



# Inventory #gradle

Interceptor, JNDI variable, CDI, JSON Binding

Óbuda University, Java Enterprise Edition

John von Neumann Faculty of Informatics

Lab 9

Dávid Bedők  
2018-03-14  
v1.0

# Inventory

JavaEE additional opportunities



**Task:** Create a simple **Inventory** application where we are going to store the following properties of the elements (`InventoryItem`) in memory: *reference* (`String`), *name* (`String`), *type* (list value: `BOOK`, `DISK`, `CASSETTE` or `MAGAZINE`) and *value* (`int`).

The objectives of the task are the following:

- ▷ JavaEE **Interceptor**
- ▷ **CDI**
  - Interceptor with the help of CDI
  - **Qualifier** based injection
  - Dynamic injection
  - **Eventhandling**
  - Injection with factories
  - Direct injection (CDI bean programmed fetching)
- ▷ Application configuration via JNDI
- ▷ **JSON Binding** annotation



[gradle|maven]\jbossinventory

# Project structure

## Inventory application



- ▷ **Data tier** (inv-persistence)
  - It stores the predefined *inventory* elements in memory with the help of a *Singleton Session Bean*.
  - InventoryItem entity, InventoryType enum
- ▷ **Logic tier** (inv-ejbservice)
  - this will be the location of the new knowledge what we would like to learn with that project
  - InventoryItemBean DTO
    - it aggregates the type of the *inventory* as a String instead of an InventoryType instance
    - InventoryItemBeanConverter converts between the entity and the bean
  - InternationalInventoryItemBean DTO
    - a special business view of the InventoryItem
    - InternationalInventoryItemConverter converts between the entity and the bean
- ▷ **Presentation tier** (inv-webservice)
  - it contains RESTful webservices to present the subtasks
  - Context root: **inventory**
  - RESTful API application path: **api**



InventoryHolder SSB stores the *inventory* elements in memory. The bean initializes the data and it gives back the list if a service would like to use that.

Two SLSB were defined inside the layer:

### ▷ InventoryFinder

```
1 @Local
2 public interface InventoryFinder {
3     InventoryItem get(String reference);
4     List<InventoryItem> list(InventoryType type);
5 }
```

### ▷ InventorySearch

```
1 @Local
2 public interface InventorySearch {
3     List<InventoryItem> list(InventoryType type, String nameTerm);
4 }
```

The implementation of the InventorySearch was written in **Groovy** language (optional task), that is way these services are separated.



We will use RESTful webservice to simulate the operations of the *Inventory* system.

### BasicRestService

- ▷ <http://localhost:8080/inventory/api/basic/{reference}>
  - It gives back an *inventory* item of the given reference in JSON format.
- ▷ <http://localhost:8080/inventory/api/basic/list/{type}>
  - It gives back a list of *inventory* items of the given type (*InventoryTypeEnum*) in JSON format.
- ▷ <http://localhost:8080/inventory/api/basic/list/{type}/{nameTerm}>
  - It gives back a list of *inventory* items of the given type and name term in JSON format.

The layer contains other webservices as well, we will present its business abilities later.

# Copyright information

## Subtask



**Subtask:** Change the BasicRestService RESTful webservice methods to automatically supplement the outputs/responses with *copyright* information (© [YEAR] Inventory).

### Original JSON

```
1 {
2   "reference": "LOR78",
3   "name": "Lorem20",
4   "type": "BOOK",
5   "value": 78
6 }
```

### Modified JSON

```
1 {
2   "reference": "LOR78",
3   "name": "Lorem20 @ 2018 Inventory",
4   "type": "BOOK",
5   "value": 78
6 }
```

This subtasks could be easily implemented with some new lines of code, but this solution would be unfavorable in multiple aspects:

- ▷ It causes redundant and *boilerplate* source code (repeating method class in several business methods).
- ▷ Independent business responsibilities in the same business methods (it contradicts the OO *encapsulation*)
- ▷ If we would like to turn on/off this additional liability we have to further increase the complexity of the code

Because of these we are going to implement this subtask with the help of the JavaEE **interceptor**.

# Interceptor

## Aspect-Oriented Programming (AOP)

- ▷ JSR 318, Enterprise JavaBeans 3.1
- ▷ Interceptors are used in conjunction with the methods (around invoke/timeout) or lifecycle events of the target class (postconstruct/predestroy).
- ▷ **Creation**: An interceptor can be defined within a target class as an interceptor method, or in an associated class called an interceptor class (recommendation: as a class because of the *Separation of Concern*<sup>1</sup>).
- ▷ **Binding**: In the place of use we are able to configure/use these with the `javax.interceptor.Interceptors` (s!) annotation, XML based configuration or with the help of the CDI. We will use the last mentioned one but the slides will cover the other options as well.

Interceptor Metadata Annotation	Description
<code>javax.interceptor.AroundInvoke</code>	Designates the method as an interceptor method.
<code>javax.interceptor.AroundTimeout</code>	Designates the method as a timeout interceptor, for interposing on timeout methods for enterprise bean timers.
<code>javax.annotation.PostConstruct</code>	Designates the method as an interceptor method for post-construct lifecycle events.
<code>javax.annotation.PreDestroy</code>	Designates the method as an interceptor method for pre-destroy lifecycle events.

<sup>1</sup> [https://en.wikipedia.org/wiki/Separation\\_of\\_concerns](https://en.wikipedia.org/wiki/Separation_of_concerns)

# Copyright interceptor

The *interceptors* are part of the *EJB context*, so by default we do not need to 'register' them, these will be activated automatically during the binding.

```
1 [...]
2 public class CopyrightInterceptor implements Serializable {
3
4     @AroundInvoke
5     public Object decorateCoinWithCopyright(InvocationContext context) throws Exception {
6         int currentYear = Calendar.getInstance().get(Calendar.YEAR);
7         final Object result = context.proceed();
8         if (result instanceof InventoryItemBean) {
9             this.modifyInventoryItemName(result, currentYear);
10        } else if (result instanceof List) {
11            List<?> resultList = (List<?>) result;
12            for (Object resultItem : resultList) {
13                if (resultItem instanceof InventoryItemBean) {
14                    this.modifyInventoryItemName(resultItem, currentYear);
15                }
16            }
17        }
18        return result;
19    }
20
21    private void modifyInventoryItemName(Object object, int currentYear) {
22        InventoryItemBean item = (InventoryItemBean) object;
23        String description = item.getName();
24        item.setName(description + " @ " + currentYear + " Inventory");
25    }
26 }
```

We can walk through the parameters, its values and types. In case of own annotations and with the help of the *Reflection API* we are able to install additional control.

We would like to embed this around a method call (`@AroundInvoke` annotation). The real call will be initiated by us (`context.proceed()`) and because of this we can prevent the service call or modify its output.

CopyrightInterceptor.java



# Interceptor binding

with the help of `ejb-jar.xml`

## XML based configuration

We did not deal with the XML based configuration before, but each annotation based configuration can be controlled in XML as well in the `ejb-jar.xml` deployment descriptor of the EJB *module*. If both configuration types exist at the same time the XML based configuration is the stronger. It is easy to say that the usage of the XML file is inconvenient, but it has a great advantage: the application does not need new *compile time* dependencies if we ignore the usage of the annotations (we do not need to put the API jars into the compile *classpath*).

```
1 <interceptor-binding>
2   <target-name>
3     hu.qwaevisz.inventory.ejbservice.facade.InventoryFacadeImpl
4   </target-name>
5   <interceptor-class>
6     hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor
7   </interceptor-class>
8   <method-name>getInventory</method-name>
9 </interceptor-binding>
```

`ejb-jar.xml`



# Interceptor binding

with the help of `@Interceptors` annotation

The CDI binding will be a more convenient solution, after we get to know the CDI we will use that only.

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @EJB
5     private InventoryFinder finder;
6
7     @Inject
8     private InventoryItemBeanConverter converter;
9
10    @Override
11    @Interceptors({ CopyrightInterceptor.class })
12    public InventoryItemBean getInventoryItem(String reference) throws AdaptorException {
13        return this.converter.to(this.finder.get(reference));
14    }
15
16    @Override
17    @Interceptors({ CopyrightInterceptor.class })
18    public List<InventoryItemBean> getInventoryItems(final InventoryType type) throws
        AdaptorException {
19        return this.converter.to(this.finder.list(type));
20    }
21    [...]
22 }
```

We will back to the injection of the `InventoryItemBean`'s converter after the preliminary section of the CDI. It can be implemented as an SLSB as well.

Multiple *interceptor* class can be listed with the help of the `@Interceptors` annotation.

InventoryFacadeImpl.java

# Testing Copyright information

via RESTful webservice



<http://localhost:8080/inventory/api/basic/LOR78>

```
13:28:43,630 INFO [hu.qwaevisz.inventory.webservice.SearchRestServiceBean] (HTTP-1)
Get Inventory by LOR78 (reference)
13:28:43,636 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-1) Copyright Interceptor activated in getInventory
13:28:43,674 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-1) Add copyright for LOR78's name.
```

```
1 {
2     "reference": "LOR78",
3     "name": "Lorem20 @ 2018 Inventory",
4     "type": "BOOK",
5     "value": 78
6 }
```

# Improvements options

## Relation of the interceptors and CDI

The usage of the *interceptors* are simple and great but we can do it in a more elegant way if we use CDI.

- ▷ We mark the *interceptor* class with the `@Interceptor` (singular) annotation.
- ▷ Create a new annotation what we will use for binding. We decorate this annotation with the `@InterceptorBinding` annotation which is an annotation to decorate annotations.
- ▷ With the new annotation we will decorate the places where we would like to use the *interceptor*.

The *interceptor* bindings what we created in that way are **turned off by default**, we have to active these in the `beans.xml`. That file is the *deployment descriptor* file of the CDI, the presence of this file active the CDI (it could be empty as well).

- ▷ JSR 299, JSR 330, JSR 316
- ▷ Integration with the *Expression Language* (EL), which allows any component to be used directly within a *Java Server Faces* page or a JSP page
- ▷ The ability to decorate injected components
- ▷ The ability to associate **interceptors** with components using typesafe interceptor bindings
- ▷ An **event-notification** model
- ▷ A *web conversion scope* in addition to the three standard scopes (*request*, *session*, and *application*) defined by the *Java Servlet* specification
- ▷ A complete *Service Provider Interface* (SPI) that allows 3<sup>rd</sup> party frameworks to integrate cleanly in the Java EE 6 environment
- ▷ With the CDI we can reach/handle resources (which have String keys in the JNDI tree) in a **type-safe** way

- ▷ From now we can use the CDI's @Inject annotation instead of the @EJB annotation.
  - The @EJB annotation can be used only where the EJB *context* is reachable (e.g.: we cannot use that in a single POJO, but we can in a *servlet*, in a *session bean* or in any class which belongs to the EJB *context*).
  - The CDI can be usable almost in any Java class (which reaches the CDI *context*), in session beans, etc (but you cannot use it inside a *Message Driven Bean* or in *Remote EJBs*).
- ▷ We have to put a `beans.xml` into the *classpath* in the EJB *module* which we would like to use the CDI (`src/main/resources/beans.xml`). To start using it the file could be entirely empty (<http://www.cdi-spec.org/faq/>).

# InterceptorItemBean converter

```
1 public interface InventoryItemBeanConverter {
2     InventoryItemBean to(InventoryItem item);
3     List<InventoryItemBean> to(List<InventoryItem> items);
4 }
```

There is no `@Local` annotation on the interface.

## InventoryItemBeanConverter.java

```
1 public class InventoryItemBeanConverterImpl implements InventoryItemBeanConverter {
2
3     @Override
4     public InventoryItemBean to(InventoryItem item) { [...] }
5
6     @Override
7     public List<InventoryItemBean> to(List<InventoryItem> items) {
8         return items.stream().map(this::to).collect(Collectors.toList());
9     }
10 }
```

There is no `@Stateless` annotation on the class.

## InventoryItemBeanConverterImpl.java

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryItemBeanConverter {
3
4     @Inject
5     private InventoryItemBeanConverter converter;
6
7     [...]
8 }
```

We use the `@Inject` annotation instead of the `@EJB`. At the background we get a proxy in that case also. The CDI engine finds that we have only one implementation of that 'service'/'resource' and this was got the `@Default` *qualifier* automatically.

## InventoryItemBeanConverter.java

# CDI vs. EJB

EJB	CDI
Provides <i>Container</i> services (with transaction handling, security, concurrency, etc.).	Provides modularization and separation of concerns, event-handling, injection
EJBs are CDI beans. EJB's have all the benefits of CDI.	-
@Stateful, @Stateless, @Singleton resolving	@RequestScoped, @SessionScoped, @ApplicationScoped, @ConversationScoped (JSF) or custom <i>scope</i> resolving
Load on startup via @Startup annotation	there is no equivalent in CDI
@Asynchronous method invocation	there is no equivalent in CDI
scheduling work with @Schedule annotation	there is no equivalent in CDI
@TransactionAttribute annotation	Using <i>EntityManagers</i> in a JTA transaction: You can use them with plain CDI, but without the <i>container-managed</i> transactions it can get really monotonous duplicating the <i>UserTransaction</i> commit/roll-back logic.
Easy synchronization with @Singleton with @Lock(READ) and @Lock(WRITE) annotations	there is no equivalent in CDI

Performance: CDI instance resolution is perhaps a bit more complex (more dynamic and contextual), but in practice there is zero performance difference between invoking an EJB vs. invoking a CDI bean.

Same fundamental design (*proxy pattern*).





**Subtask:** Log the method calls (with argument values) automatically and measure the running time of that service call in case of the `BasicRestService` RESTful webservice.

Interpret the subtask with JavaEE **interceptor**, but at this time register and bind the *interceptor* with the help of CDI.

# Register interceptors with the help of CDI

With the help of the `javax.interceptor.Interceptor` annotation we can register the class for the CDI.

```
1 [..]
2 @Interceptor
3 public class LoggedInterceptor implements Serializable {
4     [..]
5 }
```

LoggedInterceptor.java

This interceptor will be **inactive** by default, we have to activate it via the `beans.xml`.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://java.sun.com/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5         http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
6     <interceptors>
7         <class>hu.qwaevisz.inventory.ejbsservice.interceptor.LoggedInterceptor</class>
8     </interceptors>
9
10 </beans>
```

inv-ejbsservice | src | main | resources | beans.xml

# Logging and measure the running time

```
1 @Interceptor
2 public class LoggedInterceptor implements Serializable {
3
4     private static final Logger LOGGER = Logger.getLogger(LoggedInterceptor.class);
5
6     @AroundInvoke
7     public Object logMethodInvocations(InvocationContext context) throws Exception {
8         final StringBuilder info = new StringBuilder();
9         info.append(context.getTarget().getClass().getName()).append(".").append(context.getMethodName());
10        final Object[] parameters = context.getParameters();
11        if (parameters != null) {
12            int i = parameters.length - 1;
13            for (final Object parameter : parameters) {
14                info.append(parameter.toString());
15                if (i > 0) {
16                    info.append(",");
17                }
18                i--;
19            }
20        }
21        info.append(")");
22        LOGGER.info("Entering: " + info.toString());
23        final long start = System.currentTimeMillis();
24        final Object result = context.proceed();
25        final long end = System.currentTimeMillis();
26        LOGGER.info("Exiting: " + info.toString() + " - running time: " + (end - start) + "
27            millisecond(s)");
28        return result;
29    }
}
```

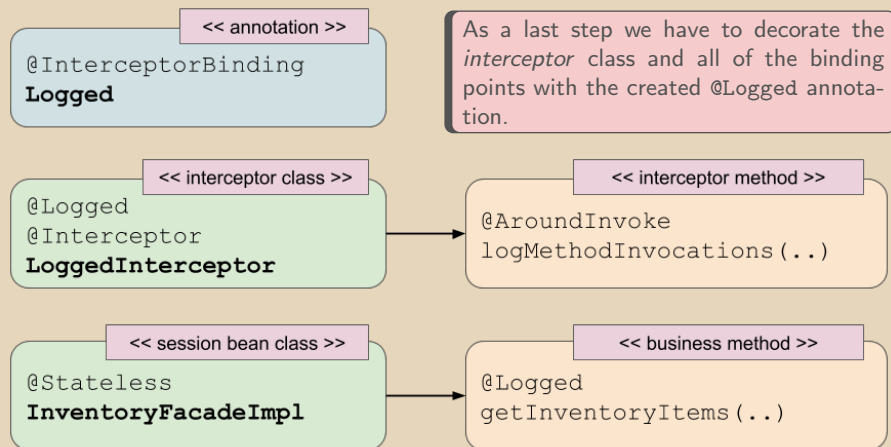
# Logged

## @InterceptorBinding annotation

```
1 package hu.qwaevisz.inventory.ejbservice.interceptor;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Inherited;
5 import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
7 import java.lang.annotation.Target;
8
9 import javax.interceptor.InterceptorBinding;
10
11 @Inherited
12 @InterceptorBinding
13 @Retention(RetentionPolicy.RUNTIME)
14 @Target({ ElementType.TYPE, ElementType.METHOD })
15 public @interface Logged {
16
17 }
```

@InterceptorBinding annotation is good to connect *interceptors* and methods, so **it is an annotation to annotate annotations**. We are going to use that (@Logged) annotation on the class of the *interceptor* and on the methods where we would like to use that interceptor.

Logged.java



# Modify the class of the interceptor

```
1  [...]
2  @Logged
3  @Interceptor
4  public class LoggedInterceptor implements Serializable {
5
6      [...]
7
8      @AroundInvoke
9      public Object logMethodInvocations(InvocationContext context)
10         throws Exception {
11         [...]
12     }
13 }
```

LoggedInterceptor.java

# Binding interceptor with the help of CDI

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @EJB
5     private InventoryFinder finder;
6
7     @Inject
8     private InventoryItemBeanConverter converter;
9
10    @Logged
11    @Override
12    @Interceptors({ CopyrightInterceptor.class })
13    public List<InventoryItemBean> getInventoryItems(final InventoryType type) throws
        AdaptorException {
14        return this.converter.to(this.finder.list(type));
15    }
16
17    @Inject
18    private InventorySearch search;
19
20    @Logged
21    @Override
22    public List<InventoryItemBean> getInventoryItems(final InventoryType type, final
        String nameTerm) throws AdaptorException {
23        return this.converter.to(this.search.list(type, nameTerm));
24    }
25
26    [...]
27 }
```

We can decorate any methods with the `@Logged` annotation where we would like to log the arguments and measure the running time.

# Testing the logging and running time measurement - I

via RESTful webservices



<http://localhost:8080/inventory/api/basic/list/BOOK>

```
13:29:52,659 INFO [hu.qwaevisz.inventory.webservice.SearchRestServiceBean] (HTTP-3)
Get Inventories by BOOK type
13:29:52,659 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-3) Copyright Interceptor activated in getInventories
13:29:52,660 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.LoggedInterceptor]
(HTTP-3) Entering:
hu.qwaevisz.inventory.ejbservice.facade.InventoryFacadeImpl.getInventories(BOOK)
13:29:52,672 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.LoggedInterceptor]
(HTTP-3) Exiting:
hu.qwaevisz.inventory.ejbservice.facade.InventoryFacadeImpl.getInventories(BOOK) -
running time: 12 millisecond(s)
13:29:52,672 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-3) Add copyright for LOR42's name.
13:29:52,672 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-3) Add copyright for LOR78's name.
13:29:52,672 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-3) Add copyright for LOR34's name.
13:29:52,672 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.CopyrightInterceptor]
(HTTP-3) Add copyright for SIT78's name.
```



# Testing the logging and running time measurement - II

via RESTful webservices



<http://localhost:8080/inventory/api/basic/list/BOOK/Lorem>

```
13:31:18,760 INFO [hu.qwaevisz.inventory.webservice.SearchRestServiceBean] (HTTP-2)
Get Inventories by BOOK type and Lorem name
13:31:18,761 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.LoggedInterceptor]
(HTTP-2) Entering:
hu.qwaevisz.inventory.ejbservice.facade.InventoryFacadeImpl.getInventories(BOOK, Lorem)
13:31:19,168 INFO [hu.qwaevisz.inventory.ejbservice.interceptor.LoggedInterceptor]
(HTTP-2) Exiting:
hu.qwaevisz.inventory.ejbservice.facade.InventoryFacadeImpl.getInventories(BOOK, Lorem)
- running time: 407 millisecond(s)
```

```
1 [
2   {
3     "reference": "LOR42",
4     "name": "Lorem10 @ 2018 Inventory",
5     "type": "BOOK",
6     "value": 42
7   },
8   {
9     [...]
10  },
11  [...]
12 ]
```

# Activate promotional period with *qualifier*

## Subtask



**Subtask:** We can buy the *inventory* items via RESTful webservice, but sometimes in case of a promotional period we would like to sell these items with a discount price (20% less). Of course the price of the item will not be changed at the persistence layer, only the retrieved `InventoryItemBean` will contain the reduced price.

In a separate *endpoint* let we have a change to get the normal and the promotional `InventoryItemBean` as well regardless of the discount period.

The price adjustment will be served through a `CostService` which is a simple *interface* with two implementations: `NormalCostService` and `PromotionalCostService`. In promotional period we are going to *inject* the appropriate implementation with the help of the CDI *qualifiers*.

In case of the separate *endpoint* we will *inject* the given implementation **dynamically**.



## PurchaseRestService

- ▷ <http://localhost:8080/inventory/api/purchase/{reference}>
  - Buy the *inventory* item of the given reference based on the injected period.
- ▷ <http://localhost:8080/inventory/api/purchase/{reference}/{pricing}>
  - Buy the *inventory* item of the given reference and the given pricing strategy.
  - PricingStrategy (enum): STANDARD or DISCOUNT

# Qualifiers

in relation to JavaEE CDI

The **qualifier** is an annotation which is good to annotate *beans*. An annotation becomes a *qualifier* if it has the `javax.inject.Qualifier` annotation. With the help of the *qualifiers* we can use the injection of multiple EJB's implementation in a *type-safe* way (we can inject the implementation which has the given *qualifier*). Moreover we do not need *session bean* at all, we can use the same technique with *CDI beans*.

If a *bean* does not have any *qualifier* it will get a `@Default` *qualifier* automatically.

## Multiple implementation in the world of EJBs

If an EJB has more than one implementation we have to set the name of the *bean* during injection (e.g.: `@Stateless(name="lorem")` and `@EJB(beanName="lorem")`). This solution is not *type-safe* enough because of the `String` value, but of course we can put that value into the `@Local/@Remote` interface as a constant.

# Promotional qualifiers

```
1 package hu.qwaevisz.inventory.ejbservice.qualifier;  
2 [..]  
3 @Qualifier  
4 @Retention(RetentionPolicy.RUNTIME)  
5 @Target({ ElementType.TYPE, ElementType.METHOD,  
6           ElementType.FIELD, ElementType.PARAMETER })  
7 public @interface Standard {  
8 }
```

## Standard.java

```
1 package hu.qwaevisz.inventory.ejbservice.qualifier;  
2 [..]  
3 @Qualifier  
4 @Retention(RetentionPolicy.RUNTIME)  
5 @Target({ ElementType.TYPE, ElementType.METHOD,  
6           ElementType.FIELD, ElementType.PARAMETER })  
7 public @interface Discount {  
8 }
```

## Discount.java

# CostService and its implementations

```
1 package hu.qwaevisz.inventory.ejb.service.cost;
2 public interface CostService {
3     int getCost(int originalValue);
4 }
```

## CostService.java

```
1 [...]
2 @Standard
3 public class NormalCostService implements CostService {
4     @Override
5     public int getCost(int originalValue) {
6         return originalValue;
7     }
8 }
```

## NormalCostService.java

```
1 [...]
2 @Discount
3 public class PromotionalCostService implements CostService {
4     private static final float DISCOUNT_PERCENT = 20;
5     @Override
6     public int getCost(int originalValue) {
7         return Math.round(originalValue * (1 - (DISCOUNT_PERCENT / 100)));
8     }
9 }
```

## PromotionalCostService.java

# Qualifier based *injection*

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @EJB
5     private InventoryFinder finder;
6
7     @Inject
8     private InventoryItemBeanConverter converter;
9
10
11     @Inject
12     @Discount
13     private CostService costService;
14
15     @Logged
16     @Override
17     public InventoryItemBean buyInventoryItem(String reference) throws AdaptorException {
18         final InventoryItemBean bean = this.converter.to(this.finder.get(reference));
19         bean.setValue(this.costService.getCost(bean.getValue()));
20         return bean;
21     }
22 }
```

InventoryFacadeImpl.java

# Testing the promotional price

via RESTful webservice



<http://localhost:8080/inventory/api/purchase/LOR78>

```
1 {  
2   "reference": "LOR78",  
3   "name": "Lorem20",  
4   "type": "BOOK",  
5   "value": 62  
6 }
```

The price of the product in the promotional period is only **62** (EUR) instead of **78** ( $78 * 0.8 = 62.4$ ).



# Dynamic injection

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3     [...]
4     @Any
5     @Inject
6     private Instance<CostService> dynamicCostService;
7
8     @Logged
9     @Override
10    public InventoryItemBean buyInventoryItem(String reference, PricingStrategy pricing)
11        throws AdaptorException {
12        final InventoryItemBean bean = this.converter.to(this.finder.get(reference));
13        bean.setValue(this.getCostService(pricing).getCost(bean.getValue()));
14        return bean;
15    }
16    private CostService getCostService(PricingStrategy pricing) {
17        CostService service = null;
18        switch (pricing) {
19            case STANDARD:
20                service = this.dynamicCostService.select(new AnnotationLiteral<Standard>() {
21                    private static final long serialVersionUID = 1L;
22                }).get();
23                break;
24            case DISCOUNT:
25                service = this.dynamicCostService.select(new DiscountQualifier()).get();
26                break;
27        }
28        return service;
29    }
30 }
```

## DiscountQualifier *selector* in a separate class

```
1 package hu.qvaevisz.inventory.ejbservice.qualifier;
2
3 import javax.enterprise.util.AnnotationLiteral;
4
5 public class DiscountQualifier extends
6     AnnotationLiteral<Discount> implements Discount {
7     private static final long serialVersionUID = 1L;
8
9 }
```

DiscountQualifier.java

### Curiosity

Here we exploit that we create an annotation as an @interface (implements Discount).

# International expansion with JSON *binding* annotation

## Subtask



**Subtask:** In the gate to international expansion an outer system would like to fetch the *inventory* items as well. This system expects something else what we provide currently, so we need a new `InternationalInventoryItemBean` type for that purpose. It contains a `product` and a `price`. The product value stands from the combination of the item name and reference. The inventory application has to convert the EUR price (we currently store the price in EUR) to an other preconfigured currency, and the value of the price will contain the combination of that converted value and the currency.

```
1 {  
2   "price": "23790 HUF",  
3   "product": "LOR78-Lorem20"  
4 }
```

- ▷ Let it be possible to configure the keys of the new *bean* without changing the name of the fields (as before in case of XML, configure the JSON output via annotation).
- ▷ The expected currency will be a startup parameter of the application (e.g.: HUF) what we store as a JNDI value under the `java:global/currency` key. We do the same with the exchange rate too (key: `java:global/exchangeRate`)



## InternationalRestService

- ▷ <http://localhost:8080/inventory/api/international/{reference}>
  - It gives back the 'international datasheet' of the inventory item in the preconfigured currency and format.
- ▷ <http://localhost:8080/inventory/api/international/notify/{reference}>
  - The same functionality as the previous *endpoint* but here during the implementation we will send event(s) as well.
  - The sub-tasks will be dismantled in order to correct the presentation order.

```
1 <?xml version='1.0' encoding='UTF-8'?>
2 <server xmlns="urn:jboss:domain:1.7">
3   <profile>
4     [..]
5     <subsystem xmlns="urn:jboss:domain:naming:1.4">
6       <remote-naming/>
7       <bindings>
8         <simple name="java:global/currency" value="HUF"/>
9         <simple name="java:global/exchangeRate" value="305"
10           type="int" />
11       </bindings>
12     </subsystem>
13     [..]
14 </profile>
15 </server>
```

Beside the simple values (simple) we have the change to define object reference (object-factory) and JNDI *alias* as well (lookup).

The variables names cannot start anyway. The *container* will build in these in its own existing hierarchy according to its own naming rules. If we use the `java:global` prefix we are going to reach the variable in the whole application server instance.

standalone.xml

# Conversion

## InternationalInventoryItemBean

The conversion of the `InternationalInventoryItemBean` are interpreted in two level (as always):

- ▷ `InternationalInventoryItemConverter` CDI *bean* of the EJB service layer creates `InternationalInventoryItemBean` from `InventoryItem`.
- ▷ **Jackson JSON library** creates a well-formed JSON instance from `InternationalInventoryItemBean` (with the mediation of JAX-RS).

# InternationalInventoryItemBean

```
1 package hu.qwaevisz.inventory.ejbservice.domain;
2
3 import org.codehaus.jackson.annotate.JsonProperty;
4
5 public class InternationalInventoryItemBean {
6
7     private String label;
8     private String price;
9
10    public InternationalInventoryItemBean(String label, String price) { [...] }
11
12    @JsonProperty("product")
13    public String getLabel() { return this.label; }
14
15    public void setLabel(String label) { this.label = label; }
16
17    @JsonProperty("price")
18    public String getPrice() { return this.price; }
19
20    public void setPrice(String price) { this.price = price; }
21
22    @Override
23    public String toString() { return this.label; }
24 }
```

By way of example we use a different name instead of label.

The @JsonProperty annotation can be put into any *getter* method and its functioning is equivalent with the @XmlElement (name = "[...]") annotation.

InternationalInventoryItemBean.java

# JSON Binding dependency



inv-ejbservice project

JAX-B have already supported the XML binding in the level of the JavaEE 6 API, but in case of JSON only the Java EE 8 supports it with the **JSON Binding API** (the standard is based on the *Jackson*, *Gson* and *Genson* libraries, the related annotation is the `@JsonbProperty`). That is way we have to add a separate dependency for that purpose in case of Java EE 6 environment.

```
1 dependencies {
2   [..]
3   compile group: 'org.codehaus.jackson', name: 'jackson-core-asl', version:
4     jacksonVersion
5   compile group: 'org.codehaus.jackson', name: 'jackson-mapper-asl', version:
6     jacksonVersion
7   [..]
8 }
```

inv-ejbservice | build.gradle

```
1 [..]
2 ext {
3   [..]
4   jacksonVersion = '1.9.9'
5   [..]
6 }
7 [..]
```

root | build.gradle



# InternationalInventoryItemConverter

```
1 package hu.qwaevisz.inventory.ejbservice.converter;
2
3 import
4     hu.qwaevisz.inventory.ejbservice.domain.InternationalInventoryItem;
5 import hu.qwaevisz.inventory.persistence.domain.InventoryItem;
6
7 public interface InternationalInventoryItemConverter {
8     InternationalInventoryItemBean to(InventoryItem item);
9
10 }
```

InternationalInventoryItemConverter.java

## Default implementation

We do not need any annotation to inject that implementation with CDI. If only one implementation exists it will automatically get the `@Default` *qualifier*, and the meaning of the `@Inject` annotation without any *qualifier* is that we would like to use the default implementation.

# InternationalInventoryItemConverterImpl

The `@Resource` annotation will retrieve the element from the JNDI tree based on the given JNDI name. The `InitialContext` is ensured by the *container*.

```
1  [...]
2  public class InternationalInventoryItemConverterImpl implements
      InternationalInventoryItemConverter {
3
4      @Resource(lookup = "java:global/currency")
      private String currency;
5
6
7      @Resource(lookup = "java:global/exchangeRate")
      private int exchangeRate;
8
9
10     @Override
11     public InternationalInventoryItemBean to(InventoryItem item) {
12         String label = item.getReference() + "-" + item.getName();
13         String price = item.getValue() * this.exchangeRate + " " +
             this.currency;
14         return new InternationalInventoryItemBean(label, price);
15     }
16 }
```

# Injection without qualifier

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @EJB
5     private InventoryFinder finder;
6
7     @Inject
8     private InternationalInventoryItemConverter internationalConverter;
9
10    @Logged
11    @Override
12    public InternationalInventoryItemBean getInternationalInventoryItem(final String
        reference) throws AdaptorException {
13        return this.internationalConverter.to(this.finder.get(reference));
14    }
15
16 }
```

InventoryFacadeImpl.java

# Tax administration data collection with event handling

## Subtask



**Subtask:** Because of some tax administration issues the system needs to notify the office when an international *inventory* item is retrieved. Two independent business process will subscribe this notification:

- ▷ A **report module** to generate a report from this collected data (we will simulate this with some logging)
- ▷ An **operator module** would like to monitor and log which preconfigured technical client supervised the query. The technical client will be part of the deployed application's configuration (we will *inject* this as a resource indirectly).

We are going to solve that subtask with JavaEE **event handling** (it is also part of the CDI).

In the first round we will define one technical clients for each deployment (SANDBOX, TEST and LIVE). We would like to differentiate a `ClientHolder` CDI *bean* which implementations are the factory resources of that clients. We would need three different *qualifiers* to reach that goal. Try to do that with less new types: a combination of a single *qualifier* and an *enum* (this technique is beneficial only if you have more than 2 implementations of a bean).

# Technical client

```
1 package hu.qwaevisz.inventory.ejbservice.domain;
2 public class Client {
3
4     private final String reference;
5     private final String name;
6
7     public Client(final String reference, String name) {
8         this.reference = reference;
9         this.name = name;
10    }
11
12    public String getReference() { return this.reference; }
13
14    public String getName() { return this.name; }
15
16    @Override
17    public String toString() {
18        return this.name + " (" + this.reference + ")";
19    }
20 }
```

Client.java

# One *qualifier* 'to rule them all'

```
1 package hu.qwaevisz.inventory.ejbservice.qualifier;
2 [...]
3 @Qualifier
4 @Retention(RetentionPolicy.RUNTIME)
5 @Target({ ElementType.TYPE, ElementType
6           })
7 public @interface ClientFlag {
8     ClientType value();
9 }
10 }
```

The `value()` 'attribute' of the *qualifier* returns an instance of a *ClientType* enum. This enum will be used to differentiate the implementations, so we are able to handle  $N$  different implementations with one *qualifier* and one *enum* (we should use it in case of  $N > 2$ ).

ClientFlag.java

```
1 package hu.qwaevisz.inventory.ejbservice.domain;
2 public enum ClientType {
3     SANDBOX,
4     LIVE,
5     TEST,
6     CUSTOM
7 }
```

ClientType.java

## CDI *bean* to provide technical client

```
1 package hu.qvaevisz.inventory.ejbservice.client;
2 import hu.qvaevisz.inventory.ejbservice.domain.Client;
3 public interface ClientHolder {
4     Client getClient();
5 }
```

ClientHolder.java

```
1 public abstract class AbstractClientHolder implements
    ClientHolder {
2
3     private final Client client;
4
5     public AbstractClientHolder(final String reference, String
        name) {
6         this.client = new Client(reference, name);
7     }
8
9     @Override
10    public Client getClient() { return this.client; }
11 }
```

Each implementation will create a `Client` which is related to the operational system. The injected CDI *bean* will ensure the technical client.

AbstractClientHolder.java

# Implementation for providing technical clients

```
1 @ClientFlag(ClientType.LIVE)
2 public class LiveClientHolder extends AbstractClientHolder {
3     public LiveClientHolder() {
4         super("MCF012", "Matthew C. Flores");
5     }
6 }
```

LiveClientHolder.java

The *injection* is clear in both implementations because each has *default constructor*. We just simply put the `@ClientFlag` *qualifier* into the implementation class.

```
1 @ClientFlag(ClientType.SANDBOX)
2 public class SandboxClientHolder extends AbstractClientHolder {
3     public SandboxClientHolder() {
4         super("SVW987", "Scott V. Wright");
5     }
6 }
```

SandboxClientHolder.java

```
1 @ClientFlag(ClientType.TEST)
2 public class TestClientHolder extends AbstractClientHolder {
3     public TestClientHolder() {
4         super("TEST", "Test System");
5     }
6 }
```

TestClientHolder.java



# Event handling

JavaEE CDI context

- ▷ Event
  - POJO
  - NotifierEvent
- ▷ Event handler(s)
  - ReportEventHandler and NotifierEventHandler
  - @Observes NotifierEvent (CDI annotation)
  - Any eventhandlers can subscribe an event
- ▷ Subscribe, registration, event sending

```
1 @Inject
2 private Event<NotifierEvent> notifier;
3 [...]
4 this.notifier.fire(new NotifierEvent([..]));
```

# NotifierEvent

## Event

```
1 package hu.qwaevisz.inventory.ejb.service.event;
2 [...]
3 public class NotifierEvent implements Serializable {
4
5     private InternationalInventoryItemBean bean;
6
7     public NotifierEvent() {}
8
9     public NotifierEvent(InternationalInventoryItemBean bean) {
10         this.bean = bean;
11     }
12
13     public InternationalInventoryItemBean getBean() {
14         return this.bean;
15     }
16
17     [...]
18 }
```

NotifierEvent.java

# ReportEventHandler

## EventHandler

```
1 package hu.qwaevisz.inventory.ejbservice.event;
2 [...]
3 public class ReportEventHandler implements Serializable {
4
5     private static final Logger LOGGER =
6         Logger.getLogger(ReportEventHandler.class);
7
8     public void handle(@Observes NotifierEvent event) {
9         LOGGER.info("Report this event: " + event);
10    }
11 }
```

ReportEventHandler.java

# NotifierEventHandler

## EventHandler

```
1 package hu.qwaevisz.inventory.ejbservice.event;
2 [...]
3 public class NotifierEventHandler implements Serializable {
4
5     private static final Logger LOGGER =
6         Logger.getLogger(NotifierEventHandler.class);
7
8     @Inject
9     @ClientFlag(ClientType.LIVE)
10    private ClientHolder clientHolder;
11
12
13
14    public void handle(@Observes NotifierEvent event) {
15        LOGGER.info("Get " + event.getBean() + " from the Inventory
16            system. Technical user: " +
17            this.clientHolder.getClient());
18    }
19 }
```

This ClientHolder injection defines which technical user will be used at this *deployment* (of course this is only kind of simulation, we do not perform any useful operation here).

# Fire an event

inv-ejbsevice project

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @EJB
5     private InventoryFinder finder;
6
7     @Inject
8     private InternationalInventoryItemConverter internationalConverter;
9
10    @Inject
11    private Event<NotifierEvent> notifier;
12
13    @Override
14    public InternationalInventoryItemBean getInternationalInventoryItemWithEvent(String
15        reference) throws AdaptorException {
16        final InternationalInventoryItemBean bean =
17            this.internationalConverter.to(this.finder.get(reference));
18        this.notifier.fire(new NotifierEvent(bean));
19        return bean;
20    }
21 }
```

InventoryFacadeImpl.java

# Produces

Using a factory during *injection*

## Issue

The introduced `ClientHolder` example is kind of simulation only. It shows how we can differentiate multiple implementations of a CDI *bean* in an elegant way without creating a lot of *qualifiers*. In the same simulation we are going to pop up an other issue as well: it could be cumbersome to modify the source code to change the injected `ClientHolder` (we have to recompile the source code). How we can create a `ClientHolder` which has an argument to define the name of the technical client? We would like to initialize that value via JNDI *tree* (or any other way).

Put the `CUSTOM ClientFlag` into that new implementation, but we have to clarify the following here:

- ▷ Beside this new `CUSTOM` parameterized `ClientHolder` we do not need the other ones.
- ▷ There are no connection between the parameterized CDI *bean* implementation and the *qualifier* + *enum* combination. We can use both techniques without the other.

The main issue here that the *container* cannot figure out what will be the value of the bean's parameter. Creating a **factory** is the solution, we will use the `javax.enterprise.inject` annotation to achieve that<sup>2</sup>.

<sup>2</sup>Do not mix the `javax.enterprise.inject.Produces` and the `javax.ws.rs.Produces` annotations

# CustomClientHolder

```
1 package hu.qvaevisz.inventory.ejbsevice.client;
2
3 public class CustomClientHolder extends AbstractClientHolder {
4
5     public CustomClientHolder(final String name) {
6         super(name.substring(0, 3).toUpperCase(), name);
7     }
8
9 }
```

CustomClientHolder.java

We cannot put the `@ClientFlag(ClientType.CUSTOM)` annotation into the class because the *container* does not know the value of the name *constructor* argument. We need a *factory* here.

# ClientHolderFactory

```
standalone.xml:  
<simple name="java:global/customClientName"  
value="Dolores M. Putto" />
```

The fourth type `ClientHolder` will be created with a *factory*. We are going to define a 'singleton' class (`@ApplicationScoped` annotation) where a `@Produces` + *qualifier* annotations will mark the appropriate `ClientHolder` instance. The *container* collects the methods which have the `@Produces` annotation and it will use them when somebody request a specific type of resource (most of the time these methods run only once).

Usage:

```
1 @Inject  
2 @ClientFlag(ClientType.CUSTOM)  
3 private ClientHolder clientHolder;
```

```
1 package hu.qwaevisz.inventory.ejbser  
2 [..]  
3 @ApplicationScoped  
4 public class ClientHolderFactory {  
5  
6     @Resource(lookup = "java:global/customClientName")  
7     private String clientName;  
8  
9     @Produces  
10    @ClientFlag(ClientType.CUSTOM)  
11    public ClientHolder getCustomClientHolder() {  
12        return new CustomClientHolder(this.clientName);  
13    }  
14 }
```

ClientHolderFactory.java



# Random number generator

## Subtask



**Subtask:** One of the most important functions in our *inventory* system is the possibility to **generate any number of random number**. It is not only good to present how we can inject random number as a resource, but also good example to show how we can force inject something, how we can initialize a resource more than one time.

During the subtask we will examine how we can *inject* simple values (numbers), which is not a syntactic challenge but it could present a special point of view.



## ConfigurationRestService

- ▷ <http://localhost:8080/inventory/api/configuration/numbers/{quantity}>
  - It will generate the given number of random number(s) between 0 and MAX where the MAX value is initialized via JNDI tree (JNDI name: `java:global/maxLength`).
- ▷ <http://localhost:8080/inventory/api/configuration/host>
  - It gives back the value of the host configuration parameter (the name is absolutely irrelevant). Originally the value of the host is initialized via JNDI tree (JNDI name: `java:global/inventoryhost`).

# Maximum value as a resource

```
1 package hu.qvaevisz.inventory.ejbservice.qualifier;
2 [..]
3 @Qualifier
4 @Retention(RetentionPolicy.RUNTIME)
5 @Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD,
6           ElementType.PARAMETER })
7 public @interface MaxNumber {
8 }
```

## MaxNumber.java

```
1 package hu.qvaevisz.inventory.ejbservice.service;
2 [..]
3 @ApplicationScoped
4 public class MaxNumberFactory {
5
6     @Resource(lookup = "java:global/maxNumber")
7     private int maxValue;
8
9     @Logged
10    @Produces
11    @MaxNumber
12    public int getMaxNumber() {
13        return this.maxValue;
14    }
15 }
```

**Important!** If we would like to refresh this value we have to restart the application server. If we would like to change this value at runtime we have to use an other technique (e.g. JMX).

## MaxNumberFactory.java

# Do we really need this complexity?

## WAT?

Do we introduced two new types in the system instead of creating a simple MAX constant? Indeed, it is difficult to beautify, from syntactical point of view we did that. But on the other hand this @MaxNumber is a resource of the system, and is later we would like to refactor that part of the code (e.g.: change this to a configurable variable), we will not have to change any other component of the system.

# Change the JNDI variable via JBoss CLI



```
java:global/maxLength
```

```
1 > jboss-cli --connect
2 [/] /subsystem=naming/binding=java\:global\/maxLength:read-resource
3 {
4     "outcome" => "success",
5     "result" => {
6         "binding-type" => "simple",
7         "cache" => undefined,
8         "class" => undefined,
9         "environment" => undefined,
10        "lookup" => undefined,
11        "module" => undefined,
12        "type" => "int",
13        "value" => "100"
14    }
15 }
16 [/]
17 /subsystem=naming/binding=java\:global\/maxLength:write-attribute(name=value,value=2000)
18 {
19     "outcome" => "success",
20     "response-headers" => {
21         "operation-requires-reload" => true,
22         "process-state" => "reload-required"
23     }
24 }
```

We have to escape the : and / characters of the JNDI name. The read and write operation names are **read-resource** and **write-attribute**.

**Important!** With the help of this technique we cannot rewrite the injected values in our application, even if we lookup the value from InitialContext. We must restart the application server.

We can browse the JNDI tree via JBoss *webconsole* as well:

<http://localhost:9990/console/App.html#naming> (Runtime | JNDI view).

# Random number as a resource

```
1 [..]
2 @Qualifier
3 @Retention(RetentionPolicy.RUNTIME)
4 @Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD,
5           ElementType.PARAMETER })
6 public @interface Random {}
```

## Random.java

```
1 package hu.qwaevisz.inventory.ejbservice.service;
2 [..]
3 @ApplicationScoped
4 public class RandomNumberFactory {
5
6     private final java.util.Random random = new
7         java.util.Random(System.currentTimeMillis());
8
9     @Inject
10    @MaxNumber
11    private int maxNumber;
12
13    @Logged
14    @Produces
15    @Random
16    public int next() {
17        return this.random.nextInt(this.maxNumber);
18    }
19 }
```

The `@MaxNumber` will be injected only one because we inject a simple `int` value. With the same technique the factory of the `@Random` number would be called only once. We need an other way to inject this resource.

# Forced *injection* a random number

inv-ejbsservice project

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @Inject
5     @Random
6     private Instance<Integer> randomNumber;
7
8     @Logged
9     @Override
10    public List<Integer> getRandomNumbers(final int quantity) throws AdaptorException {
11        final List<Integer> result = new ArrayList<>();
12        for (int i = 0; i < quantity; i++) {
13            result.add(this.randomNumber.get());
14        }
15        return result;
16    }
17
18 }
```

InventoryFacadeImpl.java

In case of the generic `Instance<T>` type we have to write the injected type to the place of the `T`, and when we call the `T.get()` method the initialization and injection will be always executed. We used the same generic type in case of dynamic injection of the CDI *beans*, but there we used the `select()` method instead of the `get()`.

# Configuration parameters

## Subtask



An application has a lot of configuration parameters which we would like to store e.g.: in JNDI tree. For example:

- ▷ `java:global/inventoryhost`, and
- ▷ `java:global/currency`

**Subtask:** For all of these values we can create a unique *qualifier* to use any of these values as a resource, but it would cause some confusion. A solution could be to use the `@Named( String )` annotation where we do not need to create new types.



# InventoryJndiPropertyProvider

```
1 package hu.qvaevisz.inventory.ejbservice.service;
2 [...]
3 public class InventoryJndiPropertyProvider {
4
5     @Resource(lookup = "java:global/inventoryhost")
6     private String inventoryHost;
7
8     @Resource(lookup = "java:global/currency")
9     private String currency;
10
11     @Produces
12     @Named("host")
13     public String getInventoryHost() {
14         return this.inventoryHost;
15     }
16
17     @Produces
18     @Named("currency")
19     public String getCurrency() {
20         return this.currency;
21     }
22 }
```

Instead of the a new *qualifier* we use the `@Named` annotation with a given argument. That is just the difference.

InventoryJndiPropertyProvider.java

# InventoryConfiguration

```
1 package hu.qwaevisz.inventory.ejbservice.service;
2
3 public class InventoryConfiguration {
4
5     private final String host;
6     private final String currency;
7
8     public InventoryConfiguration(String host, String currency) {
9         this.host = host;
10        this.currency = currency;
11    }
12
13    public String getHost() {
14        return this.host;
15    }
16
17    public String getCurrency() {
18        return this.currency;
19    }
20 }
```

Let us create a class which stores the configuration parameters. This class does not have default *constructor*, so we need a *factory* and the `@Produces` annotation to create/inject that.

InventoryConfiguration.java

# InventoryConfigurationProvider

```
1 package hu.qwaevisz.inventory.ejbservice.service;
2
3 import javax.enterprise.inject.Produces;
4 import javax.inject.Named;
5
6 public class InventoryConfigurationProvider {
7
8     @Produces
9     public InventoryConfiguration config(@Named("host") String host, @Named("currency")
10         String currency) {
11         return new InventoryConfiguration(host, currency);
12     }
13 }
```

## InventoryConfigurationProvider.java

We may use the *qualifier* not only beside the `@Inject` (but we can design the *factory* in that way as well). This solution (we annotate the arguments) could be convenient and clean in case of a `@Produces` method with arguments.

# Using the configuration parameter

inv-ejb-service project

```
1 @Stateless(mappedName = "ejb/inventoryFacade")
2 public class InventoryFacadeImpl implements InventoryFacade {
3
4     @Inject
5     private InventoryConfiguration configuration;
6
7     @Override
8     public String getHost() throws AdaptorException {
9         return this.configuration.getHost();
10    }
11
12 }
```

InventoryFacadeImpl.java

The CDI bean can be found by type, the *factory* method (which has the `@Produces` annotation) will be executed, and the `@Named` arguments will be injected automatically. The `@Named` arguments also have a *factory* method, its values initialized via JNDI.

Of course this size of complexity is not always necessary, primarily it is useful to separate the responsibility in an enterprise environment.