@AccessTimeout the Meta-Annotation Way

Example access-timeout-meta can be browsed at https://github.com/apache/tomee/tree/master/examples/access-timeout-meta

Any annotation that takes parameters can benefit from meta-annotations. Here we see how `@AccessTimeout` can be far more understandable and manageable through meta-annotations. We'll use the [access-timeout](../access-timeout/README.html) example as our use-case.

The value of the parameters supplied to `@AccessTimeout` have a dramatic affect on how what that annotation actually does. Moreover, `@AccessTimeout` has one of those designs where `-1` and `0` have signifcantly different meanings. One means "wait forever", the other means "never wait". Only a lucky few can remember which is which on a daily basis. For the rest of us it is a constant source of bugs.

Meta-Annotations to the rescue!

Creating the Meta-Annotations

As a matter of best-practices, we will put our meta-annotations in a package called `api`, for this example that gives us `org.superbiz.accesstimeout.api`. The package `org.superbiz.api` would work just as well.

The basic idea is to have a package where "approved' annotations are used and to prohibit usage of the non-meta versions of the annotations. All the real configuration will then be centralized in the `api` package and changes to timeout values will be localized to that package and automatically be reflected throuhout the application.

An interesting side-effect of this approach is that if the `api` package where the meta-annotation definitions exist is kept in a separate jar as well, then one can effectively change the configuration of an entire application by simply replacing the `api` jar.

@Metatype <small>The "root" Meta-Annotation</small>

As with all meta-annotation usage, you first need to create your own "root" meta-annotation. This is as easy as creating an annotation named `Metatype` that is annotated with itself and has `ElementType.ANNOTATION_TYPE` as its target.

```
package org.superbiz.accesstimeout.api;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Metatype
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Metatype {
}
```

@AwaitNever

When the `@AccessTimeout` annotation has the value of `0` that has the implication that one should never wait to access the bean. If the bean is busy, the caller will immediately receive an `ConcurrentAccessException`. This is hard to remember and definitely not self-documenting for those that never knew the details.

To create a meta-annotation version of `@AccessTimeout(0)` we simply need to think of a good annotation name, create that annotation, and annotate it with both `@AccessTimeout` and `@Metatype`

```
package org.superbiz.accesstimeout.api;

import javax.ejb.AccessTimeout;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Metatype
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

@AccessTimeout(0)
public @interface AwaitNever {
}
```

@AwaitForever

Just as `0` carries the special meaning of "never wait", a value of `-1` means "wait forever."

As long as we're being picky, which we can be with meta-annotations, Technically "wait forever" is not the best description. The actual methods of the `javax.util.concurrent` APIs use "await" rather than "wait". One (wait) perphaps implies a command to wait, which this is not, and the other

(await) perhaps better implies that waiting is possible but not a certainty. So we will use "await" in our annotation names.

We make our own `@AwaitForever` and annotate it with `@AccessTimeout(0)` and `@Metatype`

```
package org.superbiz.accesstimeout.api;

import javax.ejb.AccessTimeout;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Metatype
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

@AccessTimeout(-1)
public @interface AwaitForever {
}
```

@AwaitBriefly

Non `-1` and `0` values to `@AccessTimeout` actually involve the full breadth of the annotation. Here is where you get to specify the maximum number minutes, seconds, milliseconds, etc. where one might await access to the bean instance.

```
@Metatype
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})

@AccessTimeout(value = 5, unit = TimeUnit.SECONDS)
public @interface AwaitBriefly {
}
```

Configuration vs Operation

Once you create a few meta-annotations and the fun becomes common-place, questoins start to raise in your mind on how to best get the benefits of meta-annotations.

You have to really start thinking about how you want to approach your usage of meta-annotation and really put your designer hat on. The fundamental question is **configuration vs operation** and the answer is subjective; how much flexibility do you want to design into your applications and where?

Configuration names <small>describing the configuration</small>

The simplest approach is to name your meta-annotations after the **configuration** they encapsulate. We've been following that format so far with `@AwaitNever` and `@AwaitForever` to clearly reflect the contents of each meta-annotation (`@AccessTimeout(-1)` and `@AccessTimeout(0)` respectively).

The **cons** of this approach is that should you want to change the configuration of the application by only changing the meta-annotations — this is one of the potential benefits of meta-annotations. Certainly, the `@AwaitNever` meta-annotation can have no other value than `0` if it is to live up to its name.

Operation names <small>describing the code</small>

The alternate approach is to name your meta-annotations after the **operations** they apply to. In short, to describe the code itself and not the configuration. So, names like `@OrderCheckTimeout` or `@TwitterUpdateTimeout`. These names are configuration-change-proof. They would not change if the configuration changes and in fact they can facilitate finder-grained control over the configuration of an application.

The **cons** are of course it is requires far more deliberation and consideration, not to mention more annotations. Your skills as an architect, designer and ability to think as a administrator will be challenged. You must be good at wearing your dev-opts hat.

Pragmatism <small>best of both worlds</small>

Fortunately, meta-annotations are recursive. You can do a little of both.

```
@Metatype
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

@AwaitBriefly
public @interface TwitterUpdateTimeout {
}
```

Of course you still need to be very deliberate on how your annotations are used. When using a "configuration" named meta-annotation in code it can help to say to yourself, "I do not want to reconfigure this later." If that doesn't feel quite right, put the extra effort into creating an operation named annotation and use in that code.

Applying the Meta-Annotations

Putting it all together, here's how we might apply our meta-annotations to the [access-timeout](../access-timeout/README.html) example.

Before

```
package org.superbiz.accesstimeout;

import javax.ejb.AccessTimeout;
import javax.ejb.Asynchronous;
import javax.ejb.Lock;
import javax.ejb.Singleton;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import static javax.ejb.LockType.WRITE;

/**
 * @version $Revision$ $Date$
 */
@Singleton
@Lock(WRITE)
public class BusyBee {

    @Asynchronous
    public Future stayBusy(CountDownLatch ready) {
        ready.countDown();

        try {
            new CountDownLatch(1).await();
        } catch (InterruptedException e) {
            Thread.interrupted();
        }

        return null;
    }

    @AccessTimeout(0)
    public void doItNow() {
        // do something
    }

    @AccessTimeout(value = 5, unit = TimeUnit.SECONDS)
    public void doItSoon() {
        // do something
    }

    @AccessTimeout(-1)
    public void justDoIt() {
        // do something
    }

}
```

After

```
package org.superbiz.accesstimeout;

import org.superbiz.accesstimeout.api.AwaitBriefly;
import org.superbiz.accesstimeout.api.AwaitForever;
import org.superbiz.accesstimeout.api.AwaitNever;

import javax.ejb.Asynchronous;
import javax.ejb.Lock;
import javax.ejb.Singleton;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Future;

import static javax.ejb.LockType.WRITE;

/**
 * @version $Revision$ $Date$
 */
@Singleton
@Lock(WRITE)
public class BusyBee {

    @Asynchronous
    public Future stayBusy(CountDownLatch ready) {
        ready.countDown();

        try {
            new CountDownLatch(1).await();
        } catch (InterruptedException e) {
            Thread.interrupted();
        }

        return null;
    }

    @AwaitNever
    public void doItNow() {
        // do something
    }

    @AwaitBriefly
    public void doItSoon() {
        // do something
    }

    @AwaitForever
    public void justDoIt() {
        // do something
```

```
    }

}
```