02

Deploy no Heroku

Transcrição

A nossa aplicação já conta com diversos recursos interessantes. Já temos o cadastro de produtos, as páginas de listagem e de detalhes de um produto. Já se pode efetuar compras adicionando os produtos ao carrinho e enviamos até emails através da aplicação.

É chegado o momento em que podemos por a aplicação online para que nossos usuários possam acessa-la e efetuar suas compras na mesma. Para isso usaremos o Heroku (http://www.heroku.com), um servidor de aplicações que usa *Git* como base para publicação e gerenciamento das aplicações.

Para conseguirmos interagir com o Heroku (http://www.heroku.com) precisamos instalar uma ferramenta chamada Heroku Toolbelt (https://devcenter.heroku.com/articles/heroku-command-line#download-and-install) que é a ferramenta de linha de comando do Heroku (http://www.heroku.com). Lembre-se de se cadastrar no Heroku (http://www.heroku.com) para poder completar este passo do curso. Você pode fazer seu cadastro aqui: signup.heroku.com/dc (https://signup.heroku.com/dc)

Com o terminal aberto e após fazer o cadastro e instalação do *Toolbelt*. Navegaremos até a pasta do projeto com o comando `cd` . Como por exemplo: `cd workspace/pasta_do_projeto` . Onde neste caso a pasta do projeto se chama `casadocodigo` e após isso executaremos o comando `git init` . O qual cria um repositório *Git*.

Observação: Caso não esteja familiarizado com o *Git*, recomendamos que faça o curso do mesmo que está disponível aqui na Alura. Clique aqui para ver o curso de Git (https://cursos.alura.com.br/course/git)

Os próximos passos são: Adicionar todos os arquivos ao repositório *git* e fazer *commit* destas alterações no repositório. Os comandos respectivos as tarefas são: `git add -A` e `git commit -m "initial commit"` . Neste último caso estamos adicionando uma mensagem ao *commit* através da opção `-m` . Veja os passos feitos até aqui na imagem abaixo:

```
aluras-Mac-mini:~ alura$ cd Documents/paulo/workspace/casadocodigo/
aluras-Mac-mini:casadocodigo alura$ git init
Initialized empty Git repository in /Users/alura/Documents/paulo/workspace/casa
docodigo/.git/
aluras-Mac-mini:casadocodigo alura$ git add -A
aluras-Mac-mini:casadocodigo alura$ git commit -m "Inital Commit"
```

Pronto! Nosso projeto agora está incluso dentro de um repositório *git*, o qual será reconhecido posteriormente pelo *Heroku*. O que precisamos fazer agora é criar a aplicação dentro do *Heroku*. Veja o comando abaixo:

```
heroku apps:create cdcspringmvc-alura
```

Observação: Não utilize o mesmo nome para a aplicação. Procure fazer uma variação utilizando outro nome ao invés de `-alura` . Como este nome é publico, pode-se ter problemas com aplicações com o mesmo nome.

Ao executar o comando anterior, caso não tenha se autenticado ainda, o *Heroku* irá solicitar usuário e senha.

O comando anterior irá criar a aplicação dentro do servidor do *Heroku* e relacionará esta aplicação criada com o nosso repositório local e após isso irá exibir uma mensagem de sucesso.

```
aluras-Mac-mini:casadocodigo alura$ heroku apps:create cdcspringmvc-alura
Creating cdcspringmvc-alura... done, stack is cedar-14
https://cdcspringmvc-alura.herokuapp.com/ | https://git.heroku.com/cdcspringmvc
-alura.git
Git remote heroku added
```

Com esta etapa pronta, precisaremos fazer alguns ajustes em nossa aplicação para que esta possa ser executada corretamente dentro do *Heroku*. A primeira delas é a adição de um plugin de um projeto *Maven* configurado que irá fazer com que a nossa aplicação seja executada como um *jar*. Diferente do que acontece quando a aplicação é executada dentro do *Tomcat*.

Na sessão de `plugins` no arquivo `pom.xml` adicione o plugin *WebApp Runner*:

```
<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>
```

A próxima adaptação se trata a respeito do banco de dados. O *Heroku* não tem o *MySQL* em suas opções de banco de dados. Porém, este tem o *Postgres* um outro banco de dados *SQL* muito conhecido. Por este motivo, adicionaremos como dependência do nosso projeto, a biblioteca do *Postgres*:

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

Mas para que possamos usar o *Postgres* em nossa aplicação, precisaremos criar um *Data Source* para este. Atualmente nas configurações de banco de dados, só temos o *Data Source* para o *MySQL* configurado. Então faremos um outro para o *Postgres*.

O primeiro passo para estas configurações é isolar as definições das propriedades de acesso ao banco de dados em *profiles* diferentes. Em `dev` utilizaremos o *MySQL* e em produção o *Postgres*. Neste caso, na classe `JPAConfiguration` faremos as seguintes alterações:

1. O método `entityManagerFactory` precisa receber as propriedades do banco de dados por parametro e não por uma chamada direta de método.
2. O método `aditionalProperties` agora precisa ser anotado com `@Bean` e configurado com o `@Profile("dev")`.

O segundo passo serve apenas para que o próprio *Spring* configure o objeto `additionalProperties` recebido pelo método `entityManagerFactory` de acordo com o *Profile* determinado. Neste caso teremos:

O método `entityManagerFactory` recebendo o `additionalProperties` por parâmetro:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource, Proper
    LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFactoryB
  factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models");
  factoryBean.setDataSource(dataSource);

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

  factoryBean.setJpaProperties(additionalProperties);

  return factoryBean;
}
```

E o método `additionalProperties` configurado para ser executado automaticamente pelo *Spring*:

```
@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;
}
```

Para que as configurações de desenvolvimento e produção da aplicação não fiquem misturadas. Criaremos uma nova classe no pacote de configurações chamada `JPAProductionConfiguration` e dentro dela copiaremos os métodos `dataSource` e `additionalProperties` da classe `JPAConfiguration`.

Algumas diferenças são claras:1. Não anotaremos cada um dos método com o `Profile("prod")` mas sim a classe, pois está se trata de uma classe especifica de produção. 2. O Dialeto do banco de dados muda de `MySQL5Dialect` para `PostgreSQLDialect`. 3. O *driver* de conexão também muda de `com.mysql.jdbc.Driver` para `org.postgresql.Driver`.

A classe `JPAProductionConfiguration` até então deverá estar assim:

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

  @Bean
  public DataSource dataSource(){
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
```

```
        dataSource.setDriverClassName("org.postgresql.Driver");

        dataSource.setUsername("root");
        dataSource.setPassword("");
        dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo");

        return dataSource;
    }

    @Bean
    private Properties aditionalProperties(){
        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;
    }

}
```

Uma observação a ser feita é que no *Heroku* não temos o controle sobre os dados de acesso ao banco de dados. Estes dados são disponibilizados através de variaveis de ambiente. Estas podem ser acessadas através de um atributo inicializado pelo próprio *Spring*, chamado de `Environment`.

Através deste objeto `Environment` podemos usar o método `getProperty` que retorna uma propriedade deste objeto. A propriedade que queremos pode ser acessada passando o valor `DATABASE_URL` para este método. Que retornara uma *String* no seguinte formato:

```
usuario:senha@host:port/path
```

A *String* retornada segue um padrão bem conhecido chamado de **URI**. Para que não precisemos manipular *String*s para extrair cada uma das informações requeridas. Criaremos um objeto do tipo *URI* que já nos fornece método mais fáceis de extrair estes dados a partir da *String*. Vejamos o código abaixo:

```
@Autowired
//org.springframework.core.env.Environment
private Environment environment;

@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;
}
```

OBS: Se atente em usar o import que nos é fornecido pelo eclipse

```
import org.springframework.core.env.Environment;
```

Os métodos `getHost`, `getPort` e `getPath` retornam o *host*, a *porta* e o *caminho* para o banco de dados. O *getUserInfo* retorna o usuário e a senha separados por dois pontos. Para capturar o usuário, quebramos a *string* em duas partes (quebra feita nos dois pontos) através do método `split` onde como resultado teremos um *array* no qual, na primeira posição ([0]) teremos o usuário e na segunda posição ([1]) teremos a senha.

Com esta classe finalizada, não podemos esquecer de adiciona-la la lista de classes de configurações a serem carregadas pelo *Spring* na inicialização do projeto no método `getRootConfigClasses` da classe `ServletSpringMVC`.

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

E nesta mesma classe, comentar todo o método `onStartup`. Este sempre será executado e nele está configurado que o *Profile* de *dev* está ativo. Não queremos que este *profile* esteja definido no servidor em produção. Por isso deixaremos este método comentado:

```
//  @Override
//  public void onStartup(ServletContext servletContext) throws ServletException {
//    super.onStartup(servletContext);
//    servletContext.addListener(new RequestContextListener());
//    servletContext.setInitParameter("spring.profiles.active", "dev");
//  }
```

Como último passo de configuração, precisamos criar um arquivo que é requerido pelo *Heroku* para indicar como a nossa aplicação será executada quando esta for inicializada no servidor. Este arquivo deve se chamar `Procfile` e deve conter o seguinte conteúdo:

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

Neste comando, estamos definindo algumas opções para o *Heroku* disponibilizar para nossa aplicação os seguintes parametros: O *profile* ativo será o de `prod` o alvo de execução será o `webapp-runner.jar` que configuramos para o nosso projeto. A porta de acesso a aplicação será determinada pelo próprio *Heroku* e este deve executar qualquer `war` encontrado no servidor.

Para finalmente publicarmos a aplicação no servidor, devemos agora, adicionar as modificações feitas no repositório. Comita-las e então fazer o `push` da mesma para o servidor através dos seguintes comandos.

```
git add .
git commit -m "configurações de produção"
git push heroku master
```

Como resultado, teremos o carregamento de diversas bibliotecas no servidor. O *upload* da aplicação e também a mensagem de sucesso ao final de tudo com a *URL* de acesso da mesma.

Inicializando a compressão e envio da aplicação para o servidor:

```
Delta compression using up to 4 threads.
Compressing objects: 100% (150/150), done.
Writing objects: 100% (194/194), 323.28 KiB | 0 bytes/s, done.
Total 194 (delta 27), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Removing .DS_Store files
remote: -----> Java app detected
remote: -----> Installing OpenJDK 1.8... done
remote: -----> Installing Maven 3.3.3... done
remote: -----> Executing: mvn -B -DskipTests=true clean install
remote:        [INFO] Scanning for projects...
remote:        [INFO]
```

Mensagem de sucesso com *URL* de acesso a aplicação:

```
----------------
remote:        [INFO] BUILD SUCCESS
remote:        [INFO] ------------------------------------------------
----------------
remote:        [INFO] Total time: 28.672 s
remote:        [INFO] Finished at: 2015-09-03T17:13:47+00:00
remote:        [INFO] Final Memory: 28M/513M
remote:        [INFO] ------------------------------------------------
----------------
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing... done, 95.6MB
remote: -----> Launching... done, v5
remote:        https://cdcspringmvc-alura.herokuapp.com/ deployed t
```

Sucesso! A aplicação pode ser acessada através do endereço [cdcspringmvc-alura.herokuapp.com (http://cdcspringmvc-alura.herokuapp.com)](http://cdcspringmvc-alura.herokuapp.com). O primeiro carregamento pode demorar bastante, é normal.

A página da nossa aplicação será exibida sem nenhum produto cadastrado. Podemos cadastrar um novo acessando o caminho `/produtos/form` mas somos direcionados a página de autenticação. Não temos usuário cadastrado, é um outro banco agora, limpo, sem nenhum registro.