

CDI Interceptors

Example `cdi-interceptors` can be browsed at <https://github.com/apache/tomee/tree/master/examples/cdi-interceptors>

Let's write a simple application that would allow us to book tickets for a movie show. As with all applications, logging is one cross-cutting concern that we have. Apart from that, there are some methods in our application, that can be accessed only in the working hours. If accessed at non-working-hours we'll throw out an `AccessDeniedException`.

How do we mark which methods are to be intercepted? Wouldn't it be handy to annotate a method like

```
@Log
public void aMethod(){...}
```

or

```
@TimeRestricted
public void bMethod(){...}
```

Let's create these annotations that would "mark" a method for interception.

```
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
public @interface Log {
}
```

And

```
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
public @interface TimeRestricted {
}
```

Sure, you haven't missed the `@InterceptorBinding` annotation above! Now that our custom annotations are created, let's attach them (or to use a better term for it, "bind them") to interceptors.

So here's our logging interceptor. An `@AroundInvoke` method and we are almost done.

```

@Interceptor
@Log //binding the interceptor here. now any method annotated with @Log would be
intercepted by logMethodEntry
public class BookForAShowLoggingInterceptor implements Serializable {
    private static final long serialVersionUID = 8139854519874743530L;
    private Logger logger = Logger.getLogger("BookForAShowApplicationLogger");
    @AroundInvoke
    public Object logMethodEntry(InvocationContext ctx) throws Exception {
        logger.info("Before entering method:" + ctx.getMethod().getName());
        InterceptionOrderTracker.getMethodsInterceptedList().add(ctx.getMethod()
.getName());
        InterceptionOrderTracker.getInterceptedByList().add(this.getClass()
.getSimpleName());
        return ctx.proceed();
    }
}

```

Now the `@Log` annotation we created is bound to this interceptor. (Likewise we bind `@TimeRestrict` for `TimeBasedRestrictingInterceptor`. See links below for source)

That done, let's annotate at class-level or method-level and have fun intercepting!

```

@Log
@Stateful
public class BookForAShowOneInterceptorApplied implements Serializable {
    private static final long serialVersionUID = 6350400892234496909L;
    public List<String> getMoviesList() {
        List<String> moviesAvailable = new ArrayList<String>();
        moviesAvailable.add("12 Angry Men");
        moviesAvailable.add("Kings speech");
        return moviesAvailable;
    }
    public Integer getDiscountedPrice(int ticketPrice) {
        return ticketPrice - 50;
    }
    // assume more methods are present
}

```

The `@Log` annotation applied at class level denotes that all the methods should be intercepted with `BookForAShowLoggingInterceptor`.

Before we say "all done" there's one last thing we are left with! To enable the interceptors!

Lets quickly put up a `beans.xml` file like the following in `src/main/resources/META-INF/beans.xml`:

```
<beans>
  <interceptors>
    <class>org.superbiz.cdi.bookshow.interceptors.BookForAShowLoggingInterceptor
    </class>
    <class>org.superbiz.cdi.bookshow.interceptors.TimeBasedRestrictingInterceptor
    </class>
  </interceptors>
</beans>
```

By default, a bean archive has no enabled interceptors bound via interceptor bindings. An interceptor must be explicitly enabled by listing its class in the `beans.xml`.

Those lines in `beans.xml` not only "enable" the interceptors, but also define the "order of execution" of the interceptors.

The order in which a method is annotated has no real significance. Eg:

```
@TimeRestrict
@Log
void cMethod(){}


```

Here the `'BookForAShowLoggingInterceptor'` would be applied first and then `'TimeBasedRestrictingInterceptor'`

So now you know that the order is only determined by the order of definition in `'beans.xml'`. Interceptors which occur earlier in the list are called first.

Also note that a method can be marked `for` interception by multiple interceptors by applying multiple annotations as above.

This brings us to another question. In the above `case` there were two interceptors applied together. What `if` I would require about 4 such interceptors that would go along.... Having to annotate so much makes the code a little clumsy?

No worries! Just create a custom annotation which inherits from others

```
@Inherited
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
@Log
@TimeRestricted
public @interface TimeRestrictAndLog {
}


```

This is interceptor binding inheritance.

The code below demonstrates the many cases that we have discussed.

Not to forget, the old style binding with `@Interceptors(WhicheverInterceptor.class)` is also supported. Have a look at `BookForAShowOldStyleInterceptorBinding` where the comments explain how the newer way discussed above is better.

The Code

BookForAShowOneInterceptorApplied

`BookForAShowOneInterceptorApplied` shows a simple `@Log` interceptor applied.

```
package org.superbiz.cdi.bookshow.beans;

import org.superbiz.cdi.bookshow.interceptorbinding.Log;

import javax.ejb.Stateful;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

@Log
@Stateful
public class BookForAShowOneInterceptorApplied implements Serializable {
    private static final long serialVersionUID = 6350400892234496909L;

    public List<String> getMoviesList() {
        List<String> moviesAvailable = new ArrayList<String>();
        moviesAvailable.add("12 Angry Men");
        moviesAvailable.add("Kings speech");
        return moviesAvailable;
    }

    public Integer getDiscountedPrice(int ticketPrice) {
        return ticketPrice - 50;
    }
}
```

BookForAShowTwoInterceptorsApplied

`BookForAShowTwoInterceptorsApplied` shows both `@Log` and `@TimeRestricted` being applied.

```

package org.superbiz.cdi.bookshow.beans;

import org.superbiz.cdi.bookshow.interceptorbinding.Log;
import org.superbiz.cdi.bookshow.interceptorbinding.TimeRestricted;

import javax.ejb.Stateful;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

@Log
@Stateful
public class BookForAShowTwoInterceptorsApplied implements Serializable {
    private static final long serialVersionUID = 6350400892234496909L;

    public List<String> getMoviesList() {
        List<String> moviesAvailable = new ArrayList<String>();
        moviesAvailable.add("12 Angry Men");
        moviesAvailable.add("Kings speech");
        return moviesAvailable;
    }

    @TimeRestricted
    public Integer getDiscountedPrice(int ticketPrice) {
        return ticketPrice - 50;
    }
}

```

BookShowInterceptorBindingInheritanceExplored

`BookShowInterceptorBindingInheritanceExplored` shows how `@TimeRestrictAndLog` (interceptor-binding-inheritance) can be used as an alternative for annotating a method with multiple annotations explicitly.

```

package org.superbiz.cdi.bookshow.beans;

import org.superbiz.cdi.bookshow.interceptorbinding.TimeRestrictAndLog;

import javax.ejb.Stateful;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

@Stateful
public class BookShowInterceptorBindingInheritanceExplored implements Serializable {
    private static final long serialVersionUID = 6350400892234496909L;

    public List<String> getMoviesList() {
        List<String> moviesAvailable = new ArrayList<String>();
        moviesAvailable.add("12 Angry Men");
        moviesAvailable.add("Kings speech");
        return moviesAvailable;
    }

    @TimeRestrictAndLog
    public Integer getDiscountedPrice(int ticketPrice) {
        return ticketPrice - 50;
    }
}

```