRequests().antMatchers('/user/login/**').
permitAll();
       http.authorizeHttpRequests().antMatchers(GET, '/user/**').
hasAnyAuthority('ROLE_USER');
       http.authorizeHttpRequests().antMatchers(POST, '/user/**').
hasAnyAuthority('ROLE_ADMIN');
       http.authorizeHttpRequests().anyRequest().authenticated();
       http.addFilter(customAuthenticationFilter);
       http.addFilterBefore(new CustomAutorizationFilter(),Userna-
mePasswordAuthenticationFilter.class);
   }

   @Bean
   @Override public AuthenticationManager authenticationManager()
throws Exception {
       return super.authenticationManager();
   }

}
```

Neste nosso exemplo, estamos sobrescrevendo alguns métodos da classe **WebSecurityConfigurerAdapter**, que é a classe, em nosso caso específico, responsável por nos fornecer opções de segurança conforme o que queremos implementar.

A operação configure(AuthenticationManagerBuilder auth) refere-se à operação loadUserByUsername que implementamos na classe **UserServiceImpl.** E a operação configure(HttpSecurity http) configura, neste nosso exemplo, a nossa segurança de acordo com suas **Roles**.

Como vimos no início desta sessão, **JWT** são tokens de sessão que carregam dados criptografados usando um segredo privado ou uma chave pública/privada. Vamos precisar adicionar uma biblioteca em nosso pom.xml para podermos criar e gerenciar os tokens **JWT** em nossa aplicação.

```
<dependency>
   <groupId>com.auth0</groupId>
   <artifactId>java-jwt</artifactId>
   <version>4.2.1</version>
</dependency>
```

Vamos precisar agora implementar o *filter* **CustomAuthenticationFilter**:

```
public class CustomAuthenticationFilter extends UsernamePasswor-
dAuthenticationFilter {
   private final AuthenticationManager authenticationManager;
   @Autowired
   public CustomAuthenticationFilter(AuthenticationManager au-
thenticationManager) {
        this.authenticationManager = authenticationManager;
   }
   @Override
   public Authentication
   attemptAuthentication(HttpServletRequest request, HttpServle-
tResponse response) throws AuthenticationException
   {
        String username = request.getParameter("userName");
        String password = request.getParameter("password");
        UsernamePasswordAuthenticationToken authenticationToken =
new UsernamePasswordAuthenticationToken(username, password);
        return authenticationManager.authenticate(authentication-
Token);
   }
   @Override
   protected void successfulAuthentication(
           HttpServletRequest request,
           HttpServletResponse response,
           FilterChain chain,
           Authentication authResult) throws IOException, Servle-
tException
   {
        User user = (User) authResult.getPrincipal();
        Algorithm algorithm = Algorithm.HMAC256("secret".getBy-
tes());
        String acces_token = JWT.create().withSubject(user.getU-
sername())
                .withExpiresAt(new Date(System.currentTimeMillis()
+ 10 * 60 * 1000))
                .withIssuer(request.getRequestURL().toString())
                .withClaim("roles", user.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority).collect(Col-
lectors.toList()))
                .sign(algorithm);
```

```
        String refresh_token = JWT.create()
                .withSubject(user.getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis()
+ 30 * 60 * 1000))
                .withIssuer(request.getRequestURL().toString())
                .sign(algorithm);
        Map<String, String> tokens = new HashMap<>();
        tokens.put("acces_token", acces_token);
        tokens.put("refresh_token", refresh_token);
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        new ObjectMapper().writeValue(response.getOutputStream(),
tokens);
    }
}
```

Esse *filter* será responsável por interceptar as requisições para validar se o usuário que está acessando um determinado *endpoint* tem acesso à aplicação, ou seja, se esse usuário está cadastrado na entidade **Users** (Autenticação).

Também, é nesse *filter* que é gerado o token **JWT**, como pudemos ver na operação successfulAuthentication().

O próximo *filter* que teremos de implementar é o **CustomAutorizationFilter**:

```
public class CustomAutorizationFilter extends OncePerReques-
tFilter {

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain
    ) throws ServletException, IOException {
        if (request.getServletPath().equals("/user/login")) {
            filterChain.doFilter(request, response);
        } else {
            String authorizationHeader = request.getHeader(AU-
THORIZATION);
            if (authorizationHeader != null && authorizationHea-
der.startsWith("Bearer ")) {
                try {
                    String token = authorizationHeader.substrin-
g("Bearer".length());
                    Algorithm algorithm = Algorithm.HMAC256("se-
cret".getBytes());
                    JWTVerifier verifier = JWT.require(algorithm).
build();
                    DecodedJWT decodedJWT = verifier.verify(-
token);
                    String username = decodedJWT.getSubject();
                    String[] roles = decodedJWT.getClaim("ro-
les").asArray(String.class);
                    Collection<SimpleGrantedAuthority> authori-
ties = new ArrayList<>();
                    stream(roles).forEach(role -> authorities.
add(new SimpleGrantedAuthority(role)));
                    UsernamePasswordAuthenticationToken authen-
ticationToken = new UsernamePasswordAuthenticationToken(userna-
me, null, authorities);
                    SecurityContextHolder.getContext().setAu-
thentication(authenticationToken);
                    filterChain.doFilter(request, response);
```

```
                } catch (Exception exception) {
                    response.setHeader("error",exception.getMes-
sage());
                    response.sendError(HttpServletResponse.SC_
FORBIDDEN);
                }
            } else {
                filterChain.doFilter(request,response);
            }
        }
    }
}
```

É nesse *filter* que é implementada a lógica de validação das permissões (**Roles**) dando suporte à *Autorização, como vimos no início desta sessão.

Com os passos mostrados, nosso exemplo já está implementando segurança via token **JWT**. Basta agora recuperar o token **JWT** através do *endipoint* /user/login, passando os parâmetros userName e password, para então receber os campos refresh_token e acces_token.

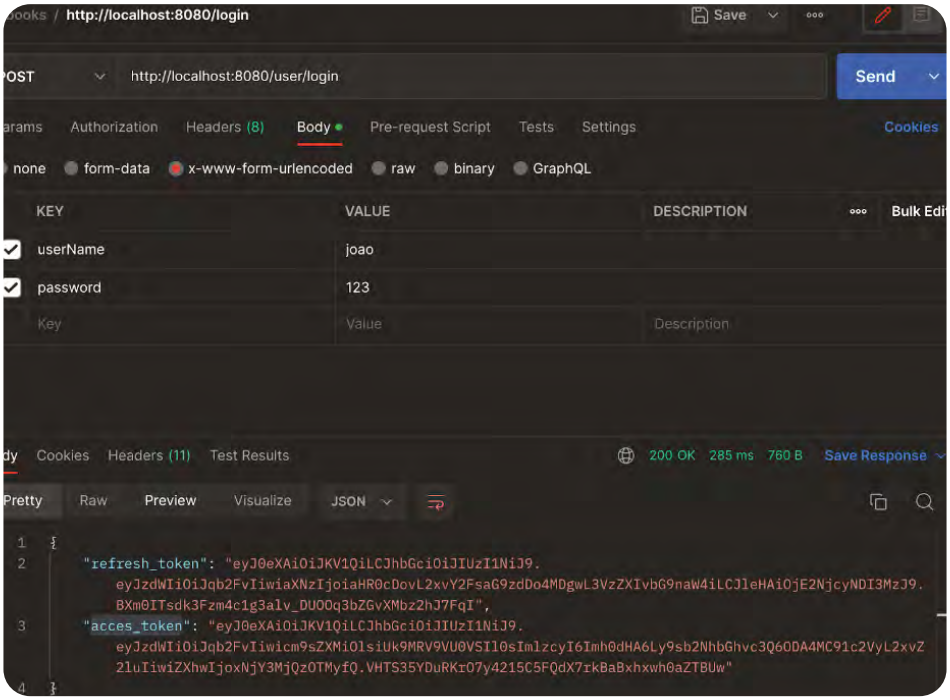


Figura 8 - Postman exemplo de requisição de login

Fonte: Do autor (2022)

Após adquirir o token, você pode enviar uma nova solicitação incluindo o parâmetro Authorization com o conteúdo do token com o prefixo “Bearer”, conforme a permissão de cada usuário.

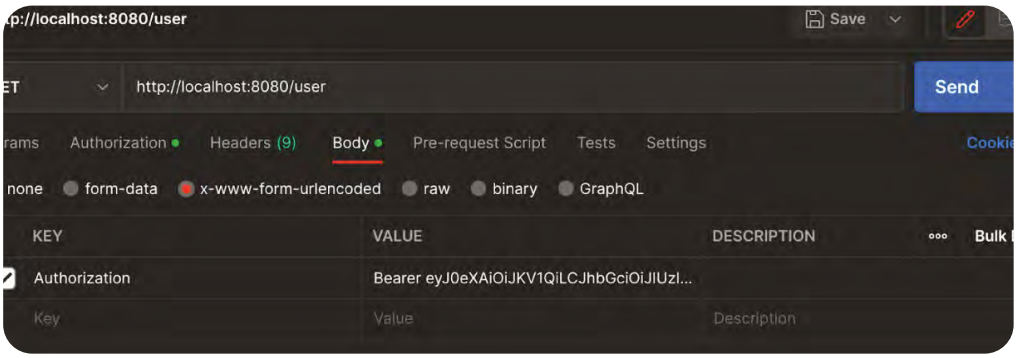


Figura 9 - Postman exemplo de requisição de endpoint enviando o token JWT

Fonte: Do autor (2022)

Você pode acessar o código completo do nosso exemplo através do link ou do código QR a seguir:

https://github.com/jeffersonoh/ebooks/tree/main/demo-spring-jwt

Cache

Cache é um recurso interessante de se ver. De forma geral, ele tem o intuito de melhorar o desempenho de uma aplicação.

Imagine uma consulta de banco de dados que é muito usada em nosso sistema, porém, os dados retornados não são passíveis de mudança. Como no caso de uma consulta que traz todos os municípios brasileiros, é dificilmente adicionado um novo município no Brasil e, no que se refere a desempenho, essa é uma consulta que retorna muitos dados.

Tendo esse cenário em mente, vamos implementar um exemplo, de maneira simplificada, exatamente desta consulta, em que o resultado é armazenado em cache na primeira consulta realizada e as consultas posteriores são realizadas no cache armazenado.

Em primeiro lugar, precisaremos adicionar a biblioteca **spring-boot-starter-cache** em nossa aplicação.

Para mais detalhes sobre a biblioteca **spring-boot-starter-cache**, consulte a sua documentação oficial disponível no link ou código QR a seguir:

https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html

Vejamos as classes usadas seguindo um padrão **MVC** para criação do nosso exemplo:

Municipio

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Municipio implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false)
    private Long id;
    private String uf;
    private String name;
}
```

MunicipioRepo

```
@Repository
public interface MunicipioRepo extends JpaRepository<Municipio,
Long> {

}
```

Classe que segue o padrão de **Repository** e é responsável por acessar e modificar os dados da entidade **Municipio**.

MunicipioService

```
public interface MunicipioService {
    Municipio save(Municipio municipio);
    List<Municipio> getAll();
}
```

Interface contendo as operações básicas.

MunicipioServiceImpl

```
@Service
@CacheConfig(cacheNames={"municipios"})
public class MunicipioServiceImpl implements
        MunicipioService{
    private final MunicipioRepo municipioRepo;
    @Autowired
    public MunicipioServiceImpl(MunicipioRepo municipioRepo) {
        this.municipioRepo = municipioRepo;
    }
    @Override
    public Municipio save(Municipio municipio) {
        return municipioRepo.save(municipio);
    }
    @Cacheable
    @Override public List<Municipio> getAll() {
        return municipioRepo.findAll();
    }
}
```

Implementação da interface **MunicipioService** contendo as operações de negócio deste nosso exemplo. Repare na operação getAll(). É através dela que vamos recuperar as informações de todos os **Municipios** cadastrados – ela está usando o cache CacheConfig.

MunicipioController

```
@RestController
@RequestMapping("/municipios")
public class MunicipioController {
    private final MunicipioService service;
    @Autowired
    public MunicipioController(MunicipioService service) {
        this.service = service;
    }
    @GetMapping
    public ResponseEntity<List<Municipio>> findAll() {
        return ResponseEntity.ok().body(service.getAll());
    }
    @PostMapping
    public ResponseEntity<Municipio> save(@RequestBody Municipio
                                                    municipio) {
        return ResponseEntity.ok().body(service.save(municipio));
    }
}
```

Vejamos nossa classe contendo os *endpoints* deste nosso exemplo.

Para “agilizar”, vamos inserir alguns dados em nosso banco de dados para demonstração do nosso exemplo no início de nossa aplicação.

Vamos usar aqui a classe **CommandLineRunner** para inserir nossos dados.

```
@Bean
CommandLineRunner run(MunicipioService municipioService) {
	return args -> {
		municipioService.save(new Municipio(null, “SC”,
		“ABELARDO LUZ”));
		municipioService.save(new Municipio(null, “SC”,
		“AGRONÔMICA”));
		municipioService.save(new Municipio(null, “SC”,
		“ÁGUAS MORNAS”));
		municipioService.save(new Municipio(null, “SC”,
		“ALFREDO WAGNER”));
		municipioService.save(new Municipio(null, “SC”, “ANGELINA”));
		municipioService.save(new Municipio(null, “SC”, “ANITA
		GARIBALDI”));
	};
}
```

Você pode acessar o código completo deste exemplo através do link ou do código QR a seguir:

https://github.com/jeffersonoh/ebooks/tree/main/demo-spring-cahe

Profiles

Profile é um conjunto de propriedades, ou mesmo classes, que devem ser habilitadas em algum ambiente específico, como produção, desenvolvimento, teste etc.

Esse recurso permite ativar diferentes perfis para vários ambientes, o que nos dá mais segurança na execução de testes em diferentes ambientes.

Para termos diferentes perfis, precisamos adicionar nos arquivos de propriedades (o **Spring Boot** aceita dois tipos de formatos de arquivos: *.properties* ou *.yml*) os sufixos de acordo com cada ambiente. Neste exemplo, usaremos o formato *.properties*.

Suponhamos que vamos ter um ambiente de **Desenvolvimento** no qual nossa aplicação rode em uma porta diferente. Então, precisaremos ter o arquivo **application-dev.properties** em nossas configurações.

```
server.port:9090
```

E em ambiente de testes, queremos rodar nossa aplicação na porta **8080**, então, teríamos o arquivo **application-test.properties** com a seguinte configuração:

```
server.port:8080
```

Para escolher em qual “Profile” nossa aplicação irá rodar, temos de especificar o “Profile” no *start*, por exemplo:

```
java -jar meuapp.jar --spring.profiles.active=dev
```

Veja outras formas de configurações de um *Profile* consultando a documentação oficial através do link ou do código QR a seguir:

https://docs.spring.io/spring-boot/docs/1.2.0.M1/reference/html/boot-features-profiles.html

Monitoramento

O **Spring Boot** possui duas bibliotecas principais que se complementam, fornecendo recursos de pós-produção para acompanhamento e monitoramento de uma aplicação.

Estamos falando aqui do **Spring Boot Actuator** e do **Spring Boot Admin**.

Spring Boot Actuator

O **Spring Boot Actuator** é uma biblioteca que disponibiliza alguns *endpoints* para monitoramento de uma aplicação, tais como funcionamento do banco de dados, coleta de tráfego, dados de consumo de CPU e memória da JVM, entre outros.

Vejamos o funcionamento dessa biblioteca de uma forma mais prática.

Primeiramente, a biblioteca deve ser adicionada ao seu pom.xml.

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

Por padrão, a sua URL de acesso é /actuator, mas isso pode ser alterado incluindo-se a informação a seguir no arquivo de propriedades de sua aplicação.

```
management.endpoints.web.base-path=/manage
// 20221101093135
// http://localhost:8080/api/manage
{
    '_links': {
        'self': {
            'href': 'http://localhost:8080/api/actuator',
            'templated': false
        },
        'health': {
            'href': 'http://localhost:8080/api/actuator/health',
            'templated': false
        },
        'health-path': {
            'href': 'http://localhost:8080/api/actuator/health/
{*path}',
            'templated': true
        }
    }
}
```

Considere adicionar, no arquivo de propriedade, as seguintes informações:

```
management.endpoint.health.show-details=always
management.endpoints.web.exposure.include=*
```

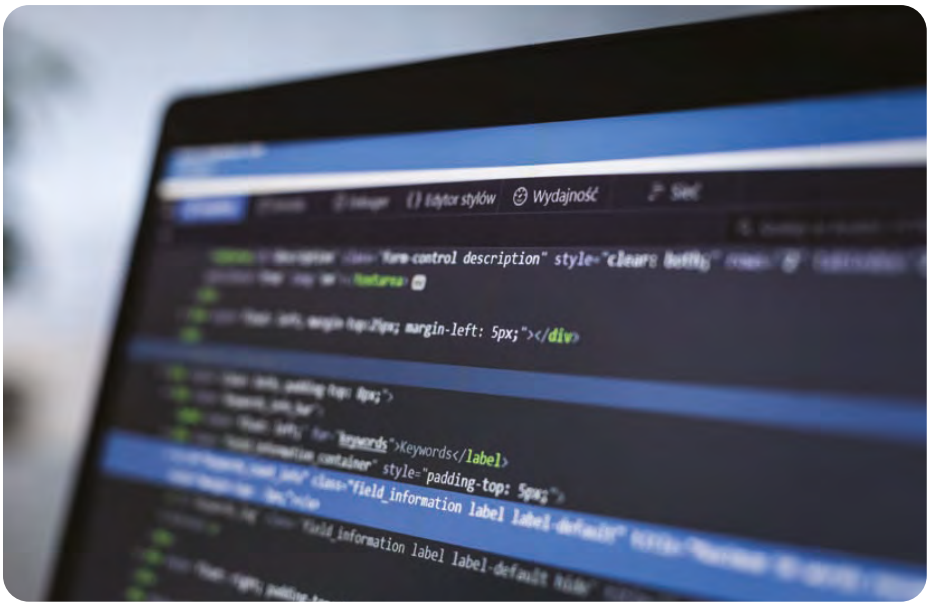
Se as adicionarmos, veremos mais informações de monitoramento da nossa aplicação.

```
// 20221101092943
// http://localhost:8080/api/manage
{
    '_links': {
        'self': {
            'href': 'http://localhost:8080/api/actuator',
            'templated': false
        },
        'beans': {
            'href': 'http://localhost:8080/api/actuator/beans',
            'templated': false
        },
        'caches-cache': {
            'href': 'http://localhost:8080/api/actuator/caches/{ca-
che}',
            'templated': true
        },
        'caches': {
            'href': 'http://localhost:8080/api/actuator/caches',
            'templated': false
        },
        'health': {
            'href': 'http://localhost:8080/api/actuator/health',
            'templated': false
        },
        'health-path': {
            'href': 'http://localhost:8080/api/actuator/health/
{*path}',
            'templated': true
        },
        'info': {
            'href': 'http://localhost:8080/api/actuator/info',
            'templated': false
        },
```

```
        'conditions': {
            'href': 'http://localhost:8080/api/actuator/condi-
tions',
            'templated': false
        },
        'configprops': {
            'href': 'http://localhost:8080/api/actuator/confi-
gprops',
            'templated': false
        },
        'configprops-prefix': {
            'href': 'http://localhost:8080/api/actuator/configprops/
{prefix}',
            'templated': true
        },
        'env': {
            'href': 'http://localhost:8080/api/actuator/env',
            'templated': false
        },
        'env-toMatch': {
            'href': 'http://localhost:8080/api/actuator/env/{toMat-
ch}',
            'templated': true
        },
        'loggers': {
            'href': 'http://localhost:8080/api/actuator/loggers',
            'templated': false
        },
        'loggers-name': {
            'href': 'http://localhost:8080/api/actuator/loggers/
{name}',
            'templated': true
        },
        'heapdump': {
            'href': 'http://localhost:8080/api/actuator/heapdump',
            'templated': false
        },
        'threaddump': {
            'href': 'http://localhost:8080/api/actuator/thread-
dump',
            'templated': false
        },
        'metrics-requiredMetricName': {
            'href': 'http://localhost:8080/api/actuator/metrics/
{requiredMetricName}',
            'templated': true
        },
```

```
        'metrics': {
            'href': 'http://localhost:8080/api/actuator/metrics',
            'templated': false
        },
        'scheduledtasks': {
            'href': 'http://localhost:8080/api/actuator/scheduled-
tasks',
            'templated': false
        },
        'mappings': {
            'href': 'http://localhost:8080/api/actuator/mappings',
            'templated': false
        }
    }
}
```

O parâmetro management.endpoint.health.show-details=always indica que o contexto /helth ficará sempre disponível. Essa é uma configuração muito comum para projetos feitos em **Spring Boot** na arquitetura de micros serviços. Já o parâmetro management.endpoints.web.exposure.include=* indica que vai disponibilizar métricas de todos os *endpoints* disponíveis na aplicação.



Agora, suponhamos que queremos saber por quanto tempo nossa aplicação está disponível.

Para isso, temos de acessar a URL:

```
http://localhost:8080/api/manage/metrics/application.ready.time
```

Perceba que, aqui, estamos passando o id application.ready.time. Esse é o id responsável por retornar a informação que queremos; existem diversos ids disponíveis dentro do contexto de /metrics.

O resultado será a seguinte saída:

```
// 20221101094331
// http://localhost:8080/api/manage/metrics/application.
ready.time {
    'name': 'application.ready.time',
    'description': 'Time taken (ms) for the application to be ready
to service requests',
    'baseUnit': 'seconds',
    'measurements': [{
        'statistic': 'VALUE',
        'value': 3.302
    }],
    'availableTags': [{
        'tag': 'main.application.class',
        'values': ['com.example.demospringactuator.DemoSpringActua-
torApplication']
    }]
}
```

Confira mais detalhes de *endpoints* disponíveis do **Actuator** em sua documentação oficial através do link ou do código QR a seguir:

https://docs.spring.io/spring-boot/docs/current/actuator-api/

Bem, você já deve ter notado que ver métricas via requisições e receber um *json* não é muito usual no dia a dia. É aí que entra o **Spring Boot Admin**. Vejamos mais detalhes sobre ele.

Spring Boot Admin

O **Spring Boot Admin** é uma aplicação Web que fornece *Dashboards* para visualização de métricas de uma maneira mais usual.

Para vermos seu funcionamento, vamos precisar ter uma aplicação contendo a biblioteca:

```
<dependency>
   <groupId>de.codecentric</groupId>
   <artifactId>spring-boot-admin-starter-server</artifactId>
   <version>2.1.6</version>
</dependency>
```

Precisaremos também incluir na classe principal da aplicação a anotação **@EnableAdminServer**.

Vamos alterar a porta padrão de **8080** para **8081**, para podermos subir as duas aplicações em ambiente local sem conflito de porta. Se você quiser fazer o mesmo, é só incluir server.port=8081 em seu arquivo de propriedade.

Após essas configurações, podemos iniciar o projeto. O resultado será a tela a seguir:

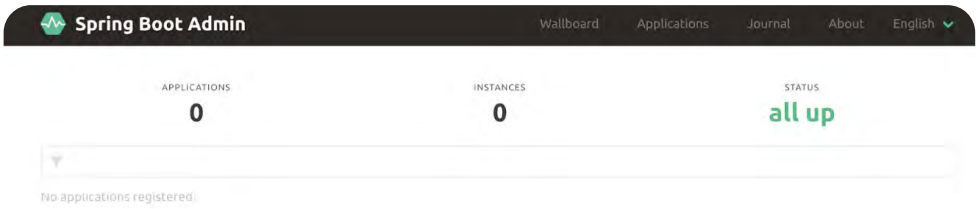


Figura 10 - Spring Boot Admin visão geral

Fonte: Do autor (2022)

Agora, precisamos conectar nosso exemplo de *client* no **Spring Boot Admin**. Para isso, vamos precisar adicionar a biblioteca **spring-boot-admin-starter-client**.

```
<dependency>
   <groupId>de.codecentric</groupId>
   <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

Logo em seguida, precisaremos inserir a informação spring.boot.admin.client.url=http://localhost:8081 no arquivo de propriedade de nossa aplicação.

Ao iniciar nosso *client*, percebemos que a aplicação foi registrada no **Spring Boot Admin**.

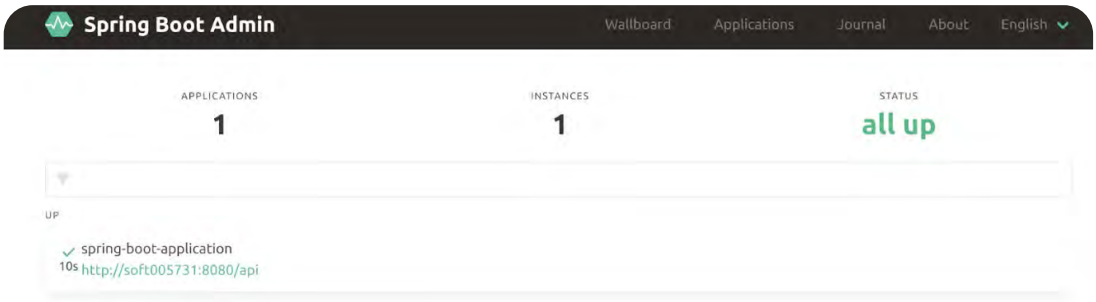


Figura 11 - Spring Boot Admin já monitorando a nossa aplicação de exemplo

Fonte: Do autor (2022)

Agora, podemos clicar em *Wallboard* para ver mais detalhes de métricas de nossa aplicação *client*:

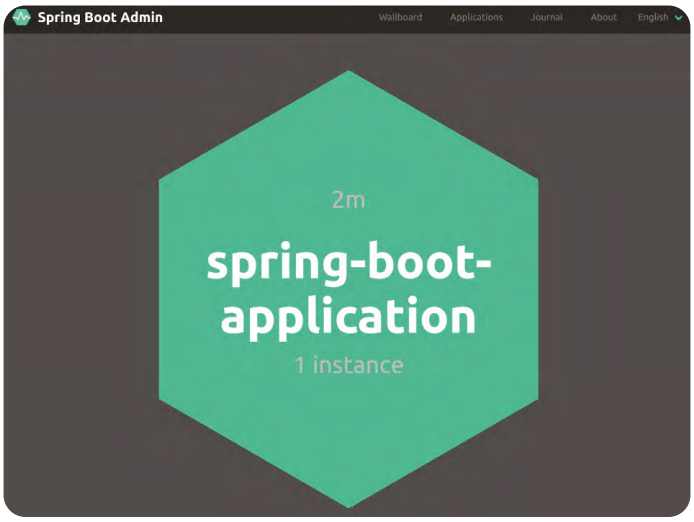


Figura 12 - Spring Boot Admin Wallboard

Fonte: Do autor

Pronto! Agora, podemos ver de maneira mais usual cada métrica de nossa aplicação *client*.

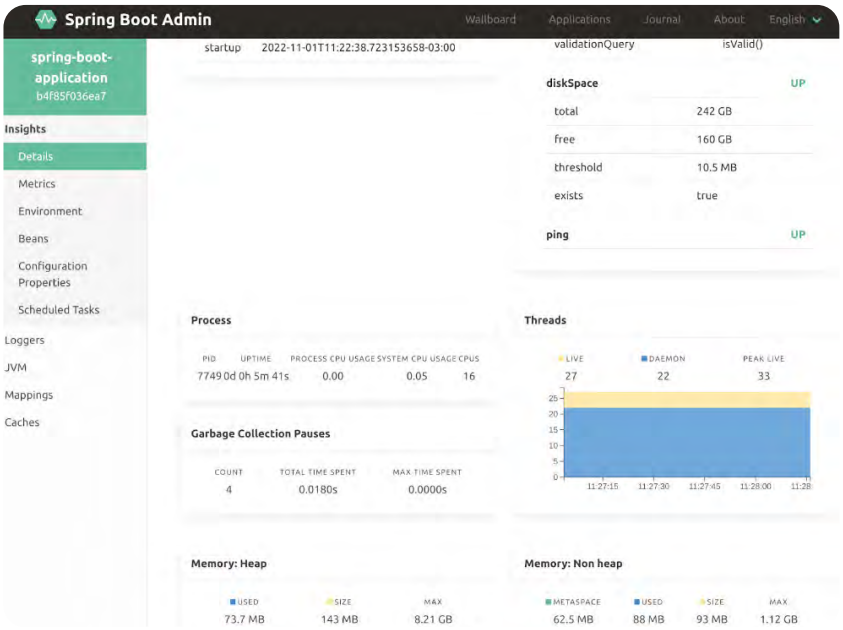


Figura 13 - Spring Boot Admin “Details”

Fonte: Do autor (2022)

Os códigos completos dos exemplos aqui apresentados podem ser vistos nos repositórios disponibilizados nos links ou códigos QR a seguir:

https://github.com/jeffersonoh/ebooks/tree/main/demo-spring-actuator

https://github.com/jeffersonoh/ebooks/tree/main/demo-spring-admin

CONCLUSÃO

Este e-book apresentou alguns conceitos-base para quem quer ingressar no contexto **Web** em **Java**. Portanto, em relação aos temas aqui abordados, é recomendado um aprofundamento maior, conforme a necessidade de cada projeto, em sua documentação de referência.

Espero que você aproveite este conteúdo da melhor maneira possível.

REFERÊNCIAS

JAVA.COM. O que é tecnologia Java e por que preciso dela? Oracle, 2022. Disponível em: https://www.java.com/pt-BR/download/help/whatis_java.html. Acesso em: 28 out. 2022.

SAMPAIO, Juliana. Java: tudo o que você precisa saber para começar. Zup Innovation, 13 maio 2021. Disponível em: https://www.zup.com.br/blog/java. Acesso em: 28 out. 2022.

IT FORUM. James Gosling, criador do Java, fala sobre carreira, Low Code e futuro da linguagem. IT Forum, 07 ago. 2020. Disponível em: https://itforum.com.br/noticias/james-gosling-criador-do-java-fala-sobre-carreira-low-code-e-futuro-da-linguagem/. Acesso em: 28 out. 2022.

Jefferson Henrique

Arquiteto de Software na Softplan e Mentor Educacional no LAB365 do SENAI/SC.

SENAI <LAB365>