HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

03 - Business tier

EJBs, managed objects, AOP, dependency injection, object pooling

AMT 2020

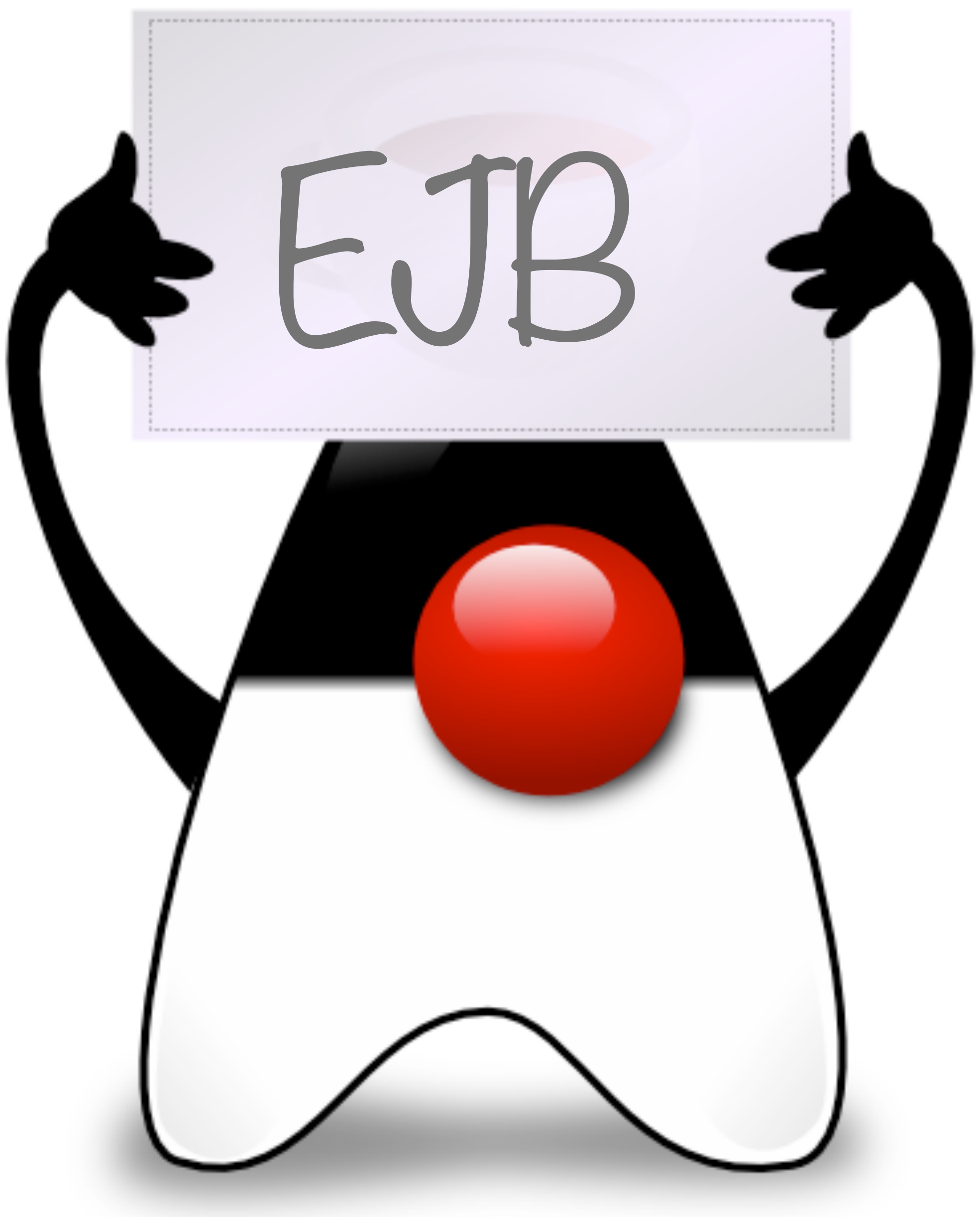
Olivier Liechti

Dependency Injection

The business tier: EJBs as an example of "managed components"

Aspect Oriented Programming

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch



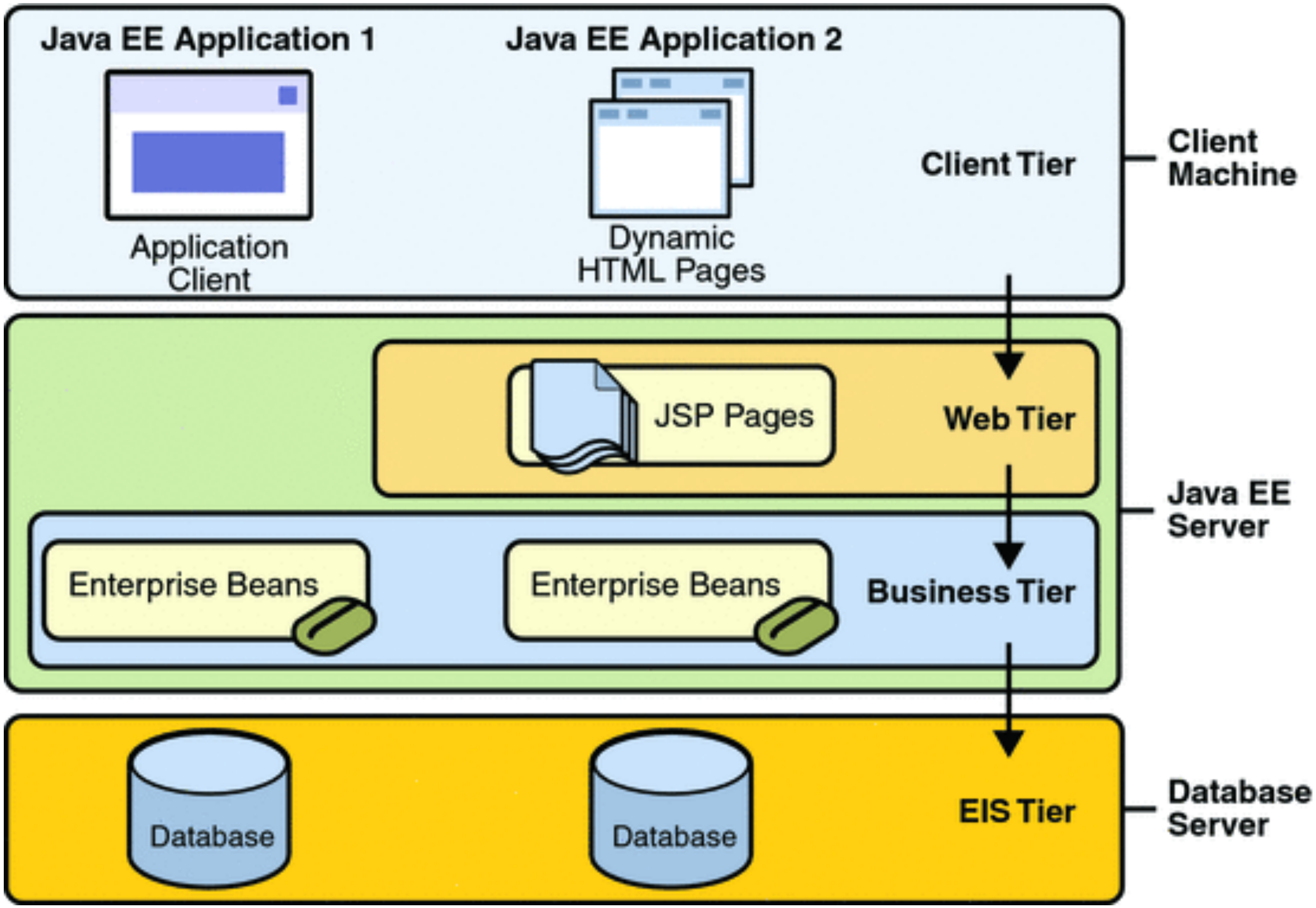
Business Services & EJB

Services in a Java EE application

- Last week, we implemented a very simple Java EE application.
- When we implemented the MVC pattern, we implemented a service as a **Plain Old Java Object** (POJO).
- **The POJO was not a managed component**. We created the instance(s) of the service (*in the web container*).
- This week, we will see an **alternative solution** for implementing Java EE services: Enterprise Java Beans (EJBs).

What is the best way to implement services, POJOs or EJBs? **There is not a single right answer to this question!** There are pros and cons in both approach.

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch



http://java.sun.com/javaee/5/docs/tutorial/doc/bnaay.html

What is an **Enterprise Java Bean** (EJB)?

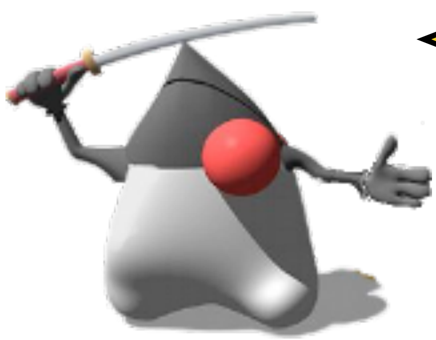
- An EJB is a **managed component**, which implements **business logic** *in a UI agnostic way*.
- The EJB container manages the **lifecycle** of the EJB instances.
- The EJB container also **mediates the access** from clients (i.e. it is an “invisible” intermediary) to EJBs. This is a form of Aspect Oriented Programming (AOP):
- This allows the EJB container to perform technical operations (especially related to **transactions** and **security**) when EJBs are invoked by clients.
- The EJB container manages a **pool** of EJB instances.
- Note: the EJB 3.2 API is **specified is JSR 345**.

What are the **4 types** of EJBs used today?

- **Stateless Session Beans** are used to implement business services, where every client request is independent.
- **Stateful Session Beans** are used for services which have a notion of conversation (e.g. shopping cart).
- **Singleton Session Beans** are used when there should be a single service instance in the app.
- **Message Driven Beans** are used together with the Java Message Service (JMS). Business logic is not invoked when a web client sends a request, but when a message arrives in a queue. We will see that later.

When you implement a stateful application in Java EE, **you have the choice to store the state in different places**. One option is to do it in the web tier (in the HTTP session). Another option is to use **Stateful Session Beans**. Many (most) developers use HTTP sessions.

In older versions of Java EE (before Java EE 5), there was another type of EJBs: **Entity Beans**.



Entity Beans were used for **accessing the database**. They were a nightmare to use and raised a number of issues. You might find them in legacy applications.

Entity Beans (as a legacy type of EJB) are **not the same thing** as **JPA Entities**, which are now widely used!

A first example


```
package ch.heigvd.amt.lab1.services;
import javax.ejb.Local;

@Local
public interface CollectorServiceLocal {

  void submitMeasure(Measure measure);

}
```

These **annotations** are processed by the application server at **deployment time**.

```
package ch.heigvd.amt.lab1.services;
import javax.ejb.Stateless;

@Stateless
public class CollectorService implements CollectorServiceLocal {

  @Override
  public void submitMeasure(Measure measure) {
    // do something with the measure (process, archive, etc.)
  }

}
```

They are an **declaration** that the service must be handled as a **managed component**!

How does a “client” find and use an EJB?

- By “**client**”, we refer to a **Java component** that wants to get a reference to the EJB and invoke its methods.
- In many cases, the client is a **servlet** or **another EJB** (i.e. a service that delegates part of the work to another service).
- The application server is providing a **naming and directory service** for managed components. Think of it as a “white pages” service that keeps track of component names and references.
- Remember that we mentioned **Dependency Injection** earlier today?

The Java Naming and Directory Interface (JNDI) provides an API to access directory services. It can be used to access an LDAP server. It can also be used to lookup components in a Java EE server.

The **first method** to find an EJB is to do an **explicit lookup**, with JNDI.

```
@WebServlet(name = "FrontController", urlPatterns = {"/FrontController"})
public class FrontController extends HttpServlet {

  private CollectorServiceLocal collectorService;

  @Override
  public void init() throws ServletException {
    super.init();
    try {
     Context ctx = new InitialContext();
     collectorService = (CollectorServiceLocal) ctx.lookup("java:module/CollectorService");
    } catch (NamingException ex) {
      Logger.getLogger(FrontController.class.getName()).log(Level.SEVERE, null, ex);
    }
  }
```

This gives me access to the app server's naming service

I am using the app server's naming service

Warning! These 2 JNDI operations are **costly** (performance-wise). You don't want to re-execute them for every single HTTP request!!!!
It is much better to do it once and to **cache the references** to the services.

The **second method** is to ask the app server to **inject a dependency** to the service.

```
@WebServlet(name = "FrontController", urlPatterns = {"/FrontController"})
public class FrontController extends HttpServlet {

  @EJB
  private CollectorServiceLocal collectorService;

}
```

Ne pas l'instancier !!!

With the @EJB annotation, **I am declaring a dependency** from between my servlet and my service. The servlet *uses* the service.

With the @EJB annotation, I am also giving instructions to the app server. The servlet and the service are **managed components**. When the app server instantiates the servlet, it **injects a value** into the **collectorService** variable.

Example


```
@Singleton
public class BeersDataStore implements BeersDataStoreLocal {

 private final List<Beer> catalog = new LinkedList<>();

 public BeersDataStore() {
    catalog.add(new Beer("Cardinal", " Feldschlösschen", "Switzerland", "Lager"));
    catalog.add(new Beer("Punk IPA", " BrewDog", "Scotland", "India Pale Ale"));
 }
...
}
```

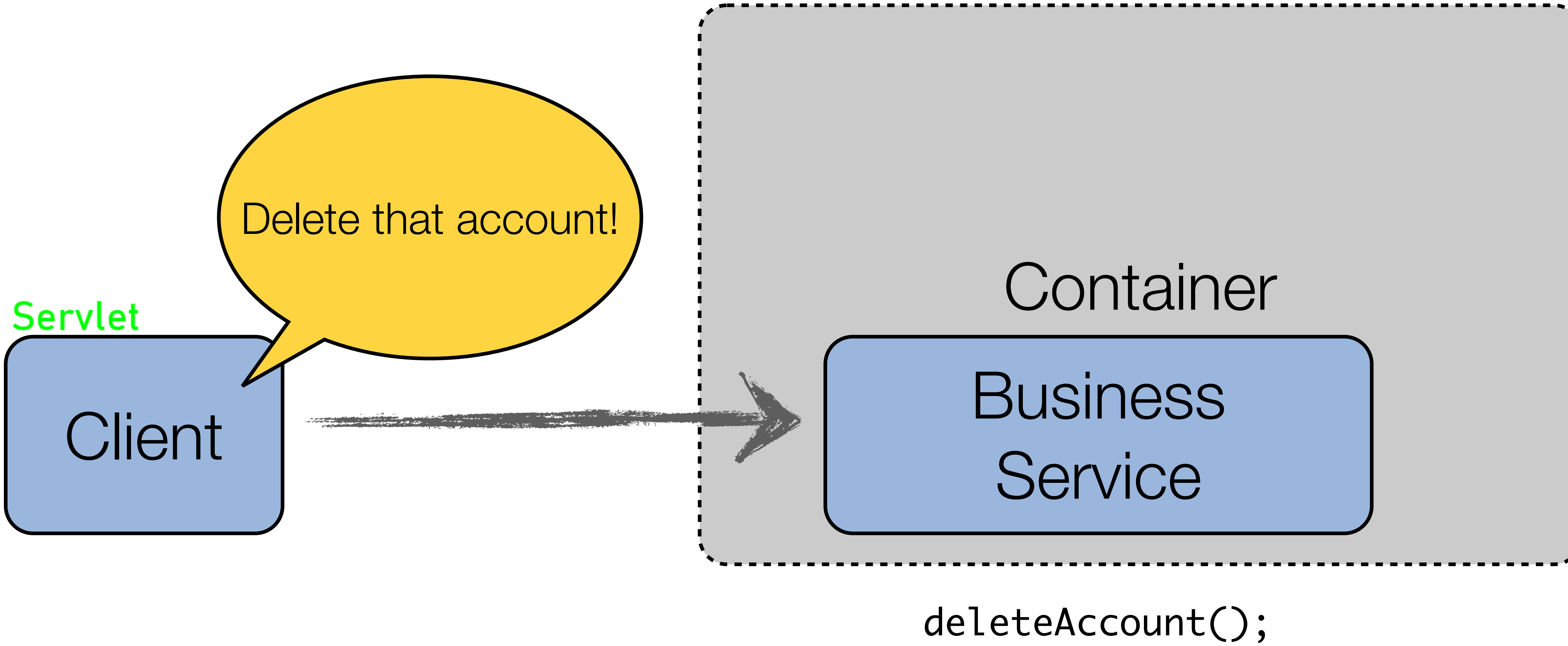
```
@Stateless
public class BeersManager implements BeersManagerLocal {

  @EJB
  BeersDataStoreLocal beersDataStore;

  @Override
  public List<Beer> getAllBeers() {
    simulateDatabaseDelay();
    return beersDataStore.getAllBeers();
  }
...
}
```

Example

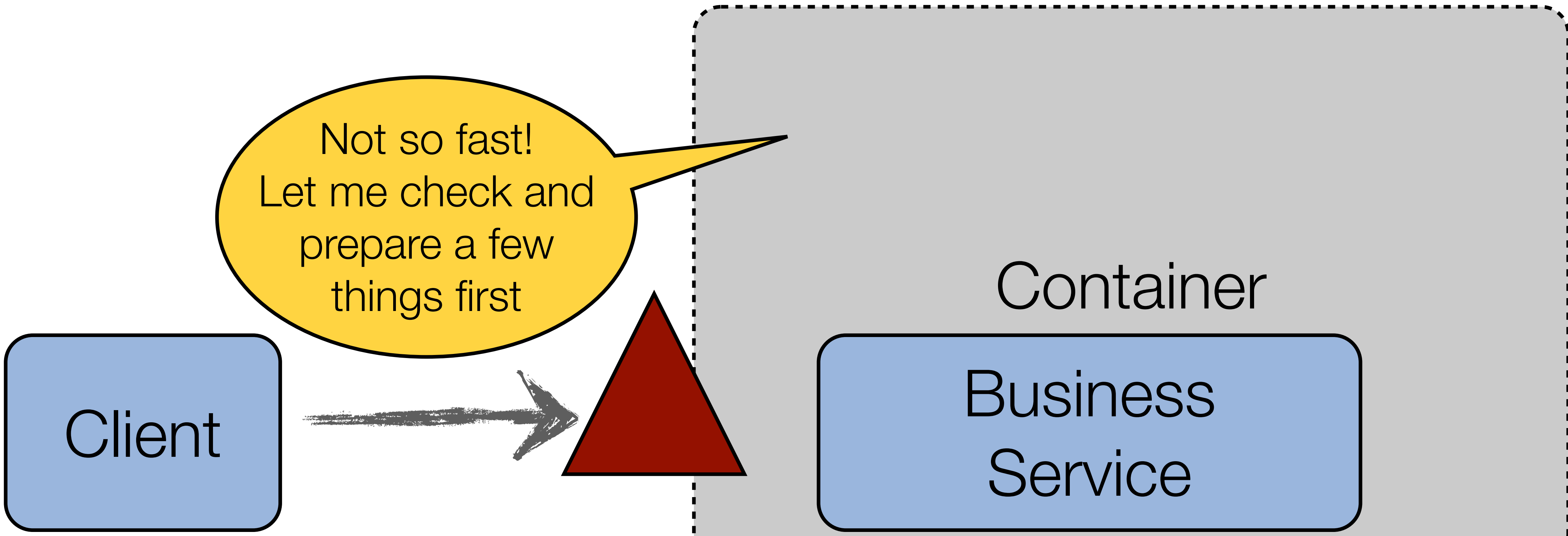

```
public class BeersServlet extends HttpServlet {

  @EJB
  BeersManagerLocal beersManager;

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
    /*
     Firstly, we need to get a model. It is not the responsibility of the servlet
     to build the model. In other words, you should avoid to put business logic
     and database access code directly in the controller. In this example, the
     beersManager takes care of the model construction.
     */
    Object model = beersManager.getAllBeers();
    ...
   }
   ...
}
```

The app server **mediates** the access between clients and EJBs. What does it mean?

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch



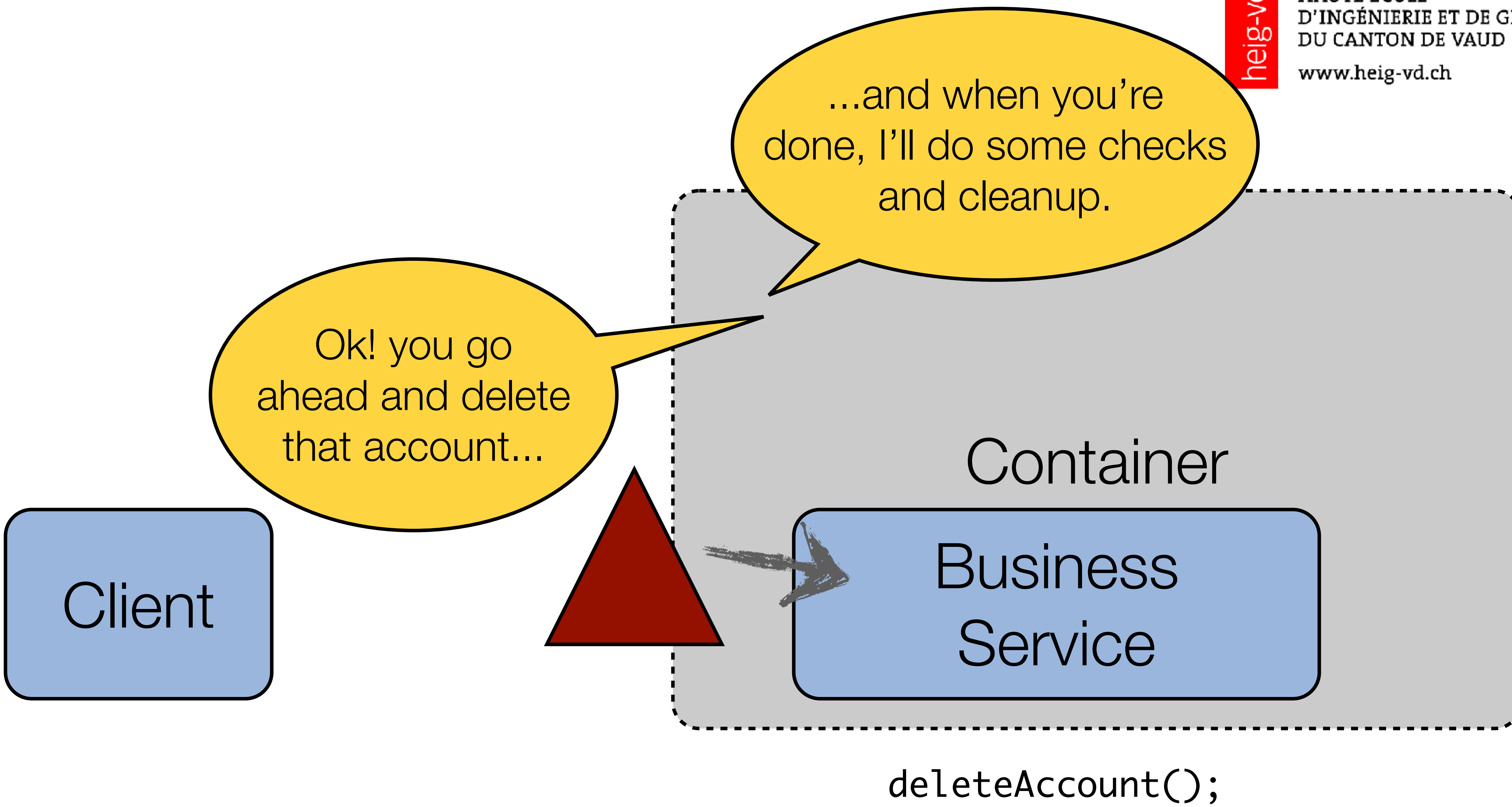
The business service, implemented as a Stateless Session Bean, is a **managed component**.
The client ***thinks*** that he has a direct reference to a Java object.
He is ***wrong***.



deleteAccount();

In reality, when the client invokes the deleteAccount() methods, the call is going **through the container**.
The container is in a position to **perform various tasks** (security checks, transaction demarcation, etc.)

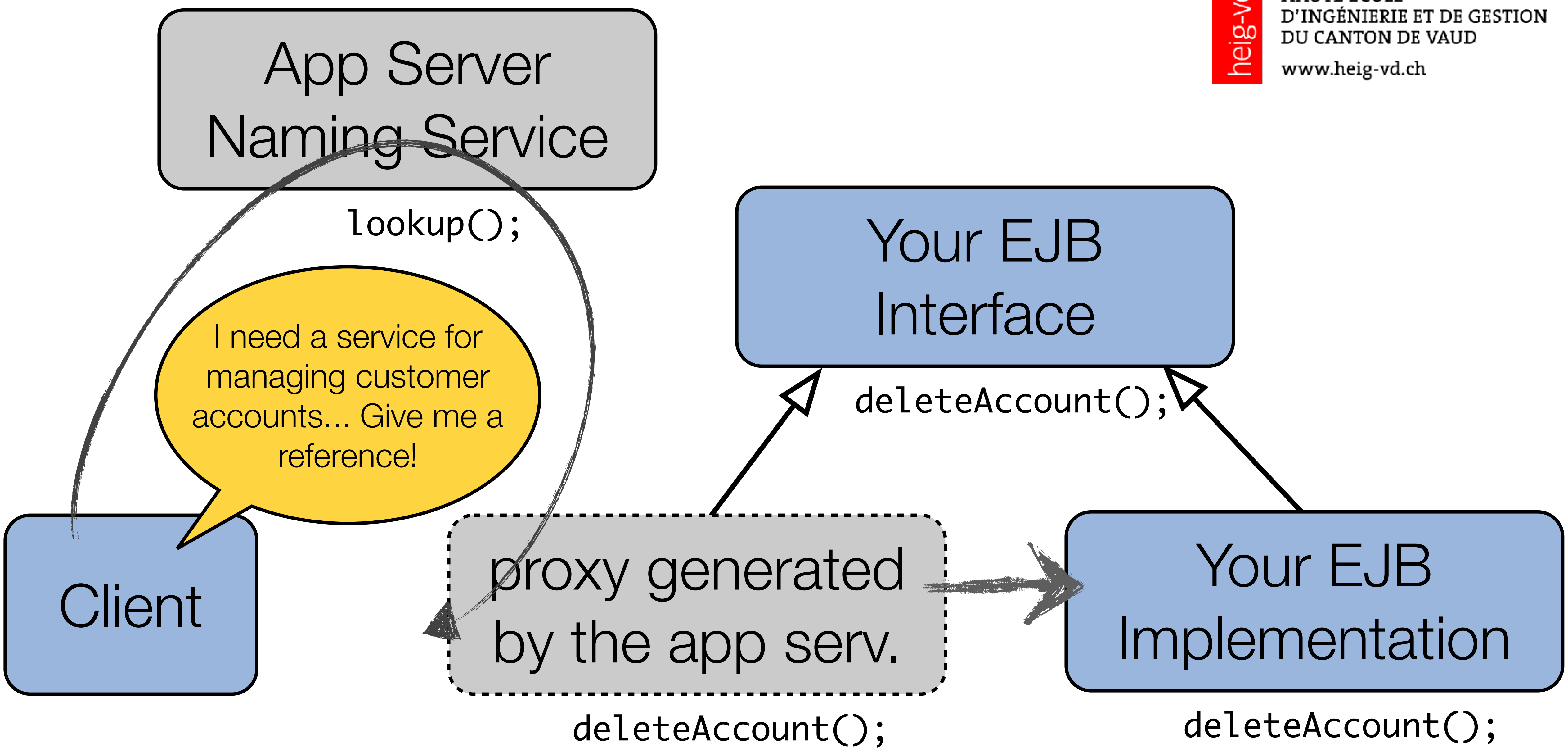
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch



When done, the container can forward the method call to the business service (your implementation).
On the way back, the response also goes back **via the container**.

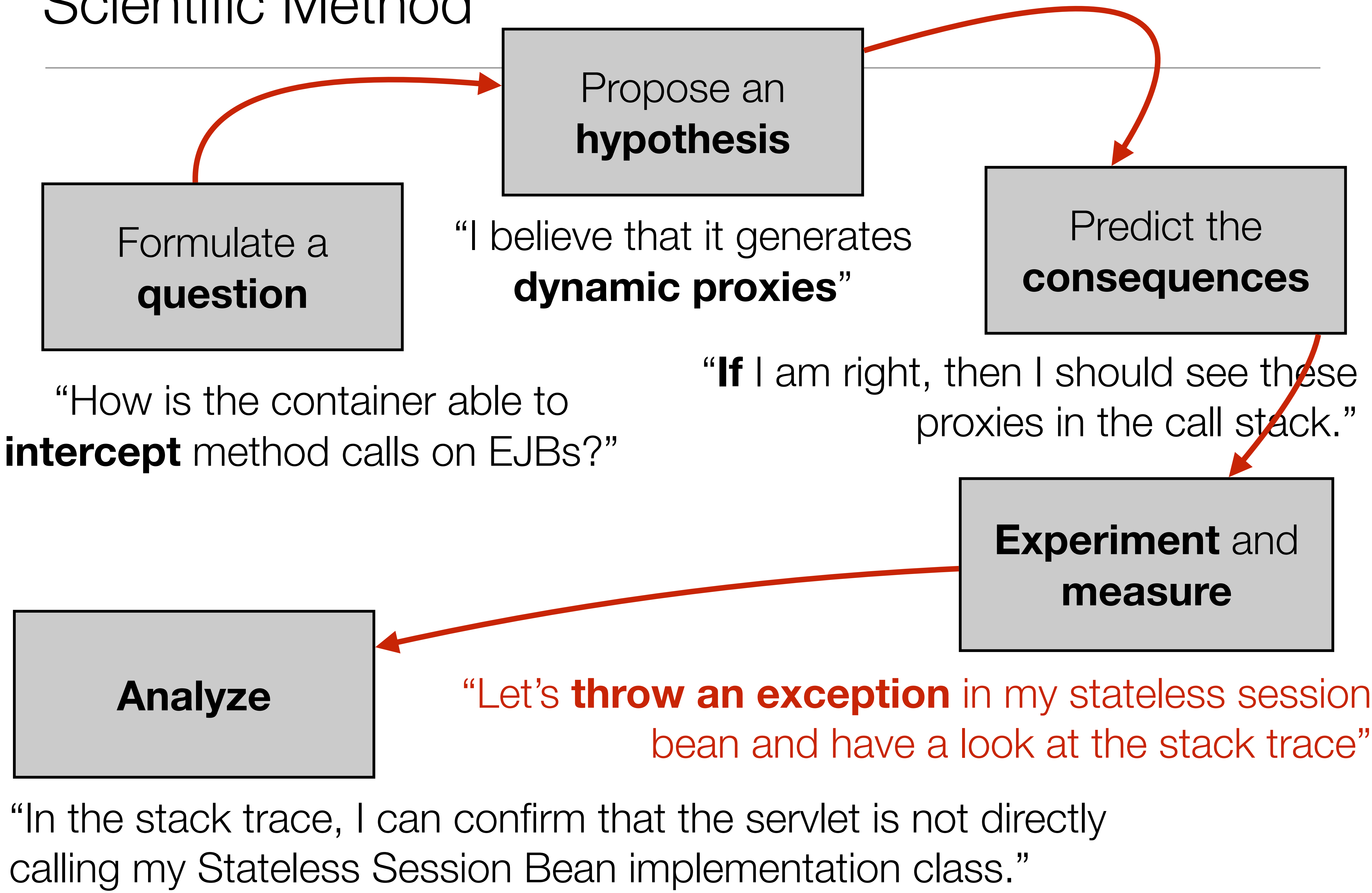
How is that possible?
How does it work?

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch



Your service implementation implements your interface.
The container dynamically generates a class, which implements the same interface. This class performs the technical tasks and invokes your class (proxy).

Scientific Method

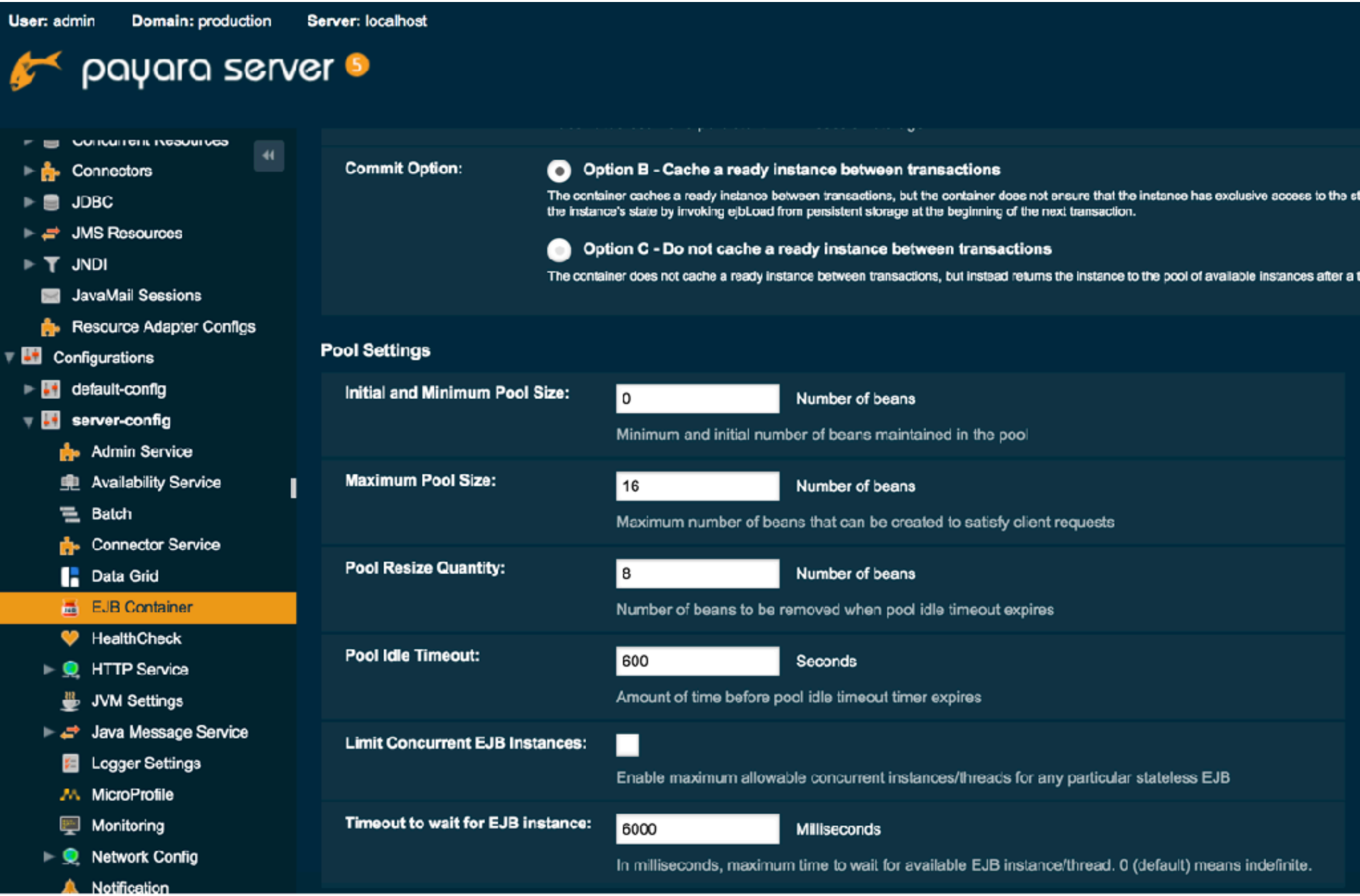


```
Caused by: java.lang.RuntimeException: just kidding
	at ch.heigvd.amt.lab1.services.CollectorService.submitMeasure(CollectorService.java:15)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:483)
	at org.glassfish.ejb.security.application.EJBSecurityManager.runMethod(EJBSecurityManager.java:1081)
	at org.glassfish.ejb.security.application.EJBSecurityManager.invoke(EJBSecurityManager.java:1153)
	at com.sun.ejb.containers.BaseContainer.invokeBeanMethod(BaseContainer.java:4786)
	at com.sun.ejb.EjbInvocation.invokeBeanMethod(EjbInvocation.java:656)
	at com.sun.ejb.containers.interceptors.AroundInvokeChainImpl.invokeNext(InterceptorManager.java:822)
	at com.sun.ejb.EjbInvocation.proceed(EjbInvocation.java:608)
	at 
org.jboss.weld.ejb.AbstractEJBRequestScopeActivationInterceptor.aroundInvoke(AbstractEJBRequestScopeActivationInter
ceptor.java:46)
	at org.jboss.weld.ejb.SessionBeanInterceptor.aroundInvoke(SessionBeanInterceptor.java:52)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:483)
	at com.sun.ejb.containers.i
	at com.sun.ejb.containers.i
	at com.sun.ejb.EjbInvocatic
	at com.sun.ejb.containers.i
	at com.sun.ejb.containers.i
	at sun.reflect.NativeMethoc
	at sun.reflect.NativeMethoc
	at sun.reflect.DelegatingMe
	at java.lang.reflect.Methoc
	at com.sun.ejb.containers.i
	at com.sun.ejb.containers.i
	at com.sun.ejb.containers.interceptors.InterceptorManager.intercept(InterceptorManager.java:369)
	at com.sun.ejb.containers.BaseContainer.__intercept(BaseContainer.java:4758)
	at com.sun.ejb.containers.BaseContainer.intercept(BaseContainer.java:4746)
	at com.sun.ejb.containers.EJBLocalObjectInvocationHandler.invoke(EJBLocalObjectInvocationHandler.java:212)
	... 34 more
```

```
@Stateless
public class CollectorService implements CollectorServiceLocal {

  @Override
  public void submitMeasure(Measure measure) {
    throw new RuntimeException("just kidding");
  }


}
```

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

The AMT book talks about pooling... what does it mean and why is it useful?


User: admin
Domain: production
Server: localhost
payara server 5
Connectors
JDBC
JMS Resources
JNDI
JavaMail Sessions
Resource Adapter Configs
Configurations
default-config
server-config
Admin Service
Availability Service
Batch
Connector Service
Data Grid
EJB Container
HealthCheck
HTTP Service
JVM Settings
Java Message Service
Logger Settings
MicroProfile
Monitoring
Network Config
Notification
Commit Option:
Option B - Cache a ready instance between transactions
The container caches a ready instance between transactions, but the container does not ensure that the instance has exclusive access to the st
the instance's state by invoking ejbLoad from persistent storage at the beginning of the next transaction.
Option C - Do not cache a ready instance between transactions
The container does not cache a ready instance between transactions, but instead returns the instance to the pool of available instances after a t
Pool Settings
Initial and Minimum Pool Size:
0
Number of beans
Minimum and initial number of beans maintained in the pool
Maximum Pool Size:
16
Number of beans
Maximum number of beans that can be created to satisfy client requests
Pool Resize Quantity:
8
Number of beans
Number of beans to be removed when pool idle timeout expires
Pool Idle Timeout:
600
Seconds
Amount of time before pool idle timeout timer expires
Limit Concurrent EJB Instances:
Enable maximum allowable concurrent instances/threads for any particular stateless EJB
Timeout to wait for EJB instance:
6000
Milliseconds
In milliseconds, maximum time to wait for available EJB instance/thread. 0 (default) means indefinite.

Why pool objects?

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

There are 2 main reasons for pooling objects

- To increase performance. Some objects take a long time to be created and initialized (e.g. DB connection object). It's better to reuse objects instead of throwing them away and recreating them.
- To set a limit on resource consumption (CPU, RAM). Under heavy load, we decide how many requests we process at the same time. It's better to have clients wait bit than to exhaust all server resources.

Scientific Method

Formulate a **question**

“How is the container able to **intercept** method calls on EJBs?”

Propose an **hypothesis**

“I believe that it generates **dynamic proxies**”

Predict the **consequences**

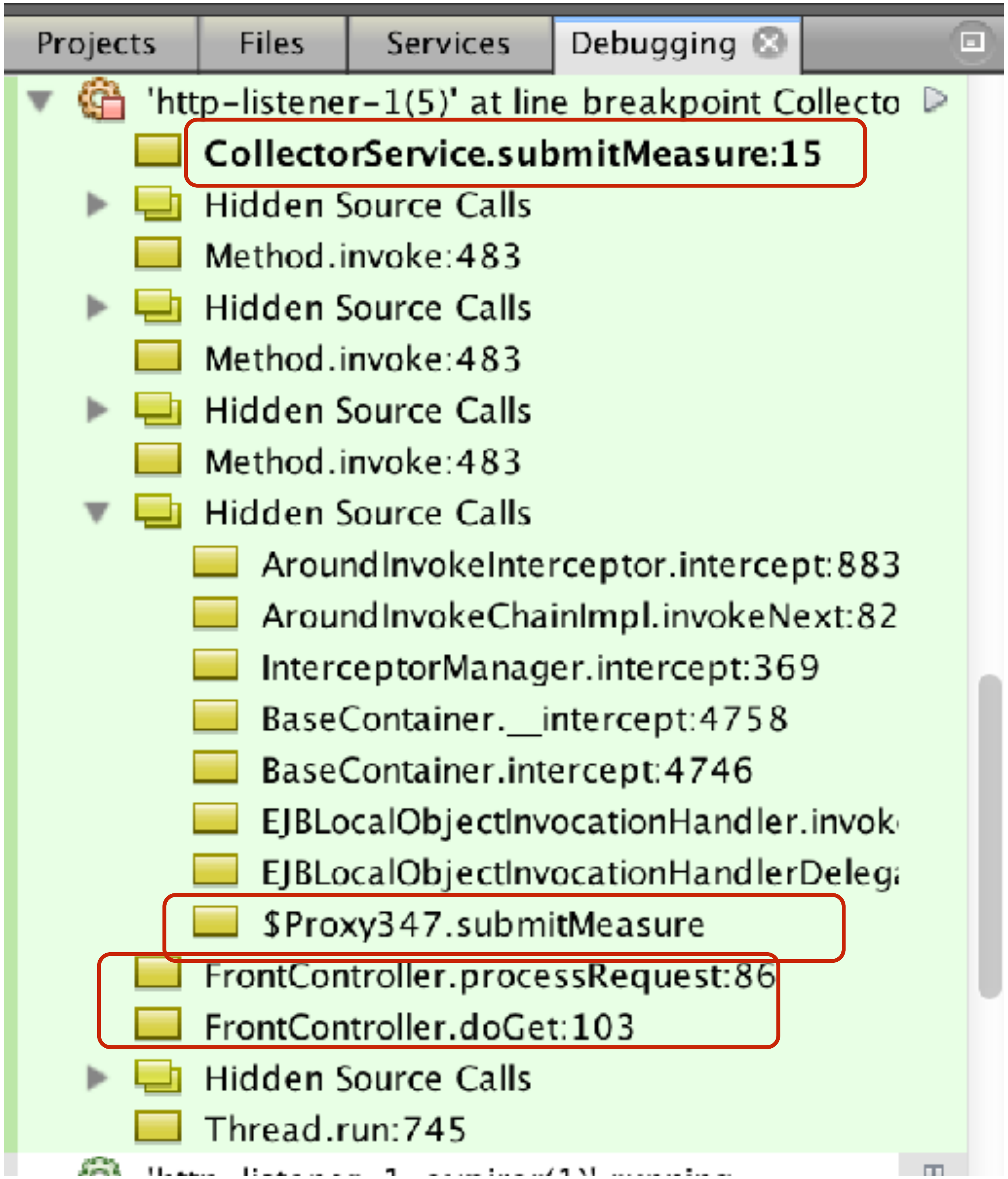
“**If** I am right, then I should see these proxies in the call stack.”

Experiment and **measure**

“Let’s **inspect the call stack in the debugger**”

Analyze

“In the debugger console, I can confirm that there are container generated classes between my servlet and my service implementation.”



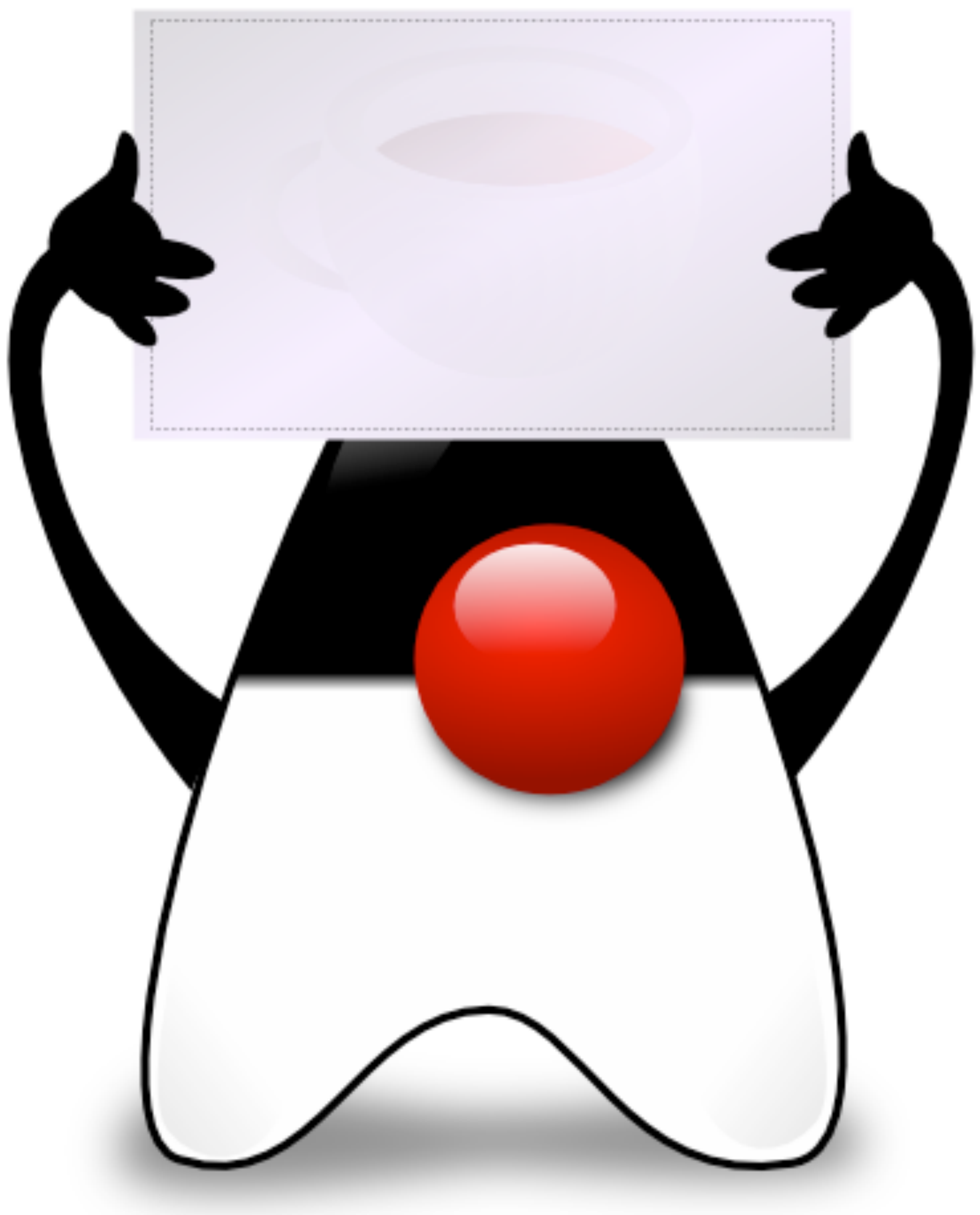
At some point, the method call is forwarded to my implementation.

The reference actually points to a proxy generated by the container. The container performs tasks that are visible in a **long call stack**!

My servlet invokes the method on its **reference** to the EJB.

*An HTTP request has arrived; GF invokes the doGet callback on my servlet (**IoC**). GF has also **injected** a **reference** to the EJB into the servlet.*

heig-vd HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch



Aspect Oriented Programming (AOP)

Aspect Oriented Programming (AOP)

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- In all applications, there are “things” that need to be done over and over and that are **orthogonal** to **business logic**.
- Examples:
  - Logging and auditing
  - Security checks (authorization)
  - Transaction management
- In traditional object-oriented design, the common approach is to implement the pure business logic and these orthogonal functions **at the same place** (in class methods).

***Separation of concerns**: business logic vs. other “aspects”*

AOP Frameworks

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

- AspectJ created at Xerox PARC in 2001 (Gregor Kiczales)
- Several other frameworks and projects have been developed (e.g. AspectWerkz), for different languages.
- Java EE was built to achieve the goal of AOP (separation of concerns). It makes the concepts and terminology visible with Interceptors (but to some extent also with EJBs)
- The Spring Framework makes it possible to use AOP concepts and relies itself on AOP for some of its features.

aspectj crosscutting objects for better modularity

http://eclipse.org/aspectj/

Aspect Oriented Programming (AOP)

- Where is my business logic? It's hard to find... What do I have to bother with all these infrastructure concerns?
- How can I get a global view for security management in my application?
- What if I need to change the way I do the auditing? I will have to go in every single method...
- *What a nightmare!!*

<<class>>
ProductManager

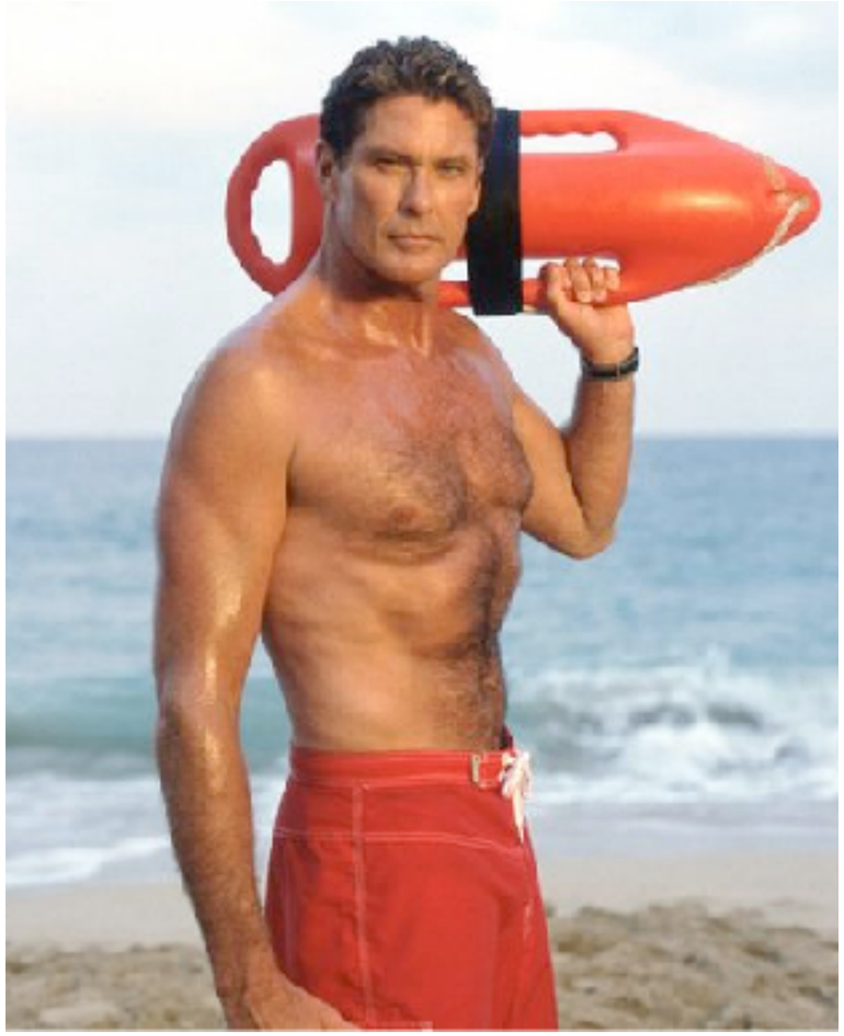
```
public void addProduct(Product p) {
  // check if the user is authenticated and authorized
  ...
  // start transaction
  ...
  // finally, some business logic
  ...
  // commit transaction
  ...
  // leave a trace in the audit trail
  ...
}
```

```
public void removeProdcut(Product p) {
  // check if the user is authenticated and authorized
  ...
  // start transaction
  ...
  // finally, some business logic
  ...
  // commit transaction
  ...
  // leave a trace in the audit trail
  ...
}
```

AOP to the rescue

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

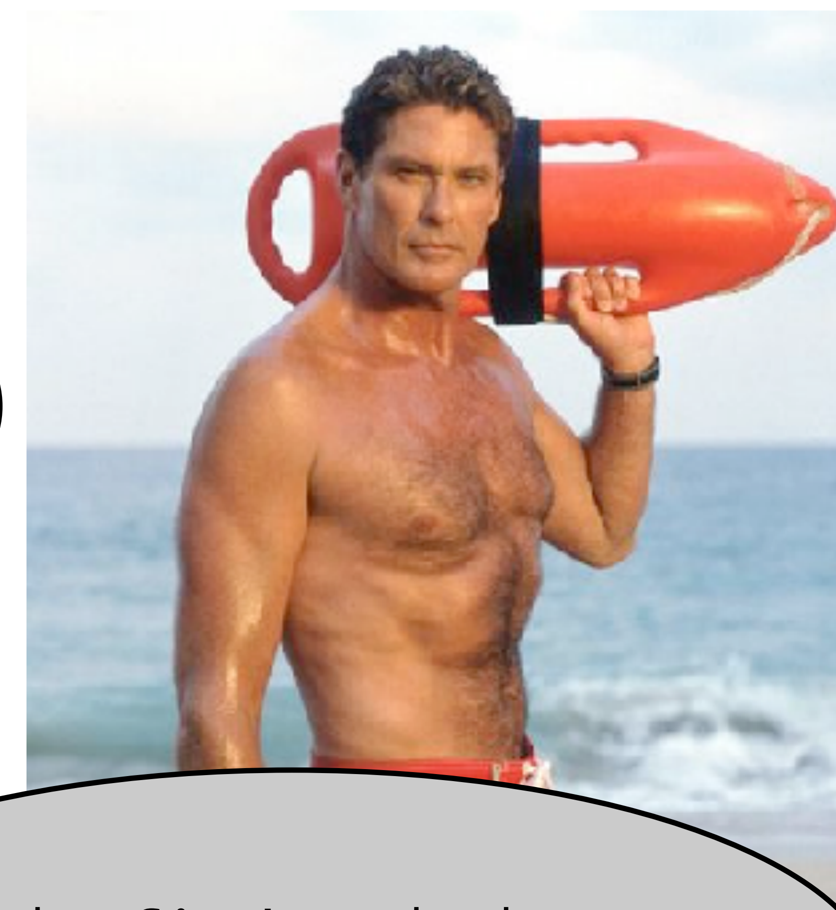
- **AOP supports the separation of concerns**. In other words, it gives a way to split the implementation of the business logic from the implementation of system-level functions.
- **Terminology**
    - An **aspect** or **cross-cutting concern** refers to **something** that needs to be done throughout the application code. Security, logging and transaction management are examples of cross-cutting concerns.
    - An **advice** is the **orthogonal logic** that is executed when a certain join point is executed (advice can be executed **before**, **after** or **around** the join point).
    - A **pointcut** is an **expression** used to define a set of join points. With a pointcut, one can specify which join points (i.e. which methods)
    - A **join point** defines **when** the orthogonal logic could be executed. For instance, the execution of a `processOrder()` **method** is a join point.



AOP to the rescue

- **AOP supports the separation of concerns**. In other words, it gives a way to split the implementation of the business logic from the implementaiton of system-level functions.
- **Terminology**
    - An **aspect** or **cross-cutting concern** refers to **something** that needs to be done throughout the application code. S… management…
    - An **advice** … a certain join… **before**, **after** o…
    - A **pointcut** is an **expression** used to define a set of join points. With a pointcut, one can specify which join points (i.e. which methods)
    - A **join point** defines **when** the orthogonal logic c… executed. For instance, the execution of a `processOrder()` **method** is a join point.

All methods that start with "`find`" in the `ch.heigvd.amt` package



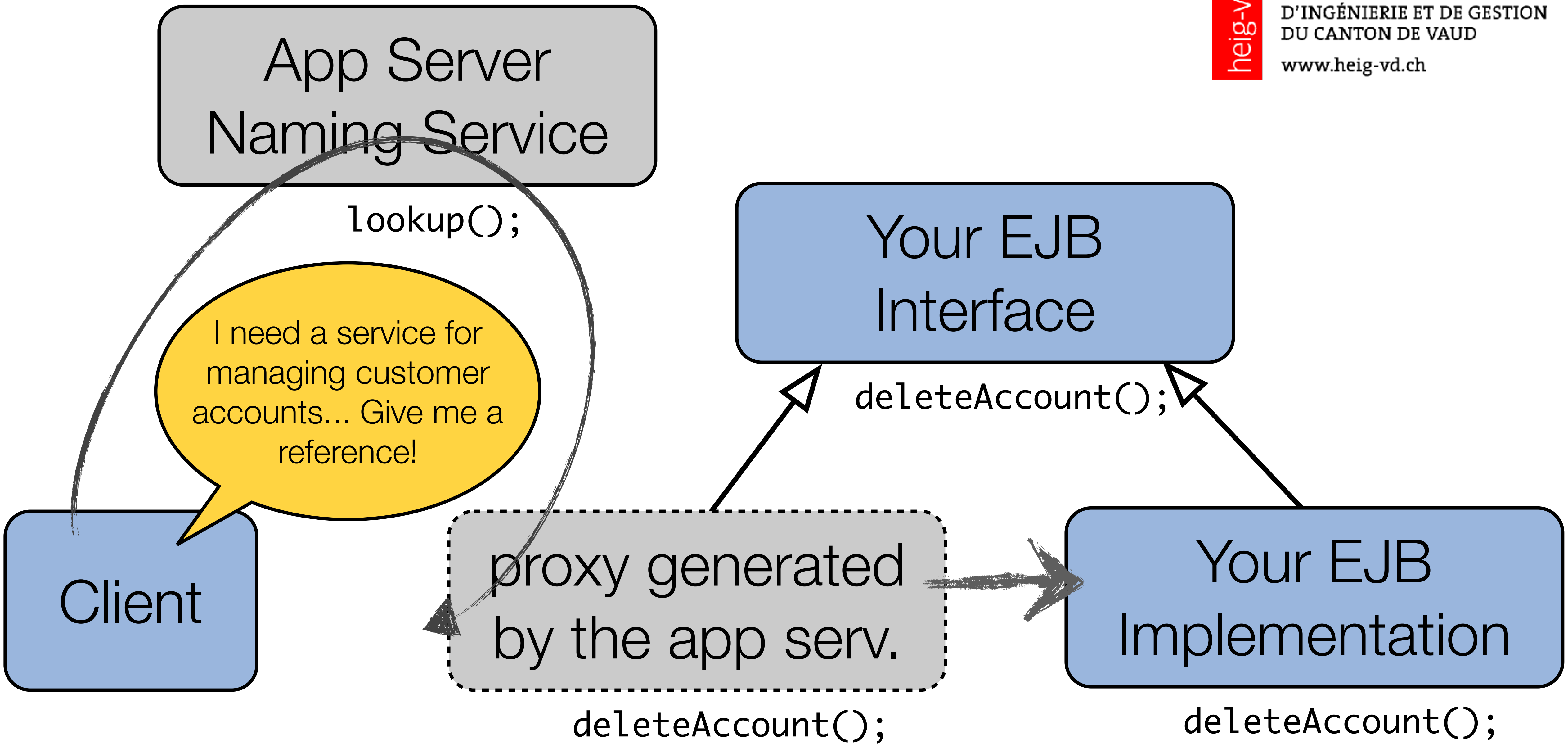
the `findSea` in the `ch.heigvd.amt.BayWatch` class

How Can it Work?

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- There are different ways to implement AOP.
- Remember that we want to "**combine**" two pieces of orthogonal code - located in two different artifacts (a "business" class and an "advice class").
- One possibility is to use a **special compilation process**. This is called "**weaving**", since the aspect code is weaved into the main business logic. As an alternative, it is possible to do the weaving as an **after-compilation** process. "Weaving" is what the AspectJ framework and toolset is doing.
- Another approach is to use **proxies** that are **dynamically generated**. This is something we can do we reflection.

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch



Your service implementation implements your interface.
The container dynamically generates a class, which implements the same interface. This class performs the technical tasks and invokes your class (proxy).

Aha! it's a mechanism we can use to implement Aspect Oriented Programming, right?

AOP in Java EE

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- Interceptors can be added **globally** (in the XML deployment descriptor) at the **class level** (apply to all methods in the class) or at the **method level**.

Interceptor Metadata Annotation	Description
`javax.interceptor.AroundConstruct`	Designates the method as an interceptor method that receives a callback after the target class is constructed
`javax.interceptor.AroundInvoke`	Designates the method as an interceptor method
`javax.interceptor.AroundTimeout`	Designates the method as a timeout interceptor for interposing on timeout methods for enterprise bean timers
`javax.annotation.PostConstruct`	Designates the method as an interceptor method for post-construct lifecycle events
`javax.annotation.PreDestroy`	Designates the method as an interceptor method for pre-destroy lifecycle events

*"Interceptors are used in conjunction with Java EE managed classes to allow developers to invoke interceptor methods on an associated target class, in conjunction with **method invocations** or **lifecycle events**. Common uses of interceptors are logging, auditing, and profiling."*

https://javaee.github.io/tutorial/interceptors.html

Examples

Binding interceptors at the **class level**:

```
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean {

    public void placeOrder(Order order) { ... }

}
```

Binding interceptors at the **method level**:

```
@Stateless
public class OrderBean {

    @Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
    public void placeOrder(Order order) { ... }

}
```

Examples

Implementing an interceptor:

```
@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}
```

Applying the interceptor:

```
@Interceptors(HelloInterceptor.class)
public void setName(String name) {
    this.name = name;
}
```

AOP in the Spring Framework

- AOP is used in the Spring Framework to:
  - provide **declarative** enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative **transaction management**
  - allow users to implement **custom aspects**, complementing their use of OOP with AOP

"If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter."

AOP with Spring: Pointcuts

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Pointcuts can be declared with an annotation (or with XML...)

```
@PointCut(expression)
private void aNameForThisSetOfMethods {}
```

The **expression** is based on the AspectJ pointcut language. Here are some examples:

the execution of any public method:
```
execution(public * *(..))
```
the execution of any method with a name beginning with "set":
```
execution(* set*(..))
```
the execution of any method defined by the AccountService interface:
```
execution(* com.xyz.service.AccountService.*(..))
```
the execution of any method on a Spring bean named 'tradeService':
```
bean(tradeService)
```
the execution of any method on a Spring bean with a name matching the wildcard expression
```
bean(*Service)
```

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
```

http://static.springsource.org/spring/docs/2.5.6/reference/aop.html#aop-pointcuts

Defining a Pointcut “Inline”

HAUTE ÉCOLE
D’INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

```
@Aspect
public class MyFirstAspect {

    @Before("execution(public * ch.heigvd.osf..*.*(..))")
    public void myMethod(JoinPoint jp) {
     System.out.println("My advice has been applied...");
     System.out.println("target: " + jp.getTarget());
     System.out.println("this: " + jp.getThis());
     System.out.println("signature: " + jp.getSignature());
    }
}
```

```
<aop:aspectj-autoproxy/>

<bean id="myFirstAspect" class="ch.heigvd.osf.hellospringaop.aspects.MyFirstAspect">
</bean>
```

Notes:

- `myMethod` will be executed before **any public method** in **any class** in the `ch.heigvd.osf package` (or in a **sub-package**) is called.
- `myMethod` has access to runtime information

Using an @Aspect to Define Pointcuts


```
package ch.heigvd.osf.system;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemPointCuts {

  @Pointcut("execution(* create*(..))")
  public void createMethods() {}

  @Pointcut("execution(* update*(..))")
  public void updateMethods() {}

  @Pointcut("execution(* delete*(..))")
  public void deleteMethods() {}

  @Pointcut("createMethods() && updateMethods() && deleteMethods()")
  public void allCRUDMethods() {}

}
```

Using an @Aspect to Implement Advices


```
package ch.heigvd.osf.system.logging;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class MyLoggingAspect {
  @Before("ch.heigvd.osf.system.SystemPointCuts.allCRUDMethods()")
  public void doLogOperation() {
    log.info(“About to call a CRUD method....”);
  }
}
```

Here, we work with:

- one pointcut, which is defined in the SystemPointCuts aspect (see previous slide)
- this pointcut defines a set of several join points: all the methods with a name starting with either create, update or delete
- one advice, which states that before every execution of the join points matching the pointcut, we will execute the doLogOperation