developerWorks®

Playful web development, Part 1: Manage user authentication with the Play Framework and Scala

Implementing a starter authentication UI with Play, Silhouette, and MongoDB

Pablo Pedemonte

November 12, 2015

Implement user management for your Play Framework applications and put your Scala skills to work. Pablo Pedemonte takes you through building a basic Play application that handles authentication and authorization. You can even use the application code as a starter for your own Play projects to shorten user-management development time.

View more content in this series

Implementing authentication in modern web applications can entail a significant amount of work. You need to enable users to authenticate via multiple mechanisms (credentials; social providers using OAuth1, OAuth2, or OpenID). User registration and password reset usually require email-based flows. And requests and views both must be aware of the identity (if any) of the logged-in user.

This tutorial presents a starter authentication application built with the Play Framework. Play is a member of a new generation of *reactive* web frameworks, such as Node.js and Vert.x, that are designed with scalability in mind. Play also brings development-friendly features such as native XML and JSON handling, in-browser error reporting in development mode, built-in test helpers, and Selenium integration. You can write Play applications in Java™ or Scala, but Scala is preferable. Functional languages are best suited to a reactive programming style. Although Java has finally embraced functional programming concepts in version 8, it lags behind Scala's extensive provision of functional features.

My starter application shows Scala and Play in action by implementing:

Play for scalability

By using asynchronous I/O, Play enforces a programming model in which application code *reacts* to events triggered when I/O has finished. In the meantime, threads running application code don't block and can process other requests. This design results in efficient use of processor cores: Play can handle heavy traffic with a handful of threads. And Play doesn't store any server-side session state, which favours horizontal scalability by putting session affinity out of the way: Any server in a cluster can pick up a request.

- Email-based user sign-up
- Credentials (email and password), and Twitter authentication via OAuth1
- Email-based password reset
- Credentials and Twitter account linking
- Examples of user-aware views, HTTP requests, and Ajax calls

The application uses Silhouette for the authentication work, and MongoDB for user persistence. All request processing and the interactions with MongoDB are completely asynchronous. You can use this application as a seed for your own projects to spare yourself the effort of implementing authentication from scratch.

I assume that you have basic familiarity with the Scala language. (If you have a Java background and need an introduction to Scala, I suggest reading the developerWorks *The busy Java developer's guide to Scala* series.) I also assume at least minimal experience with Play; basic knowledge about controllers, routes, and views is enough. For an introduction to Play, see the Getting Started section of the Play documentation. (The code in this tutorial uses the Play Framework version 2.4.2, so read the documentation for that version.) And be sure to check out Part 2 in this series, where I show you how to deploy Play applications on IBM Bluemix™.

Getting set up

Play at work

Companies that have adopted Play include LinkedIn, and Coursera. At the time of writing, the project has 2,411 forks on GitHub, and Stack Overflow has 10,308 questions tagged as *playframework*.

To build and run the starter application, you need Play 2.4.2 or newer and MongoDB on your system.

Install MongoDB and Play

Install MongoDB by following the instructions for your platform in the downloads section of the MongoDB site. On most Linux distributions, you can install MongoDB by using the corresponding package manager. For Mac, you can use Homebrew, and for Windows a MSI installer.

The Play Framework (since version 2.4.0) requires Java 8, so make sure you have a Java SE 8 SDK installed. To install Play, download and decompress the minimal activator — a 1MB ZIP file containing a start script that will download Play's code and dependencies (around 450MB) when run for the first time. For convenience, you might want to add the activator's folder to your system path.

Download and run the application

The sample application project is hosted at IBM Bluemix DevOps Services. You can obtain the source code by cloning the project's Git repository (you must sign in or register first). Alternatively, see Download to get the application as a ZIP file. When you have the code, run the application by following these steps:

1. Register an application in the Twitter Application Management site. Enter `http://dwdemo.com:9000/auth/social/twitter` as the callback URL.
2. Add `127.0.0.1 dwdemo.com` to your hosts file so that the application can be addressed by the same domain registered at Twitter.
3. Set the consumer key and consumer secret properties by copying the corresponding values from the Twitter application page to lines 28 and 29 of the conf/silhouette.conf file.
4. Start MongoDB server by running `mongod --dbpath` *`folder`*, where *`folder`* is a directory where MongoDB will store the database files.
5. To start the application, go to the root folder where you cloned the code and `activator run` it.

After the application starts (this step will take a while the first time), open it by going to http://dwdemo.com:9000. You'll see a welcome screen where users can sign up to create an account in the application.

Application configuration

With your environment set up, you're ready to delve into the application configuration.

Main configuration file

The main configuration file is conf/application.conf, the default location where Play looks for configuration properties. Listing 1 shows the relevant parts.

Listing 1. conf/application.conf

```
play.modules.enabled += "play.modules.reactivemongo.ReactiveMongoModule"
play.modules.enabled += "module.Module"

mongodb.uri = "mongodb://localhost:27017/demodb"

mail.from="dwplaydemo <mailrobot@dwplaydemo.net>"
mail.reply="No reply <noreply@dwplaydemo.net>"

play.mailer {
  mock = true
  host = localhost
}

play.http.filters = "utils.Filters"
play.http.errorHandler = "utils.ErrorHandler"

include "silhouette.conf"
```

The application.conf file declares two classes that configure dependency-injection bindings. `ReactiveMongoModule` makes reactive Mongo bindings injectable into application classes. The `module.Module` class specifies injection bindings for the application, mainly for Silhouette classes. The `mongodb.uri` property defines how to connect to MongoDB, and `play.mailer` sets up a mock mailing service for testing the sign-up and reset-password flows in development mode. A mock mailer logs emails to the console; in production, the application must use a SMTP server.

The `utils.Filters` class defines the filter pipeline for the application. Currently, this class uses Play's Cross-Site Request Forgery (CSRF) filter for protecting `POST` requests. The `utils.ErrorHandler` class has the role of setting the application global request error-handling policy; it defines redirects to error pages for internal server and page-not-found conditions. More important, the class also defines redirects to the sign-in page for nonauthorized or nonauthenticated attempts to access protected resources. The included silhouette.conf file declares Silhouette-specific settings. I'll go through those settings in the "Silhouette, in a nutshell" section.

Application routes

The conf/routes file defines the routes for the application-specific pages and authentication flows. Listing 2 shows the application-specific routes.

Listing 2. Application-specific routes

```
# Application
GET     /                          controllers.Application.index
GET     /profile                   controllers.Application.profile

# Rest api
GET     /rest/profile              controllers.RestApi.profile

# Public assets
GET     /assets/*file              controllers.Assets.versioned(path="/public", file: Asset)
GET     /webjars/*file             controllers.WebJarAssets.at(file)
```

The application consists of two pages. The index page is user aware and works either with a logged-in user or if anonymously accessed. The profile page is secured, demanding authorized access (otherwise, it redirects to the index page). The `/rest/profile` URL maps to a secure REST API endpoint returning the profile information for the logged-in user in JSON format. The last two URLs are the standard routes for public assets such as stylesheets, images, and JavaScript code. Listing 3 shows the authentication routes and flows.

Listing 3. Authentication routes

```
GET     /auth/signup               controllers.Auth.startSignUp
POST    /auth/signup               controllers.Auth.handleStartSignUp
GET     /auth/signup/:token        controllers.Auth.signUp(token:String)

GET     /auth/reset                controllers.Auth.startResetPassword
POST    /auth/reset                controllers.Auth.handleStartResetPassword
GET     /auth/reset/:token         controllers.Auth.resetPassword(token:String)
POST    /auth/reset/:token         controllers.Auth.handleResetPassword(token:String)

GET     /auth/signin               controllers.Auth.signIn
POST    /auth/authenticate         controllers.Auth.authenticate
GET     /auth/social/:providerId   controllers.Auth.socialAuthenticate(providerId:String)
GET     /auth/signout              controllers.Auth.signOut
```

The sign-up flow starts with a `GET` request for the sign-up page. From that page, users `POST` their sign-up information (email address, password, first and last name) to `/auth/signup`. The application processes the post by sending an email message with a `/auth/signup/:token` URL that users must follow to complete the sign-up operation. The flow for password resetting is similar: Users `GET` the password-reset page, where they `POST` an email address to `/auth/reset`. This post generates an email message containing a `/auth/reset/:token` URL. This URL takes users to a page where they enter and `POST` a new password to `/auth/reset/:token` to complete the process.

The `/auth/signin` route gives access to the sign-in page. The `auth/authenticate` route is the URL endpoint for credentials authentication, and `/auth/social/:providerId` the one for social authentication. Currently, the only supported social provider is Twitter, via OAuth1. The `/auth/signout` route exposes the endpoint for signing out from the application.

Modeling users and identity profiles

The application implements account linking, so users are associated with multiple identity profiles coming from different authentication providers (in this application, credentials or Twitter OAuth1). I represent identity profiles via a `Profile` class. A `User` has a list of identity profiles. Listing 4 shows the relevant code in app/models/User.scala.

Listing 4. User model

```
case class Profile(
  loginInfo:LoginInfo,
  confirmed: Boolean,
  email:Option[String],
  firstName: Option[String],
  lastName: Option[String],
  fullName: Option[String],
  passwordInfo:Option[PasswordInfo],
  oauth1Info: Option[OAuth1Info],
  avatarUrl: Option[String])

case class User(id: UUID, profiles: List[Profile]) extends Identity {
  def profileFor(loginInfo:LoginInfo) = profiles.find(_.loginInfo == loginInfo)
  def fullName(loginInfo:LoginInfo) = profileFor(loginInfo).flatMap(_.fullName)
}

object User {
  implicit val passwordInfoJsonFormat = Json.format[PasswordInfo]
  implicit val oauth1InfoJsonFormat = Json.format[OAuth1Info]
  implicit val profileJsonFormat = Json.format[Profile]
  implicit val userJsonFormat = Json.format[User]
}
```

Identity profiles (and therefore users) are uniquely identified by Silhouette's `LoginInfo` class — essentially a (user ID, provider ID) tuple. A profile can be already confirmed or waiting for confirmation. This feature is handy for profiles associated with the credentials provider, which must be confirmed as the last step of the sign-up process. Profiles also have some basic identity information (email address, user name, and avatar URL), all of which is optional because the identity information will vary among providers. A profile associated with the credentials provider stores a Silhouette `PasswordInfo` object holding the hashed password. A profile created by the OAuth1 Twitter provider stores a Silhouette `OAuth1Info` instance with authentication token and secret data. To support other authentication providers, the `Profile` class must be extended with extra fields (for example, an `oauth2Info:OAuth2Info` property for OAuth2).

The `User` class is a wrapper around a list of profiles that provides two convenience accessors for the profile and full name associated with a given `LoginInfo`. The `User` companion object declares automatic conversions from the model classes from and to JSON — necessary because the MongoDB driver works with JSON objects.

Model persistence

The application encapsulates the persistence code in data access objects (DAOs) for the `User`, `PasswordInfo`, and `OAuth1Info` classes. In app/daos/UserDao.scala, you'll find the `UserDao` trait, shown in Listing 5.

Listing 5. `UserDao` trait

```
trait UserDao {
  def save(user:User):Future[User]
  def find(loginInfo:LoginInfo):Future[Option[User]]
  def find(userId:UUID):Future[Option[User]]
  def confirm(loginInfo:LoginInfo):Future[User]
  def link(user:User, profile:Profile):Future[User]
  def update(profile:Profile):Future[User]
}
```

Users can be persisted and queried by ID or `LoginInfo`. The DAO also implements operations for confirming an identity profile, linking a new identity profile to a user, and updating an identity profile. Note the asynchronous nature of the DAO: All of the operations return an instance of `Future`, Scala's standard class for modeling computations that will eventually complete. Also in app/daos/UserDao.scala, you can find the MongoDB implementation of the `UserDao` trait, shown in Listing 6.

Listing 6. `MongoUserDao` class

```
class MongoUserDao extends UserDao {
  lazy val reactiveMongoApi = current.injector.instanceOf[ReactiveMongoApi]
  val users = reactiveMongoApi.db.collection[JSONCollection]("users")

  def find(loginInfo:LoginInfo):Future[Option[User]] =
    users.find(Json.obj("profiles.loginInfo" -> loginInfo)).one[User]

  def find(userId:UUID):Future[Option[User]] =
    users.find(Json.obj("id" -> userId)).one[User]

  def save(user:User):Future[User] =
    users.insert(user).map(_ => user)

  def confirm(loginInfo:LoginInfo):Future[User] = for {
    _ <- users.update(Json.obj(
      "profiles.loginInfo" -> loginInfo
    ), Json.obj("$set" -> Json.obj("profiles.$.confirmed" -> true)))
    user <- find(loginInfo)
  } yield user.get

  def link(user:User, profile:Profile) = for {
    _ <- users.update(Json.obj(
      "id" -> user.id
    ), Json.obj("$push" -> Json.obj("profiles" -> profile)))
    user <- find(user.id)
  } yield user.get

  def update(profile:Profile) = for {
    _ <- users.update(Json.obj(
      "profiles.loginInfo" -> profile.loginInfo
    ), Json.obj("$set" -> Json.obj("profiles.$" -> profile)))
    user <- find(profile.loginInfo)
  } yield user.get
}
```

The `MongoUserDao` class obtains a hook to the reactive Mongo API via Play's dependency injector and gets a reference to the collection that stores users. From there, the class operates using MongoDB's collection API with Play's JSON objects. Silhouette also requires DAOs for the `PasswordInfo` and `OAuth1Info` classes. Their implementation is similar to the `MongoUserDao` class. You can find the DAOs in app/daos/PasswordInfoDao.scala and app/daos/OAuth1InfoDao.scala in the full source code.

Testing the DAOs

The persistence code is the foundation of the authentication mechanism, so it's a good idea to make sure that it's working correctly before you move on. Play provides helpers and stubs that simplify test writing. For testing the persistence code, I'll use Play's `FakeApplication` class. This class will work using the same configuration as the actual application, except for the `mongodb.uri` property, which points to a test database. Listing 7 shows the code, which is in test/daos/DaoSpecResources.scala.

Listing 7. Creating a fake test application

```
def fakeApp = FakeApplication(additionalConfiguration =
     Map("mongodb.uri" -> "mongodb://localhost:27017/test"))

def withUserDao[T](t:MongoUserDao => T):T = running(fakeApp) {
  val userDao = new MongoUserDao
  Await.ready(userDao.users.drop(), timeout)
  t(userDao)
}
```

After declaring a fake application, the code defines a generic `withUserDao` method, which receives a function that takes a `MongoUserDao` and does the actual testing. That function runs in the context of the fake application, after the `users` collection in the test database of the fake application is cleared. The `withUserDao` method can be used to run a suite of specs2 tests such as the one in test/daos/UserSpecDao.scala, shown in Listing 8.

Listing 8. specs2 example user DAO test

```
"UserDao" should {
  "save users and find them by userId" in withUserDao { userDao =>
    val future = for {
      _ <- userDao.save(credentialsTestUser)
      maybeUser <- userDao.find(credentialsTestUser.id)
    } yield maybeUser.map(_ == credentialsTestUser)
    Await.result(future, timeout) must beSome(true)
  }
}
```

User service

Silhouette requires an implementation of an `IdentityService` trait for doing the authentication work. Listing 9 shows the implementation — a wrapper around an injected `UserDao`— in app/services/UserService.scala.

Listing 9. User service class

```
class UserService @Inject() (userDao:UserDao) extends IdentityService[User] {
  def retrieve(loginInfo:LoginInfo) = userDao.find(loginInfo)
  def save(user:User) = userDao.save(user)
  def find(id:UUID) = userDao.find(id)
  def confirm(loginInfo:LoginInfo) = userDao.confirm(loginInfo)
  def link(user:User, socialProfile:CommonSocialProfile) = {
    val profile = toProfile(socialProfile)
    if (user.profiles.exists(_.loginInfo == profile.loginInfo))
      Future.successful(user) else userDao.link(user, profile)
  }

  def save(socialProfile:CommonSocialProfile) = {
    val profile = toProfile(socialProfile)
    userDao.find(profile.loginInfo).flatMap {
      case None => userDao.save(User(UUID.randomUUID(), List(profile)))
      case Some(user) => userDao.update(profile)
    }
  }

  private def toProfile(p:CommonSocialProfile) = Profile(
    loginInfo = p.loginInfo,
    confirmed = true,
    email = p.email,
    firstName = p.firstName,
    lastName = p.lastName,
    fullName = p.fullName,
    passwordInfo = None,
    oauth1Info = None,
    avatarUrl = p.avatarURL
  )
}
```

The `save(user:User)` method persists a user during the sign-up flow. The `save(p:CommonSocialProfile)` method handles the case when a user has authenticated via a social provider. In this case, the application creates a new user if no user exists with the specified profile; otherwise, it updates the corresponding identity profile.

User tokens

The application generates user tokens as part of the sign-up and password-reset flows. User tokens are sent by email to the user, who must go to a URL based on the mailed token IDs to continue the flow. The models/UserToken.scala file implements tokens as a class holding user and token IDs and expiration data, as shown in Listing 10.

Listing 10. User token

```
case class UserToken(id:UUID, userId:UUID, email:String, expirationTime:DateTime, isSignUp:Boolean) {
  def isExpired = expirationTime.isBeforeNow
}

object UserToken {
  implicit val toJson = Json.format[UserToken]

  def create(userId:UUID, email:String, isSignUp:Boolean) =
    UserToken(UUID.randomUUID(), userId, email, new DateTime().plusHours(12), isSignUp)
}
```

User tokens are persisted to a MongoDB collection, so the companion object defines the required JSON formatting. From here, things happen exactly as with users. The application manipulates tokens by using the `UserTokenService` class (in services/UserTokenService.scala). This service class wraps an injected user token DAO, as shown in Listing 11.

Listing 11. User token service

```
class UserTokenService @Inject() (userTokenDao:UserTokenDao) {
  def find(id:UUID) = userTokenDao.find(id)
  def save(token:UserToken) = userTokenDao.save(token)
  def remove(id:UUID) = userTokenDao.remove(id)
}
```

`UserTokenDao` is a trait implemented by a `MongoUserTokenDao`. The `UserTokenDao` code is similar to the user DAO, and you can find it in daos/UserTokenDao.scala.

Silhouette, in a nutshell

The principal characteristic of the Silhouette framework is flexibility. Silhouette implements a set of stand-alone authentication components, and it's up to the developer to configure and combine them to build authentication logic. The main components are:

- **Identity Service**: Silhouette relies on an implementation of an `IdentityService` trait to handle all the operations related to retrieving users. In this way, user management is completely decoupled from the framework. The `UserService` class, described in the "User service" section, implements an identity service backed by MongoDB.
- **`AuthInfoRepository`**: Silhouette needs to know how to persist user credentials. The framework delegates this job to an implementation of the `AuthInfoRepository` trait. The application uses a composite repository that combines the `PasswordInfoDao` and `OAuth1InfoDao` classes described in the "Model persistence" section.
- **Authenticator:** Authenticators track a user after a successful authentication. They are tokens storing data such as its validity status and the login information for a user. Silhouette has implementations based on cookies, the Play stateless session, HTTP headers, and JSON Web Tokens (JWT).
- **Authenticator Service**: Every authenticator has an associated authenticator service responsible for an authenticator's lifecycle: creation, initialization, updates, renewal, and expiration.
- **Environment**: The environment defines the key components needed by a Silhouette application. It's type-parameterized by the user and authenticator types (in the application,

the `User` class defined in Listing 4 and a `CookieAuthenticator`). The environment is built by passing the identity service implementation (`UserService`) and the authenticator service implementation. I'm using the `CookieAuthenticatorService` class, required by the `CookieAuthenticator` type.

- **Provider**: A provider is a service that handles the authentication of a user. The application uses Silhouette's `CredentialsProvider` for local authentication and the OAuth1 `TwitterProvider`.
- **`SocialProviderRegistry`**: This is a placeholder for all the social providers supported by the application. In this case, it contains the `TwitterProvider` instance.

Configuring Silhouette components

Mind your Play version

The Play Framework adopted dependency injection as a built-in mechanism as of version 2.4.0. Older versions work with a completely different mechanism for passing parameters to controllers. Some libraries still don't work with Play 2.4.x. Because of Play's relative immaturity, changes between major releases tend to be disruptive.

Silhouette components are configured and combined by means of dependency injection. Play uses Google Guice as the default dependency-injection implementation. (You can plug in other implementations if you want.) The Guice `module.Module` class defines the bindings that Silhouette requires, starting with the basic declarations shown in Listing 12.

Listing 12. Dependency-injection bindings

```
class Module extends AbstractModule with ScalaModule {

  def configure() {
    bind[IdentityService[User]].to[UserService]
    bind[UserDao].to[MongoUserDao]
    bind[UserTokenDao].to[MongoUserTokenDao]
    bind[DelegableAuthInfoDAO[PasswordInfo]].to[PasswordInfoDao]
    bind[DelegableAuthInfoDAO[OAuth1Info]].to[OAuth1InfoDao]
    bind[IDGenerator].toInstance(new SecureRandomIDGenerator())
    bind[PasswordHasher].toInstance(new BCryptPasswordHasher)
    bind[FingerprintGenerator].toInstance(new DefaultFingerprintGenerator(false))
    bind[EventBus].toInstance(EventBus())
    bind[Clock].toInstance(Clock())
  }

  // ... Bindings for Silhouette components follow
}
```

The Listing 12 code defines bindings for Silhouette's identity service; the user, password, and OAuth1 DAOs; and a few objects needed by Silhouette's main components. Listing 13 shows the definition for those components, also in `module.Module`.

Listing 13. Dependency-injection bindings, continued

```
@Provides def provideEnvironment(
    identityService: IdentityService[User],
    authenticatorService: AuthenticatorService[CookieAuthenticator],
    eventBus: EventBus): Environment[User, CookieAuthenticator] = {
  Environment[User, CookieAuthenticator](identityService, authenticatorService, Seq(), eventBus)
}

@Provides def provideAuthenticatorService(
    fingerprintGenerator: FingerprintGenerator,
    idGenerator: IDGenerator,
    configuration: Configuration,
    clock: Clock): AuthenticatorService[CookieAuthenticator] = {
  val config = configuration.underlying.as[CookieAuthenticatorSettings]("silhouette.authenticator")
  new CookieAuthenticatorService(config, None, fingerprintGenerator, idGenerator, clock)
}

@Provides def provideCredentialsProvider(
    authInfoRepository: AuthInfoRepository,
    passwordHasher: PasswordHasher): CredentialsProvider = {
  new CredentialsProvider(authInfoRepository, passwordHasher, Seq(passwordHasher))
}

@Provides def provideAuthInfoRepository(
  passwordInfoDAO: DelegableAuthInfoDAO[PasswordInfo],
  oauth1InfoDAO: DelegableAuthInfoDAO[OAuth1Info]): AuthInfoRepository = {
   new DelegableAuthInfoRepository(passwordInfoDAO, oauth1InfoDAO)
}

@Provides def provideTwitterProvider(
    httpLayer: HTTPLayer,
    tokenSecretProvider: OAuth1TokenSecretProvider,
    configuration: Configuration): TwitterProvider = {
  val settings = configuration.underlying.as[OAuth1Settings]("silhouette.twitter")
  new TwitterProvider(httpLayer, new PlayOAuth1Service(settings), tokenSecretProvider, settings)
}

@Provides def provideOAuth1TokenSecretProvider(
    configuration: Configuration, clock: Clock): OAuth1TokenSecretProvider = {
  val cfg = configuration.underlying.as[CookieSecretSettings]("silhouette.oauth1TokenSecretProvider")
  new CookieSecretProvider(cfg, clock)
}

@Provides def provideSocialProviderRegistry(
    twitterProvider: TwitterProvider): SocialProviderRegistry = {
  SocialProviderRegistry(Seq(twitterProvider))
}
```

The `Environment`, which authenticates `User` instances via a `CookieAuthenticator`, is instantiated with:

- An `IdentityService` bound to the `UserService` class
- An `AuthenticatorService` bound to a `CookieAuthenticatorService`
- An empty list of request providers (not used in this application)
- An `EventBus` that can used to broadcast authentication events (not used in this application)

The `CredentialsProvider` is created by injecting an `AuthInfoRepository` backed by a `DelegableAuthInfoRepository` implementation that delegates credentials persistence to the `PasswordInfoDao` and `Oauth1InfoDao` classes. The `TwitterProvider` class requires an implementation of the `OAuth1TokenSecretProvider` trait, which defines how to persist token secrets during the OAuth1 dance. Finally, the application defines a `SocialProviderRegistry` that lists the `TwitterProvider` as the only social provider available.

The Silhouette configuration file

The bindings for the cookie authenticator service, the Twitter provider, and the OAuth1 token secret provider access configuration properties that are defined in the conf/silhouette.conf file (shown in Listing 14), which is included by the main conf/application.conf file (see Listing 1).

Listing 14. Silhouette configuration file

```
authenticator.cookieName="authenticator"
authenticator.cookiePath="/"
authenticator.secureCookie=false
authenticator.httpOnlyCookie=true
authenticator.useFingerprinting=true
authenticator.authenticatorIdleTimeout=30 minutes
authenticator.authenticatorExpiry=12 hours

oauth1TokenSecretProvider.cookieName="OAuth1TokenSecret"
oauth1TokenSecretProvider.cookiePath="/"
oauth1TokenSecretProvider.secureCookie=false
oauth1TokenSecretProvider.httpOnlyCookie=true
oauth1TokenSecretProvider.expirationTime=5 minutes

twitter.requestTokenURL="https://api.twitter.com/oauth/request_token"
twitter.accessTokenURL="https://api.twitter.com/oauth/access_token"
twitter.authorizationURL="https://api.twitter.com/oauth/authorize"
twitter.callbackURL="http://dwdemo.com:9000/auth/social/twitter"
twitter.consumerKey=${?TWITTER_CONSUMER_KEY}
twitter.consumerSecret=${?TWITTER_CONSUMER_SECRET}
```

See the Silhouette documentation for a detailed explanation of these properties. Note the use of the `TWITTER_CONSUMER_KEY` and `TWITTER_CONSUMER_SECRET` environment variables to avoid exposing OAuth1 secrets in the source code.

Secure actions

Silhouette defines two ad-hoc actions useful for implementing protected request handlers. These actions are available to controllers that mix in the `Silhouette` controller trait:

- `UserAwareAction`: This action **might** be executed by an authenticated user. The request received by the action will have an `identity` property of type `Option[U]` (for some application-dependent user type `U`) that will be defined if the request was issued by an authenticated user.
- `SecuredAction`: This action **must** be executed by an authenticated user. Otherwise, it will invoke the `onNotAuthorized` method of the application's error handler (in my application it redirects to the sign-in page, as explained in the "Application configuration" section). This action sets a request `identity` property with type `U` (for some application-dependent user type `U`), as shown in Listing 15.

Listing 15. Secured requests in Silhouette

```
class ApplicationController extends Silhouette {
  def someUserAwareAction = UserAwareAction.async {implicit request =>
    request.identity match {
      case None => // Request sent anonymously ...
      case Some(u) => // Request sent by authenticated user u
    }
  }

  def someSecureAction = SecureAction.async { implicit request =>
    logger.info(s"Logged user: ${request.identity}")
    ...
  }
}
```

User management and authentication

Having gone through the user model and Silhouette's configuration, you are ready to understand the authentication code. You'll examine sign-up and authentication in detail. All the code snippets in this section come from the `Auth` controller in the controllers/Auth.scala file. The `Auth` controller interfaces with all the Silhouette components described in the "Silhouette, in a nutshell" section and also the user and user token services. All of these components must be injected into the controller's constructor. The controller implements secure request handlers, so it mixes in the `Silhouette` controller trait (see the "Secure actions" section). Listing 16 shows the `Auth` code.

Listing 16. `Auth` controller class declaration

```
class Auth @Inject() (
  val messagesApi: MessagesApi,
  val env:Environment[User,CookieAuthenticator],
  socialProviderRegistry: SocialProviderRegistry,
  authInfoRepository: AuthInfoRepository,
  credentialsProvider: CredentialsProvider,
  userService: UserService,
  userTokenService: UserTokenService,
  avatarService: AvatarService,
  passwordHasher: PasswordHasher,
  configuration: Configuration,
  mailer: Mailer) extends Silhouette[User,CookieAuthenticator] {

    // ... auth controller code ...
}
```

User sign-up

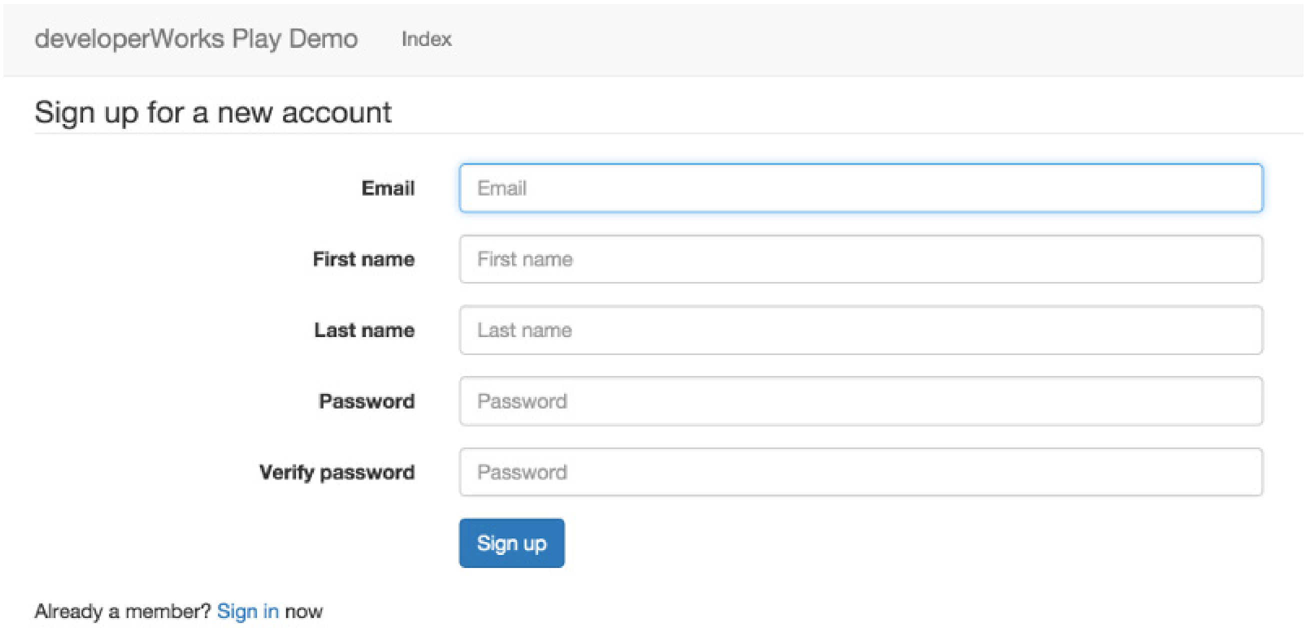
The sign-up flow starts at the `startSignUp` method. As shown in Listing 17, `startSignUp` is a user-aware asynchronous request handler.

Listing 17. The `startSignUp` method

```
def startSignUp = UserAwareAction.async { implicit request =>
  Future.successful(request.identity match {
    case Some(user) => Redirect(routes.Application.index)
    case None => Ok(views.html.auth.startSignUp(signUpForm))
  })
}
```

If a user is associated with the request, the method redirects to the index page. Otherwise, it serves the sign-up page, shown in Figure 1.

Figure 1. Sign-up page



Sign up for a new account

Email	Email
First name	First name
Last name	Last name
Password	Password
Verify password	Password

Already a member? Sign in now

The sign-up page consists of a form asking for the user email, first and last names, and password (twice for verification purposes). When submitted, the form is handled by the `handleStartSignUp` method, shown in Listing 18.

Listing 18. The `handleStartSignUp` method

```
def handleStartSignUp = Action.async { implicit request =>
  signUpForm.bindFromRequest.fold(
    bogusForm => Future.successful(BadRequest(views.html.auth.startSignUp(bogusForm))),
    signUpData => {
      val loginInfo = LoginInfo(CredentialsProvider.ID, signUpData.email)
      userService.retrieve(loginInfo).flatMap {
        case Some(_) =>
          Future.successful(Redirect(routes.Auth.startSignUp()).flashing(
            "error" -> Messages("error.userExists", signUpData.email)))
        case None =>
          val profile = Profile(
            loginInfo = loginInfo, confirmed=false, email=Some(signUpData.email),
            firstName=Some(signUpData.firstName), lastName=Some(signUpData.lastName),
            fullName=Some(s"${signUpData.firstName} ${signUpData.lastName}"),
            passwordInfo = None, oauth1Info = None, avatarUrl = None)
          for {
            avatarUrl <- avatarService.retrieveURL(signUpData.email)
            user <- userService.save(User(id = UUID.randomUUID(),
              profiles = List(profile.copy(avatarUrl = avatarUrl))))
            _ <- authInfoRepository.add(loginInfo, passwordHasher.hash(signUpData.password))
            token <- userTokenService.save(UserToken.create(user.id, signUpData.email, true))
          } yield {
            mailer.welcome(profile, link = routes.Auth.signUp(token.id.toString).absoluteURL())
            Ok(views.html.auth.finishSignUp(profile))
          }
      }
    }
  )
}
```

The code in Listing 18 starts by binding the request form to a `signUpForm` class. If the binding fails because the form is invalid (email address isn't valid, first or last name is empty, or passwords don't match) the method goes again to the sign-up page, displaying the validation errors. Otherwise, the method first checks if the system has a user registered with the received email. If so, again the user is redirected to the sign-up page with an error message.

After the user passes all the checks, the method instantiates an identity profile with the form's sign-up data and persists a user with that profile by calling `userService.save`. Then the method invokes `authInfoRepository.add` (which delegates to `PasswordInfoDao.save`) to persist the credentials, and creates a token. The process finishes by sending a welcome email with the token ID and redirecting to the finish sign-up page, which instructs the user to check for incoming email. The email links to the `/auth/signup/:token` route. That route maps to the `signUp` method, shown in Listing 19, which completes the sign-up operation.

Listing 19. The `signUp` method

```
def signUp(tokenId:String) = Action.async { implicit request =>
  val id = UUID.fromString(tokenId)
  userTokenService.find(id).flatMap {
    case None =>
      Future.successful(NotFound(views.html.errors.notFound(request)))
    case Some(token) if token.isSignUp && !token.isExpired =>
      userService.find(token.userId).flatMap {
        case None => Future.failed(new IdentityNotFoundException(Messages("error.noUser")))
        case Some(user) =>
          val loginInfo = LoginInfo(CredentialsProvider.ID, token.email)
          for {
            authenticator <- env.authenticatorService.create(loginInfo)
            value <- env.authenticatorService.init(authenticator)
            _ <- userService.confirm(loginInfo)
            _ <- userTokenService.remove(id)
            result <- env.authenticatorService.embed(value, Redirect(routes.Application.index()))
          } yield result
      }
    case Some(token) =>
      userTokenService.remove(id).map {_ => NotFound(views.html.errors.notFound(request))}
  }
}
```

The `signUp` method starts by verifying that the token ID exists in the database. If the token ID isn't in the database, the method redirects to the application's not-found error page. Then `signUp` verifies that the token ID corresponds to a sign-up token, that the token hasn't expired, and that the user associated with the token exists. If all the verifications succeed, the code proceeds to finish the sign-up process and to sign in the user, too. The sign-up is completed by recording that the user has confirmed the sign-up and by deleting the sign-up token. The sign-in consists of three steps:

1. Creating an authenticator (as explained in the "Silhouette, in a nutshell" section, a token recording authenticated user data), by calling `env.authenticatorService.create`
2. Initializing the authenticator (`env.authenticatorService.init`)
3. Embedding the authenticator in the request handler's response and redirecting to the index page (`env.authenticatorService.embed`)

This sequence completes the sign-up process. These three steps appear in the authentication code also.

Authentication via credentials

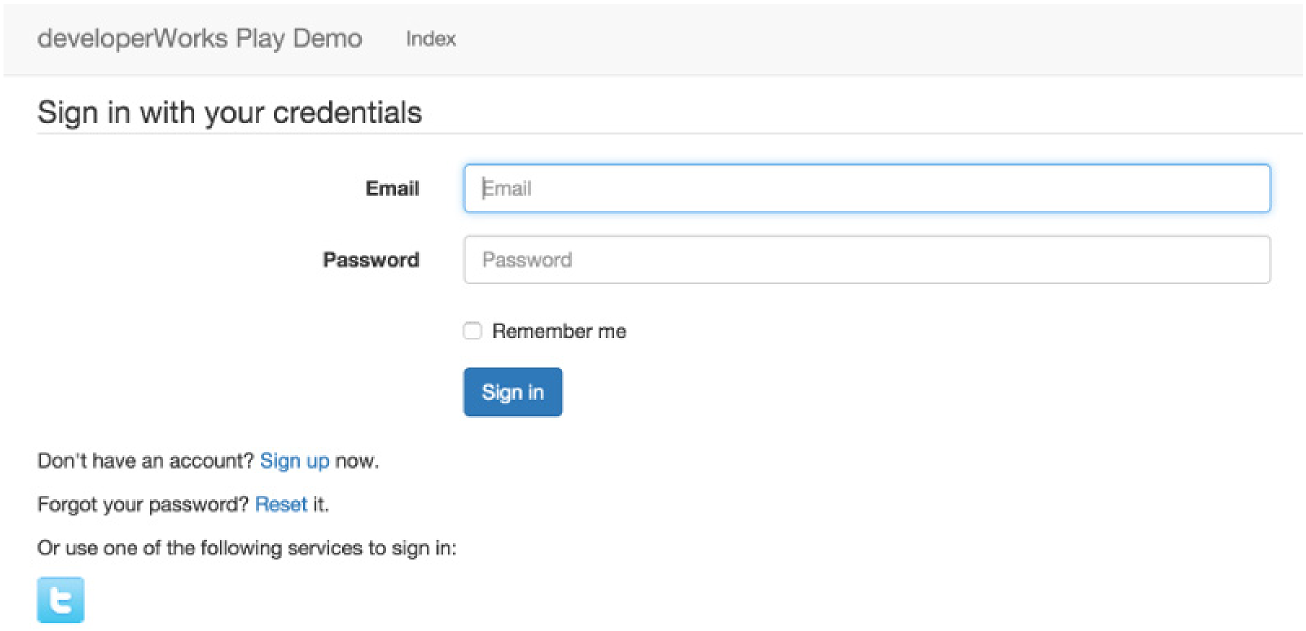
The asynchronous `authenticate` method, shown in Listing 20, implements the logic for authenticating users with their credentials defined during sign-up.

Listing 20. The `authenticate` method

```
def authenticate = Action.async { implicit request =>
  signInForm.bindFromRequest.fold(
    bogusForm => Future.successful(
      BadRequest(views.html.auth.signIn(bogusForm, socialProviderRegistry))),
    signInData => {
      val credentials = Credentials(signInData.email, signInData.password)
      credentialsProvider.authenticate(credentials).flatMap { loginInfo =>
        userService.retrieve(loginInfo).flatMap {
          case None =>
            Future.successful(Redirect(routes.Auth.signIn())
              .flashing("error" -> Messages("error.noUser")))
          case Some(user) if !user.profileFor(loginInfo).map(_.confirmed).getOrElse(false) =>
            Future.successful(Redirect(routes.Auth.signIn())
              .flashing("error" -> Messages("error.unregistered", signInData.email)))
          case Some(_) => for {
            authenticator <- env.authenticatorService.create(loginInfo).map {
              case authenticator if signInData.rememberMe => authenticator.copy(...) // Extend lifetime
              case authenticator => authenticator
            }
            value <- env.authenticatorService.init(authenticator)
            result <- env.authenticatorService.embed(value, Redirect(routes.Application.index()))
          } yield result
        }
      }.recover {
        case e:ProviderException =>
          Redirect(routes.Auth.signIn()).flashing("error" -> Messages("error.invalidCredentials"))
      }
    }
  )
}
```

The `authenticate` method is called from the sign-in page shown in Figure 2.

Figure 2. Sign-in page


developerWorks Play Demo Index

Sign in with your credentials

Email: Email

Password: Password

Remember me

Sign in

Don't have an account? Sign up now.

Forgot your password? Reset it.

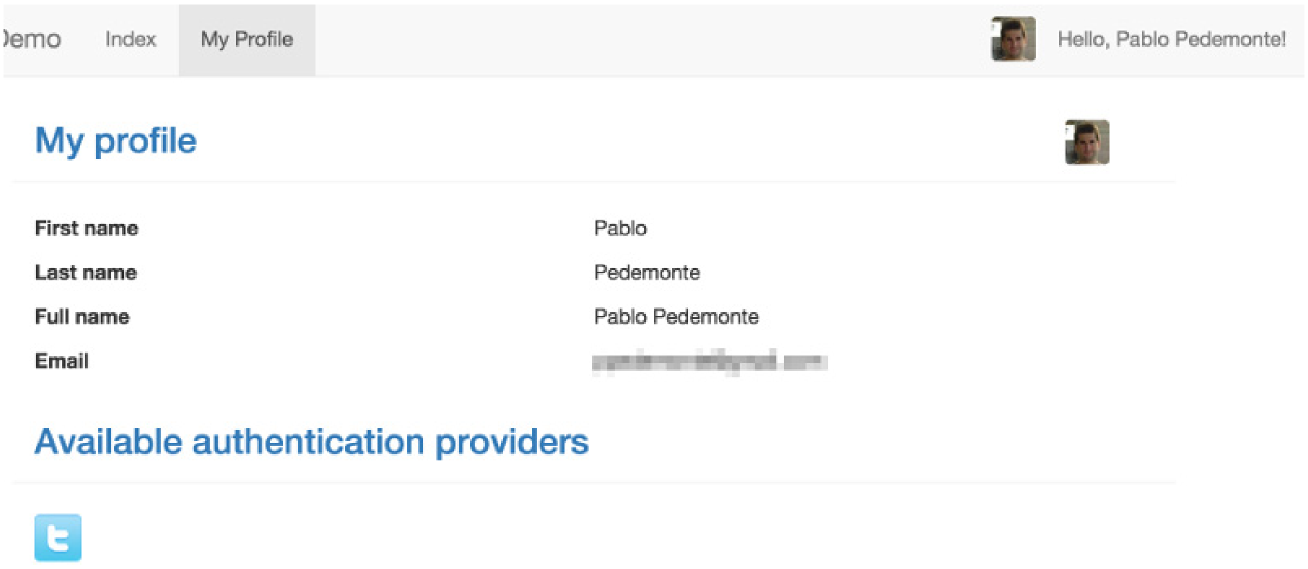
Or use one of the following services to sign in:

The `authenticate` method's logic is simpler than it looks. As usual, the method tries to bind the request payload to a `signInForm` (a tuple with an email and a password, and a remember-me flag). If the form isn't valid, the authentication finishes with a redirect to the sign-in page that shows the validation errors. Otherwise, the method tries to authenticate by calling `credentialsProvider.authenticate`. If the authentication fails, it returns a `Future` containing an exception, from which the code recovers by going back to the sign-in page with an appropriate error message. Otherwise,`credentialsProvider.authenticate` returns a `Future` with a `LoginInfo` instance. From here, the code checks if the user associated with the `LoginInfo` exists, and if so whether that user has completed the registration. If these checks pass, the code performs the three steps outlined in Listing 19— that is, create an authenticator, initialize it, and embed it in the response (a redirect to the index page). As an intermediate step, if the Remember me check box was selected, the code modifies the authenticator by creating a copy with an extended lifetime. (For the sake of simplicity, I'm omitting these details from the listing.)

Authentication via Twitter

Twitter OAuth1 authentication happens when the user clicks the Twitter icon at the bottom left of the sign-in page (Figure 2), or when an authenticated user selects Twitter from the Available authentication providers section of the user profile page (shown in Figure 3).

Figure 3. Profile page (only available for authenticated users)



My profile

First name	Pablo
Last name	Pedemonte
Full name	Pablo Pedemonte
Email	[illegible]

Available authentication providers

In both cases, the application sends a request to the `/auth/social/:providerId` route, with the `providerId` set to the `twitter` string. Listing 21 shows the `socialAuthenticate` method associated with the route.

Listing 21. Twitter authentication

```
def socialAuthenticate(providerId:String) = UserAwareAction.async { implicit request =>
  (socialProviderRegistry.get[SocialProvider](providerId) match {
    case Some(p:SocialProvider with CommonSocialProfileBuilder) => p.authenticate.flatMap {
      case Left(result) => Future.successful(result)
      case Right(authInfo) => for {
        profile <- p.retrieveProfile(authInfo)
        user <- request.identity.fold(userService.save(profile))(userService.link(_,profile))
        authInfo <- authInfoRepository.save(profile.loginInfo, authInfo)
        authenticator <- env.authenticatorService.create(profile.loginInfo)
        value <- env.authenticatorService.init(authenticator)
        result <- env.authenticatorService.embed(value, Redirect(routes.Application.index()))
      } yield result
    }
    case _ => Future.successful(
      Redirect(request.identity.fold(routes.Auth.signIn())(_ => routes.Application.profile()))
     .flashing("error" -> Messages("error.noProvider", providerId))
    )
  }).recover {
    case e:ProviderException =>
      logger.error("Provider error", e)
      Redirect(request.identity.fold(routes.Auth.signIn())(_ => routes.Application.profile()))
        .flashing("error" -> Messages("error.notAuthenticated", providerId))
  }
}
```

The request handler is user aware. The logic is driven by the following premise: If the request is anonymous, it's a sign-in operation; otherwise, the request was triggered from the user profile page and denotes an account-linking operation. The code starts verifying that the provider ID is available in the social provider registry. If not, the method redirects to the sign-in or to the profile page, depending on whether the request denotes a sign-in or a link operation. After this check, the code calls the provider's `authenticate` method and the authentication dance starts:

1. The first time that the `authenticate` call is completed, it returns a `Left(result)` value. This `result` is a redirect to the external site that's performing the authentication process. When the external site completes the authentication, it returns to the application via the callback URL configured in silhouette.conf. Listing 14 shows that the callback points back to the route associated with the `socialAuthenticate` method, so it is reexecuted.
2. On reexecution, `socialAuthenticate` again invokes the provider's `authenticate` method, this time returning a `Right(authInfo)` value that's processed as follows:
    a. The application asks the social provider for an identity profile associated with the obtained `authInfo`.
    b. The application checks if the request is anonymous. If it is anonymous, the operation is deemed as a sign-in and a user is created or updated (see Listing 9). Otherwise, the user is linking accounts, and the identity profile is linked to the existing user.
    c. The code saves the `authInfo` and finishes with the usual three steps described in Listing 19: Create an authenticator, initialize it, and embed it in the response (a redirect to the index page).

If at any point in the OAuth1 dance a call to `authenticate` fails, the method recovers by redirecting to the index or the user profile page.

Securing REST API calls

If a user attempts nonauthenticated access to a protected resource, the application redirects the request to the index page (see the "Application configuration" section). But this error policy is not acceptable for securing REST API endpoints. A REST API error response must have a correct HTTP status and a payload explaining the error condition, rather than redirecting the user to an application's page. REST API request handlers must override the default error policy. Silhouette makes that task easy: Controllers mixing in the `Silhouette` trait inherit error handlers that by default have no effect. But they can be overridden to customize error handling on a controller basis. Listing 22 shows the application's REST API controller.

Listing 22. Rest API controller

```
class RestApi @Inject() (
  val messagesApi: MessagesApi,
  val env:Environment[User,CookieAuthenticator]) extends Silhouette[User,CookieAuthenticator] {

  def profile = SecuredAction.async { implicit request =>
    val json = Json.toJson(request.identity.profileFor(request.authenticator.loginInfo).get)
    val prunedJson = json.transform(
      (__ \ 'loginInfo).json.prune andThen
      (__ \ 'passordInfo).json.prune andThen
      (__ \ 'oauth1Info).json.prune)
    prunedJson.fold(
      _ => Future.successful(InternalServerError(Json.obj("error" -> Messages("error.profileError")))),
      js => Future.successful(Ok(js))
    )
  }

  override def onNotAuthenticated(request:RequestHeader) = {
    Some(Future.successful(Unauthorized(Json.obj("error" -> Messages("error.profileUnauth")))))
  }
}
```

The REST API consists of a single secure request handler returning a JSON object with profile information for the logged-in user. The JSON object is pruned from sensitive information before being sent to the caller. The class overrides the `onNonAuthenticated` method, so anonymous calls to the REST API return a response with unauthorized status and a payload with an error message.

Wrapping up

This tutorial explains how to set up a basic yet complete Play application that implements user management and authentication. With minimal and localized changes (for example, implementing your own DAOs to use a persistence mechanism other than MongoDB) you can adapt the application to your own projects, without needing to implement user management from scratch. Along the way, the code shows some of the developer-friendly features that Play offers, such as using a fake application for testing DAOs, automatic conversion of classes to JSON, and automatic form binding and validation.

Did you know that Play applications can run on Bluemix? Read the next tutorial in this series, where I show how to deploy the authentication application to the IBM cloud.

Downloadable resources

Description	Name	Size
Application code	dwPlayDemo.zip	1070KB

Related topics

- LinkedIn, and Coursera: Find out why these companies have adopted Scala and the Play Framework.
- *The busy Java developer's guide to Scala* (Ted Neward, developerWorks, January 2008): Check out this series of introductory Scala articles aimed at developers with a Java background.
- Play Documentation: Read the Getting Started section for a birds-eye view of the Play Framework.
- developerWorks Premium: Provides an all-access pass to powerful tools, curated technical library from Safari Books Online, conference discounts and proceedings, SoftLayer and Bluemix credits, and more.
- Silhouette: Visit the website for Silhouette, an authentication library for Play Framework applications supporting OAuth1, OAuth2, OpenID, and credentials authentication schemes.
- Google Guice: Dig into Play's default dependency-injection implementation.
- Play Framework: Download the Play Framework and start implementing your own reactive applications.
- MongoDB: Get MondoDB, a NoSQL database designed to efficiently store, query, and manipulate JSON-like documents.