05

Mãos na massa: Publicando o projeto

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Crie sua conta pessoal no Heroku. Acesse o endereço https://www.heroku.com (https://www.heroku.com) e procure pela opção **Sign Up**. Preencha com seus dados pessoais e finalize o processo de cadastro para poder continuar.

2) Instale o **Heroku CLI**, acessando este link (https://devcenter.heroku.com/articles/heroku-cli) e instale o **Heroku CLI**.

3) Feita a instalação, abra o terminal (ou o Prompt de Comando, se você utilizar Windows) e digite:

```
heroku login
```

Feito o login, você deverá ver a mensagem: *Authentication successful.*

4) Para que o Heroku rode sua aplicação, ela precisa ser uma aplicação executada a partir de um *JAR*. Use um plugin para realizar esse trabalho. Abra o seu **pom.xml** e procure pelas tags `<plugins>` e `</plugins>` . Dentro dela, depois de qualquer `<plugin>` que já possa existir, adicione o código abaixo:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.3</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals><goal>copy</goal></goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>com.github.jsimone</groupId>
                        <artifactId>webapp-runner</artifactId>
                        <version>7.0.57.2</version>
                        <destFileName>webapp-runner.jar</destFileName>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
```

5) Como o Heroku só suporta atualmente o **PostgreSQL**, adicione a sua dependência no **pom.xml**. Abra seu **pom.xml** e adicione o código abaixo, entre as *tags* `<dependencies>` e `</dependencies>` :

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.4-1201-jdbc41</version>
</dependency>
```

6) Na classe `JPAConfiguration`, faça com que o método `additionalProperties` seja um `@Bean` e anote-o com `@Profile("dev")`. Além disso, receba-o como parâmetro no método `entityManagerFactory`:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource,
        Properties additionalProperties) {

    LocalContainerEntityManagerFactoryBean factoryBean =
        new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models");
    factoryBean.setDataSource(dataSource);

    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    factoryBean.setJpaVendorAdapter(vendorAdapter);
    factoryBean.setJpaProperties(additionalProperties);

    return factoryBean;
}

@Bean
@Profile("dev")
public Properties additionalProperties() {
    Properties props = new Properties();
    props.setProperty("hibernate.dialect",
        "org.hibernate.dialect.MySQL5Dialect");
    props.setProperty("hibernate.show_sql", "true");
    props.setProperty("hibernate.hbm2ddl.auto", "update");

    return props;
}
```

7) Crie a classe `JPAProductionConfiguration` no pacote `br.com.casadocodigo.loja.conf`, pegando os dados de conexão pelo `Enviroment` injetado pelo Spring. Também anote a classe como sendo o *profile* de produção com `@Profile("prod")`:

```
@Profile("prod")
public class JPAProductionConfiguration {

    @Autowired
    private Environment environment;

    @Bean
    public Properties additionalProperties() {
        Properties props = new Properties();
        props.setProperty("hibernate.dialect",
            "org.hibernate.dialect.PostgreSQLDialect");
        props.setProperty("hibernate.show_sql", "true");
        props.setProperty("hibernate.hbm2ddl.auto", "update");
        return props;
    }

    @Bean
    public DataSource dataSource() throws URISyntaxException {
```

```
        DriverManagerDataSource dataSource =
            new DriverManagerDataSource();
        dataSource.setDriverClassName("org.postgresql.Driver");

        URI dbUrl = new URI(environment.getProperty("DATABASE_URL"));
        dataSource.setUrl("jdbc:postgresql://" + dbUrl.getHost() +
            ":" + dbUrl.getPort() + dbUrl.getPath());
        dataSource.setUsername(dbUrl.getUserInfo().split(":")[0]);
        dataSource.setPassword(dbUrl.getUserInfo().split(":")[1]);

        return dataSource;
    }
}
```

8) Na classe `ServletSpringMVC`, adicione a nova classe de configuração no método `getRootConfigClasses`. Além disso, comente o método `onStartup`:

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] {AppWebConfiguration.class , JPAConfiguration.class,
        SecurityConfiguration.class, JPAProductionConfiguration.class};
}
```

9) O Heroku precisa que você crie um arquivo chamado **Procfile**. Esse arquivo contém a linha de execução da sua aplicação, que o Heroku irá executar. Crie-o na raiz do seu projeto com o seguinte conteúdo:

```
web: java $JAVA_OPTS -jar -Dspring.profiles.active=prod target/dependency/webapp-runner.jar --p
```

10) Pelo terminal (ou Prompt de Comando), acesse a pasta raiz do seu projeto, e faça a criação do repositório, o `commit` e o `push` do seu código, com os seguintes comandos:

```
git init
git add -A
git commit -m "Sua mensagem de commit"
heroku apps:create nome-do-seu-projeto
git push heroku master
```

11) Agora, crie uma URL aleatória para criar um usuário que permita você acessar o *admin* do seu sistema da Casa do Código. Então, crie um método no `HomeController` que realize a inserção dos dados do usuário e permita que ele acesse o sistema:

```
@Controller
public class HomeController {

    @Autowired
    private UsuarioDAO usuarioDao;

    @Transactional
    @ResponseBody
    @RequestMapping("/url-magica-maluca-oajksfbvad6584i57j54f9684nvi658efnoewfmnvowefnoeijn")
```

```
    public String urlMagicaMaluca() {
        Usuario usuario = new Usuario();
        usuario.setNome("Admin");
        usuario.setEmail("admin@casadocodigo.com.br");
        usuario.setSenha("$2a$10$lt7pS7Kxxe5JfP.vjLNSyOXP11eHgh7RoPxo5fvvbMCZkCUss2DGu");
        usuario.setRoles(Arrays.asList(new Role("ROLE_ADMIN")));

        usuarioDao.gravar(usuario);

        return "Url Mágica executada";
    }

    // restante do código da classe omitido

}
```

Lembrando que a URL pode ser qualquer uma, modifique conforme desejar.

12) Além disso, lembre-se de liberar o acesso a esse método na classe de segurança. Então, na classe `SecurityConfiguration`, no método `configure`, adicione o código abaixo:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    // não altere nada acima, apenas acrescente a linha abaixo:
    .antMatchers("/url-magica-maluca-oajksfbvad6584i57j54f9684nvi658efnoewfmnvowefnoeijn").perm
    // Coloque o código acima antes da linha   >>     .anyRequest().authenticated()
}
```

13) Na classe `Role`, crie dois construtores, um vazio e um que recebe o nome da *role*:

```
public Role() {

}

public Role(String nome) {
    this.nome = nome;
}
```

14) Na classe `Usuario`, guando um usuário é salvo, garanta que a sua *role* será gravada também, alterando a anotação `@OneToMany` do atributo `roles`:

```
@OneToMany(fetch=FetchType.EAGER,
    cascade=CascadeType.PERSIST)
private List<Role> roles = new ArrayList<Role>();
```

15) Na classe `UsuarioDAO`, crie o método `gravar`:

```
public void gravar(Usuario usuario) {
    manager.persist(usuario);
}
```

16) Por fim, realize a publicação do seu novo código para o Heroku com `commit` e depois `push` das suas alterações:

```
git add .
git commit -m "Sua mensagem de commit"
git push heroku master
```