



# Construção de Sistemas de Software

## Distribuição

### Padrões para as várias camadas

Distribuição, concorrência e sessões

Client and Server Session State  
Remote Façade  
Data Transfer Object  
Optimistic and Pessimistic Locking

EJB

JAX-WS

Apresentação

MVC  
Presentation Model  
Front controller  
Page controller  
Page template

JSP

Servlet

EL

JavaFX

Negócio

Domain Model  
Transaction Script  
Table Module

Dados

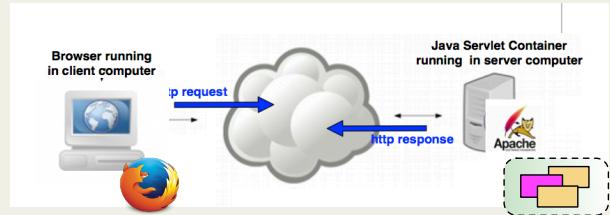
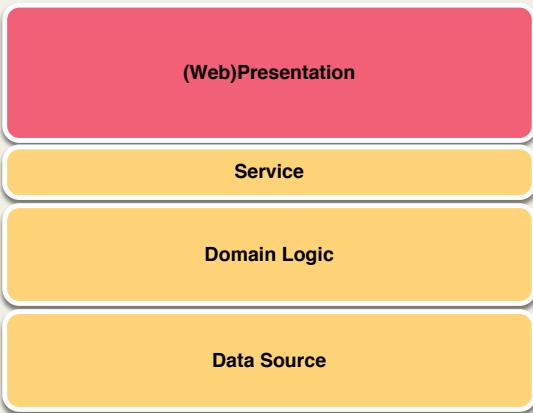
Row data gateway  
Table data gateway  
Active Record  
Data Mapper

JPA

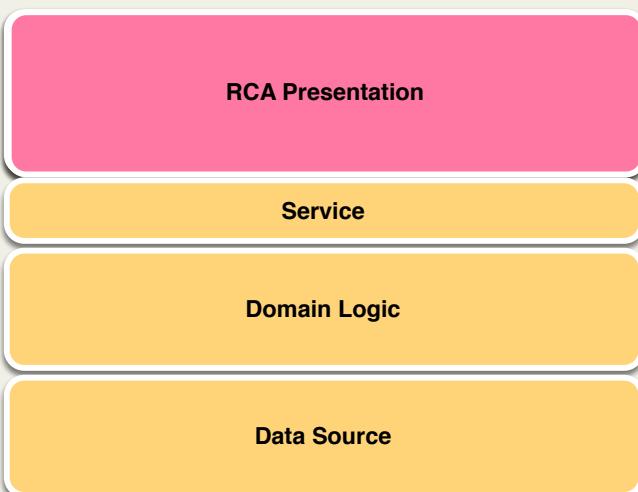
JDBC



## Onde correr o código?

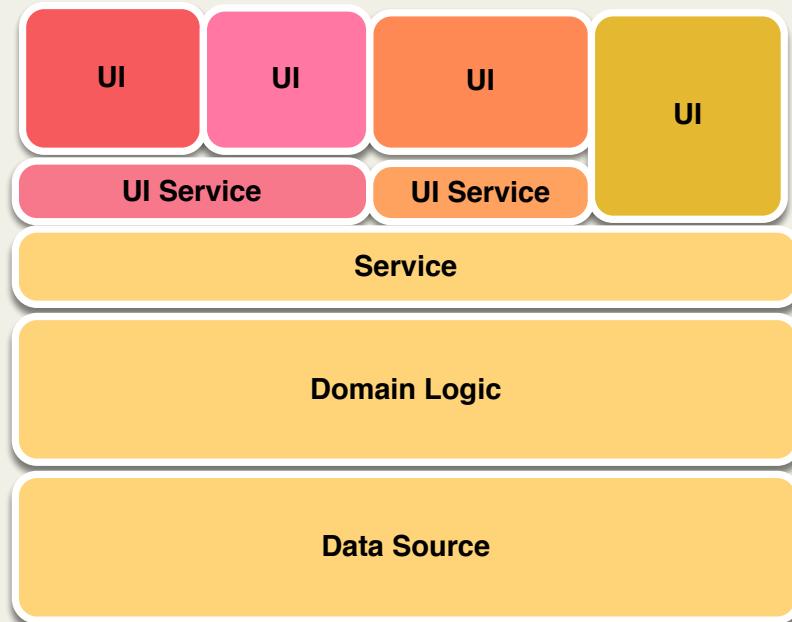


## Onde correr o código?



## Onde correr o código?

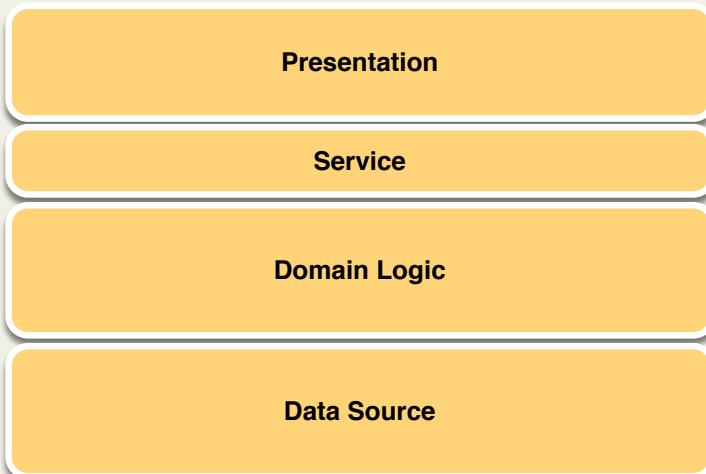
---



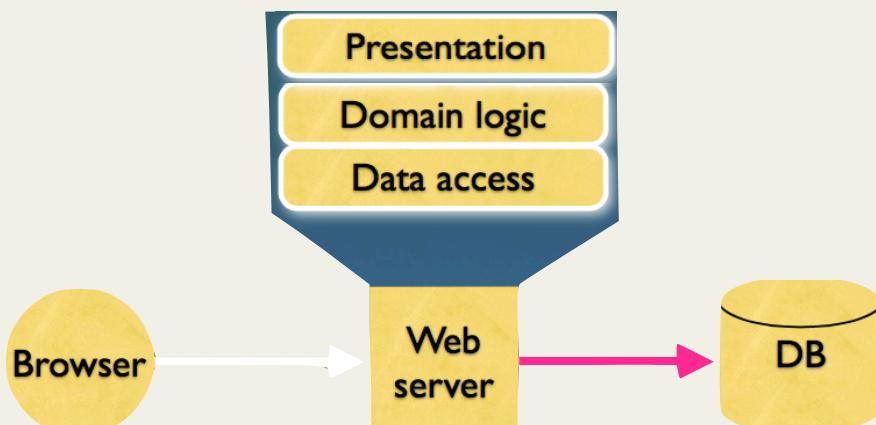
## Onde correr o código?

---

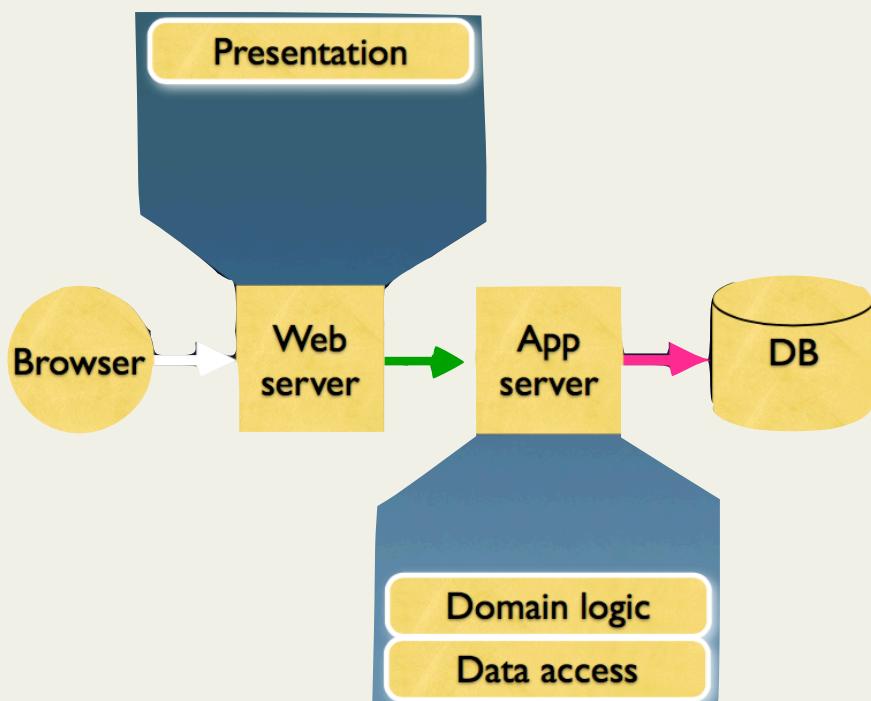
- A decisão de onde correr o código responsável pela **apresentação** depende em primeiro lugar do tipo de UI pretendido



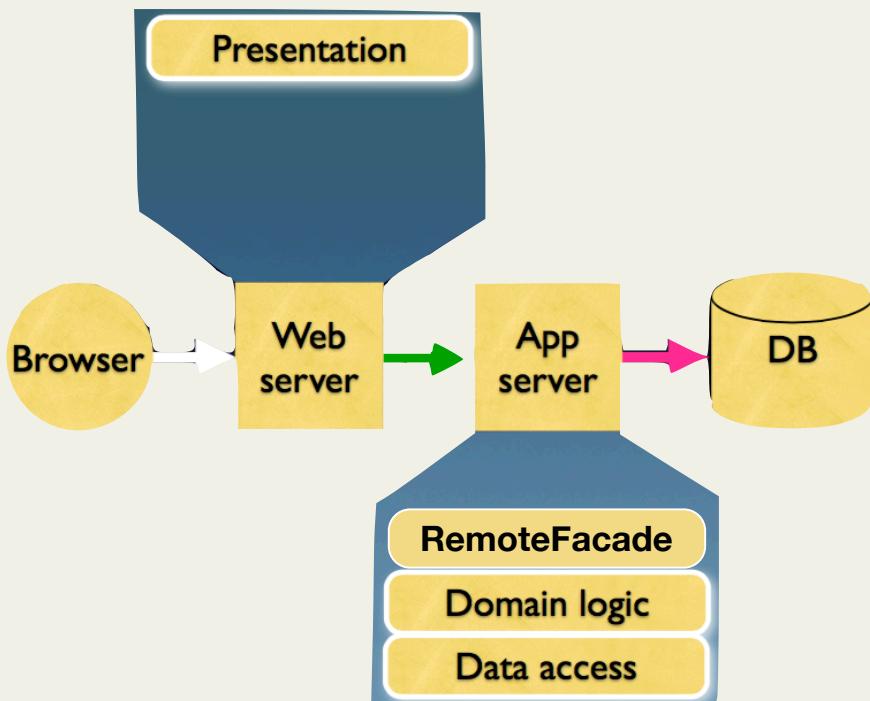
## Correr tudo num servidor



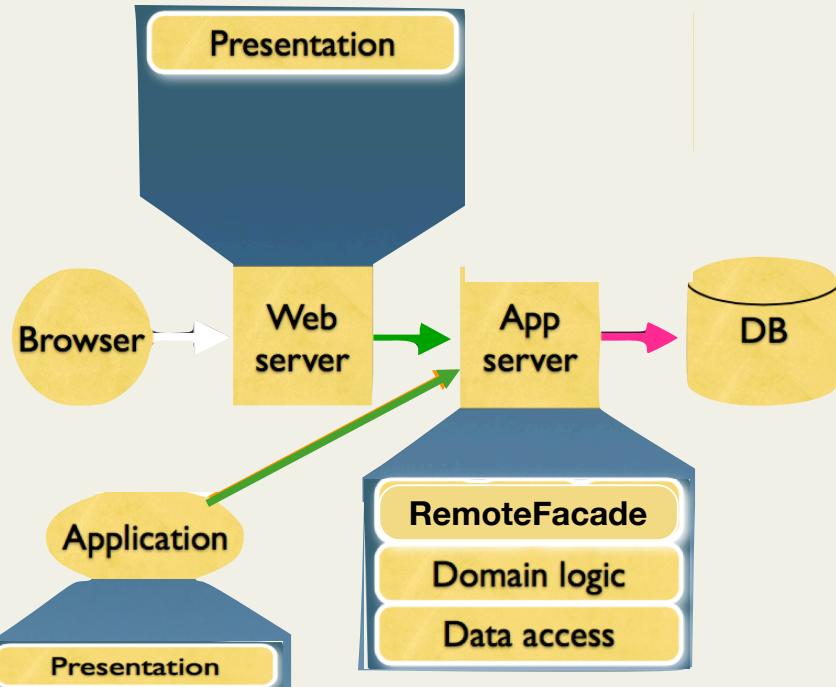
## Correr em dois servidores



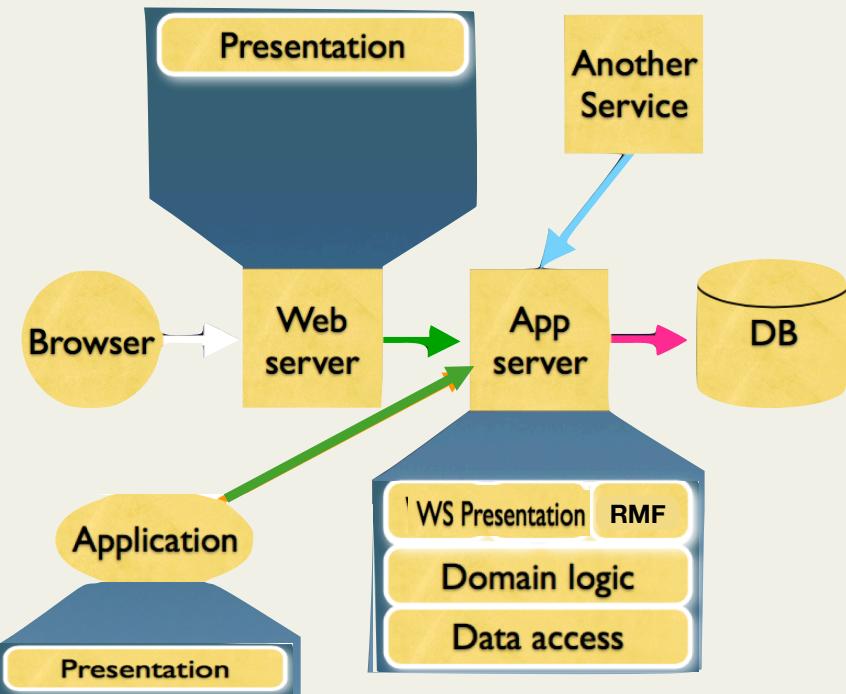
## Correr em dois servidores



## Correr também nos clientes



## Correr também nos clientes



## Desafios da Distribuição

- Acessos paralelos ou concorrentes
  - Gestão de recursos partilhados
- Clientes locais e remotos
  - Como separar os pedidos dos clientes
  - Como transferir informação
- Gestão de
  - segurança
  - escalabilidade
  - disponibilidade



## Solução oferecida pelo Java EE

- O **Java EE**, ou **Java Platform Enterprise Edition**, estende o Java SE com várias especificações (APIs)
- Em 2017 a Oracle decidiu ceder os direitos sobre o Java EE à Eclipse Foundation, implicando uma mudança de nome
- Em 2018, a Eclipse Foundation lançou o **Jakarta EE**

For many years, Java EE has been a major platform for mission-critical enterprise applications. In order to accelerate business application development for a cloud-native world, leading software vendors collaborated to move Java EE technologies to the Eclipse Foundation where they will evolve under the Jakarta EE brand.



## Solução oferecida pelo Java EE

- Fornecer serviços *standard* e APIs para os usar, que são úteis para a generalidade das aplicações empresariais
- Um dos principais objetivos é a ***separation of concerns***
  - Possibilitar que código relacionado com o negócio esteja separado do código relacionado com as propriedades não funcionais
  - Permitir que os programadores se possam concentrar em exclusivo na implementação do código relacionado com o negócio

## Solução oferecida pelo Java EE

---

- As aplicações são **programadas** usando um conjunto de **serviços standard** de acordo com o que é definido por várias APIs como
  - Enterprise Java Beans (EJB)
  - Java Transaction API (JTA)
  - Java Web Services (JAX-WS)maioritariamente através de **anotações** no código
- Entre este **serviços** temos
  - Gestão de acessos concorrentes
    - Sessões
    - Serviços Web
    - Troca de mensagens
  - Controlo de transações
    - Com um ou mais sistemas transacionais
  - *Timers* para agendamentos programados

## Solução oferecida pelo Java EE

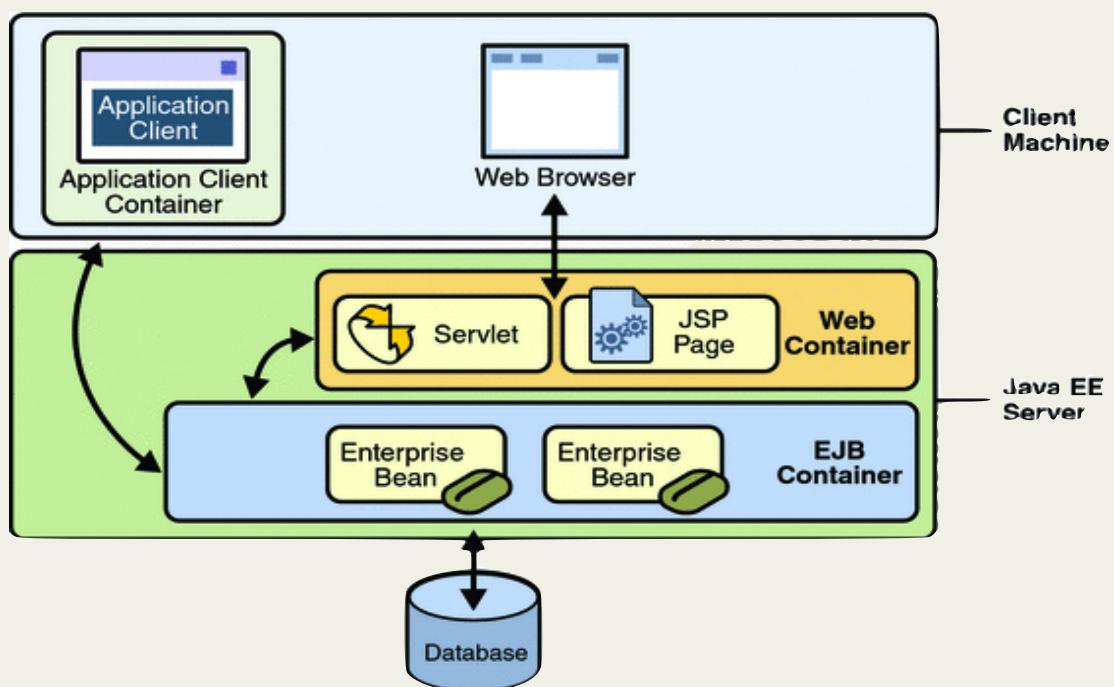
---

- As aplicações são **programadas** usando um conjunto de **serviços standard** de acordo com o que é definido por várias APIs muito frequentemente através de **anotações**
- As aplicações são **executadas** recorrendo a um **Application Server** que fornece implementações destas APIs e dos serviços standard do Java EE
- Entre os **Java EE Application Servers** mais populares temos
  - Wildfly (JBoss/RedHat)
  - Glassfish, Weblogic (Oracle)
  - WebSphere (IBM)

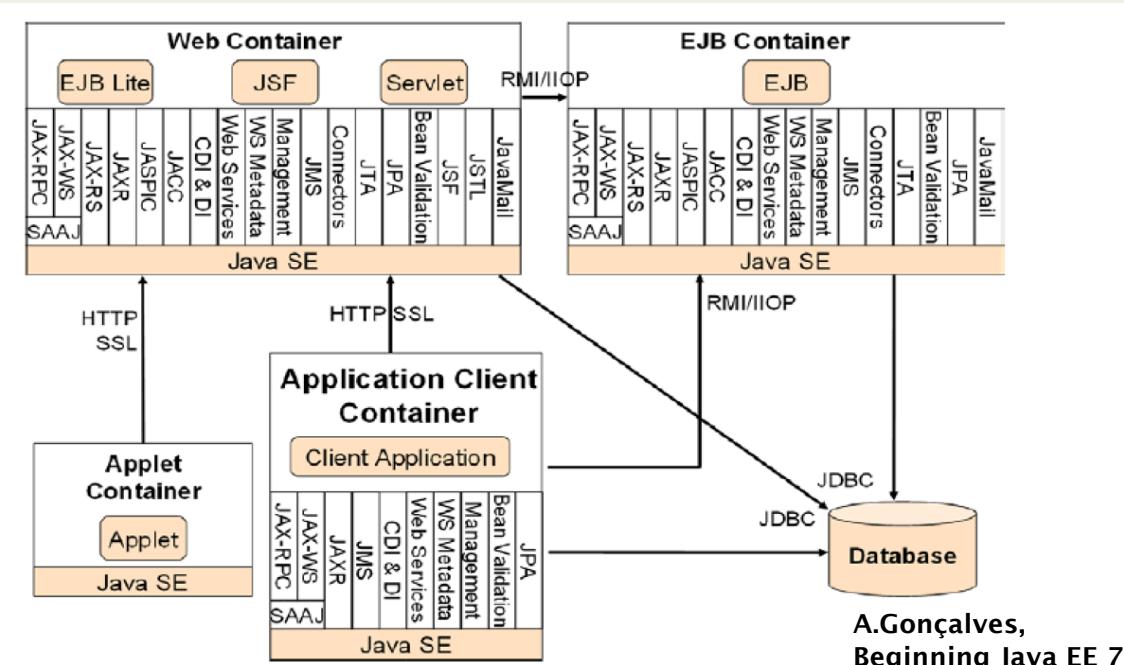
## Servidores e *Containers Java EE*

- Os servidores Java EE são chamados **servidores aplicacionais** porque permitem **servir dados aplicacionais aos clientes**
  - por analogia com os servidores web servem páginas *web* aos *browsers*
- A prestação de serviços aos diferentes componentes das aplicações EE faz-se na forma de 3 tipos de **containers**
  - **Web/Servlet Container**
  - **EJB Container**
  - **Application Client Container** (opcional)

## Aplicações Java EE

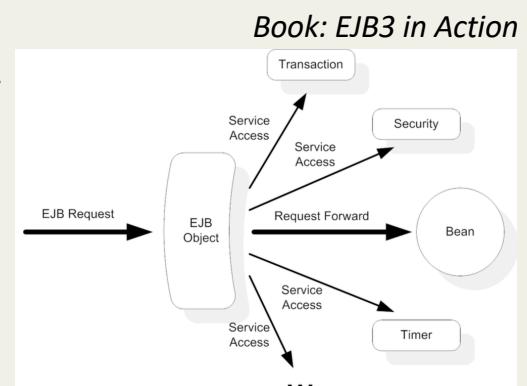


## APIs & Serviços / Container



## EJB Container

- Fornece a infraestrutura de execução para os **Enterprise Java Beans**
- Interceta as invocações que são realizadas pelos clientes dos EJBs que estão a executar no *container* e, de acordo com os serviços que são por estes requeridos, executa um conjunto de tarefas adicionais (**container services**)
  - examinar as credenciais de segurança
  - obter objeto da sessão
  - começar uma transação
  - invocar elementos da persistência
  - ...



## **EJB Container**

- Em vez da aplicação que desenvolvemos ter o controlo total sobre a acessos, transações, etc, este papel está a cargo do *EJB container*
- Este papel é baseado no princípio da **Inversão do Controlo**
  - o fluxo de controlo é passado do *container* para a aplicação
- Como controlar e interagir com o *container*?
  - através da **chamada de métodos** em determinados pontos do ciclo de vida da aplicação
  - através da **injeção de dependências (DI)**



## **Inversão do Controlo**

### **Chamada de métodos durante o ciclo de vida**

- Através de diferentes anotações é possível indicar que um determinado método deve ser chamado pelo *container* em certas situações
  - quando a aplicação arranca
  - quando uma entidade é persistida
  - quando é recebido um pedido
  - ...
- São usados para isso **Method callbacks** e **Event listeners**
  - @WebListener**
  - @PostConstruct**
  - @PostLoad @PreUpdate @PrePersist**

## Inversão do Controlo

---

### Injeção de Dependências

- A aplicação declara apenas os recursos que necessita para assegurar as suas funcionalidades
- O *container* lida com as complexidades da instanciação, inicialização e gestão da utilização dos recursos fornecendo referências destes à aplicação sempre que requeridos
  - Os objetos são criados pelo *container*
  - As aplicações declaram quais os objetos que pretendem
  - O *container* passa à aplicação a referência para esses objetos

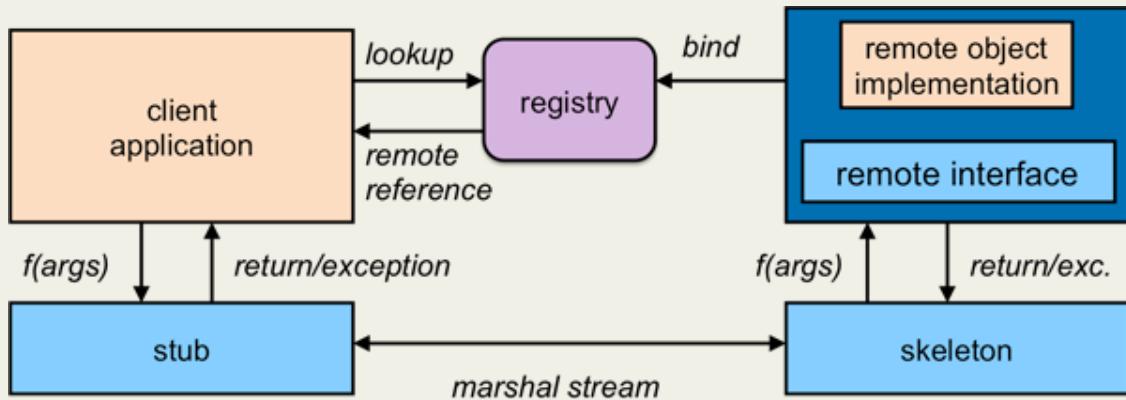
## Solução oferecida pelo Java EE: Vantagens

---

- **Simplicidade**
  - Simplifica o desenvolvimento de grandes aplicações distribuídas, já que permitem que os programadores se concentrem nos problemas do negócio — o **EJB container** sabe lidar com a gestão de acessos paralelos, transações, segurança, autenticação, autorização, etc.
- **Portabilidade**
  - As aplicações podem ser movidas para diferentes Java EE *containers*
- **Escalabilidade + Transparência da Localização**
  - Os EJBs de uma aplicação podem ser distribuídos por várias máquinas e a sua localização vai ser transparente para os clientes



## RMI



## RMI

```
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloClient
{
    public static void main(String arg[])
    {
        //

        // I download server's stubs ==> must set a SecurityManager
        System.setSecurityManager(new RMISecurityManager());

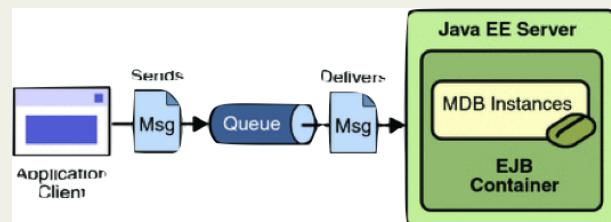
        try
        {
            Hello obj = (Hello) Naming.lookup( "//" +
                "servico.di.ciencias.ulisboa.pt" +
                ":8080" +
                "/HelloServer");           //objectname in registry
            System.out.println(obj.sayHello());
        }
        catch (Exception e)
        {
            System.out.println("HelloClient exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

transparência de localização  
significa o código não ter  
coisas destas



## Enterprise Java Beans

- Componentes que servem para **encapsular a lógica de negócio** da aplicação e ajudar a lidar com várias fontes de complexidade
- **Message Driven Beans (MDB)**
  - Funcionam como *listeners* para um tipo particular de **mensagens**
  - Permitem as aplicações processar mensagens dos clientes **assincronamente**



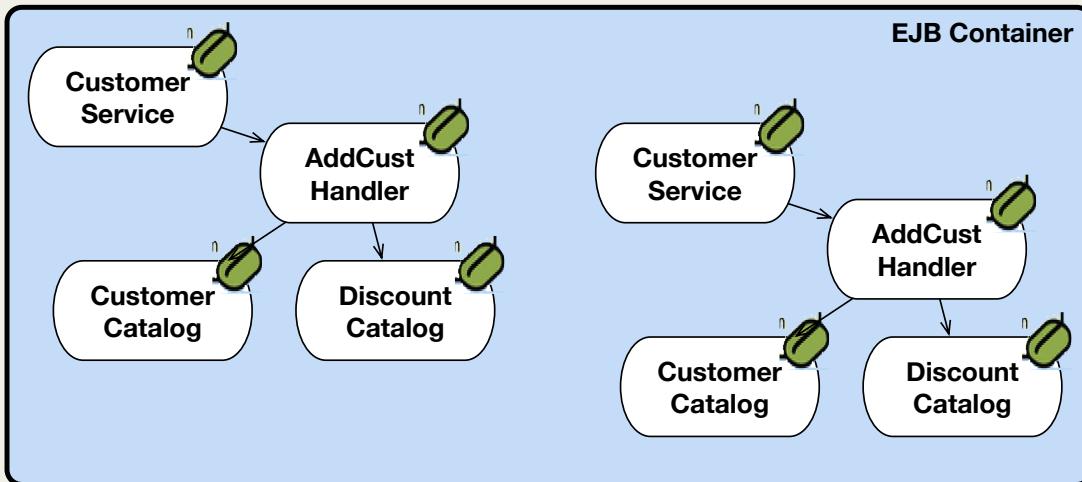
- **Session Beans**

- Para comunicação **síncrona** por chamadas a procedimentos/invocação de métodos
- Pode ou não guardar estado entre chamadas

## Enterprise Java Beans

- Componentes que servem para **encapsular a lógica de negócio** da aplicação e ajudar a lidar com várias fontes de complexidade
- **Message Driven Beans (MDB)**
  - ...
- **Session Beans**
  - Funcionam como uma **sessão** —ligação entre o cliente e o servidor com duração finita— no contexto da qual o cliente faz os pedidos subjacentes a uma **transação de negócio**
  - Permitem programar a aplicação de forma a que a interação de cada cliente com a aplicação é como se fosse a única que estivesse a ocorrer naquele momento — o *container* trata do resto

## Session Beans



## Session Beans

- Podemos programar estes *beans* como se de uma aplicação *desktop* com um único utilizador se tratasse
- Suportam RMI e acesso remoto de serviços web baseados em SOAP
- São criados pelo *container* e fornecidos à a aplicação de acordo com as suas necessidades
- Há três tipos de *session beans*:
  - **Stateless** — não mantém nada em memória entre pedidos
  - **Stateful** — guardam o estado da conversação com o cliente nos seus atributos
  - **Singleton** — é instanciado apenas uma vez por aplicação e guarda estado que é partilhado por todos os clientes da aplicação



## **Stateless Session Beans**

- Caracterizam-se por não manterem nada em memória entre pedidos, ou seja, não mantém estado conversacional
- São guardados numa *pool* para servirem diferentes clientes em diferentes pedidos
- Donde resulta que são
  - os mais simples de usar
  - os melhores para atingir escalabilidade (número pequeno de *beans* consegue servir um número grande de pedidos)
- Têm poder suficiente para concretizar as tarefas que podem ser realizadas no contexto de uma única chamada de um método
- São definidos por classes anotadas com (`javax.ejb`) **@Stateless**



## **Stateless Session Beans: Exemplo**

```
@Stateless
public class CustomerService {
    /**
     * The business object facade that handles this use case request
     */
    @EJB
    private AddCustomerHandler customerHandler;

    /**
     * Adds a customer given its VAT number, denomination, phone number, ...
     */
    public CustomerDTO addCustomer(int vat, String denomination, int phoneNumber, ...)

    /**
     * Adds a customer given its VAT number, denomination, phone number, ...
     */
    public Collection<DiscountDTO> getDiscounts() throws ApplicationException {...}
}
```



## Stateless Session Beans: Exemplo

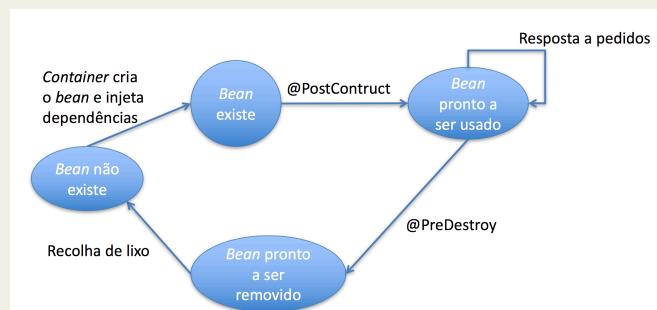
```
@Stateless  
public class CustomerCatalog {
```

- `getCustomer(int) : Customer`
- `addCustomer(int, String, int, Discount) : Customer`
- `getCustomerById(int) : Customer`
- `getCustomers() : Iterable<Customer>`



## Stateless Session Beans: Ciclo de Vida

- O *container* gera uma *pool* de *beans* (objetos) de cada tipo (classe), todos criados por ele
- Sempre que um cliente (aplicação) necessitar de um *bean*, o *container* obtém um da *pool* respetiva ou cria um novo, e entrega-o à aplicação
- Quando o pedido do cliente terminar, o *bean* volta a estar disponível para ser reutilizado em outro qualquer pedido (razão pela qual não se pode usar atributos para guardar estado)



## **Stateless Session Beans: Utilização**

- Para declarar que se pretende utilizar um *bean*:
  - definir um atributo com o tipo do *bean* pretendido e a anotação **@EJB**
- Em tempo de execução, sempre que for preciso, o *container* tratará de fornecer a referência a um *bean* do tipo pretendido
  - o objeto pode ser criado de novo ou pode ser reutilizado da *pool*
- Para a injecção da dependência ser possível é preciso que o objeto que precisa do *bean* tenha sido criado pelo *container*
  - a instanciação dos objetos da classe que declara o atributo foi delegada ao *container*



## **Stateless Session Beans: Exemplo de Utilização**

```
@Stateless
public class AddCustomerHandler implements IAddCustomerHandler

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

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

```
Discount discount = discountCatalog.getDiscount(discountType);
```

Qual o efeito de declararmos um atributo  
**private int counter**



## ***Stateful Session Beans***

- Caracterizam-se por guardarem o estado da conversação com um cliente enquanto a sessão com esse cliente durar
  - os atributos são mantidos entre chamadas sucessivas dos seus métodos na mesma sessão
- São definidos por classes anotadas com **@Stateful**
- As instâncias dos *stateful session beans* são controlados pelos clientes, no sentido em que:
  - são os clientes que determinam quando são criados e destruídos
  - os clientes são donos destas instâncias e portanto não são partilhadas



## ***Stateful Session Beans: Exemplo***

```
@Stateful
public class ProcessSaleHandler implements IProcessSaleHandler {

    @EJB private SaleCatalog saleCatalog;
    ...
    private Sale currentSale;
    ...
}
```

- criar uma instância do *handler (stateless session bean)* para cada instância da transação de negócio
- o atributo *currentSale* manterá o seu valor entre os sucessivos pedidos a esse *handler*
- **cabe ao cliente do handler obter e manter a referência** para essa instância



## ***Stateful Session Beans***

- As instâncias destes *beans*
  - São **criadas** pelo *container* quando um cliente obtém uma referência para uma instância através de
    - injeção de dependência pelo *container* (num outro *stateful bean*)
    - o cliente faz uma procura através do **JNDI** (*Java Naming and Directory Interface*)
  - São **mantidas** pelo *container*
  - São **removidas** a seguir à execução de um método anotado com **@Remove** a pedido do cliente (há ainda a invocação dos métodos anotados com **@PreDestroy**) ou após um **timeout**
- As instâncias dos *stateful session beans* que são mantidas por um largo número de clientes concorrentes pode ter impacto significativo na memória consumida

## ***Stateful Session Beans: Exemplo de Utilização***

```
@WebServlet("/ProcessSale")
public class ProcessSalePageController extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        final HttpSession session = request.getSession();
        ...
    }

    public IProcessSaleHandler lookup() {
        Context context;
        try {
            context = new InitialContext();
            String lookupName =
                "java:app/salesys-business/ProcessSaleHandler!facade.IProcessSaleHandler";
            final IProcessSaleHandler saleHandler =
                (IProcessSaleHandler) context.lookup(lookupName);
            return saleHandler;
        } catch (NamingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return null;
    }
}
```



## JNDI: espaço de nomes

- Cada vez que um *session bean* com os seus interfaces é instalado no container, cada bean/interface fica automaticamente ligado a um **nome JNDI portável**

```
java:<scope>[/<app-name>]/<module-name>/<bean-name>[ !<fully-qualified-interface-name>]
```

- Na consola do servidor aplicacional podem ver

```
INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-2) WFLYEJB0473:  
JNDI bindings for session bean named 'ProcessSaleHandler' in  
deployment unit 'subdeployment "salesys-business.jar" of deployment "salesys-  
ear-1.0.ear"' are as follows:
```

```
...
```

```
java:app:salesys-business/ProcessSaleHandler!facade.IProcessSaleHandler
```

```
...
```



## JNDI: espaço de nomes

- Cada vez que um *session bean* com os seus interfaces é instalado no container, cada bean/interface fica automaticamente ligado a um **nome JNDI portável**

```
java:<scope>[/<app-name>]/<module-name>/<bean-name>[ !<fully-qualified-interface-name>]
```

- Há vários scopes, entre os quais estão **java:global**, **java:app**

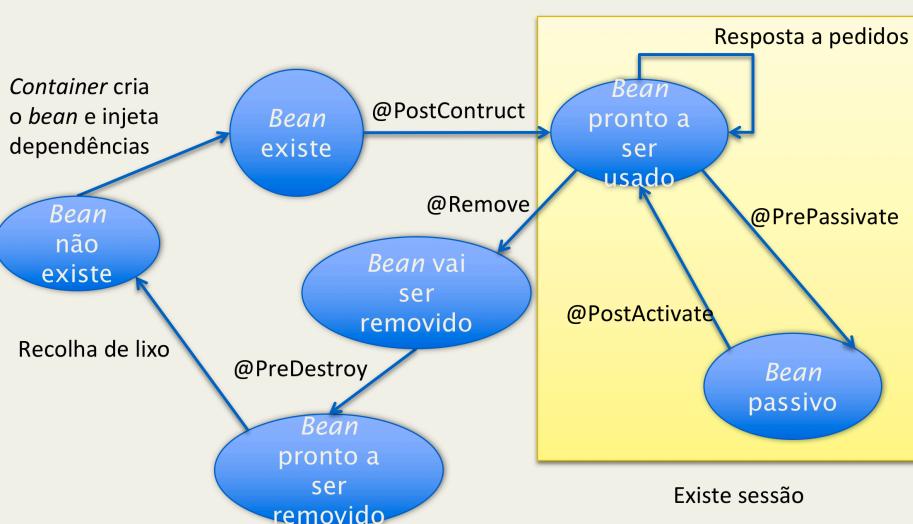
- The **java:app** namespace is used to look up local enterprise beans packaged **within the same application**. That is, the enterprise bean is packaged within an EAR file containing multiple Java EE modules.

```
java:app[/module name]/enterprise bean name[/interface name]
```

## **Stateful Session Beans**

- A técnica da **passivation** endereça este problema
- As instâncias dos *stateful session beans*
  - São **passivated** pelo *container* — temporariamente removidas do ambiente de execução e guardadas em memória secundária — e quando necessários **reactivated**
- Ainda assim
  - no estado destes *beans*, deve ser privilegiada a utilização de objetos “magros” com atributos de tipos primitivos em vez de objetos “gordos” contendo outros objetos
  - as diferenças que existem relativamente aos *stateless session beans* tem consequências em termos de escalabilidade

## **Stateful Session Beans: Ciclo de vida**



## **Session State**

---

- Em geral designa os **dados que precisam de ser acumulados durante uma transação de negócio** mas que não estão ainda prontos para ser persistidos
  - numa loja virtual, o conteúdo do carrinho de compras
- Em certas situações é possível **conceber a transação de negócio** de forma a que **não seja preciso guardar** qualquer informação
- Em certas situações é possível **evitar guardar** qualquer informação

## **Session State**

---

- Em certas situações é possível conceber a transação de negócio de forma a que **não seja preciso guardar** qualquer informação
  - no *SaleSys*, pode-se evitar ter de guardar a venda corrente, modificando o caso de uso e passe a ser fornecido um identificador único na criação de uma venda e enviando este identificador em cada pedido para acrescentar um produto à venda
- Em certas situações é possível **evitar guardar** qualquer informação
  - numa aplicação web, na resposta com os dados sobre um produto que dá a possibilidade de saber mais informação, é codificado que o pedido gerado nesse caso é para o mesmo produto (usando por exemplo o seu identificador único)

## **Session State**

---

- Sendo mesmo preciso ter estado, este pode ser guardado
  - **do lado do servidor** por exemplo em *stateful session beans* e/ou *HTTP session objects*
  - **do lado do cliente** por exemplo em *cookies* (*web apps*) ou em *POJO* (*desktop apps*)
- Não ter de guardar estado do lado do servidor tem várias vantagens
- Mas guardar estado do lado do cliente pode ser impraticável, apresenta alguns desafios e tem custos associados
- Há ainda uma outra possibilidade
  - persistir o estado da sessão usando uma base de dados

## **Session State**

---

- Não ter de guardar estado do **lado do servidor** tem várias vantagens
  - necessários muito menos recursos, com muito menos objetos servem-se mais pedidos
  - escalabilidade mais fácil de conseguir  
*scaling out* — adicionar mais máquinas/instâncias do servidor
  - não é preciso lidar com sessões canceladas
- Guardar estado do **lado do cliente**
  - pode ser impraticável por exigir a transferência de muita informação para o cliente
  - exige ainda ter soluções que garantam a segurança e integridade dos dados transferidos e arcar com os custos

## Distribuição e Acesso aos *Session Beans*

- Diferentes tipos de acesso aos *session beans*
  - por clientes **locais** (que correm dentro da mesma JVM)
  - por clientes **remotos**
- A interação é baseada em
  - **local procedure calls**
  - **remote procedure calls/method invocations**
- O desempenho destes dois tipos de acesso é significativamente diferente
- Assim como a maquinaria envolvida para os fornecer
  - o programador do *session bean* tem de **indicar os acessos que o bean suporta** através da definição de **business interfaces**
  - o programador do cliente indica o **acesso pretendido** escolhendo um *business interface*

## *Session Beans: Interfaces*

- Podem oferecer **business interfaces**
  - definidos por interfaces Java com anotações **@Local @Remote**
  - indicam se as aplicações clientes vão ter de estar a correr na mesma JVM do *session bean* ou não e portanto se a comunicação vai envolver ou não *RMI*
  - podem ser **role-based** selecionando os métodos disponíveis aos clientes, baseados no papel que se pretende que estes desempenhem

```
@Remote  
public interface IAddCustomerHandlerRemote {  
  
    public CustomerDTO addCustomer (int vat, String denomination,  
                                    int phoneNumber, int discountType)  
        throws ApplicationException;  
  
    public Iterable<Discount> getDiscounts() throws ApplicationException;  
}
```

```
@Stateless  
public class AddCustomerHandler implements IAddCustomerHandlerRemote {  
  
    ...
```

## Session Beans: Interfaces

### Consequências da escolha

- os parâmetros e resultados das chamadas **remotas** vão ser **mais isoladas** já que o cliente e o *bean* operam sempre em cópias diferentes dos objetos
  - nas chamadas locais ambos podem modificar o mesmo objeto
- as chamadas **remotas** vão ser **mais lentas** do que as locais (mesmo que os componentes estejam a correr no mesmo nó) pois há muito trabalho extra envolvido
  - interfaces remotos tenderão a ter **parâmetros de grânulo mais grosso**
  - aplicação do padrão **remote facade**
- em interfaces remotos os tipos dos parâmetros e do retorno têm de ser válidos para usar com RMI (**serializáveis**).



## Session Beans: Interfaces

### Como escolher?

- Clientes remotos precisam de interfaces remotos
  - decidir por ter ou não clientes remotos vai depender do tipo de clientes que a aplicação tem de ter
- *Beans* que prestam serviços a outros com quem tem fortes ligações devem oferecer estes serviços através de *business interfaces* locais
- Pode ser necessário fazer a distribuição de componentes por razões de desempenho
  - por exemplo ter as componentes web a correr numa máquina diferente dos ejbs



## **Remote Facade**

---

- A interação baseada em invocações de métodos de grânulo fino que mantém separadas as modificações e as observações (de acordo com o *Command and Query Responsibility Segregation principle*) não é apropriada quando temos código a correr em diferentes máquinas (processos, JVMs)
- A solução proposta por este padrão é ter uma *Remote Facade*
  - Não tem lógica do domínio, apenas traduz chamadas **a métodos de grânulo grosso** em chamadas sobre objetos de grânulo fino
    - exemplo, para obter e modificar informação de diferentes objetos
  - Funciona como *remote gateway* para estes objetos
  - Mas pode ter responsabilidades relacionadas com propriedades não funcionais como propriedades de segurança
- Pode ser desenhada baseada nas necessidades de utilização particulares de certos clientes

## **Data Transfer Objects (DTO)**

---

- A interação com um interface remoto como o *Remote Facade* exige em geral transferir muitos dados quer de entrada quer de saída
  - Múltiplos dados de saída necessitam de ser agrupados
  - Múltiplos dados de entrada espalhados em diferentes parâmetros afeta a legibilidade
  - Grafos de objetos complicados com informação que não é necessária, e pode nem ser serializável, afeta o desempenho e manutenção
- A solução proposta por este padrão é usar **Data Transfer Objects** — objetos que servem exclusivamente para transportar dados dos fornecedores dos serviços para os seus clientes ou vice-versa
  - Desenhados tipicamente em torno das necessidades dos clientes
  - A sua utilização é especialmente recomendada quando é necessário transferir múltiplos dados entre processos, mas são úteis também para limitar as dependências

## Data Transfer Objects (DTO)

- Classes são tipicamente constituídas por pouco mais do que um conjunto de atributos, *getters* e *setters*
  - para agregar informação de vários objetos de forma a que esta possa ser transferida numa única chamada
- Os tipos dos atributos são
  - primitivos ou tipos simples como *String* e *Date*
  - coleções como Listas ou Mapas
  - outros DTO
- Têm de ter a capacidade de se serializar num formato que possa ser transferido
  - binário, formatos textuais XML, JSON, etc
  - o interface **Serializable** do Java serve para marcar as classes que são serializáveis em binário
  - formato depende do tipo de acesso em que vão ser usados

## Data Transfer Objects (DTO)

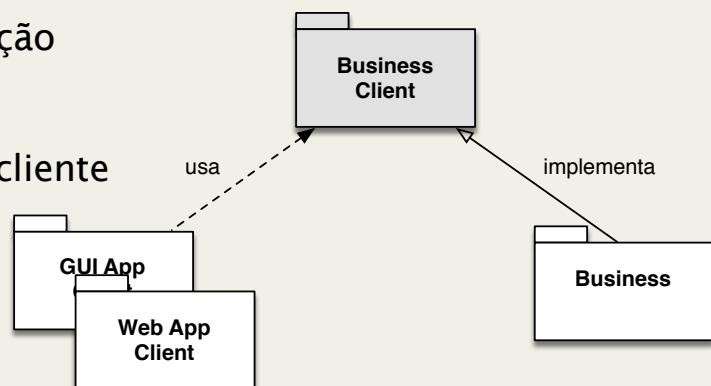
```
@Remote
public interface IAddCustomerHandlerRemote {
    public CustomerDTO addCustomer (int vat, String denomination,
                                    int phoneNumber, int discountType)
                                    throws ApplicationException;
    public Iterable<Discount> getDiscounts() throws ApplicationException;
}
```

```
public class CustomerDTO implements Serializable {
    private static final long serialVersionUID = -4087131153704256744L;
    public final int vatNumber;
    public final String designation;
    public final int id;
    public CustomerDTO(int vatNumber, String designation, int id) {
        this.vatNumber = vatNumber;
        this.designation = designation;
        this.id = id;
    }
}
```



## Organização de uma Aplicação Java EE

- Para que a aplicação possa ser instalada num servidor, responsável pela sua execução, precisa de ter os seus elementos empacotados no **formato enterprise archive**, em ficheiros com a extensão **.ear**
- A aplicação é geralmente composta pelo seguintes **projetos**:
  - Camada de negócio (com EJB)
  - Interface da camada de negócio
  - Camada de apresentação
    - *Web application*
    - Outras aplicações cliente



## Gestão de Entidades (outra vez)

- Anteriormente vimos aplicações em que a gestão das entidades estava totalmente a cargo da aplicação — **application-managed persistence**
  - Criar e fechar *EntityManagerFactory*
  - Criar e fechar *EntityManager*
  - Criar e tratar de *Transações*
- Os *Entity Managers* neste caso são **non-managed** (por oposição a **container-managed**)
  - As aplicações eram autónomas, i.e., não precisavam de ser executadas dentro de um *container*, podiam ser executadas numa simples JVM
- As transações eram **locais** (só havia um recurso transacional) e a gestão das transações era feita também pela aplicação

## Gestão de Entidades em Aplicações em *Containers*

---

- No caso de aplicações Java EE, a gestão das entidades pode ficar a cargo do *container* — ***container-managed persistence***
- A aplicação pode usar vários recursos transacionais e portanto as transações podem ser distribuídas
- O *container* (recorrendo agestor de transações e gestor de recursos) faz a maior parte do trabalho de gestão por nós:
  - Cria objetos ***EntityManagerFactory*** e faz a sua gestão em ***pools*** (tal como acontece com os *beans*)
  - Cria objetos ***EntityManager*** e controla o seu ciclo de vida
  - Faz a ***gestão de transações*** baseada em JTA (Java Transaction API)
- Exige a utilização de
  - ***JTA Transactions***
  - ***JTA-aware data sources***

## Gestão de Entidades em Aplicações em *Containers*

---

- As componentes cujo ciclo de vida é gerido por um *container*
  - por exemplo, EJB, *servlets*, etcpodem aceder a um *EntityManager* através da anotação ***@PersistenceContext*** no atributo com o tipo ***EntityManager*** na classe correspondente
- A instância do ***EntityManager*** é injetada pelo *container*

```
1  ...
2  @Stateless
3  public class CustomerCatalog {
4
5+    * Entity manager for accessing the persistence service ...
6  ...
7  @PersistenceContext
8  private EntityManager em;
9
```



## Gestão de Entidades

```
public class CustomerCatalog {  
    * Entity manager factory for accessing the persistence service  
    private EntityManagerFactory emf;  
  
    public void addCustomer (int vat, String designation, int phoneNumber, Discount discountType)  
        throws ApplicationException {  
        EntityManager em = emf.createEntityManager();  
        try {  
            em.getTransaction().begin();  
            Discount mergedDiscountType = em.merge(discountType);  
            Customer c = new Customer(vat, designation, phoneNumber, mergedDiscountType);  
            em.persist(c);  
            em.getTransaction().commit();  
        } catch (Exception e) {  
            if (em.getTransaction().isActive())  
                em.getTransaction().rollback();  
            throw new ApplicationException("Error adding customer", e);  
        } finally {  
            em.close();  
        }  
    }  
  
    public class CustomerCatalog {  
        * Entity manager for accessing the persistence service  
        @PersistenceContext  
        private EntityManager em;  
  
        @Transactional(Transaction.TxType.REQUIRES_NEW)  
        public Customer addCustomer (int vat, String designation, int phoneNumber, Discount discountType) {  
            Customer customer = new Customer(vat, designation, phoneNumber, discountType);  
            em.persist(customer);  
            return customer;  
        }  
    }  
}
```

## Transações geridas pelo *Container* (CMT)

- Numa aplicação típica os *session beans* estabelecem as **fronteiras das transações**
- Assim, as transações geridas pelo *ejb container* estão **por omissão ativas** para as chamadas a todos os métodos dos *session beans*, não requerendo qualquer configuração
  - **@Stateless**
  - **@TransactionManagement(TransactionManagementType.CONTAINER)**
- O que faz o *ejb container* exatamente?
  - começa uma transação, faz *commit* ou *rollbak* por nós
  - sempre associado à execução de um método de um *bean*
  - de acordo com as anotações usadas para indicar como queremos que sejam geridas as transações

**@Transactional(REQUIRED/REQUIRED\_NEW/SUPPORTS/...)**

## Atributos de transação

Attribute	Description
REQUIRED	This attribute (default value) means that a method must always be invoked within a transaction. The container creates a new transaction if the method is invoked from a nontransactional client. If the client has a transaction context, the business method runs within the client's transaction. You should use REQUIRED if you are making calls that should be managed in a transaction, but you can't assume that the client is calling the method from a transaction context.
REQUIRES_NEW	The container always creates a new transaction before executing a method, regardless of whether the client is executed within a transaction. If the client is running within a transaction, the container suspends that transaction temporarily, creates a second one, commits or rolls it back, and then resumes the first transaction. This means that the success or failure of the second transaction has no effect on the existing client transaction. You should use REQUIRES_NEW when you don't want a rollback to affect the client.
SUPPORTS	The EJB method inherits the client's transaction context. If a transaction context is available, it is used by the method; if not, the container invokes the method with no transaction context. You should use SUPPORTS when you have read-only access to the database table.
MANDATORY	The container requires a transaction before invoking the business method but should not create a new one. If the client has a transaction context, it is propagated; if not, a javax.ejb.EJBTransactionRequiredException is thrown.
NOT_SUPPORTED	The EJB method cannot be invoked in a transaction context. If the client has no transaction context, nothing happens; if it does, the container suspends the client's transaction, invokes the method, and then resumes the transaction when the method returns.
NEVER	The EJB method must not be invoked from a transactional client. If the client is running within a transaction context, the container throws a javax.ejb.EJBException.

## Transações geridas pelo Container

### O que faz o *ejb container* exatamente?

```
@Transactional(TransactionType.REQUIRES_NEW)
public Customer addCustomer (int vat, String designation, int phoneNumber, Discount discountType) {
    Customer customer = new Customer(vat, designation, phoneNumber, discountType);
    em.persist(customer);
    return customer;
}
```

Sempre que é invocado este método, o *container*

- inicia uma nova transação (suspendendo outra que eventualmente existe)
- cria um *PersistenceContext* e associa-o à transação
- executa o corpo do método
- antes de passar o controlo para o cliente termina a transação fazendo *commit* ou *rollback*, analisando o atributo *rollbackonly*
- após o método retornar, resume a transação suspensa (no caso desta existir)
- a transação suspensa não é afetada por eventual *rollback* da transação criada

## Transações geridas pelo Container

---

O que faz o *ejb container* exatamente se não forem usadas quaisquer anotações?

- Quando um cliente invoca um método de um *session bean*, a chamada é interceptada pelo *container* que vai verificar se já há uma transação em progresso
  - se esta não existe, o *container* dá início a uma transação e só depois é que é chamado o método do *session bean*
  - se existe, dá-se a junção a essa transação (**join parent transaction**)
- Quando o método retorna, antes de ser transferido o controlo para o cliente, se foi começada uma transação, esta é terminada com um *commit* ou *rollback*
- As instâncias dos *entity managers* injetadas pelo *container* partilham o mesmo contexto de persistência para uma particular transação

## Transações geridas pelo Container

---

- Quando a transação está pronta para *commit* o contexto de persistência propagará as alterações existentes para a base de dados
- Se durante a execução de um método que temos a certeza ser executado sempre dentro de uma transação são encontrados erros ou problemas ao nível da lógica da aplicação que justifiquem que a transação seja abortada, pode ser invocado o método ***setRollbackOnly*** sobre o contexto da sessão para marcar a transação como sendo para fazer ***rollback***
- O contexto da sessão pode ser obtido do *container* através de injeção

***@Resource***  
***SessionContext context***

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER) ←  
public class OrderManagerBean {  
    @Resource  
    private SessionContext context; ← ② Injects EJB context  
    ...  
    @TransactionAttribute(TransactionAttributeType.REQUIRED)  
    public void placeSnagItOrder(Item item, Customer customer){  
        try {  
            if (!bidsExisting(item)){  
                validateCredit(customer);  
                chargeCustomer(customer, item);  
                removeItemFromBidding(item);  
            }  
        } catch (CreditValidationException cve) {  
            context.setRollbackOnly(); ← ③ Defines Transaction  
        } catch (CreditProcessingException cpe) {  
            context.setRollbackOnly(); ← attribute for method  
        } catch (DatabaseException de) {  
            context.setRollbackOnly();  
        }  
    } ← ④ Rolls back on  
} ← exception
```

## Transações Geridas pelo Container

- É ainda possível associar o lançamento de exceções aplicacionais com a falha de uma transação com a anotação **@ApplicationException(rollback=true)**  
no cabeçalho da classe que define a exceção
- As exceções aplicacionais são sempre passadas ao cliente
- As exceções do sistema (que são embrulhadas como **EJBException**) levam sempre ao *rollback*

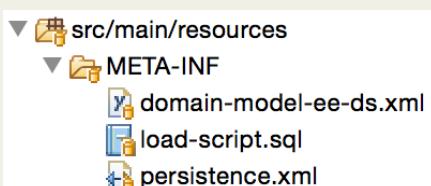
```
...  
@ApplicationException(rollback=true)  
public class CreditValidationException extends Exception { ← ③ Specifies Application-  
    ...  
@ApplicationException(rollback=true)  
public class CreditProcessingException extends Exception { ← Exception  
    ...  
@ApplicationException(rollback=false)  
public class DatabaseException extends  
    RuntimeException { ← ④ Marks RuntimeException  
    ...  
}
```

## Acesso a recursos transacionais

- O tipo de transações e a unidade de persistência continua a ser definida no **persistence.xml**
  - Especifica que vamos usar JTA
  - Indica o nome JNDI do recurso transacional — uma *JTA-aware data source*; este é o nome a ser usado pelo *container*
  - A informação requerida para configurar a *data source* (*driver, url da ligação, user, password, pool size, etc*) são definidos separadamente num outro ficheiro de configuração
- A diretoria ou jar cuja diretoria META-INF inclui o **persistence.xml** é a raiz da unidade de persistência



## Gestão de Entidades em Aplicações em *Containers*



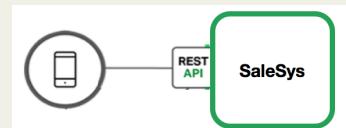
```
<persistence-unit name="domain-model-ee-business" transaction-type="JTA">
    <jta-data-source>java:/jdbc/domain_model_ee_ds</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <shared-cache-mode>NONE</shared-cache-mode>
```

```
<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>
```



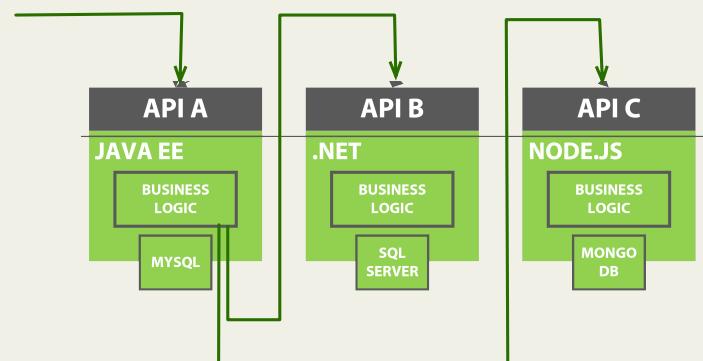
## Web Services (WS)

- Permitem expor na internet, para consumo em outras aplicações, serviços prestados pela camada de negócio
- Essas aplicações tanto podem ser da mesma organização
  - exemplo, componentes que têm apenas responsabilidades de apresentação (por exemplo, cliente para dispositivo móvel)
  - componente web e *backend*
- Como podem ser aplicações de diferentes organizações
  - supply chain app @ manufacturer*
  - company order ws @ supplier*



## Web Services (WS)

- Constituem atualmente um **standard para a integração de aplicações** e negócios (*B2B business-to-business integration*)
- Promovem a **interoperabilidade** — clientes podem correr em plataformas diferentes daquelas em que os serviços estão a correr e estar implementados em diferentes linguagens de programação
- Permite que as aplicações fiquem **fracamente ligados** (*loosely coupled*)



## Web Services

---

- Serviços **SOAP**
  - Baseado no *Simple Object Access Protocol*
  - As mensagens trocadas (sobre HTTP) são documentos XML com vários elementos (*Envelope, Header, Body*)



- Serviços **RESTful**
  - Baseado no estilo arquitetónico *Representational State Transfer*
  - As mensagens trocadas (sobre HTTP) podem ser XML, JSON ou outro formato de hipermédia

## Serviços SOAP

---

- Baseado no *Simple Object Access Protocol*
- As mensagens trocadas (sobre HTTP) são documentos XML com vários elementos
  - *Envelope, Header, Body*
- Juntamente com esquemas XML, define um *framework* de envio de mensagens **fortemente tipado**
- São suportados dois estilos
  - RPC
  - Integração por mensagens (orientado aos documentos)
- Vamos ver apenas o estilo RPC

## Serviços SOAP

- As **operações** que o serviço oferece são explicitamente definidas, juntamente com a estrutura XML do pedido e da resposta de cada operação
  - cada parâmetro de input é associado a um tipo
- Esta descrição é feita recorrendo ao **WSDL (Web Service Definition Language)**
  - há outras coisas (por exemplo, relativas à camada de transporte) que são definidas nesta descrição
  - a descrição wsdl define o **endpoint** do serviço
- Existem várias ferramentas que, a partir do URL em que um serviço publica a sua descrição wsdl, geram o código necessário para programar um cliente que queira interagir com o serviço numa determinada linguagem de programação



## WSDL: Exemplo

```
<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

```
                <types>
                    <schema targetNamespace="http://example.com/stockquote.xsd"
                           xmlns="http://www.w3.org/2000/10/XMLSchema">
                        <element name="TradePriceRequest">
                            <complexType>
                                <all>
                                    <element name="tickerSymbol" type="string"/>
                                </all>
                            </complexType>
                        </element>
                        <element name="TradePrice">
                            <complexType>
                                <all>
                                    <element name="price" type="float"/>
                                </all>
                            </complexType>
                        </element>
                    </schema>
                </types>
```



## Serviços SOAP no Java EE

- A definição de serviços SOAP nas aplicações Java EE é apoiado por duas API:
  - **JAX-B (Java Extra Bindings)**: descreve a ligação entre os dados do pedido/resposta e os objetos Java correspondentes
  - **JAX-WS (Java Extra Web Service)**: permite fazer a definição do serviço (do lado do servidor)



Ciências  
ULisboa | Informática

## Serviços SOAP no Java EE

- Os *stateless session beans* podem oferecer um **service endpoint interface**
- Estes interfaces são definidos com anotações do **javax.jws** directamente na implementação do *session bean* ou num interface que implementem
  - @WebService    @WebMethod**
- Os métodos com anotação **@WebMethod** são os expostos aos clientes do web service (porém se não for usada nenhuma vez a anotação **WebMethod** então todos os métodos públicos são expostos)
- Um cliente pode aceder ao interface *web service* que um *stateless session bean* ofereça através de invocação dos métodos do *service endpoint interface*

## Serviços SOAP no Java EE: Exemplo

```
@Stateless
@WebService
public class AddCustomerHandler implements IAddCustomerHandlerRemote {

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

    * The discount's catalog
    private DiscountCatalog discountCatalog;

    * Adds a new customer with a valid Number. It checks that there is no other ...
    public CustomerDTO addCustomer (int vat, String denomination, ...

    public List<Discount> getDiscounts() throws ApplicationException {
}
```

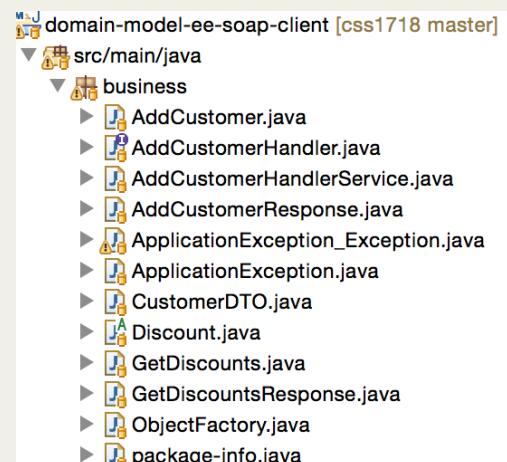


## Serviços SOAP no Java EE: Cliente

```
// Make a service
AddCustomerHandlerService service = new AddCustomerHandlerService();

// Now use the service to get a stub which implements the SDI.
AddCustomerHandler customerHandler = service.getAddCustomerHandlerPort();

// Make the actual call
customerHandler.addCustomer(168027852, "Customer 1", 217500255, 1);
```



## Serviços RESTful

---

- Todo montado em torno de **recursos** que são
  - **identificados** por URIs
    - no caso do HTTP, urls
  - observáveis através das suas **representações**, as quais incluem *metadados* (na forma de pares nome-valor, em que o nome define a estrutura do valor e a sua semântica)
  - manipulados através de um conjunto de **ações** simples e bem definidas
    - no caso do HTTP, os verbos GET, PUT, ...
    - coleção standard de códigos de resposta



## Serviços RESTful

---

- **Invocações**

**GET <http://wine.com/wines/123/reviews/4>**

- verbo HTTP indica a ação a realizar
- caminho identifica o recurso

- **Tipos de caminhos**

- **caminhos de coleções**
  - › <http://wine.com/wines>
  - › <http://wine.com/wines/123/reviews>
  - › <http://wine.com/wines/syrah>
- **caminhos de instância**
  - › <http://wine.com/wines/123>
  - › <http://wine.com/wines/123/reviews/4>



## Serviços RESTful

- show reviews of wine #123
  - › **GET** <http://wines.com/wines/123/reviews>
- create a review of wine #123
  - › **POST** <http://wines.com/wines/123/reviews>
- update review #4 of wine #123
  - › **PUT** <http://wines.com/wines/123/reviews/4>
- delete review #4 of wine #123
  - › **DELETE** <http://wines.com/wines/123/reviews/4>



## Serviços RESTful

- As mensagens trocadas são **representações de recursos**, por exemplo, em XML, JSON ou outro formato de hipermédia
- As representações são sujeitas a negociação, i.e., selecionadas dinamicamente baseadas nas capacidades do emissor e desejos do receptor e a natureza dos dados
  - `text/plain` `text/xml`
  - `application/xml` `application/json`
- Todas as interações são **sem estado**

Content-Type Help

- application/json
- application/octet-stream
- application/x-www-form-urlencoded
- application/xml
- image/jpeg
- image/png
- image/gif
- multipart/form-data
- text/plain

A screenshot of a REST client interface. The top navigation bar includes tabs for Body, Headers, Auth, and Files. The Headers tab is selected. Below it, there is a table with two rows. The first row has 'Header Name' as 'Accept' and 'Header Value' as 'application/json'. The second row has 'Header Name' as 'Content-Type' and 'Header Value' as 'application/json'. To the right of the table are '+' and '-' buttons for adding or removing header entries. The word 'REQUEST' is displayed at the top right of the interface.



## Serviços RESTful

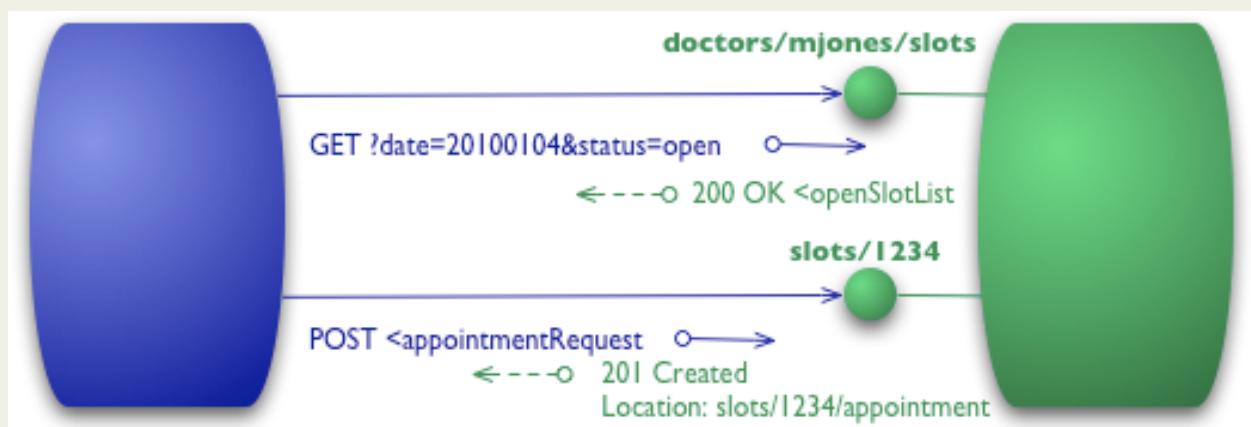
- Baseados no estilo arquitetónico **Representational State Transfer (REST)**
- De acordo com este estilo, estes serviços devem não só dar os dados pedidos como informar o consumidor do serviço sobre quais as operações que este pode fazer

**HATEOAS (Hypermedia as the Engine of Application State)** is a constraint of the REST application architecture. A client interacts with a network application that application servers provide dynamically entirely through hypermedia. A REST client needs no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia..

- O objetivo é promover a **discoverability**, e ter o protocolo a fornecer a sua própria documentação.
- Muitas implementações não seguem esta restrição (aka **REST/Level 3**)

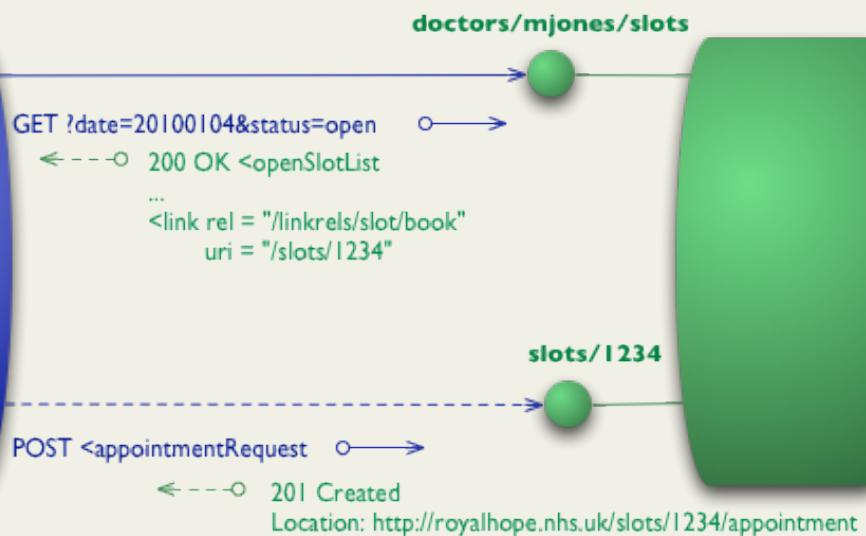


## REST/Level 2



<https://www.martinfowler.com/articles/richardsonMaturityModel.html>

## REST/Level 3



<https://www.martinfowler.com/articles/richardsonMaturityModel.html>

## Serviços REST no Java EE

- A definição de serviços REST nas aplicações Java EE é apoiado pela API:
  - **JAX-RS** (*Java API for RESTful Web Service*) implementada por exemplo por
    - **Jersey** (de referência)
    - **RESTEasy** (JBoss)
- Serviços REST podem ser definidos com anotações do **javax.ws.rs** numa classe POJO ou *stateless session bean*
  - **@Path("...")**   **@Produces("...")**   **@Consumes("...")**
  - **@GET**   **@POST**   **@PathParam("...")** ...
- É criado um objeto por pedido

## Serviços RESTful: Exemplo

```
@Path(Customers.HREF)
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
@Stateless
public class Customers {

    public static final String REL = "customers";
    public static final String HREF = "v1/customers";

    @EJB private ICustomerServiceRemote customerService;
    @Context private UriInfo uriInfo;

    @GET
    @Path("{id}")
    public CustomerGet getCustomer(@PathParam("id") int id) {
        try {
            CustomerDTO c = customerService.getCustomer(id);
            String href = uriInfo.getBaseUri().toString() + HREF + "/" + c.id;
            return new CustomerGet(href, c);
        } catch (ApplicationException e) {
            throw new NotFoundException();
        }
    }
}
```

## Serviços RESTful: Exemplo

```
public class CustomerGet implements Serializable {

    private static final long serialVersionUID = 7243549697187713023L;

    private int id;
    private String href;
    private String designation;
    private int vatNumber;

    public CustomerGet() {}

    public CustomerGet(String href, CustomerDTO customer) {}

    public int getId() {}

    public void setId(int id) {}

    public String getHref() {}

    public void setHref(String href) {}

    public String getDesignation() {}
```



## Serviços RESTful: Exemplo

The screenshot shows the CocoaRestClient interface. At the top, the URL is set to `http://localhost:8080/domain-model-ee-rest/rest/v1/customers/1` and the Method is set to `GET`. Below the URL input, there are tabs for `Body`, `Headers`, `Auth`, and `Files`. The `Body` tab is selected, showing a single row with the value `1`. Below the body area are three radio buttons: `Raw Input` (selected), `Field Input/Multipart File Upload`, and `Raw File Input`. The `REQUEST` section is empty. In the `RESPONSE` section, the `Body` tab is selected, displaying the following JSON response:

```
1 {  
2   "id": 1,  
3   "href": "http://localhost:8080/domain-model-ee-rest/rest/v1/customers/1",  
4   "designation": "antonia",  
5   "vatNumber": 197672337  
6 }
```

## Serviços RESTful: Exemplo

```
@POST  
public Response addCustomer(@Valid CustomerData customerData) throws URISyntaxException {  
    try {  
        CustomerDTO customer = customerService.addCustomer(customerData.getVAT(),  
            customerData.getDenomination(), customerData.getPhoneNumber(),  
            customerData.discountType());  
        String href = uriInfo.getBaseUri().toString() + HREF + "/" + customer.id;  
        return Response.created(new URI(href)).build();  
    } catch (ApplicationException e) {  
        return Response.status(Status.PRECONDITION_FAILED).build();  
    }  
}  
  
@GET  
public Response getCustomers() {  
    CustomerList res = new CustomerList();  
    for (CustomerDTO customer : customerService.getCustomers()) {  
        String href = uriInfo.getBaseUri().toString() + HREF + "/" + customer.id;  
        res.addUser(new CustomerGet(href, customer));  
    }  
    res.setMeta(new CollectionMeta(res.size(), null, null, 1, res.size()));  
    return Response.ok(res).build();  
}
```



## Serviços RESTful: Exemplo

```
curl -X POST "http://localhost:8080/domain-model-ee-rest/rest/v1/customers" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"phoneNumber\":123123123,\"vat\":179422561,\"discountType\":2,\"denomination\":\"alcides\"}"
```

```
> POST /domain-model-ee-rest/rest/v1/customers HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.43.0
> accept: application/json
> Content-Type: application/json
> Content-Length: 84
>
* upload completely sent off: 84 out of 84 bytes
< HTTP/1.1 201 Created
< Connection: keep-alive
< X-Powered-By: Undertow/1
< Server: WildFly/10
< Location: http://localhost:8080/domain-model-ee-rest/rest/v1/customers/2
< Content-Length: 0
< Date: Thu, 24 May 2018 14:10:56 GMT
```



Ciências  
ULisboa

The screenshot shows the CocoaRestClient interface. At the top, there's a header bar with the title 'CocoaRestClient'. Below it, a search bar contains the URL 'http://localhost:8080/domain-model-ee-rest/rest/v1/customers' and a dropdown for 'Method' set to 'GET'. To the right of the method is a 'Submit' button.

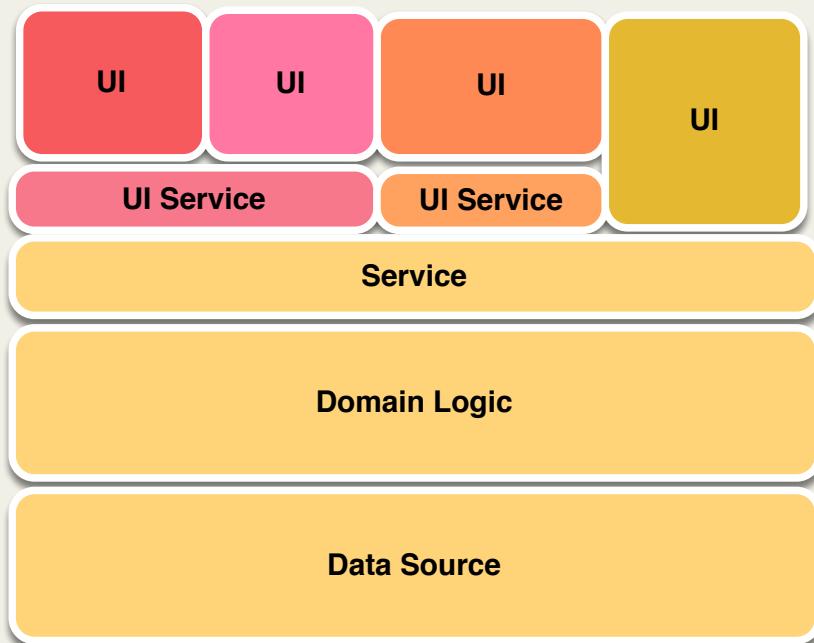
The main area is divided into two sections: 'REQUEST' and 'RESPONSE'.

In the 'REQUEST' section, there are tabs for 'Body', 'Headers', 'Auth', and 'Files'. The 'Body' tab is selected, showing a single line of text: '1'. Below this, there are three radio buttons: 'Raw Input' (selected), 'Field Input/Multipart File Upload', and 'Raw File Input'.

In the 'RESPONSE' section, there are tabs for 'Body', 'Headers (200)', and 'Sent Headers'. The 'Body' tab is selected, displaying a JSON response:

```
1 {
2   "meta": {
3     "pageSize": 2,
4     "nextPage": null,
5     "previousPage": null,
6     "currentPageNumber": 1,
7     "totalCount": 2
8   },
9   "customers": [
10    {
11      "id": 1,
12      "href": "http://localhost:8080/domain-model-ee-rest/rest/v1/customers/1",
13      "designation": "antonia",
14      "vatNumber": 197672337
15    },
16    {
17      "id": 2,
18      "href": "http://localhost:8080/domain-model-ee-rest/rest/v1/customers/2",
19      "designation": "alcides",
20      "vatNumber": 179422561
21    }
22  ]
23 }
```

## Web Services:



Ciências  
ULisboa

## Objetivos de CSS em Recapitação

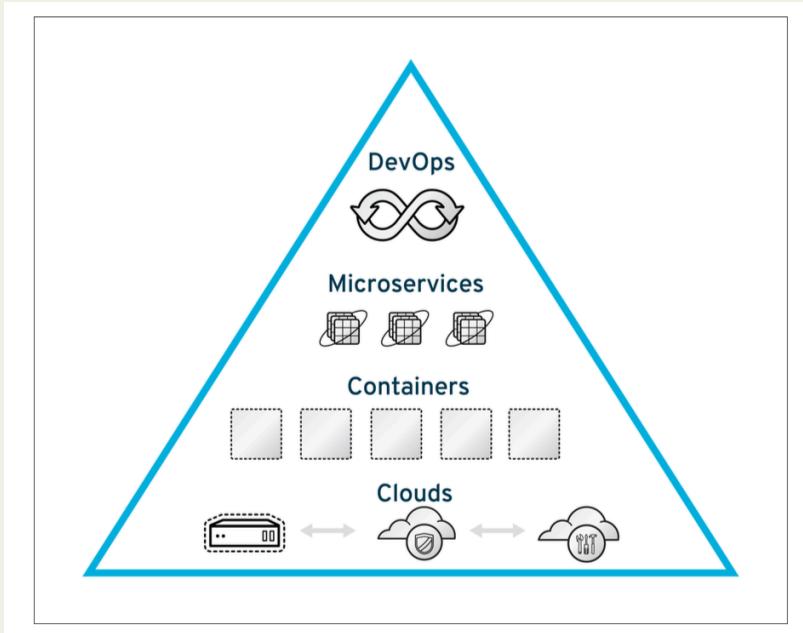
- Adquirir competências no desenho e implementação de sistemas empresariais através da **aplicação de padrões**, que traduzem as **boas práticas** de arquitetura e desenho destes sistemas.
  - Na prática isto implica ganhar competências no desenvolvimento de sistemas
    - concorrentes e distribuídos,
    - construídos a partir de componentes locais ou distribuídas pela Web
    - que utilizem servidores web, servidores aplicacionais e servidores de base de dados
- e em particular na utilização de *frameworks* de componentes atualmente usados na construção destes sistemas.



Ciências  
ULisboa

## ***Modern Enterprise Application Development***

<https://www.oreilly.com/ideas/modern-java-ee-design-patterns>



*Figure 2-3. The pyramid of modern enterprise application development*