Deadbolt 2

Powerful authorization for your Play 2 application

Steve Chaloner

Deadbolt 2

Powerful authorization for your Play 2 application

Steve Chaloner

This book is for sale at http://leanpub.com/deadbolt-2

This version was published on 2016-02-18

Tweet This Book!

Please help Steve Chaloner by spreading the word about this book on Twitter!

The suggested hashtag for this book is #deadbolt2.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#deadbolt2

It’s all for Greet & Lotte.

Contents

1. Deadbolt 2

1.1 About the author

Steve Chaloner has been a software developer, consultant and mentor since 1999. He specialises in Java and Scala, but believes in using the right tool for the job. The right tool for him, for web-based applications, is the Play framework.

He is the author of several open-source projects, the most successful of which is Deadbolt, an authorization system for Play.

In 2011, he was selected as one of the Expert Reviewers for the Play! Framework Cookbook, along with the creator of Play and one of its oldest contributors. Since then, he has been an expert reviewer for four other books on Play, covering Java, Scala and reactive programming.

In 2012, Steve co-founded the Belgian Play Framework User Group with Ben Verbeken. This later merged with BeScala, the Belgian Scala User Group.

Steve blogs at http://www.objectify.be and tweets at https://twitter.com/steve_objectify.

1.2 About the book

This book was written in Markdown, and its source files are available on GitHub at https://github.com/schaloner/deadbolt-2-guide.

2. Introduction

At the date of writing this, the small code experiment that turned into Deadbolt is nearly five years old. In that time, it has grown in capabilities, power and - hopefully - usefulness. It has certainly reached the point where a few badly-written lines of text can adequately cover its features and use. Once I factored in that it now has support for Java and Scala, and that its new architecture allows for easy extension with other languages, it became clear that a short booklet (or booklet-like document) was required.

I hope this document (booklet-like or otherwise) will prove useful as you integrate Deadbolt 2 into your application.

2.1 History

Back in September 2010, I was embarking on my first project using the Play! Framework (for fans of unnecessary detail, it was version 1.03.2, back when it still had an exclamation mark) and discovering the Secure module it shipped with was unsuitable for the required authorization. As a result, Deadbolt 1.0 was written to provide AND/OR/NOT support for roles. Sometime later, dynamic rule support was added and other new features would be released as use cases and bug reports cropped up.

The user guide for Deadbolt 1 - which I can still highly recommend if you need authorization support in your Play 1 apps - starts with this:

> Deadbolt is an authorization mechanism for defining access rights to certain controller methods or parts of a view using a simple AND/OR/NOT syntax. It is based on the original Secure module that comes with the Play! framework.
>
> Note that Deadbolt doesn't provide authentication! You can still use the existing Secure module alongside Deadbolt to provide authentication, and in cases where authentication is handled outside your app you can just hook up the authorization mechanism to whatever auth system is used.

How much of this still holds true for Deadbolt 2? More than 50% and less than 100%, give or take.

- Deadbolt is still used for authorization
- It can control access to controllers
- It can control access to templates
- The capabilities have expanded beyond the original role-based static checks
- Deadbolt 2 is based on Deadbolt 1, so it's related to the old Secure module in spirit, if not in implementation
- You can (or should be able to) combine Deadbolt 2 with any authentication system

Deadbolt 2 v1.0 was released at roughly the same time as Play 2.0, and was essentially the logic of Deadbolt 1 exposed in the Play 2 style. Nine months after that initial release - nine months, I should add, of woefully inadequate Scala support - I re-designed the architecture to a more modular approach, and made a few small changes to the API to remove anachronistic elements.

There is now a core module written in Java, and separate idiomatic modules for Java and Scala. This is slightly different to the architecture of Play 2 itself, where the core and the Scala API are co-located.

2.2 Java versus Scala

I have tried my best, within the constraints of both languages and my knowledge of them, to give equal capabilities to each version of Deadbolt 2. Scala is generally detailed after Java in this book for two reasons.

The first reason is the alphabet.

The second is that by writing the Scala section last, I have a chance to increase my knowledge of a language that is frequently beautiful and occasionally a little bit like an ice-cream headache.

2.3 Versions

The examples in this book are for Play 2.4, and use the following versions of Deadbolt

- `"be.objectify" %% "deadbolt-java" % "2.4.3"`
- `"be.objectify" %% "deadbolt-scala" % "2.4.3"`

User demand lead to some API-level changes in the 2.4.4 Scala release; these changes will be detailed in the Scala section.

2.4 Target audience

This book works on the following assumptions:

- you have a working knowledge of Play
- you know Java or Scala (or both)

If you want to learn Play, the Play website[1] is the best place to start.

There are also several books that you might want to check out.

- Play For Scala[2]

[1] https://playframework.com/documentation/2.4.x/Home
[2] https://www.manning.com/books/play-for-scala

- Play For Java[3]
- Learning Play Framework 2[4]
- Reactive Web Applications[5] covers more advanced usage of Play

2.5 Acknowledgements

Nothing which attempts to be useful can gestate in a vacuum. Many people donated their time, knowledge and critical capabilities to help make this book more useful.

- Peter Hilton (@PeterHilton[6]) - software developer, speaker, writer, author of "Play for Scala" and beer lover.
- Francis De Brabandere (@somatik[7]) - software developer, technology enthusiast, start-up founder and beer lover.
- Daryl Greensill (@BigClumsyOaf[8]) - todo: check one-line write-up

2.6 Feedback

Any and all feedback of this is greatly appreciated. You can send feedback using…

- The book's repository[9] on GitHhub
- By emailing deadbolt-book@objectify.be[10]

[3]https://www.manning.com/books/play-for-java
[4]https://www.packtpub.com/web-development/learning-play-framework-2
[5]https://www.manning.com/books/reactive-web-applications
[6]https://twitter.com/PeterHilton
[7]https://twitter.com/somatik
[8]https://twitter.com/BigClumsyOaf
[9]https://github.com/schaloner/deadbolt-2-guide
[10]mailto:deadbolt-book@objectify.be

3. Root concepts

Deadbolt is centered around a single idea - constraining access to a resource to a specific group of users. I've had several e-mails from developers who have thought that Deadbolt had a "restrict from" approach, and therefore could not understand why the authorization was failing so spectacularly; to forestall future questions about this, I want to make it completely clear - Deadbolt uses "restrict to". For example, a controller action annotated with `@Restrict(@Group("foo"))` would only allow users with the "foo" role to access the method.

Two mechanisms are provided to declare these constraints - one at the template level and another at the controller level. In each case, there are differences between how these are applied in Java and Scala applications, so specific details will be provided in later chapters. The capabilities of each version are roughly similar, taking into account the idiosyncrasies of each language.

3.1 Template-level constraints

For a Play application that uses server-side rendering, Deadbolt provides several template tags that will conceal or reveal DOM elements based on your specifications.

A couple of basic use cases are

- Only displayed a "Log in" link if there is no user present
- Even if a user is logged in, only display an "Administration" link if the user has administrative privileges

However, it is **extremely** important to note that using these tags will only give you a cleaner UI, one that is better tailored to the user's privileges. It will **not** secure your server-side code in any way except - possibly - obscurity. Server-side routes can be invoked from outside of templates by changing the URL in the browser's address bar, using command-line tools such as cURL and many other ways.

If you have seen the original Dawn Of The Dead (Romero, 1978), you may remember the protagonists concealing the entrance to their living quarters using a panel of painted hardboard. There are no additional defensive layers behind this concealment. When a zombified protagonist breaks through the hardboard, knowing there is something he wants on the other side, all security is gone. Minutes later, there's blood everywhere and the survivors have to flee. If you haven't seen it, apologies for the spoiler.

Template security is like painted hardboard - the features it offers are certainly nice to have, but a further level of defensive depth is required. For this, you need controller action security - otherwise, the zombies will get you.

3.2 Controller-level restrictions

The controller layer is most vunerable part of your application to external attack, because that is the part that's visible to whichever networks it is on. Attack in this sense may be a concious attack on your system, or inadvertant damage caused by unauthorized users who are otherwise authenticated in your system. Deadbolt can help with both of these scenarios in the same way, by limiting the capabilities of any given user at the application level.

Controller authorization blocks or allows access to an action. Whereas template restrictions are essentially a boolean evaluation - "if user satisfies these conditions, then…", controller authorization is quite a bit more powerful. Specifically, while an authorized result is generated from your application code, unauthorized results can be customised as required; you can return any status code you like along with any content you like. If you're feeling particularly nasty, why not send a 302 redirect to a not-suitable-for-work website? If you want to, the option is there.

3.3 Core entities

Deadbolt has three interfaces which can be used to represent authorization entities in your application - `Subject`, `Role` and `Permission`.

Subject

The core entity of Deadbolt is the `be.objectify.deadbolt.core.models.Subject` interface. This should be implemented by whatever your application considers to be an authorizable entity - in other words, a user. Its sole purpose is to give access to the rights and permissions held by the user.

```java
public interface Subject
{
    List<? extends Role> getRoles();

    List<? extends Permission> getPermissions();

    String getIdentifier();
}
```

While it is possible to not implement the `Role` and `Permission` interfaces - for example, if you only use dynamic security - it is highly recommended that you always implement `Subject`.

`Subject` was originally known as `RoleHolder`, but this swiftly became an anacronysm as Deadbolt gained capabilities beyond checking roles. As of Deadbolt 2.0, `RoleHolder` became ´Subject´.

Role

A `Role` is essentially a wrapper around a string. It is the primary entity for the `Restrict` and `Restrictions` constraints. Role A is equal to Role B, even if they are different objects, if they have **exactly** the same name. Role names should be case-aware, so "Admin" is not the same as "admin".

As `Role` is an interface, it can be implemented as a class or an enum.

If you do not require roles in your application, you do not need to implement this interface - just return an empty list from `Subject#getRoles`.

Permission

A `Permission`, just like a `Role`, is essentially a wrapper around a string. It is the primary entity for `Pattern` constraints, and has different interpretations depending on the type of pattern constraint that is being applied to it. For example, a `PatternType.EQUALITY` test would perform a case-sensitive comparison between a user's permissions and the test value. A `PatternType.REGEX` would assess a user's permissions in the context of regular expressions, and so on.

As `Permission` is an interface, it can be implemented as a class or an enum.

If you do not require permissions in your application, you do not need to implement this interface - just return an empty list from `Subject#getPermissions`.

3.4 Hooks

There are two hooks that can be used to integrate Deadbolt into your application - `DeadboltHandler` and `DynamicResourceHandler`. In the Java version, these are represented by interfaces; in the Scala version, they are traits. There are some small differences between them caused by design differences with the Java and Scala APIs themselves, so exact breakdowns of these types will be covered in the language-specific sections. For now, it's enough to remind ourselves of where we are working in terms of a HTTP request.

A HTTP request has a life cycle. At a high level, it is

- Sent
- Received
- Processed
- Answered

The *processed* point is where our web applications live. In a sense, the high level life cycle is repeated again here, as the request is sent from the container into the application, received by the app, processed and answered. Controller constraints occur at the point where the container (the Play server, in this case) hands the request over to the application; templates work during the processing phase as a response body is rendered. Both places are where any `DeadboltHandler` and `DynamicResourceHandler` instances are active.

I Deadbolt for Java

4. Using Deadbolt 2 with Play 2 Java projects

Deadbolt 2 for Java provides an idiomatic API for dealing with Java controllers and templates rendered from Java controllers in Play applications. It takes advantage of the features such as access to the HTTP context - vital, of course, for a framework that says it embraces HTTP - to give access to the current request and session, and the annotation-driven interceptor support.

4.1 The Deadbolt Handler

For any module - or framework - to be useable, it must provide a mechanism by which it can be hooked into your application. For Deadbolt, the central hook is the `be.objectify.deadbolt.java.DeadboltHandl` interface. The four methods defined by this interface are crucial to Deadbolt - for example, `DeadboltHandler#getSubject` gets the current user (or subject, to use the correct security terminology), whereas `DeadboltHandler#onAccessFailure` is used to generate a response when authorization fails.

DeadboltHandler implementations should be stateless.

For each method, an `F.Promise` is returned. If the promise may complete to have an empty value, e.g. calling `getSubject` when no subject is present, the return type is `F.Promise<Optional>`.

Despite the use of the definite article in the section title, you can have as many Deadbolt handlers in your app as you wish.

Performing pre-constraint tests

Before a constraint is applied, the `F.Promise<Optional<Result>> beforeAuthCheck(Http.Context context)` method of the current handler is invoked. If the resulting promise completes to a non-empty `Optional`, the target action is not invoked and instead the result of `beforeAuthCheck` is used for the HTTP response; if the resulting promise completes to an empty `Optional` the action is invoked with the Deadbolt constraint applied to it.

Obtaining the subject

To get the current subject, the `F.Promise<Optional<Subject>> getSubject(Http.Context context)` method is invoked. Returning an empty `Optional` indicates there is no subject present - this is a valid scenario.

Dealing with authorization failure

When authorization fails, the `F.Promise<Result> onAccessFailure(Http.Context context, String content)` method is used to obtain a result for the HTTP response. The result required from the `F.Promise` returned from this method is a regular `play.mvc.Result`, so it can be anything you chose. You might want to return a 403 forbidden, redirect to a location accessible to everyone, etc.

Dealing with dynamic constraints

Dynamic constraints, which are `Dynamic` and `Pattern.CUSTOM` constraints, are dealt with by implementations of `DynamicResourceHandler`; this will be explored in a later chapter. For now, it's enough to say `F.Promise<Optional<DynamicResourceHandler>> getDynamicResourceHandler(Http.Context context)` is invoked when a dynamic constraint it used.

4.2 Expose your DeadboltHandlers with a HandlerCache

Unlike earlier versions of Deadbolt, in which handlers were declared in `application.conf` and created reflectively, Deadbolt now uses dependency injection to achieve the same functionality in a type-safe and more flexible manner. Various components of Deadbolt, which will be explored in later chapters, require an instance of `be.objectify.deadbolt.java.cache.HandlerCache` - however, no such implementations are provided.

Instead, you need to implement your own version. This has two requirements:

- You have a get() method which returns the application-wide default handler
- You have an apply(String handlerKey) method which returns a named handler

Here's one possible implementation, using hard-coded handlers.

```
@Singleton
public class MyHandlerCache implements HandlerCache {

    private final Map<String, DeadboltHandler> handlers = new HashMap<>();

    // handler keys is an application-specific enum
    public MyHandlerCache() {
        // See below regarding the default handler
        handlers.put(HandlerKeys.DEFAULT.key, new MyDeadboltHandler());
        handlers.put(HandlerKeys.ALT.key, new MyAlternativeDeadboltHandler());
        handlers.put(HandlerKeys.BUGGY.key, new BuggyDeadboltHandler());
        handlers.put(HandlerKeys.NO_USER.key, new NoUserDeadboltHandler());
    }

    @Override
    public DeadboltHandler apply(final String key) {
        return handlers.get(key);
    }

    @Override
    public DeadboltHandler get() {
        return handlers.get(HandlerKeys.DEFAULT.key);
    }
}
```

One interesting (and annoying) quirk is the way in which template and controller constraints obtain the default handler. The template constraints are written in Scala, so the `HandlerCache#get()` method can be used. Controllers, on the other hand, are configured via annotations and it's not possible to have a default value of `null` for an annotation value and so the standard handler name of `defaultHandler` is used. In the example above, `HandlerKeys.DEFAULT.key` uses this value, as declared as `be.objectify.deadbolt.java.ConfigKeys#DEFAULT_HANDLER_KEY`.

Finally, create a small module which binds your implementation.

```java
package com.example.modules

import be.objectify.deadbolt.java.cache.HandlerCache;
import play.api.Configuration;
import play.api.Environment;
import play.api.inject.Binding;
import play.api.inject.Module;
import scala.collection.Seq;
import security.MyHandlerCache;

import javax.inject.Singleton;

public class CustomDeadboltHook extends Module {
    @Override
    public Seq<Binding<?>> bindings(final Environment environment,
                                    final Configuration configuration) {
        return seq(bind(HandlerCache.class).to(MyHandlerCache.class).in(Singleton.class));
    }
}
```

4.3 application.conf

Declare the necessary modules

Both `be.objectify.deadbolt.java.DeadboltModule` and your custom bindings module must be declared in the configuration.

```
play {
  modules {
    enabled += be.objectify.deadbolt.java.DeadboltModule
    enabled += com.example.modules.CustomDeadboltHook
  }
}
```

Tweaking Deadbolt

Deadbolt Java-specific configuration lives in the `deadbolt.java` namespace.

There are two settings:

- `deadbolt.java.view-timeout`
  - The millisecond timeout applied to blocking calls when rendering templates. Defaults to 1000ms.
- `deadbolt.java.cache-user`
  - A flag to indicate if the subject should be cached on a per-request basis. Defaults to false.

Personally, I prefer the HOCON (Human-Optimized Config Object Notation) syntax supported by Play, so I would recommend the following:

```
deadbolt {
  java {
    cache-user=true
    view-timeout=500
  }
}
```

JPA

After all the effort I made to ensure Deadbolt is as non-blocking as possible, JPA emerged from the mist to bite me on the ass; entity managers, it seems, do not like the kind of multi-threaded usage implicit in Play's asynchronous behaviour.

Luckily, a solution is at hand. Less luckily, it's blocking.

To address this, you can put Deadbolt into blocking mode - this ensures all DB calls made in the Deadbolt layer are made from the same thread; this has performance implications, but it's unavoidable with JPA.

To switch to blocking mode, set `deadbolt.java.blocking` to true in your configuration.

The default timeout is 1000 milliseconds - to change this, use `deadbolt.java.blocking-timeout` in your configuration.

This example configuration puts Deadbolt in blocking mode, with a timeout of 2500 milliseconds:

```
deadbolt {
    java {
        blocking=true
        blocking-timeout=2500
    }
}
```

5. Java controller constraints

If you like annotations in Java code, you're in for a treat. If you don't, this may be a good time to consider the Scala version.

One very important point to bear in mind is the order in which Play evaluates annotations. Annotations applied to a method are applied to annotations applied to a class. This can lead to situations where Deadbolt method constraints deny access because information from a class constraint. See the section on deferring method-level interceptors for a solution to this.

As with the previous chapter, here is a a breakdown of all the Java annotation-driven interceptors available in Deadbolt Java, with parameters, usages and tips and tricks.

Static constraints

Static constraints, are implemented entirely within Deadbolt because it can finds all the information needed to determine authorization automatically. For example, if a constraint requires two roles, "foo" and "bar" to be present, the logic behind the `Restrict` constraint knows it just needs to check the roles of the current subject.

The static constraints available are

- SubjectPresent
- SubjectNotPresent
- Restrict
- Pattern - when using EQUALITY or REGEX

Dynamic constraints

Dynamic constraints are, as far as Deadbolt is concerned, completely arbitrary; they're handled by implementations of `DynamicResourceHandler`.

The dynamic constraints available are

- Dynamic
- Pattern - when using CUSTOM

5.1 SubjectPresent

`SubjectPresent` is one of the simplest constraints offered by Deadbolt. It checks if there is a subject present, by invoking `DeadboltHandler#getSubject` and allows access if the result is an `Optional` containing a value.

`@SubjectPresent` can be used at the class or method level.

Parameter	Type	Default	Notes
content	String	””	A hint to indicate the content expected in the response. This value will be passed to `DeadboltHandler#onAccessFailure`. The value of this parameter is completely arbitrary.
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
deferred	boolean	false	If true, the interceptor will not be applied until a `DeadboltDeferred` annotation is encountered.

Require a subject for all actions in a controller

```java
@SubjectPresent
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method will not be invoked unless there is a subject present
        ...
    }

    public F.Promise<Result> search()
    {
        // this method will not be invoked unless there is a subject present
        ...
    }
}
```

Require a subject for specific actions in a controller

```
// Deny access to a single method of a controller unless there is a user present
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method is accessible to anyone
        ...
    }

    @SubjectPresent
    public F.Promise<Result> search()
    {
        // this method will not be invoked unless there is a subject present
        ...
    }
}
```

5.2 SubjectNotPresent

`SubjectNotPresent` is the opposite in functionality of `SubjectPresent`. It checks if there is a subject present, by invoking `DeadboltHandler#getSubject` and allows access only if the result is an empty `Optional`.

`@SubjectNotPresent` can be used at the class or method level.

Parameter	Type	Default	Notes
content	String	””	A hint to indicate the content expected in the response. This value will be passed to `DeadboltHandler#onAccessFailure`. The value of this parameter is completely arbitrary.
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
deferred	boolean	false	If true, the interceptor will not be applied until a `DeadboltDeferred` annotation is encountered.

Require NO subject for all actions in a controller

```java
@SubjectNotPresent
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method will not be invoked if there is a subject present
        ...
    }

    public F.Promise<Result> search()
    {
        // this method will not be invoked if there is a subject present
        ...
    }
}
```

Require NO subject for specific actions in a controller

```
// Deny access to a single method of a controller if there is a user present
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method is accessible to anyone
        ...
    }

    @SubjectNotPresent
    public F.Promise<Result> search()
    {
        // this method will not be invoked unless there is not a subject present
        ...
    }
}
```

5.3 Restrict

The Restrict constraint requires that a) there is a subject present, and b) the subject has ALL the roles specified in the at least one of the `Groups` in the constraint. The key thing to remember about `Restrict` is that it ANDs together the role names within a group and ORs between groups.

Notation

The role names specified in the annotation can take two forms.

1. Exact form - the subject must have a role whose name matches the required role exactly. For example, for a constraint `@Restrict("foo")` the Subject *must* have a `Role` whose name is "foo".
2. Negated form - if the required role starts starts with a !, the constraint is negated. For example, for a constraint `@Restrict("!foo")` the Subject *must not* have a `Role` whose name is "foo".

`@Restrict` can be used at the class or method level.

Parameter	Type	Default	Notes
value	Group[]		For each `Group`, the roles that must (or in the case of negation, must not) be held by the `Subject`. When the restriction is applied, the `Group` instances are OR'd together.
content	String	””	A hint to indicate the content expected in the response. This value will be passed to `DeadboltHandler#onAccessFailure`. The value of this parameter is completely arbitrary.
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
deferred	boolean	false	If true, the interceptor will not be applied until a `DeadboltDeferred` annotation is encountered.

Both roles are required for all actions in a controller

```java
@Restrict(@Group{"editor", "viewer"})
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method will not be invoked unless the subject has editor and viewer roles
        ...
    }

    public F.Promise<Result> search()
    {
        // this method will not be invoked unless the subject has editor and viewer roles
        ...
    }
}
```

Example 2

Each action has different role requirements

```java
public class MyController extends Controller
{
    @Restrict(@Group("editor"))
    public F.Promise<Result> edit()
    {
        // this method will not be invoked unless the subject has editor role
        ...
    }

    @Restrict(@Group("view"))
    public F.Promise<Result> view()
    {
        // this method will not be invoked unless the subject has viewer role
        ...
    }
}
```

Negated roles

```java
@Restrict(@Group({"editor", "!viewer"}))
public class MyController extends Controller
{
    public F.Promise<Result> edit()
    {
        // this method will not be invoked unless the subject has editor role AND does NOT\
 have the viewer role
        ...
    }

    public F.Promise<Result> view()
    {
        // this method will not be invoked unless the subject has editor role AND does NOT\
 have the viewer role
        ...
    }
}
```

Require the 'editor' OR 'viewer' roles

```
@Restrict({@Group("editor"), @Group("viewer")})
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method will not be invoked unless the subject has editor or viewer roles
        ...
    }

    public F.Promise<Result> search()
    {
        // this method will not be invoked unless the subject has editor or viewer roles
        ...
    }
}
```

Require the subject to have (customer AND viewer) OR (support AND viewer) roles

```
@Restrict({@Group({"customer", "viewer"}), @Group({"support", "viewer"})})
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method will not be invoked unless the subject has customer and viewer,
        // or support and viewer roles
        ...
    }
}
```

Require the subject to have (customer AND NOT viewer) OR (support AND NOT viewer) roles

```
@Restrict({@Group("customer", "!viewer"), @Group("support", "!viewer")})
public class MyController extends Controller
{
    public F.Promise<Result> index()
    {
        // this method will not be invoked unless the subject has customer but not viewer \
roles,
        // or support but not viewer roles
        ...
    }
}
```

5.4 Dynamic

The most flexible constraint - this is a completely user-defined constraint that uses DynamicResourceHandler#isAllowed to determine access.

`@Dynamic` can be used at the class or method level.

Parameter	Type	Default	Notes
value	String		The name of the constraint.
meta	String		Additional information passed into `isAllowed`.
content	String	””	A hint to indicate the content expected in the response. This value will be passed to `DeadboltHandler#onAccessFailure`. The value of this parameter is completely arbitrary.
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
deferred	boolean	false	If true, the interceptor will not be applied until a `DeadboltDeferred` annotation is encountered.

Using a user-defined test to determine access

```
@Dynamic(name = "name of the test")
public F.Promise<Result> someMethod()
{
    // the method will execute if the user-defined test returns true
}
```

5.5 Pattern

This uses the Subjects Permissions to perform a variety of checks.

`@Pattern` uses a `Subject`'s `Permission`s to perform a variety of checks. The check depends on the pattern type.

- EQUALITY - the subject must have a permission whose value is exactly the same as the `value` parameter
- REGEX - the subject must have a permission which matches the regular expression given in the `value` parameter
- CUSTOM - the `DynamicResourceHandler#checkPermission` function is used to determine access

It's possible to invert the constraint by setting the `invert` parameter to true. This changes the meaning of the constraint in the following way.

- EQUALITY - the subject must NOT have a permission whose value is exactly the same as the `value` parameter
- REGEX - the subject must have NO permissions that match the regular expression given in the `value` parameter
- CUSTOM - the `DynamicResourceHandler#checkPermission` function, where the OPPOSITE of the Boolean resolved from the function is used to determine access

`@Pattern` can be used at the class or method level.

A note on inverted custom constraints

When using `invert` and `CUSTOM`, care must be taken to ensure the desired result is achieved. For example, consider an implementation of `checkPermission` where a subject is required and that subject must have a certain attribute; access is denied if there is no subject present OR that attribute is not present. When inverting the result, access would be allowed if the subject is not present OR that attribute is not present. This is because when `invert` is true, the boolean resolved from `checkPermission` is negated.

If you only mean to allow access if a subject is present but does not have the attribute, you will need to engage in some annoying double negation.

Just before `checkPermission` is invoked, the value of `invert` is stored in the arguments of the HTTP context using the `ConfigKeys.PATTERN_INVERT` key; you can use this to determine what to return.

In the following example, one of four things can happen: - A subject is present, and it satisfies the arbitrary test. - A subject is present, and it does not satisfy the arbitrary test - A subject is not present, and invert is true - A subject is not present, and invert is false - There is also a fallback assumption that invert is false if `ConfigKeys.PATTERN_INVERT` is not in the context; Deadbolt guarantees this will not happen, but it doesn't hurt to make sure

Inverting the test, not the need to have a subject

```
@Override
public F.Promise<Boolean> checkPermission(final String permissionValue,
                                          final DeadboltHandler deadboltHandler,
                                          final Http.Context ctx)
{
    // just checking for zombies...just like I do every night before I go to bed
    return deadboltHandler.getSubject(ctx)
                          .map(option ->
        option.map(subject -> subject.getPermissions()
                                     .stream()
                                     .filter(perm -> perm.getValue().contains("zombie"))
                                     .count() > 0)
              .orElseGet(() -> (Boolean) ctx.args.getOrDefault(ConfigKeys.PATTERN_INVERT,
                                                               false)));
}
```

So, in cases where we have a subject we just test like usual; the negation of the result, if required by `invert`, will be handled by Deadbolt. In cases where there is no subject, we return the value of `invert` itself - if it's false, no negation will be internally applied, and if it's true it will be negated to false and access denied. Thus, the test for `perm.getValue().contains("zombie")` is separated from the requirement to have a subject.

TL;DR Double negation sucks.

Parameter	Type	Default	Notes
value	String		The pattern value. Its context depends on the pattern type.
patternType	PatternType	EQUALITY	Additional information passed into `isAllowed`.
content	String	””	A hint to indicate the content expected in the response. This value will be passed to `DeadboltHandler#onAccessFailure`. The value of this parameter is completely arbitrary.
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
deferred	boolean	false	If true, the interceptor will not be applied until a `DeadboltDeferred` annotation is encountered.
invert	boolean	false	Invert the result of the test.

Testing for equality of permissions

```java
@Pattern("admin.printer")
public F.Promise<Result> someMethodA()
{
    // subject must have a permission with the exact value "admin.printer"
}
```

Testing for regex matching of permissions

```java
@Pattern(value = "(.)*\.printer", patternType = PatternType.REGEX)
public F.Promise<Result> someMethodB()
{
    // subject must have a permission that matches the regular expression (without quotes)\
 "(.)*\.printer"
}
```

Using a user-defined test to determine access

```java
@Pattern(value = "something arbitrary", patternType = PatternType.CUSTOM)
public F.Promise<Result> someMethodC()
{
    // the checkPermssion method of the current handler's DynamicResourceHandler will be u\
sed.  This is a user-defined test
}
```

Inverting the test

```java
@Pattern(value = "(.)*\.printer", patternType = PatternType.REGEX, invert = true)
public F.Promise<Result> someMethodB()
{
    // subject must have no permissions that end in .printer
}
```

5.6 Unrestricted

`Unrestricted` allows you to over-ride more general constraints, i.e. if a controller is annotated with `@SubjectPresent` but you want an action in there to be accessible to everyone.

```java
@SubjectPresent
public class MyController extends Controller
{
    public F.Promise<Result> foo()
    {
        // a subject must be present for this to be accessible
    }

    @Unrestricted
    public F.Promise<Result> bar()
    {
        // anyone can access this action
    }
}
```

You can also flip this on its head, and use it to show that a controller is explicitly unrestricted - used in this way, it's a marker of intent rather than something functional. Because method-level constraints are evaluated first, you can have still protected actions within an `@Unrestricted` controller.

```java
@Unrestricted
public class MyController extends Controller
{
    @SubjectPresent
    public F.Promise<Result> foo()
    {
        // a subject must be present for this to be accessible
    }

    public F.Promise<Result> bar()
    {
        // anyone can access this action
    }
}
```

`@Unrestricted` can be used at the class or method level.

Parameter	Type	Default	Notes
content	String	””	A hint to indicate the content expected in the response. This value will be passed to `DeadboltHandler#onAccessFailure`. The value of this parameter is completely arbitrary.
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
deferred	boolean	false	If true, the interceptor will not be applied until a `DeadboltDeferred` annotation is encountered.

5.7 Deferring method-level annotation-driven interceptors

Play executes method-level annotations before controller-level annotations. This can cause issues when, for example, you want a particular action to be applied for method and before the method annotations. A good example is `Security.Authenticated(Secured.class)`, which sets a user's user name for `request().username()`. Combining this with method-level annotations that require a user would fail, because the user would not be present at the time the method interceptor is invoked.

One way around this is to apply `Security.Authenticated` on every method, which violates DRY and causes bloat.

```java
public class DeferredController extends Controller
{
    @Security.Authenticated(Secured.class)
    @Restrict(value="admin")
    public F.Promise<Result> someAdminFunction()
    {
        return F.Promise.promise(accessOk::render)
                        .map(Results::ok);
    }

    @Security.Authenticated(Secured.class)
    @Restrict(value="editor")
    public F.Promise<Result> someEditorFunction()
    {
        return F.Promise.promise(accessOk::render)
                        .map(Results::ok);
    }
}
```

A better way is to set the `deferred` parameter of the Deadbolt annotation to `true`, and then use `@DeferredDeadbolt` at the controller level to execute the method-level annotations at controller-annotation time. Since annotations are processed in order of declaration, you can specify `@DeferredDeadbolt` after `@Security.Authenticated` and so achieve the desired effect.

Deferring authorization checks

```java
@Security.Authenticated(Secured.class)
@DeferredDeadbolt
public class DeferredController extends Controller
{
    @Restrict(value="admin", deferred=true)
    public F.Promise<Result> someAdminFunction()
    {
        return F.Promise.promise(accessOk::render)
                        .map(Results::ok);
    }

    @Restrict(value="editor", deferred=true)
    public F.Promise<Result> someEditorFunction()
    {
        return F.Promise.promise(accessOk::render)
                        .map(Results::ok);
    }
}
```

Specifying a controller-level restriction as `deferred` will work, if the annotation is higher in the annotation list than `@DeferredDeadbolt` annotation, but this is essentially pointless. If your constraint is already at the class level, there's no need to defer it. Just ensure it appears below any other annotations you wish to have processed first.

5.8 Invoking DeadboltHandler#beforeAuthCheck independently

`DeadboltHandler#beforeAuthCheck` is invoked by each interceptor prior to running the actual constraint tests. If the method call returns an empty `Optional`, the constraint is applied, otherwise the contained in the `Optional` result is returned. The same logic can be invoked independently, using the `@BeforeAccess` annotation, in which case the call to `beforeRoleCheck` itself becomes the constraint. Or, to say it in code,

```
result = preAuth(true, ctx, deadboltHandler)
            .flatMap(preAuthResult -> preAuthResult.map(F.Promise::pure)
                                                           .orElseGet(() -> delegate.call(ctx))
```

`@BeforeAccess` can be used at the class or method level.

Parameter	Type	Default	Notes
handlerKey	String	“defaultHandler”	The name of a handler in the `HandlerCache`
alwaysExecute	boolean	true	By default, if another Deadbolt action has already been executed in the same request and has allowed access, `beforeAuthCheck` will not be executed again. Set this to true if you want it to execute unconditionally.
deferred	boolean	false	If true, the interceptor will not be applied until a DeadboltDeferred annotation is encountered.

5.9 Customising the inputs of annotation-driven actions

One of the problems with Deadbolt’s annotations is they require strings to specify, for example, role names or pattern values. It would be far safer to use enums, but this is not possible for a module - it would completely kill the generic applicability of the annotations. If Deadbolt shipped with an enum containing roles, how would you extend it? You would be stuck with whatever was specified, or forced to fork the codebase and customise it. Similarly, annotations can neither implement interfaces or be extended.

To address this situation, Deadbolt has three constraints whose inputs can be customised to some degree. The trick lies, not with inheritence, but delegation and wrapping. The constraints are

- Restrict
- Dynamic
- Pattern

Here, I’ll explain how to customise the Restrict constraint to use enums as annotation parameters, but the principle is the same for each constraint.

To start, create an enum that represents your roles.

Using an enum to define roles

```java
public enum MyRoles implements Role
{
    foo,
    bar,
    hurdy;

    @Override
    public String getRoleName()
    {
        return name();
    }
}
```

To allow the AND/OR syntax that `Restrict` uses, another annotation to group them together is required.

```java
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRolesGroup
{
    MyRoles[] value();
}
```

Next, create a new annotation to drive your custom version of Restrict. Note that an array of `MyRoles` values can be placed in the annotation. The standard `Restrict` annotation is also present to provide further configuration. This means your customisations are minimised.

Defining a custom entry point

```java
@With(CustomRestrictAction.class)
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
@Documented
@Inherited
public @interface CustomRestrict
{
    MyRolesGroup[] value();

    Restrict config();
}
```

The code above contains `@With(CustomRestrictAction.class)` in order to specify the action that should be triggered by the annotation. This action can be implemented as follows.

Mapping custom roles to Deadbolt's requirements

```java
@Override
public Result call(Http.Context context) throws Throwable
{
    final CustomRestrict outerConfig = configuration;
    final RestrictAction restrictAction = new RestrictAction(configuration.config(),
                                                             this.delegate)

    {
        @Override
        public List<String[]> getRoleGroups()
        {
            final List<String[]> roleGroups = new ArrayList<>();
            for (MyRolesGroup mrg : outerConfig.value())
            {
                final List<String> group = new ArrayList<>();
                for (MyRoles role : mrg.value())
                {
                    group.add(role.getName());
                }
                roleGroups.add(group.toArray(group.size()));
            }
            return roleGroups;
        }
    };
    return restrictAction.call(context);
}
```

To use your custom annotation, you apply it as you would any other Deadbolt annotation.

Using custom roles

```java
@CustomRestrict(value = {MyRoles.foo, MyRoles.bar}, config = @Restrict(""))
public static F.Promise<Result> customRestrictOne()
{
    return F.Promise.promise(accessOk::render)
                    .map(Results::ok);
}
```

Each customisable action has one or more extension points. These are

Action class	**Extension points**
RestrictAction	* List<String[]> getRoleGroups()
DynamicAction	* String getValue() * String getMeta()
PatternAction	* String getValue()

6. Deadbolt Java Templates

Before you get your hopes up that Play templates can be written in Java, I'm talking about Scala templates rendered from Java controllers.

Using template constraints, you can exclude portions of templates from being generated on the server-side. Template constraints have the same possibilities as controller constraints.

This is not a client-side DOM manipulation, but rather the exclusion of content when templates are rendered. This also means that any logic inside the constrained content will not execute if authorization fails.

A subject is required for the content to be rendered

```
@subjectPresent {
    <!-- paragraphs will not be rendered, satellites will not be repositioned -->
    <!-- and light speed will not be engaged if no subject is present -->
    <p>Let's see what this thing can do...</p>
    @repositionSatellite
    @engageLightSpeed
}
```

One important thing to note here is that templates are blocking, so any Futures used need to be completed for the resuly to be used in the template constraints. As a result, each constraint can take a function that expresses a Long, which is the millisecond value of the timeout. It defaults to 1000 milliseconds, but you can change this globally by setting the `deadbolt.java.view-timeout` value in your `application.conf`.

Handlers

By default, template constraints use the default Deadbolt handler, as obtained via `<YourDeadboltHandlerImpl>#get()` but as with controller constraints you can pass in a specific handler. The cleanest way to do this is to pass the handler into the template and then pass it into the constraints.

Fallback content

Each constraint has an `xOr` variant, which allows you to render content in place of the unauthorized content. This takes the form `<constraint>Or`, for example `subjectPresentOr`

Providing fallback content when a subject is not present

```
@subjectPresentOr {
    <button>Log out</button>
} {
    <button>Log in</button>
}
```

In each case, the fallback content is defined as a second `Content` block following the primary body.

Timeouts

Because templates use blocking calls when rendering, the promises returned from the Deadbolt handler, etc, need to be completed during the rendering process. A timeout, with a default value of 1000ms, is used to wait for the completion but you may want to change this. You can do this in two ways.

Set a global timeout

If you want to change the default timeout, define `deadbolt.java.view-timeout` in your configuration and give it a millisecond value, e.g.

Define timeouts in milliseconds

```
deadbolt {
  java {
    view-timeout=1500
  }
}
```

Use a supplier to provide a timeout

All Deadbolt templates have a `timeout` parameter which defaults to returning the app-wide value - 1000L if nothing else if defined, otherwise whatever `deadbolt.java.view-timeout` is set to. But - and here's the nice part - the `timeout` parameter is not a `Long` but rather a `Supplier<Long>`. This means you can use a timeout that fluctuates based on some metric - say, the number of timeouts that occur during template rendering.

How do I know if timeouts are occurring?

That's a good question. And the answer is - you need to implement `be.objectify.deadbolt.java.TemplateFailur` and bind it using a module; see "Expose your DeadboltHandlers with a HandlerCache" section in chapter 4 for more details on this. If you re-use that chapter 4 module, the binding will look something like this.

Declaring a template failure listener

```
public Seq<Binding<?>> bindings(final Environment environment,
                                final Configuration configuration) {
    return seq(bind(HandlerCache.class).to(MyHandlerCache.class).in(Singleton.class),
               bind(TemplateFailureListener.class).to(MyTemplateFailureListener.class).in(\
Singleton.class));
}
```

Making it a singleton allows you to keep a running count of the failure level; if you're using it for other purposes, then scope it accordingly.

6.1 SubjectPresent

Sometimes, you don't need fine-grained checked - you just need to see if there **is a** user present.

- be.objectify.deadbolt.java.views.html.subjectPresent
- be.objectify.deadbolt.java.views.html.subjectPresentOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.get()	The handler to use to apply the constraint.
timeout	() -> Long	A function returning `deadbolt.java.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Examples

Using the default Deadbolt handler

```
@import be.objectify.deadbolt.java.views.html.{subjectPresent, subjectPresentOr}

@subjectPresent() {
   This content will be present if handler#getSubject results in a Some
}

@subjectPresentOr() {
   This content will be present if handler#getSubject results in a Some
} {
        fallback content
}
```

Using a specific Deadbolt handler

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.{subjectPresent, subjectPresentOr}

@subjectPresent(handler = handler) {
    This content will be present if handler#getSubject results in a Some
}

@subjectPresentOr(handler = handler) {
    This content will be present if handler#getSubject results in a Some
} {
        fallback content
}
```

6.2 SubjectNotPresent

Sometimes, you don't need fine-grained checked - you just need to see if there **is no** user present

- be.objectify.deadbolt.java.views.html.subjectNotPresent
- be.objectify.deadbolt.java.views.html.subjectNotPresentOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.get()	The handler to use to apply the constraint.
timeout	() -> Long	A function returning `deadbolt.java.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Examples

Using the default Deadbolt handler

```
@import be.objectify.deadbolt.java.views.html.{subjectNotPresent, subjectNotPresentOr}

@subjectNotPresent() {
    This content will be present if handler#getSubject results in a None
}

@subjectNotPresentOr() {
    This content will be present if handler#getSubject results in a None
} {
        fallback content
}
```

Using a specific Deadbolt handler

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.{subjectNotPresent, subjectNotPresentOr}

@subjectNotPresent(handler = handler) {
    This content will be present if handler#getSubject results in a None
}

@subjectNotPresentOr(handler = handler) {
    This content will be present if handler#getSubject results in a None
} {
        fallback content
}
```

6.3 Restrict

Use `Subjects` `Roles` to perform AND/OR/NOT checks. The values given to the builder must match the `Role.name` of the subject's roles.

- be.objectify.deadbolt.java.views.html.restrict
- be.objectify.deadbolt.java.views.html.restrictOr

AND is defined as an `Array[String]`, OR is a `List[Array[String]]`, and NOT is a rolename with a `!` preceding it.

`anyOf` and `allOf` are convenience functions for creating a `List[Array[String]]` and an `Array[String]`. You can import both of them using `@import be.objectify.deadbolt.core.utils.TemplateUtils.{` `allOf}`.

In earlier versions of Deadbolt, `anyOf` and `allOf` were called the more succinct and less readable `la` (list of arrays) and `as` (array of strings). These methods are still available, but are deprecated.

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.get()	The handler to use to apply the constraint.
roles	List[Array[String]]		The AND/OR/NOT restrictions. One array defines an AND, multiple arrays define OR. See notes on `anyOf` and `allOf` above.
timeout	() -> Long	A function returning `deadbolt.java.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Examples

The subject is obtained from the default handler, and must have the foo role

```
@import be.objectify.deadbolt.java.views.html.restrict

@restrict(roles = anyOf(allOf("foo"))) {
    Subject requires the foo role for this to be visible
}
```

The subject is obtained from the default handler, and must have the foo AND bar role

```
@import be.objectify.deadbolt.java.views.html.restrict

@restrict(roles = anyOf(allOf("foo", "bar")) {
     Subject requires the foo AND bar roles for this to be visible
}
```

The subject is obtained from the default handler, and must have the foo OR bar role

```
@import be.objectify.deadbolt.java.views.html.restrict

@restrict(roles = anyOf(allOf("foo"), allOf("bar"))) {
     Subject requires the foo OR bar role for this to be visible
}
```

Providing fallback content

```
@import be.objectify.deadbolt.java.views.html.restrictOr

@restrictOr(roles = anyOf(allOf("foo", "bar"))) {
     Subject requires the foo AND bar roles for this to be visible
} {
        Subject does not have the necessary roles
}
```

Getting the subject via a specific Deadbolt handler

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.restrict

@restrict(roles = anyOf(allOf("foo"), allOf("bar")), handler = handler) {
     Subject requires the foo OR bar role for this to be visible
}
```

6.4 Pattern

Use the `Subject`s `Permission`s to perform a variety of checks.

- be.objectify.deadbolt.java.views.html.pattern
- be.objectify.deadbolt.java.views.html.patternOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.get()	The handler to use to apply the constraint.
value	String		The value of the pattern, e.g. a regex or a precise match.
patternType	PatternType	PatternType.EQUALITY	
timeout	() -> Long	A function returning `deadbolt.java.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Examples

The subject must have a permission with the exact value 'admin.printer'

```
@import be.objectify.deadbolt.java.views.html.pattern

@pattern(value = "admin.printer") {
    Subject must have a permission with the exact value "admin.printer" for this to be vis\
ible
}
```

The subject must have a permission that matches the specified regular expression

```
@import be.objectify.deadbolt.java.views.html.pattern

@pattern(value = "(.)*\.printer", patternType = PatternType.REGEX) {
        Subject must have a permission that matches the regular expression (without quotes) "(.)*
\.printer" for this to be visible
}
```

A custom test is applied to determine access

```
@import be.objectify.deadbolt.java.views.html.pattern

@pattern(value = "something arbitrary", patternType = PatternType.CUSTOM) {
        DynamicResourceHandler#checkPermission must result in true for this to be visible
}
```

Providing fallback content

```
@import be.objectify.deadbolt.java.views.html.patternOr

@patternOr(value = "something arbitrary", patternType = PatternType.CUSTOM) {
    DynamicResourceHandler#checkPermission must result in true for this to be visible
} {
    Tough luck
}
```

Using a specific Deadbolt handler

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.pattern

@pattern(handler = handler, value = "(.)*\.printer", patternType = PatternType.REGEX) {
        Subject must have a permission that matches the regular expression (without quotes) "(.)*
\.printer" for this to be visible
}
```

Inverting the constraint

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.pattern

@pattern(value = "(.)*\.printer", patternType = PatternType.REGEX) {
    Subject must have no permissions that end with '.printer'
}
```

6.5 Dynamic

The most flexible constraint - this is a completely user-defined constraint that uses `DynamicResourceHandler#isAllowed` to determine access.

- be.objectify.deadbolt.java.views.html.dynamic
- be.objectify.deadbolt.java.views.html.dynamicOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.get()	The handler to use to apply the constraint.
name	String		The name of the constraint, passed into the `DynamicResourceHandler`.
meta	String	null	
timeout	() -> Long	A function returning `deadbolt.java.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Examples

The `DynamicResourceHandler` is obtained from the default handler and is used to apply a named constraint to the content

```
@import be.objectify.deadbolt.java.views.html.dynamic

@dynamic(name = "someName") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

Providing meta data to the constraint

```
@import be.objectify.deadbolt.java.views.html.dynamic

@dynamic(name = "someName", meta = "foo") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

Meta data does not have to be hard-coded

```
@(someMetaValue: String)
@import be.objectify.deadbolt.java.views.html.dynamic

@dynamic(name = "someName", meta = someMetaValue) {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

Providing fallback content

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.dynamicOr

@dynamicOr(handler = handler, name = "someName") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
} {
    Better luck next time
}
```

Using a specific Deadbolt handler

```
@(handler: be.objectify.deadbolt.java.DeadboltHandler)
@import be.objectify.deadbolt.java.views.html.dynamic

@dynamic(handler = handler, name = "someName") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

7. A deeper look at dynamic rules

The `@Dynamic` annotation and `@dynamic` template tag allow you to specify arbitrary rules to control access. For example:

- Simple (contrived) examples
  - Allow access if today is Tuesday
  - Allow access if a user has a date of birth defined
- Some more complex examples
  - Allow access if the user is a beta tester AND the controller/action/template area is available for beta testing
  - Allow a maximum of 300 requests per hour per API key
- Blacklisting
  - Deny access only if user registered within the last 30 days

7.1 Using sessions and requests in your rules

In order to make these decisions, information is helpful. You may need access to the session, or the current request. Both are available from the `Http.Context` object passed into your `DynamicResourceHandler` (DRH) implementations.

Sessions

In Play, sessions implement the `java.util.Map` interface and so data can get accessed in the usual way, i.e. session.get(key).

```
public class MyDynamicResourceHandler implements DynamicResourceHandler
{
    public F.Promise<Boolean> isAllowed(final String name,
                                        final String meta,
                                        final DeadboltHandler deadboltHandler,
                                        final Http.Context ctx)
    {
            return F.Promise.promise(() -> ctx.session())
                            .map(session -> // and so on
            ...
    }


    public boolean checkPermission(String permissionValue,
                                   DeadboltHandler deadboltHandler,
                                   Http.Context ctx)
    {
            return F.Promise.promise(() -> ctx.session())
                            .map(session -> // and so on
            ...
    }
}
```

You can also store write data into the session - just don't forget that sessions are stored as client-side cookies! Any and all confidential information should be kept out of sessions.

Request query parameters

Using the `Http.Request#queryString()` method, you can get a map of all query parameters for the current request. For example, for the URL

```
http://localhost:9000/users?userName=foo
```

a call to `queryString()` would result in a map containing the key `userName` mapped to the value `foo`.

```
public class MyDynamicResourceHandler implements DynamicResourceHandler
{
    public F.Promise<Boolean> isAllowed(final String name,
                                        final String meta,
                                        final DeadboltHandler deadboltHandler,
                                        final Http.Context ctx)
    {
            return F.Promise.promise(() -> ctx.request())
                            .map(request -> request.queryString())
                            .map((Map<String, String> queryStrings) -> // and so on
                ...
    }


    public F.Promise<Boolean> checkPermission(final String permissionValue,
                                              final DeadboltHandler deadboltHandler,
                                              final Http.Context ctx)
    {
            return F.Promise.promise(() -> ctx.request())
                            .map(request -> request.queryString())
                            .map((Map<String, String> queryStrings) -> // and so on
                ...
    }
}
```

A major problem here is that your DRH needs knowledge of the methods - or, more precisely, the parameters of the method - to which it is applied. This can be alleviated somewhat through the use of convention, or by passing metadata into the DRH. Both will be covered later in this chapter.

Request bodies

When using a HTTP POST or PUT operation, the request body will typically contain information. Bodies can be accessed in a variety of formats, including custom formats.

```
public class MyDynamicResourceHandler implements DynamicResourceHandler
{
    public F.Promise<Boolean> isAllowed(final String name,
                                        final String meta,
                                        final DeadboltHandler deadboltHandler,
                                        final Http.Context ctx)
    {
        return F.Promise.promise(() -> ctx.request().body())
                        .map(body -> body.asJson())
                        .map( // and so on
                    ...
    }


    public F.Promise<Boolean> checkPermission(final String permissionValue,
                                              final DeadboltHandler deadboltHandler,
                                              final Http.Context ctx)
    {
        return F.Promise.promise(() -> ctx.request().body())
                        .map(body -> body.as(MyCustomType.class))
                        .map(myCustomType -> // and so on
                    ...
    }
}
```

Once you have obtained the information from the body, you’re back in the decision-making process of your rule.

Request path parameters

Path parameters are, unfortunately, the point at which we hit a problem. Play encourages developers to create clean, RESTful URLs, for example

```
http://localhost:9000/users/foo
```

in place of the query parameter example used earlier,

```
http://localhost:9000/users?userName=foo
```

The benefits of clean URLs are many - they can be bookmarked, are easily shared and are more easily cached, to name just a few. The one thing that path parameters are not, in Play, is available at runtime. There is no way in Play to get these parameters without deriving them manually by comparing the route definition with the actual URL.

It has been stated[1] on the Play Framework group that path parameters should not be accessible by calling Request#queryStrings(), because path parameters are not query parameters. This is correct, but doesn't really help in the real world - things will be much easier when path parameters can be accessed as easily as other request information.

7.2 Strategies for using dynamic resource handlers

To the best of my knowledge, there are two ways in which to use dynamic resource handlers in Deadbolt:

1. Use a single `DynamicResourceHandler` that deals with all dynamic security
2. Use a single `DynamicResourceHandler` as a façade

1. Use a single, potentially huge DRH

A friend of mine, sitting in a second-year university course discussion on object-oriented design, witnessed a student put up his hand and say, "I don't get it. Why don't we just put everything in one big class?". If you would ask a similar question, this is the approach for you. For the rest of us, I think we can all appreciate that any DRH dealing with more than a couple of separate dynamic restrictions would get very large, very quickly.

2. Use a single DRH façade that dispatches to other DRHs

If you have a single DRH acting as a façade, you can aggregate or compose logic into discrete chunks. Furthermore, by extending `AbstractDynamicResourceHandler`, you can create specific handlers for individual methods in the façade, i.e. `isAllowed`-specific handlers and `checkPermission`-specific ones.

[1] https://groups.google.com/d/msg/play-framework/9qssE8s8aQA/pGXHrBf7gOYJ

```
public class FacadeDRH implements DynamicResourceHandler
{
    private final Map<String, DynamicResourceHandler> allowed = new HashMap<>();
    private final Map<String, DynamicResourceHandler> permitted = new HashMap<>();

    public FacadeDRH()
    {
        // populate the maps, either through new instance creation, constructor parameters\
, etc
    }

    public F.Promise<Boolean> isAllowed(final String name,
                                        final String meta,
                                        final DeadboltHandler handler,
                                        final Http.Context ctx)
    {
        return allowed.get(name).isAllowed(name,
                                           meta,
                                           handler,
                                           ctx);
    }


    public F.Promise<Boolean> checkPermission(final String permissionValue,
                                              final DeadboltHandler handler,
                                              final Http.Context ctx)
    {
        return permitted.get(permissionValue).checkPermission(permissionValue,
                                                              handler,
                                                              ctx);
    }
}
```

8. Integrating with authentication providers

Authorization is all well and good, but some constraints are pointless if the user is not known to the application. You could write a `Dynamic` constraint that only allows access on Thursdays - this doesn't need to know anything about even the concept of a user; a `Restrict` constraint, on the other hand, uses `Roles` obtained from a `Subject`. The question is, what is a subject and how do we know who it is?

Imagine an application where you can post short messages and read the messages of others, something along the lines of Twitter. By default, you can read any message on the system unless the user has marked that message as restricted to only users who are logged into the application.. In order to write a message, you have to have an account and be logged in. We can sketch out a controller for this very simple application thus:

A controller with Deadbolt constraints

```java
package be.objectify.messages;

import javax.inject.Inject;
import play.libs.F;
import play.libs.Json;
import play.mvc.Controller;
import play.mvc.Result;
import be.objectify.deadbolt.java.actions.SubjectPresent;

public class Messages extends Controller {

    private final MessageDao messageDao;

    @Inject
    public MessagesController(final MessageDao messageDao) {
        this.messageDao = messageDao;
    }

    public F.Promise<Result> getPublicMessages() {
        return F.Promise.promise(messageDao::getAllPublic)
                        .map(Json::toJson)
                        .map(Results::ok);
    }

    @SubjectPresent
    public F.Promise<Result> getAllMessages() {
        return F.Promise.promise(messageDao::getAll)
```

```
                        .map(Json::toJson)
                        .map(Results::ok);
    }

    @SubjectPresent
    public F.Promise<Result> createMessage() {
        return F.Promise.promise(() -> body.asJson())
                        .map(json -> Json.fromJson(json, Message.class))
                        .map(messageDao::save)
                        .map(Results::ok);
    }
}
```

This very simple app has a very simple `routes` file to go with it:

The application routes

```
GET     /messages/public    be.objectify.messages.Messages.getPublicMessages()
GET     /messages           be.objectify.messages.Messages.getAllMessages()
POST    /create             be.objectify.messages.Messages.createMessage()
```

An authenticated user can access all three of these routes and obtain a successful result. An unauthenticated user would hit a brick wall when accessing `/messages` or `/create` - specifically, they would run into whatever behaviour was specified in the `onAuthFailure` method of the current `DeadboltHandler`.

This is a good time to review the difference between the HTTP status codes `401 Unauthorized` and `403 Forbidden`. A 401 means you don't have access *at the moment*, but you should try again after authenticating. A 403 means the subject cannot access the resource with their current authorization rights, and re-authenticating will not solve the problem - in fact, the specification explicitly states that you shouldn't even attempt to re-authenticate. A well-behaved application should respect the difference between the two.

We can consider the `onAuthFailure` method to be the Deadbolt equivalent of a brick wall. For a `DeadboltHandler` used by a RESTful controller, the status code should be enough to indicate the problem. If you have an application that uses server-side rendering, you may well want to return content in the body of the response. The end result is the same though - You Can't Do That. Note the return type, a `Promise` containing a `Result` and not an `Optional<Result>` - access has very definitely failed at this point, and it needs to be dealt with.

Handling access failure

```java
public F.Promise<Result> onAuthFailure(final Http.Context context,
                                       final String content) {
    return F.Promise.promise(Results::forbidden);
}
```

Unlike `onAuthFailure`, the `beforeAuthCheck` method allows for the possibility that everything is fine. It's perfectly reasonable to return an empty option from this method, indicating that no action should be taken and the request should continue unimpeded.

Doing nothing before a constraint is applied

```java
public F.Promise<Optional<Result>> beforeAuthCheck(final Http.Context context) {
    return F.Promise.promise(Optional::empty);
}
```

This has the net effect of not getting in the way of the constraint; with this implementation, a user accessing `/messages` in our little application would receive a 403. But, our application aims to be well behaved and return a 401 when it's needed, so a little more work is required. Because `beforeAuthCheck` is only called when a constraint has been applied, we can use this to trigger an authenication request if needed. In this application, we're going to say that every constraint requires an authenticated subject to be present - do not confuse this with `@SubjectPresent` constraints in the example controller, the same would equally be true if we were using `@Restrict` or `@Pattern`. For the more advanced app that uses the subject-less dynamic rule of Thursdays-only, either more logic is required or (preferably) a different `DeadboltHandler` implementation is used.

The logic here is simple - if a user is present, no action is required otherwise short-cut the request with a 401 response.

Requiring a subject

```java
public F.Promise<Optional<Result>> beforeAuthCheck(final Http.Context context) {
    return getSubject(context).map(maybeSubject ->
        maybeSubject.map(subject -> Optional.<Result>empty())
                    .orElseGet(() -> Optional.of(Results.unauthorized())));
}
```

On the other hand, you may choose to never implement any logic in `beforeAuthCheck` and instead have the behaviour driven by authorization failure. The choice is entirely in the hands of the implementor; personally, I tend to use `onAuthFailure` to handle the 401/403 behaviour, because it removes the assumptions required by implementing checks in `beforeAuthCheck`.

It will become very clear, very quickly, the same approach is used for all authentication systems; this means that swapping out authentication without affecting authorization is both possible and trivial. We'll start with the built-in authentication mechanism of Play and adapt from there; with surprisingly few changes, you'll see how we can move from basic authentication through to using anything from Play-specific OAuth libraries such as Play Authenticate[1] and even HTTP calls to dedicated identity platforms such as Auth0[2].

8.1 Play's built-in authentication support

Play ships with a very simple interceptor that requires an authenticated user to be present; there's no concept of authorization. It uses an annotation-driven approach similar to that of Deadbolt, allowing you to annotate either entire controllers at the class level, or individual methods within a controller.

Using Play's authentication support

```java
@Security.Authenticated
public F.Promise<Result> getAllMessages() {
    return F.Promise.promise(messageDao::getAll)
                    .map(Json::toJson)
                    .map(Results::ok);
}
```

By default, the `Security.Authenticated` annotation will trigger an interceptor that uses `Security.Authenticator` to look in the session for a value mapped to `"username"` - if the value is non-null, the user is considered to be authenticated. If you want to customize how the user identification string is obtained, you can extend `Security.Authenticator` implement your own solution.

Customizing Play's authentication support

```java
package be.objectify.messages.security;

import java.util.Optional;
import javax.inject.Inject;
import be.objectify.messages.dao.UserDao;
import be.objectify.messages.models.User;
import play.mvc.Http;
import play.mvc.Security;

public class AuthenticationSupport extends Security.Authenticator {

    private final UserDao userDao;

    @Inject
```

[1]https://joscha.github.io/play-authenticate/
[2]https://auth0.com/

```
    public AuthenticationSupport(final UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public String getUsername(final Http.Context context) {
        return getTokenFromHeader(context).flatMap(userDao::findByToken)
                                          .map(User::getIdentifier)
                                          .orElse(null);
    }

    private Optional<String> getTokenFromHeader(final Http.Context context) {
        return Optional.ofNullable(context.request().headers().get("X-AUTH-TOKEN"))
                       .filter(arr -> arr.length == 1)
                       .filter(arr -> arr[0] != null)
                       .map(arr -> arr[0]);
    }
}
```

The class of this customized implementation can then be passed to the annotation with `@Security.Authenticated(AuthenticationSupport.class)`. However, all mention of Deadbolt's constraints have vanished and so we've replaced a fine-grained authorization system with a coarse-grained authentication-only system. To fix this, we need to revert back to using Deadbolt in the controller and move `AuthenticationSupport` (or even the basic `Security.Authenticator`) integration into the `DeadboltHandler`.

Integrating Play's authentication with Deadbolt

```
package be.objectify.messages.security;

import java.util.Optional;
import javax.inject.Inject;
import be.objectify.deadbolt.core.models.Subject;
import be.objectify.deadbolt.java.AbstractDeadboltHandler;
import be.objectify.messages.dao.UserDao;
import play.libs.F;
import play.mvc.Http;
import play.mvc.Result;
import play.mvc.Results;

public class MyDeadboltHandler extends AbstractDeadboltHandler {

    private final AuthenticationSupport authenticator;
    private final UserDao userDao;

    @Inject
    public MyDeadboltHandler(final AuthenticationSupport authenticator,
                             final UserDao userDao) {
        this.authenticator = authenticator;
        this.userDao = userDao;
    }

    @Override
    public F.Promise<Optional<Result>> beforeAuthCheck(final Http.Context context) {
        return getSubject(context).map(maybeSubject ->
            maybeSubject.map(subject -> Optional.<Result>empty())
                        .orElseGet(() -> Optional.of(Results.unauthorized())));
    }

    @Override
    public F.Promise<Optional<Subject>> getSubject(final Http.Context context) {
        return F.Promise.promise(() ->
            Optional.ofNullable(authenticator.getUsername(context))
                    .flatMap(userDao::findByUserName)
                    .map(user -> user));
    }

    @Override
    public F.Promise<Result> onAuthFailure(final Http.Context context,
                                           final String content) {
        // you could also use the behaviour of the authenticator, e.g.
        // return F.Promise.promise(() -> authenticator.onUnauthorized(context));
        return F.Promise.promise(() -> forbidden("You can't do that"));
    }
}
```

There are three things going on here, only one of which is explicitly tied into the authentication system, and that is `getSubject`. Of the other two methods, `onAuthFailure` gives an arbitrary (but hopefully meaningful) response and `beforeAuthCheck` is essentially generic code. There is scope here for performance improvements and will depend on your specific implementations; for example, the user retrieved from the database by the authenticator can be stored in `context.args` for re-use by the Deadbolt handler.

Per-request caching of the user

```java
public class AuthenticationSupport extends Security.Authenticator {

    // other methods

    @Override
    public String getUsername(final Http.Context context) {
        final Optional<User> maybeUser = getTokenFromHeader(context).flatMap(userDao::find\
ByToken);
        return maybeUser.map(user -> {
            context.args.put("user", maybeUser);
            return user.getIdentifier();
        }).orElse(null);
    }
}

public class MyDeadboltHandler extends AbstractDeadboltHandler {

    // other methods

    @Override
    public F.Promise<Optional<Subject>> getSubject(final Http.Context context) {
        return F.Promise.promise(() -> (Optional<Subject>)context.args.computeIfAbsent(
                "user",
                key -> {
                    final String userName = authenticator.getUsername(context);
                    return userDao.findByUserName(userName);
                }));
    }
}
```

8.2 Third-party user management

I like the concept of third-party user (or identity) management. It minimizes sensitive data held locally, provides features like multifactor authentication and provides a unified API for multiple authentication sources. This last feature makes it very easy to create a simple integration point with Deadbolt, driving interaction with the user management on a per-event basis (with a little caching thrown in). The sequence for this looks a little complicated, but it can be broken down into 4 distinct steps:

- the initial authentication
- subsequent use of cached user details
- re-retrieving user details when the cache doen't contain them

- re-authenticating when the user's authentication has expired on the user management platform

If you're using Deadbolt, it's reasonable to assume you have one of two security models - either all actions require authorization or some actions require authorization, and that authorization may simply be "a user must be present". As a result, the point the initial authentication occurs depends on your application. The good news, as far as this requirement goes, is the implementation is both quite simple and common to all cases. It comes down to the implementation of the `onAuthFailure` method of your `DeadboltHandler`, and might look something like this:

Triggering authentication when authorization fails - naive version

```
public F.Promise<Result> onAuthFailure(final Http.Context context,
                                       final String contentType) {
    return F.Promise.promise(login::render)
                    .map(Results::unauthorized);
}
```

But wait, this is wrong! As discussed above, this assumes that all authorization failure occurs because there is no user present, and this ignores the difference between `401 Unauthorized` and `403 Forbidden`. A better implementation wiil take this into account, by checking if there is a subject present. If there is a subject present, it's a 403; if there isn't, it's a 401 and we can redirect to somewhere the user can log in.

Triggering authentication when authorization fails

```
public F.Promise<Result> onAuthFailure(final Http.Context context,
                                       final String contentType) {
    return getSubject(context).map(maybeSubject ->
        maybeSubject.map(subject ->
            Optional.of((User)subject))
                    .map(denied::render)
                    .orElseGet(() -> login.render()))
                    .map(Results::unauthorized);
}
```

There's still a problem here - while the rendered output observes the difference between unauthorized and forbidden, but the HTTP status code is hard-wired to a 401. One more tweak should fix this.

Synchronizing the content and HTTP status code

```java
@Override
public F.Promise<Result> onAuthFailure(final Http.Context context,
                                       final String s) {
    return getSubject(context)
            .map(maybeSubject ->
                         maybeSubject.map(subject ->
                                                  Optional.of((User)subject))
                                     .map(user ->
                                                  new F.Tuple<>(true,
                                                                denied.render(user)))
                                     .orElseGet(() ->
                                                  new F.Tuple<>(false,
                                                                login.render(clientId,
                                                                             domain,
                                                                             redirectUri))))
            .map(subjectPresentAndContent ->
                     subjectPresentAndContent._1
                         ? Results.forbidden(subjectPresentAndContent._2)
                         : Results.unauthorized(subjectPresentAndContent._2));
}
```

Integrating with Auth0

Auth0[3] is a great identity management platform, and I'm not writing that just because they gave me a t-shirt. One of the nice features they offer is a whole bunch of code you can pretty much drop into your application, including code for a Play 2 Scala controller. Since we're in the Java portion of the book, and the Java example provided by Auth0 uses the JEE Servlet API, I've rewritten the Scala version for Java. This was the only customization required, which I have to say was pretty impressive - the total time to integrate and have a working solution was less than 15 minutes.

There are three core elements to the solution. These are, in no particular order,

- a log-in page
- a controller to receive callbacks from Auth0
- a DeadboltHandler implementation

I've also added a small utility class called `AuthSupport` to help with the cache usage, which also makes testing easier, but this contains code that could happily live in the controller.

A working example for this section can be found at auth0-integration4. To run it, you will need to create an application on Auth0 and fill in the client ID, client secret, etc, into `conf/application.conf`. For the redirect URI, you can use `http://localhost:9000/callback` - don't forget to adjust the port if necessary.

[3]https://auth0.com/
4https://github.com/schaloner/deadbolt-2-guide-examples/tree/master/auth0-integration

AuthSupport

This class has two simple function - it standardises the key used for caching the subject, and it wraps the cache result in an `Optional`.

Integrating the authentication flow

```java
package be.objectify.whale.security;

import java.util.Optional;
import javax.inject.Inject;
import javax.inject.Singleton;
import be.objectify.whale.models.User;
import play.cache.CacheApi;
import play.mvc.Http;

@Singleton
public class AuthSupport {

    private final CacheApi cache;

    @Inject
    public AuthSupport(final CacheApi cache) {
        this.cache = cache;
    }

    public Optional<User> currentUser(final Http.Context context) {
        return Optional.ofNullable(cache.get(cacheKey(context.session().get("idToken"))));
    }

    public String cacheKey(final String key) {
        return "user.cache." + key;
    }
}
```

The log-in page

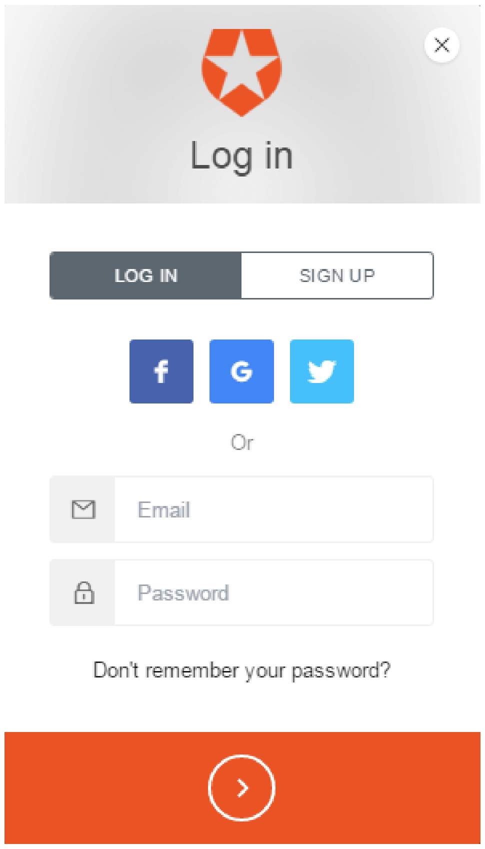
In order to log in, Auth0 provide a JavaScript solution that customises the form based on your configuration options; for example, an app registered in Auth0 for username/password support plus a couple of OAuth providers will receive a form that reflects those choices. The simplest possible implementation of a log-in page, without concern for appearance, is as follows.

An Auth0 log-in form

```
@(clientId: String, domain: String, redirectUri: String)

<!DOCTYPE html>
<html lang="en">
    <body>
        <div id="root">
            Log-in area
        </div>
        <script src="https://cdn.auth0.com/js/lock-7.12.min.js"></script>
        <script>
            var lock = new Auth0Lock('@clientId', '@domain');
            lock.show({
                container: 'root',
                callbackURL: '@redirectUri',
                responseType: 'code',
                authParams: { scope: 'openid profile' }
            });
        </script>
    </body>
</html>
```

In the browser, you now have a completely functional log-in form that will trigger the authentication flow. Once the form is submitted, Auth0 takes over until the authentication requirements are satisfied and then we receive a callback.



An Auth0 log-in form

The controller

The bulk of the logic is contained here, and this code is reasonably generic - barring the `User` and `UserDao` classes, this code can be used in any Play 2 application. In broad terms, three things happen during a successful authentication flow - all of these are rooted in the `callback` method of the controller.

1. The controller receives a callback, in the form of a HTTP request from Auth0 containing an authorization code.
2. The controller makes a HTTP request to Auth0 to get the token.
3. The controller makes a HTTP request to Auth0 to get the user details.

Processing the callback from Auth0

```
public F.Promise<Result> callback(final F.Option<String> maybeCode,
                                  final F.Option<String> maybeState) {
    return maybeCode.map(code -> getToken(code) // get the authentication token
             .flatMap(token -> getUser(token))  // get the user details
             .map(userAndToken -> {
                 // userAndToken._1 is the user
                 // userAndToken._2 is the token
                 cache.set(authSupport.cacheKey(userAndToken._2._1),
                           userAndToken._1,
                           60 * 15); // cache the subject for 15 minutes
                 session("idToken",
                         userAndToken._2._1);
                 session("accessToken",
                         userAndToken._2._2);
                 return redirect(routes.Application.index());
             }))
            .getOrElse(F.Promise.pure(badRequest("No parameters supplied")));
}
```

This callback provides the starting point for further interaction with Auth0 by giving us an authorization code. With this code, we can request token information; `access_token` allows us to work with the subject's attributes, and ´id_token´ is a signed Json Web Token used to authenticate API calls.

Get the token information

```
private F.Promise<F.Tuple<String, String>> getToken(final String code) {
    final ObjectNode root = Json.newObject();
    root.put("client_id",
             this.clientId);
    root.put("client_secret",
             this.clientSecret);
    root.put("redirect_uri",
             this.redirectUri);
    root.put("code",
             code);
    root.put("grant_type",
             "authorization_code");
    return WS.url(String.format("https://%s/oauth/token",
                                this.domain))
             .setHeader(Http.HeaderNames.ACCEPT,
                        Http.MimeTypes.JSON)
             .post(root)
             .map(WSResponse::asJson)
```

```
                .map(json -> new F.Tuple<>(json.get("id_token").asText(),
                                           json.get("access_token").asText()));
}
```

With these token data, we can retrieve the subject attributes. At this point, it’s possible to cache the subject to reduce network calls. In this example, we have no concept of a database and so we rely entirely on Auth0 to provide subject information. If you keep some user information local, this might be a good place to either create or retrieve that information.

Get the subject attributes

```
private F.Promise<F.Tuple<User, F.Tuple<String, String>>>
                        getUser(final F.Tuple<String, String> token) {
    return WS.url(String.format("https://%s/userinfo",
                                this.domain))
             .setQueryParameter("access_token",
                                token._2)
             .get()
             .map(WSResponse::asJson)
             .map(json -> new User(json.get("user_id").asText(),
                                   json.get("name").asText(),
                                   json.get("picture").asText()))
             .map(localUser -> new F.Tuple<>(localUser,
                                             token));
}
```

Logging out simply requires the token information to be removed from the session, and the removal of the subject from the cache.

Log the subject out

```
public F.Promise<Result> logOut() {
    return F.Promise.promise(() -> {
        final Http.Session session = session();
        final String idToken = session.remove("idToken");
        session.remove("accessToken");
        cache.remove(authSupport.cacheKey(idToken));
        return "ignoreThisValue";
    }).map(id -> redirect(routes.AuthController.logIn()));
}
```

This is a lot of code, but authentication is now handled. We now have a way to log in, and a way to retrieve the user details from Auth0. This controller needs to be exposed in the routes file, and this also provides a nice overview to see what we’ve achieved.

Authentication controller routes

```
GET   /logIn     be.objectify.whale.controllers.AuthController.logIn()
GET   /callback  be.objectify.whale.controllers.AuthController.callback(code: play.libs.F.O\
ption[String], state: play.libs.F.Option[String])
GET   /logOut    be.objectify.whale.controllers.AuthController.logOut()
GET   /denied    be.objectify.whale.controllers.AuthController.denied()
```

Now we have a `/logIn` route, that means you can have an explicit link to log in from your application. The one thing remaining to do is to have the log-in view displayed automatically when authorization fails.

The DeadboltHandler

There are only two methods that are required for this example to work. `getSubject` will retrieve the subject from the cache, and `onAuthFailure` will handle things as discussed above.

Authentication controller routes

```java
@Override
public F.Promise<Optional<Subject>> getSubject(final Http.Context context) {
    return F.Promise.promise(() -> Optional.ofNullable(cache.get(authSupport.cacheKey(cont\
ext.session().get("idToken")))));
}

@Override
public F.Promise<Result> onAuthFailure(final Http.Context context,
                                       final String s) {
    return getSubject(context)
            .map(maybeSubject ->
                         maybeSubject.map(subject -> Optional.of((User)subject))
                                     .map(user ->
                                              new F.Tuple<>(true,
                                              denied.render(user)))
                                     .orElseGet(() ->
                                              new F.Tuple<>(false,
                                                            login.render(clientId,
                                                                         domain,
                                                                         redirectUri))))
            .map(subjectPresentAndContent ->
                     subjectPresentAndContent._1
                         ? Results.forbidden(subjectPresentAndContent._2)
                         : Results.unauthorized(subjectPresentAndContent._2));
}
```

Improvements

This is a very simple example, but it demonstrates how easily it is to use event-driven behaviour and third-party identity management. There is one major problem, however, and you have until the end of this sentence to figure out what it is.

When the subject attributes are retrieved from Auth0, the resulting `User` object is cached for an arbitrary time - 15 minutes, in this case. With the implementation of `DeadboltHandler` given above, once that 15 minutes have passed the user will need to re-authenticate. However, it's possible their authenticate period on Auth0 is still valid and so we're placing an unnecessary burden on the end user. A simple improvement would be to attempt retrieval of the user attributes from `DeadboltHandler#getSubject` when the cache doesn't contain the user.

It's also possible to store meta data in Auth0, and so you can represent your roles and permissions there and bind them into local models when retrieving the subject's attributes.

II Deadbolt for Scala

9. Using Deadbolt 2 with Play 2 Scala projects

Deadbolt for Scala provides an idiomatic API for dealing with Scala controllers and templates rendered from Scala controllers in Play applications.

9.1 The Deadbolt Handler

For any module - or framework - to be useable, it must provide a mechanism by which it can be hooked into your application. For Deadbolt, the central hook is the `be.objectify.deadbolt.scala.DeadboltHand` trait. The four functions defined by this trait are crucial to Deadbolt - for example, `DeadboltHandler#getSubject` gets the current user (or subject, to use the correct security terminology), whereas `DeadboltHandler#onAccessFailure` is used to generate a response when authorization fails.

DeadboltHandler implementations should be stateless.

For each method, a `Future` is returned. If the future may complete to have an empty value, e.g. calling `getSubject` when no subject is present, the return type is `Future[Option]`.

Despite the use of the definite article in the section title, you can have as many Deadbolt handlers in your app as you wish.

Performing pre-constraint tests

Before a constraint is applied, the `Future[Option[Result]] beforeAuthCheck[A](request: Request[A])` function of the current handler is invoked. If the resulting future completes `Some`, the target action is not invoked and instead the result of `beforeAuthCheck` is used for the HTTP response; if the resulting future completes to `None` the action is invoked with the Deadbolt constraint applied to it.

Obtaining the subject

To get the current subject, the `Future[Option[Subject]] getSubject[A](request: Request[A])` function is invoked. Returning a `None` indicates there is no subject present - this is a valid scenario.

Dealing with authorization failure

When authorization fails, the `Future[Result] onAccessFailure[A](request: Request[A])` function is used to obtain a result for the HTTP response. The result required from the `Future` returned from this method is a regular `play.api.mvc.Result`, so it can be anything you chose. You might want to return a 403 forbidden, redirect to a location accessible to everyone, etc.

Dealing with dynamic constraints

Dynamic constraints, which are `Dynamic` and `Pattern.CUSTOM` constraints, are dealt with by implementations of `DynamicResourceHandler`; this will be explored in a later chapter. For now, it's enough to say `Future[Optional[DynamicResourceHandler]] getDynamicResourceHandler[A](request: Request[A])` is invoked when a dynamic constraint it used.

9.2 Expose your DeadboltHandlers with a HandlerCache

Deadbolt uses dependency injection to expose handlers in a type-safe manner. Various components of Deadbolt, which will be explored in later chapters, require an instance of `be.objectify.deadbolt.scala.` - however, no such implementations are provided.

Instead, you need to implement your own version. This trait extends `Function[HandlerKey, DeadboltHandler]` and `Function0[DeadboltHandler]` and uses them as follows

- `handlers()` will express the default handler
- handlers(key: HandlerKey) will express a named handler

Here's one possible implementation, using hard-coded handlers.

An example handler cache

```scala
@Singleton
class MyHandlerCache extends HandlerCache {
    val defaultHandler: DeadboltHandler = new MyDeadboltHandler

    // HandlerKeys is an user-defined object, containing instances
    // of a case class that extends HandlerKey
    val handlers: Map[Any, DeadboltHandler] =
        Map(HandlerKeys.defaultHandler -> defaultHandler,
            HandlerKeys.altHandler ->
                  new MyDeadboltHandler(Some(MyAlternativeDynamicResourceHandler)),
            HandlerKeys.userlessHandler -> new MyUserlessDeadboltHandler)

    // Get the default handler.
    override def apply(): DeadboltHandler = defaultHandler

    // Get a named handler
    override def apply(handlerKey: HandlerKey): DeadboltHandler = handlers(handlerKey)
}
```

Finally, create a small module which binds your implementation.

Binding the handler cache

```
package com.example.modules

import be.objectify.deadbolt.scala.cache.HandlerCache
import play.api.inject.{Binding, Module}
import play.api.{Configuration, Environment}
import com.example.security.MyHandlerCache

class CustomDeadboltHook extends Module {
    override def bindings(environment: Environment,
                          configuration: Configuration): Seq[Binding[_]] = Seq(
        bind[HandlerCache].to[MyHandlerCache]
    )
}
```

9.3 application.conf

Declare the necessary modules

Both `be.objectify.deadbolt.scala.DeadboltModule` and your custom bindings module must be declared in the configuration.

Enable your module

```
play {
  modules {
    enabled += be.objectify.deadbolt.scala.DeadboltModule
    enabled += com.example.modules.CustomDeadboltHook
  }
}
```

9.4 Using compile-time dependency injection

If you prefer to wire everything together with compile-time dependency, you don't need to create a custom module or add `DeadboltModule` to `play.modules`.

Instead, dependencies are handled using a custom `ApplicationLoader`. To make things easier, various Deadbolt components are made available via the `be.objectify.deadbolt.scala.DeadboltComponents` trait. You will still need to provide a couple of things, such as your `HandlerCache` implementation, and you'll then have access to all the usual pieces of Deadbolt.

An example ApplicationLoader for compile-time DI

```
class CompileTimeDiApplicationLoader extends ApplicationLoader  {
  override def load(context: Context): Application
             = new ApplicationComponents(context).application
}

class ApplicationComponents(context: Context)
                            extends BuiltInComponentsFromContext(context)
                            with DeadboltComponents
                            with EhCacheComponents {

  // Define a pattern cache implementation
  // defaultCacheApi is a component from EhCacheComponents
  override lazy val patternCache: PatternCache = new DefaultPatternCache(defaultCacheApi)

  // Declare something required by MyHandlerCache
  lazy val subjectDao: SubjectDao = new TestSubjectDao

  // Specify the DeadboltHandler implementation to use
  override lazy val handlers: HandlerCache = new MyHandlerCache(subjectDao)

  // everything from here down is application-level
  // configuration, unrelated to Deadbolt, such as controllers, routers, etc
  // ...
}
```

The components provided by Deadbolt are

- `scalaAnalyzer` - constraint logic
- `deadboltActions` - for composing actions
- `actionBuilders` - for building actions
- `viewSupport` - for template constraints
- `patternCache` - for caching regular expressions. You need to define this yourself in the application loader, but as in the example above it's easy to use the default implementation
- `handlers` - the implementation of `HandlerCache` that you provide
- `configuration` - the application configuration
- `ecContextProvider` - the execution context for concurrent operations. Defaults to `scala.concurrent.Execu`
- `templateFailureListenerProvider` - for listening to Deadbolt-related errors that occur when rendering templates. Defaults to a no-operation implementation

Once you've defined your `ApplicationLoader`, you need to add it to your `application.conf`.

Specify the application loader to use

```
play {
  application {
    loader=com.example.myapp.CompileTimeDiApplicationLoader
  }
}
```

9.5 Tweaking Deadbolt

Deadbolt Scala-specific configuration lives in the `deadbolt.scala` namespace.

There is one setting, `deadbolt.scala.view-timeout`, which is millisecond timeout applied to blocking calls when rendering templates. This defaults to 1000ms.

Personally, I prefer the HOCON (Human-Optimized Config Object Notation) syntax supported by Play, so I would recommend the following:

Example configuration

```
deadbolt {
  scala {
    view-timeout=500
  }
}
```

Execution context

By default, all futures are executed in the scala.concurrent.ExecutionContext.global context. If you want to provide a separate execution context, you can plug it into Deadbolt by implementing the DeadboltExecutionContextProvider trait.

Providing a custom execution context

```
import be.objectify.deadbolt.scala.DeadboltExecutionContextProvider

class CustomDeadboltExecutionContextProvider extends DeadboltExecutionContextProvider {
    override def get(): ExecutionContext = ???
}
```

NB: This provider is invoked twice, once in `DeadboltActions` and once in `ViewSupport`. Make sure you take this into account when you implement the `get()` function.

Once you've implemented the provider, you need to declare it in your custom module (see `CustomDeadboltHook` above for further information).

Binding the custom execution context provider

```
class CustomDeadboltHook extends Module {
    override def bindings(environment: Environment,
                          configuration: Configuration): Seq[Binding[_]] = Seq(
        bind[HandlerCache].to[MyHandlerCache],
        bind[DeadboltExecutionContextProvider].to[CustomDeadboltExecutionContextProvider]
    )
}
```

10. Scala controller constraints

Controller-level constraints can be added in two different ways - through the use of action builders and through action composition. The resulting behaviour is identical, so choose whichever suites your style best.

10.1 Controller constraints with the action builder

To get started, inject `ActionBuilders` into your controller.

```
class ExampleController @Inject() (actionBuilder: ActionBuilders) extends Controller
```

You now have builders for all the constraint types, which we'll take a quick look at in a minute. In the following examples I'm using the default handler, i.e. `.defaultHandler()` but it's also possible to use a different handler with `.key(HandlerKey)` or pass in a handler directly using `.withHandler(DeadboltHandler)`.

10.2 Controller constraints with action composition

Using the `DeadboltActions` class, you can compose constrained functions. To get started, inject `DeadboltActions` into your controller.

```
class ExampleController @Inject() (deadbolt: DeadboltActions) extends Controller
```

You now have functions equivalent to those of the builders mentioned above. In the following examples I'm using the default handler, i.e. no handler is specified, but it's also possible to use a different handler with `handler = <some handler, possibly from the handler cache>`.

10.3 SubjectPresent

Sometimes, you don't need fine-grained checks - you just need to see if there **is** a user present.

Action builder

DeadboltHandler#getSubject must result in a Some for access to be granted

```
def someFunctionA = actionBuilder.SubjectPresentAction().defaultHandler() { Ok(accessOk())\
 }
```

Action composition

Parameter	Type	Default	Notes
handler	DeadboltHandler	HandlerCache.apply()	The DeadboltHandler instance to use.

DeadboltHandler#getSubject must result in a Some for access to be granted

```
def someFunctionA = deadbolt.SubjectPresent() {
  Action {
    Ok(accessOk())
  }
}
```

10.4 SubjectNotPresent

Sometimes, you don't need fine-grained checks - you just need to see if there **is no** user present.

Action builder

DeadboltHandler#getSubject must result in a None for access to be granted

```
def someFunctionB = actionBuilder.SubjectNotPresentAction().defaultHandler() { Ok(accessOk\
()) }
```

Action composition

Parameter	Type	Default	Notes
handler	DeadboltHandler	HandlerCache.apply()	The DeadboltHandler instance to use.

DeadboltHandler#getSubject must result in a None for access to be granted

```
def someFunctionB = deadbolt.SubjectNotPresent() {
  Action {
    Ok(accessOk())
  }
}
```

10.5 Restrict

Restrict uses a `Subject`’s `Roles` to perform AND/OR/NOT checks. The values given to the builder must match the `Role.name` of the subject’s roles.

AND is defined as an `Array[String]` (or more correctly, `String*`), OR is a `List[Array[String]]`, and NOT is a rolename with a `!` preceding it.

Action builder

Parameter	Type	Default	Notes
roles	List[Array[String]]		Allows the definition of OR’d constraints by having multiple arrays in the list.
roles	String*		A short-hand of defining a single role or AND constraint.

The subject must have the foo role

```
def restrictedFunctionA = actionBuilder.RestrictAction("foo")
                                      .defaultHandler() { Ok(accessOk()) }
```

The subject must have the foo AND bar roles

```
def restrictedFunctionB = actionBuilder.RestrictAction("foo", "bar")
                                      .defaultHandler() { Ok(accessOk()) }
```

The subject must have the foo OR bar roles

```
def restrictedFunctionC = actionBuilder.RestrictAction(List(Array("foo"), Array("bar")))
                                      .defaultHandler() { Ok(accessOk()) }
```

Action composition

Parameter	Type	Default	Notes
roleGroups	List[Array[String]]		Allows the definition of OR’d constraints by having multiple arrays in the list.
handler	DeadboltHandler	HandlerCache.apply()	The DeadboltHandler instance to use.

The subject must have the foo role

```scala
def restrictedFunctionA = deadbolt.Restrict(List(Array("foo")) {
  Action {
    Ok(accessOk())
  }
}
```

The subject must have the foo AND bar roles

```scala
def restrictedFunctionB = deadbolt.Restrict(List(Array("foo", "bar")) {
  Action {
    Ok(accessOk())
  }
}
```

The subject must have the foo OR bar roles

```scala
def restrictedFunctionB = deadbolt.Restrict(List(Array("foo"), Array("bar")) {
  Action {
    Ok(accessOk())
  }
}
```

10.6 Pattern

Pattern uses a `Subject`'s `Permissions` to perform a variety of checks. The check depends on the pattern type.

- EQUALITY - the subject must have a permission whose value is exactly the same as the `value` parameter
- REGEX - the subject must have a permission which matches the regular expression given in the `value` parameter
- CUSTOM - the `DynamicResourceHandler#checkPermission` function is used to determine access

It's possible to invert the constraint by setting the `invert` parameter to true. This changes the meaning of the constraint in the following way.

- EQUALITY - the subject must NOT have a permission whose value is exactly the same as the `value` parameter
- REGEX - the subject must have NO permissions that match the regular expression given in the `value` parameter
- CUSTOM - the `DynamicResourceHandler#checkPermission` function, where the OPPOSITE of the Boolean resolved from the function is used to determine access

Action builder

Parameter	Type	Default	Notes
value	String		The value of the permission.
patternType	PatternType	PatternType.EQUALITY	One of EQUALITY, REGEX or CUSTOM.
invert	Boolean	false	Invert the result of the test

subject must have a permission with the exact value 'admin.printer'

```
def permittedFunctionA = actionBuilders.PatternAction(value = "admin.printer",
                                                      patternType = PatternType.EQUALITY)
                                             .defaultHandler() { Ok(accessOk()) }
```

subject must have a permission that matches the regular expression (without quotes) '(.).printer'*

```
def permittedFunctionB = actionBuilders.PatternAction(value = "(.)*\.printer",
                                                       patternType = PatternType.REGEX)
                                      .defaultHandler() { Ok(accessOk()) }
```

checkPermission is used to determine access

```
// the checkPermssion function of the current handler's DynamicResourceHandler
// will be used.  This is a user-defined test
def permittedFunctionC = actionBuilders.PatternAction(value = "something arbitrary",
                                                       patternType = PatternType.CUSTOM)
                                       .defaultHandler() { Ok(accessOk()) }
```

subject must have no permissions that end in .printer

```
def permittedFunctionB = actionBuilders.PatternAction(value = "(.)*\.printer",
                                                       patternType = PatternType.REGEX,
                                                       invert = true)
                                      .defaultHandler() { Ok(accessOk()) }
```

Action composition

Parameter	Type	Default	Notes
value	String		The value of the permission.
patternType	PatternType	PatternType.EQUALITY	One of EQUALITY, REGEX or CUSTOM.
handler	DeadboltHandler	HandlerCache.apply()	The DeadboltHandler instance to use.
invert	Boolean	false	Invert the result of the test

subject must have a permission with the exact value 'admin.printer'

```
def permittedFunctionA = deadbolt.Pattern(value = "admin.printer",
                                          patternType = PatternType.EQUALITY) {
  Action {
    Ok(accessOk())
  }
}
```

subject must have a permission that matches the regular expression (without quotes) '(.).printer'*

```scala
def permittedFunctionB = deadbolt.Pattern(value = "(.)*\.printer",
                                          patternType = PatternType.REGEX) {
  Action {
    Ok(accessOk())
  }
}
```

checkPermission is used to determine access

```scala
// the checkPermssion function of the current handler's DynamicResourceHandler
// will be used.
def permittedFunctionC = deadbolt.Pattern(value = "something arbitrary",
                                          patternType = PatternType.CUSTOM) {
  Action {
    Ok(accessOk())
  }
}
```

subject must have no permissions that end in .printer

```scala
def permittedFunctionB = deadbolt.Pattern(value = "(.)*\.printer",
                                          patternType = PatternType.REGEX,
                                          invert = true) {
  Action {
    Ok(accessOk())
  }
}
```

10.7 Dynamic

The most flexible constraint - this is a completely user-defined constraint that uses `DynamicResourceHandler#isAllowed` to determine access.

Action builder

Parameter	Type	Default	Notes
name	String		The name of the constraint.
meta	String		Additional information for the constraint implementation to use.

use the constraint associated with the name 'someClassifier' to control access

```scala
def foo = actionBuilder.DynamicAction(name = "someClassifier")
                       .defaultHandler() { Ok(accessOk()) }
```

Action composition

Parameter	Type	Default	Notes
name	String		The name of the constraint.
meta	String		Additional information for the constraint implementation to use.
handler	DeadboltHandler	HandlerCache.apply()	The DeadboltHandler instance to use.

use the constraint associated with the name 'someClassifier' to control access

```scala
def foo = deadbolt.Dynamic(name = "someClassifier") {
  Action {
    Ok(accessOk())
  }
}
```

11. Deadbolt Scala Templates

This is not a client-side DOM manipulation, but rather the exclusion of content when templates are rendered. This also means that any logic inside the constrained content will not execute if authorization fails.

A subject is required for the content to be rendered

```
@subjectPresent {
  <!-- paragraphs will not be rendered, satellites will not be repositioned -->
  <!-- and light speed will not be engaged if no subject is present -->
  <p>Let's see what this thing can do...</p>
  @repositionSatellite
  @engageLightSpeed
}
```

One important thing to note here is that templates are blocking, so any Futures used need to be completed for the resuly to be used in the template constraints. As a result, each constraint can take a function that expresses a Long, which is the millisecond value of the timeout. It defaults to 1000 milliseconds, but you can change this globally by setting the `deadbolt.scala.view-timeout` value in your `application.conf`.

11.1 Handlers

By default, template constraints use the default Deadbolt handler but as with controller constraints you can pass in a specific handler. The cleanest way to do this is to pass the handler into the template and then pass it into the constraints. Another advantage of this approach is you can pass in a wrapped version of the handler that will cache the subject; if you have a lot of constraints in a template, this can yield a significant gain.

Fallback content

Each constraint has an `xOr` variant, which allows you to render content in place of the unauthorized content. This takes the form `<constraint>Or`, for example `subjectPresentOr`

Providing fallback content when a subject is not present

```
@subjectPresentOr {
    <!-- paragraphs will not be rendered, satellites will not be repositioned -->
    <!-- and light speed will not be engaged if no subject is present -->
    <p>Let's see what this thing can do...</p>
    @repositionSatellite
    @engageLightSpeed
} {
    <marquee>Sorry, you are not authorized to perform clichéd actions from B movies.  Suff\
er the marquee!</marquee>
}
```

In each case, the fallback content is defined as a second `Content` block following the primary body.

Timeouts

Because templates use blocking calls when rendering, the futures returned from the Deadbolt handler, etc, need to be completed during the rendering process. A timeout, with a default value of 1000ms, is used to wait for the completion but you may want to change this. You can do this in two ways.

Set a global timeout

If you want to change the default timeout, define `deadbolt.scala.view-timeout` in your configuration and give it a millisecond value, e.g.

Define timeouts in milliseconds

```
deadbolt {
  scala {
    view-timeout=1500
  }
}
```

Use a supplier to provide a timeout

All Deadbolt templates have a `timeout` parameter which defaults to expressing the app-wide value - 1000L if nothing else if defined, otherwise whatever `deadbolt.java.view-timeout` is set to. But - and here's the nice part - the `timeout` parameter is not a `Long` but rather a `Function0<Long>`. This means you can use a timeout that fluctuates based on some metric - say, the number of timeouts that occur during template rendering.

How do I know if timeouts are occurring?

That's a good question. And the answer is - you need to implement `be.objectify.deadbolt.scala.TemplateFailu` and bind it using a module; see "Expose your DeadboltHandlers with a HandlerCache" section in chapter 8 for more details on this. If you re-use that chapter 8 module, the binding will look something like this.

Declaring a template failure listener

```scala
class CustomDeadboltHook extends Module {
    override def bindings(environment: Environment,
                          configuration: Configuration): Seq[Binding[_]] = Seq(
        bind[HandlerCache].to[MyHandlerCache],
        bind[TemplateFailureListener].to[MyTemplateFailureListener]
    )
}
```

Making it a singleton allows you to keep a running count of the failure level; if you're using it for other purposes, then scope it accordingly.

11.2 SubjectPresent

Sometimes, you don't need fine-grained checked - you just need to see if there **is a** user present.

- be.objectify.deadbolt.scala.views.html.subjectPresent
- be.objectify.deadbolt.scala.views.html.subjectPresentOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.apply()	The handler to use to apply the constraint.
timeout	() ⇒ Long	A function returning `deadbolt.scala.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Example 1

The default Deadbolt handler is used to obtain the subject.

```
@subjectPresent() {
    This content will be present if handler#getSubject results in a Some
}

@subjectPresentOr() {
    This content will be present if handler#getSubject results in a Some
} {
        fallback content
}
```

Example 2

A specific Deadbolt handler is used to obtain the subject.

```
@(handler: DeadboltHandler)
@subjectPresent(handler = handler) {
    This content will be present if handler#getSubject results in a Some
}

@subjectPresentOr(handler = handler) {
    This content will be present if handler#getSubject results in a Some
} {
        fallback content
}
```

11.3 SubjectNotPresent

Sometimes, you don't need fine-grained checked - you just need to see if there **is no** user present.

- be.objectify.deadbolt.scala.views.html.subjectNotPresent
- be.objectify.deadbolt.scala.views.html.subjectNotPresentOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.apply()	The handler to use to apply the constraint.
timeout	() ⇒ Long	A function returning `deadbolt.scala.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Example 1

The default Deadbolt handler is used to obtain the subject.

```
@subjectNotPresent() {
    This content will be present if handler#getSubject results in a None
}

@subjectNotPresentOr() {
    This content will be present if handler#getSubject results in a None
} {
        fallback content
}
```

Example 2

A specific Deadbolt handler is used to obtain the subject.

```
@(handler: DeadboltHandler)
@subjectNotPresent(handler = handler) {
    This content will be present if handler#getSubject results in a None
}

@subjectNotPresentOr(handler = handler) {
    This content will be present if handler#getSubject results in a None
} {
        fallback content
}
```

11.4 Restrict

Use `Subjects Roles` to perform AND/OR/NOT checks. The values given to the constraint must match the `Role.name` of the subject's roles.

AND is defined as an `Array[String]`, OR is a `List[Array[String]]`, and NOT is a rolename with a `!` preceding it.

- be.objectify.deadbolt.scala.views.html.restrict
- be.objectify.deadbolt.scala.views.html.restrictOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.apply()	The handler to use to apply the constraint.
roles	List[Array[String]]		The AND/OR/NOT restrictions. One array defines an AND, multiple arrays define OR.
timeout	() ⇒ Long	A function returning `deadbolt.scala.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Example 1

The subject must have the "foo" role.

```
@restrict(roles = List(Array("foo"))) {
    Subject requires the foo role for this to be visible
}
```

Example 2

The subject must have the "foo" AND "bar" roles.

```
@restrict(List(Array("foo", "bar")) {
     Subject requires the foo AND bar roles for this to be visible
}
```

Example 3

The subject must have the "foo" OR "bar" roles.

```
@restrict(List(Array("foo"), Array("bar"))) {
     Subject requires the foo OR bar role for this to be visible
}
```

Example 3

The subject must have the “foo” OR “bar” roles, or fallback content will be displayed.

```
@restrictOr(List(Array("foo", "bar")) {
     Subject requires the foo AND bar roles for this to be visible
} {
        Subject does not have the necessary roles
}
```

11.5 Pattern

Use the `Subject`s `Permission`s to perform a variety of checks.

- be.objectify.deadbolt.scala.views.html.pattern
- be.objectify.deadbolt.scala.views.html.patternOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.apply()	The handler to use to apply the constraint.
value	String		The value of the pattern, e.g. a regex or a precise match.
patternType	PatternType	PatternType.EQUALITY	
timeout	() ⇒ Long	A function returning `deadbolt.scala.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Example 1

The subject and `DynamicResourceHandler` are obtained from the default handler, and must have a permission with the exact value "admin.printer".

```
@pattern("admin.printer") {
    Subject must have a permission with the exact value "admin.printer" for this to be vis\
ible
}
```

Example 2

The subject and `DynamicResourceHandler` are obtained from the default handler, and must have a permission that matches the specified regular expression.

```
@pattern("(.)*\.printer", PatternType.REGEX) {
        Subject must have a permission that matches the regular expression (without quotes) "(.)*
\.printer" for this to be visible
}
```

Example 3

The `DynamicResourceHandler` is obtained from the default handler and used to apply the custom test

```
@pattern("something arbitrary", PatternType.CUSTOM) {
        DynamicResourceHandler#checkPermission must result in true for this to be visible
}
```

Example 4

Fallback content is displayed if the user does not have a permission exactly matching "admin.printer".

```
@patternOr("admin.printer") {
    Subject must have a permission with the exact value "admin.printer" for this to be vis\
ible
} {
        Subject did not have necessary permissions
}
```

11.6 Dynamic

The most flexible constraint - this is a completely user-defined constraint that uses `DynamicResourceHandler#isAllowed` to determine access.

- be.objectify.deadbolt.scala.views.html.dynamic
- be.objectify.deadbolt.scala.views.html.dynamicOr

Parameter	Type	Default	Notes
handler	DeadboltHandler	handlerCache.apply()	The handler to use to apply the constraint.
name	String		The name of the constraint, passed into the `DynamicResourceHandler`.
meta	PatternType	null	
timeout	() ⇒ Long	A function returning `deadbolt.scala.view-timeout` if it's defined, otherwise 1000L	The timeout applied to blocking calls.

Example 1

The `DynamicResourceHandler` is obtained from the default handler and is used to apply a named constraint to the content.

```
@dynamic(name = "someName") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

Example 2

The `DynamicResourceHandler` is obtained from the default handler and is used to apply a named constraint to the content with some hard-coded meta data.

```
@dynamic(name = "someName", meta = "foo") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

Example 3

The `DynamicResourceHandler` is obtained from the default handler and is used to apply a named constraint to the content with some dynamically-defined meta data.

```
@(someMetaValue: String)
@dynamic(name = "someName", meta = someMetaValue) {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```

Example 4

The `DynamicResourceHandler` is obtained from a specific handler and is used to apply a named constraint.

```
@(handler: DeadboltHandler)
@dynamic(handler = handler, name = "someName") {
    DynamicResourceHandler#isAllowed must result in true for this to be visible
}
```