

Sistemas Distribuídos

DESENVOLVIMENTO DE PROTÓTIPO DE UMA APLICAÇÃO
BASEADA EM JAVA APPLICATION SERVER
MILEIDE LENCASTRE GOMES

ÍNDICE

Introdução.....	2
Configuração WildFly no Eclipse	2
Configuração de <i>DataSource</i> no <i>WildFly</i>	5
Criação da Aplicação com EJB.....	6
Arquitetura do Projeto.....	8
Criação da Base de Dados no MySql	8
Configuração de JPA	9
JPA Entities.....	9
Criação de Stateless EJB.....	10
Data Transfere Objects DTO	11
Criação de JSF.....	13
Web Application.....	14
Página Principal.....	14
Pacientes.....	14
Consultas.....	14
Médicos.....	14
Conclusão.....	15
Bibliografia	16

Introdução

O presente relatório técnico é relativo ao trabalho final da unidade curricular de sistemas distribuídos, em que o objetivo é implementar um pequeno protótipo de uma aplicação baseada em *java Application Server*.

Deste modo, o tema escolhido para o desenvolvimento do trabalho foi a criação de uma aplicação onde fosse possível observar os médicos e as respetivas especialidades, de uma determinada clínica, e em função das necessidades do utilizador, este poderia marcar uma consulta. No entanto, é importante referir que, tratando-se apenas de um protótipo, a aplicação em questão não está totalmente operacional, no que diz respeito a todas as especificações que está deveria cumprir.

O protótipo da aplicação foi desenvolvido na especificação *Jakarta EE : Enterprise Java Beans (EJB)*, e o servidor utilizado foi o WildflyServer.

Configuração WildFly no Eclipse

Inicialmente, o servidor que fora escolhido para o desenvolvimento deste processo foi o *GlassFish*. Entretanto, este servidor apresentou diversas complicações durante o processo de configuração no IDE *Eclipse*. Por este motivo, escolheu-se, na lista dos diversos *Java Application Servers* existentes e disponíveis, o *Wildfly*.

Os passos para a configuração deste servidor no *Eclipse* não difere da configuração do *GlassFish*. Para este processo foram feitos os seguintes procedimentos:

1. Foi feito o download através do URL : <https://wildfly.org/downloads/>

2. No *Eclipse*, na opção *Show View*, escolhe-se a opção *Server* e a seguir *Servers*.

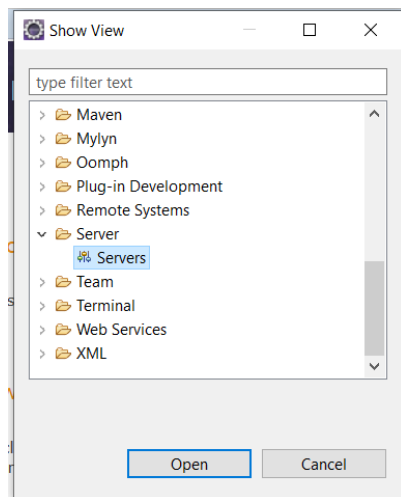


Figura 1- Show View Eclipse

Após estes passos aparece a seguinte janela. E é só carregar em *Next*.

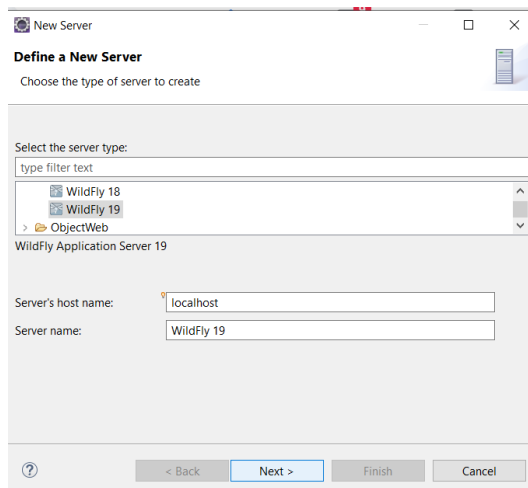


Figura 2 - Adição de novo servidor no eclipse

3. De seguida, é apresentada a próxima imagem, onde é necessário introduzir o *path* do *Wildfly* cujo download foi feito posteriormente.

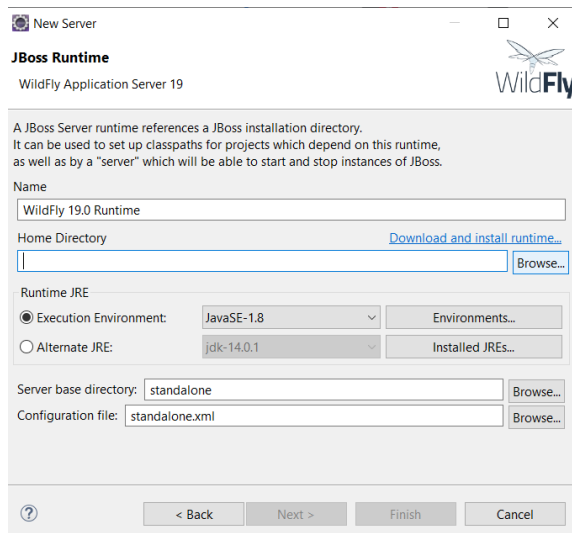


Figura 3 - Escolha do path do servidor

4. Seguidamente, a janela a baixo representada é mostrada no ecrã , onde é necessário apenas carregar em *Finish*.

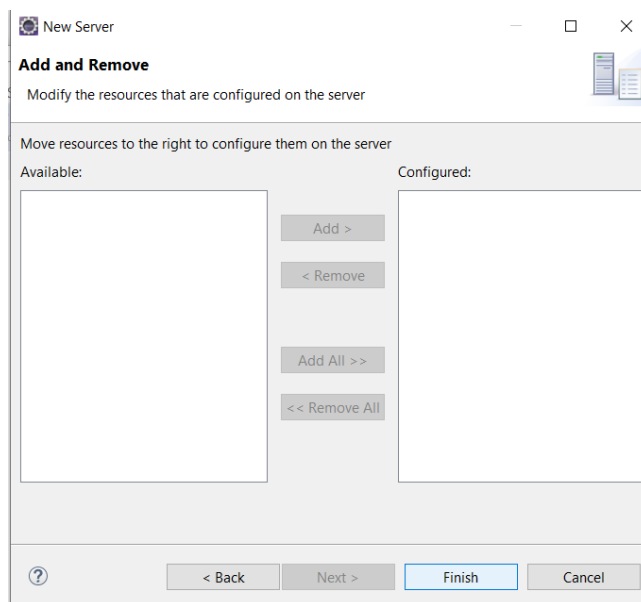


Figura 4 - Ficheiros existentes no servidor

5. Por fim, o último passo é iniciar o servidor.

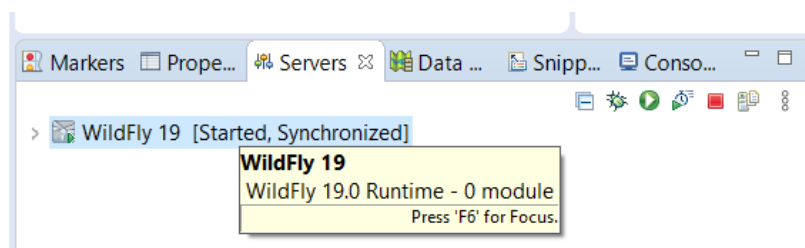


Figura 5 - Inicialização do servidor

Configuração de *DataSource* no *WildFly*

O *Wildfly* Server não é fornecido com o driver JDBC para MySQL. Portanto, foi necessário colocar o arquivo .jar para o MySQLDriver no *path* onde o *Wildfly* pode encontrá-lo. Este foi um processo

Todas as "extensões" do Wildfly estão localizadas em WILDFLY_HOME / modules / system / layers / base. Navegando para esta pasta, encontra-se as pastas do *package*. Essas pastas correspondem ao pacote da extensão. Instalou-se o driver JDBC do MySQL, e para tal, criou-se uma árvore de pastas(*folder tree*) com / mysql. Uma vez lá, criou-se uma outra pasta, chamada "*main*". Nesta pasta, copiou-se o arquivo JDBC JAR. Seguidamente, criou-se um arquivo module.xml com o seguinte conteúdo:

```
1 <module xmlns="urn:jboss:module:1.5" name="com.mysql">
2   <resources>
3     <resource-root path="mysql-connector-java-8.0.12.jar" />
4   </resources>
5   <dependencies>
6     <module name="javax.api"/>
7     <module name="javax.transaction.api"/>
8   </dependencies>
9 </module>
```

Figura 6 - module.xml

Posto isto, isso, a pasta ficou com o seguinte aspecto:

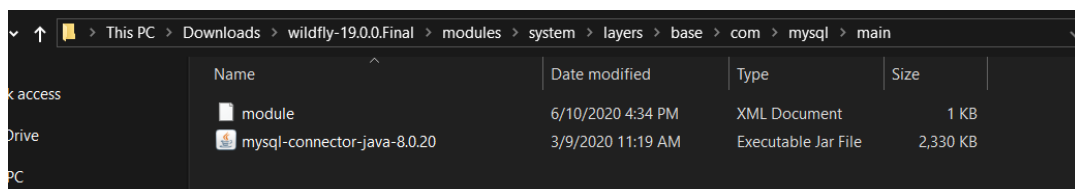


Figura 7 - Diretório onde está o ficheiro xml. e o arquivo jar

O passo seguinte foi informar ao servidor sobre o novo driver JDBC. Abriu-se o arquivo de configuração (WILDFLY_HOME / standalone / configuration / standalone.xml) e navegou-se até a seção "<drivers>". Por fim, adicionou-se o seguinte bloco após a entrada "<driver>" para H2:

```
26 <driver name="mysql" module="com.mysql">
27   <driver-class>com.mysql.cj.jdbc.Driver</driver-class>
28   <xa-datasource-class>com.mysql.cj.jdbc.MySQLXADataSource</xa-datasource-class>
29 </driver>
30
```

Figura 8 - bloco de código adicionado no ficheiro standalone.xml do servidor

A próxima figura, representa os procedimentos no WildFly, para a configuração da nova datasource. É definido o JNDI Name, que será utilizado posteriormente no processo de conexão da aplicação à base de dados, o nome da driver e o url da conexão.

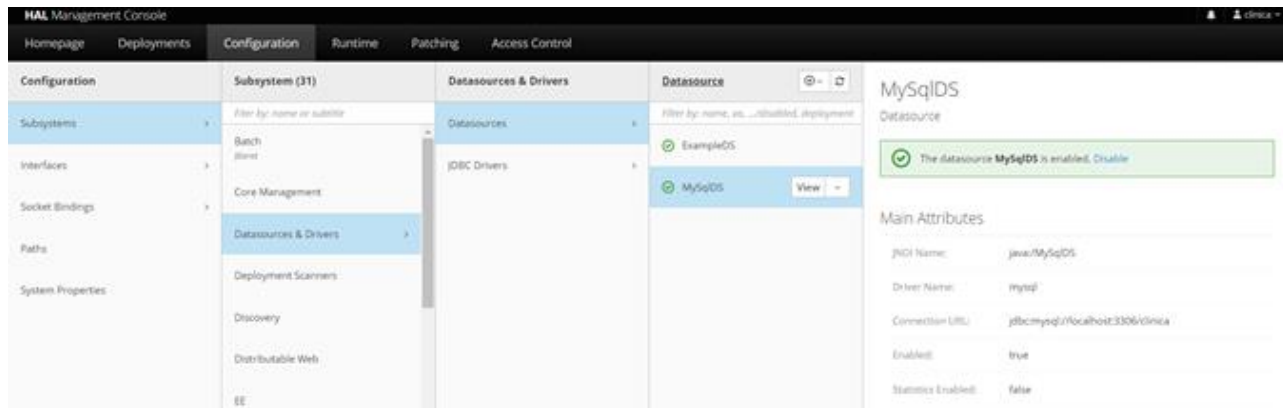


Figura 9 - configuração de uma nova datasource no Wildfly

Criação da Aplicação com EJB

EJBs são empacotados num arquivo JAR. As aplicações Web são empacotados numa Aplicação Web aRchive (WAR). Se os EJBs tiverem que ser acedidos remotamente, o cliente terá que ter acesso às interfaces de negócios(*business interfaces*). Portanto, as *business interfaces* EJB e os objetos compartilhados são empacotados num JAR separado, denominado por JAR do cliente EJB. Além disso, como os EJBs e as aplicações web devem ser implementados como uma única aplicação, estes precisam ser empacotados num EAR.

Deste modo , foram criados 4 diferentes projectos :

- Projeto EJB que cria EJB JAR
- Projeto do cliente EJB que contém classes de negócios e compartilhadas (entre EJB e cliente) classes
- Projeto da Web que gera WAR
- Projeto EAR que gera EAR contendo EJB JAR, cliente EJB JAR e WAR

A seguinte imagem, mostra os projectos a cima referidos.

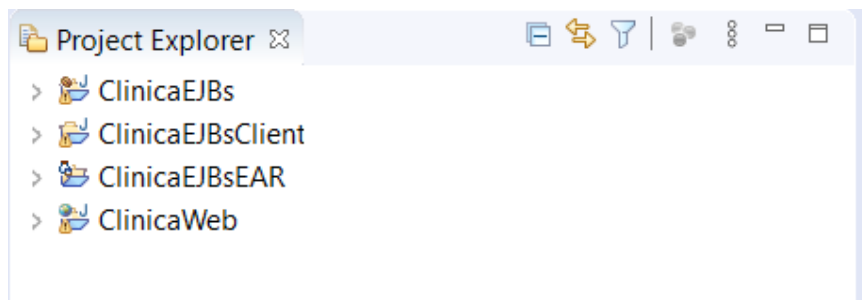


Figura 10 - Projectos Criados para o desenvolvimento da App

O projeto ClinicaEJBs contém as classes de persistência JPA e os Session Beans utilizados que contém a implementação dos Remote Beans presentes no projecto EJBsClient.

O projeto ClinicaEJBsClient contém a interface remota dos Sessions Beans, presentes em ClinicaEJBs, como também os DTOs da aplicação. É importante salientar que, para a criação dos DTOs, é necessário que seja possível aceder às classes de JPA que estão num projeto diferente, isto seria possível através de dependências Maven, no entanto o projeto em questão não sendo Maven foi necessário exportar o projeto ClinicaEJBs como Jar e adicioná-lo ao Class Path do projeto ClinicaEJBsClient.

O projeto ClinicaWeb é o projeto que contém os JSF Beans e as páginas de xhtml que servirão como UI para o utilizador da aplicação. Como os EJB Beans foram criados com tanto Remote Interface como “No interface view” é possível aceder a ambos através dos JSF Beans, para a aplicação em questão foram acedidos através de RemoteBeans, embora tenham sido exploradas ambas as formas de acesso.

Por fim, o projeto enterprise application (EAR): contem arquivos JAR que são compartilhados pelos projetos na Enterprise application. Contem links para todos os projetos na aplicação.

EJBs são empacotados num arquivo JAR. As Web Apps são empacotadas num Web Application archive (WAR). Para que os EJBs possam ser acedidos remotamente, o cliente precisa ter acesso às interfaces de negócios. Portanto, as business interfaces e objetos compartilhados são empacotados num JAR separado, chamado EJBClient JAR. Além disso, se EJBs e WebApps devem ser implementados como uma única aplicação, estes precisam ser empacotados num EAR. Portanto, na maioria dos casos, o aplicativo com EJBs não é um projeto único, mas sim quatro projetos diferentes

Arquitetura do Projeto

A aplicação irá fluir da seguinte maneira: O navegador solicita a página JSF no servidor. A página JSF possui referências aos managed beans JSF que atuam como backing beans para dados nos formulários das páginas JSF. Os beans JSF têm referências aos EJBs que atuam como camada de serviço. Os EJBs na camada Serviço chamam métodos de persistência em entidades JPA. As operações do gestor de entidades mantêm os dados na base de dados. A figura a baixo, apresenta num esquema a arquitetura e fluxo da aplicação:

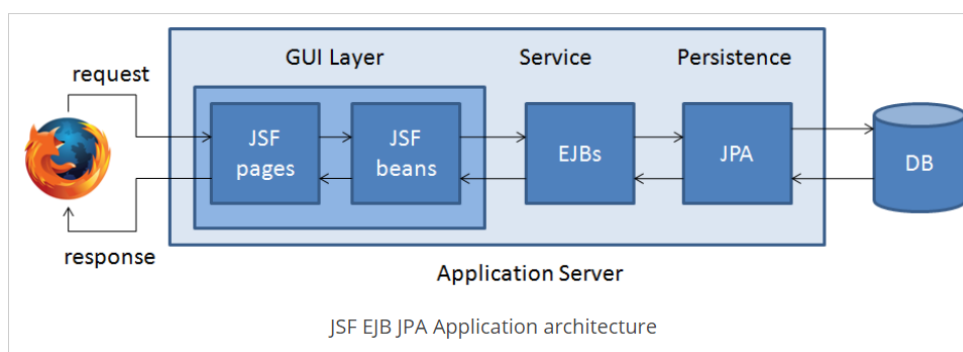


Figura 11 - Arquitetura do Projeto Desenvolvido

Criação da Base de Dados no MySql

A base de dados desenvolvida para este trabalho consiste em 4 entidades: Paciente, Médico, Consulta e Especialidade. O Paciente possui uma relação um para muitos com a Consulta. A entidade Medico possui esta mesma relação com a Consulta, ou seja, um para muitos. E, por fim, a entidade especialidade apresenta uma relação muitos para um com a entidade Medico.

Seguidamente, apresenta-se o modelo de dados criado com as entidades e respectivos atributos:

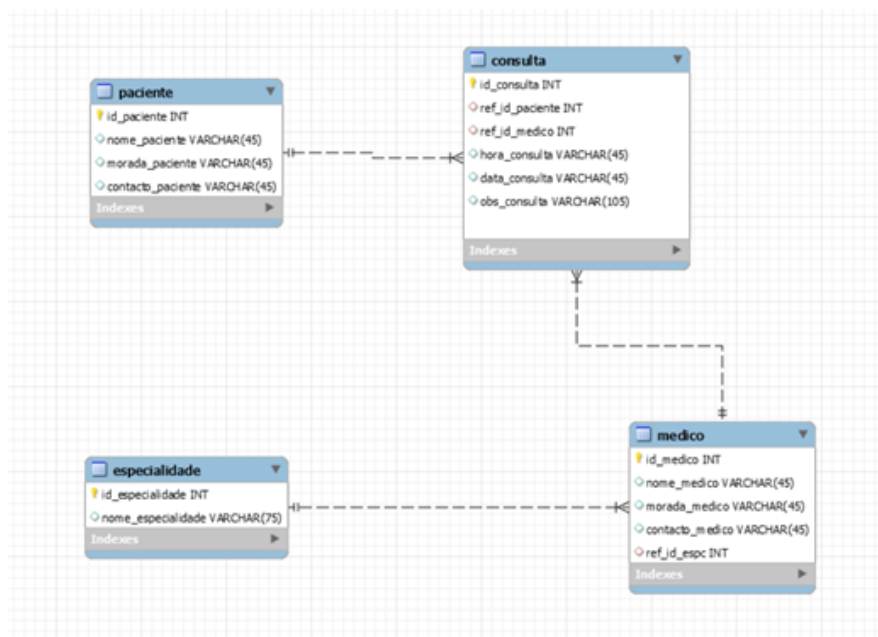


Figura 12 - Modelo da Base de Dados

Configuração de JPA

Foi utilizado o JPA para aceder ao MySQL e estabelecer uma conexão, a qual ficou denominada por ClinicaCONN. Para tal, o projeto ClinicaEJBs foi convertido num projeto JPA. O Eclipse inclui `persistence.xml`, exigido pelo JPA. Teve-se de configurar a fonte de dados JPA em `persistence.xml`. Este processo resumiu-se na definição do tipo de transação como JTA, e na digitação do nome JNDI que foi configurado para a base de dados MySQL.

Esta configuração permite ter acesso automático à todas entidades definidas na base de dados. No entanto, ainda terão de ser criadas as *JPA Entities*.

JPA Entities

A entidade representa uma instância de objeto único que normalmente está relacionada a uma tabela. Qualquer objeto Java antigo simples (POJO) pode ser convertido numa entidade anotando a classe com `@Entity`. Os membros da classe são mapeados para as colunas de uma tabela na base de dados. As classes de entidade são classes Java simples, portanto, podem estender ou incluir outras classes Java ou mesmo outra entidade JPA.

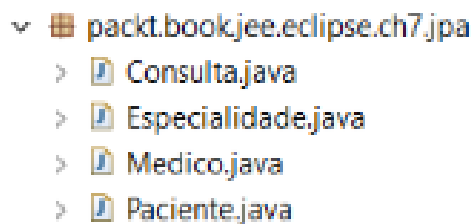
Acedendo a JPA Tools, através do projeto ClinicaEJBs no Eclipse, gera-se as entidades. A figura seguinte demonstra a entidade Paciente :

```

1 package packt.book.jee.eclipse.ch7.jpaa;
2
3 import java.io.Serializable;
4
5 /**
6  * The persistent class for the paciente database table.
7  */
8 @Entity
9 @NamedQuery(name = "Paciente.findAll", query = "SELECT p FROM Paciente p")
10 public class Paciente implements Serializable {
11     private static final long serialVersionUID = 1L;
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     @Column(name = "id_paciente")
16     private int idPaciente;
17
18     @Column(name = "contacto_paciente")
19     private String contactoPaciente;
20
21     @Column(name = "morada_paciente")
22     private String moradaPaciente;
23
24     @Column(name = "nome_paciente")
25     private String nomePaciente;
26
27     @Column(name = "password_user")
28     private String passwordUser;
29
30     private String username;
31
32
33

```

Figura 13 - Entidade Paciente



```

v packt.book.jee.eclipse.ch7.jpaa
  > Consulta.java
  > Especialidade.java
  > Medico.java
  > Paciente.java

```

Figura 14 - Todas Entidades

Criação de Stateless EJB

Neste procedimento, as classes (ex: PacienteBean, ConsultaBean, etc) são criadas com as anotações `@Stateless` e `@Localbean`. Estas classes também implementam a interface `CourseBeanRemote`, definida no projeto `ClinicaEJBClient`, que pode ser acedida por diferentes clientes se tiverem acesso à mesma.

```

1 package packt.book.jee.eclipse.ch7.ejb;
2
3 import java.util.ArrayList;
4
14
15 /**
16  * Session Bean implementation class PacienteBean
17  */
18 @Stateless
19 @LocalBean
20 public class PacienteBean implements PacienteBeanRemote {
21

```

Figura 15 - pacienteBean

```

1 package packt.book.jee.eclipse.ch7.ejb;
2
3 import java.util.List;
4
8
9
10 @Remote
11 public interface PacienteBeanRemote {
12
13
14     public List<PacienteDTO> getPaciente();
15
16
17
18 }
19

```

Figura 16 - PacienteBeanRemote

Data Transfere Objects DTO

Os DTO's permitem retornar informações das entidades do EJB de forma mais leve. Por outro lado, permitem expor apenas os campos que se pertenda que os clientes usem. No entanto, obriga transferir dados entre entidades e DTOs.

As classes, anteriormente criadas (ex: PacienteBean), foram criadas como um bean remoto e local, ou seja, com exibição sem interface. A implementação do método da interface remota retornará DTOs e o método local retornará entidades JPA. Para tal, foi adicionado o método getPaciente (assim como para as outras entidades) ao EJB. Criou-se o PacienteDTO, um objeto de transferência de dados, que é um POJO, que retornará instâncias do DTO a partir do método getCourses. OS DTO estão no projeto ClinicatEJBsClient porque serão compartilhados entre o EJB e seu cliente.

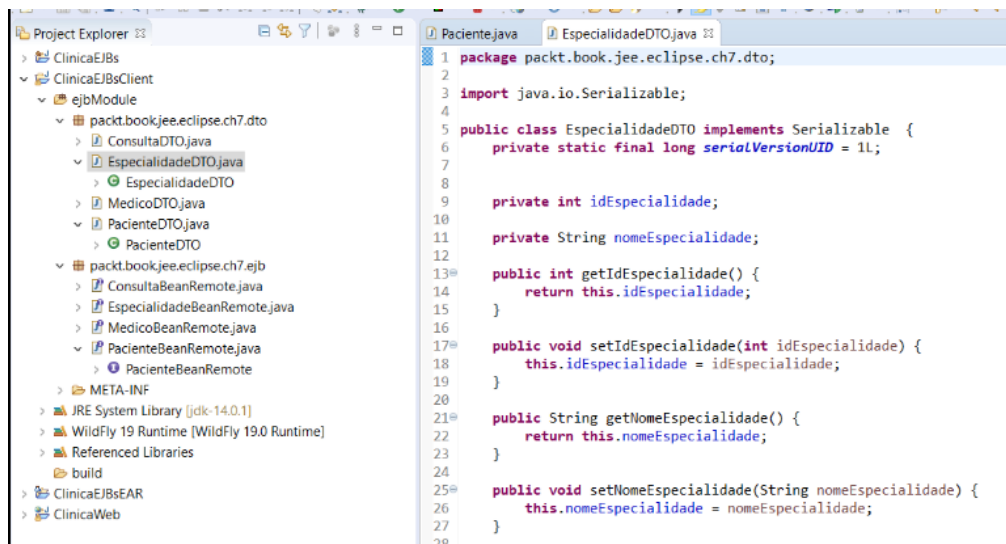


Figura 17 - EspecialidadeDTO

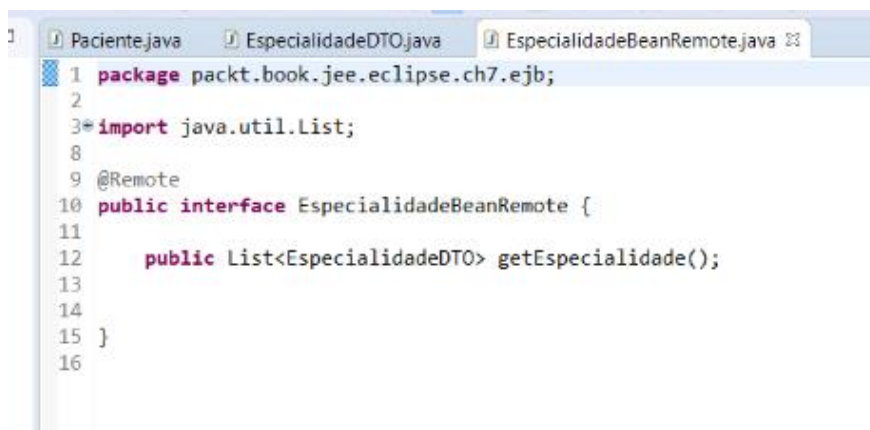


Figura 18 - EspecialidadeBeanRemote

O método `getPaciente` foi implementado no `PacienteBean` EJB. Para obter os cursos na base de dados, o EJB primeiro precisa obter uma instância do `EntityManager`, que é obtida através da criação de um `EntityManagerFactory`. Em seguida, essa instância é passada para a classe de serviço, que realmente obteve os dados da base de dados através de APIs JPA.

```

1 package packt.book.jee.eclipse.ch7.ejb;
2
3 import java.util.ArrayList;
4
5 /**
6  * Session Bean implementation class EspecialidadeBean
7  */
8 @Stateless
9 @LocalBean
10 public class EspecialidadeBean implements EspecialidadeBeanRemote {
11
12     @PersistenceContext
13     EntityManager entityManager;
14
15     public List<Especialidade> getEspecialidadeEntities() {
16         // Use named query created in Course entity using @NameQuery
17         TypedQuery<Especialidade> especialidadeQuery = entityManager.createNamedQuery("Especialidade.findAll", Espec
18         return especialidadeQuery.getResultList();
19     }
20
21     public EspecialidadeBean() {
22         // TODO Auto-generated constructor stub
23     }
24
25     @Override
26     public List<EspecialidadeDTO> getEspecialidade() {
27         List<Especialidade> especialidadeEntities = getEspecialidadeEntities();

```

Figura 19 - EspecialidadeBean

```

31
32 public EspecialidadeBean() {
33     // TODO Auto-generated constructor stub
34 }
35
36 @Override
37 public List<EspecialidadeDTO> getEspecialidade() {
38     List<Especialidade> especialidadeEntities = getEspecialidadeEntities();
39     List<EspecialidadeDTO> especialidades = new ArrayList<EspecialidadeDTO>();
40     for (Especialidade especialidadeEntity : especialidadeEntities) {
41         EspecialidadeDTO especialidade = new EspecialidadeDTO();
42         especialidade.setIdEspecialidade(especialidadeEntity.getIdEspecialidade());
43         especialidade.setNomeEspecialidade(especialidadeEntity.getNomeEspecialidade());
44         especialidades.add(especialidade);
45     }
46     return especialidades;
47 }
48
49

```

Figura 20 - EspecialidadeBean

Criação de JSF

Para que haja exibição dos conteúdos da base de dados ao utilizador, foram criadas página JSF com código em xhtml. Foram igualmente criadas managed beans que servem para invocar o método getPaciente (ou getMedico, getConsulta, etc) do PacienteEJB.

Por outro lado, o projeto ClinicaWeb precisa ter acesso à business interface do EJB, que está no ClinicaEJBsClient. Portanto, adicionou-se a referência de CourseManagementEJBsClient a CourseManagementWeb.

Web Application

Página Principal

Home Page

Apenas tem permissão a aceder a informações a cerca dos pacientes, médicos e respetivas consultas

[Ver os Pacientes](#) [Ver as Consultas](#) [Ver os Medicos](#)

Figura 21 - Página Principal da App

Pacientes

Pacientes

ID	Nome	Morada	Contacto
1	Paulo	Lamego	923758934
2	Ana	Porto	932153199
3	Anabela	Lamego	910203041
4	Jorge	Lamego	910204031
5	Maria	Lamego	930204571

[Página Principal](#)

Figura 22 - Página com os Pacientes na App

Consultas

Consultas

ID	Data	Hora	Observação	Paciente	Medico
1	12JUN2020	14h30	As dores no joelho estão a piorar	Paulo	Pedro
2	30MAY2020	14h30	as manchas na pele desapareceram	Maria	Carlos
3	29MAY2020	14h30	a pele continua muito seca	Jorge	Carlos

[Página Principal](#)

Figura 23 - Página com as Consultas na App e os respetivos Pacientes e Medicos

Médicos

Medicos

ID	Contacto	Morada	Nome	Especialidade
1	929292921	Lamego	Pedro	Clinica Geral
2	929292970	Lamego	Carlos	Clinica Geral

[Página Principal](#)

Figura 24 - Página com os Medicos e respetivas especialidades na App

Conclusão

O desenvolvimento deste trabalho serviu para alargar o leque de conhecimentos, no que diz respeito ao desenvolvimento de aplicações web. Bem como, possibilitou ganhar conhecimento a cerca de diferentes formas de o fazer.

A aplicação desenvolvida é baseada em Jakarta EE Enterprise Java Beans. Os EJBs são ideais para escrever lógica de negócios em aplicações web. Podem atuar como a ponte perfeita entre componentes da interface da web, como o JSF, e objetos de acesso a dados, como o JTO. Os EJBs podem ser distribuídos por vários servidores de aplicações JEE, o que melhora a escalabilidade da aplicação, e o seu ciclo de vida é gerenciado pelo contêiner. Os EJBs podem ser facilmente injetados em objetos gerenciados ou podem ser consultados através de JNDI.

É importante salientar, que a elaboração deste projecto, de certo modo, foi “facilitada” pelo *Eclipse*, que por sua vez, facilita a criação e o consumo de EJBs. No entanto, não foi uma tarefa fácil de todo. Exigindo grande esforço da parte do grupo que elaborou a parte prática do trabalho, para que possa perceber o funcionamento de todos os processos envolventes no processo de criação da aplicação web, desde a instalação do servidor no eclipse a configuração da datasource neste, como a conexão da aplicação com a base de dados, os processos de transmissão de dados, e finalmente, a exibição dos dados ao utilizador final.

Contudo, apesar de todos os contratempos, este trabalho foi extremamente proveitoso. Embora, não tenha sido executado da forma mais perfeita, serviu para absorver conhecimentos.

Bibliografia

1. Kulkarni, R. (2018). Java EE 8 Development with Eclipse-Third Edition
2. <https://synaptiklabs.com/posts/adding-the-mysql-jdbc-driver-into-wildfly/>
3. <https://www.baeldung.com/eclipse-wildfly-configuration>
4. <http://www.thejavageek.com/2015/01/09/creating-jsf-ejb-jpa-application-using-eclipse-wildfly/?fbclid=IwAR1jSAOpAz-u5KnISH9CNyXz3uF2cUYAD2e66a8MsxC8cOLbBGrdARgtjrY>