Inventory #maven

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 bea 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 webservices 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 (`InventoryType` *enum*) in `JSON` format.
- `http://localhost:8080/inventory/api/basic/list/{type}/{nameTerm}`
  - It gives back a list of *inventory* items of the given tpye 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*[1]).
- **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
`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.

[1] 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 configuraiton 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 *jar*s 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.CopyrightIntercepto
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 convinient 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 webservices

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 elegent 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 3rd 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 EJB*s).
- 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

```java
public interface InventoryItemBeanConverter {
  InventoryItemBean to(InventoryItem item);
  List<InventoryItemBean> to(List<InventoryItem> items);
}
```

There is no @Local annotation on the interface.

InventoryItemBeanConverter.java

```java
public class InventoryItemBeanConverterImpl implements InventoryItemBeanConverter {

  @Override
  public InventoryItemBean to(InventoryItem item) { [..] }

  @Override
  public List<InventoryItemBean> to(List<InventoryItem> items) {
    return items.stream().map(this::to).collect(Collectors.toList());
  }
}
```

There is no @Stateless annotation on the class.

InventoryItemBeanConverterImpl.java

```java
@Stateless(mappedName = "ejb/inventoryFacade")
public class InventoryFacadeImpl implements Inve

  @Inject
  private InventoryItemBeanConverter converter;

  [..]
}
```

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 *Container* 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 *scope* 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 *EntityManagers* in a `JTA` transaction: You can use them with plain `CDI`, but without the *container-managed* transactions it can get really monotonous duplicating the *UserTransaction* commit/rollback 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*).

Logging with *interceptor*

Subtask

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
           http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
5
6    <interceptors>
7      <class>hu.qwaevisz.inventory.ejbservice.interceptor.LoggedInterceptor</class>
8    </interceptors>
9
10 </beans>
```

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

Logging and measure the running time

```java
@Interceptor
public class LoggedInterceptor implements Serializable {

  private static final Logger LOGGER = Logger.getLogger(LoggedInterceptor.class);

  @AroundInvoke
  public Object logMethodInvocations(InvocationContext context) throws Exception {
    final StringBuilder info = new StringBuilder();
    info.append(context.getTarget().getClass().getName()).append(".").append(context.getMet
    final Object[] parameters = context.getParameters();
    if (parameters != null) {
      int i = parameters.length - 1;
      for (final Object parameter : parameters) {
        info.append(parameter.toString());
        if (i > 0) {
          info.append(",");
        }
        i--;
      }
    }
    info.append(")");
    LOGGER.info("Entering: " + info.toString());
    final long start = System.currentTimeMillis();
    final Object result = context.proceed();
    final long end = System.currentTimeMillis();
    LOGGER.info("Exiting: " + info.toString() + " - running time: " + (end - start) + "
        millisecond(s)");
    return result;
  }
}
```

LoggedInterceptor.java

Logged

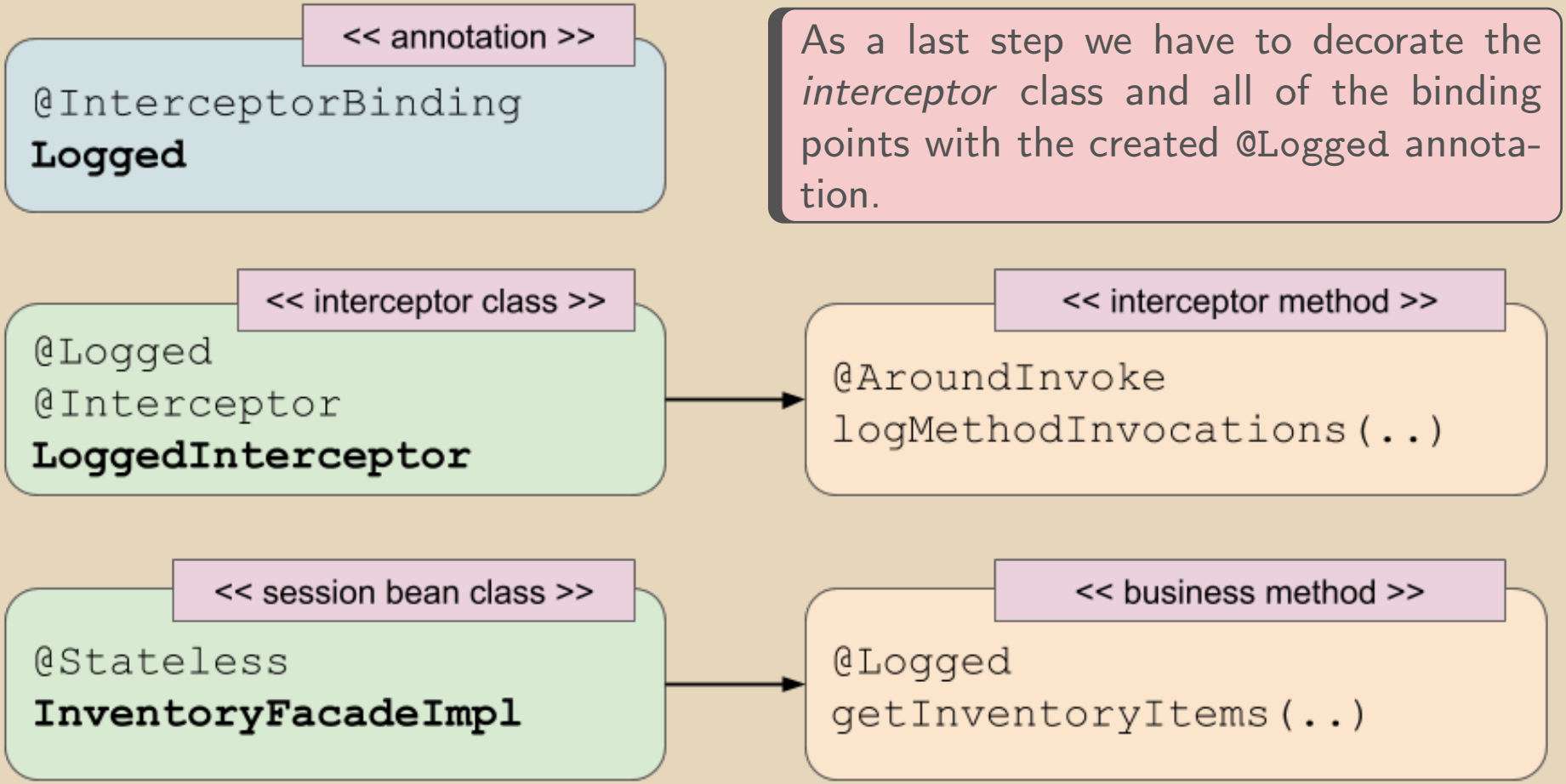
@InterceptorBinding annotation

```
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

Overview


<< annotation >>
@InterceptorBinding
Logged
As a last step we have to decorate the *interceptor* class and all of the binding points with the created @Logged annotation.
<< interceptor class >>
@Logged
@Interceptor
LoggedInterceptor
<< interceptor method >>
@AroundInvoke
logMethodInvocations(..)
<< session bean class >>
@Stateless
InventoryFacadeImpl
<< business method >>
@Logged
getInventoryItems(..)

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)
         throws Exception {
10     [..]
11   }
12
13 }
```

LoggedInterceptor.java

Binding interceptor with the help of CDI

```java
@Stateless(mappedName = "ejb/inventoryFacade")
public class InventoryFacadeImpl implements InventoryFacade {

  @EJB
  private InventoryFinder finder;

  @Inject
  private InventoryItemBeanConverter converter;

  @Logged
  @Override
  @Interceptors({ CopyrightInterceptor.class })
  public List<InventoryItemBean> getInventoryItems(final InventoryType type) throws
       AdaptorException {
    return this.converter.to(this.finder.list(type));
  }

  @Inject
  private InventorySearch search;

  @Logged
  @Override
  public List<InventoryItemBean> getInventoryItems(final InventoryType type, final
       String nameTerm) throws AdaptorException {
    return this.converter.to(this.search.list(type, nameTerm));
  }

  [..]
}
```

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

InventoryFacadeImpl.java

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 webservices, 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 `PromotionalCo`
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 becaue 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,
      ElementType.FIELD, ElementType.PARAMETER })
6 public @interface Standard {
7
8 }
```

Standard.java

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

Discount.java

CostService and its implementations

```java
package hu.qwaevisz.inventory.ejbservice.cost;
public interface CostService {
  int getCost(int originalValue);
}
```

CostService.java

```java
[..]
@Standard
public class NormalCostService implements CostService {
  @Override
  public int getCost(int originalValue) {
    return originalValue;
  }
}
```

NormalCostService.java

```java
[..]
@Discount
public class PromotionalCostService implements CostService {
  private static final float DISCOUNT_PERCENT = 20;
  @Override
  public int getCost(int originalValue) {
    return Math.round(originalValue * (1 - (DISCOUNT_PERCENT / 100)));
  }
}
```

PromotionalCostService.java

Qualifier based *injection*

```java
@Stateless(mappedName = "ejb/inventoryFacade")
public class InventoryFacadeImpl implements InventoryFacade {

  @EJB
  private InventoryFinder finder;

  @Inject
  private InventoryItemBeanConverter converter;

  @Inject
  @Discount
  private CostService costService;

  @Logged
  @Override
  public InventoryItemBean buyInventoryItem(String reference) throws AdaptorException {
    final InventoryItemBean bean = this.converter.to(this.finder.get(reference));
    bean.setValue(this.costService.getCost(bean.getValue()));
    return bean;
  }
}
```

InventoryFacadeImpl.java

Testing the promotional price

via RESTful webservices

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)
         throws AdaptorException {
11     final InventoryItemBean bean = this.converter.to(this.finder.get(reference));
12     bean.setValue(this.getCostService(pricing).getCost(bean.getValue()));
13     return bean;
14   }
15   private CostService getCostService(PricingStrategy pricing) {
16     CostService service = null;
17     switch (pricing) {
18       case STANDARD:
19         service = this.dynamicCostService.select(new AnnotationLiteral<Standard>() {
20           private static final long serialVersionUID = 1L;
21         }).get();
22         break;
23       case DISCOUNT:
24         service = this.dynamicCostService.select(new DiscountQualifier()).get();
25         break;
26     }
27     return service;
28   }
29 }
```

InventoryFacadeImpl.java

DiscountQualifier *selector* in a separate class

```java
package hu.qwaevisz.inventory.ejbservice.qualifier;

import javax.enterprise.util.AnnotationLiteral;

public class DiscountQualifier extends
    AnnotationLiteral<Discount> implements Discount {

  private static final long serialVersionUID = 1L;

}
```

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.

Defining JNDI variable

JBoss

```
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"
               type="int" />
10       </bindings>
11     </subsystem>
12     [..]
13   </profile>
14   [..]
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

```java
package hu.qwaevisz.inventory.ejbservice.domain;

import org.codehaus.jackson.annotate.JsonProperty;

public class InternationalInventoryItemBean {

  private String label;
  private String price;

  public InternationalInventoryItemBean(String label, String price) { [..] }

  @JsonProperty("product")
  public String getLabel() { return this.label; }

  public void setLabel(String label) { this.label = labe

  @JsonProperty("price")
  public String getPrice() { return this.price;}

  public void setPrice(String price) { this.price = pric

  @Override
  public String toString() { return this.label; }
}
```

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 / *root* project (dependencyManagement)

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  <project [..]> [..]
2    <properties>
3      <version.jackson>1.9.9</version.jackson>
4    </properties> [..]
5    <dependencyManagement>
6      <dependencies>
7        [..]
8        <dependency>
9          <groupId>org.codehaus.jackson</groupId>
10         <artifactId>jackson-core-asl</artifactId>
11         <version>${version.jackson}</version>
12         <scope>provided</scope>
13       </dependency>
14       <dependency>
15         <groupId>org.codehaus.jackson</groupId>
16         <artifactId>jackson-mapper-asl</artifactId>
17         <version>${version.jackson}</version>
18         <scope>provided</scope>
19       </dependency>
20     </dependencies>
21   </dependencyManagement>
22 </project>
```

InternationalInventoryItemConverter

```
1  package hu.qwaevisz.inventory.ejbservice.converter;
2
3  import
       hu.qwaevisz.inventory.ejbservice.domain.InternationalInventoryIte
4  import hu.qwaevisz.inventory.persistence.domain.InventoryItem;
5
6  public interface InternationalInventoryItemConverter {
7
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")
5    private String currency;
6
7    @Resource(lookup = "java:global/exchangeRate")
8    private int exchangeRate;
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 }
```

InternationalInventoryItemConverterImpl.java

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

```java
package hu.qwaevisz.inventory.ejbservice.domain;
public class Client {

  private final String reference;
  private final String name;

  public Client(final String reference, String name) {
    this.reference = reference;
    this.name = name;
  }

  public String getReference() { return this.reference; }

  public String getName() { return this.name; }

  @Override
  public String toString() {
    return this.name + " (" + this.reference + ")";
  }
}
```

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, ElementTyp
       })
6  public @interface ClientFlag {
7
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

```java
package hu.qwaevisz.inventory.ejbservice.client;
import hu.qwaevisz.inventory.ejbservice.domain.Client;
public interface ClientHolder {
  Client getClient();
}
```

ClientHolder.java

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

```java
public abstract class AbstractClientHolder imp
    ClientHolder {

  private final Client client;

  public AbstractClientHolder(final String reference, String
      name) {
    this.client = new Client(reference, name);
  }

  @Override
  public Client getClient() { return this.client; }
}
```

AbstractClientHolder.java

Implementation for providing technical clients

```java
@ClientFlag(ClientType.LIVE)
public class LiveClientHolder extends AbstractClientHolder {
  public LiveClientHolder() {
    super("MCF012", "Matthew C. Flores");
  }
}
```

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.

```java
@ClientFlag(ClientType.SANDBOX)
public class SandboxClientHolder extends AbstractClientHolder {
  public SandboxClientHolder() {
    super("SVW987", "Scott V. Wright");
  }
}
```

SandboxClientHolder.java

```java
@ClientFlag(ClientType.TEST)
public class TestClientHolder extends AbstractClientHolder {
  public TestClientHolder() {
    super("TEST", "Test System");
  }
}
```

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.ejbservice.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 =
         Logger.getLogger(ReportEventHandler.class);
6
7    public void handle(@Observes NotifierEvent event) {
8      LOGGER.info("Report this event: " + event);
9    }
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 =
         Logger.getLogger(NotifierEventHandler.class);
6
7    @Inject
8    @ClientFlag(ClientType.LIVE)
9    private ClientHolder clientHolder;
10
11
12
13
14   public void handle(@Observes NotifierEvent event) {
15     LOGGER.info("Get " + event.getBean() + " from the Inventory
           system. Technical user: " +
           this.clientHolder.getClient());
16   }
17 }
```

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).

NotifierEventHandler.java

Fire an event

inv-ejbservice 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
         reference) throws AdaptorException {
15     final InternationalInventoryItemBean bean =
           this.internationalConverter.to(this.finder.get(reference));
16     this.notifier.fire(new NotifierEvent(bean));
17     return bean;
18   }
19 }
```

InventoryFacadeImpl.java

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[2].

[2] Do not mix the `javax.enterprise.inject.Produces` and the `javax.ws.rs.Produces` annotations

CustomClientHolder

```java
package hu.qwaevisz.inventory.ejbservice.client;

public class CustomClientHolder extends AbstractClientHolder {

  public CustomClientHolder(final String name) {
    super(name.substring(0, 3).toUpperCase(), name);
  }

}
```

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 'singletion' class (`@ApplicationScoped` annotation) where a `@Produces` + *qualifier* annoations 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: On of the most important function 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 initialized 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/maxNumber).
- 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).

Maxium value as a resource

```java
package hu.qwaevisz.inventory.ejbservice.qualifier;
[..]
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD,
     ElementType.PARAMETER })
public @interface MaxNumber {

}
```

MaxNumber.java

```java
package hu.qwaevisz.inventory.ejbservice.service;
[..]
@ApplicationScoped
public class MaxNumberFactory {

  @Resource(lookup = "java:global/maxNumber")
  private int maxValue;

  @Logged
  @Produces
  @MaxNumber
  public int getMaxNumber() {
    return this.maxValue;
  }
}
```

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/maxNumber

```
> jboss-cli --connect
[/] /subsystem=naming/binding=java\:global\/maxNumber:read-resource
{
    "outcome" => "success",
    "result" => {
        "binding-type" => "simple",
        "cache" => undefined,
        "class" => undefined,
        "environment" => undefined,
        "lookup" => undefined,
        "module" => undefined,
        "type" => "int",
        "value" => "100"
    }
}
[/]
      /subsystem=naming/binding=java\:global\/maxNumber:write-attribute(name=value,value=200
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

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,
       ElementType.PARAMETER })
5  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
         java.util.Random(System.currentTimeMillis());
7
8    @Inject
9    @MaxNumber
10   private int maxNumber;
11
12   @Logged
13   @Produces
14   @Random
15   public int next() {
16     return this.random.nextInt(this.maxNumber);
17   }
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.

RandomNumberFactory.java

Forced *injection* a random number

inv-ejbservice 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

```java
package hu.qwaevisz.inventory.ejbservice.service;
[..]
public class InventoryJndiPropertyProvider {

  @Resource(lookup = "java:global/inventoryhost")
  private String inventoryHost;

  @Resource(lookup = "java:global/currency")
  private String currency;

  @Produces
  @Named("host")
  public String getInventoryHost() {
    return this.inventoryHost;
  }

  @Produces
  @Named("currency")
  public String getCurrency() {
    return this.currency;
  }
}
```

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

InventoryJndiPropertyProvider.java

InventoryConfiguration

```java
package hu.qwaevisz.inventory.ejbservice.service;

public class InventoryConfiguration {

  private final String host;
  private final String currency;

  public InventoryConfiguration(String host, String currency) {
    this.host = host;
    this.currency = currency;
  }

  public String getHost() {
    return this.host;
  }

  public String getCurrency() {
    return this.currency;
  }
}
```

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

```java
package hu.qwaevisz.inventory.ejbservice.service;

import javax.enterprise.inject.Produces;
import javax.inject.Named;

public class InventoryConfigurationProvider {

  @Produces
  public InventoryConfiguration config(@Named("host") String host, @Named("currency")
      String currency) {
    return new InventoryConfiguration(host, currency);
  }

}
```

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-ejbservice 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.