

01

Subindo o projeto

Transcrição

Seja bem-vindo ao curso de CDI da Alura.

Neste curso vamos ver alguns recursos bem peculiares e alguns um pouco mais avançados.

do CDI.

Na plataforma da Alura, temos um curso introdutório de [JSF \(<https://cursos.alura.com.br/course/jsf-cdi>\)](https://cursos.alura.com.br/course/jsf-cdi) no qual abordamos como utilizar o CDI.

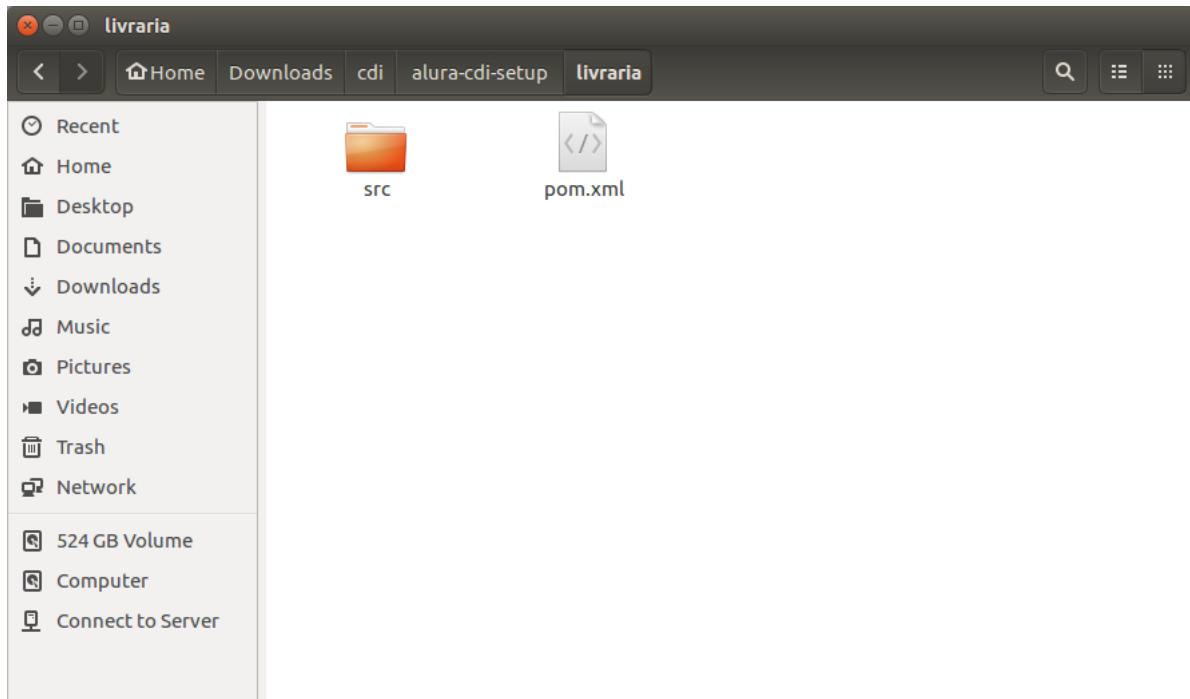
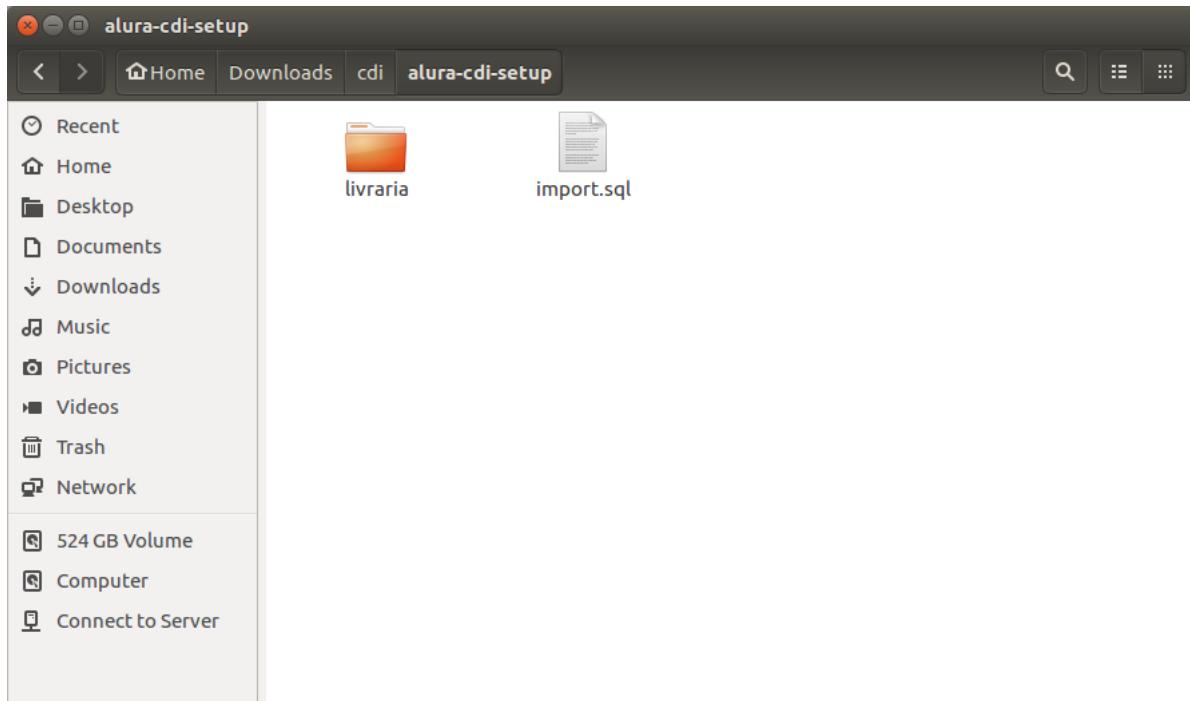
Nós iremos utilizar como base o projeto do JSF com CDI. Nossa objetivo é configurar o projeto para utilizar o CDI e, em seguida, criar uma biblioteca construída por nós, e que poderá ser reutilizada em outros projetos que utilizem a plataforma Java EE.

Apesar de utilizar como base o projeto de um outro curso, o projeto atual está um pouco diferente. Agora estamos utilizando o [Maven \(<https://maven.apache.org/>\)](https://maven.apache.org/) para fazer o gerenciamento do `_build_`. Em vez de adicionar de forma manual todos os `jars` dentro do `_classpath_`, vamos utilizar o Maven para baixar os `jars` e adicioná-los dentro do `_classpath_`.

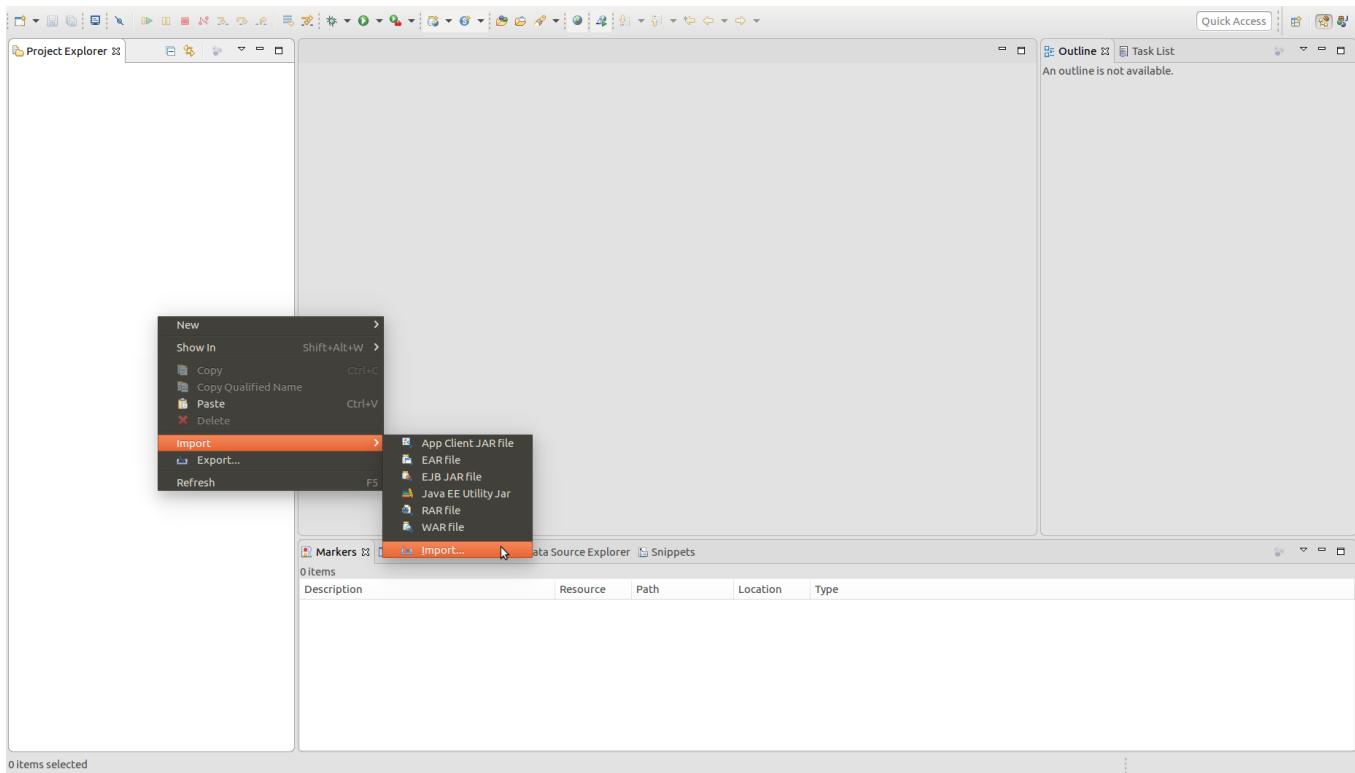
Se você não conhece o Maven, sugerimos que você faça antes nosso [curso de Maven \(<https://cursos.alura.com.br/course/maven-build-do-zero-a-web>\)](https://cursos.alura.com.br/course/maven-build-do-zero-a-web). E voltar depois para este curso, já com um entendimento do que é o Maven e o que a ferramenta nos possibilita fazer.

A mesma recomendação serve para o JSF. Vamos falar de algumas coisas específicas do JSF, utilizando o CDI, mas é interessante que você já conheça um pouco do JSF.

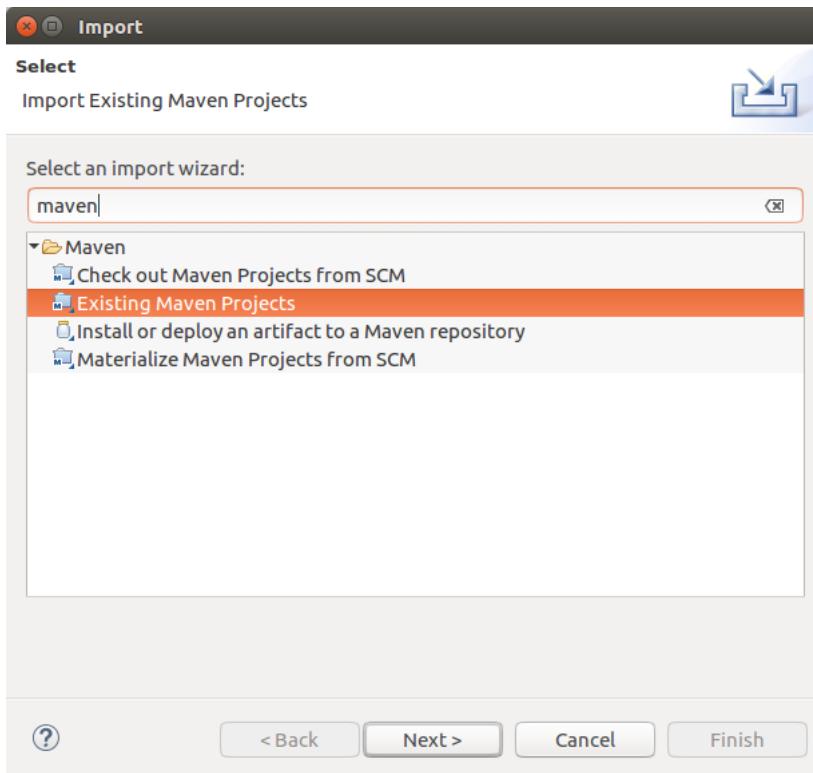
Vamos começar importando o nosso projeto. Como citado anteriormente, vamos utilizar um projeto Maven. Descompactamos o projeto em algum diretório do sistema, e temos o diretório `livraria`, que dentro dele possui o arquivo `pom.xml` do Maven. Agora podemos simplesmente importar o projeto no Eclipse.



Clique com o botão direito na área branca à esquerda do Eclipse. Em seguida, escolha a opção `Import > Import....`. Você também pode realizar esse mesmo passo usando o menu `File > Import....`.

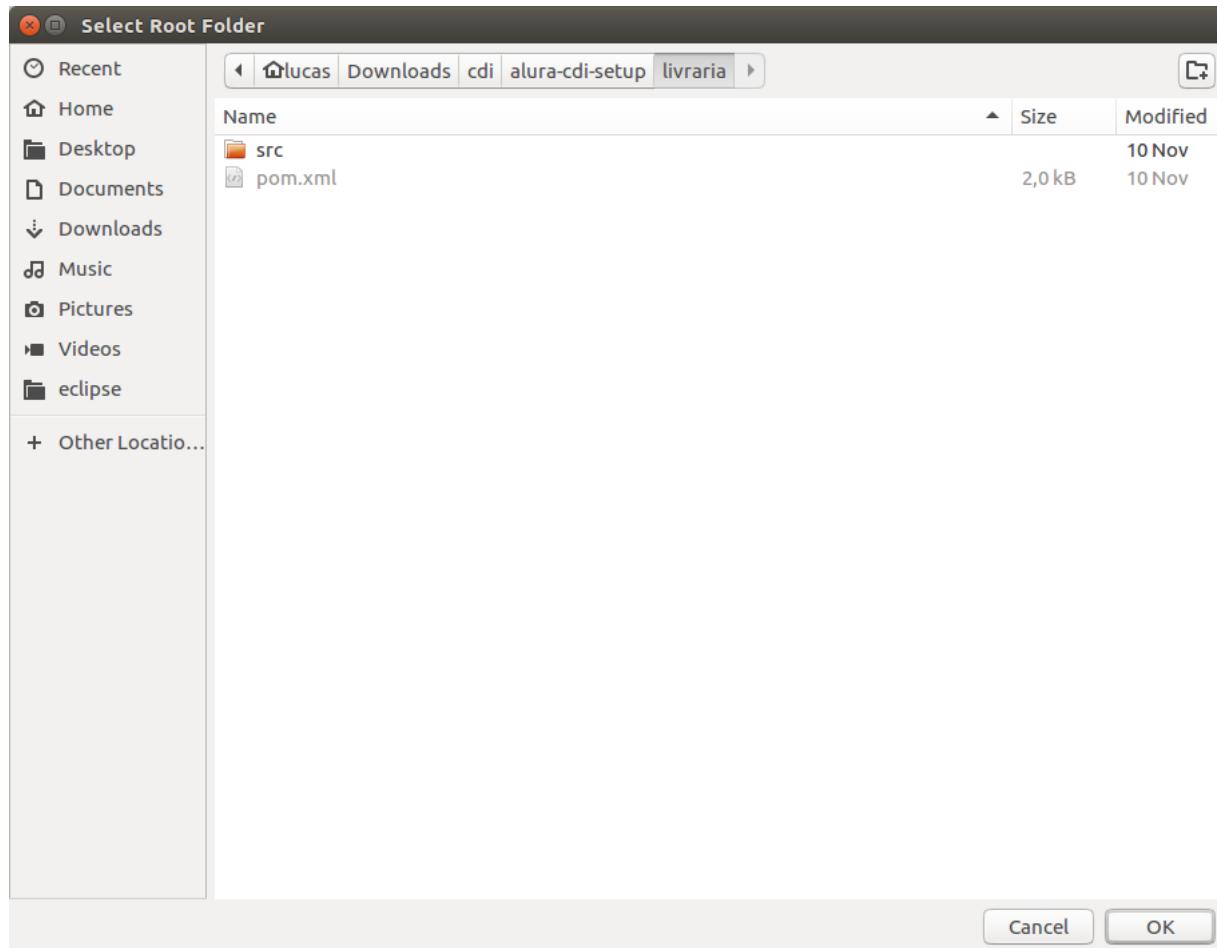
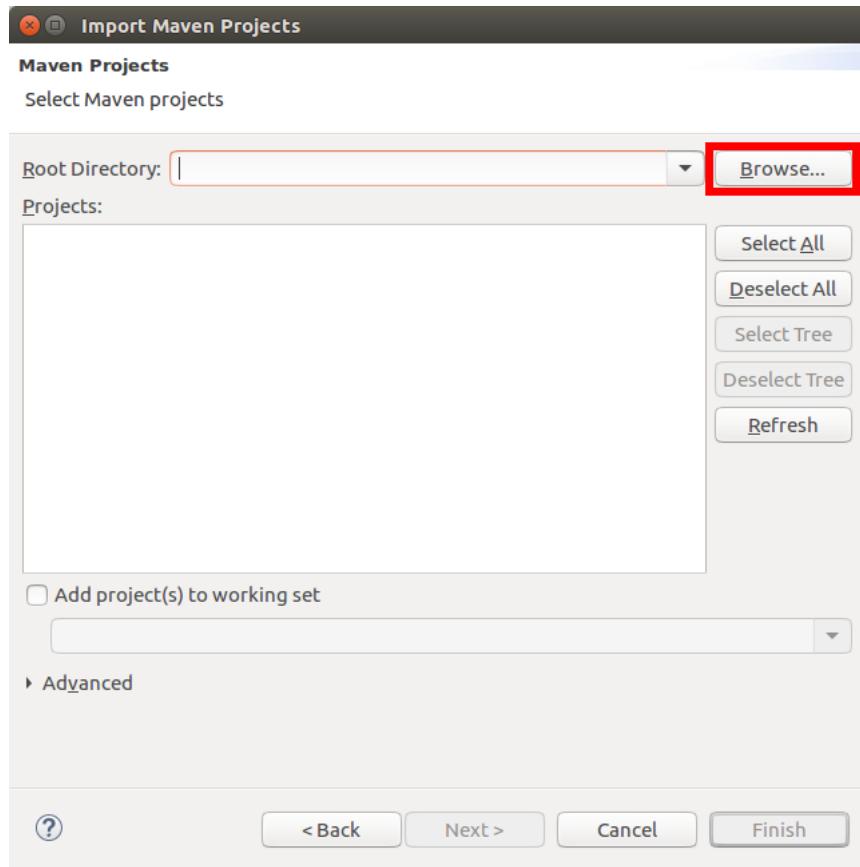


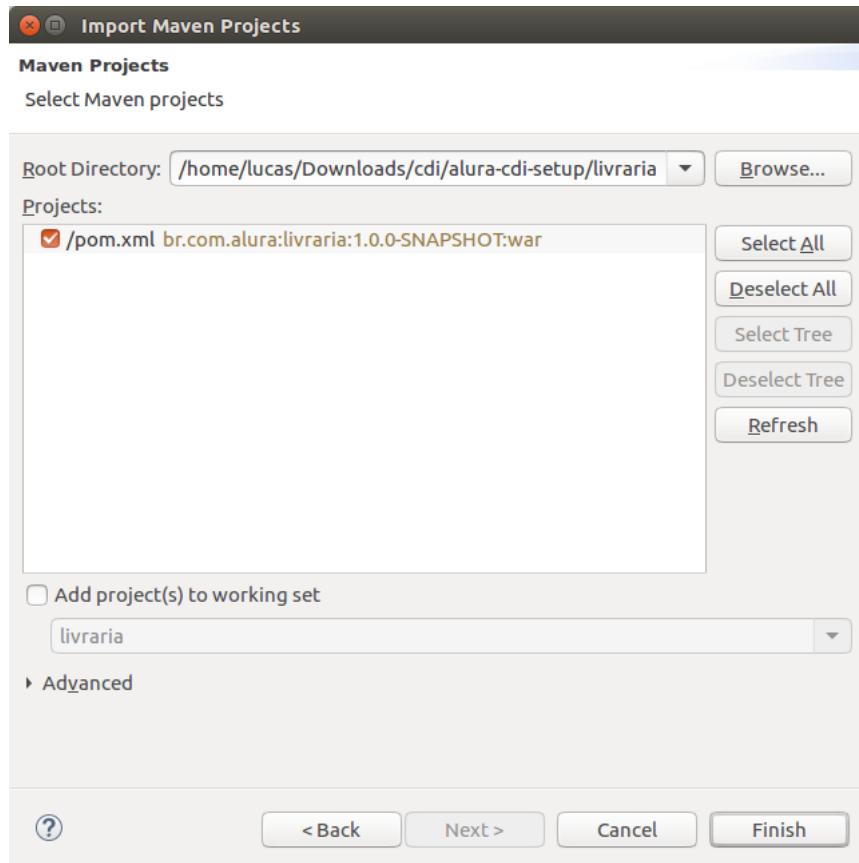
Na próxima tela, escolha a opção "Existing Maven Project" - você pode pesquisar pela palavra "maven" na caixa de busca, assim será mais fácil encontrar a opção.



Na próxima tela, clique em "Browse..." para selecionar o diretório onde o projeto foi descompactado. Selecione o diretório `livraria`.

Perceba que ao selecionar uma pasta que possui o projeto Maven, o Eclipse lista o arquivo `pom.xml` em "Projects". Para finalizar clique em "Finish".

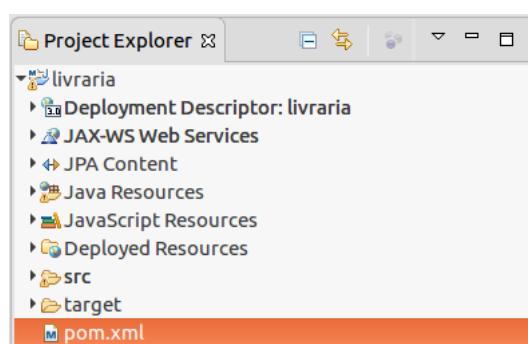




A primeira vez que você importar o projeto, talvez tenha que esperar um pouco, pois o Maven baixará todas as dependências do repositório central. Depois que ele fizer isto pela primeira vez, é feito um "cache" local, que sempre reutilizará as dependências locais que já foram baixadas.

Entendendo as dependências do projeto

Após a finalização da importação do projeto, vamos dar uma olhada nas dependências que o nosso projeto possui, abrindo o arquivo `pom.xml`.



No seguinte trecho de código, é possível ver que estamos utilizando a versão 8 do Java.

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Na seguinte linha, que é possível encontrar logo no início do arquivo, indica que será gerado um pacote `.war`.

```
<packaging>war</packaging>
```

O projeto utiliza o Hibernate. Em seguida temos duas dependências que são referentes ao JSF - já que este é utilizado pela aplicação.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.1.8.Final</version>
</dependency>

<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.13</version>
</dependency>

<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.13</version>
</dependency>
```

A próxima dependência é o driver do MySQL, para que seja possível se conectar com o banco de dados.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.22</version>
</dependency>
```

A dependência do `hibernate-validator` está inclusa porque o projeto utiliza algumas coisas da Bean Validation.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.3.0.Final</version>
</dependency>
```

E por fim, duas dependências referentes ao [PrimeFaces \(http://www.primefaces.org/\)](http://www.primefaces.org/): O PrimeFaces em si e seus temas.

```
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>all-themes</artifactId>
    <version>1.0.10</version>
</dependency>

<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
```

```
<version>5.3</version>
</dependency>
```

O PrimeFaces pede que seja adicionado seu repositório para obter a dependência do tema. Portanto, no seguinte trecho de código, temos adicionada a configuração do repositório.

```
<repositories>
  <repository>
    <id>prime-repo</id>
    <name>PrimeFaces Maven Repository</name>
    <url>http://repository.primefaces.org</url>
    <layout>default</layout>
  </repository>
</repositories>
```

Para finalizar, último a tag `finalName` define o nome do artefato gerado, que nesse caso será `livraria.war`. E foi também adicionado um *plugin* do Maven para gerar os nossos `wars`.

```
<build>
  <finalName>livraria</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.6</version>
    </plugin>
  </plugins>
</build>
```

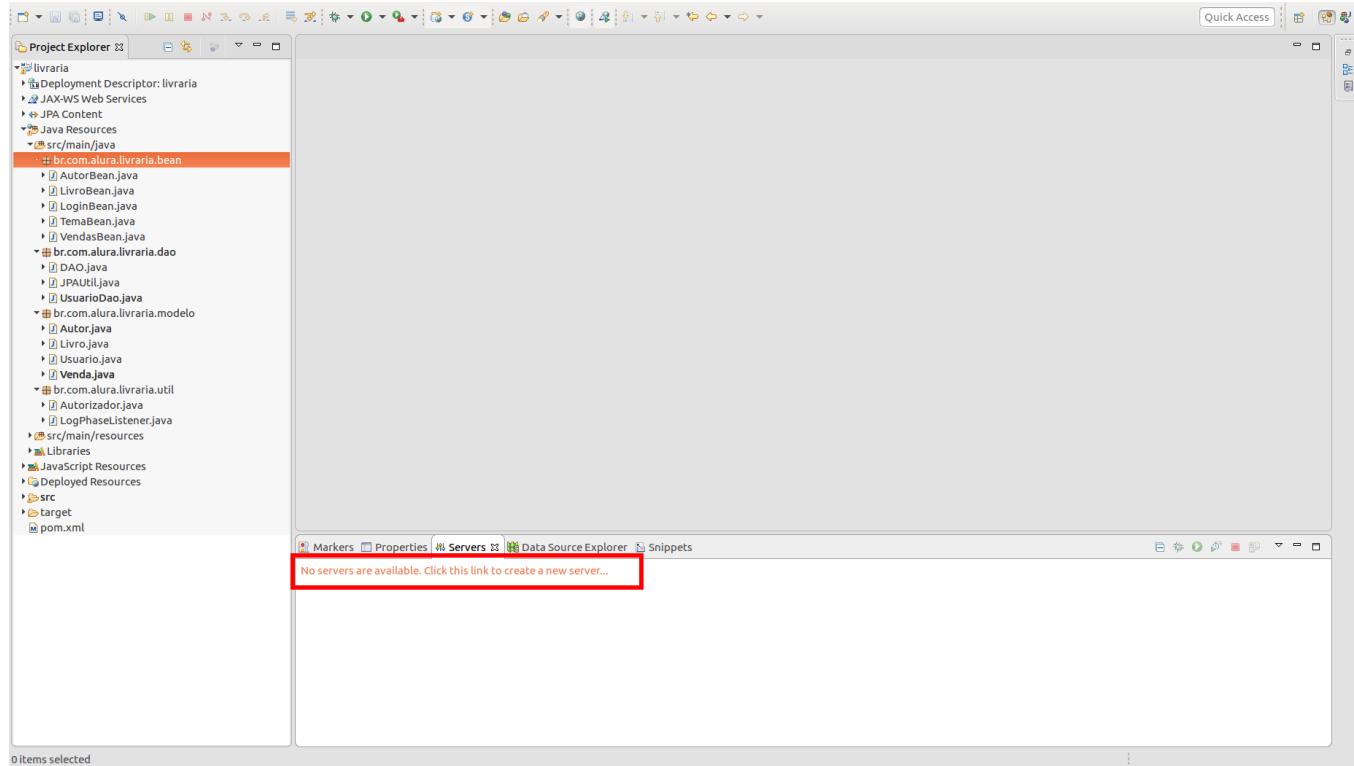
Ao expandir a pasta `Java Resources > src/main/java`, podemos ver os pacotes e classes da aplicação. Os beans estão relacionados com nossas `views`, os arquivos `.xhtml`. Temos também os DAOs, os modelos e classes utilitárias.



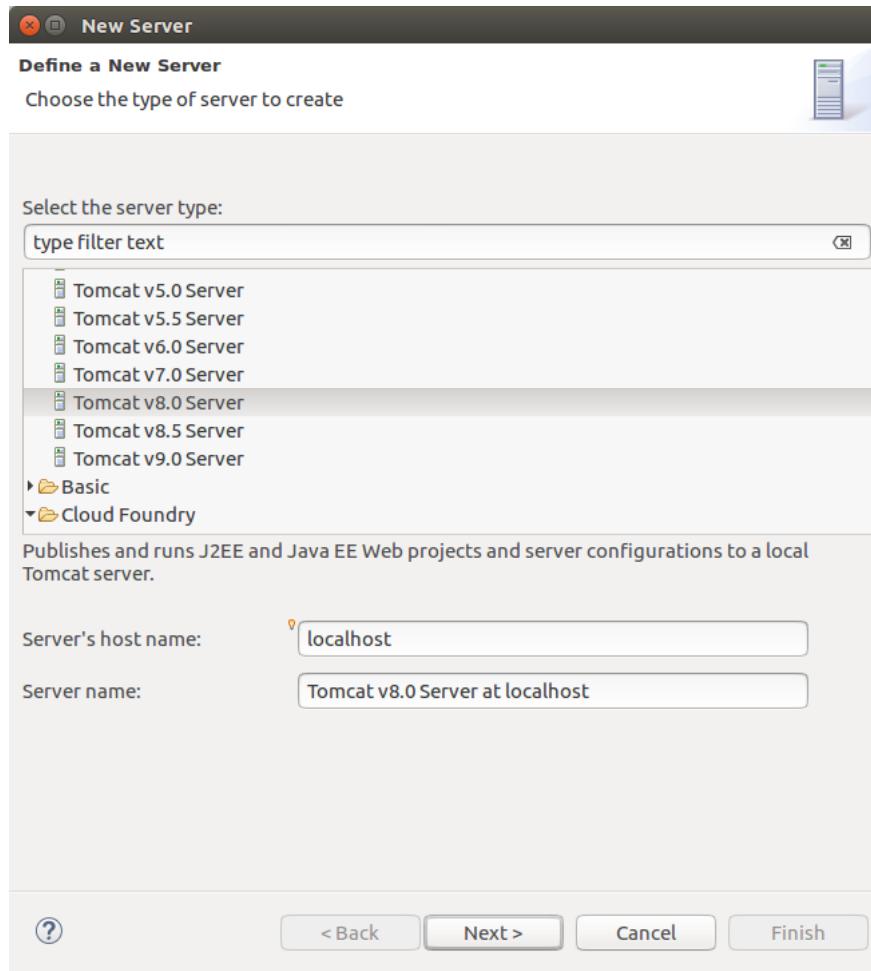
Configurando o Servlet Container

Agora que conseguimos importar nossas aplicação, vamos precisar de alguém para rodar a aplicação Web. Nesse momento não vamos utilizar um servidor de aplicação, mas um Servlet Container, o [Tomcat \(http://tomcat.apache.org/\)](http://tomcat.apache.org/). Após fazer o Download do Tomcat, descompacte em um diretório de sua preferência.

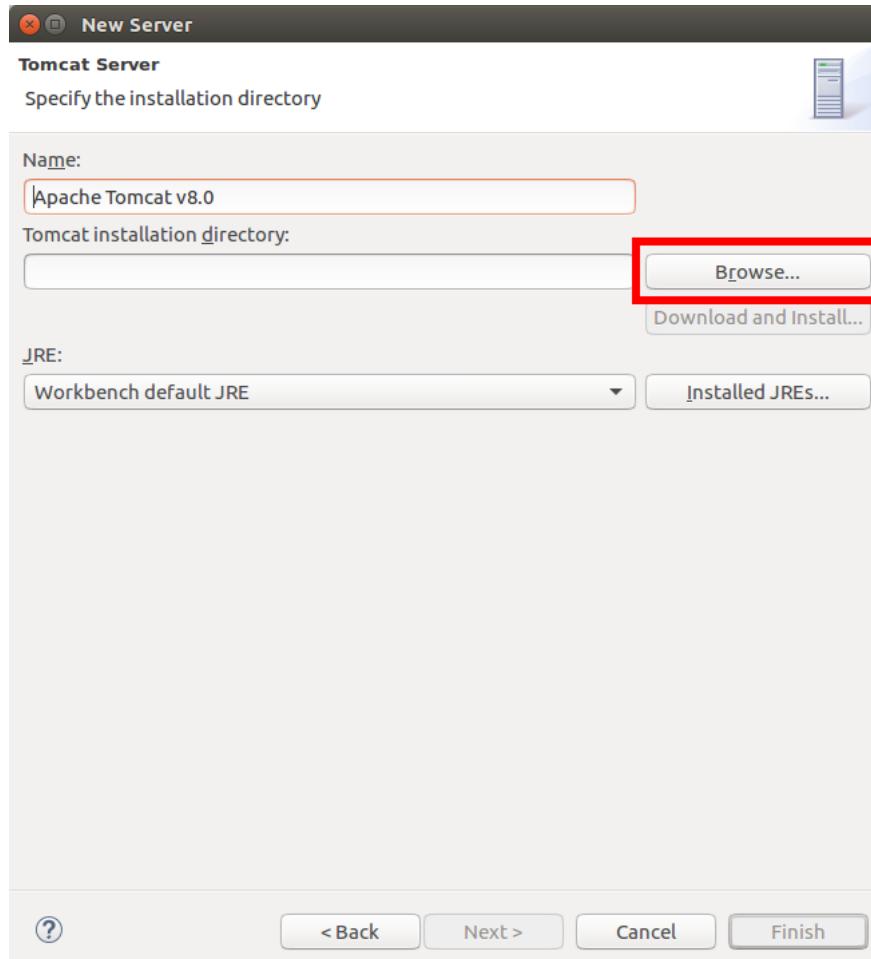
No Eclipse, na aba Servers, clique na mensagem "No servers are available. Click this link to create a new server...".



Na tela que irá parecer, Selecione Apache > Tomcat v8.0 Server e clique em "Next".

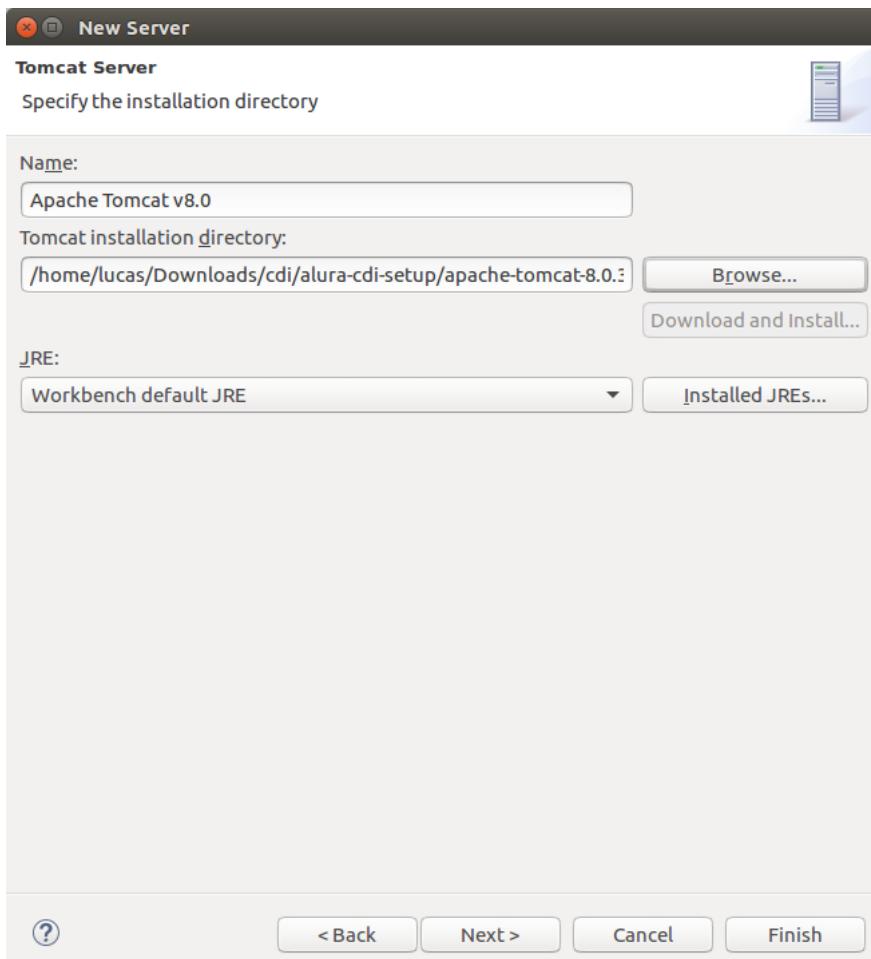


Na próxima tela, clique em "Browser" e selecione o diretório do Tomcat. Feito isso, clique em "Finish".



A screenshot of a file explorer window. The left sidebar shows recent locations like Home, Desktop, and Downloads. The main area displays the contents of the 'apache-tomcat-8.0.39' directory. The table lists the following files and folders:

Name	Size	Modified
bin	9 Nov	9 Nov
conf	9 Nov	9 Nov
lib	9 Nov	9 Nov
logs	9 Nov	9 Nov
temp	9 Nov	9 Nov
webapps	9 Nov	9 Nov
work	9 Nov	9 Nov
LICENSE	57,0 kB	9 Nov
NOTICE	1,4 kB	9 Nov
RELEASE-NOTES	6,7 kB	9 Nov



Na aba Servers, já é possível ver o Tomcat configurado.



Configurando o banco de dados

Resta um último passo. Nossa aplicação se comunicará com o banco de dados. Se você olhar dentro do diretório `src/main/resources > META-INF`, existe o arquivo `persistence.xml`. O `persistence.xml` é o arquivo de configuração para que a aplicação possa se comunicar com o banco de dados.

Devido a seguinte linha do `persistence.xml`, nossa aplicação espera que exista um banco de dados chamado `livrariadb`. Vamos criar essa banco de dados agora.

```
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/livrariadb" />
```

Segundo nossa configuração, o banco de dados MySQL não possui senha:

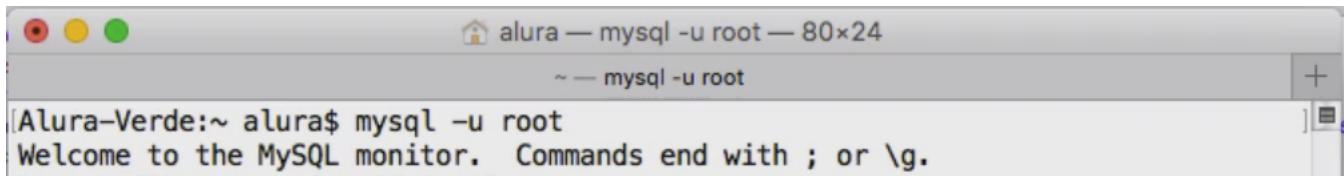
```
<property name="javax.persistence.jdbc.password" value="" />
```

Vamos nos conectar a servidor do MySQL:

```
$ mysql -u root
```

E em seguida, já conectados, vamos criar o banco de dados:

```
> create database livrariadb;
```



A screenshot of a terminal window titled "alura — mysql -u root — 80x24". The window shows the MySQL monitor welcome message: "[Alura-Verde:~ alura\$ mysql -u root Welcome to the MySQL monitor. Commands end with ; or \g.".

Ao acessarmos o banco, e listarmos as tabelas, vamos ver que não existe nenhuma tabela porque acabamos de criar o banco.

```
> use livrariadb;  
Database changed  
  
> show tables;  
Empty set (0,00 sec)
```

Para iniciarmos com alguns dados e não seja preciso que esses dados tenham que ser cadastrados manualmente, vamos importar um *script*. Saia do `mysql` utilizando o comando `exit`.

```
> exit
```

Vamos utilizar o comando `cd` e navegar até o diretório onde se encontra o arquivo `import.sql`. Dentro desse arquivo existem vários comandos SQL que desejamos que o MySQL execute.



The screenshot shows a terminal window with the title "setup — bash — 80x24". The window contains the SQL script "import.sql". The script starts with creating a table "Autor" with columns "id", "email", and "nome", and ends with locking and unlocking tables.

```
$ cat import.sql

DROP TABLE IF EXISTS `Autor`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `Autor` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `email` varchar(255) DEFAULT NULL,
  `nome` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=14 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

-- 
-- Dumping data for table `Autor`
-- 

LOCK TABLES `Autor` WRITE;
/*!40000 ALTER TABLE `Autor` DISABLE KEYS */;
INSERT INTO `Autor` VALUES (1,'sergio.lopes@caelum.com.br','Sergio Lopes'),(2,'nico.steppat@caelum.com.br');
/*!40000 ALTER TABLE `Autor` ENABLE KEYS */;
UNLOCK TABLES;

/* restante do arquivo... */
```

Queremos que o *script* seja executado dentro do banco de dados `livrariadb`. Para importar, utilizamos o comando `mysql`, fornecendo os dados de login, o nome do banco no qual desejamos executar o *script* e o caminho para o *script*.

```
$ mysql -u root livrariadb < import.sql
```

Após digitar o comando, pressione "Enter". Pode demorar alguns segundos. Vamos logar novamente no MySQL para conferir se os dados foram importados.

Vamos utilizar o comando `use livrariadb` para selecionar o banco de dados e em seguida o comando `show tables` para listar as tabelas.

```
$ mysql -u root livrariadb

> use livrariadb;

> show tables;
+-----+
| Tables_in_livrariadb |
+-----+
| Autor                |
| Livro                |
| Livro_Autor          |
| Usuario              |
+-----+
4 rows in set (0,00 sec)
```

Agora vamos fazer um *querie* para verificar se foi inserido o usuário `admin` com a senha `12345` na tabela `Usuario`.

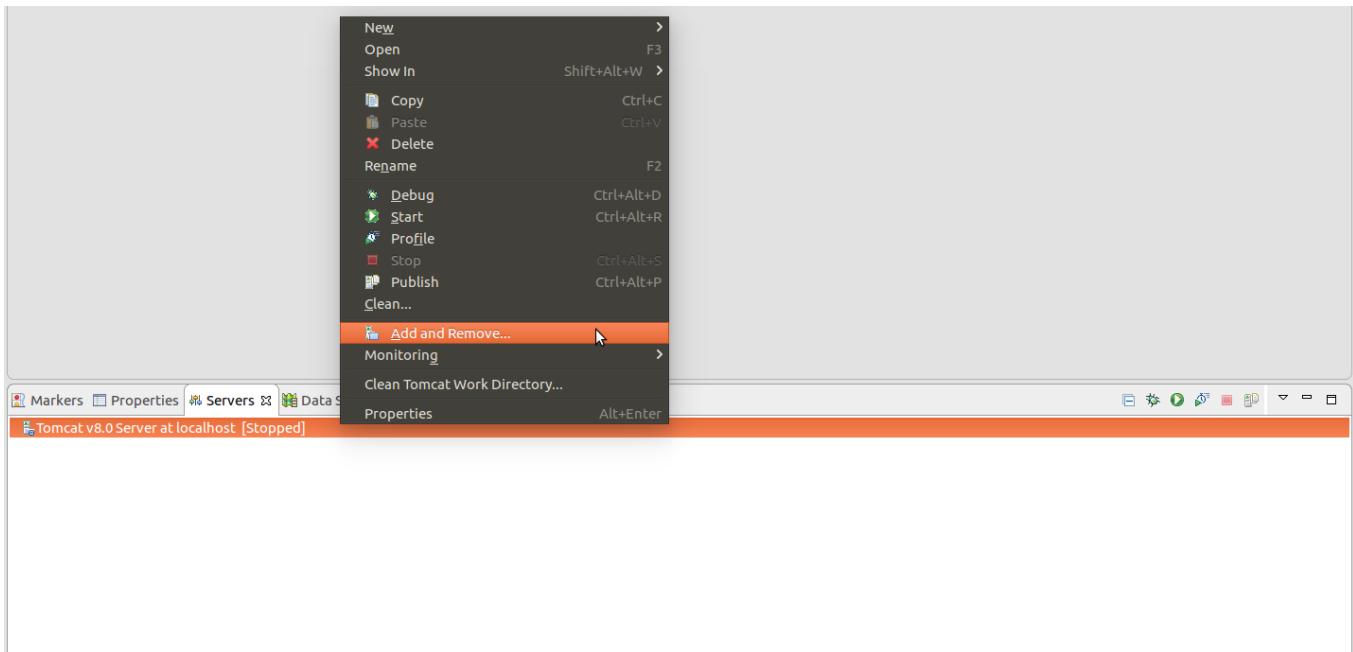
```
> select * from Usuario;
+----+-----+-----+
| id | email           | senha |
+----+-----+-----+
| 1  | admin@caelum.com.br | 12345 |
+----+-----+-----+
1 row in set (0,00 sec)
```

Pronto! Podemos ver que um registro já foi inserido. Temos nosso banco de dados totalmente populado. Desta forma, não precisamos ficar nos preocupando em cadastrar as informações manualmente.

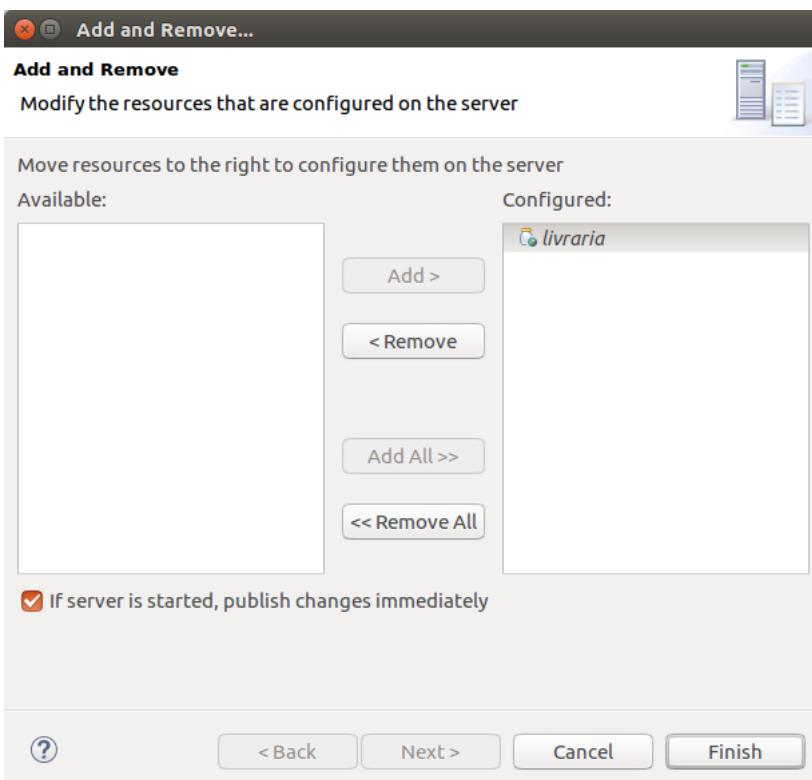
Fazendo deploy do projeto

Agora que configuramos o Tomcat e banco de dados, vamos subir nossa aplicação.

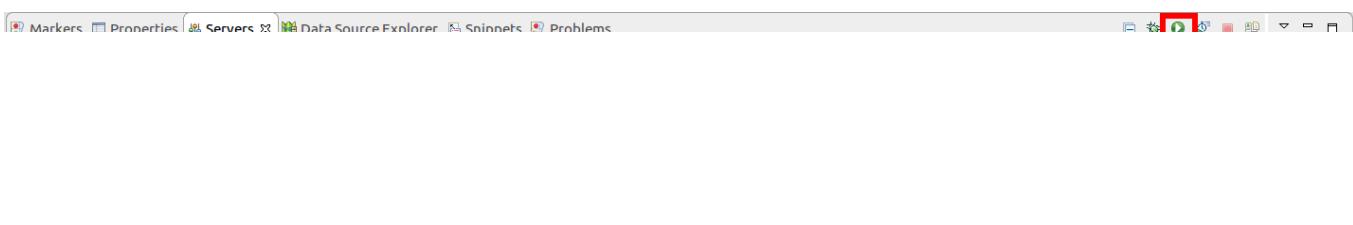
No Eclipse, na aba Servers, clique com o botão direito no Tomcat e escolha a opção "Add and Remove".



Na tela que irá aparecer, selecione o projeto e clique no botão "Add". Em seguida, clique em "Finish".



Clique no botão destacado na imagem seguinte, para iniciar o Tomcat.



Não recebemos nenhuma mensagem de erro no Console, então aparentemente, tudo ok.

```

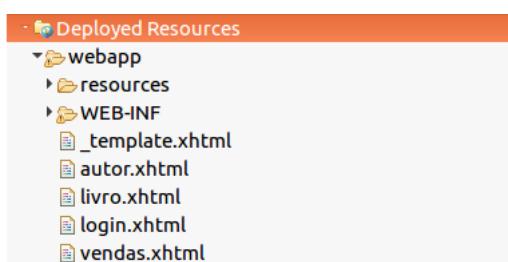
Tomcat v8.0 Server at localhost [Apache Tomcat]/usr/lib/jvm/java-8-oracle/bin/java (2 de dez de 2016 10:27:53)
dez 02, 2016 10:27:58 AM com.sun.faces.config.ConfigureListener contextInitialized
INFO: Initializing Mojarra 2.2.13 ( 20160203-1910 unable to get svr info) for context '/livraria'
dez 02, 2016 10:27:58 AM com.sun.faces.spi.InjectionProviderFactory createInstance
INFO: JSF1048: PostConstruct/PreDestroy annotations present. ManagedBeans methods marked with these annotations will have said annotations processed.
dez 02, 2016 10:27:59 AM org.hibernate.validator.internal.util.Version <clinit>
INFO: HV000001: Hibernate Validator 4.3.0.Final
dez 02, 2016 10:28:00 AM org.primefaces.webapp.PostConstructApplicationEventListener processEvent
INFO: Running on PrimeFaces 5.3
dez 02, 2016 10:28:00 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]
dez 02, 2016 10:28:00 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-nio-8009"]
dez 02, 2016 10:28:00 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 4233 ms

```

Para conferir a aplicação, basta acessar <http://localhost:8080/livraria/login.xhtml> (<http://localhost:8080/livraria/login.xhtml>) no seu navegador.



Para verificar de onde vem o Login, se você expandir a pasta Deployed Resources > webapp , poderá ver que lá se encontram os arquivos que podemos acessar.



Se você tentar acessar [a página dos livros \(http://localhost:8080/livraria/livro.xhtml\)](http://localhost:8080/livraria/livro.xhtml), será direcionado para a página de login. Existe uma regra na classe Autorizador (pacote br.com.alura.livraria.util) que indica quando você não possui um usuário logado, então, a aplicação redirecionará para a página de login.

Você pode efetuar o login utilizando os dados que buscamos anteriormente no banco:

usuário: admin@caelum.com.br , senha: 12345 .

Após o login temos acesso à página de livros.