



CONSTRUÇÃO DE SISTEMAS DE SOFTWARE

APLICAÇÕES EMPRESARIAIS COM JAVA EE

Exercícios



Aplicações Java EE

1. *Java EE*

- (a) Qual é a diferença entre o Java SE e o Java EE? Dê alguns exemplos de APIs e serviços definidos pelo Java EE.
- (b) Indique algumas características das aplicações empresariais que o Java EE endereça e explique como.

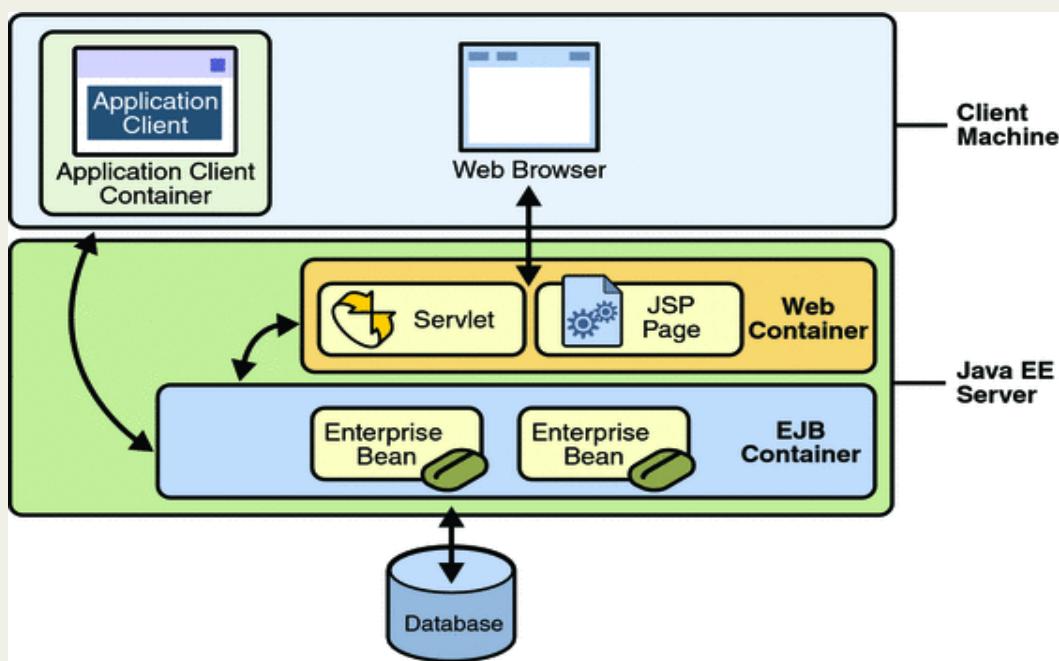
2. *Application Server*

- (a) A que usualmente se chama um *Java Application Server*? Indique 2 exemplos de *Java Application Servers* que atualmente podemos usar para correr as nossas aplicações Java EE.
- (b) Explique o que é um *ejb container* e o papel que cumpre no tratamento das chamadas RMI que chegam ao *Application Server*.
- (c) Quais são os outros *containers* Java EE e qual o papel deles?
- (d) De que forma os componentes das aplicações Java EE indicam os serviços de que precisam?
- (e) Quais os mecanismos usado por os servidores Java EE para fornecerem esses serviços?

3. Descreva sumariamente a arquitetura típica de uma aplicação Java EE que oferece dois interfaces, um na forma de uma aplicação cliente web e outro numa aplicação desktop. Através de um diagrama mostre os principais (tipos) de componentes da aplicação e a forma como comunicam.



Aplicações Java EE



Exercícios

4. (a) Que papel têm os *Enterprise Java Beans* numa aplicação Java EE?
- (b) Os *Session Beans* são um dos tipos de *Enterprise Java Beans*. Que tipo de *business interfaces* podem ter? Como se definem? Que implicações tem cada uma das escolhas? Como se escolhe?
- (c) Os *Stateless Session Beans* são um dos tipos de *Session Beans*. O que os caracteriza? Qual o seu ciclo de vida?
- (d) Como pode um componente Java EE ganhar acesso a um *Session Bean*?

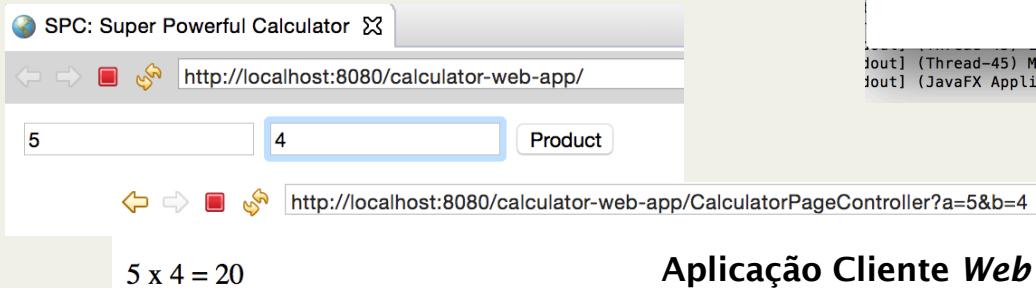
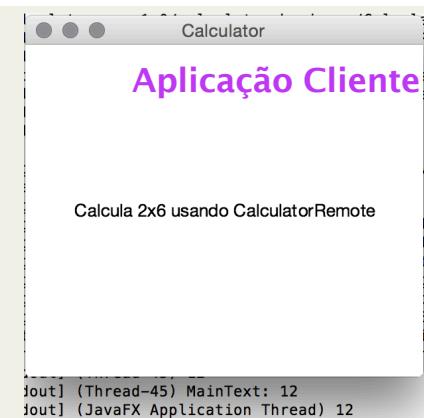


Exercícios

5. Pretende-se programar uma aplicação Java EE muito simples: uma calculadora que só sabe fazer multiplicações! A aplicação deve poder ser utilizada remotamente:

- recorrendo simplesmente a um navegador web;
- executando uma aplicação cliente Java num *application client container*;

```
.../wildfly-10.0.0.Final/bin/appclient.sh calculator-ear/target/calculator-ear-1.0.ear#calculator-app-client.jar
```



SPC: Super Powerful Calculator

http://localhost:8080/calculator-web-app/

5 4 Product

http://localhost:8080/calculator-web-app/CalculatorPageController?a=5&b=4

5 x 4 = 20

Aplicação Cliente Web

Exercícios

Tendo em conta estes requisitos e que a lógica de negócio (complicadíssima) está toda na classe Calculator mostrada abaixo:

```
public class Calculator {  
  
    public int product (int a, int b) {  
        return a * b;  
    }  
  
}
```

- (a) Defina a estrutura de projetos de que vai precisar.
- (b) Transforme os objetos da classe `Calculator` em EJBs apropriados para a tarefa e defina o(s) *business interface* necessário(s).
- (c) Programe a aplicação cliente Java que utiliza a aplicação para saber o resultado de 2×6



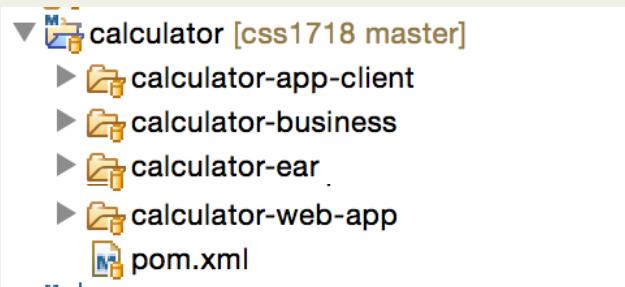
Estrutura do Código de uma Aplicação Java EE

- O código de uma aplicação Java EE do tipo do exemplo é organizado tipicamente em torno de 4 tipos de módulos.
- O Eclipse permite definir estes módulos com diferentes tipos de projetos



Estrutura do Código de uma Aplicação Java EE

Podemos usar um projeto maven para definir esta estrutura.



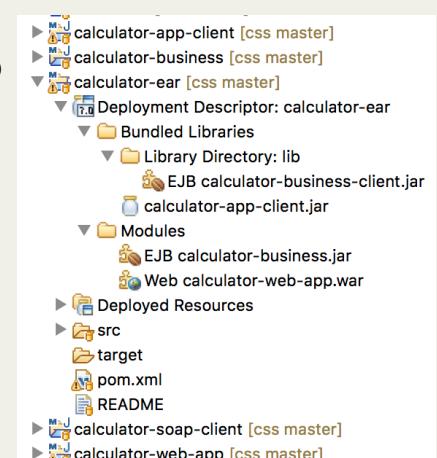
```
<groupId>pt.ulisboa.ciencias.di</groupId>
<artifactId>calculator</artifactId>
<version>1.0</version>

<packaging>pom</packaging>

<modules>
    <module>calculator-business</module>
    <module>calculator-web-app</module>
    <module>calculator-ear</module>
    <module>calculator-app-client</module>
</modules>
```

Estrutura do Código de uma Aplicação Java EE

- **calculator-business** contém o código que implementa a lógica do negócio (em particular, os EJBs)
- **calculator-web-app** contém o código do cliente web, i.e., a camada de apresentação que permite usar a aplicação através de um browser
- **calculator-app-client** que contém o código da aplicação cliente que vai correr no *client application container*
- **calculator-ear** que não contém código, apenas o pom que define quais os módulos a incluir no ear e onde vai fazer o deployment da aplicação web e dos EJBs



pom do calculator-ear

```
<packaging>ear</packaging>

<dependencies>

    <!-- Depend on the ejb module and war so that we can package them -->
    <dependency>
        <groupId>pt.ulisboa.ciencias.di</groupId>
        <artifactId>calculator-web-app</artifactId>
        <version>1.0</version>
        <type>war</type>
    </dependency>

    <dependency>
        <groupId>pt.ulisboa.ciencias.di</groupId>
        <artifactId>calculator-business</artifactId>
        <version>1.0</version>
        <type>ejb</type>
    </dependency>

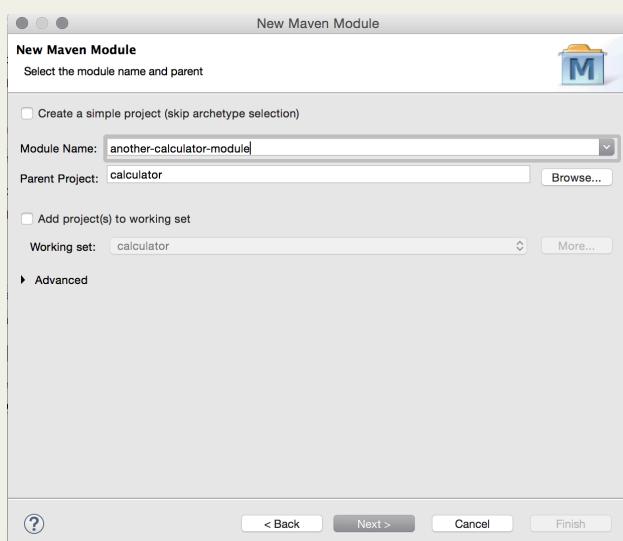
    <dependency>
        <groupId>pt.ulisboa.ciencias.di</groupId>
        <artifactId>calculator-app-client</artifactId>
        <version>1.0</version>
        <type>app-client</type>
    </dependency>

</dependencies>
```



Criação desta estrutura

- Criar um novo projeto *maven*, que vai ser a raiz — no exemplo **Calculator**
- Adicionar os diferentes módulos



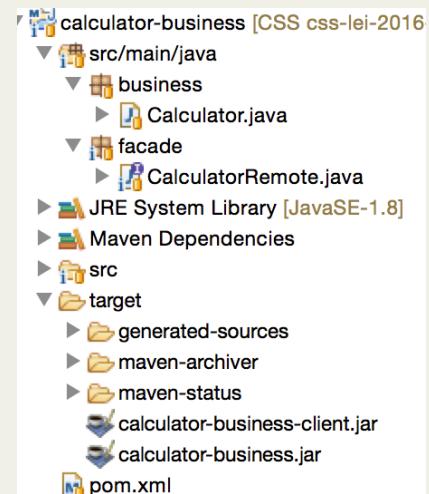
```
<build>
<finalName>${project.artifactId}</finalName>
<plugins>
    <plugin>
        <artifactId>maven-ejb-plugin</artifactId>
        <version>${version.ejb.plugin}</version>
        <configuration>
            <!-- Tell Maven we are using EJB 3.1 -->
            <ejbVersion>3.1</ejbVersion>
            <generateClient>true</generateClient>
            <clientIncludes>
                <clientInclude>facade/**</clientInclude>
            </clientIncludes>
        </configuration>
    </plugin>
</plugins>
```



Session Bean + Business interfaces

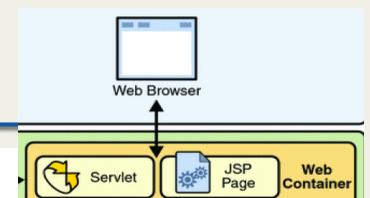
```
@Remote  
public interface CalculatorRemote {  
  
    public int product (int a, int b);  
  
}
```

```
@Stateless  
public class Calculator implements CalculatorRemote {  
  
    public int product (int a, int b) {  
        return a * b;  
    }  
  
}
```



Aplicação Cliente Web

```
@WebServlet("/CalculatorPageController")  
public class CalculatorPageController extends HttpServlet {  
    private static final long serialVersionUID = 1L;  
  
    @EJB private CalculatorRemote calculator;  
  
    /**  
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)  
     */  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
        int a = Integer.parseInt(request.getParameter("a"));  
        int b = Integer.parseInt(request.getParameter("b"));  
        request.setAttribute("result", calculator.product(a, b));  
        request.getRequestDispatcher("result.jsp").forward(request, response);  
    }  
  
}
```

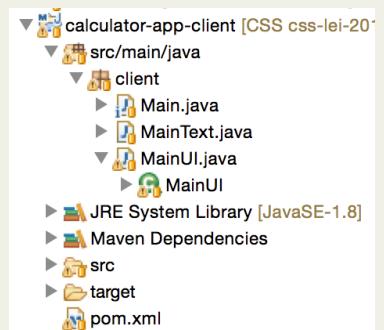


```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">  
    <title>Insert title here</title>  
</head>  
<body>  
    <p><c:out value ="${param['a']}></c:out> x <c:out value ="${param['b']}></c:out> = <c:out value ="${result}></c:out></p>  
    </body>  
</html>
```



Aplicação Cliente

```
public class Main {  
  
    @EJB  
    private static CalculatorRemote calculator;  
  
    public static void main(String[] args) {  
        System.out.println(calculator.product(2, 6));  
        MainText m = new MainText();  
        m.run();  
        MainUI.main(args);  
    }  
  
    public static int mult (int a, int b) {  
        return calculator.product(a, b);  
    }  
}
```

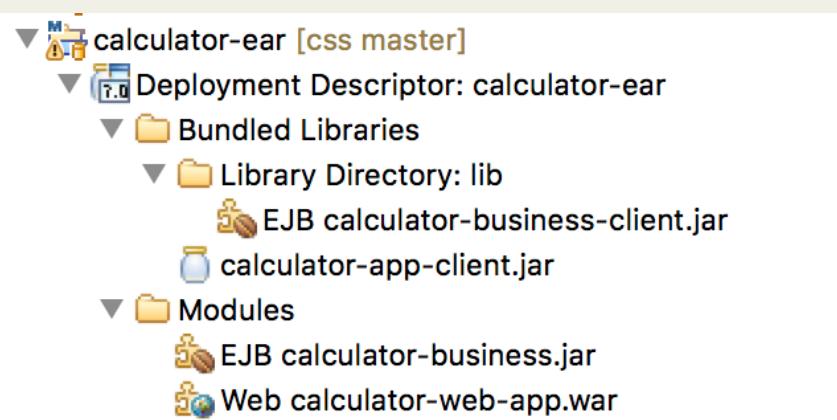
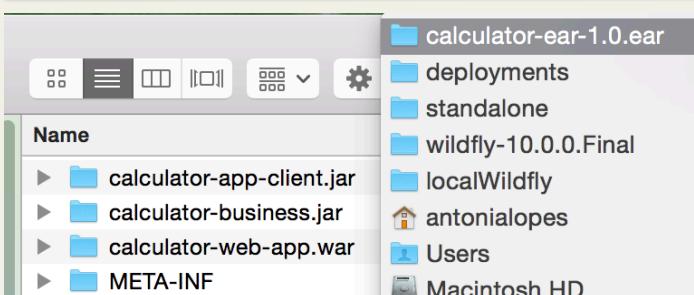


```
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.apache.maven.plugins</groupId>  
      <artifactId>maven-jar-plugin</artifactId>  
      <version>${version.jar.plugin}</version>  
      <configuration>  
        <archive>  
          <addMavenDescriptor>false</addMavenDescriptor>  
          <!-- need to set the Main class for the appclient -->  
          <manifest>  
            <mainClass>client.Main</mainClass>  
          </manifest>  
        </archive>  
      </configuration>  
    </plugin>  
  </plugins>  
</build>
```



Ciências
ULisboa

Organização do ear



Ciências
ULisboa

Exercícios

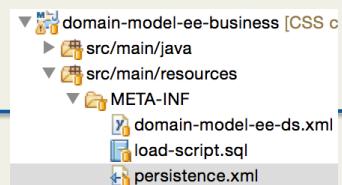
6. Persistência e Transações Geridas pelo Container

Recorde a forma como anteriormente vimos que uma aplicação pode realizar a gestão de entidades JPA e a forma como programaticamente é possível fazer a gestão de transações (`EntityTransaction`).

- No caso de aplicações Java EE que alternativas existem?
- Que implicações tem em termos de transações, escolhermos entregar ao *container* a gestão das entidades JPA?
- O que temos de fazer para ter transações geridas pelo *container*? O que é que isso significa? De que forma é possível mudar o comportamento por omissão?



Unidade de Persistência Gerida por Container



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="domain-model-ee-business" transaction-type="JTA">
        <!-- The datasource is deployed as <EAR>/META-INF/sri-srv-ds.xml, you can
            find it in the source at ear/src/main/resources/META-INF/domain-model-ee-ds.xml -->
        <jta-data-source>java:/jdbc/domain_model_ee_ds</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <shared-cache-mode>NONE</shared-cache-mode>
        <properties>
            <property name="javax.persistence.schema-generation.database.action"
                value="drop-and-create" />
            <property name="javax.persistence.schema-generation.create-source"
                value="metadata" />
            <property name="javax.persistence.schema-generation.drop-source"
                value="metadata" />
            <property name="javax.persistence.sql-load-script-source"
                value="META-INF/load-script.sql" />
        </properties>
    </persistence-unit>
</persistence>
```

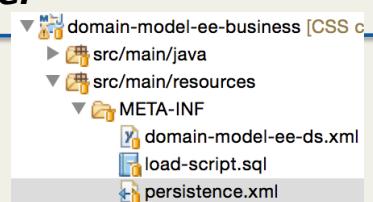
TIPO DE TRANSAÇÕES

NÃO HÁ ENTIDADES EXCLUÍDAS



Unidade de Persistência Gerida por Container

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://www.jboss.org/ironjacamar/schema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema http://docs.jboss.org/ironjacamar/schema/datasources_1_0.xsd">
  <!-- The datasource is bound into JNDI at this location. We reference
  this in META-INF/persistence.xml -->
  <datasource jndi-name="java:/jdbc/domain_model_ee_ds"
    pool-name="mysql_domain_model_ee_ds_Pool" enabled="true" use-java-context="true">
    <connection-url>jdbc:mysql://dbserver.alunos.di.fc.ul.pt:3306/css000</connection-url>
    <driver>domain-model-ee-ear-1.0.ear_com.mysql.jdbc.Driver_5_1</driver>
    <pool>
      <min-pool-size>5</min-pool-size>
      <max-pool-size>15</max-pool-size>
    </pool>
    <security>
      <user-name>css000</user-name>
      <password>css000</password>
    </security>
    <statement>
      <prepared-statement-cache-size>100</prepared-statement-cache-size>
      <share-prepared-statements/>
    </statement>
  </datasource>
</datasources>
```



Exercícios

7. Pretende-se desenvolver uma versão do sistema SaleSys como uma aplicação Java EE. A aplicação deve poder ser utilizada remotamente:

- recorrendo simplesmente a um navegador web;
- executando uma aplicação cliente Java num *application client container*.

A implementação deve seguir uma arquitetura em camadas com a camada de negócio organizada de acordo com o padrão *Domain Model* e com a persistência realizada com o JPA. A implementação da apresentação web deve ser organizada de acordo com os padrões *MVC* e *Front Controller*.

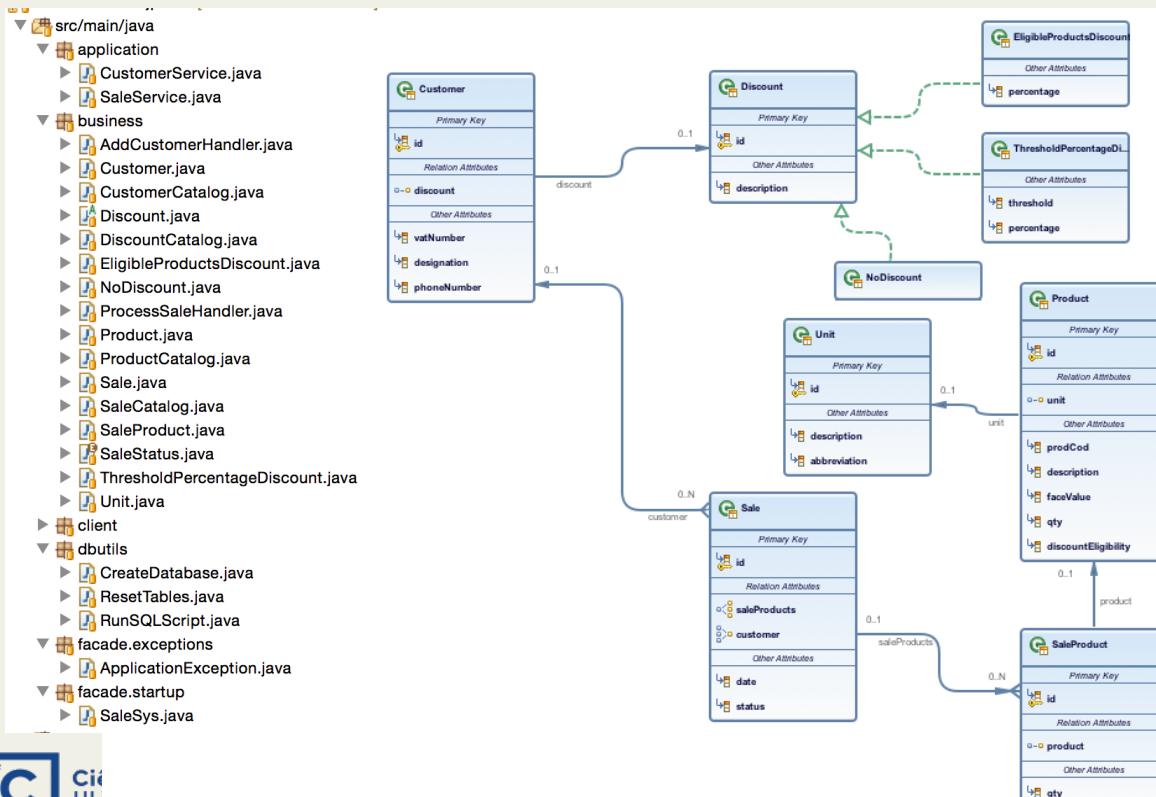
Exercícios

Tomando como ponto de partida a versão do SaleSys desenvolvida anteriormente, como uma aplicação Java SE + JPA em que a gestão de entidades era feita pela aplicação e as transações usadas eram *resource-local*:

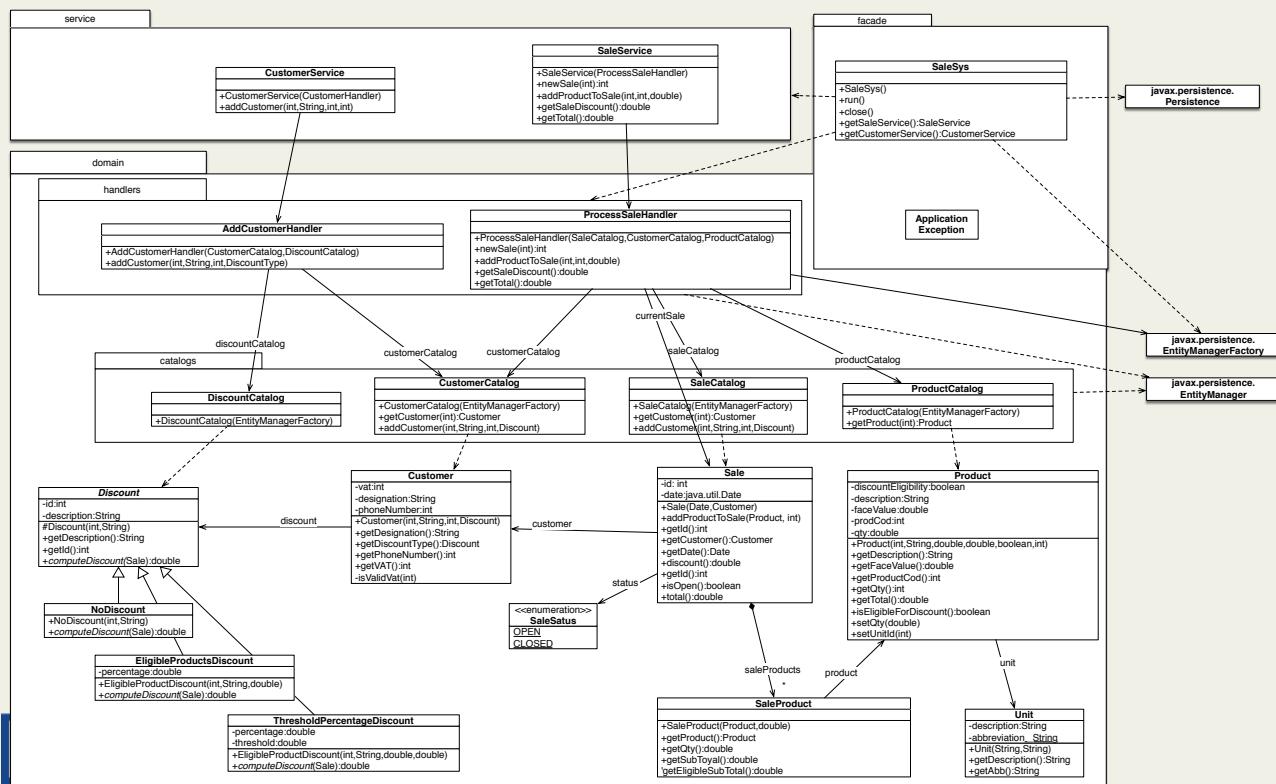
- Transforme as classes que implementam os serviços em EJB apropriados para a tarefa e defina o(s) *business interface* necessário(s) assim como tipos adequados para fazer a transferência de dados entre os serviços e os seus clientes. Analise em que medida será apropriado modificar os casos de uso de forma a não haver necessidade de manter estado (da interação).
- Analise as restantes classes envolvidas na solução — *handlers*, catálogos, entidades — modificando-as de forma apropriada.



Versão Java SE + JPA



Versão Java SE + ORM com JPA



Session Beans & Business Interfaces

- Os objetos das classes da camada *Services* são os candidatos perfeitos a serem os objetos que podem ser acedidos remotamente (*remote facade*).
 - Transformar a classe **application.CustomerService** para ser um *bean* de sessão sem estado que oferece acesso remoto aos métodos **addCustomer** e **getDiscounts**.
 - Anotando a classe com **@Stateless** e fazendo-a implementar um interface **ICustomerServiceRemote** declarado como sendo **@Remote**
- Tendo em conta que estes serviços vão ser expostos remotamente é preciso analisar
 - se os métodos têm a granularidade certa e
 - se os tipos de entrada e saída dos métodos são adequados e proceder aos ajustes necessários — definição de DTOs e interfaces
- Porque os clientes remotos apenas podem depender dos *business interfaces* e dos tipos de que estes dependem, é conveniente ter um pacote com estes tipos

Transferência de Dados

```
public class AddCustomerHandler {  
    public Iterable<Discount> getDiscounts() throws ApplicationException {  
        EntityManager em = emf.createEntityManager();  
        DiscountCatalog discountCatalog = new DiscountCatalog(em);  
        try {  
            return discountCatalog.getDiscounts();  
        } finally {  
            em.close();  
        }  
    }  
  
    public class CustomerService {  
        /**  
         * @return  
         */  
        public Iterable<IDiscount> getDiscounts() throws ApplicationException {  
            List<IDiscount> result = new LinkedList<>();  
            customerHandler.getDiscounts().forEach(result::add);  
            return result;  
        }  
    }  
  
    public interface IDiscount {  
        String getDescription();  
        int getId();  
    }  
}
```



Transferência de Dados

- Nesta caso pode se usar objetos **Discount** para fazer a transferência de dados entre módulos (ou aplicações) remotas. Deve se no entanto definir um interface que dê acesso apenas ao que é necessário (*getters*).
- De forma mais geral, decidir incluir a classe do domínio pode implicar a transmissão de mais informação do que a que é necessária, pois as classes de domínio podem referir outras que não têm interesse serem transmitidas entre módulos (ou aplicações) ou que não são serializáveis.
- Por causa da comunicação ser remota temos de nos preocupar com a quantidade de informação que vamos transmitir para fora da nossa aplicação. Assim, optamos por usar, na maioria das vezes, *Data Transfer Objects (DTO)*, que têm como único propósito carregar informação entre módulos (ou aplicações) remotas.
- No nosso caso, vamos criar uma classe **CustomerDTO** com os atributos da classe **Customer**. Esta classe tem de ser serializável, para poder ser usada nas invocações remotas. Usualmente os DTO são implementados em Java seguindo a convenção *Java beans* (nomeadamente, com um construtor sem parâmetros e *getters* e *setters* para as propriedades).



```

@Remote
public interface ICustomerServiceRemote {

    public CustomerDTO addCustomer (int vat, String denomination,
                                    int phoneNumber, int discountType)
        throws ApplicationException;

    public Collection<IDiscount> getDiscounts() throws ApplicationException;
}

```

```

public interface IDiscount extends Serializable {

    String getDescription();
    int getId();
}

```

```

public class CustomerDTO implements Serializable {

    private static final long serialVersionUID = -4087131153704256744L;
    private final int vatNumber;
    private final String designation;
    private final int id;

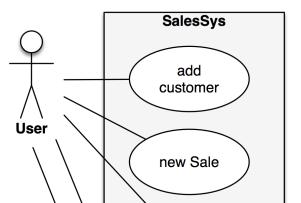
    public CustomerDTO(int vatNumber, String designation, int id) {
        this.vatNumber = vatNumber;
        this.designation = designation;
        this.id = id;
    }

    public String getDesignation() {
        return designation;
    }

    public int getId() {
        return id;
    }

    public int getVatNumber() {
        return vatNumber;
    }
}

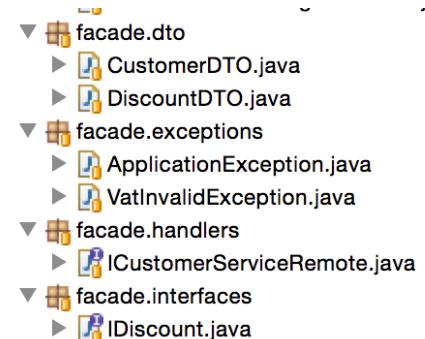
```



```

<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <artifactId>maven-ejb-plugin</artifactId>
            <version>${version.ejb.plugin}</version>
            <configuration>
                <!-- Tell Maven we are using EJB 3.1 -->
                <ejbVersion>3.1</ejbVersion>
                <generateClient>true</generateClient>
                <clientIncludes>
                    <clientInclude>facade/**</clientInclude>
                </clientIncludes>
            </configuration>
        </plugin>
    </plugins>
</build>

```



Services & Handlers

```
@Stateless
public class CustomerService implements ICustomerServiceRemote {

    @EJB
    private AddCustomerHandler addHandler;

    @Override
    public CustomerDTO addCustomer (int vat, String denomination,
                                    int phoneNumber, int discountType) throws ApplicationException {
        Customer c = addHandler.addCustomer(vat, denomination, phoneNumber, discountType);
        return new CustomerDTO(c.getVATNumber(), c.getDesignation(), c.getId());
    }
}
```



Services & Handlers

```
@Stateless
public class AddCustomerHandler {

    /**
     * The customer's catalog
     */
    @EJB
    private CustomerCatalog customerCatalog;

    /**
     * The discount's catalog
     */
    @EJB
    private DiscountCatalog discountCatalog;

    public Customer addCustomer (int vat, String denomination,
                                int phoneNumber, int discountType) throws ApplicationException {
        Discount discount = discountCatalog.getDiscount(discountType);
        try {
            return Customer c = customerCatalog.addCustomer(vat, denomination, phoneNumber, discount);
        } catch (Exception e) {
            throw new ApplicationException ("Error adding customer with VAT " + vat, e);
        }
    }

    /**
     * @return The list of discounts supported by the system.
     */
    public List<Discount> getDiscounts() throws ApplicationException {
        return discountCatalog.getDiscounts();
    }
}
```



Mudança na gestão de entidades e transações

- As classes que antes tinham um atributo que era uma fabrica de gestores de entidades (**EntityManagerFactory emf**) para permitir criar gestores de entidades, deixam de ter este atributo. Esta gestão passa para o *EJB container* e deve ser removido da aplicação.
- Consoante a utilização da aplicação, o servidor aplicacional estabelece uma ou mais ligações à base de dados e cria e gera diversos objetos **EntityManagerFactory**.



Mudança na gestão de entidades e transações

- Por cada pedido à aplicação para criar um *customer*, o *container* inicia uma nova transação ao qual fica associado um contexto de persistência. No fim do pedido faz *commit* das alterações nesse contexto (ou *rollback* de acordo com a informação do atributo **rollbackOnly**).
- Para ter acesso ao contexto de persistência da transação do pedido corrente deve ser declarada a necessidade de um **EntityManager** e o servidor irá injetar automaticamente esta referência no objeto (o contexto de persistência é o da transação).
- Como o objeto **EntityManager** é injetado pelo *container* não necessitamos de o construir e não temos de o fechar também.
- A gestão de transações e entidades numa aplicação Java EE é realizada pelo *EJB container* e é necessário fazer algumas alterações nos handlers que estavam antes a fazer eles próprios esta gestão.



Catálogos

- As classes **CustomerCatalog** e **DiscountCatalog** não precisam de manter estado e portanto não precisam de ter atributos cuja informação perdura por vários pedidos.
- O único atributo que têm é o **EntityManager**, mas que é injetado pelo servidor.
- Deste modo, estas duas classes podem ser elas próprias *beans* de sessão sem estado (mas só de acesso local).
- Podemos portanto declarar na classe **AddCustomerHandler** que necessitamos dos dois referidos catálogos durante a sessão (usando a anotação **@EJB**) e assim é o servidor que faz a gestão da criação e atribuição destes objetos às várias sessões.

```
@Stateless
public class CustomerCatalog {

    /**
     * Entity manager for accessing the persistence service
     */
    @PersistenceContext
    private EntityManager em;

    * Finds a customer given its VAT number.□
    public Customer getCustomer (int vat) throws ApplicationException {
        TypedQuery<Customer> query = em.createNamedQuery(Customer.FIND_BY_VAT_NUMBER, Customer.class);
        query.setParameter(Customer.NUMBER_VAT_NUMBER, vat);
        try {
            return query.getSingleResult();
        } catch (PersistenceException e) {
            throw new ApplicationException ("Customer not found.");
        }
    }

    * Adds a new customer.□
    @Transactional(Transaction.TxType.REQUIRES_NEW)
    public Customer addCustomer (int vat, String designation, int phoneNumber, Discount discountType)
        throws ApplicationException {
        Customer customer = new Customer(vat, designation, phoneNumber, discountType);
        em.persist(customer);
        return customer;
    }

    public Customer getCustomerById(int id) throws ApplicationException {
        Customer c = em.find(Customer.class, id);
        if (c == null)
            throw new ApplicationException("Customer with id " + id + " not found");
        else
            return c;
    }
}
```

```

@Entity
@NamedQueries({
    @NamedQuery(name=Customer.FIND_BY_VAT_NUMBER, query="SELECT c FROM Customer c WHERE c.vatNumber = :" + 
        Customer.NUMBER_VAT_NUMBER),
    @NamedQuery(name=Customer.FIND_ALL_CUSTOMERS, query="SELECT c FROM Customer c")
})
public class Customer {

    // Named query name constants
    public static final String FIND_BY_VAT_NUMBER = "Customer.findByVATNumber";
    public static final String NUMBER_VAT_NUMBER = "vatNumber";
    public static final String FIND_ALL_CUSTOMERS = "Customer.findAllCustomers";

    // Customer attributes

    * Customer primary key. Needed by JPA. Notice that it is not part of the...
    @Id @GeneratedValue private int id;

    * Customer's VAT number
    @Column(nullable = false, unique = true) private int vatNumber;

    * Customer's name. In case of a company, this represents its commercial denomination ...
    @Column(nullable = false) private String designation;

    * Customer's contact number
    @SuppressWarnings("unused")
    private int phoneNumber;

    * Customer's discount. This discount is applied to eligible products.
    @OneToOne @JoinColumn(nullable = false) private Discount discount;

    // 1. constructor

    * Constructor needed by JPA.
    Customer() {}

    * Creates a new customer given its VAT number, its designation, phone contact, and discount type.
    public Customer(int vatNumber, String designation, int phoneNumber, Discount discountType) {}
}

```

Impacto de Mudança de Implementação do JPA

- Embora o *Hibernate* e o *EclipseLink* sejam duas implementações do JPA, há algumas diferenças entre elas (em elementos não especificados na API).
- Vão notar três coisas que diferem entre o *Hibernate* e o *EclipseLink*, nomeadamente
 - No *Hibernate* os nomes das tabelas e dos atributos respeitam fielmente os nomes das classes e dos atributos, em oposição ao *EclipseLink* que atribuía nomes com todas as letras em maiúsculas;
 - o nome da tabela de sequências automáticas chama-se agora *hibernate_sequence*;
 - o *EclipseLink* assumia por omissão a relação de composição como uma relação *OneToOne* e o *Hibernate* obriga a indicar que relação se pretende. Por exemplo, na classe *Product* é obrigatório anotar o atributo *unit* com uma anotação, enquanto que no *EclipseLink* era assumida a anotação *@OneToOne*.
- O esforço de adaptação deverá ser mínimo



Cliente Web

- Na implementação do *Front Controller* seguimos de novo o GoF *Command*, mas a criação dos objetos é delegada no servidor aplicacional.
- Esta abordagem tem duas vantagens:
 - o servidor faz uma melhor gestão dos objetos do que o nosso *Front Controller*, pois estávamos a criar objetos novos a cada pedido e o servidor reutiliza os objetos entre pedidos, só criando objetos distintos para pedidos concorrentes;
 - queremos injetar os *serviços* nas ações e isso só acontece se for o servidor a criar os objetos das ações.
- A injeção dos serviços nas ações é necessária porque agora os serviços são *beans* de sessão e são geridos pelo servidor.
- As ações agora serão EJBs sem estado.
- Antes as ações obtinham o serviço a partir do *ServletContext*. Vamos deixar de o fazer e em vez disso vamos injetar a dependência.



```
/*
 * @WebServlet(FrontController.ACTION_PATH + "/*")
 */
public class FrontController extends HttpServlet {
    private static final long serialVersionUID = 1L;
    static final String ACTION_PATH = "/action";
    private InitialContext context;
    /**
     * Maps http actions to the objects that are going to handle them
     */
    protected HashMap<String, String> actionHandlers;
    private Properties appProperties;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException {
        String actionURL = request.getPathInfo();
        String actionJNDI = getActionHandlerAddress(ACTION_PATH + actionURL);
        Action actionCommand;
        try {
            actionCommand = (Action) context.lookup(actionJNDI);
        } catch (NamingException e) {
            actionCommand = new UnknownAction();
        }
        actionCommand.process(request, response);
    }

    @Override
    public void init() {
        String propertiesFileName = "/WEB-INF/app.properties";
        actionHandlers = new HashMap<>();
        actionHandlers.put("unknownAction", "java:module/UnknownAction");
        appProperties = new Properties();
        try {
            context = new InitialContext();
            appProperties.load(getClass().getResourceAsStream(propertiesFileName));
            for (Entry<Object, Object> keyValue : appProperties.entrySet()) {

```

Cliente web

```
@Stateless
public class CreateCustomerAction extends Action {

    @EJB private ICustomerServiceRemote addCustomerHandler;

    @Override
    public void process(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        NewCustomerModel model = createHelper(request);
        request.setAttribute("model", model);

        if (validInput(model)) {
            try {
                addCustomerHandler.addCustomer(intValue(model.getVATNumber()),
                    model.getDesignation(), intValue(model.getPhoneNumber()), intValue(model.getDiscountType()));
                model.clearFields();
                model.addMessage("Customer successfully added.");
            } catch (ApplicationException e) {
                model.addMessage("Error adding customer: " + e.getMessage());
            }
        } else
            model.addMessage("Error validating customer data");

        request.getRequestDispatcher("/addCustomer/newCustomer.jsp").forward(request, response);
    }
}
```



Atualização do ficheiro WEB-INF/app.properties

- Nas mensagens na consola aquando da instalação da aplicação podem ver que para cada EJB são impressos os vários endereços JNDI.

```
15:11:38,803 INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-5) WFLYEJB0473: JNDI bindings for session bean named 'UnknownAction' in deployment
java:global/domain-model-ee-ear-1.0/domain-model-ee-web-client/UnknownAction!controller.web.inputController.actions.UnknownAction
java:app/domain-model-ee-web-client/UnknownAction!controller.web.inputController.actions.UnknownAction
java:module/UnknownAction!controller.web.inputController.actions.UnknownAction
java:global/domain-model-ee-ear-1.0/domain-model-ee-web-client/UnknownAction
java:app/domain-model-ee-web-client/UnknownAction
java:module/UnknownAction
```

```
15:11:38,803 INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-5) WFLYEJB0473: JNDI bindings for session bean named 'CreateCustomerAction' in deployment
java:global/domain-model-ee-ear-1.0/domain-model-ee-web-client/CreateCustomerAction!controller.web.inputController.actions.CreateCustomerAction
java:app/domain-model-ee-web-client/CreateCustomerAction!controller.web.inputController.actions.CreateCustomerAction
java:module/CreateCustomerAction!controller.web.inputController.actions.CreateCustomerAction
java:global/domain-model-ee-ear-1.0/domain-model-ee-web-client/CreateCustomerAction
java:app/domain-model-ee-web-client/CreateCustomerAction
java:module/CreateCustomerAction
```

- Para cada ação é preciso copiar o JNDI do módulo e atualizar a entrada no ficheiro *app.properties* respetiva. Depois de fazer o *deploy* da aplicação, está pronta para testar a adição de clientes

```
SaleSys: menu principal app.properties
1 # URL wiring
2
3 # New customer use case
4 appRoot/action/clientes/novoCliente = java:module/NewCustomerAction
5 appRoot/action/clientes/criarCliente = java:module/CreateCustomerAction
6
```



Cliente Java RMI

```
public class Main {  
  
    @EJB  
    private static ICustomerServiceRemote addCustomerHandler;  
  
    /**  
     * An utility class should not have public constructors  
     */  
    private Main() {  
    }  
  
    /**  
     * A simple interaction with the application services  
     *  
     * @param args Command line parameters  
     */  
    public static void main(String[] args) {  
        presentation.fx.Startup.startGUI(addCustomerHandler);  
    }  
}
```

