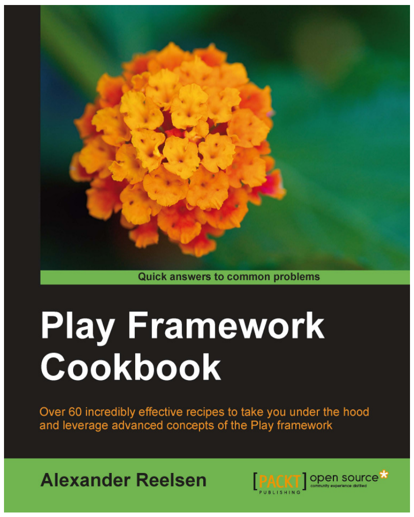
Play Framework Cookbook

Alexander Reelsen



Chapter No. 2
"Using Controllers"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Using Controllers"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Alexander Reelsen is a software engineer living in Munich, Germany, where he has been working on different software systems, for example, a touristic booking engine, a campaign management and messaging platform, and a b2b ecommerce portal. He has been using the Play framework since 2009 and was immediately astonished by the sheer simplicity of this framework, while still being pure Java. His other interests includes scaling shared-nothing web architectures and NoSQL databases.

Being a system engineer most of the time, when he started playing around with Linux at the age of 14, Alexander got to know software engineering during studies and decided that web applications are more interesting than system administration.

If not hacking in front of his notebook, he enjoys playing a good game of basketball or streetball.

Sometimes he even tweets at `http://twitter.com/spinscale` and can be reached anytime at `alexander@reelsen.net`.

If I do not thank my girlfriend for letting me spend more time with the laptop than with her while writing this book, I fear unknown consequences. So, thanks Christine!

Uncountable appreciation goes out to my parents for letting me spent days and (possibly not knowing) nights in front of the PC, and to my brother Stefan, who introduced me into the world of IT - which worked pretty well until now.

Thanks for the inspiration, fun, and fellowship to all my current and former colleagues, mainly of course to the developers. They always open up views and opinions to make developing enjoyable.

Many thanks go out to the Play framework developers and especially Guillaume, but also to the other core developers. Additionally, thanks to all of the people on the mailing list providing good answers to many questions and all the people working on tickets and helping to debug issues I had while writing this book.

My last appreciation goes to everyone at Packt Publishing, who was involved in this book.

Play Framework Cookbook

The Play framework is the new kid on the block of Java frameworks. By breaking the existing standards it tries not to abstract away from HTTP as with most web frameworks, but tightly integrates with it. This means quite a shift for Java programmers. Understanding the concepts behind this shift and its impact on web development with Java are crucial for fast development of Java web applications.

The Play Framework Cookbook starts where the beginner's documentation ends. It shows you how to utilize advanced features of the Play framework—piece by piece and completely outlined with working applications!

The reader will be taken through all layers of the Play framework and provided with in-depth knowledge with as many examples and applications as possible. Leveraging the most from the Play framework means, learning to think simple again in a Java environment. Think simple and implement your own renderers, integrate tightly with HTTP, use existing code, and improve sites' performance with caching and integrating with other web 2.0 services. Also get to know about non-functional issues like modularity, integration into production, and testing environments. In order to provide the best learning experience during reading of Play Framework Cookbook, almost every example is provided with source code. Start immediately integrating recipes into your Play application.

What This Book Covers

Chapter 1, Basics of the Play Framework, explains the basics of the Play framework. This chapter will give you a head start about the first steps to carry out after you create your first application. It will provide you with the basic knowledge needed for any advanced topic.

Chapter 2, Using Controllers, will help you to keep your controllers as clean as possible, with a well defined boundary to your model classes.

Chapter 3, Leveraging Modules, gives a brief overview of some modules and how to make use of them. It should help you to speed up your development when you need to integrate existing tools and libraries.

Chapter 4, Creating and Using APIs, shows a practical example of integrating an API into your application, and provides some tips on what to do when you are a data provider yourself, and how to expose an API to the outside world.

Chapter 5, Introduction To Writing Modules, explains everything related to writing modules.

Chapter 6, Practical Module Examples, shows some examples used in productive applications. It also shows an integration of an alternative persistence layer, how to create a Solr module for better search, and how to write an alternative distributed cache implementation among others.

Chapter 7, Running Into Production, explains the complexity that begins once the site goes live. This chapter is targeted towards both groups, developers, as well as system administrators.

Appendix, Further Information About the Play Framework, gives you more information about where you can find help with Play.

2
Using Controllers

In this chapter, we will cover:

- URL routing using annotation-based configuration
- Basics of caching
- Using HTTP digest authentication
- Generating PDFs in your controllers
- Binding objects using custom binders
- Validating objects using annotations
- Adding annotation-based right checks to your controller
- Rendering JSON output
- Writing your own renderRSS method as controller output

Introduction

This chapter will help you to keep your controllers as clean as possible, with a well defined boundary to your model classes. Always remember that controllers are really only a thin layer to ensure that your data from the outside world is valid before handing it over to your models, or something needs to be specifically adapted to HTTP. The chapter will start with some basic recipes, but it will cover some more complex topics later on with quite a bit code, of course mostly explained with examples.

URL routing using annotation-based configuration

If you do not like the routes file, you can also describe your routes programmatically by adding annotations to your controllers. This has the advantage of not having any additional `config` file, but also poses the problem of your URLs being dispersed in your code.

You can find the source code of this example in the `examples/chapter2/annotation-controller` directory.

How to do it...

Go to your project and install the router module via `conf/dependencies.yml`:

```
require:
    - play
    - play -> router head
```

Then run `playdeps` and the router module should be installed in the `modules/` directory of your application. Change your controller like this:

```
@StaticRoutes({
        @ServeStatic(value="/public/", directory="public")
})
public class Application extends Controller {

    @Any(value="/", priority=100)
    public static void index() {
        forbidden("Reserved for administrator");
    }

    @Put(value="/", priority=2, accept="application/json")
    public static void hiddenIndex() {
        renderText("Secret news here");
    }

    @Post("/ticket")
    public static void getTicket(String username, String password) {
        String uuid = UUID.randomUUID().toString();
        renderJSON(uuid);
    }
}
```

How it works...

Installing and enabling the module should not leave any open questions for you at this point. As you can see in the controller, it is now filled with annotations that resemble the entries in the `routes.conf` file, which you could possibly have deleted by now for this example. However, then your application will not start, so you have to have an empty file at least.

The `@ServeStatic` annotation replaces the static command in the routes file. The `@StaticRoutes` annotation is just used for grouping several `@ServeStatic` annotations and could be left out in this example.

Each controller call now has to have an annotation in order to be reachable. The name of the annotation is the HTTP method, or `@Any`, if it should match all HTTP methods. Its only mandatory parameter is the value, which resembles the URI—the second field in the `routes.conf`. All other parameters are optional. Especially interesting is the `priority` parameter, which can be used to give certain methods precedence. This allows a lower prioritized catch-all controller like in the preceding example, but a special handling is required if the URI is called with the PUT method. You can easily check the correct behavior by using curl, a very practical command line HTTP client:

```
curl -v localhost:9000/
```

This command should give you a result similar to this:

```
> GET / HTTP/1.1
> User-Agent: curl/7.21.0 (i686-pc-linux-gnu) libcurl/7.21.0
OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 403 Forbidden
< Server: Play! Framework;1.1;dev
< Content-Type: text/html; charset=utf-8
< Set-Cookie: PLAY_FLASH=;Path=/
< Set-Cookie: PLAY_ERRORS=;Path=/
< Set-Cookie: PLAY_SESSION=0c7df945a5375480993f51914804284a3bb
ca726-%00___ID%3A70963572-b0fc-4c8c-b8d5-871cb842c5a2%00;Path=/
< Cache-Control: no-cache
< Content-Length: 32
<

<h1>Reserved for administrator</h1>
```

You can see the HTTP error message and the content returned. You can trigger a PUT request in a similar fashion:

```
curl -X PUT -v localhost:9000/

> PUT / HTTP/1.1
> User-Agent: curl/7.21.0 (i686-pc-linux-gnu) libcurl/7.21.0
OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 200 OK
< Server: Play! Framework;1.1;dev
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: PLAY_FLASH=;Path=/
< Set-Cookie: PLAY_ERRORS=;Path=/
< Set-Cookie: PLAY_SESSION=f0cb6762afa7c860dde3fe1907e8847347
6e2564-%00___ID%3A6cc88736-20bb-43c1-9d43-42af47728132%00;Path=/
< Cache-Control: no-cache
< Content-Length: 16

Secret news here
```

As you can see now, the priority has voted the controller method for the PUT method which is chosen and returned.

There's more...

The router module is a small, but handy module, which is perfectly suited to take a first look at modules and to understand how the routing mechanism of the Play framework works at its core. You should take a look at the source if you need to implement custom mechanisms of URL routing.

Mixing the configuration file and annotations is possible

You can use the router module and the routes file—this is needed when using modules as they cannot be specified in annotations. However, keep in mind that this is pretty confusing. You can check out more info about the router module at `http://www.playframework.org/modules/router`.

Basics of caching

Caching is quite a complex and multi-faceted technique, when implemented correctly. However, implementing caching in your application should not be complex, but rather the mindwork before, where you think about what and when to cache, should be. There are many different aspects, layers, and types (and their combinations) of caching in any web application. This recipe will give a short overview about the different types of caching and how to use them.

You can find the source code of this example in the `chapter2/caching-general` directory.

Getting ready

First, it is important that you understand where caching can happen—inside and outside of your Play application. So let's start by looking at the caching possibilities of the HTTP protocol. HTTP sometimes looks like a simple protocol, but is tricky in the details. However, it is one of the most proven protocols in the Internet, and thus it is always useful to rely on its functionalities.

HTTP allows the caching of contents by setting specific headers in the response. There are several headers which can be set:

- **Cache-Control**: This is a header which must be parsed and used by the client and also all the proxies in between.
- **Last-Modified**: This adds a timestamp, explaining when the requested resource had been changed the last time. On the next request the client may send an If-Modified-Since header with this date. Now the server may just return a HTTP 304 code without sending any data back.
- **ETag**: An ETag is basically the same as a Last-Modified header, except it has a semantic meaning. It is actually a calculated hash value resembling the resource behind the requested URL instead of a timestamp. This means the server can decide when a resource has changed and when it has not. This could also be used for some type of optimistic locking.

So, this is a type of caching on which the requesting client has some influence on. There are also other forms of caching which are purely on the server side. In most other Java web frameworks, the HttpSession object is a classic example, which belongs to this case.

Play has a cache mechanism on the server side. It should be used to store big session data, in this case any data exceeding the 4KB maximum cookie size. Be aware that there is a semantic difference between a cache and a session. You should not rely on the data being in the cache and thus need to handle cache misses.

You can use the Cache class in your controller and model code. The great thing about it is that it is an abstraction of a concrete cache implementation. If you only use one node for your application, you can use the built-in ehCache for caching. As soon as your application needs more than one node, you can configure a memcached in your `application.conf` and there is no need to change any of your code.

Furthermore, you can also cache snippets of your templates. For example, there is no need to reload the portal page of a user on every request when you can cache it for 10 minutes.

This also leads to a very simple truth. Caching gives you a lot of speed and might even lower your database load in some cases, but it is not free. Caching means you need RAM, lots of RAM in most cases. So make sure the system you are caching on never needs to swap, otherwise you could read the data from disk anyway. This can be a special problem in cloud deployments, as there are often limitations on available RAM.

The following examples show how to utilize the different caching techniques. We will show four different use cases of caching in the accompanying test. First test:

```
public class CachingTest extends FunctionalTest {

    @Test
    public void testThatCachingPagePartsWork() {
        Response response = GET("/");
        String cachedTime = getCachedTime(response);
        assertEquals(getUncachedTime(response), cachedTime);

        response = GET("/");
        String newCachedTime = getCachedTime(response);
        assertNotSame(getUncachedTime(response), newCachedTime);
        assertEquals(cachedTime, newCachedTime);
    }

    @Test
    public void testThatCachingWholePageWorks() throws Exception {
        Response response = GET("/cacheFor");
        String content = getContent(response);
        response = GET("/cacheFor");
        assertEquals(content, getContent(response));
        Thread.sleep(6000);
        response = GET("/cacheFor");
        assertNotSame(content, getContent(response));
    }

    @Test
    public void testThatCachingHeadersAreSet() {
        Response response = GET("/proxyCache");
        assertIsOk(response);
        assertHeaderEquals("Cache-Control", "max-age=3600", response);
    }

    @Test
    public void testThatEtagCachingWorks() {
        Response response = GET("/etagCache/123");
        assertIsOk(response);
        assertContentEquals("Learn to use etags, dumbass!", response);

        Request request = newRequest();

        String etag = String.valueOf("123".hashCode());
        Header noneMatchHeader =  new Header("if-none-match", etag);
        request.headers.put("if-none-match", noneMatchHeader);

        DateTime ago = new DateTime().minusHours(12);
        String agoStr = Utils.getHttpDateFormatter().format(ago.
          toDate());
        Header modifiedHeader = new Header("if-modified-since",
          agoStr);
        request.headers.put("if-modified-since", modifiedHeader);

        response = GET(request, "/etagCache/123");
        assertStatus(304, response);
    }


    private String getUncachedTime(Response response) {
        return getTime(response, 0);
    }

    private String getCachedTime(Response response) {
        return getTime(response, 1);
    }

    private String getTime(Response response, intpos) {
        assertIsOk(response);
        String content = getContent(response);
        return content.split("\n")[pos];
    }
}
```

The first test checks for a very nice feature. Since play 1.1, you can cache parts of a page, more exactly, parts of a template. This test opens a URL and the page returns the current date and the date of such a cached template part, which is cached for about 10 seconds. In the first request, when the cache is empty, both dates are equal. If you repeat the request, the first date is actual while the second date is the cached one.

The second test puts the whole response in the cache for 5 seconds. In order to ensure that expiration works as well, this test waits for six seconds and retries the request.

The third test ensures that the correct headers for proxy-based caching are set.

The fourth test uses an HTTP ETag for caching. If the `If-Modified-Since` and `If-None-Match` headers are not supplied, it returns a string. On adding these headers to the correct ETag (in this case the `hashCode` from the string `123`) and the date from 12 hours before, a 302 Not-Modified response should be returned.

How to do it...

Add four simple routes to the configuration as shown in the following code:

```
GET     /                    Application.index
GET     /cacheFor            Application.indexCacheFor
GET     /proxyCache          Application.proxyCache
GET     /etagCache/{name}    Application.etagCache
```

The application class features the following controllers:

```
public class Application extends Controller {

    public static void index() {
        Date date = new Date();
        render(date);
    }

    @CacheFor("5s")
    public static void indexCacheFor() {
        Date date = new Date();
        renderText("Current time is: " + date);
    }

    public static void proxyCache() {
        response.cacheFor("1h");
        renderText("Foo");
    }
```

```
        @Inject
        private static EtagCacheCalculator calculator;

        public static void etagCache(String name) {
            Date lastModified = new DateTime().minusDays(1).toDate();
            String etag = calculator.calculate(name);
            if(!request.isModified(etag, lastModified.getTime())) {
                throw new NotModified();
            }
            response.cacheFor(etag, "3h", lastModified.getTime());
            renderText("Learn to use etags, dumbass!");
        }
    }
```

As you can see in the controller, the class to calculate ETags is injected into the controller. This is done on startup with a small job as shown in the following code:

```
    @OnApplicationStart
    public class InjectionJob extends Job implements BeanSource {

        private Map<Class, Object>clazzMap = new HashMap<Class, Object>();

        public void doJob() {
            clazzMap.put(EtagCacheCalculator.class, new
              EtagCacheCalculator());
            Injector.inject(this);
        }

        public <T> T getBeanOfType(Class<T>clazz) {
            return (T) clazzMap.get(clazz);
        }
    }
```

The calculator itself is as simple as possible:

```
    public class EtagCacheCalculator implements ControllerSupport {

        public String calculate(String str) {
            return String.valueOf(str.hashCode());
        }
    }
```

The last piece needed is the template of the `index()` controller, which looks like this:

```
    Current time is: ${date}
    #{cache 'mainPage', for:'5s'}
    Current time is: ${date}
    #{/cache}
```

How it works...

Let's check the functionality per controller call. The `index()` controller has no special treatment inside the controller. The current date is put into the template and that's it. However, the caching logic is in the template here because not the whole, but only a part of the returned data should be cached, and for that a `#{cache}` tag used. The tag requires two arguments to be passed. The `for` parameter allows you to set the expiry out of the cache, while the first parameter defines the key used inside the cache. This allows pretty interesting things. Whenever you are in a page where something is exclusively rendered for a user (like his portal entry page), you could cache it with a key, which includes the user name or the session ID, like this:

```
#{cache 'home-' + connectedUser.email, for:'15min'}
${user.name}
#{/cache}
```

This kind of caching is completely transparent to the user, as it exclusively happens on the server side. The same applies for the `indexCacheFor()` controller. Here, the whole page gets cached instead of parts inside the template. This is a pretty good fit for non-personalized, high performance delivery of pages, which often are only a very small portion of your application. However, you already have to think about caching before. If you do a time consuming JPA calculation, and then reuse the cache result in the template, you have still wasted CPU cycles and just saved some rendering time.

The third controller call `proxyCache()` is actually the most simple of all. It just sets the proxy expire header called `Cache-Control`. It is optional to set this in your code, because your Play is configured to set it as well when the `http.cacheControl` parameter in your `application.conf` is set. Be aware that this works only in production, and not in development mode.

The most complex controller is the last one. The first action is to find out the last modified date of the data you want to return. In this case it is 24 hours ago. Then the ETag needs to be created somehow. In this case, the calculator gets a String passed. In a real-world application you would more likely pass the entity and the service would extract some properties of it, which are used to calculate the ETag by using a pretty-much collision-safe hash algorithm. After both values have been calculated, you can check in the request whether the client needs to get new data or may use the old data. This is what happens in the `request.isModified()` method.

If the client either did not send all required headers or an older timestamp was used, real data is returned; in this case, a simple string advising you to use an ETag the next time. Furthermore, the calculated ETag and a maximum expiry time are also added to the response via `response.cacheFor()`.

A last specialty in the `etagCache()` controller is the use of the `EtagCacheCalculator`. The implementation does not matter in this case, except that it must implement the `ControllerSupport` interface. However, the initialization of the injected class is still worth a mention. If you take a look at the `InjectionJob` class, you will see the creation of the class in the `doJob()` method on startup, where it is put into a local map. Also, the `Injector.inject()` call does the magic of injecting the `EtagCacheCalculator` instance into the controllers. As a result of implementing the `BeanSource` interface, the `getBeanOfType()` method tries to get the corresponding class out of the map. The map actually should ensure that only one instance of this class exists.

There's more...

Caching is deeply integrated into the Play framework as it is built with the HTTP protocol in mind. If you want to find out more about it, you will have to examine core classes of the framework.

More information in the ActionInvoker

If you want to know more details about how the `@CacheFor` annotation works in Play, you should take a look at the `ActionInvoker` class inside of it.

Be thoughtful with ETag calculation

Etag calculation is costly, especially if you are calculating more then the last-modified stamp. You should think about performance here. Perhaps it would be useful to calculate the ETag after saving the entity and storing it directly at the entity in the database. It is useful to make some tests if you are using the ETag to ensure high performance. In case you want to know more about ETag functionality, you should read RFC 2616.

You can also disable the creation of ETags totally, if you set `http.useETag=false` in your `application.conf`.

Use a plugin instead of a job

The job that implements the `BeanSource` interface is not a very clean solution to the problem of calling `Injector.inject()` on start up of an application. It would be better to use a plugin in this case.

See also

The cache in Play is quite versatile and should be used as such. We will see more about it in all the recipes in this chapter. However, none of this will be implemented as a module, as it should be. This will be shown in *Chapter 6*, *Practical Module Examples*.

Using HTTP digest authentication

As support for HTTP, basic authentication is already built-in with Play. You can easily access `request.user` and `request.password` in your controller as using digest authentication is a little bit more complex. To be fair, the whole digest authentication is way more complex.

You can find the source code of this example in the `chapter2/digest-auth` directory.

Getting ready

Understanding HTTP authentication in general is quite useful, in order to grasp what is done in this recipe. For every HTTP request the client wants to receive a resource by calling a certain URL. The server checks this request and decides whether it should return either the content or an error code and message telling the client to provide needed authentication. Now the client can re-request the URL using the correct credentials and get its content or just do nothing at all.

When using HTTP basic authentication, the client basically just sends some user/password combination with its request and hopes it is correct. The main problem of this approach is the possibility to easily strip the username and password from the request, as there are no protection measures for basic authentication. Most people switch to an SSL-encrypted connection in this case in order to mitigate this problem. While this is perfectly valid (and often needed because of transferring sensitive data), another option is to use HTTP digest authentication. Of course digest authentication does not mean that you cannot use SSL. If all you are worrying about is your password and not the data you are transmitting, digest authentication is just another option.

In basic authentication the user/password combination is sent in almost cleartext over the wire. This means the password does not need to be stored as cleartext on the server side, because it is a case of just comparing the hash value of the password by using MD5 or SHA1. When using digest authentication, only a hash value is sent from client to server. This implies that the client and the server need to store the password in cleartext in order to compute the hash on both sides.

How to do it...

Create a user entity with these fields:

```
@Entity
public class User extends Model {

        public String name;
        public String password;      // hashed password
        public String apiPassword; // cleartext password
}
```

Create a controller which has a `@Before` annotation:

```
public class Application extends Controller {

    @Before
    static void checkDigestAuth() {
        if (!DigestRequest.isAuthorized(request)) {
            throw new UnauthorizedDigest("Super Secret Stuff");
        }
    }

    public static void index() {
        renderText("The date is " + new Date());
    }
}
```

The controller throws an `UnauthorizedDigest` exception, which looks like this:

```
public class UnauthorizedDigest extends Result {

    String realm;

    public UnauthorizedDigest(String realm) {
        this.realm = realm;
    }

    @Override
    public void apply(Request request, Response response) {
        response.status = Http.StatusCode.UNAUTHORIZED;
        String auth = "Digest realm=" + realm + ", nonce=" +
           Codec.UUID();
        response.setHeader("WWW-Authenticate", auth);
    }
}
```

The digest request handles the request and checks the authentication:

```
class DigestRequest {

    private Map<String,String>params = new HashMap<String,String>();
    private Request request;

    public DigestRequest(Request request) {
        this.request = request;
    }
```

```
        public booleanisValid() {
    ...
        }

        public booleanisAuthorized() {
            User user = User.find("byName", params.get("username")).
                         first();
            if (user == null) {
                throw new UnauthorizedDigest(params.get("realm"));
            }

            String digest = createDigest(user.apiPassword);
            return digest.equals(params.get("response"));
        }

        private String createDigest(String pass) {
    ...
        }

        public static booleanisAuthorized(Http.Request request) {
            DigestRequest req = new DigestRequest(request);
            return req.isValid() && req.isAuthorized();
        }
    }
```

How it works...

As you can see, all it takes is four classes. The user entity should be pretty clear, as it only exposes three fields, one being a login and two being passwords. This is just to ensure that you should never store a user's master password in cleartext, but use additional passwords if you implement some cleartext password dependant application.

The next step is a controller, which returns a HTTP 403 with the additional information requiring HTTP digest authentication. The method annotated with the `Before` annotation is always executed before any controller method as this is the perfect place to check for authentication. The code checks whether the request is a valid authenticated request. If this is not the case an exception is thrown. In Play, every `Exception` which extends from `Result` actually can return the request or the response.

Taking a look at the `UnauthorizedDigest` class you will notice that it only changes the HTTP return code and adds the appropriate WWW-Authenticate header. The WWW-Authenticate header differs from the one used with basic authentication in two ways. First it marks the authentication as "Digest", but it also adds a so-called nonce, which should be some random string. This string must be used by the client to create the hash and prevents bruteforce attacks by never sending the same `user/password/nonce` hash combination.

The heart of this recipe is the `DigestRequest` class, which actually checks the request for validity and also checks whether the user is allowed to authenticate with the credentials provided or not. Before digging deeper, it is very useful to try the application using `curl` and observing what the headers look like. Call `curl` with the following parameters:

```
curl --digest --user alex:test -v localhost:9000
```

The response looks like the following (unimportant output and headers have been stripped):

```
> GET / HTTP/1.1
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Digest realm=Super Secret Stuff, nonce=3ef81305-
745c-40b9-97d0-1c601fe262ab
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Issue another request to this URL: 'HTTP://localhost:9000'

> GET / HTTP/1.1
> Authorization: Digest username="alex", realm="Super Secret Stuff",
nonce="3ef81305-745c-40b9-97d0-1c601fe262ab", uri="/", response="6e97a
12828d940c7dc1ff24dad167d1f"
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 20
<
This is top secret!
```

Curl actually issues two requests. The first returns a 403 "not authorized" error, but also the nonce, which is used together with the username and password in the second request to create the response field inside the WWW-Authenticate header. As the client also sends the nonce and the username inside the header, the server can reconstruct the whole response on the server side. This means it is actually stateless and the server does not need to store any data between two requests.

Looking at the `DigestRequest` class, it is comprised of three core methods: `isValid()`, `isAuthorized()`, and `createDigest()`. The `isValid()` method checks whether a request contains all the needed data in order to be able to compute and compare the hash. The `isAuthorized()` method does a database lookup of the user's cleartext password and hands it over to the `createDigest` method, which computes the response hash and returns true if the computed hash with the local password is the same as the hash sent in the request. If they are not, the authentication has to fail.

The static `DigestRequest.isAuthorized()` method is a convenient method to keep the code in the controller as short as possible.

There are two fundamental disadvantages in the preceding code snippet. First, it is implementation dependent, because it directly relies on the user entity and the password field of this entity. This is not generic and has to be adapted for each implementation. Secondly, it only implements the absolute minimum subset of HTTP digest authentication. Digest authentication is quite complex if you want to support it with all its variations and options. There are many more options and authentication options, hashing algorithms, and optional fields which have to be supported in order to be RFC-compliant. You should see this only as a minimum starting point to get this going. Also this should not be thought of as secure, because without an additional header called "qop", every client will switch to a less secure mode. You can read more about that in RFC2069 and RFC2617.

There's more...

You can also verify this recipe in your browser by just pointing it to `http://localhost:9000/`. An authentication window requiring you to enter username and password will popup.

Get more info about HTTP digest authentication

As this recipe has not even covered five percent of the specification, you should definitely read the corresponding RFC at `http://tools.ietf.org/html/rfc2617` as well as RFC2069 at `http://tools.ietf.org/html/rfc2617`.

See also

In order to be application-independent you could use annotations to mark the field of the entity to be checked. The recipe *Rendering JSON output* will show you how to use an annotation to mark a field not to be exported via JSON.

Generating PDFs in your controllers

Generating binary content is a standard procedure in every web application, be it a dynamic generated image such as a CAPTCHA or user-specific document such as an invoice or an order confirmation. Play already supports the `renderBinary()` command in the controller to send binary data to the browser, however this is quite low level. This recipe shows how to combine the use of Apache FOP – which allows creation of PDF data out of XML-based templates – and the Play built-in templating mechanism to create customized PDF documents in real time.

You can find the source code of this example in the `chapter2/pdf` directory.

Getting ready

As there is already a PDF module included in Play, you should make sure you disable it in your application in order to avoid clashes. This of course only applies, if it has already been enabled before.

How to do it...

First you should download Apache FOP from `http://www.apache.org/dyn/closer.cgi/xmlgraphics/fop` and unpack it into your application. Get the ZIP file and unzip it so that there is a `fop-1.0` directory in your application depending on your downloaded version.

Now you have to copy the JAR files into the `lib/` directory, which is always included in the classpath when your application starts.

```
cp fop-1.0/build/fop.jar lib/
cp fop-1.0/lib/*.jar lib/
cp fop-1.0/examples/fo/basic/simple.fo app/views/Application/index.fo
rm lib/commons*
```

Make sure to remove the commons JAR files from the `lib` directory, as Play already provides them. In case of using Windows, you would have to use `copy` and `del` as commands instead of the Unix commands `cp` and `rm`. Instead of copying these files manually you could also add the entry to `conf/dependencies.yml`. However, you would have to exclude many dependencies manually, which can be removed as well.

Create a dummy User model, which is rendered in the PDF:

```
public class User {

        public String name = "Alexander";
        public String description = "Random: " +
RandomStringUtils.randomAlphanumeric(20);
}
```

You should now replace the content of the freshly copied `app/views/Application/index.fo` file to resemble something from the user data like you would do it in a standard HTML template file in Play:

```
<fo:block font-size="18pt"
            ...
            padding-top="3pt">
    ${user.name}
</fo:block>

<fo:block font-size="12pt"
            ...
            text-align="justify">
    ${user.description}
</fo:block>
```

Change the application controller to call `renderPDF()` instead of `render()`:

```
import static pdf.RenderPDF.renderPDF;

public class Application extends Controller {

    public static void index() {
        User user = new User();
        renderPDF(user);
    }
}
```

Now the only class that needs to be implemented is the `RenderPDF` class in the PDF package:

```
public class RenderPDF extends Result {

        private static FopFactoryfopFactory = FopFactory.
         newInstance();
        private static TransformerFactorytFactory =
         TransformerFactory.newInstance();
        private VirtualFiletemplateFile;

        public static void renderPDF(Object... args) {
            throw new RenderPDF(args);
        }

        public RenderPDF(Object ... args) {
            populateRenderArgs(args);
            templateFile = getTemplateFile(args);
        }

        @Override
        public void apply(Request request, Response response) {
            Template template = TemplateLoader.load(templateFile);

            String header = "inline; filename=\"" + request.
actionMethod + ".pdf\"";
            response.setHeader("Content-Disposition", header);

            setContentTypeIfNotSet(response, "application/pdf");

            try {
                    Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF,
response.out);
                    Transformer transformer = tFactory.
newTransformer();
                    Scope.RenderArgsargs = Scope.RenderArgs.current();
                    String content = template.render(args.data);
                    InputStream is = IOUtils.toInputStream(content);
                    Source src = new StreamSource(is);
                    javax.xml.transform.Result res = new SAXResult(fop.
getDefaultHandler());
                    transformer.transform(src, res);
            } catch (FOPException e) {
                    Logger.error(e, "Error creating pdf");
            } catch (TransformerException e) {
                    Logger.error(e, "Error creating pdf");
            }
        }

        private void populateRenderArgs(Object ... args) {
            Scope.RenderArgsrenderArgs = Scope.RenderArgs.current();
            for (Object o : args) {
                    List<String> names = LocalVariablesNamesTracer.
getAllLocalVariableNames(o);
                    for (String name : names) {
                        renderArgs.put(name, o);
                    }
            }
            renderArgs.put("session", Scope.Session.current());
            renderArgs.put("request", Http.Request.current());
            renderArgs.put("flash", Scope.Flash.current());
            renderArgs.put("params", Scope.Params.current());
            renderArgs.put("errors", Validation.errors());
        }

        private VirtualFilegetTemplateFile(Object ... args) {
            final Http.Request request = Http.Request.current();
            String templateName = null;
            List<String>renderNames = LocalVariablesNamesTracer.getAll
LocalVariableNames(args[0]);
            if (args.length> 0 &&args[0] instanceof String
&&renderNames.isEmpty()) {
                templateName = args[0].toString();
            } else {
                templateName = request.action.replace(".", "/") +
".fo";
            }

            if (templateName.startsWith("@")) {
                templateName = templateName.substring(1);
                if (!templateName.contains(".")) {
                    templateName = request.controller + "." +
templateName;
                }
                templateName = templateName.replace(".", "/") + ".fo";
            }

            VirtualFile file = VirtualFile.search(Play.templatesPath,
templateName);
            return file;
        }

}
```

How it works...

Before trying to understand how this example works, you could also fire up the included example of this application under `examples/chapter2/pdf` and open `http://localhost:9000/` which will show you a PDF that includes the user data defined in the entity.

When opening the PDF, an XML template is rendered by the Play template engine and later processed by Apache FOP. Then it is streamed to the client. Basically, there is a new `renderPDF()` method created, which does all this magic. This method is defined in the `pdf.RenderPDF` class. All you need to hand over is a user to render.

The `RenderPDF` is only a rendering class, similar to the `DigestRequest` class in the preceding recipe. It consists of a static `renderPDF()` method usable in the controller and of three additional methods.

The `getTemplateFile()` method finds out which template to use. If no template was specified, a template with the name as the called method is searched for. Furthermore it is always assumed that the template file has a `.fo` suffix. The `VirtualFile` class is a Play helper class, which makes it possible to use files inside archives (like modules) as well. The `LocalVariablesNamesTracer` class allows you to get the names and the objects that should be rendered in the template.

The `populateRenderArgs()` method puts all the standard variables into the list of arguments which are used to render the template, for example, the session or the request.

The heart of this recipe is the `apply()` method, which sets the response content type to `application/pdf` and uses the Play built-in template loader to load the `.fo` template. After initializing all required variables for ApacheFOP, it renders the template and hands the rendered string over to the FOP transformer. The output of the PDF creation has been specified when calling the FopFactory. It goes directly to the output stream of the response object.

There's more...

As you can see, it is pretty simple in Play to write your own renderer. You should do this whenever possible, as it keeps your code clean and allows clean splitting of view and controller logic. You should especially do this to ensure that complex code such as Apache FOP does not sneak in to your controller code and make it less readable.

This special case poses one problem. Creating PDFs might be a long running task. However, the current implementation does not suspend the request. There is a solution to use the `await()` code from the controller in your own responses as seen in *Chapter 1*.

More about Apache FOP

Apache FOP is a pretty complex toolkit. You can create really nifty PDFs with it; however, it has quite a steep learning curve. If you intend to work with it, read the documentation under `http://xmlgraphics.apache.org/fop/quickstartguide.html` and check the examples directory (where the `index.fo` file used in this recipe has been copied from).

Using other solutions to create PDFs

There are many other solutions, which might fit your needs better than one based on `xsl-fo`. Libraries such as iText can be used to programmatically create PDFs. Perhaps even the PDF module available in the module repository will absolutely fit your needs.

See also

There is also the recipe *Writing your own renderRSS method as controller output* for writing your own RSS renderer at the end of this chapter.

Binding objects using custom binders

You might already have read the Play documentation about object binding. As validation is extremely important in any application, it basically has to fulfill several tasks.

First, it should not allow the user to enter wrong data. After a user has filled a form, he should get a positive or negative feedback, irrespective of whether the entered content was valid or not. The same goes for storing data. Before storing data you should make sure that storing it does not pose any future problems as now the model and the view layer should make sure that only valid data is stored or shown in the application. The perfect place to put such a validation is the controller.

As a HTTP request basically is composed of a list of keys and values, the web framework needs to have a certain logic to create real objects out of this argument to make sure the application developer does not have to do this tedious task.

You can find the source code of this example in the `chapter2/binder` directory.

How to do it...

Create or reuse a class you want created from an item as shown in the following code snippet:

```
public class OrderItem {

    @Required public String itemId;
    public Boolean hazardous;
    public Boolean bulk;
    public Boolean toxic;
    public Integer piecesIncluded;

    public String toString() {
       return MessageFormat.format("{0}/{1}/{2}/{3}/{4}", itemId,
piecesIncluded, bulk, toxic, hazardous);
    }
}
```

Create an appropriate form snippet for the `index.xml` template:

```
#{form @Application.createOrder()}
<input type="text" name="item" /><br />
<input type="submit" value="Create Order">
#{/form}
```

Create the controller:

```
public static void createOrder(@Valid OrderItem item) {
    if (validation.hasErrors()) {
        render("@index");
    }

        renderText(item.toString());
}
```

Create the type binder doing this magic:

```
@Global
public class OrderItemBinder implements TypeBinder<OrderItem> {

    @Override
    public Object bind(String name, Annotation[] annotations, String
value,
            Class actualClass) throws Exception {

        OrderItem item = new OrderItem();
        List<String> identifier = Arrays.asList(value.split("-", 3));

        if (identifier.size() >= 3) {
            item.piecesIncluded = Integer.parseInt(identifier.get(2));
}

        if (identifier.size() >= 2) {
            int c = Integer.parseInt(identifier.get(1));
            item.bulk = (c & 4) == 4;
            item.hazardous = (c & 2) == 2;
            item.toxic = (c & 1) == 1;
        }

        if (identifier.size() >= 1) { item.itemId = identifier.get(0);
}

        return item;
    }
}
```

How it works...

With the exception of the binder definition all of the preceding code has been seen earlier. By working with the Play samples you already got to know how to handle objects as arguments in controllers. This specific example creates a complete object out of a simple String. By naming the string in the form value (`<input …name="item" />`) the same as the controller argument name (`createOrder(@Valid OrderItem item)`) and using the controller argument class type in the `OrderItemBinder` definition (`OrderItemBinder implements TypeBinder<OrderItem>`), the mapping is done.

The binder splits the string by a hyphen, uses the first value for item ID, the last for `piìesIncluded`, and checks certain bits in order to set some Boolean properties.

By using `curl` you can verify the behavior very easily as shown:

```
curl -v -X POST --data "item=Foo-3-5" localhost:9000/order

Foo/5/false/true/true
```

Here `Foo` resembles the item ID, `5` is the `piecesIncluded` property, and `3` is the argument means that the first two bits are set and so the hazardous and toxic properties are set, while bulk is not.

There's more...

The `TypeBinder` feature has been introduced in Play 1.1 and is documented at `http://www.playframework.org/documentation/1.2/controllers#custombinding`.

Using type binders on objects

Currently, it is only possible to create objects out of one single string with a `TypeBinder`. If you want to create one object out of several submitted form values you will have to create your own plugin for this as workaround. You can check more about this at:

`http://groups.google.com/group/play-framework/browse_thread/thread/62e7fbeac2c9e42d`

Be careful with JPA using model classes

As soon as you try to use model classes with a type binder you will stumble upon strange behavior, as your objects will always only have null or default values when freshly instanced. The JPA plugin already uses a binding and overwrites every binding you are doing.

Validating objects using annotations

Basic validation should be clear for you now. It is well-documented in the official documentation and shows you how to use the different annotations such as `@Min`, `@Max`, `@Url`, `@Email`, `@InFuture`, `@InPast`, or `@Range`. You should go a step forward and add custom validation. An often needed requirement is to create some unique string used as identifier. The standard way to go is to create a UUID and use it. However, validation of the UUID should be pretty automatic and you want to be sure to have a valid UUID in your models.

You can find the source code of this example in the `chapter2/annotation-validation` directory.

Getting ready

As common practice is to develop your application in a test driven way, we will write an appropriate test as first code in this recipe. In case you need more information about writing and using tests in Play, you should read `http://www.playframework.org/documentation/1.2/test`.

This is the test that should work:

```
public class UuidTest extends FunctionalTest {

    @Test
    public void testThatValidUuidWorks() {
        String uuid = UUID.randomUUID().toString();
        Response response = GET("/" + uuid);
        assertIsOk(response);
        assertContentEquals(uuid + " is valid", response);
    }

    @Test
    public void testThatInvalidUuidWorksNot() {
        Response response = GET("/absolutely-No-UUID");
        assertStatus(500, response);
    }
}
```

So whenever a valid UUID is used in the URL, it should be returned in the body and whenever an invalid UUID has been used, it should return HTTP error 500.

How to do it...

Add an appropriate configuration line to your `conf/routes` file:

```
GET     /{uuid}    Application.showUuid
```

Create a simple @UUID annotation, practically in its own annotations or validations package:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Constraint(checkWith = UuidCheck.class)
public @interface Uuid {
        String message() default "validation.invalid.uuid";
}
```

Create the appropriate controller, which uses the @Uuid annotation:

```
public class Application extends Controller {

    public static void showUuid(@Uuid String uuid) {
        if (validation.hasErrors()) {
            flash.error("Fishy uuid");
            error();
        }

        renderText(uuid + " is valid");
    }
}
```

Create the check, which is triggered by the validation. You might want to put it into the checks package:

```
public class UuidCheck extends AbstractAnnotationCheck<Uuid> {

    @Override
    public booleanisSatisfied(Object validatedObject, Object value,
    OValContext context, Validator validator)
    throws OValException {

        try {
            UUID.fromString(value.toString());
            return true;
        } catch (IllegalArgumentException e) {}

        return false;
    }
}
```

How it works...

When starting your application via `play test` and going to `http://localhost:9000/@tests` you should be able to run the `UuidTest` without problems.

Except the `UuidCheck` class, most of this here is old stuff. The Uuid annotation has two specialties. First it references the `UuidCheck` with a constraint annotation and second you can specify a message as argument. This message is used for internationalization.

The UuidCheck class is based on an Oval class. Oval is a Java library and used by the Play framework for most of the validation tasks and can be pretty easily extended as you can see here. All you need to implement is the `isSatisfied()` method. In this case it has tried to convert a String to a UUID. If it fails, the runtime exception thrown by the conversion is caught and false is returned, marking the check as invalid.

There's more...

The oval framework is pretty complex and the logic performed here barely scratches the surface. For more information about oval, check the main documentation at `http://oval.sourceforge.net/`.

Using the configure() method for setup

The `AbstractAnnotationCheck` class allows you to overwrite the configure(T object) method (where T is generic depending on your annotation). This allows you to set up missing annotation parameters with default data; for example, default values for translations. This is done by many of the already included Play framework checks as well.

Annotations can be used in models as well

Remember that the annotation created above may also be used in your models, so you can label any String as a UUID in order to store it in your database and to make sure it is valid when validating the whole object.

```
@Uuid public String registrationUuid;
```

Adding annotation-based right checks to your controller

Sooner or later any business application will have some login/logout mechanism and after that there are different types of users. Some users are allowed to do certain things, others are not. This can be solved via a check in every controller, but is way too much overhead. This recipe shows a clean and fast (though somewhat limited) solution to creating security checks, without touching anything of the business logic inside your controller.

You can find the source code of this example in the `chapter2/annotation-rights` directory.

Getting ready

Again we will start with a test, which performs several checks for security:

```
public class UserRightTest extends FunctionalTest {

    @Test
    public void testSecretsWork() {
        login("user", "user");
        Response response = GET("/secret");
        assertIsOk(response);
        assertContentEquals("This is secret", response);
    }

    @Test
    public void testSecretsAreNotFoundForUnknownUser() {
        Response response = GET("/secret");
        assertStatus(404, response);
    }

    @Test
    public void testSuperSecretsAreAllowedForAdmin() {
        login("admin", "admin");
        Response response = GET("/top-secret");
        assertIsOk(response);
        assertContentEquals("This is top secret", response);
    }

    @Test
    public void testSecretsAreDeniedForUser() {
        login("user", "user");
        Response response = GET("/top-secret");
        assertStatus(403, response);
    }

    private void login(String user, String pass) {
        String data = "username=" + user + "&password=" + pass;
        Response response = POST("/login",
                APPLICATION_X_WWW_FORM_URLENCODED, data);
        assertIsOk(response);
    }
}
```

As you can see here, every test logs in with a certain user first, and then tries to access a resource. Some are intended to fail, while some should return a successful access. Before every access, a login is needed using the `login()` method. In case you wonder why a simple HTTP request stores the returned login credentials for all of the next requests, this is actually done by the logic of the `FunctionalTest`, which stores all returned cookies from the login request during the rest of the test.

How to do it...

Add the needed routes:

```
POST    /login          Application.login
GET      /secret        Application.secret
GET      /top-secret    Application.topsecret
```

Create User and Right entities:

```
@Entity
public class User extends Model {

    public String username;
    public String password;
    @ManyToMany
    public Set<Right> rights;

    public booleanhasRight(String name) {
        Right r = Right.find("byName", name).first();
        return rights.contains(r);
    }
}
```

A simple entity representing a right and consisting of a name is shown in the following code:

```
@Entity
public class Right extends Model {

    @Column(unique=true)
    public String name;
}
```

Create a Right annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Right {
    String value();
}
```

Lastly, create all the controller methods:

```
public class Application extends Controller {

    public static void index() {
        render();
    }

    @Before(unless = "login")
    public static void checkForRight() {
        String sessionUser = session.get("user");
        User user = User.find("byUsername", sessionUser).first();
        notFoundIfNull(user);

        Right right = getActionAnnotation(Right.class);
        if (!user.hasRight(right.value())) {
            forbidden("User has no right to do this");
        }
    }

    public static void login(String username, String password) {
        User user = User.find("byUsernameAndPassword", username,
password).first();

        if (user == null) {
            forbidden();
        }

        session.put("user", user.username);
    }

    @Right("Secret")
    public static void secret() {
        renderText("This is secret");
    }

    @Right("TopSecret")
    public static void topsecret() {
        renderText("This is top secret");
    }
}
```

How it works...

Going through this step by step reveals surprisingly few new items, but rather a simple and concise change at the core of each controller call. Neither the routes are new, nor the entity definitions, or its possibility to create the `hasRight()` method. The only real new logic is inside the controller. The logic here is not meant as business logic of your application but rather permission checking. On the one hand every security aware controller has a `@Right` annotation at its definition, which defines the required right as a text string.

On the other hand all the logic regard permissions is executed at the `checkForRight()` method before every controller call. It inspects the annotation value and checks whether the currently logged-in user has this specific annotation value as a right set using the `hasRight()` method defined in the user entity.

There's more...

This is a pretty raw method to check for rights. It imposes several design weaknesses and severe performance issues. But it is a start to go further.

Be flexible with roles instead of rights

The security model here is pretty weak. You should think of using roles on user level instead of rights, and check these roles for the rights called. This allows you to create less fine-grained permission checks such as a "Content editor" and a "publisher" role for example.

More speed with caching

The whole code presented here can be pretty slow. First you could cache the roles or rights of a certain user. Furthermore you could cache the security right of the controller action and the login credentials, which are looked up on every request.

Increased complexity with context-sensitive rights

The security checks compared here are very simple. If you want to have a right, then only the owner of an object can change it, you are not completely off with the solution presented here. You need to define more logic inside your controller call.

Check out the deadbolt module

There is a module which already does more checks and has more options than this example. You should take a look at the deadbolt module at `http://www.playframework.org/modules/deadbolt`. This module offers restrictions not only on the controller level, but also inside Views. Deadbolt also provides arbitrary dynamic security, which allows you to provide application-specific security strategies.

Rendering JSON output

As soon as a web application consists of a very fast frontend, no or seldom complete page reloads occur. This implies a complete rendering by the browser, which is one of the most time consuming tasks when loading a webpage in the browser. This means you have to get the data as fast as possible to the client. You can either send them as pre-rendered HTML, XML, or JSON format. Sending the data in either JSON or XML means the application on the browser side can render the data itself and decide how and when it should be displayed. This means your application should be able to create JSON or XML-based responses.

As JSON is quite popular, this example will not only show you how to return the JSON representation of an entity, but also how to make sure sensitive data such as a password will not get sent to the user.

Furthermore some hypermedia content will be added to the response, like an URL where more information can be found.

You can find the source code of this example in the `chapter2/json-render-properties` directory.

Getting ready

Beginning with a test is always a good idea:

```
public class JsonRenderTest extends FunctionalTest {

    @Test
    public void testThatJsonRenderingWorks() {
        Response response = GET("/user/1");
        assertIsOk(response);

        User user = new Gson().fromJson(getContent(response), User.
class);
        assertNotNull(user);
        assertNull(user.password);
        assertNull(user.secrets);
        assertEquals(user.login, "alex");
        assertEquals(user.address.city, "Munich");
        assertContentMatch("\"uri\":\"/user/1\"", response);
    }
}
```

This expects a JSON reply from the request and parses it into a User instance with the help of gson, a JSON library from Google, which is also used by Play for serializing. As we want to make sure that no sensitive data is sent, there is a check for nullified values of password and secrets properties. The next check goes for a user property and for a nested property inside another object. The last check has to be done by just checking for an occurrence of the string, because the URL is not a property of the user entity and is dynamically added by the special JSON serializing routine used in this example.

How to do it...

Create your entities first. This example consists of a user, an address, and a `SuperSecretData` entity:

```
@Entity
public class User extends Model {

    @SerializedName("userLogin")
    public String login;
    @NoJsonExport
    public String password;
    @ManyToOne
    public Address address;
    @OneToOne
    public SuperSecretData secrets;

    public String toString() {
        return id + "/" +  login;
    }
}


@Entity
public class Address extends Model {
    public String street;
    public String city;
    public String zip;
}


@Entity
public class SuperSecretData extends Model {
    public String secret = "foo";
}
```

The controller is simple as well:

```
public static void showUser(Long id) {
    User user = User.findById(id);
    notFoundIfNull(user);
    renderJSON(user, new UserSerializer());
}
```

The last and most important part is the serializer used in the controller above:

```
public class UserSerializer implements JsonSerializer<User> {

    public JsonElement serialize(User user, Type type,
        JsonSerializationContext context) {
        Gsongson = new GsonBuilder()
            .setExclusionStrategies(new LocalExclusionStrategy())
            .create();

      JsonElementelem = gson.toJsonTree(user);
      elem.getAsJsonObject().addProperty("uri", createUri(user.id));
      return elem;
    }

    private String createUri(Long id) {
      Map<String,Object> map = new HashMap<String,Object>();
      map.put("id", id);
      return Router.reverse("Application.showUser", map).url;
    }


    public static class LocalExclusionStrategy implements
ExclusionStrategy {

      public booleanshouldSkipClass(Class<?>clazz) {
        return clazz == SuperSecretData.class;
      }

      public booleanshouldSkipField(FieldAttributes f) {
        return f.getAnnotation(NoJsonExport.class) != null;
      }
    }
}
```

How it works...

The entities used in this example are simple. The only differences are the two annotations in the User entity. First there is a `SerializedNamed` annotation, which uses the annotation argument as field name in the json output – this annotation comes from the gson library. The `@NoJsonExport` annotation has been specifically created in this example to mark fields that should not be exported like a sensitive password field in this example. The address field is only used as an example to show how many-to-many relations are serialized in the JSON output.

As you might guess, the `SuperSecretData` class should mark the data as secret, so this field should not be exported as well. However, instead of using an annotation, the functions of the Google gson library will be utilized for this.

The controller call works like usual except that the `renderJson()` method gets a specific serializer class as argument to the object it should serialize.

The last class is the `UserSerializer` class, which is packed with features, although it is quite short. As the class implements the `JsonSerializer` class, it has to implement the `serialize()` method. Inside of this method a gson builder is created, and a specific exclusion strategy is added. After that the user object is automatically serialized by the gson object. Lastly another property is added. This property is the URI of the `showUser()` controller call, in this case something like `/user/{id}` . You can utilize the Play internal router to create the correct URL.

The last part of the serializer is the `ExclusionStrategy`, which is also a part of the gsonserializer. This strategy allows exclusion of certain types of fields. In this case the method `shouldSkipClass()` excludes every occurrence of the `SuperSecretData` class, where the method `shouldSkipFields()` excludes fields marked with the `@NoJsonExport` annotation.

There's more...

If you do not want to write your own JSON serializer you could also create a template ending with `.json` and write the necessary data like in a normal HTML template. However there is no automatic escaping, so you would have to take care of that yourself.

More about Google gson

Google gson is a pretty powerful JSON library. Once you get used to it and learn to utilize its features it may help you to keep your code clean. It has very good support for specific serializers and deserializers, so make sure you check out the documentation before trying to build something for yourself. Read more about it at `http://code.google.com/p/google-gson/`.

Alternatives to Google gson

Many developers do not like the gson library at all. There are several alternatives. There is a nice example of how to integrate FlexJSON. Check it out at `http://www.lunatech-research.com/archives/2011/04/20/play-framework-better-json-serialization-flexjson`.

Writing your own renderRSS method as controller output

Nowadays, an almost standard feature of web applications is to provide RSS feeds, irrespective of whether it is for a blog or some location-based service. Most clients can handle RSS out of the box. The Play examples only carry an example with hand crafted RSS feeds around. This example shows how to use a library for automatic RSS feed generation by getting the newest 20 post entities and rendering it either as RSS, RSS 2.0 or Atom feed.

You can find the source code of this example in the `chapter2/render-rss` directory.

Getting ready

As this recipe makes use of the ROME library to generate RSS feeds, you need to download ROME and its dependency JDOM first. You can use the Play dependency management feature again. Put this in your `conf/dependencies.yml`:

```
require:
    - play
    - net.java.dev.rome -> rome 1.0.0
```

Now as usual a test comes first:

```
public classFeedTest extends FunctionalTest {

    @Test
    public void testThatRss10Works() throws Exception {
        Response response = GET("/feed/posts.rss");
        assertIsOk(response);
        assertContentType("application/rss+xml", response);
        assertCharset("utf-8", response);
        SyndFeed feed = getFeed(response);
        assertEquals("rss_1.0", feed.getFeedType());
    }

    @Test
    public void testThatRss20Works() throws Exception {
        Response response = GET("/feed/posts.rss2");
        assertIsOk(response);
        assertContentType("application/rss+xml", response);
        assertCharset("utf-8", response);
        SyndFeed feed = getFeed(response);
        assertEquals("rss_2.0", feed.getFeedType());
    }

    @Test
    public void testThatAtomWorks() throws Exception {
        Response response = GET("/feed/posts.atom");
        assertIsOk(response);
        assertContentType("application/atom+xml", response);
        assertCharset("utf-8", response);
        SyndFeed feed = getFeed(response);
        assertEquals("atom_0.3", feed.getFeedType());
    }

    private SyndFeedgetFeed(Response response) throws Exception {
        SyndFeedInput input = new SyndFeedInput();
        InputSource s = new InputSource(IOUtils.toInputStream
                          (getContent(response)));
        return input.build(s);
    }
}
```

This test downloads three different kinds of feeds, `rss1`, `rss2`, and `atom` feeds, and checks the feed type for each. Usually you should check the content as well, but as most of it is made up of random chars at startup, it is dismissed here.

How to do it...

The first definition is an entity resembling a post:

```
@Entity
public class Post extends Model {

    public String author;
    public String title;
    public Date createdAt;
    public String content;

    public static List<Post>findLatest() {
        return Post.findLatest(20);
    }
```

```
        public static List<Post>findLatest(int limit) {
            return Post.find("order by createdAt DESC").fetch(limit);
        }
    }
```

A small job to create random posts on application startup, so that some RSS content can be rendered from application start:

```
    @OnApplicationStart
    public class LoadDataJob extends Job {

        // Create random posts
        public void doJob() {
            for (int i = 0 ; i < 100 ; i++) {
                Post post = new Post();
                post.author = "Alexander Reelsen";
                post.title = RandomStringUtils.
                 randomAlphabetic(RandomUtils.nextInt(50));
                post.content = RandomStringUtils.
                 randomAlphabetic(RandomUtils.nextInt(500));
                post.createdAt = new Date(new Date().getTime() +
                 RandomUtils.nextInt(Integer.MAX_VALUE));
                post.save();
            }
        }
    }
```

You should also add some metadata in the `conf/application.conf` file:

```
    rss.author=GuybrushThreepwood
    rss.title=My uber blog
    rss.description=A blog about very cool descriptions
```

The `routes` file needs some controllers for rendering the feeds:

```
    GET     /                  Application.index
    GET     /feed/posts.rss    Application.renderRss
    GET     /feed/posts.rss2   Application.renderRss2
    GET     /feed/posts.atom   Application.renderAtom
    GET     /post/{id}         Application.showPost
```

The Application controller source code looks like this:

```
    import static render.RssResult.*;

    public class Application extends Controller {

        public static void index() {
              List<Post> posts = Post.findLatest(100);
            render(posts);
        }

        public static void renderRss() {
              List<Post> posts = Post.findLatest();
            renderFeedRss(posts);
        }

        public static void renderRss2() {
              List<Post> posts = Post.findLatest();
            renderFeedRss2(posts);
        }

        public static void renderAtom() {
              List<Post> posts = Post.findLatest();
            renderFeedAtom(posts);
        }

        public static void showPost(Long id) {
              List<Post> posts = Post.find("byId", id).fetch();
              notFoundIfNull(posts);
              renderTemplate("Application/index.html", posts);
        }
    }
```

You should also adapt the `app/views/Application/index.html` template to show posts and to put the feed URLs in the header to make sure a browser shows the RSS logo on page loading:

```
#{extends 'main.html' /}
#{set title:'Home' /}
#{set 'moreHeaders' }
<link rel="alternate" type="application/rss+xml" title="RSS 1.0 Feed"
href="@@{Application.renderRss2()}" />
<link rel="alternate" type="application/rss+xml" title="RSS 2.0 Feed"
href="@@{Application.renderRss()}" />
<link rel="alternate" type="application/atom+xml" title="Atom Feed"
href="@@{Application.renderAtom()}" />
#{/set}

#{list posts, as:'post'}
<div>
<h1>#{a @Application.showPost(post.id)}${post.title}#{/a}</h1><br />
by ${post.author} at ${post.createdAt.format()}
<div>${post.content.raw()}</div>
</div>
#{/list}
```

You also have to change the default `app/views/main.html` template, from which all other templates inherit to include the `moreHeaders` variable:

```
<html>
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html;
          charset=utf-8">
        #{get 'moreHeaders' /}
        <link rel="shortcut icon" type="image/png" href="@{'/public/
images/favicon.png'}">
    </head>
    <body>
        #{doLayout /}
    </body>
</html>
```

The last part is the class implementing the different `renderFeed` methods. This is again a `Result` class:

```
public class RssResult extends Result {

    private List<Post> posts;
    private String format;

    public RssResult(String format, List<Post> posts) {
        this.posts = posts;
        this.format = format;
    }

    public static void renderFeedRss(List<Post> posts) {
        throw new RssResult("rss", posts);
    }

    public static void renderFeedRss2(List<Post> posts) {
        throw new RssResult("rss2", posts);
    }

    public static void renderFeedAtom(List<Post> posts) {
        throw new RssResult("atom", posts);
    }

    public void apply(Request request, Response response) {
        try {
            SyndFeed feed = new SyndFeedImpl();
            feed.setAuthor(Play.configuration.getProperty
               ("rss.author"));
            feed.setTitle(Play.configuration.getProperty
               ("rss.title"));
            feed.setDescription(Play.configuration.getProperty
               ("rss.description"));
            feed.setLink(getFeedLink());

            List<SyndEntry> entries = new ArrayList<SyndEntry>();
            for (Post post : posts) {
              String url = createUrl("Application.showPost", "id",
                                     post.id.toString());
              SyndEntry entry = createEntry(post.title, url,
                                     post.content, post.createdAt);
              entries.add(entry);
            }

            feed.setEntries(entries);

            feed.setFeedType(getFeedType());
            setContentType(response);

              SyndFeedOutput output = new SyndFeedOutput();
              String rss = output.outputString(feed);
              response.out.write(rss.getBytes("utf-8"));
        } catch (Exception e) {
              throw new UnexpectedException(e);
        }
    }

    private SyndEntrycreateEntry (String title, String link, String
description, Date createDate) {
        SyndEntry entry = new SyndEntryImpl();
        entry.setTitle(title);
        entry.setLink(link);
        entry.setPublishedDate(createDate);

        SyndContententryDescription = new SyndContentImpl();
        entryDescription.setType("text/html");
        entryDescription.setValue(description);

        entry.setDescription(entryDescription);

        return entry;
    }

    private void setContentType(Response response) {
        ...
    }
```

```
        private String getFeedType() {
            ...
        }

        private String getFeedLink(){
            ...
        }

        private String createUrl(String controller, String key, String
    value) {
            ...
        }
    }
```

How it works...

This example is somewhat long at the end. The post entity is a standard model entity with a helper method to find the latest posts. The `LoadDataJob` fills the in-memory database on startup with hundreds of random posts.

The `conf/routes` file features showing an index page where all posts are shown, as well as showing a specific post and of course showing all three different types of feeds.

The controller makes use of the declared `findLatest()` method in the post entity to get the most up-to-date entries. Furthermore the `showPost()` method also utilizes the `index.html` template so you do not need to create another template to view a single entry. All of the used `renderFeed` methods are defined in the `FeedResult` class.

The `index.html` template file features all three feeds in the header of the template. If you take a look at `app/views/main.html`, you might notice the inclusion of the `moreHeaders` variable in the header. Using the @@ reference to a controller in the template creates absolute URLs, which can be utilized by any browser.

The FeedResult class begins with a constructor and the three static methods used in the controller, which render RSS, RSS 2.0, or Atom feeds appropriately.

The main work is done in the `apply()` method of the class. A SyndFeed object is created and filled with meta information like blog name and author defined in the `application.conf` file.

After that a loop through all posts generates entries, each including the author, content, title, and the generated URL. After setting the content type and the feed type—RSS or atom—the data is written to the response output stream.

The helper methods have been left out to save some lines inside this example. The `setContentType()` method returns a specific content type, which is different for RSS and atom feeds. The `getFeedType()` method returns "rss_2.0", "rss_1.0", or "atom_0.3" depending on the feed to be returned. The `getFeedLink()` method returns an absolute URL for any of the three feed generating controller actions. The `createUrl()` method is a small helper to create an absolute URL with a parameter which in this case is an ID. This is needed to create absolute URLs for each post referenced in the feed.

The example also uses ROME to extract the feed data again in the test, which is not something you should do to ensure the correct creation of your feed. Either use another library, or, if you are proficient in checking corner cases by hand, do it manually.

There's more...

This is (as with most of the examples here) only the tip of the iceberg. Again, you could also create a template to achieve this, if you wanted to keep it simple. The official documentation lists some of the preparing steps to create your own templates ending with `.rss` at `http://www.playframework.org/documentation/1.2/routes#Customformats`

Using annotations to make your code more generic

This implementation is implementation specific. You could make it far more generic with the use of annotations at the Post entity:

```
@Entity
public class Post extends Model {

    @FeedAuthor
    public String author;
    @FeedTitle
    public String title;
    @FeedDate
    public Date createdAt;
    @FeedContent
    public String content;
}
```

Then you could change the render signature to the following:

```
    public RssResult(String format, List<? extends Object> data) {
        this.data = data;
        this.format = format;
    }

    public static void renderFeedRss(List<? extends Object> data) {
        throw new RssResult("rss", data);
    }
```

By using the generics API you could check for the annotations defined in the Post entity and get the content of each field.

Using ROME modules

ROME comes with a bunch of additional modules. It is pretty easy to add GeoRSS information or MediaWiki-specific RSS. This makes it pretty simple to extend features of your feeds.

Cache at the right place

Especially RSS URLs are frequently under heavy loads – one misconfigured client type can lead to some sort of denial of service attack on your system. So again a good advice is to cache here. You could cache the SyndFeed object or the rss string created out of it. However, since play 1.1 you could also cache the complete controller output. See *Basics of caching* at the beginning of *Chapter 4* for more information about that.