

Visionary: enter the Reactive World

Welcome. You are creating a new product that will reshape the world. But your project has to be perfect. No downtime is acceptable, user experience must be unparalleled. You want everything a system can be: resilient, scalable, interactive.

Rings a bell, doesn't it? Quite the common idealistic approach that makes so many engineers cringe. Idealists that want everything without knowing the limitations a system has. Asking for the impossible. Or do they?

As technology advances, things that were thought impossible as suddenly run-of-the-mill occurrences. Remember when the [C10k problem](#) seemed unsolvable? Fortunately for us, developers and visionaries, asking for resilient, scalable and interactive application is no longer a dream. In fact, as the [Reactive Manifesto](#) states, the very opposite is true.

In this series of articles we want to show one way of tackling this problem. We will leverage modern technologies and platforms, like [Play framework](#) and [Amazon AWS](#), to build a sample application that follows these principles: resilience, scalability and interactivity.

Through the series we will show you how to create the application, how to deploy it with minimal friction into Amazon AWS, and how to configure the platform so the application scales automatically thus providing the user with a smooth experience and avoiding downtime due to server errors.

We hope that this helps to inspire you when developing your next application. The sample we develop in here is a basic one, but it is a foundation stone you can use to grow your own vision. To that aim, we provide all the source and scripts via [Github](#), so you can adapt them as you need.

Let's start.

Steps to Building a Reactive Application

This article takes you through most of the steps required to build a reactive application using Play Framework and Amazon AWS. All the explanations relate to the sample code provided in the [Github](#) repository, any details not discussed in the article may be understood by checking the code.

There are several reasons why you should choose the technologies we use here for any of your projects:

- [Amazon AWS](#) provides an excellent platform to manage your applications. It reduces the amount of time you have to spend on sysadmin tasks and turns scaling the application into something simple.
- [Play framework](#) is a modern Java framework that unites the productivity of Ruby on Rails with the reliability and performance of the Java Virtual Machine.
- [Ansible](#) is an excellent tool to manage your servers with minimal effort, using plain text files to set them up and providing all the capabilities you have come to expect from a DevOps tool.

We hope that through this example you get to know these tools and get a glimpse on how they may benefit your present and future deployments.

Before starting with the example itself, let's do a small exercise of imagination. You want to start coding your application, and to that aim you sketch a project plan with some generic steps to follow:

1. Setup a development environment, including Amazon AWS services you may use

2. Develop the code
3. Setup a staging area (development) in Amazon AWS
4. Setup a test environment in Amazon AWS
5. Setup a production environment in Amazon AWS
6. Launch the application
7. Scale it as the demand grows
8. Profit!

In the article, we focus mainly in points 1, 2, 3 and 7. Be aware that the instructions for point 3 can easily be replicated for point 4 and 5, so in fact we can say we are also covering these points. We would like to explain you the secret behind point 8, but we have been forbidden to do that.

This first chapter focusses mostly in developing the code (point 2) and a partial setup of the development environment (point 1). We start by [describing the technologies](#) involved in the application and how they fit each other. We continue by giving step by step instructions on how to [construct the application](#), and then we do a [test run of the deployed application](#). Finally, we talk about [configuring the application for other environments](#).

Although the source code we create uses DynamoDB, and you need to create some tables in it to be able to run the application, we have decided to group all the Amazon AWS configuration in the next chapter so we can focus a bit more on the Play side of the project in the current one.

Building the sample Play application

The first step is to create our Play Framework application. The purpose is to create a sample with basic functionality that can be extended later as required. To that aim, we provide a skeleton that covers common needs like authentication or connection to multiple databases.

The application consists on a page that displays the most popular tags in [Flickr](#) and, once a tag is selected, a list of photos with that tag is shown to the user. Access to the page that displays the images is secured, requiring authentication by the user. Photos' details are loaded via a background task and stored in [DynamoDB](#), while user data is stored in a MySQL database provided by [Amazon RDS](#).

Description of Technologies

This section summarizes the most relevant components used to build the application. The purpose is to familiarize ourselves with the different elements we can find when browsing the code or when extending the application for our own purposes.

Play Framework

[Play Framework](#) is a modern framework to build highly-scalable applications using Java (as in our scenario) or Scala. One of the main benefits of Play is its productivity: the framework provides many helpers and embedded APIs that simplify common tasks like doing requests to other web sites or validating user submitted data. This translates into less time spent in writing boilerplate and more time spent creating the logic of your application.

Play is a stateless framework, which makes it a perfect choice for cloud environments like Amazon AWS. Being stateless means that any Play deployment lacks the concept of a Session object like the one a traditional JEE container provides. Instead, Play provides a cryptographically secured cookie as a limited and temporary storage for data. When a request hits a server, the application receives the cookie and can retrieve the authentication status of the user, and any other stored data, from it.

The overhead of this step is negligible when using caches and it simplifies application deployment in cloud environments, as in the cloud a server could fail at any time and its session objects would be gone.

Akka

[Akka](#) is a toolkit to build fault tolerant applications in the JVM, using either Java or Scala. Play Framework is built on top of Akka, which means that when you create a Play application you get access to all the Akka functionality, including Actor systems and remoting.

A benefit of this integration is the creation of different execution contexts. An execution context can be considered, in a very simplified definition, as a thread pool reserved to run a task, usually an actor. These threads will only be used by that actor, which allows you to isolate slow tasks in its own thread pool and to avoid any impact on the performance of other parts of the application.

In this application we use Akka as a cron job manager. Akka provides a scheduler from which you can send a message to an actor at a given time. As this task is completely asynchronous, this allows you to run heavy lifting tasks (in this scenario, querying the Flickr API to obtain data on a few hundred images) without any performance impact in the processing of Http requests, as they run in different execution contexts.

WebJars

[WebJars](#) are client side libraries, like [Jquery](#), packaged as Jars. This has two big benefits. First, we can manage the dependencies in the client side libraries easily, avoiding clashes between versions and removing issues caused by transitive dependencies. Second, we can manage these dependencies via standard tools like Ivy, Maven or SBT, as we are working with Jars.

Ebean

[Ebean](#) is a JPA compliant ORM, used by Play Framework as the default ORM. Being JPA compliant, in the application we define our database model as simple POJOs with annotations and the system will generate the necessary queries to retrieve the data from the database.

Ebean provides advanced capabilities, like automatic ddl generation based on the application model, but we are not using them in this project.

Bootstrap

[Bootstrap](#) has become the de-facto front-end framework for web development. Bootstrap provides a cross-browser set of tools, including a responsive layout, default styles and useful Javascript libraries. This allows developers to produce good looking web applications with minimal effort and without having to be Css experts.

In this application we load Bootstrap via WebJars.

Play Authenticate

[Play Authenticate](#) is one of several authentication plugins for Play Framework. For anyone familiar with Play Framework, it is similar to [SecureSocial](#) but it is tailored to work only with Java. There is a sample application for Play Authenticate which we use as a the starting point for our project.

Play Authenticate provides both authentication and authorization. The authentication system allows a user to register with a username and password or to provide credentials via well known 3rd party systems like Twitter or Facebook, using the OAuth protocol. The authorization system is provided by [Deadbolt 2](#).

In the application we use this component to limit access to the page that contains the images. Only authenticated users are allowed to see that area of the website.

Amazon AWS: RDS MySQL

[Amazon RDS](#) provides on demand relational databases. The service has high reliability and manages backups and other sysadmin tasks, freeing you from having to spend time in them.

In this application we use MySQL to store all user-related data as required by Play Authenticate. There is a *models* folder in the application that contains all the POJOs annotated with Ebean.

Amazon AWS: DynamoDB

[Amazon DynamoDB](#) provides a fast and scalable NoSQL database, similar to [Riak](#). It enforces some limitations, like maximum size of stored elements, but in exchange you get a very fast service that can evolve along your data, a benefit of not having a fixed schema in the database. The service manages common tasks like maintenance of backups automatically, letting you focus on development.

In the application we use DynamoDB to store metadata about tags and images obtained from Flickr. The data is all text (String) and includes information like the name of a tag or the url of an image. By using DynamoDB we can later on extend the model as required and with minimal effort, adding new fields like how many users vote for a particular image or how many times has a particular tag become popular.

Constructing the Application

In this section we describe how to build the application itself. Instead of describing every single step in detail, we focus on the main areas of the website and what is needed to build them. Please check the source code in [Github](#) to see the full implementation.

Install Play

To install Play Framework simply follow the [instructions](#) in the website. You want to download the latest release (at the time of this writing, Play 2.1.1) and unzip the file in a folder of your choice. Then add the folder to your *PATH* as per your system specification.

Once this is done, run the command:

\$play

in a terminal window. You should get an output like this:

```
pvillega@pvillega-Inspiron-530: ~  
pvillega@pvillega-Inspiron-530:~$ play  
  
play! 2.1.1 (using Java 1.7.0_21 and Scala 2.10.0), http://www.playframework.org  
  
This is not a play application!  
  
Use 'play new' to create a new Play application in the current directory,  
or go to an existing application and launch the development console using 'play'  
.  
  
You can also browse the complete documentation at http://www.playframework.org.  
pvillega@pvillega-Inspiron-530:~$
```

It shows that Play 2.1.1 has been installed and, in red, that the current folder is not a valid Play application.

If you are going to work with several Play projects, I advise you to check [Play Version Manager](#), a tool that allows you to swap between different Play Framework versions in one single machine. The steps to install it are outside the scope of this document, but it's simple enough and a good choice if you will work a lot with Play Framework.

Create the Application

Once we have installed Play, we can create the application. Run the command:

```
$play new play-sample-app
```

and Play prompts you to select a name for the application and to select if you want a simple Java or Scala application. Select a simple Java application, and the basic layout will be created. The output should be similar to the one in the following image:


```
pvillega@pvillega-Inspiron-530: ~/play-sample-app
pvillega@pvillega-Inspiron-530:~/play-sample-app$ play run
[info] Loading global plugins from /home/pvillega/.sbt/plugins
[info] Loading project definition from /home/pvillega/play-sample-app/project
[info] Set current project to play-sample-app (in build file:/home/pvillega/play-sample-app/)

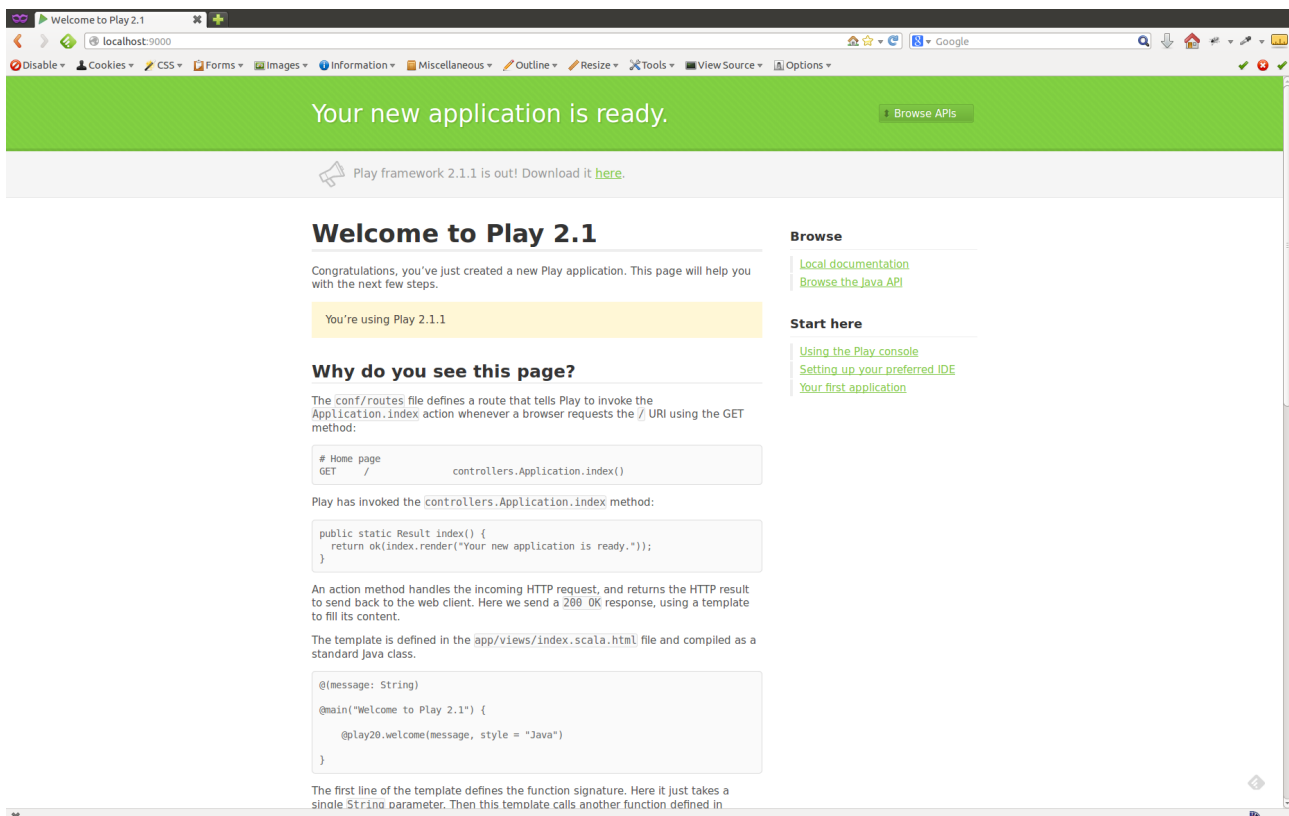
[info] Updating {file:/home/pvillega/play-sample-app/}play-sample-app...
[info] Done updating.
--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)

[info] Compiling 4 Scala sources and 2 Java sources to /home/pvillega/play-sample-app/target/scala-2.10/classes...
[info] play - Application started (Dev)
```

Accessing that address displays the default application:



Add dependencies

Now that we have a running project, the next step is to add the external dependencies of our application. As mentioned above, we are using WebJars, Amazon AWS, MySQL and other components. For Play to recognize them, we have to define these dependencies in SBT, so the

corresponding jars can be retrieved.

The file that defines the dependencies is located at *project/Build.scala*. Dependencies are usually retrieved from standard repositories like [Maven Central](#), but in some cases we want to pull them from other locations as the files are not stored in Maven Central. This means that we have to define both *dependencies* and *resolvers* in this file.

Dependencies indicate a jar we want to add to our application classpath and are defined as a sequence in the file. Dependencies follow the format:

```
"<group>" % "<artifact>" % "<version>"
```

For example, the dependency for Webjars is defined as:

```
"org.webjars" %% "webjars-play" % "2.1.0-2"
```

While the dependency for MySQL is defined as:

```
"mysql" % "mysql-connector-java" % "5.1.25"
```

As we mentioned above, by using WebJars we can import Javascript libraries via our dependencies. In our application we import Bootstrap this way. Bootstrap has a dependency in JQuery, which means that by including Bootstrap we have access to both Bootstrap and JQuery in our project. The dependency on Bootstrap is defined as:

```
"org.webjars" % "bootstrap" % "2.3.2"
```

Play provides some keywords to define common dependencies for Play projects, that way instead of adding the full structure we can use the keyword to define that dependency. For example, to import Ebean we can write:

```
javaEbean
```

and Play will resolve that to the correct dependency when compiling.

Resolvers are location where we can find jars that can be used as dependencies. A resolver has two parts, a name and a url. The name is a human readable name used for developers to identify the repository, while the url is the location where files not available in Maven Central may be located.

An example of a resolver is:

```
resolvers += Resolver.url("play-easymail (release)", url("http://joscha.github.com"))  
(Resolver.ivyStylePatterns)
```

This resolver contains jars for *play-easymail* which are stored in a Github repository. When compiling, SBT will check this repository for files it has not been able to locate in other locations. If SBT can't locate a dependency in any of the defined resolvers nor in the default ones (Maven Central) compilation will fail.

The final version of the file, including all the dependencies, will look like this:

```
1 object ApplicationBuild extends Build {  
2  
3   val appName          = "play-sample-app"  
4   val appVersion       = "1.0-SNAPSHOT"  
5  
6   val appDependencies = Seq(  
7     javaCore,  
8     javaJdbc,  
9     javaEbean,  
10    "org.webjars" %% "webjars-play" % "2.1.0-2",  
11    "org.webjars" % "bootstrap" % "2.3.2",
```



```

12     "com.amazonaws" % "aws-java-sdk" % "1.4.5",
13     "com.feth" %% "play-authenticate" % "0.2.5-SNAPSHOT",
14     "be.objectify" %% "deadbolt-java" % "2.1-SNAPSHOT",
15     "mysql" % "mysql-connector-java" % "5.1.25"
16 )
17
18 val main = play.Project(appName, appVersion, appDependencies).settings(
19     resolvers += Resolver.url("Objectify Play Repository (release)", url("http://scha
20     resolvers += Resolver.url("Objectify Play Repository (snapshot)", url("http://scha
21     resolvers += Resolver.url("play-easymail (release)", url("http://joscha.github.com
22     resolvers += Resolver.url("play-easymail (snapshot)", url("http://joscha.github.com
23     resolvers += Resolver.url("play-authenticate (release)", url("http://joscha.github
24     resolvers += Resolver.url("play-authenticate (snapshot)", url("http://joscha.github
25 )
26
27 }

```

Add Play Authenticate

Once we have the dependencies defined, the next step is to integrate Play Authenticate. Play Authenticate comes with a sample application that includes a Bootstrap layout and basic functionality required in any working application, like resending registration emails to users or merging different external accounts for a given user.

We use the Play Authenticate sample application as a basis for our layout. There are several files copied over, including images, views and controllers. Only the most relevant elements are discussed in this section, the remainder can be checked in the source code we provide.

Configuration

The first component we talk about is the configuration. Configuration files for Play Authenticate are located in two folders, *conf* and *conf/play-authenticate*.

The folder *conf/play-authenticate* contains configuration specific to components of Play Authenticate. For example, the file *deadbolt.conf* defines the handler for Authorization in the application, while *smtp.conf* includes all the SMTP mailer settings that allow the website to send emails to the user. By default we are using a mock SMTP server.

The file *mine.conf* has the settings required to authenticate via external services like Twitter, using OAuth, including consumer and secret keys. The application sample you can find in Github already provides keys for some of these services, so you can test this integration without having to register a new application.

The folder *conf* contains two files relevant to Play Authenticate. The first one is *play.plugins*, which defines a list of plugins to be loaded by the application on startup. As you can see in the following listing of the contents:

```

1500:com.typesafe.plugin.CommonsMailerPlugin
10000:be.objectify.deadbolt.java.DeadboltPlugin
10005:service.MyUserServicePlugin
10010:com.feth.play.module.pa.providers.oauth2.google.GoogleAuthProvider

```

```
10020:com.feth.play.module.pa.providers.oauth2.facebook.FacebookAuthProvider
10040:providers.MyUsernamePasswordAuthProvider
10060:com.feth.play.module.pa.providers.oauth1.twitter.TwitterAuthProvider
```

We define which Play authenticate modules we want to use. As an example, removing the `TwitterAuthProvider` entry from the file and restarting the server would disable authentication via Twitter OAuth service.

The second relevant file is *messages.en* which has the I18N keys for the application. In this website we only provide the file for English, but you could easily create a *messages.es* file and provide a Spanish translation for the website, including all the Play Authenticate screens.

Model

The folder *app/models* contains Ebean entities used by Play Authenticate to store data on registered users. The data will be stored in a relational database, in our website inside a MySQL database.

An example of the files included is *UserPermission*, a class that stores the access levels available in the system:

```
1 @Entity
2 public class UserPermission extends Model implements Permission {
3     /**
4      *
5      */
6     private static final long serialVersionUID = 1L;
7
8     @Id
9     public Long id;
10
11     public String value;
12
13     public static final Model.Finder<Long, UserPermission> find = new Model.Finder<
14         Long.class, UserPermission.class>;
15
16     public String getValue() {
17         return value;
18     }
19
20     public static UserPermission findByValue(String value) {
21         return find.where().eq("value", value).findUnique();
22     }
23 }
```

As you can see in lines 1 and 8, this class follows standard JPA annotations (Ebean is JPA compliant) and it also includes some support methods to search for specific instances of the class in the database.

The default model classes included shouldn't be extended, as they are used by Play Authenticate and any modification may cause issues in the authentication and authorization system. But feel free

to use them as templates for your own model classes.

Controllers

Play Authenticate provides some controllers in the sample application. The most relevant is *app/controllers/Application.java*, which manages the main urls of the webapp. Remember that in a Play Framework application the mapping between url and controller is defined inside *conf/routes*.

The Application controller is a good example on how to secure methods. An extract of its contents follows:

```
1 public class Application extends Controller {
2
3     public static final String FLASH_MESSAGE_KEY = "message";
4     public static final String FLASH_ERROR_KEY = "error";
5     public static final String USER_ROLE = "user";
6
7     public static Result index() {
8         return ok(index.render());
9     }
10
11    public static User getLocalUser(final Session session) {
12        final AuthUser currentUser = PlayAuthenticate.getUser(session);
13        final User localUser = User.findByAuthUserIdentity(currentUser);
14        return localUser;
15    }
16
17    @Restrict(@Group(Application.USER_ROLE))
18    public static Result profile() {
19        final User localUser = getLocalUser(session());
20        return ok(profile.render(localUser));
21    }
22
23    public static Result login() {
24        return ok(login.render(MyUsernamePasswordAuthProvider.LOGIN_FORM));
25    }
26 }
```

If you check line 17, you will see that we are restricting access to the *profile* method so only authenticated users can access it. The other methods are not restricted, so any user could potentially access them and obtain a response.

WebJars

One change over the default layout of Play Authenticate is how we include the Bootstrap files. As mentioned several times, we use WebJars to add Bootstrap in the application. After the dependency was defined in the previous sections, we have to load it.

This is achieved by editing the file *app/views/main/scala/html* and adding the following lines to load the relevant Css and Javascript:

```
@* Retrieve css/Javascript dependencies via WebJars *@
```

```
<link rel='stylesheet'
href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.css"))'>
<script type='text/javascript'
src='@routes.WebJarAssets.at(WebJarAssets.locate("jquery.min.js"))'></script>
<script type='text/javascript'
src='@routes.WebJarAssets.at(WebJarAssets.locate("js/bootstrap.min.js"))'></scr
ipt>
```

WebJars provide a helper class, *WebJarsAssets*, which locates the dependencies and provides the proper path to load them.

Note that we are including also the Javascript for JQuery. As mentioned before, Bootstrap has a dependency on JQuery, which means that we can also use it in our application.

Connect to Amazon AWS

Now that we have the layout and the authentication component integrated, it is time to use services from Amazon AWS.

Configuration

To be able to connect to many AWS services from a Java application you need the [Amazon AWS SDK](#) jar. In the dependencies section we added that jar via the following dependency:

```
"com.amazonaws" % "aws-java-sdk" % "1.4.5"
```

The SDK provides methods to connect to S3, DynamoDB and SQS among other Amazon AWS services. But to do so, it requires your Amazon access key and secret key as well as the endpoint for the services. You can obtain your access and secret keys from your [Amazon AWS Management Console](#).

The [best practice](#) regarding your keys is to use AWS Identity and Access Management (IAM) roles to create limited roles with access to only a component, for example a role that can only read from S3. This way, even if your keys are leaked, you limit the amount of data an attacker may retrieve. The specific steps on securing the users are outside the scope of this chapter, but the [documentation](#) for IAM is very detailed, and it shows the steps you need to follow to create the roles you need.

In our sample application the Amazon AWS details are stored under the file at *conf/aws/credentials/conf*. This file is a properties file, loaded via *conf/application.conf*, which stores the access and secret keys. It also stores the endpoint for DynamoDB, as each Amazon region is independent and accessing the wrong one means that the tables won't exist. The default region in the sample application is set to connect to DynamoDB in Europe (*dynamodb.eu-west-1.amazonaws.com*).

The configuration file also provides database configuration for our Amazon RDS connection details. As Amazon RDS gives access to the endpoint of the machine, you can easily generate a jdbc url to connect to it.

MySQL Model

At this point we have the jars and the configuration. Now we can define the database model. In this sample application we use MySQL to store the user data generated by Play Authenticate. As mentioned in previous sections, we use Ebean as ORM to access the data from the tables, via model classes defined at *app/models*. This means that we need no extra helper classes to save or load the data.

Having the model defined in our code, we need to create the corresponding tables in the database. Play Framework provides the [Evolutions](#) system to keep the database schema up to date automatically. Evolutions uses SQL files stored at `conf/evolutions/default/` which contain SQL commands that are run against the database. These commands can create or alter tables, insert or remove rows, or do any other valid operation.

When the application is launched the system checks if any Evolution script needs to be applied. If required, the relevant commands are executed and the database schema is updated so the application can run as expected.

What this means for us is that, after setting the connection details in our configuration files, we don't have to worry about configuring MySQL tables as Play Framework takes care of that using the evolution scripts we provide.

DynamoDB

As mentioned before, in the sample application we use DynamoDB to store metadata of Flickr images. The steps to configure DynamoDB in Amazon AWS are shown in the next article, where we configure all the services we need in Amazon AWS. In this section we show the parts relevant to our Java application.

The first thing we need to do is to define the model we use. DynamoDB has no schema, you just define tables that may store any kind of data, with some limitations imposed by the system. To facilitate the interaction between our Java code and this NoSQL database, Amazon AWS SDK provides some data modelling tools that map the data stored in a table to a POJO. One example of such mapping is the following file:

```
1 @DynamoDBTable(tableName = "tag")
2 public class Tag {
3
4     private String id;
5     private String name;
6
7     @DynamoDBHashKey
8     @DynamoDBAutoGeneratedKey
9     public String getId() { return id; }
10    public void setId(String id) { this.id = id; }
11
12    @DynamoDBAttribute
13    public String getName() { return name; }
14    public void setName(String name) { this.name = name; }
15
16 }
```

As you can see in lines 1, 7, 8 and 12 we use annotations to map between this POJO and the contents in DynamoDB. The annotations define the table name where the data will be stored (line 1), the primary Hash Key and its generation method (lines 7 and 8) and which fields are to be stored as simple attributes in the table (line 12).

We have two of such model classes, stored under `app/models/dynamo`, one to map tags (*Tag.java*) and one to map photos (*Photo.java*). They store metadata for the tags and images retrieved from Flickr. The relation between the tables is established via the id (HashKey) of Tag. A photo related to a tag has its id (HashKey) set to the id of the Tag to which it belongs. The primary key of the image is complemented by a RangeKey on the date the metadata of the photo is stored in the database, thus providing a valid composite key for the row.

Retrieving and saving the data from DynamoDB requires the use of a *DynamoDBMapper* class that manages the save and load operations. To simplify calls to these operations, a service class has been added to *app/service/DynamoDBService* which provides methods to load and save data to DynamoDB. The service class is a singleton implemented as an Enum, and a partial view of its implementation follows:

```
1 public enum DynamoDbService {
2     INSTANCE;
3
4     // Load AWS keys from configuration
5     private String accessKey = Play.application().configuration().getString("aws.accessKey");
6     private String secretKey = Play.application().configuration().getString("aws.secretKey");
7     private String endpoint = Play.application().configuration().getString("aws.endpoint");
8
9     // Set up connection to Dynamo DB
10    private AWSCredentials awsCredentials;
11    private AmazonDynamoDB dynamo;
12    private DynamoDBMapper mapper;
13
14    DynamoDbService() {
15        awsCredentials = new BasicAWSCredentials(accessKey, secretKey);
16        dynamo = new AmazonDynamoDBClient(awsCredentials);
17        dynamo.setEndpoint(endpoint);
18        mapper = new DynamoDBMapper(dynamo);
19    }
20
21    public List<Tag> getAllTags() {
22        final DynamoDBScanExpression scanExpression = new DynamoDBScanExpression();
23        final List<Tag> tags = mapper.scan(Tag.class, scanExpression);
24        return tags;
25    }
26
27    public void saveTags(List<Tag> tags) {
28        mapper.batchSave(tags);
29    }
30 }
```

Using an Enum as the implementation of a Singleton is currently considered [the best way](#) to implement Singletons. As you can see on line 14, the constructor of the Enum initializes the connection to DynamoDB, using the Amazon AWS credentials (access key, secret key and endpoint).

Lines 21 and 27 show operations that load and save tags using the SDK. The use of this service allows us to decouple the implementation details of connecting to DynamoDB from the controllers that will use the service, so we can modify the storage implementation with no impact on the rest of the application.

Collect data from Flickr

At this stage we have created most of the logic required by our application. Now we have to implement the retrieval of data from Flickr via its [API](#). To do that, we take advantage of the integration between Play Framework and Akka. We query the API and store the relevant metadata

in DynamoDB inside an Akka actor, and we set a scheduler that sends a message to that actor every 24h. That way we have the top tags of each day stored in our system.

Actor

The [Akka actor](#) in our sample application is defined at *app/actors/RetrieveTagsAndPhotos.java*. It is an Untyped Actor that will get images from Flickr using its API when it receives a message. The actor ignores the type of message received, any message will trigger the collection of images.

The implementation is as follows:

```
1 public class RetrieveTagsAndPhotos extends UntypedActor {
2
3     final String key = Play.application().configuration().getString("flickr.key");
4     final String secret = Play.application().configuration().getString("flickr.secret");
5     final String restUrl = Play.application().configuration().getString("flickr.url");
6
7     /**
8      * When receiving the message it connects to Flickr to retrieve a list of tags and
9      * Only stores new tags
10     * @param message received message (ignored)
11     */
12     @Override
13     public void onReceive(Object message) {
14         Logger.info("Retrieving data in the background");
15         final DynamoDbService db = DynamoDbService.INSTANCE;
16
17         Logger.info("Obtain a list of tags");
18         final F.Promise<play.libs.WS.Response> resp = play.libs.WS.url(restUrl)
19             .setQueryParameter("method", "flickr.tags.getHotList")
20             .setQueryParameter("api_key", key)
21             .get();
22
23         final play.libs.WS.Response response = resp.get();
24         final List<Tag> tags = Tag.convertToTagList(response.asXml());
25
26         Logger.info("Save tags into dynamo db");
27         db.saveTags(tags);
28
29         Logger.info("Obtain a list of photos for each tag");
30         List<Photo> photos;
31         for(Tag tag: tags) {
32             Logger.debug("Photos for tag: " + tag.getName());
33             final F.Promise<play.libs.WS.Response> respImg = play.libs.WS.url(restUrl)
34                 .setQueryParameter("method", "flickr.photos.search")
35                 .setQueryParameter("api_key", key)
36                 .setQueryParameter("tags", tag.getName())
37                 .get();
38
39             final play.libs.WS.Response responseImg = respImg.get();
40             photos = Photo.convertToPhotoList(responseImg.asXml(), tag.getId());
```



```

41
42         Logger.info("Save photos into dynamo db");
43         db.savePhotos(photos);
44     }
45
46     Logger.info("All data retrieved and stored in dynamo db");
47 }
48 }

```

The only method in the actor is defined at line 13, *onReceive*. This method is executed when the actor receives a message. In the implementation, as stated above, we ignore the message itself.

The code connects to the Flickr API as we can see in lines 18 and 33, using the *WS* object provided by Play. *WS* facilitates creating requests to third party applications, providing helper methods to build the request and even sign it if necessary. Fortunately, Flickr provides some API methods which don't require authentication, and that simplifies the code necessary to obtain the images.

You may notice that the calls return a *Promise* that wraps the *WS.Response*. This means that this request is asynchronous; that is, when we execute the *get* call in *WS* we don't block the thread running the code. Instead, Play processes the request to the target url in the background and we proceed with the rest of the code in the meantime. Only when we try to access the value contained in the *Promise* will Play block the thread if the response has not been obtained in the background before getting to that fragment of code.

To save the returned xml into DynamoDB we use the *DynamoDBService* defined in the previous section, as well as some helper methods in our model POJOs that map the returned xml to our implementation.

The sample application provided already contains valid Flickr API keys, so you can retrieve images without having to register a new client in Flickr. You can find the keys in *conf/application.conf*, at the bottom of the file.

Scheduler

To set the scheduler, we create a private method in our *app/Global.java* class:

```

1    /*
2    Schedule Actors to retrieve data in the background, asynchronously
3    */
4    private void scheduleJobs() {
5        ActorSystem actorSystem = Akka.system();
6
7        ActorRef retrieveTopicsActor = actorSystem.actorOf(new Props(RetrieveTagsAndPh
8        // execute every day to get new images
9        actorSystem.scheduler().schedule(Duration.create(0, TimeUnit.MILLISECONDS), Du
10    }

```

This private method will be called from the *onStart* method in *Global*, and it will trigger a message to the defined actor when the application loads for the first time. After that first execution, it will schedule the next message to the actor, to be sent after 24h in a recurring manner. This means that the message will be sent once per day when our sample application is running.

A comment on the current implementation. The scheduler as it is implemented runs once per instance, which means that if we have several instances running our application we will be running the retrieval of data once per instance per day. In this sample application this is not an issue, but in

some scenarios you may want to ensure that the job is done only once. Akka provides solutions for this, although they are outside the scope of this article. Check Akka documentation for details on [possible solutions](#) to this scenario.

Display Images

Now that we have the images stored in the database, we want to show them. We add a method to our controller in *Application.java*:

```
1 @Restrict(@Group(Application.USER_ROLE))
2 public static Result seePhotos() {
3     DynamoDbService db = DynamoDbService.INSTANCE;
4     List<Tag> tags = db.getAllTags();
5     List<Photo> photos = db.getPhotosForTag(tags.get(0).getId());
6     return ok(seePhotos.render(tags, photos));
7 }
```

As you can see at line 1, this method is only accessible for authenticated users. The method uses the *DynamoDBService* to load a list of tags and photos to be shown to the user. The template iterates over the arrays and shows a list of tags and the photos for the first tag.

In the page itself we may want to see the images of another tag. Instead of loading all the images at once, as it would be slow, the template also contains an Ajax request that calls another method:

```
1 @Restrict(@Group(Application.USER_ROLE))
2 public static Result getImagesForTag() {
3     String tagId = request().getQueryString("tag");
4     Logger.info(String.format("Requested images for tag [%s]", tagId));
5     DynamoDbService db = DynamoDbService.INSTANCE;
6     List<Photo> photos = db.getPhotosForTag(tagId);
7
8     ObjectNode json = Json.newObject();
9     ArrayNode array = json.putArray("images");
10    for(Photo p: photos) {
11        ObjectNode node = array.addObject();
12        node.put("image", p.getImageUrl());
13        node.put("thumbnail", p.getThumbnailUrl());
14        node.put("text", p.getText());
15    }
16    return ok(json);
17 }
```

This method is also secured, as shown at line 1, and receives a tag name in the request (line 3). The method loads the images for the tags and returns a Json object (lines 8 to 16) with the metadata of the photos. The Json object is processed by the client to display the images requested by the user.

Try the Application

Now that our sample code has been completed, we can execute the application in our development environment to see it in action. We open a terminal window, move to the folder where we have the application and run:

```
$play run
```

And we should see something like follows:

```
pvillega@pvillega-Inspiron-530: ~/Dropbox/Projectes/amazon-article/play-sample-app
pvillega@pvillega-Inspiron-530:~/Dropbox/Projectes/amazon-article/play-sample-app$ play run
[info] Loading global plugins from /home/pvillega/.sbt/plugins
[info] Loading project definition from /home/pvillega/Dropbox/Projectes/amazon-article/play-sample-app/project
[info] Set current project to play-sample-app (in build file:/home/pvillega/Dropbox/Projectes/amazon-article/play-sample-app/)

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)

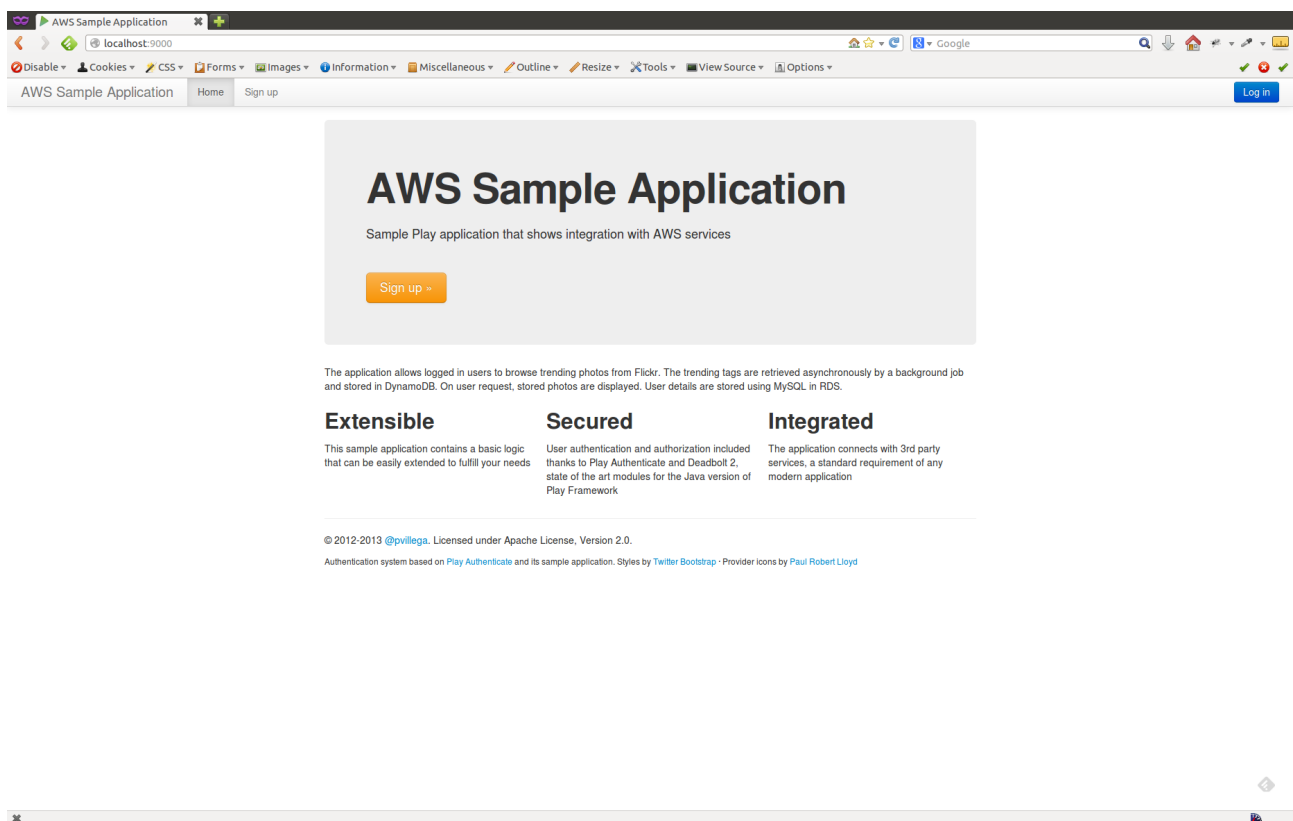
21:37:09.586 [New I/O worker #1] [I][play] - database [default] connected at
jdbc:mysql://localhost/awsService

21:37:16.455 [New I/O worker #1] [I][play] - Starting application default Akka
a system.

21:37:16.524 [New I/O worker #1] [I][play] - Application started (Dev)

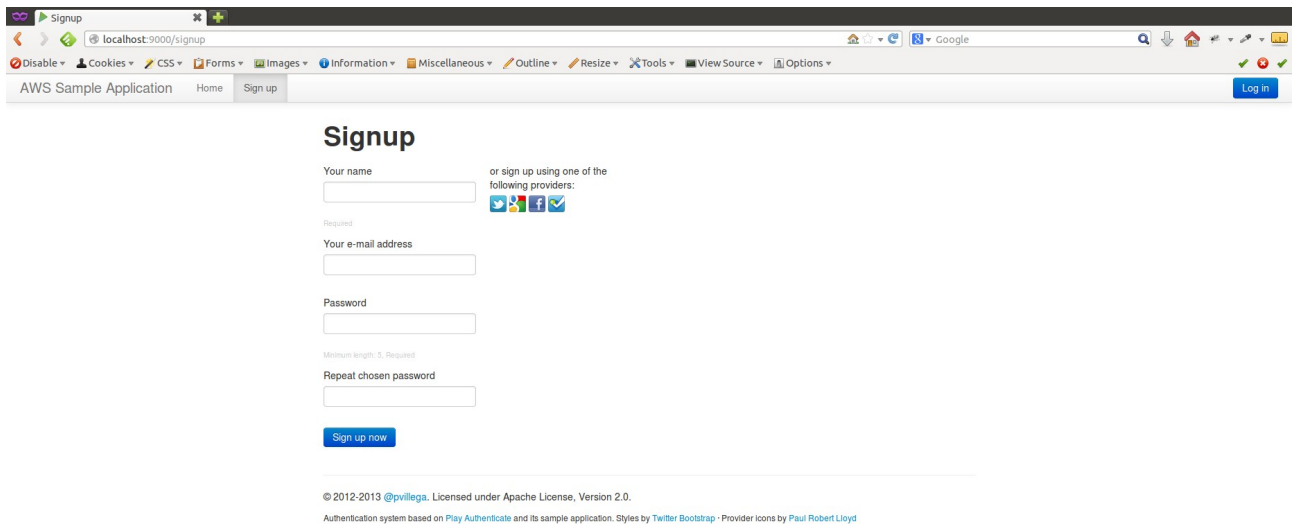
21:37:16.665 [application-akka.actor.default-dispatcher-3] [I][application] - Retrieving data in the background
```

On start, the logs show us the application retrieving images from Flickr. You can verify this is an asynchronous task by loading the main page while the external data is being loaded. Go to <http://localhost:9000> to see the main page of the application:



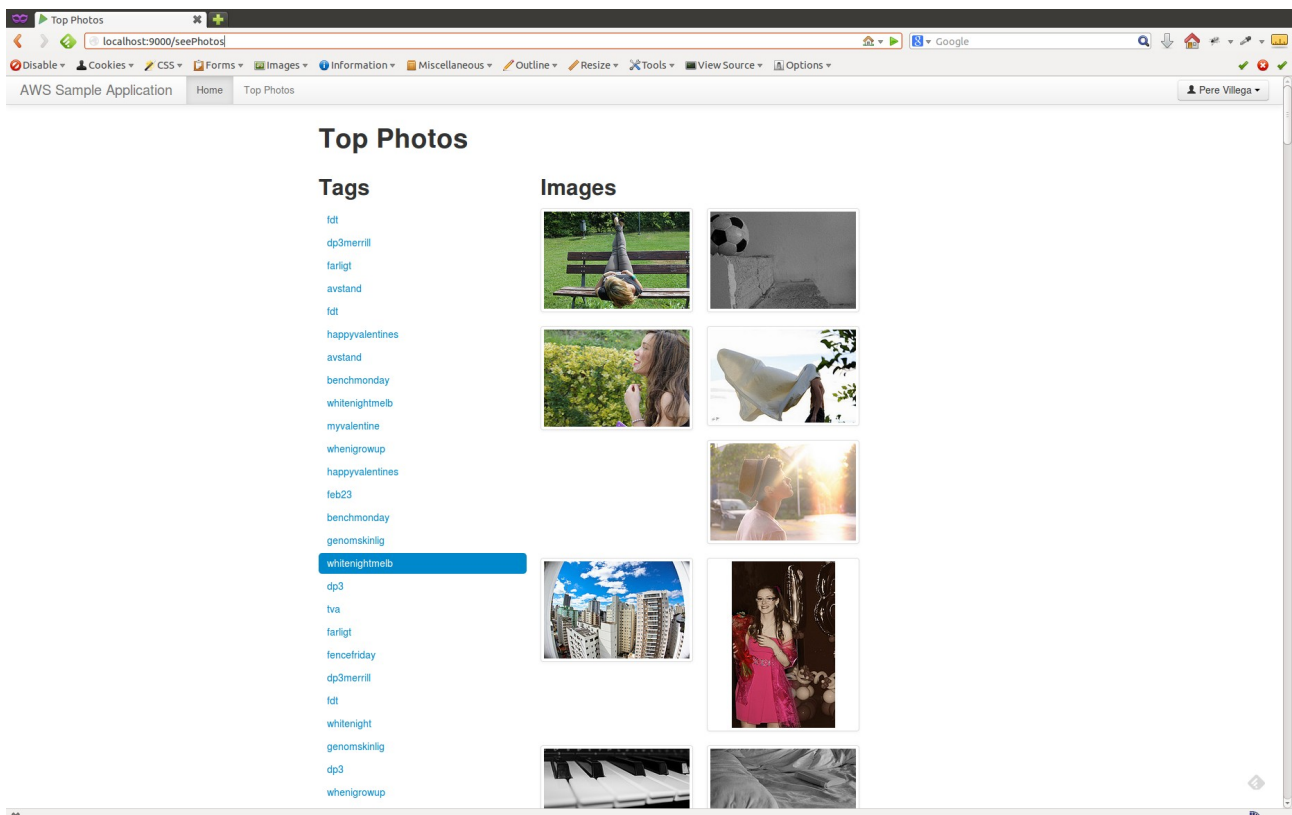
As you can see you can only try to Sign Up before being able to see any image, thanks to the authorization requirements we added when creating the application. Press one of the buttons to be

redirected to the Sign up page and complete the process:



The screenshot shows a web browser window with the address bar at `localhost:9000/signup`. The page has a navigation bar with "AWS Sample Application", "Home", and "Sign up" (highlighted). A "Log in" button is in the top right. The main content area is titled "Signup" and contains a form with the following fields: "Your name" (text input), "Your e-mail address" (text input, marked "Required"), "Password" (text input, marked "Minimum length: 5. Required"), and "Repeat chosen password" (text input). To the right of the "Your name" field, there is a link "or sign up using one of the following providers:" with icons for Twitter, Facebook, and Google+. At the bottom of the form is a blue "Sign up now" button. Below the form, there is a copyright notice: "© 2012-2013 @pvillega. Licensed under Apache License, Version 2.0." and a note: "Authentication system based on Play Authenticate and its sample application. Styles by Twitter Bootstrap - Provider icons by Paul Robert Lloyd".

Once authenticated you are able to see the photos page, where you have a list of tags and, once a tag is selected, a list of images associated to that tag:



Configuration for Other Environments

Up to now we have been talking about configuration in a development environment. But we want to deploy this application in test or production servers, where the settings will be different: changes to the database connection details, OAuth credentials and more.

Play Framework uses an [advanced configuration system](#). One of the advantages of this system is that you can define configuration values as JVM system properties. What is more, these system properties *override* any settings in the *conf* files.

What this means for us is that we can define our development environment in our configuration files. Once we want to deploy in another environment, we just need to provide the appropriate JVM parameters to Play. For example, to override our Amazon AWS access key we can add this system property:

```
-Daws.accesskey=A_New_Key
```

This overrides the key defined at *conf/aws/credentials.conf*. We can do the same with any other keys we want to replace in the new environment.

This has an additional advantage, as we can define these properties inside an environment property in our server, in *JAVA_HOME*, decoupling the source code and the deployment server and reducing the chances of leaking keys by mistake.

Next Steps

At this stage we have a working Play application with the expected functionality. But we want to make this application available to the general public, and right now we are running it in our development machine. No one outside our internal network can see it, and that's not what we want.

Fortunately, Amazon AWS provides two solutions to this issue. The first one is to deploy the application in an [EC2](#) instance. The second is to use [Elastic BeanStalk](#) to deploy a war file generated via the [Play2War](#) Play Framework plugin.

The following chapters explain how to deploy the application in both Amazon services.