Silhouette Documentation

Release 1

Christian Kaps

September 15, 2014

Contents

Silhouette is an authentication library for Play Framework applications that supports several authentication methods, including OAuth1, OAuth2, OpenID, Credentials or custom authentication schemes.

It can be integrated as is, or used as a building block and customized to meet specific application requirements, thanks to its loosely coupled design.

The project is named after the fictional crime fighter character Silhouette, from the Watchmen graphic novel and movie.

CHAPTER 1

Basics

1.1 Features

Easy to integrate

Silhouette comes with an Activator template that gives you a complete sample application which is 100% customizable. You must only select the template `play-silhouette-seed` in your Activator UI. It was never been easier to start a new Silhouette application.

Authentication support

Out of the box support for leading social services such as Twitter, Facebook, Google, LinkedIn and GitHub. It also provides a credentials provider with supports local login functionality.

Asynchronous, non-blocking operations

We follow the Reactive Manifesto. This means that all requests and web service calls are asynchronous, non blocking operations. For the event handling part of Silhouette we use Akka's Event Bus implementation. And lastly all persistence interfaces are defined to return Scala Futures.

Very customizable, extendable and testable

From the ground up Silhouette was designed to be as customizable, extendable and testable as possible. All components can be enhanced via inheritance or replaced based on their traits, thanks to its loosely coupled design.

Internationalization support

Silhouette makes it very easy to internationalize your application by passing the Play Framework `Request` and `Lang` objects around, if internationalization comes into play.

Well tested

Silhouette is a security component which protects your users from being compromised by attackers. Therefore we try to cover the complete code with unit and integration tests.

Follows the OWASP Authentication Cheat Sheet

Silhouette implements and promotes best practices such as described by the OWASP Authentication Cheat Sheet like Password Strength Controls, SSL Client Authentication or use of authentication protocols that require no password.

1.2 Installation

For Play 2.2 use version 0.9

```
libraryDependencies ++= Seq(
  "com.mohiva" %% "play-silhouette" % "0.9"
)
```

For Play 2.3 use version 1.0

This version is cross compiled against Scala 2.10.4 and 2.11.0

```
libraryDependencies ++= Seq(
  "com.mohiva" %% "play-silhouette" % "1.0"
)
```

If you want to use the latest snapshot, add the following instead:

```
resolvers += "Sonatype Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots/"

libraryDependencies ++= Seq(
  "com.mohiva" %% "play-silhouette" % "1.1-SNAPSHOT"
)
```

1.3 Examples

This page lists some examples which shows how Silhouette can be implemented.

Silhouette Seed Template

The Silhouette Seed project is an Activator template which shows how Silhouette can be implemented in a Play Framework application. It's a starting point which can be extended to fit your needs.

Link

Silhouette Slick Seed Template

A fork of the Silhouette Seed project which implements Slick as database access library.

Link

Silhouette Cake Seed Template

Seed project for Play Framework with Silhouette, using the Cake Pattern for dependency injection.

Link

Silhouette Rest Seed

Seed project for Play Framework with Silhouette, expose rest api for signup and signin.

Link

1.4 Releases

The `master` branch contains the current development version. It should be working and passing all tests at any time, but it's unstable and represents a work in progress.

Released versions are indicated by tags.

Release numbers will follow Semantic Versioning.

1.5 Help

If you need help with the integration of Silhouette into your project, don't hesitate and ask questions in our mailing list or on Stack Overflow.

CHAPTER 2

Core

2.1 Actions

2.1.1 Securing your Actions

Silhouette provides a replacement for Play's built in Action class named `SecuredAction`. This action intercepts requests and checks if there is an authenticated user. If there is one, the execution continues and your code is invoked.

```
class Application(env: Environment[User, CachedCookieAuthenticator])
  extends Silhouette[User, CachedCookieAuthenticator] {

  /**
   * Renders the index page.
   *
   * @returns The result to send to the client.
   */
  def index = SecuredAction { implicit request =>
    Ok(views.html.index(request.identity))
  }
}
```

There is also a `UserAwareAction` that can be used in actions that need to know if there is a current user but can be executed even if there isn't one.

```
class Application(env: Environment[User, CachedCookieAuthenticator])
  extends Silhouette[User, CachedCookieAuthenticator] {

  /**
   * Renders the index page.
   *
   * @returns The result to send to the client.
   */
  def index = UserAwareAction { implicit request =>
    val userName = request.identity match {
      case Some(identity) => identity.fullName
      case None => "Guest"
    }
    Ok("Hello %s".format(userName))
  }
}
```

For not authenticated users you can implement a global or local fallback action.

Global Fallback

You can implement the `SecuredSettings` trait into your `Global` object. This trait provides a method called `onNotAuthenticated`. If you implement this method, then every time a user calls a restricted action, the result specified in the global fallback method will be returned.

```
object Global extends GlobalSettings with SecuredSettings {

  /**
   * Called when a user isn't authenticated.
   *
   * @param request The request header.
   * @param lang The current selected lang.
   * @return The result to send to the client.
   */
  override def onNotAuthenticated(request: RequestHeader, lang: Lang) = {
    Some(Future.successful(Unauthorized("No access")))
  }
}
```

Local Fallback

Every controller which is derived from `Silhouette` base controller has a method called `notAuthenticated`. If you override these method, then you can return a not-authenticated result similar to the global fallback but only for these specific controller. The local fallback has precedence over the global fallback.

```
class Application(env: Environment[User, CachedCookieAuthenticator])
  extends Silhouette[User, CachedCookieAuthenticator] {

  /**
   * Called when a user isn't authenticated.
   *
   * @param request The request header.
   * @return The result to send to the client.
   */
  override def notAuthenticated(request: RequestHeader): Option[Future[SimpleResult]] = {
    Some(Future.successful(Unauthorized("No access")))
  }

  /**
   * Renders the index page.
   *
   * @returns The result to send to the client.
   */
  def index = SecuredAction { implicit request =>
    Ok(views.html.index(request.identity))
  }
}
```

Note: If you don't implement one of the both fallback methods, a 401 response with a simple message will be displayed to the user.

2.1.2 Adding Authorization

Silhouette provides a way to add authorization logic to your controller actions. This is done by implementing an `Authorization` object that is passed to the `SecuredAction` as a parameter.

After checking if a user is authenticated the `Authorization` instance is used to verify if the execution should be allowed or not.

```
/**
 * A trait to define Authorization objects that let you hook
 * an authorization implementation in SecuredActions.
 *
 * @tparam I The type of the identity.
 */
trait Authorization[I <: Identity] {

  /**
   * Checks whether the user is authorized to execute an action or not.
   *
   * @param identity The identity to check for.
   * @param request The current request header.
   * @param lang The current lang.
   * @return True if the user is authorized, false otherwise.
   */
  def isAuthorized(identity: I)(implicit request: RequestHeader, lang: Lang): Boolean
}
```

This is a sample implementation that only grants access to users that logged in using a given provider:

```
case class WithProvider(provider: String) extends Authorization[User] {
  def isAuthorized(user: User)(implicit request: RequestHeader, lang: Lang) = {
    user.identityId.providerId == provider
  }
}
```

Here's how you would use it:

```
def myAction = SecuredAction(WithProvider("twitter")) { implicit request =>
    // do something here
}
```

For not authorized users you can implement a global or local fallback action similar to the fallback for not-authenticated users.

Global Fallback

You can implement the `SecuredSettings` trait into your `Global` object. This trait provides a method called `onNotAuthorized`. If you implement this method, then every time a user calls an action on which he isn't authorized, the result specified in the global fallback method will be returned.

```
object Global extends GlobalSettings with SecuredSettings {

  /**
   * Called when a user isn't authorized.
   *
   * @param request The request header.
   * @param lang The current selected lang.
   * @return The result to send to the client.
```

```
    */
  override def onNotAuthorized(request: RequestHeader, lang: Lang) = {
    Some(Future.successful(Forbidden("Not authorized")))
  }
}
```

Local Fallback

Every controller which is derived from `Silhouette` base controller has a method called `notAuthorized`. If you override these method, then you can return a not-authorized result similar to the global fallback but only for these specific controller. The local fallback has precedence over the global fallback.

```
class Application(env: Environment[User, CachedCookieAuthenticator])
  extends Silhouette[User, CachedCookieAuthenticator] {

  /**
   * Called when a user isn't authorized.
   *
   * @param request The request header.
   * @return The result to send to the client.
   */
  override def notAuthorized(request: RequestHeader): Option[Future[SimpleResult]] = {
    Some(Future.successful(Forbidden("Not authorized")))
  }

  /**
   * Renders the index page.
   *
   * @returns The result to send to the client.
   */
  def index = SecuredAction(WithProvider("twitter")) { implicit request =>
    Ok(views.html.index(request.identity))
  }
}
```

Note: If you don't implement one of the both fallback methods, a 403 response with a simple message will be displayed to the user.

2.1.3 Handle Ajax requests

If you send Ajax and normal requests to your Play app, then you should tell your app that it should handle Ajax requests differently, so that it can respond with a JSON result, for example. There are two different methods to achieve this. The first method uses a non-standard HTTP request header. Then on the Play side you can check for this header and respond with a suitable result. The second approach uses Content negotiation to serve different versions of a document based on the `ACCEPT` request header.

Non-standard header

The example below uses a non-standard HTTP request header inside a secured action and inside a fallback method for non-authenticated users.

The JavaScript part with JQuery

```
$.ajax({
    headers: { 'IsAjax': 'true' },
    ...
});
```

The Play part with a local fallback method for not-authenticated users

```
class Application(env: Environment[User, CachedCookieAuthenticator])
  extends Silhouette[User, CachedCookieAuthenticator] {

  /**
   * Called when a user isn't authenticated.
   *
   * @param request The request header.
   * @return The result to send to the client.
   */
  override def notAuthenticated(request: RequestHeader): Option[Future[SimpleResult]] = {
    val result = request.headers.get("IsAjax") match {
      case Some("true") => Json.obj("result" -> "No access")
      case _ => "No access"
    }

    Some(Future.successful(Unauthorized(result)))
  }

  /**
   * Renders the index page.
   *
   * @returns The result to send to the client.
   */
  def index = SecuredAction { implicit request =>
    val result = request.headers.get("IsAjax") match {
      case Some("true") => Json.obj("identity" -> request.identity)
      case _ => views.html.index(request.identity)
    }

    Ok(result)
  }
}
```

Content negotiation

By default Silhouette supports content negotiation for the most common media types: `text/plain`, `text/html`, `application/json` and `application/xml`. So if no local or global fallback methods are implemented, Silhouette responds with the appropriate response based on the `ACCEPT` header defined by the user agent. The response format will default to plain text in case the request does not match one of the known media types. The example below uses content negotiation inside a secured action and inside a fallback method for not-authenticated users.

The JavaScript part with JQuery

```
$.ajax({
    headers: {
        Accept : "application/json; charset=utf-8",
        "Content-Type": "application/json; charset=utf-8"
    },
    ...
})
```

The Play part with a local fallback method for not-authenticated users

```
class Application(env: Environment[User, CachedCookieAuthenticator])
  extends Silhouette[User, CachedCookieAuthenticator] {

  /**
   * Called when a user isn't authenticated.
   *
   * @param request The request header.
   * @return The result to send to the client.
   */
  override def notAuthenticated(request: RequestHeader): Option[Future[SimpleResult]] = {
    val result = render {
      case Accepts.Json() => Json.obj("result" -> "No access")
      case Accepts.Html() => "No access"
    }

    Some(Future.successful(Unauthorized(result)))
  }

  /**
   * Renders the index page.
   *
   * @returns The result to send to the client.
   */
  def index = SecuredAction { implicit request =>
   val result = render {
      case Accepts.Json() => Json.obj("identity" -> request.identity)
      case Accepts.Html() => views.html.index(request.identity)
    }

    Ok(result)
  }
}
```

2.2 Identity

Silhouette defines a user through its `Identity` trait. This trait doesn't define any defaults expect the *Login information* which contains the data about the provider that authenticated that identity. This login information must be stored with the identity. If the application supports the concept of "merged identities", i.e., the same user being able to authenticate through different providers, then make sure that the login information gets stored separately. Later you can see how this can be implemented.

2.2.1 Implement an identity

As pointed out in the introduction of this chapter a user is an representation of an `Identity`. So to define a user you must create a class that extends the `Identity` trait. Your user can contain any information. There are no restrictions how a user must be constructed.

Below we define a simple user with a unique ID, the login information for the provider which authenticates that user and some basic information like name and email.

```
case class User(,
  userID: Long,
  loginInfo: LoginInfo,
```

```
  name: String,
  email: Option[String]) extends Identity
```

2.2.2 The identity service

Silhouette relies on an implementation of `IdentityService` to handle all the operations related to retrieving identities. Using this delegation model you are not forced to use a particular model object or a persistence mechanism but rather provide a service that translates back and forth between your models and what Silhouette understands.

The `IdentityService` defines a raw type which must be derived from `Identity`. This has the advantage that your service implementation returns always your implemented `Identity`.

Let's illustrate how a user defined implementation can look like:

```
/**
 * A custom user service which relies on the previous defined `User`.
 */
class UserService(userDAO: UserDAO) extends IdentityService[User] {

  /**
   * Retrieves a user that matches the specified login info.
   *
   * @param loginInfo The login info to retrieve a user.
   * @return The retrieved user or None if no user could be retrieved for the given login info.
   */
  def retrieve(loginInfo: LoginInfo): Future[Option[User]] = userDAO.findByLoginInfo(loginInfo)
}
```

2.2.3 Link an identity to multiple login information

Silhouette doesn't provide built-in functionality to link multiple login information to a single user, but it makes this task very easy by providing the basics. In the abstract this task can be done by linking the different login information returned by each provider, with a single user identified by an unique ID. The unique user will be represented by your implementation of `Identity` and the login information will be returned by every provider implementation after a successful authentication. Now with this basic knowledge it's up to you to implement the linking in such a way that it fits into your application architecture.

2.3 Providers

In Silhouette a provider is a service that handles the authentication of an identity. It typically reads authorization information and returns information about an identity.

2.3.1 Social providers

A social provider allows an identity to authenticate it on your website with an existing account from an external social website like Facebook, Google or Twitter. Following you can find a list of all supported providers grouped by authentication protocol.

OAuth1

- LinkedInProvider (www.linkedin.com)
- TwitterProvider (www.twitter.com)
- XingProvider (www.xing.com)

OAuth2

- FacebookProvider (www.facebook.com)
- FoursquareProvider (www.foursquare.com)
- GitHubProvider (www.github.com)
- GoogleProvider (www.google.com)
- InstagramProvider (www.instagram.com)
- LinkedInProvider (www.linkedin.com)
- VKProvider (www.vk.com)

2.3.2 Credentials provider

Silhouette supports also local authentication with the credentials provider. This provider accepts credentials and returns the login information for an identity after a successful authentication. Typically credentials consists of an identifier(a username or email address) and a password.

The provider supports the change of password hashing algorithms on the fly. Sometimes it may be possible to change the hashing algorithm used by the application. But the hashes stored in the backing store can't be converted back into plain text passwords, to hash them again with the new algorithm. So if a user successfully authenticates after the application has changed the hashing algorithm, the provider hashes the entered password again with the new algorithm and stores the authentication info in the backing store.

2.3.3 Social profile builders

It can be the case that more than the common profile information supported by Silhouette should be returned from a social provider. Additional profile information could be, as example, the location or the gender of a user. To solve this issue Silhouette has a very neat solution called "profile builders". The advantage of this solution is that it prevents a developer to duplicate existing code by overriding the providers to return the additional profile information. Instead the developer can mixin a profile builder which adds only the programming logic for the additional fields.

The default social profile builder

Silhouette ships with a built-in profile builder called `CommonSocialProfileBuilder` which returns the `CommonSocialProfile` object after successful authentication. This profile builder is mixed into all provider implementations by instantiating it with their companion objects. So instead of instantiating the provider with the new keyword, you must call the apply method on its companion object.

```
FacebookProvider(cacheLayer, httpLayer, oAuthSettings)
```

Hint: All provider implementations are abstract, so they cannot be instantiated without a profile builder. This is because of the fact that a concrete type in Scala cannot be overridden by a different concrete type. And therefore we cannot mixin the common profile builder by default, because it couldn't be overridden by a custom profile builder.

Write a custom social profile builder

As noted above it is very easy to write your own profile builder implementations. Lets take a look on the following code examples. The first one defines a custom social profile that differs from the common social profile by the additional gender field.

```
case class CustomSocialProfile[A <: AuthInfo](
  loginInfo: LoginInfo,
  authInfo: A,
  firstName: Option[String] = None,
  lastName: Option[String] = None,
  fullName: Option[String] = None,
  email: Option[String] = None,
  avatarURL: Option[String] = None,
  gender: Option[String] = None) extends SocialProfile[A]
```

Now we create a profile builder which can be mixed into to the Facebook provider to return our previous defined custom profile.

```
trait CustomFacebookProfileBuilder extends SocialProfileBuilder[OAuth2Info] {
  self: FacebookProvider =>

  /**
   * Defines the profile to return by the provider.
   */
  override type Profile = CustomSocialProfile[OAuth2Info]

  /**
   * Parses the profile from the Json response returned by the Facebook API.
   */
  override protected def parseProfile(parser: JsonParser, json: JsValue): Try[Profile] = Try {
    val commonProfile = parser(json)
    val gender = (json \ "gender").asOpt[String]

    CustomSocialProfile(
      loginInfo = commonProfile.loginInfo,
      authInfo = commonProfile.authInfo,
      firstName = commonProfile.firstName,
      lastName = commonProfile.lastName,
      fullName = commonProfile.fullName,
      avatarURL = commonProfile.avatarURL,
      email = commonProfile.email,
      gender = gender)
  }
}
```

As you can see there is no need to duplicate any Json parsing. The only thing to do is to query the gender field from the Json response returned by the Facebook API.

Now you can mixin the profile builder by instantiating the Facebook provider with the profile builder.

```
new FacebookProvider(cacheLayer, httpLayer, oAuthSettings) with CustomFacebookProfileBuilder
```

2.4 Error handling

Every provider implementation in Silhouette throws either an `AccessDeniedException` or an `AuthenticationException`. The `AccessDeniedException` will be thrown if a user provides incorrect credentials or a social provider denies an authentication attempt. In any other case the `AuthenticationException` will be thrown.

Due the asynchronous nature of Silhouette, all exceptions will be throw in Futures. This means that exceptions may not be caught in a `try catch` block because Futures may be executed in separate Threads. To solve this problem, Futures have its own error handling mechanism. They can be recovered from an error state into desirable state with the help of the `recover` and `recoverWith` methods. These both methods accepts an partial function which transforms the `Exception` into any other type. For more information please visit the documentation of the `Future` trait.

All controller implementations which are derived from the `Silhouette` base controller, can make the usage of the `exceptionHandler` method. This is a default recover implementation which transforms an `AccessDeniedException` into a 403 Forbidden result and an `AuthenticationException` into a 401 Unauthorized result. The following code snippet shows how an error can be recovered from a failed authentication try.

```
authenticate().map { result =>
  // Do something with the result
}.recoverWith(exceptionHandler)
```

Note: The `exceptionHandler` method calls the local or global fallback methods under the hood to return the appropriate result.

2.5 Caching

Silhouette caches some authentication artifacts. So if you use a clustered environment for your application then make sure that you use a distributed cache like Redis or Memcached.

2.6 Events

Silhouette provides event handling based on Akka's Event Bus. The following events are provided within the core of Silhouette, although only the three marked of them are fired from core.

- SignUpEvent
- LoginEvent
- LogoutEvent
- AccessDeniedEvent
- AuthenticatedEvent *
- NotAuthenticatedEvent *
- NotAuthorizedEvent *

It is very easy to propagate own events over the event bus by implementing the `SilhouetteEvent` trait.

```
case class CustomEvent() extends SilhouetteEvent
```

2.6.1 Use the event bus

The event bus is available in every Silhouette controller over the environment variable `env.eventBus`. You can also inject the event bus into other classes like services or DAOs. You must only take care that you use the same event bus. There exists a singleton event bus by calling `EventBus()`.

2.6.2 Listen for events

To listen for events you must implement a listener based on an `Actor` and then register the listener, with the event to listen, on the event bus instance:

```
val listener = system.actorOf(Props(new Actor {
  def receive = {
    case e @ LoginEvent(identity, request, lang) => println(e)
    case e @ LogoutEvent(identity, request, lang) => println(e)
  }
}))

val eventBus = EventBus()
eventBus.subscribe(listener, classOf[LoginEvent[User]])
eventBus.subscribe(listener, classOf[LogoutEvent[User]])
```

2.6.3 Publish events

Publishing events is also simple:

```
val eventBus = EventBus()
eventBus.publish(LoginEvent[User](identity, request, lang))
```

2.7 Logging

Silhouette uses named loggers for logging. So in your application you have a more fine-grained control over the log entries logged by Silhouette.

As an example, in your logging.xml you can set the logging level for the complete `com.mohiva` namespace to `ERROR` and then define a more detailed level for some components.

```
<configuration debug="false">
    ...
    <logger name="com.mohiva" level="ERROR" />
    <logger name="com.mohiva.play.silhouette.core.AccessDeniedException" level="INFO" />
    ...
</configuration>
```

2.8 Glossary

2.8.1 Identity

The Identity trait represents an authenticated user.

2.8.2 Social profile

The social profile contains the profile data returned from the social providers. Silhouette provides a default social profile which contains the most common profile data a provider can return. But it is also possible to define an own social profile which can be consists of more data.

2.8.3 Login information

Contains the data about the provider that authenticated an identity. This information is mostly public available and it simply consists of a unique provider ID and a unique key which identifies a user on this provider (userID, email, ...). This information will be represented by the LoginInfo trait.

2.8.4 Authentication information

The authentication information contains secure data like access tokens, hashed passwords and so on, which should never be exposed to the public. To retrieve other than by Silhouette supported information from a provider, try to connect again with this information and fetch the missing data. Due its nature, the information will be represented by different implementations. Mostly every provider implementation defines its own AuthInfo implementation.

CHAPTER 3

Configuration

3.1 Configuration

3.1.1 Introduction

Silhouette doesn't dictate the form and the location of the configuration. So you can configure it over a database, a configuration server or the Play Framework configuration mechanism.

Generally configuration in Silhouette is handled over configuration objects. These objects must be filled with data and then passed to the instance to configure.

Note: For reasons of simplification we use the Play Framework configuration syntax in our examples.

3.1.2 OAuth1 based providers

To configure OAuth1 based providers you must use the `OAuth1Settings` class. This object has the following form:

```
case class OAuth1Settings(
  requestTokenURL: String,
  accessTokenURL: String,
  authorizationURL: String,
  callbackURL: String,
  consumerKey: String,
  consumerSecret: String)
```

Callback URL

The `callbackURL` must point to your action which is responsible for the authentication over your defined providers. So if you define the following route as example:

```
GET  /authenticate/:provider  @controllers.SocialAuthController.authenticate(provider)
```

Then your `callbackURL` must have the following format:

```
callbackURL="https://your.domain.tld/authenticate/linkedin"
```

Example

Your configuration could then have this format:

```
linkedin {
  requestTokenURL="https://api.linkedin.com/uas/oauth/requestToken"
  accessTokenURL="https://api.linkedin.com/uas/oauth/accessToken"
  authorizationURL="https://api.linkedin.com/uas/oauth/authenticate"
  callbackURL="https://your.domain.tld/authenticate/linkedin"
  consumerKey="your.consumer.key"
  consumerSecret="your.consumer.secret"
}

twitter {
  requestTokenURL="https://twitter.com/oauth/request_token"
  accessTokenURL="https://twitter.com/oauth/access_token"
  authorizationURL="https://twitter.com/oauth/authenticate"
  callbackURL="https://your.domain.tld/authenticate/twitter"
  consumerKey="your.consumer.key"
  consumerSecret="your.consumer.secret"
}

xing {
  requestTokenURL="https://api.xing.com/v1/request_token"
  accessTokenURL="https://api.xing.com/v1/access_token"
  authorizationURL="https://api.xing.com/v1/authorize"
  callbackURL="https://your.domain.tld/authenticate/xing"
  consumerKey="your.consumer.key"
  consumerSecret="your.consumer.secret"
}
```

To get the consumerKey/consumerSecret keys you need to log into the developer site of each service and register your application.

3.1.3 OAuth2 based providers

To configure OAuth2 based providers you must use the `OAuth2Settings` class. This object has the following form:

```
case class OAuth2Settings(
  authorizationURL: String,
  accessTokenURL: String,
  redirectURL: String,
  clientID: String,
  clientSecret: String,
  scope: Option[String] = None,
  authorizationParams: Map[String, String] = Map(),
  accessTokenParams: Map[String, String] = Map(),
  customProperties: Map[String, String] = Map())
```

Redirect URL

The `redirectURL` must point to your action which is responsible for the authentication over your defined providers. So if you define the following route as example:

```
GET  /authenticate/:provider  @controllers.SocialAuthController.authenticate(provider)
```

Then your `redirectURL` must have the following format:

```
redirectURL="https://your.domain.tld/authenticate/facebook"
```

Example

Your configuration could then have this format:

```
facebook {
  authorizationUrl="https://graph.facebook.com/oauth/authorize"
  accessTokenUrl="https://graph.facebook.com/oauth/access_token"
  redirectURL="https://your.domain.tld/authenticate/facebook"
  clientId="your.client.id"
  clientSecret="your.client.secret"
  scope=email
}

foursquare {
  authorizationUrl="https://foursquare.com/oauth2/authenticate"
  accessTokenUrl="https://foursquare.com/oauth2/access_token"
  redirectURL="https://your.domain.tld/authenticate/foursquare"
  clientId="your.client.id"
  clientSecret="your.client.secret"
}

github {
  authorizationUrl="https://github.com/login/oauth/authorize"
  accessTokenUrl="https://github.com/login/oauth/access_token"
  redirectURL="https://your.domain.tld/authenticate/github"
  clientId="your.client.id"
  clientSecret="your.client.secret"
}

google {
  authorizationUrl="https://accounts.google.com/o/oauth2/auth"
  accessTokenUrl="https://accounts.google.com/o/oauth2/token"
  redirectURL="https://your.domain.tld/authenticate/google"
  clientId="your.client.id"
  clientSecret="your.client.secret"
  scope="profile email"
}

instagram {
  authorizationUrl="https://api.instagram.com/oauth/authorize"
  accessTokenUrl="https://api.instagram.com/oauth/access_token"
  redirectURL="https://your.domain.tld/authenticate/instagram"
  clientId="your.client.id"
  clientSecret="your.client.secret"
}

linkedin {
  authorizationUrl="https://www.linkedin.com/uas/oauth2/authorization"
  accessTokenUrl="https://www.linkedin.com/uas/oauth2/accessToken"
  redirectURL="https://your.domain.tld/authenticate/linkedin"
  clientId="your.client.id"
  clientSecret="your.client.secret"
}
```

```
vk {
  authorizationUrl="http://oauth.vk.com/authorize"
  accessTokenUrl="https://oauth.vk.com/access_token"
  redirectURL="https://your.domain.tld/authenticate/vk"
  clientId="your.client.id"
  clientSecret="your.client.secret"
}
```

To get the clientId/clientSecret keys you need to log into the developer site of each service and register your application.