Spring Social Reference Manual

Craig Walls
Keith Donald

Spring Social Reference Manual

by Craig Walls and Keith Donald

1.0.0.M3

Table of Contents

1. Spring Social Overview

1.1 Introduction

The Spring Social project enables your applications to establish Connections with Software-as-a-Service (SaaS) Providers such as Facebook and Twitter to invoke APIs on behalf of Users.

1.2 Socializing applications

The phrase "social networking" often refers to efforts aimed at bringing people together. In the software world, those efforts take the form of online social networks such as Facebook, Twitter, and LinkedIn. Roughly half a billion of this world's internet users have flocked to these services to keep frequent contact with family, friends, and colleagues.

Under the surface, however, these services are just software applications that gather, store, and process information. Just like so many applications written before, these social networks have users who sign in and perform some activity offered by the service.

What makes these applications a little different than traditional applications is that the data that they collect represent some facet of their users' lives. What's more, these applications are more than willing to share that data with other applications, as long as the user gives permission to do so. This means that although these social networks are great at bringing people together, as software services they also excel at bringing applications together.

To illustrate, imagine that Paul is a member of an online movie club. A function of the movie club application is to recommend movies for its members to watch and to let its members maintain a list of movies that they have seen and those that they plan to see. When Paul sees a movie, he signs into the movie club site, checks the movie off of his viewing list, and indicates if he liked the movie or not. Based on his responses, the movie club application can tailor future recommendations for Paul to see.

On its own, the movie club provides great value to Paul, as it helps him choose movies to watch. But Paul is also a Facebook user. And many of Paul's Facebook friends also enjoy a good movie now and then. If Paul were able to connect his movie club account with his Facebook profile, the movie club application could offer him a richer experience. Perhaps when he sees a movie, the application could post a message on his Facebook wall indicating so. Or when offering suggestions, the movie club could factor in the movies that his Facebook friends liked.

Social integration is a three-way conversation between a service provider, a service consumer, and a user who holds an account on both the provider and consumer. All interactions between the consumer and the service provider are scoped to the context of the user's profile on the service provider.

In the narrative above, Facebook is the service provider, the movie club application is the service consumer, and Paul is the user of both. The movie club application may interact with Facebook on behalf of Paul, accessing whatever Facebook data and functionality that Paul permits, including seeing Paul's list of friends and posting messages to his Facebook wall.

From the user's perspective, both applications provide some valuable functionality. But by connecting the user's account on the consumer application with his account on the provider application, the user brings together two applications that can now offer the user more value than they could individually.

With Spring Social, your application can play the part of the service consumer, interacting with a service provider on behalf of its users. The key features of Spring Social are:

- A "Connect Framework" that handles the core authorization and connection flow with service providers.
- A "Connect Controller" that handles the OAuth exchange between a service provider, consumer, and user in a web application environment.
- Java bindings to popular service provider APIs including Facebook, Twitter, LinkedIn, TripIt, GitHub, and Gowalla.
- A "Signin Controller" that allows users to authenticate with your application by signing in with their Provider accounts, such as their Twitter or Facebook accounts.

1.3 How to get

Spring Social is divided into the modules described in Table 1.1, “Spring Social Modules”.

Table 1.1. Spring Social Modules

Name	Description
spring-social-core	Spring Social's Connect Framework and OAuth client support.
spring-social-web	Spring Social's `ConnectController` which uses the Connect Framework to manage connections in a web application environment.
spring-social-facebook	Spring Social's Facebook API binding
spring-social-twitter	Spring Social's Twitter API binding.
spring-social-linkedin	Spring Social's LinkedIn API binding.
spring-social-github	Spring Social's GitHub API binding.
spring-social-gowalla	Spring Social's Gowalla API binding.
spring-social-tripit	Spring Social's TripIt API binding.
spring-social-test	Support for testing Connect implementations and API bindings.

Which of these modules your application needs will largely depend on what facets of Spring Social you intend to use. At very minimum, you'll need the core module in your application's classpath:

```
<dependency>
  <groupId>org.springframework.social</groupId>
  <artifactId>spring-social-core</artifactId>
  <version>${org.springframework.social-version}</version>
```

```
</dependency>
```

To let Spring Social handle the back-and-forth authorization handshake between a web application and a service provider, you'll need the web module:

```
<dependency>
<groupId>org.springframework.social</groupId>
<artifactId>spring-social-web</artifactId>
<version>${org.springframework.social-version}</version>
</dependency>
```

The remaining modules are elective, depending on which of the supported service providers you intend for your application to interact with. For example, you'll only need the GitHub module if your application needs to invoke the Github API:

If you are developing against a milestone version, such as 1.0.0.M3, you will need to add the following repository in order to resolve the artifact:

```
<repository>
  <id>org.springframework.maven.milestone</id>
  <name>Spring Maven Milestone Repository</name>
  <url>http://maven.springframework.org/milestone</url>
</repository>
```

If you are testing out the latest nightly build version (e.g. 1.0.0.BUILD-SNAPSHOT), you will need to add the following repository:

```
<repository>
  <id>org.springframework.maven.snapshot</id>
  <name>Spring Maven Snapshot Repository</name>
  <url>http://maven.springframework.org/snapshot</url>
</repository>
```

2. Service Provider 'Connect' Framework

The `spring-social-core` module includes a *Service Provider 'Connect' Framework* for managing connections to Software-as-a-Service (SaaS) providers such as Facebook and Twitter. This framework allows your application to establish connections between local user accounts and accounts those users have with external service providers. Once a connection is established, it can be be used to obtain a strongly-typed Java binding to the ServiceProvider's API, giving your application the ability to invoke the API on behalf of a user.

To illustrate, consider Facebook as an example ServiceProvider. Suppose your application, AcmeApp, allows users to share content with their Facebook friends. To support this, a connection needs to be established between a user's AcmeApp account and her Facebook account. Once established, a FacebookApi instance can be obtained and used to post content to the user's wall. Spring Social's 'Connect' framework provides a clean API for managing service provider connections such as this.

2.1 Core API

The `Connection<A>` interface models a connection to an external service provider such as Facebook:

```
public interface Connection<A> {

    ConnectionKey getKey();

    String getDisplayName();

    String getProfileUrl();

    String getImageUrl();

    void sync();

    boolean test();

    boolean hasExpired();

    void refresh();

    UserProfile fetchUserProfile();

    void updateStatus(String message);

    A getApi();

    ConnectionData createData();

}
```

Each connection is uniquely identified by a composite key consisting of a providerId (e.g. 'facebook') and connected providerUserId (e.g. '1255689239', for Keith Donald's Facebook ID). This key tells you what provider user the connection is connected to.

A connection has a number of meta-properties that can be used to render it on a screen, including a displayName, profileUrl, and imageUrl. As an example, the following HTML template snippet could be used to generate a link to the connected user's profile on the provider's site:

```
<img src="${connection.imageUrl}" /> <a href="${connection.profileUrl}">${connection.displayName}</a>
```

The value of these properties may depend on the state of the provider user's profile. In this case, sync() can be called to synchronize these values if the user's profile is updated.

A connection can be tested to determine if its authorization credentials are valid. If invalid, the connection may have expired or been revoked by the provider. If the connection has expired, a connection may be refreshed to renew its authorization credentials.

A connection provides several operations that allow the client application to invoke the ServiceProvider's API in a uniform way. This includes the ability to fetch a model of the user's profile and update the user's status in the provider's system.

A connection's parameterized type <A> represents the Java binding to the ServiceProvider's native API. An instance of this API binding can be obtained by calling `getApi()`. As an example, a Facebook connection instance would be parameterized as Connection<FacebookApi>. `getApi()` would return a FacebookApi instance that provides a Java binding to Facebook's graph API for a specific Facebook user.

Finally, the internal state of a connection can be captured for transfer between layers of your application by calling `createData()`. This could be used to persist the connection in a database, or serialize it over the network.

To put this model into action, suppose we have a reference to a Connection<TwitterApi> instance. Suppose the connected user is the Twitter user with screen name 'kdonald'.

1. Connection#getKey() would return ('twitter', '14718006') where '14718006' is @kdonald's Twitter-assigned user id that never changes.

2. Connection#getDisplayName() would return '@kdonald'.

3. Connection#getProfileUrl() would return 'http://twitter.com/kdonald'.

4. Connection#getImageUrl() would return 'http://a0.twimg.com/profile_images/105951287/IMG_5863_2_normal.jpg'.

5. Connection#sync() would synchronize the state of the connection with @kdonald's profile.

6. Connection#test() would return true indicating the authorization credentials associated with the Twitter connection are valid. This assumes Twitter has not revoked the AcmeApp client application, and @kdonald has not reset his authorization credentials (Twitter connections do not expire).

7. Connection#hasExpired() would return false.

8. Connection#refresh() would not do anything since connections to Twitter do not expire.

9. Connection#fetchUserProfile() would make a remote API call to Twitter to get @kdonald's profile data and normalize it into a UserProfile model.

10. Connection#updateStatus(String) would post a status update to @kdonald's timeline.

11. Connection#getApi() would return a TwitterApi giving the client application access to the full capabilities of Twitter's native API.

12. Connection#createData() would return ConnectionData that could be serialized and used to restore the connection at a later time.

2.2 Establishing connections

So far we have discussed how existing connections are modeled, but we have not yet discussed how new connections are established. The manner in which connections between local users and provider users are established varies based on the authorization protocol used by the ServiceProvider. Some service providers use OAuth, others use Basic Auth, others may use something else. Spring Social currently provides native support for OAuth-based service providers, including support for OAuth 1 and OAuth 2. This covers the leading social networks, such as Facebook and Twitter, all of which use OAuth to secure their APIs. Support for other authorization protocols can be added by extending the framework.

Each authorization protocol is treated as an implementation detail where protocol-specifics are kept out of the core Connection API. A ConnectionFactory abstraction encapsulates the construction of connections that use a specific authorization protocol. In the following sections, we will discuss the major ConnectionFactory classes provided by the framework. Each section will also describe the protocol-specific flow required to establish a new connection.

OAuth2 service providers

OAuth 2 is rapidly becoming a preferred authorization protocol, and is used by major service providers such as Facebook, Github, Foursquare, Gowalla, and 37signals. In Spring Social, a OAuth2ConnectionFactory is used to establish connections with a OAuth2-based service provider:

```
public class OAuth2ConnectionFactory<A> extends ConnectionFactory<A> {

    public OAuth2Operations getOAuthOperations();

    public Connection<A> createConnection(AccessGrant accessGrant);

}
```

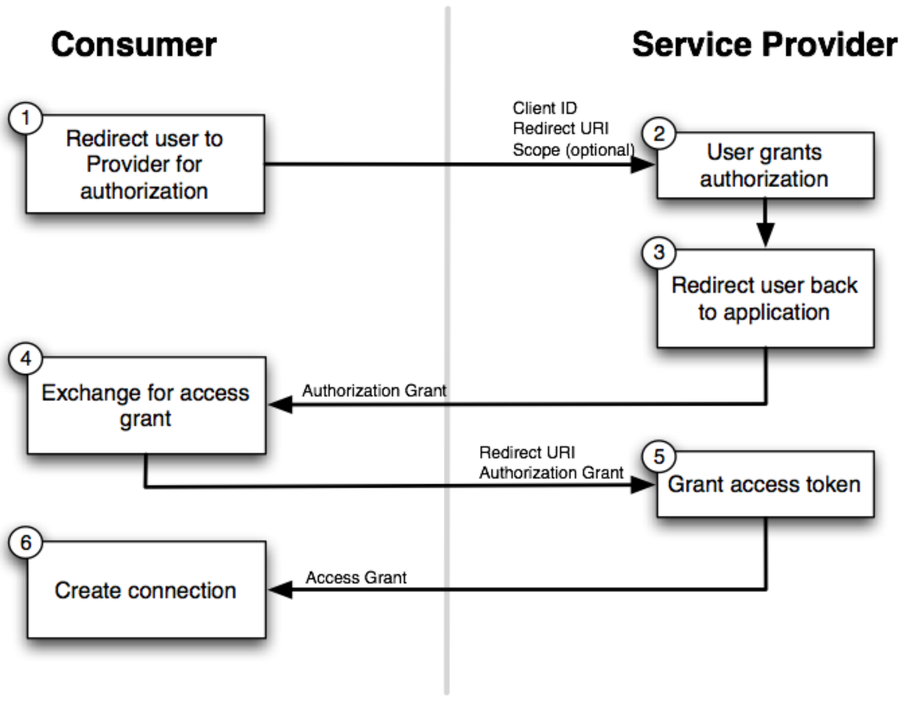
`getOAuthOperations()` returns an API to use to conduct the authorization flow, or "OAuth Dance", with a service provider. The result of this flow is an `AccessGrant` that can be used to establish a connection with a local user account by calling `createConnection`. The OAuth2Operations interface is shown below:

```
public interface OAuth2Operations {

    String buildAuthorizeUrl(GrantType grantType, OAuth2Parameters parameters);
    AccessGrant exchangeForAccess(String authorizationCode, String redirectUri, MultiValueMap<String, String> a

}
```

Callers are first expected to call buildAuthorizeUrl(GrantType, OAuth2Parameters) to construct the URL to redirect the user to for connection authorization. Upon user authorization, the authorizationCode returned by the provider should be exchanged for an AccessGrant. The AccessGrant should then used to create a connection. This flow is illustrated below:



As you can see, there is a back-and-forth conversation that takes place between the application and the service provider to grant the application access to the provider account. This exchange, commonly known as the "OAuth Dance", follows these steps:

1. The flow starts by the application redirecting the user to the provider's authorization URL. Here the provider displays a web page asking the user if he or she wishes to grant the application access to read and update their data.

2. The user agrees to grant the application access.

3. The service provider redirects the user back to the application (via the redirect URI), passing an authorization code as a parameter.

4. The application exchanges the authorization code for an access grant.

5. The service provider issues the access grant to the application. The grant includes an access token and a refresh token. One receipt of these tokens, the "OAuth dance" is complete.

6. The application uses the AccessGrant to establish a connection between the local user account and the external provider account. With the connection established, the application can now obtain a reference to the Service API and invoke the provider on behalf of the user.

The example code below shows use of a FacebookConnectionFactory to create a connection to Facebook using the OAuth2 server-side flow illustrated above. Here, FacebookConnectionFactory is a subclass of OAuth2ConnectionFactory:

```
FacebookConnectionFactory connectionFactory = new FacebookConnectionFactory("clientId", "clientSecret");
OAuth2Operations oauth2Operations = connectionFactory.getOAuth2Operations();
String authorizeUrl = oauth2Operations.buildAuthorizeUrl(GrantType.AUTHORIZATION_CODE, new OAuth2Parameters("ca
response.sendRedirect(authorizeUrl);
// when the provider callback is received with the authorizationCode parameter:
AccessGrant accessGrant = oauth2Operations.exchangeForAccess(authorizationCode, "callbackUrl");
Connection<FacebookApi> connection = connectionFactory.createConnection(accessGrant);
```

The following example illustrates the client-side "implicit" authorization flow also supported by OAuth2. The difference between this flow and the server-side "authorization code" flow above is the provider callback directly contains the access grant (no additional exchange is necessary). This flow is appropriate for clients incapable of keeping the access grant credentials confidential, such as a mobile device or JavaScript-based user agent.

```
FacebookConnectionFactory connectionFactory = new FacebookConnectionFactory("clientId", "clientSecret");
OAuth2Operations oauth2Operations = connectionFactory.getOAuth2Operations();
String authorizeUrl = oauth2Operations.buildAuthorizeUrl(GrantType.IMPLICIT_GRANT, new OAuth2Parameters("callba
response.sendRedirect(authorizeUrl);
// when the provider callback is received with the access_token parameter:
AccessGrant accessGrant = new AccessGrant(accessToken);
Connection<FacebookApi> connection = connectionFactory.createConnection(accessGrant);
```

OAuth1 service providers

OAuth 1 is the previous version of the OAuth protocol. It is more complex OAuth 2, and sufficiently different that it is supported separately. Twitter, Linked In, and TripIt are some of the well-known ServiceProviders that use OAuth 1. In Spring Social, the OAuth1ConnectionFactory allows you to create connections to a OAuth1-based Service Provider:

```
public class OAuth1ConnectionFactory<A> extends ConnectionFactory<A> {

    public OAuth1Operations getOAuthOperations();

    public Connection<A> createConnection(OAuthToken accessToken);

}
```

Like a OAuth2-based provider, `getOAuthOperations()` returns an API to use to conduct the authorization flow, or "OAuth Dance". The result of the OAuth 1 flow is an `OAuthToken` that can be used to establish a connection with a local user account by calling `createConnection`. The OAuth1Operations interface is shown below:

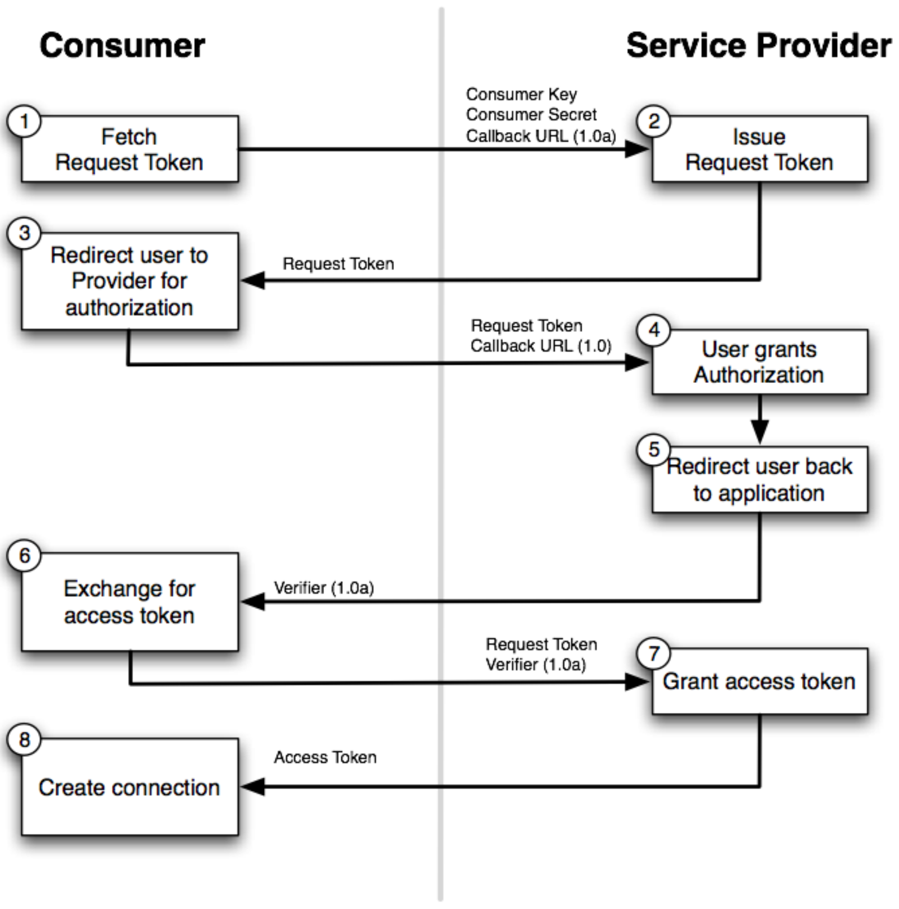
```
public interface OAuth1Operations {
    OAuthToken fetchRequestToken(String callbackUrl, MultiValueMap<String, String> additionalParameters);

    String buildAuthorizeUrl(String requestToken, OAuth1Parameters parameters);

    OAuthToken exchangeForAccessToken(AuthorizedRequestToken requestToken, MultiValueMap<String, String> additi

}
```

Callers are first expected to call fetchNewRequestToken(String) obtain a temporary token from the ServiceProvider to use during the authorization session. Next, callers should call buildAuthorizeUrl(String, OAuth1Parameters) to construct the URL to redirect the user to for connection authorization. Upon user authorization, the authorized request token returned by the provider should be exchanged for an access token. The access token should then used to create a connection. This flow is illustrated below:



1. The flow starts with the application asking for a request token. The purpose of the request token is to obtain user approval and it can only be used to obtain an access token. In OAuth 1.0a, the consumer callback URL is passed to the provider when asking for a request token.

2. The service provider issues a request token to the consumer.

3. The application redirects the user to the provider's authorization page, passing the request token as a parameter. In OAuth 1.0, the callback URL is also passed as a parameter in this step.

4. The service provider prompts the user to authorize the consumer application and the user agrees.

5. The service provider redirects the user's browser back to the application (via the callback URL). In OAuth 1.0a, this redirect includes a verifier code as a parameter. At this point, the request token is authorized.

6. The application exchanges the authorized request token (including the verifier in OAuth 1.0a) for an access token.

7. The service provider issues an access token to the consumer. The "dance" is now complete.

8. The application uses the access token to establish a connection between the local user account and the external provider account. With the connection established, the application can now obtain a reference to the Service API and invoke the provider on behalf of the user.

The example code below shows use of a TwitterConnectionFactory to create a connection to Facebook using the OAuth1 server-side flow illustrated above. Here, TwitterConnectionFactory is a subclass of OAuth1ConnectionFactory:

```
TwitterConnectionFactory connectionFactory = new TwitterConnectionFactory("consumerKey", "consumerSecret");
OAuth1Operations oauth1Operations = connectionFactory.getOAuth1Operations();
String requestToken = oauth1Operations.fetchRequestToken("callbackUrl");
String authorizeUrl = oauth1Operations.buildAuthorizeUrl(requestToken, OAuth1Parameters.NONE);
response.sendRedirect(authorizeUrl);
// when the provider callback is received with the oauth_token and oauth_verifier parameters:
OAuthToken accessToken = oauth1Operations.exchangeForAccessToken(new AuthorizedRequestToken(oauthToken, oauthVe
Connection<TwitterApi> connection = connectionFactory.createConnection(accessToken);
```

Registering ConnectionFactory instances

As you will see in subsequent sections of this reference guide, Spring Social provides infrastructure for establishing connections to one or more providers in a dynamic, self-service manner. For example, one client application may allow users to connect to Facebook, Twitter, and LinkedIn. Another might integrate Github and Pivotal Tracker. To make the set of connectable providers easy to manage and locate, Spring Social provides a registry for centralizing connection factory instances:

```
ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();
registry.addConnectionFactory(new FacebookConnectionFactory("clientId", "clientSecret"));
registry.addConnectionFactory(new TwitterConnectionFactory("consumerKey", "consumerSecret"));
registry.addConnectionFactory(new LinkedInConnectionFactory("consumerKey", "consumerSecret"));
```

This registry implements a locator interface that other objects can use to lookup connection factories dynamically:

```
public interface ConnectionFactoryLocator {

    ConnectionFactory<?> getConnectionFactory(String providerId);

    <A> ConnectionFactory<A> getConnectionFactory(Class<A> serviceApiType);

    Set<String> registeredProviderIds();
}
```

Example usage of a ConnectionFactoryLocator is shown below:

```
// generic lookup by providerId
ConnectionFactory<?> connectionFactory = locator.getConnectionFactory("facebook");

// typed lookup by service api type
ConnectionFactory<FacebookApi> connectionFactory = locator.getConnectionFactory(FacebookApi.class);
```

2.3 Persisting connections

After a connection has been established, you may wish to persist it for later use. This makes things convenient for the user since a connection can simply be restored from its persistent form and does not need to be established again. Spring Social provides a ConnectionRepository interface for managing the persistence of a user's connections:

```
public interface ConnectionRepository {

    MultiValueMap<String, Connection<?>> findConnections();

    List<Connection<?>> findConnectionsToProvider(String providerId);

    MultiValueMap<String, Connection<?>> findConnectionsForUsers(MultiValueMap<String, String> providerUserIds

    Connection<?> findConnection(ConnectionKey connectionKey);

    <A> Connection<A> findPrimaryConnectionToApi(Class<A> apiType);

    <A> Connection<A> findConnectionToApiForUser(Class<A> apiType, String providerUserId);

    <A> List<Connection<A>> findConnectionsToApi(Class<A> apiType);

    void addConnection(Connection<?> connection);

    void updateConnection(Connection<?> connection);

    void removeConnectionsToProvider(String providerId);

    void removeConnection(ConnectionKey connectionKey);

}
```

As you can see, this interface provides a number of operations for adding, updating, removing, and finding Connections. Consult the JavaDoc API of this interface for a full description of these operations. Note that all operations on this repository are scoped relative to the "current user" that has authenticated with your local application. For standalone, desktop, or mobile environments that only have one user this distinction isn't important. In a multi-user web application environment, this implies ConnectionRepository instances will be request-scoped.

For multi-user environments, Spring Social provides a UsersConnectionRepository that provides access to the global store of connections across all users:

```
public interface UsersConnectionRepository {

    String findUserIdWithConnection(Connection connection);

    Set<String> findUserIdsConnectedTo(String providerId, Set<String> providerUserIds);

    ConnectionRepository createConnectionRepository(String userId);

}
```

As you can see, this repository acts as a factory for ConnectionRepository instances scoped to a single user, as well as exposes a number of multi-user operations. These operations include the ability to lookup the local userIds associated with connections to support provider user sign-in and "registered friends" scenarios. Consult the JavaDoc API of this interface for a full description.

JDBC-based persistence

Spring Social provides a JdbcUsersConnectionRepository implementation capable of persisting connections to a RDBMS. The database schema designed to back this repository is defined in JdbcUsersConnectionRepository.sql. The implementation also provides support for encrypting authorization credentials so they are not stored in plain-text.

The example code below demonstrates construction and usage of a JdbcUsersConnectionRepository:

```
// JDBC DataSource pointing to the DB where connection data is stored
DataSource dataSource = ...;
// locator for factories needed to construct Connections when restoring from persistent form
ConnectionFactoryLocator connectionFactoryLocator = ...;
// encryptor of connection authorization credentials
TextEncryptor encryptor = ...;

UsersConnectionRepository usersConnectionRepository =
    new JdbcUsersConnectionRepository(dataSource, connectionFactoryLocator, encryptor);

// create a connection repository for the single-user 'kdonald'
ConnectionRepository repository = usersConnectionRepository.createConnectionRepository("kdonald");

// find kdonald's primary Facebook connection
Connection<FacebookApi> connection = repository.findPrimaryConnectionToApi(FacebookApi.class);
```

3. Adding Support for a New Service Provider

Spring Social makes it easy to add support for service providers that are not already supported by the framework. If you review the existing client modules, such as spring-social-twitter and spring-social-facebook, you will discover they are implemented in a consistent manner and they apply a set of well-defined extension points. In this chapter, you will learn how to add support for new service providers you wish to integrate into your applications.

3.1 Process overview

The process of adding support for a new service provider consists of several steps:

1. Create a source project for the client code e.g. `spring-social-twitter`.

2. Develop or integrate a Java binding to the provider's API e.g. `TwitterApi`.

3. Create a ServiceProvider model that allows users to authorize with the remote provider and obtain authorized API instances e.g. `TwitterServiceProvider`.

4. Create an ApiAdapter that maps the provider's native API onto the uniform Connection model e.g. `TwitterApiAdapter`.

5. Finally, create a ConnectionFactory that wraps the other artifacts up and provides a simple interface for establishing connections e.g. `TwitterConnectionFactory`.

The following sections of this chapter walk you through each of the steps with examples.

3.2 Creating a source project for the provider client code

A Spring Social client module is a standard Java project that builds a single jar artifact e.g. spring-social-twitter.jar. We recommend the code structure of a client module follow the guidelines described below.

Code structure guidelines

We recommend the code for a new Spring Social client module reside within the `org.springframework.social.{providerId}` base package, where {providerId} is a unique identifier you assign to the service provider you are adding support for. Consider some of the providers already supported by the framework as examples:

Table 3.1. Spring Social Client Modules

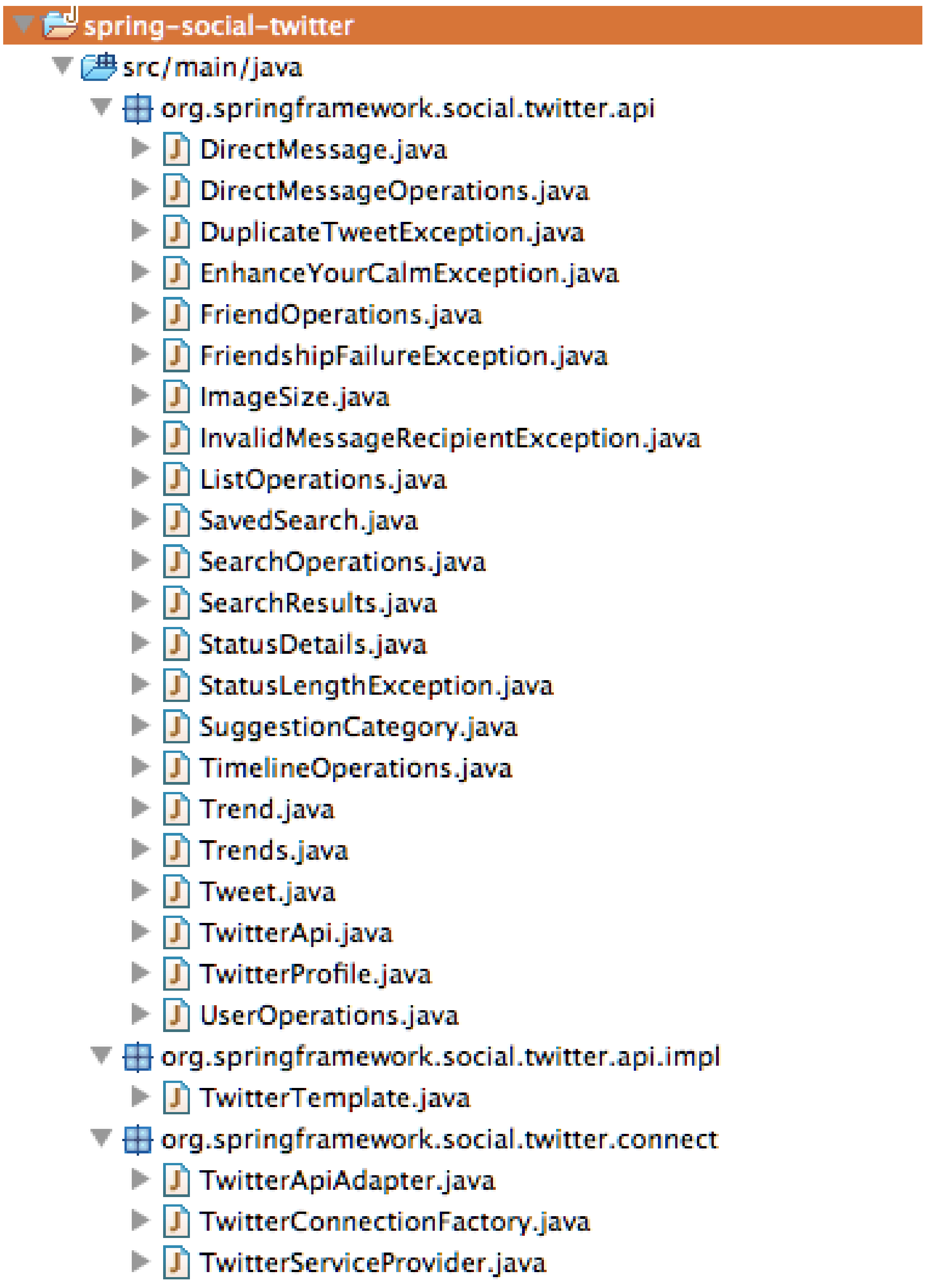
Provider ID	Artifact Name	Base Package
facebook	spring-social-facebook	org.springframework.social.facebook
twitter	spring-social-twitter	org.springframework.social.twitter

Within the base package, we recommend the following subpackage structure:

Table 3.2. Module Structure

Subpackage	Description
api	The public interface that defines the API binding.
api.impl	The implementation of the API binding.
connect	The types necessary to establish connections to the service provider.

You can see this recommended structure in action by reviewing one of the other client modules such as spring-social-twitter:



```
spring-social-twitter
  src/main/java
    org.springframework.social.twitter.api
      DirectMessage.java
      DirectMessageOperations.java
      DuplicateTweetException.java
      EnhanceYourCalmException.java
      FriendOperations.java
      FriendshipFailureException.java
      ImageSize.java
      InvalidMessageRecipientException.java
      ListOperations.java
      SavedSearch.java
      SearchOperations.java
      SearchResults.java
      StatusDetails.java
      StatusLengthException.java
      SuggestionCategory.java
      TimelineOperations.java
      Trend.java
      Trends.java
      Tweet.java
      TwitterApi.java
      TwitterProfile.java
      UserOperations.java
    org.springframework.social.twitter.api.impl
      TwitterTemplate.java
    org.springframework.social.twitter.connect
      TwitterApiAdapter.java
      TwitterConnectionFactory.java
      TwitterServiceProvider.java
```

Here, the central service API type, TwitterApi, is located in the api package along with its supporting operations types and data transfer object types. The primary implementation of that interface, TwitterTemplate, is located in the api.impl package (along with other package-private impl types have that been excluded from this view). Finally, the connect package contains the implementations of various connect SPIs that enable connections to Twitter to be established and persisted.

3.3 Developing a Java binding to the provider's API

Spring Social favors the development of strongly-typed Java bindings to external service provider APIs. This provides a simple, domain-oriented interface for Java applications to use to consume the API. When adding support for a new service provider, if no suitable Java binding already exists you'll need to develop one. If one already exists, such as Twitter4j for example, it is possible to integrate it into the framework.

Designing a new Java API binding

API developers retain full control over the design and implementation of their Java bindings. That said, we offer several design guidelines in an effort to improve overall consistency and quality:

- *Favor separating the API binding interface from the implementation.* This is illustrated in the spring-social-twitter example in the previous section. There, "TwitterApi" is the central API binding type and it is declared in the org.springframework.social.twitter.api package with other public types. "TwitterTemplate" is the primary implementation of this interface and is located in the org.springframework.social.twitter.api.impl subpackage along with other package-private implementation types.

- *Favor organizing the API binding hierarchically by RESTful resource.* REST-based APIs typically expose access to a number of resources in an hierarchical manner. For example, Twitter's API provides access to "status timelines", "searches", "lists", "direct messages", "friends", and "users". Rather than add all operations across these resources to a single flat "TwitterApi" interface, the TwitterApi interface is organized hierarchically:

  ```
  public interface TwitterApi {

      DirectMessageOperations directMessageOperations();

      FriendOperations friendOperations();

      ListOperations listOperations();

      SearchOperations searchOperations();

      TimelineOperations timelineOperations();

      UserOperations userOperations();

  }
  ```

  DirectMessageOperations, for example, contains API bindings to Twitter's "direct_messages" resource:

  ```
  public interface DirectMessageOperations {
  ```

```
    List<DirectMessage> getDirectMessagesReceived();

    List<DirectMessage> getDirectMessagesSent();

    void sendDirectMessage(String toScreenName, String text);

    void sendDirectMessage(long toUserId, String text);

    void deleteDirectMessage(long messageId);
}
```

Implementing a new Java API binding

API developers are free to implement their Java API binding with whatever REST/HTTP client they see fit. That said, Spring Social's existing API bindings such as spring-social-twitter all use Spring Framework's RestTemplate in conjunction with the Jackson JSON ObjectMapper and Apache HttpComponents HTTP client. RestTemplate is a popular REST client that provides a uniform object mapping interface across a variety of data exchange formats (JSON, XML, etc). Jackson is the leading Java-based JSON marshalling technology. Apache HttpComponents has proven to be the most robust HTTP client (if it is not available on the classpath Spring Social will fallback to standard J2SE facilities, however). To help promote consistency across Spring Social's supported bindings, we do recommend you consider these implementation technologies (and please let us know if they do not meet your needs).

Spring Social has adopted a convention where each API implementation class is named "{ProviderId}Template" e.g. TwitterTemplate. We favor this convention unless there is a good reason to deviate from it. As discussed in the previous section, we recommend keeping implementation types separate from the public API types. We also recommend keeping internal implementation details package-private.

The way in which an API binding implementation is constructed will vary based on the API's authorization protocol. For APIs secured with OAuth1, the consumerKey, consumerSecret, accessToken, and accessTokenSecret will be required for construction:

```
public TwitterTemplate(String consumerKey, String consumerSecret, String accessToken, String accessTokenSecret
```

For OAuth2, only the access token should be required:

```
public FacebookTemplate(String accessToken) { ... }
```

Each request made to the API server needs to be signed with the authorization credentials provided during construction of the binding. This signing process consists of adding an "Authorization" header to each client request before it is executed. For OAuth1, the process is quite complicated, and is used to support an elaborate request signature verification algorithm between the client and server. For OAuth2, it is a lot simpler, but does still vary across the various drafts of the OAuth2 specification.

To encapsulate this complexity, for each authorization protocol Spring Social provides a ProtectedResourceClientFactory you may use to construct a pre-configured RestTemplate instance that performs the request signing for you. For OAuth1:

```
public TwitterTemplate(String consumerKey, String consumerSecret, String accessToken, String accessTokenSecret)
    this.restTemplate = ProtectedResourceClientFactory.create(consumerKey, consumerSecret, accessToken, access
}
```

An OAuth2 example:

```
public FacebookTemplate(String accessToken) {
    this.restTemplate = ProtectedResourceClientFactory.draft10(accessToken);
}
```

Once the REST client has been configured as shown above, you simply use it to implement the various API operations. The existing Spring Social client modules all invoke their RestTemplate instances in a standard manner:

```
public TwitterProfile getUserProfile() {
    return restTemplate.getForObject(buildUri("account/verify_credentials.json"), TwitterProfile.class);
}
```

A note on RestTemplate usage: we do favor the RestTemplate methods that accept a URI object instead of a uri String. This ensures we always properly encode client data submitted in URI query parameters, such as screen_name below:

```
public TwitterProfile getUserProfile(String screenName) {
    return restTemplate.getForObject(buildUri("users/show.json", Collections.singletonMap("screen_name", screen
}
```

For complete implementation examples, consult the source of the existing API bindings included in Spring Social. The `spring-social-twitter` and `spring-social-facebook` modules provide particularly good references.

Testing a new Java API binding

As part of the spring-social-test module, Spring Social includes a framework for unit testing API bindings. This framework consists of a "MockRestServiceServer" that can be used to mock out API calls to the remote service provider. This allows for the development of independent, performant, automated unit tests that verify client API binding and object mapping behavior.

To use, first create a MockRestServiceServer against the RestTemplate instance used by your API implementation:

```
TwitterTemplate twitter = new TwitterTemplate("consumerKey", "consumerSecret", "accessToken", "accessTokenSecre
MockRestServer mockServer = MockRestServiceServer.createServer(twitter.getRestTemplate());
```

Then, for each test case, record expectations about how the server should be invoked and answer what it should respond with:

```
@Test
public void getUserProfile() {
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setContentType(MediaType.APPLICATION_JSON);

    mockServer.expect(requestTo("https://api.twitter.com/1/account/verify_credentials.json"))
        .andExpect(method(GET))
        .andRespond(withResponse(jsonResource("verify-credentials"), responseHeaders));

    TwitterProfile profile = twitter.userOperations().getUserProfile();
    assertEquals(161064614, profile.getId());
    assertEquals("kdonald", profile.getScreenName());
}
```

In the example above the response body is written from a verify-credentials.json file located in the same package as the test class:

```
private Resource jsonResource(String filename) {
    return new ClassPathResource(filename + ".json", getClass());
}
```

The content of the file should mirror the content the remote service provider would return, allowing the client JSON deserialization behavior to be fully tested:

```
{
    "id":161064614,
    "screen_name":"kdonald"
}
```

For complete test examples, consult the source of the existing API bindings included in Spring Social. The `spring-social-twitter` and `spring-social-facebook` modules provide particularly good references.

Integrating an existing Java API binding

If you are adding support for a popular service provider, chances are a Java binding to the provider's API may already exist. For example, the Twitter4j library has been around for awhile and provides a complete binding to Twitter's API. Instead of developing your own binding, you may simply wish to integrate what already exists. Spring Social's connect framework has been carefully designed to support this scenario.

To integrate an existing API binding, simply note the binding's primary API interface and implementation. For example, in Twitter4j the main API interface is named "Twitter" and instances are constructed by a TwitterFactory. You can always construct such an API instance directly, and you'll see in the following sections how to expose an instance as part of a Connection.

3.4 Creating a ServiceProvider model

As described in the previous section, a client binding to a secure API such as Facebook or Twitter requires valid user authorization credentials to work. Such credentials are generally obtained by having your application conduct an authorization "dance" or handshake with the service provider. Spring Social provides the ServiceProvider<A> abstraction to handle this "authorization dance". The abstraction also acts as a factory for native API (A) instances.

Since the authorization dance is protocol-specific, a ServiceProvider specialization exists for each authorization protocol. For example, if you are connecting to a OAuth2-based provider, you would implement OAuth2ServiceProvider. After you've done this, your implementation can be used to conduct the OAuth2 dance and obtain an authorized API instance. This is typically done in the context of a ConnectionFactory as part of establishing a new connection to the provider. The following sections describe the implementation steps for each ServiceProvider type.

OAuth2

To implement an OAuth2-based ServiceProvider, first create a subclass of AbstractOAuth2ServiceProvider named {ProviderId}ServiceProvider. Parameterize <A> to be the Java binding to the ServiceProvider's's API. Define a single constructor that accepts an clientId and clientSecret. Finally, implement getApi(String) to return a new API instance.

See `org.springframework.social.facebook.connect.FacebookServiceProvider` as an example OAuth2ServiceProvider:

```
public final class FacebookServiceProvider extends AbstractOAuth2ServiceProvider<FacebookApi> {

    public FacebookServiceProvider(String clientId, String clientSecret) {
        super(new OAuth2Template(clientId, clientSecret,
           "https://graph.facebook.com/oauth/authorize",
           "https://graph.facebook.com/oauth/access_token"));
    }

    public FacebookApi geteApi(String accessToken) {
        return new FacebookTemplate(accessToken);
    }

}
```

In the constructor, you should call super, passing up the configured OAuth2Template that implements OAuth2Operations. The OAuth2Template will handle the "OAuth dance" with the provider, and should be configured with the provided clientId and clientSecret, along with the provider-specific authorizeUrl and accessTokenUrl.

In getApi(String), you should construct your API implementation, passing it the access token needed to make authorized requests for protected resources.

OAuth1

To implement an OAuth1-based ServiceProvider, first create a subclass of AbstractOAuth1ServiceProvider named {ProviderId}ServiceProvider. Parameterize <A> to be the Java binding to the ServiceProvider's API. Define a single constructor that accepts a consumerKey and consumerSecret. Finally, implement getApi(String, String) to return a new API instance.

See `org.springframework.social.twitter.connect.TwitterServiceProvider` as an example OAuth1ServiceProvider:

```
public final class TwitterServiceProvider extends AbstractOAuth1ServiceProvider<TwitterApi> {

    public TwitterServiceProvider(String consumerKey, String consumerSecret) {
        super(consumerKey, consumerSecret, new OAuth1Template(consumerKey, consumerSecret,
            "https://twitter.com/oauth/request_token",
            "https://twitter.com/oauth/authorize",
            "https://twitter.com/oauth/access_token"));
    }

    public TwitterApi getApi(String accessToken, String secret) {
        return new TwitterTemplate(getConsumerKey(), getConsumerSecret(), accessToken, secret);
    }

}
```

In the constructor, you should call super, passing up the the consumerKey, secret, and configured OAuth1Template. The OAuth1Template will handle the "OAuth dance" with the provider. It should be configured with the provided consumerKey and consumerSecret, along with the provider-specific requestTokenUrl, authorizeUrl, and accessTokenUrl.

In getApi(String, String), you should construct your API implementation, passing it the four tokens needed to make authorized requests for protected resources.

Consult the JavaDoc API of the various service provider types for more information and subclassing options.

3.5 Creating an ApiAdapter

As discussed in the previous chapter, one of the roles of a Connection is to provide a common abstraction for a linked user account that is applied across all service providers. The role of the ApiAdapter is to map a provider's native API interface onto this uniform Connection model. A connection delegates to its adapter to perform operations such as testing the validity of its API credentials, setting metadata values, fetching a user profile, and updating user status:

```
public interface ApiAdapter<A> {

    boolean test(A api);

    void setConnectionValues(A api, ConnectionValues values);

    UserProfile fetchUserProfile(A api);
    void updateStatus(A api, String message);

}
```

Consider `org.springframework.social.twitter.connect.TwitterApiAdapter` as an example implementation:

```
public class TwitterApiAdapter implements ApiAdapter<TwitterApi> {

    public boolean test(TwitterApi api) {
        try {
            api.userOperations().getUserProfile();
            return true;
        } catch (BadCredentialsException e) {
            return false;
        }
    }

    public void setConnectionValues(TwitterApi api, ConnectionValues values) {
        TwitterProfile profile = api.userOperations().getUserProfile();
        values.setProviderUserId(Long.toString(profile.getId()));
        values.setDisplayName("@" + profile.getScreenName());
        values.setProfileUrl(profile.getProfileUrl());
        values.setImageUrl(profile.getProfileImageUrl());
    }

    public UserProfile fetchUserProfile(TwitterApi api) {
        TwitterProfile profile = api.userOperations().getUserProfile();
        return new UserProfileBuilder().setName(profile.getName()).setUsername(profile.getScreenName()).build(
    }

    public void updateStatus(TwitterApi api, String message) {
        api.timelineOperations().updateStatus(message);
    }

}
```

As you can see, test(...) returns true if the API instance is functional and false if it is not. setConnectionValues(...) sets the connection's providerUserId, displayName, profileUrl, and imageUrl properties from TwitterProfile data. fetchUserProfile(...) maps a TwitterProfile onto the normalized UserProfile model. updateStatus(...) update's the user's Twitter status. Consult the JavaDoc for ApiAdapter and Connection for more information and implementation guidance. We also recommend reviewing the other ApiAdapter implementations for additional examples.

3.6 Creating a ConnectionFactory

By now, you should have an API binding to the provider's API, a ServiceProvider<A> implementation for conducting the "authorization dance", and an ApiAdapter<A> implementation for mapping onto the uniform Connection model. The last step in adding support for a new service provider is to create a ConnectionFactory that wraps up these artifacts and provides a simple interface for establishing Connections. After this is done, you may use your connection factory directly, or you may add it to a registry where it can be used by the framework to establish connections in a dynamic, self-service manner.

Like a ServiceProvider<A>, a ConnectionFactory specialization exists for each authorization protocol. For example, if you are adding support for a OAuth2-based provider, you would extend from OAuth2ConnectionFactory. Implementation guidelines for each type are provided below.

OAuth2

Create a subclass of OAuth2ConnectionFactory<A> named {ProviderId}ConnectionFactory and parameterize A to be the Java binding to the service provider's API. Define a single constructor that accepts a clientId and clientSecret. Within the constructor call super, passing up the assigned providerId, a new {ProviderId}ServiceProvider instance configured with the clientId/clientSecret, and a new {Provider}ApiAdapter instance.

See `org.springframework.social.facebook.connect.FacebookConnectionFactory` as an example OAuth2ConnectionFactory:

```
public class FacebookConnectionFactory extends OAuth2ConnectionFactory<FacebookApi> {
    public FacebookConnectionFactory(String clientId, String clientSecret) {
        super("facebook", new FacebookServiceProvider(clientId, clientSecret), new FacebookApiAdapter());
    }
}
```

OAuth1

Create a subclass of OAuth1ConnectionFactory<A> named {ProviderId}ConnectionFactory and parameterize A to be the Java binding to the service provider's API. Define a single constructor that accepts a consumerKey and consumerSecret. Within the constructor call super, passing up the assigned providerId, a new {ProviderId}ServiceProvider instance configured with the consumerKey/consumerSecret, and a new {Provider}ApiAdapter instance.

See `org.springframework.social.twitter.connect.TwitterConnectionFactory` as an example OAuth1ConnectionFactory:

```
public class TwitterConnectionFactory extends OAuth1ConnectionFactory<FacebookApi> {
    public TwitterConnectionFactory(String consumerKey, String consumerSecret) {
        super("twitter", new TwitterServiceProvider(consumerKey, consumerSecret), new TwitterApiAdapter());
    }
}
```

Consult the source and JavaDoc API for ConnectionFactory and its subclasses more information, examples, and advanced customization options.

4. Connecting to Service Providers

4.1 Introduction

In Chapter 2, *Service Provider 'Connect' Framework*, you learned how Spring Social's *Service Provider 'Connect' Framework* can be used to manage user connections that link your application's user accounts with accounts on external service providers. In this chapter, you'll learn how to control the connect flow in a web application environment.

Spring Social's `spring-social-web` module includes `ConnectController`, a Spring MVC controller that coordinates the connection flow between an application and service providers. `ConnectController` takes care of redirecting the user to the service provider for authorization and responding to the callback after authorization.

4.2 Configuring ConnectController

As `ConnectController` directs the connection flow, it depends on a couple of other objects to assist in the creation and persistence of connections. `ConnectController` works with one or more `ConnectionFactory`s to exchange authorization details with the provider and to create connections. Once a connection has been established, `ConnectController` hands it off to a `ConnectionRepository` to be persisted.

Spring Social comes with an implementation of `ConnectionFactory` for each of the supported service providers:

- `TwitterConnectionFactory`
- `FacebookConnectionFactory`
- `LinkedInConnectionFactory`
- `TripItConnectionFactory`
- `GitHubConnectionFactory`
- `GowallaConnectionFactory`

`ConnectController` relies on an implementation of `ConnectionFactoryLocator` (see the section called “Registering ConnectionFactory instances”) to find a connection factory for a specific provider. Spring Social's `ConnectionFactoryRegistry` is an implementation of `ConnectionFactoryLocator` that keeps a Map-based registry of connection factories. The following class constructs a `ConnectionFactoryRegistry` containing `ConnectionFactory`s for Twitter, Facebook, and TripIt using Spring's Java configuration style:

```
@Configuration
public class ConnectionFactoryConfig {
    @Bean
    public ConnectionFactoryLocator connectionFactoryLocator() {
        ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();
        registry.addConnectionFactory(new TwitterConnectionFactory(twitterConsumerKey, twitterConsumerSecret));
        registry.addConnectionFactory(new FacebookConnectionFactory(facebookClientId, facebookClientSecret));
        registry.addConnectionFactory(new TripItConnectionFactory(tripItConsumerKey, tripItConsumerSecret));
        return registry;
    }

    @Value("${twitter.consumerKey}")
    private String twitterConsumerKey;

    @Value("${twitter.consumerSecret}")
    private String twitterConsumerSecret;

    @Value("${facebook.clientId}")
    private String facebookClientId;

    @Value("${facebook.clientSecret}")
    private String facebookClientSecret;

    @Value("${tripit.consumerKey}")
    private String tripItConsumerKey;

    @Value("${tripit.consumerSecret}")
    private String tripItConsumerSecret;

}
```

Here, three connection factories--one each for Twitter, Facebook, and TripIt--are registered with `ConnectionFactoryRegistry` via the `addConnectionFactory()` method. If we wanted to add support for connecting to other providers, we would simply register their connection factories here. Because consumer keys and secrets may be different across environments (e.g., test, production, etc) it is recommended that these values be externalized. Therefore, they are wired in with `@Value` as property placeholder values to be resolved by Spring's property placeholder support.

`ConnectController` will use the `ConnectionFactory`s that it obtains through `ConnectionFactoryLocator` to perform the authorization exchange with each provider and ultimately to create a connection. Once a connection has been created, it must be persisted and associated with the user's account. For that, `ConnectController` depends on a `ConnectionRepository`.

As discussed in Section 2.3, “Persisting connections”, `ConnectionRepository` defines operations for persisting and restoring connections for a specific user. Therefore, when configuring a `ConnectionRepository` bean, it must be scoped such that it can be created on a per-user basis. The following Java-based configuration shows how to configure `ConnectionRepository` bean in request scope for the currently authenticated user:

```
@Configuration
public class ConnectionRepositoryConfig {

    @Inject
    private UsersConnectionRepository usersConnectionRepository;

    @Bean
    @Scope(value="request")
    public ConnectionRepository connectionRepository(@Value("#{request.userPrincipal}") Principal principal) {
        return usersConnectionRepository.createConnectionRepository(principal.getName());
    }

}
```

The `connectionRepository()` method is injected with a `Principal` (pulled from the request with a Spring Expression Language expression). The `Principal` is passed to the `UsersConnectionRepository`'s `createConnectionRepository()` method to create a `ConnectionRepository` for the current user in the context of the current web request.

This means that we're also going to need to configure a `UsersConnectionRepository` bean. Here's one configured using Spring's Java configuration style:

```

@Configuration
public class UsersConnectionRepositoryConfig {

    @Bean
    public UsersConnectionRepository usersConnectionRepository(DataSource dataSource,
            ConnectionFactoryLocator connectionFactoryLocator, TextEncryptor textEncryptor) {
        return new JdbcUsersConnectionRepository(dataSource, connectionFactoryLocator, textEncryptor);
    }

}
```

`JdbcUsersConnectionRepository` is created with a references to a `DataSource`, a `ConnectionFactoryLocator`, and a text encryptor. It will use the `DataSource` to access the RDBMS when persisting and restoring connections. When restoring connections, it will use the `ConnectionFactoryLocator` to locate ConnectionFactory instances.

`JdbcUsersConnectionRepository` uses a `TextEncryptor` to encrypt the credentials (e.g., access tokens and secrets) obtained during authorization when writing them to the database. Spring Security 3.1 makes a few useful text encryptors available via static factory methods in its `Encryptors` class. For example, a no-op text encryptor is useful at development time and can be configured like this:

```

@Configuration
@Profile("dev")
public class DevEncryptionConfig {

    @Bean
    public TextEncryptor textEncryptor() {
        return Encryptors.noOpText();
    }

}
```

Notice that this configuration class is annotated with `@Profile("dev")`. Spring 3.1 introduced the profile concept where certain beans will only be created when certain profiles are active. Here, the `@Profile` annotation ensures that this `TextEncryptor` will only be created when "dev" is an active profile. For production-time purposes, a stronger text encryptor is recommended and can be created when the "production" profile is active:

```
@Configuration
@Profile("production")
public class ProductionEncryptionConfig {

    @Bean
    public TextEncryptor textEncryptor(@Value("${security.encryptPassword}") String password,
            @Value("${security.encryptSalt}") String salt) {
        return Encryptors.text(password, salt);
    }

}
```

Configuring connection support in XML

Up to this point, the connection support configuration has been done using Spring's Java-based configuration style. But you can configure it in either Java configuration or XML. Here's the XML equivalent of the `ConnectionFactoryRegistry` configuration:

```
<bean id="connectionFactoryLocator" class="org.springframework.social.connect.support.ConnectionFactoryRegistry
    <property name="connectionFactories">
        <list>
            <bean class="org.springframework.social.twitter.connect.TwitterConnectionFactory">
                <constructor-arg value="${twitter.consumerKey}" />
                <constructor-arg value="${twitter.consumerSecret}" />
            </bean>
            <bean class="org.springframework.social.facebook.connect.FacebookConnectionFactory">
                <constructor-arg value="${facebook.appId}" />
                <constructor-arg value="${facebook.appSecret}" />
            </bean>
            <bean class="org.springframework.social.tripit.connect.TripItConnectionFactory">
                <constructor-arg value="${tripit.consumerKey}" />
                <constructor-arg value="${tripit.consumerSecret}" />
            </bean>
        </list>
    </property>
</bean>
```

This is functionally equivalent to the Java-based configuration of `ConnectionFactoryRegistry` shown before. The only casual difference is that the connection factories are injected as a list into the `connectionFactories` property rather than with the `addConnectionFactory()` method.

Here's an XML equivalent of the `JdbcUsersConnectionRepository` and `ConnectionRepository` configurations shown before:

```
<bean id="usersConnectionRepository" class="org.springframework.social.connect.jdbc.JdbcUsersConnectionReposit
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="connectionFactoryLocator" />
    <constructor-arg ref="textEncryptor" />
</bean>
```

```
<bean id="connectionRepository" factory-method="createConnectionRepository" factory-bean="usersConnectionReposi
    <constructor-arg value="#{request.userPrincipal.name}" />
</bean>
```

Likewise, here is the equivalent configuration of the `TextEncryptor` beans:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    <beans profile="dev">
        <bean id="textEncryptor" class="org.springframework.security.crypto.encrypt.Encryptors" factory-method=
    </beans>

    <beans profile="production">
        <bean id="textEncryptor" class="org.springframework.security.crypto.encrypt.Encryptors" factory-method=
            <constructor-arg value="${security.encryptPassword}" />
            <constructor-arg value="${security.encryptSalt}" />
        </bean>
    </beans>

</beans>
```

Just like the Java-based configuration, profiles are used to select which of the text encryptors will be created.

4.3 Creating connections with `ConnectController`

With `ConnectController`'s dependencies configured, `ConnectController` will be able to coordinate the connection process for the service providers whose connection factories are registered with `ConnectionFactoryRegistry`. `ConnectController` is a Spring MVC controller and can be configured as a bean in your application's Spring MVC configuration as follows:

```
@Configuration
public class ConnectControllerConfig {

    @Bean
    public ConnectController connectController(@Value("${application.secureUrl}") String applicationUrl,
            ConnectionFactoryLocator connectionFactoryLocator, Provider<ConnectionRepository> connectionReposit
        return new ConnectController(applicationUrl, connectionFactoryLocator, connectionRepositoryProvider);
    }

}
```

Or, if you prefer Spring's XML-based configuration, then you can configure `ConnectController` like this:

```
<bean class="org.springframework.social.connect.web.ConnectController">
    <constructor-arg value="${application.secureUrl}" />
    <!-- relies on by-type autowiring for the other constructor-args -->
```

```
</bean>
```

In either case, `ConnectController` is constructed with the base URL for the application. `ConnectController` will use this URL to construct callback URLs used in the authorization flow. Since the base URL of an application will be different between environments, it is recommended that you externalize it. Here the URL is specified as a placeholder variable.

`ConnectController` supports authorization flows for either OAuth 1 or OAuth 2, relying on `OAuth1Operations` or `OAuth2Operations` to handle the specifics for each protocol. `ConnectController` will obtain the appropriate OAuth operations interface from one of the provider connection factories registered with `ConnectionFactoryRegistry`. It will select a specific `ConnectionFactory` to use by matching the connection factory's ID with the URL path. The path pattern that `ConnectController` handles is "/connect/{providerId}". Therefore, if `ConnectController` is handling a request for "/connect/twitter", then the `ConnectionFactory` whose `getProviderId()` returns "twitter" will be used. (As configured in the previous section, `TwitterConnectionFactory` will be chosen.)

The flow that `ConnectController` follows is slightly different, depending on which authorization protocol is supported by the service provider. For OAuth 2-based providers, the flow is as follows:

- `GET /connect/{providerId}` - Displays a web page showing connection status to the provider.
- `POST /connect/{providerId}` - Initiates the connection flow with the provider.
- `GET /connect/{providerId}?code={code}` - Receives the authorization callback from the provider, accepting an authorization code. Uses the code to request an access token and complete the connection.
- `DELETE /connect/{providerId}` - Severs a connection with the provider.

For an OAuth 1 provider, the flow is very similar, with only a subtle difference in how the callback is handled:

- `GET /connect/{providerId}` - Displays a web page showing connection status to the provider.
- `POST /connect/{providerId}` - Initiates the connection flow with the provider.
- `GET /connect/{providerId}?oauth_token={request token}&oauth_verifier={verifier}` - Receives the authorization callback from the provider, accepting a verification code. Exchanges this verification code along with the request token for an access token and completes the connection. The `oauth_verifier` parameter is optional and is only used for providers implementing OAuth 1.0a.
- `DELETE /connect/{providerId}` - Severs a connection with the provider.

Displaying a connection page

Before the connection flow starts in earnest, a web application may choose to show a page that offers the user information on their connection status. This page would offer them the opportunity to create a connection between their account and their social profile. `ConnectController` can display such a page if the browser navigates to `/connect/{provider}`.

For example, to display a connection status page for Twitter, where the provider name is "twitter", your application should provide a link similar to this:

```
<a href="<c:url value="/connect/twitter" />">Connect to Twitter</a>
```

`ConnectController` will respond to this request by first checking to see if a connection already exists between the user's account and Twitter. If not, then it will with a view that should offer the user an opportunity to create the connection. Otherwise, it will respond with a view to inform the user that a connection already exists.

The view names that `ConnectController` responds with are based on the provider's name. In this case, since the provider name is "twitter", the view names are "connect/twitterConnect" and "connect/twitterConnected".

Initiating the connection flow

To kick off the connection flow, the application should `POST` to `/connect/{providerId}`. Continuing with the Twitter example, a JSP view resolved from "connect/twitterConnect" might include the following form:

```
<form action="<c:url value="/connect/twitter" />" method="POST">
    <p>You haven't created any connections with Twitter yet. Click the button to create
       a connection between your account and your Twitter profile.
       (You'll be redirected to Twitter where you'll be asked to authorize the connection.)</p>
    <p><button type="submit"><img src="<c:url value="/resources/social/twitter/signin.png" />"/></button></p>
</form>
```

When `ConnectController` handles the request, it will redirect the browser to the provider's authorization page. In the case of an OAuth 1 provider, it will first fetch a request token from the provider and pass it along as a parameter to the authorization page. Request tokens aren't used in OAuth 2, however, so instead it passes the application's client ID and redirect URI as parameters to the authorization page.

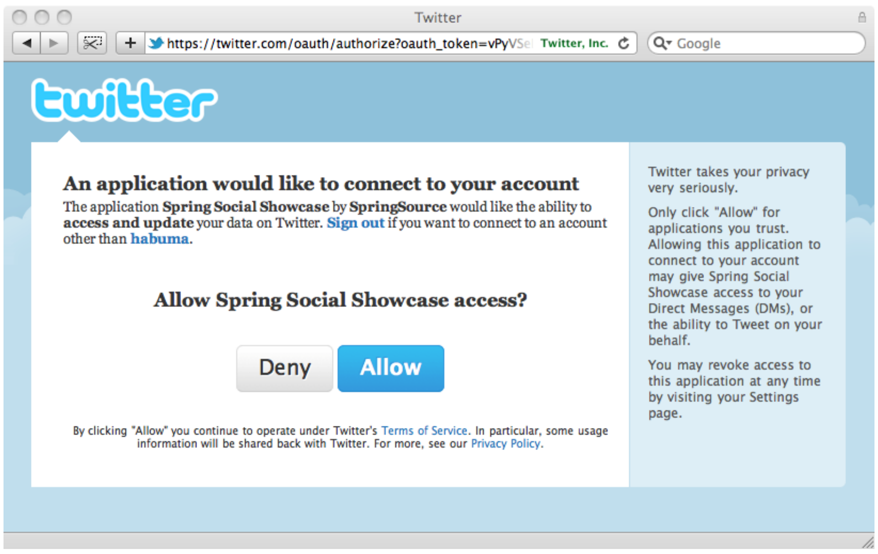
For example, Twitter's authorization URL has the following pattern:

```
https://twitter.com/oauth/authorize?oauth_token={token}
```

If the application's request token were "vPyVSe"[1], then the browser would be redirected to https://twitter.com/oauth/authorize?oauth_token=vPyVSe and a page similar to the following would be displayed to the user (from Twitter)[2]:

[1]This is just an example. Actual request tokens are typically much longer.

[2]If the user has not yet signed into Twitter, the authorization page will also include a username and password field for authentication into Twitter.



An application would like to connect to your account

The application **Spring Social Showcase** by **SpringSource** would like the ability to **access and update** your data on Twitter. Sign out if you want to connect to an account other than habuma.

Allow Spring Social Showcase access?

Deny Allow

By clicking "Allow" you continue to operate under Twitter's Terms of Service. In particular, some usage information will be shared back with Twitter. For more, see our Privacy Policy.

Twitter takes your privacy very seriously.

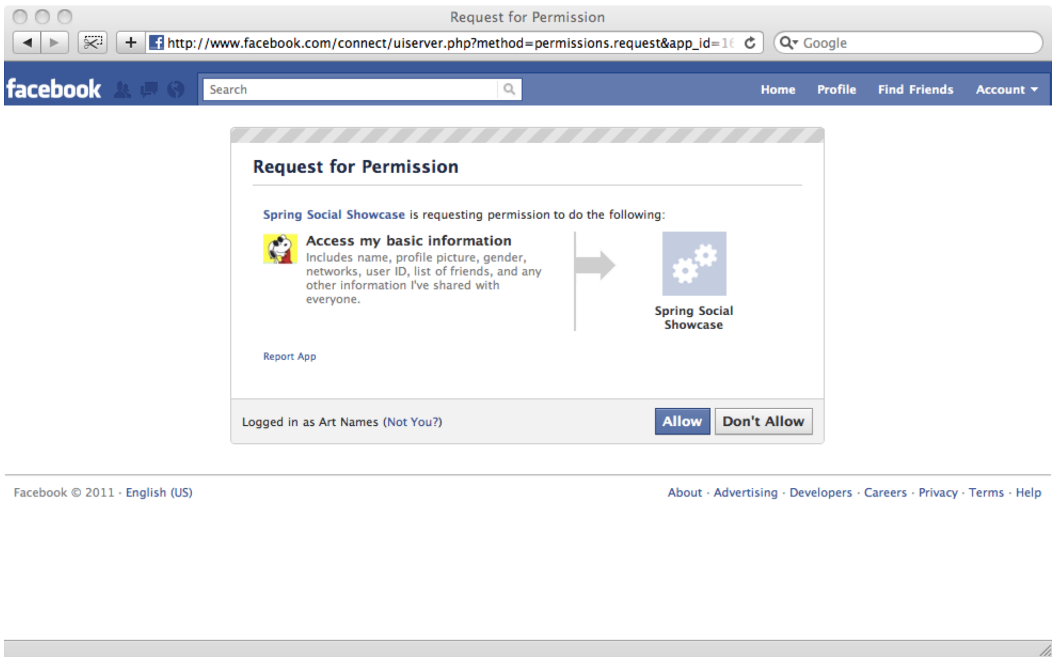
Only click "Allow" for applications you trust. Allowing this application to connect to your account may give Spring Social Showcase access to your Direct Messages (DMs), or the ability to Tweet on your behalf.

You may revoke access to this application at any time by visiting your Settings page.

In contrast, Facebook is an OAuth 2 provider, so its authorization URL takes a slightly different pattern:

```
https://graph.facebook.com/oauth/authorize?client_id={clientId}&redirect_uri={redirectUri}
```

Thus, if the application's Facebook client ID is "0b754" and it's redirect URI is "http://www.mycoolapp.com/connect/facebook", then the browser would be redirected to https://graph.facebook.com/oauth/authorize?client_id=0b754&redirect_uri=http://www.mycoolapp.com/connect/facebook and Facebook would display the following authorization page to the user:


Request for Permission
http://www.facebook.com/connect/uiserver.php?method=permissions.request&app_id=1
Google
facebook
Search
Home
Profile
Find Friends
Account

Request for Permission

Spring Social Showcase is requesting permission to do the following:

Access my basic information
Includes name, profile picture, gender, networks, user ID, list of friends, and any other information I've shared with everyone.

Spring Social Showcase

Report App

Logged in as Art Names (Not You?) Allow Don't Allow

Facebook © 2011 · English (US)

About · Advertising · Developers · Careers · Privacy · Terms · Help

If the user clicks the "Allow" button to authorize access, the provider will redirect the browser back to the authorization callback URL where `ConnectController` will be waiting to complete the connection.

The behavior varies from provider to provider when the user denies the authorization. For instance, Twitter will simply show a page telling the user that they denied the application access and does not redirect back to the application's callback URL. Facebook, on the other hand, will redirect back to the callback URL with error information as request parameters.

Authorization scope

In the previous example of authorizing an application to interact with a user's Facebook profile, you notice that the application is only requesting access to the user's basic profile information. But there's much more that an application can do on behalf of a user with Facebook than simply harvest their profile data. For example, how can an application gain authorization to post to a user's Facebook wall?

OAuth 2 authorization may optionally include a scope parameter that indicates the type of authorization being requested. On the provider, the "scope" parameter should be passed along to the authorization URL. In the case of Facebook, that means that the Facebook authorization URL pattern should be as follows:

```
https://graph.facebook.com/oauth/authorize?client_id={clientId}&redirect_uri={redirectUri}&scope={scope}
```

`ConnectController` accepts a "scope" parameter at authorization and passes its value along to the provider's authorization URL. For example, to request permission to post to a user's Facebook wall, the connect form might look like this:

```
<form action="<c:url value="/connect/twitter" />" method="POST">
    <input type="hidden" name="scope" value="publish_stream,offline_access" />
    <p>You haven't created any connections with Twitter yet. Click the button to create
       a connection between your account and your Twitter profile.
       (You'll be redirected to Twitter where you'll be asked to authorize the connection.)</p>
    <p><button type="submit"><img src="<c:url value="/resources/social/twitter/signin.png" />"/></button></p>
</form>
```

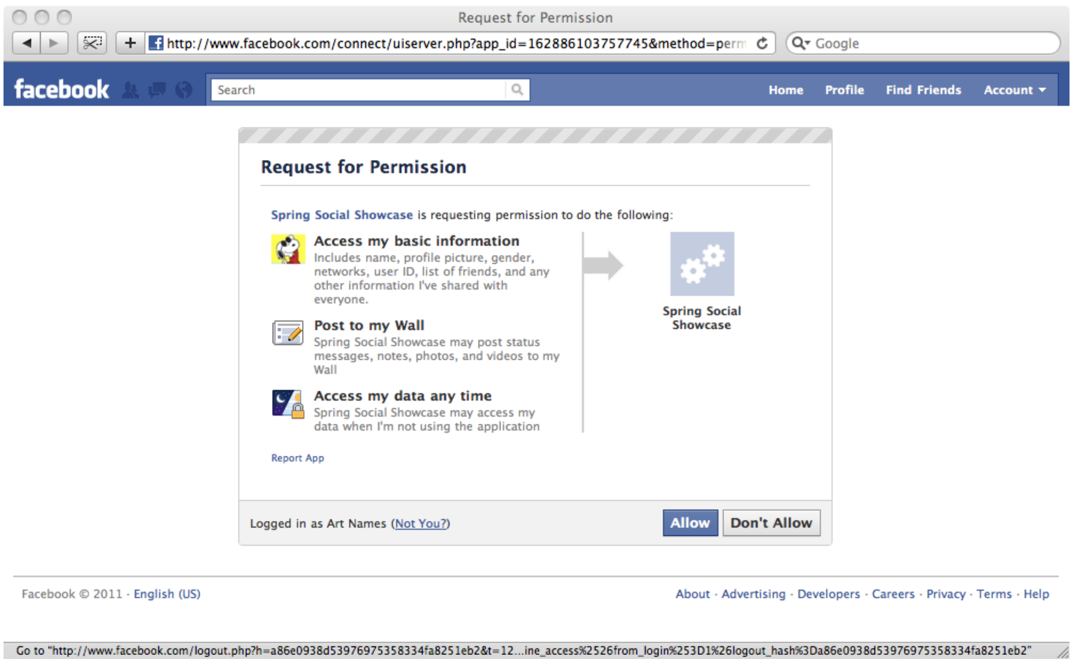
The hidden "scope" field contains the scope values to be passed along in the `scope>` parameter to Facebook's authorization URL. In this case, "publish_stream" requests permission to post to a user's wall. In addition, "offline_access" requests permission to access Facebook on behalf of a user even when the user isn't using the application.

Note

OAuth 2 access tokens typically expire after some period of time. Per the OAuth 2 specification, an application may continue accessing a provider after a token expires by using a refresh token to either renew an expired access token or receive a new access token (all without troubling the user to re-authorize the application).

Facebook does not currently support refresh tokens. Moreover, Facebook access tokens expire after about 2 hours. So, to avoid having to ask your users to re-authorize ever 2 hours, the best way to keep a long-lived access token is to request "offline_access".

When asking for "publish_stream,offline_access" authorization, the user will be prompted with the following authorization page from Facebook:


Request for Permission

http://www.facebook.com/connect/uiserver.php?app_id=162886103757745&method=perm

Request for Permission

Spring Social Showcase is requesting permission to do the following:

Access my basic information
Includes name, profile picture, gender, networks, user ID, list of friends, and any other information I've shared with everyone.

Post to my Wall
Spring Social Showcase may post status messages, notes, photos, and videos to my Wall

Access my data any time
Spring Social Showcase may access my data when I'm not using the application

Report App

Logged in as Art Names (Not You?)

Facebook © 2011 · English (US)

Go to "http://www.facebook.com/logout.php?h=a86e0938d53976975358334fa8251eb2&t=12...ine_access%2526from_login%253D1%26logout_hash%3Da86e0938d53976975358334fa8251eb2"

Scope values are provider-specific, so check with the service provider's documentation for the available scopes. Facebook scopes are documented at http://developers.facebook.com/docs/authentication/permissions.

Responding to the authorization callback

After the user agrees to allow the application have access to their profile on the provider, the provider will redirect their browser back to the application's authorization URL with a code that can be exchanged for an access token. For OAuth 1.0a providers, the callback URL is expected to receive the code (known as a verifier in OAuth 1 terms) in an `oauth_verifier` parameter. For OAuth 2, the code will be in a `code` parameter.

`ConnectController` will handle the callback request and trade in the verifier/code for an access token. Once the access token has been received, the OAuth dance is complete and the application may use the access token to interact with the provider on behalf of the user. The last thing that `ConnectController` does is to hand off the access token to the `ServiceProvider` implementation to be stored for future use.

Disconnecting

To delete a connection via `ConnectController`, submit a DELETE request to "/connect/{provider}".

In order to support this through a form in a web browser, you'll need to have Spring's `HiddenHttpMethodFilter` [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/filter/HiddenHttpMethodFilter.html] configured in your application's web.xml. Then you can provide a disconnect button via a form like this:

```
<form action="<c:url value="/connect/twitter" />" method="post">
   <div class="formInfo">
      <p>Spring Social Showcase is connected to your Twitter account.
         Click the button if you wish to disconnect.</p>
   </div>
   <button type="submit">Disconnect</button>
   <input type="hidden" name="_method" value="delete" />
```

```
</form>
```

When this form is submitted, `ConnectController` will disconnect the user's account from the provider. It does this by calling the `disconnect()` method on each of the `Connection`s returned by the provider's `getConnections()` method.

4.4 Connection interceptors

In the course of creating a connection with a service provider, you may want to inject additional functionality into the connection flow. For instance, perhaps you'd like to automatically post a tweet to a user's Twitter timeline immediately upon creating the connection.

`ConnectController` may be configured with one or more connection interceptors that it will call at points in the connection flow. These interceptors are defined by the `ConnectInterceptor` interface:

```
public interface ConnectInterceptor<A> {

    void preConnect(ConnectionFactory<A> connectionFactory, WebRequest request);

    void postConnect(Connection<A> connection, WebRequest request);

}
```

The `preConnect()` method will be called by `ConnectController` just before redirecting the browser to the provider's authorization page. `postConnect()` will be called immediately after a connection has been persisted linking the user's local account with the provider profile.

For example, suppose that after connecting a user account with their Twitter profile , you want to immediately post a tweet about that connection to the user's Twitter timeline. To accomplish that, you might write the following connection interceptor:

```
public class TweetAfterConnectInterceptor implements ConnectInterceptor<TwitterApi> {

    public void preConnect(ConnectionFactory<TwitterApi> provider, WebRequest request) {
        // nothing to do
    }

    public void postConnect(Connection<TwitterApi> connection, WebRequest request) {
        connection.updateStatus("I've connected with the Spring Social Showcase!");
    }
}
```

This interceptor can then be injected into `ConnectController` when it is created:

```
@Bean
public ConnectController connectController(@Value("${application.secureUrl}") applicationUrl,
        ConnectionFactoryLocator connectionFactoryLocator, Provider<ConnectionRepository> connectionRepositoryP
    ConnectController controller = new ConnectController(applicationUrl, connectionFactoryLocator, connectionRe
    controller.addInterceptor(new TweetAfterConnectInterceptor());
    return controller;
}
```

Or, as configured in XML:

```
<bean class="org.springframework.social.connect.web.ConnectController">
    <constructor-arg value="${application.secureUrl}" />
    <property name="interceptors">
        <list>
            <bean class="org.springframework.social.showcase.twitter.TweetAfterConnectInterceptor" />
        </list>
    </property>
</bean>
```

Note that the `interceptors` property is a list and can take as many interceptors as you'd like to wire into it. When it comes time for `ConnectController` to call into the interceptors, it will only invoke the interceptor methods for those interceptors whose service operations type matches the service provider's operations type. In the example given here, only connections made through a service provider whose operation type is `TwitterApi` will trigger the interceptor's methods.

5. Signing in with Service Provider Accounts

5.1 Introduction

In order to ease sign in for their users, many applications allow sign in with a service provider such as Twitter or Facebook. With this authentication technique, the user signs into (or may already be signed into) his or her provider account. The application then tries to match that provider account to a local user account. If a match is found, the user is automatically signed into the application.

Spring Social supports such service provider-based authentication with `ProviderSignInController` from the `spring-social-web` module. `ProviderSignInController` works very much like `ConnectController` in that it goes through the OAuth flow (either OAuth 1 or OAuth 2, depending on the provider). Instead of creating a connection at the end of process, however, `ProviderSignInController` attempts to find a previously established connection and uses the connected account to authenticate the user with the application. If no previous connection matches, the flow will be sent to the application's sign up page so that the user may register with the application.

5.2 Enabling provider sign in

To add provider sign in capability to your Spring application, configure `ProviderSignInController` as a bean in your Spring MVC application:

```
<bean class="org.springframework.social.connect.signin.web.ProviderSignInController">
    <constructor-arg value="${application.secureUrl}" />
    <!-- relies on by-type autowiring for the other constructor-args -->
</bean>
```

The `ProviderSignInController` bean requires a single explicit `<constructor-arg>` value to specify the application's base secure URL. `ProviderSignInController` will use this URL to construct the callback URL used in the authentication flow. As with `ConnectController`, it is recommended that the application URL be externalized so that it can vary between deployment environments.

When authenticating via an OAuth 2 provider, `ProviderSignInController` supports the following flow:

- `POST /signin/{providerId}` - Initiates the sign in flow by redirecting to the provider's authentication endpoint.
- `GET /signin/{providerId}?code={verifier}` - Receives the authentication callback from the provider, accepting a code. Exchanges this code for an access token. It uses this access token to lookup a connected account and then authenticates to the application through the sign in service.
  - If the received access token doesn't match any existing connection, `ProviderSignInController` will redirect to a sign up URL. The sign up URL is "/signup" (relative to the application root).

For OAuth 1 providers, the flow is only slightly different:

- `POST /signin/{providerId}` - Initiates the sign in flow. This involves fetching a request token from the provider and then redirecting to Provider's authentication endpoint.
- `GET /signin/{providerId}?oauth_token={request token}&oauth_verifier={verifier}` - Receives the authentication callback from the provider, accepting a verification code. Exchanges this verification code along with the request token for an access token. It uses this access token to lookup a connected account and then authenticates to the application through the sign in service.
  - If the received access token doesn't match any existing connection, `ProviderSignInController` will redirect to a sign up URL. The sign up URL is "/signup" (relative to the application root).

ProviderSignInController's dependencies

As shown above, `ProviderSignInController` can be configured as a Spring bean given only a single constructor argument. Nevertheless, `ProviderSignInController` depends on a handful of other beans to do its job.

- A `ConnectionFactoryLocator` to lookup the ConnectionFactory used to create the Connection to the provider.
- A `UsersConnectionRepository` to find the user that has the connection to the provider user attempting to sign-in.
- A `ConnectionRepository` to persist a new connection when a new user signs up with the application after a failed sign-in attempt.
- A `SignInService` to sign a user into the application when a matching connection is found.

Because `ProviderSignInController`'s constructor is annotated with `@Inject`, those dependencies will be given to `ProviderSignInController` via autowiring. You'll still need to make sure they're available as beans in the Spring application context so that they can be autowired.

You should have already configured most of these dependencies when setting up connection support (in the previous chapter). The `SignInService` is exclusively used for provider sign in and so a `SignInService` bean will need to be added to the configuration. But first, you'll need to write an implementation of the `SignInService` interface.

The `SignInService` interface is defined as follows:

```
public interface SignInService {
    void signIn(String localUserId);
}
```

The `signIn()` method takes a single argument which is the local application user's user ID normalized as a `String`. No other credentials are necessary here because by the time this method is called the user will have signed into the provider and their connection with that provider has been used to prove the user's identity. Implementations of this interface should use this user ID to authenticate the user to the application.

Different applications will implement security differently, so each application must implement `SignInService` in a way that fits its unique security scheme. For example, suppose that an application's security is based Spring Security and simply uses a user's account ID as their principal. In that case, a simple implementation of `SignInService` might look like this:

```
@Service
public class SpringSecuritySignInService implements SignInService {
    public void signIn(String localUserId) {
        SecurityContextHolder.getContext().setAuthentication(new UsernamePasswordAuthenticationToken(localUserI
    }
}
```

Adding a provider sign in button

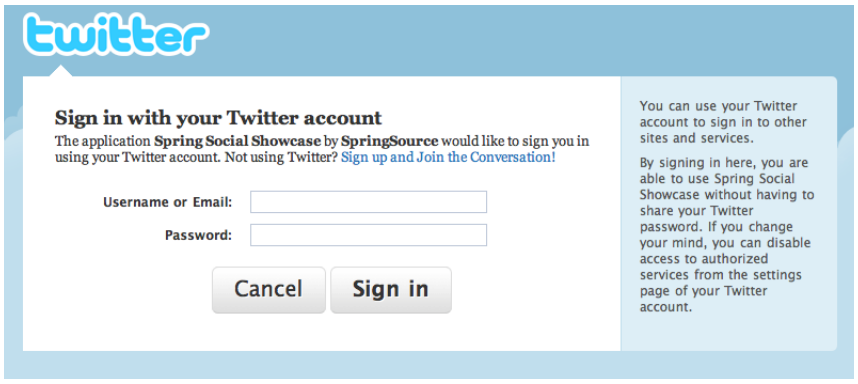
With `ProviderSignInController` and a `SignInService` configured, the backend support for provider sign in is in place. The last thing to do is to add a sign in button to your application that will kick off the authentication flow with `ProviderSignInController`.

For example, the following HTML snippet adds a "Signin with Twitter" button to a page:

```
<form id="tw_signin" action="<c:url value="/signin/twitter"/>" method="POST">
    <button type="submit"><img src="<c:url value="/resources/social/twitter/sign-in-with-twitter-d.png"/>" />
    </button>
</form>
```

Notice that the path used in the form's `action` attribute maps to the first step in `ProviderSignInController`'s flow. In this case, the provider is identified as "twitter".

Clicking this button will trigger a POST request to "/signin/twitter", kicking off the Twitter sign in flow. If the user has not yet signed into Twitter, the user will be presented with the following page from Twitter:



Sign in with your Twitter account

The application **Spring Social Showcase** by **SpringSource** would like to sign you in using your Twitter account. Not using Twitter? Sign up and Join the Conversation!

You can use your Twitter account to sign in to other sites and services.

By signing in here, you are able to use Spring Social Showcase without having to share your Twitter password. If you change your mind, you can disable access to authorized services from the settings page of your Twitter account.

After signing in, the flow will redirect back to the application to complete the sign in process.

5.3 Signing up after a failed sign in

If `ProviderSignInController` can't find a local user associated with a provider user attempting to sign-in, it will put the sign-in on hold and redirect the user to an application sign up page. By default, the sign up

URL is "/signup", relative to the application root. You can override that default by setting the `signupUrl` property on the controller. For example, the following configuration of `ProviderSignInController` sets the sign up URL to "/register":

```
<bean class="org.springframework.social.connect.signin.web.ProviderSignInController">
    <constructor-arg value="${application.url}" />
    <property name="signupUrl" value="/register" />
</bean>
```

Before redirecting to the sign up page, `ProviderSignInController` collects some information about the authentication attempt. This information can be used to prepopulate the sign up form and then, after successful registration, to establish a connection between the new account and the provider account.

To prepopulate the sign up form, you can fetch the user profile data from a connection retrieved from `ProviderSignInUtils.getConnection()`. For example, consider this Spring MVC controller method that setups up the sign up form with a `SignupForm` to bind to the sign up form:

```
@RequestMapping(value="/signup", method=RequestMethod.GET)
public SignupForm signupForm(WebRequest request) {
    Connection<?> connection = ProviderSignInUtils.getConnection(request);
    if (connection != null) {
        return SignupForm.fromProviderUser(connection.fetchUserProfile());
    } else {
        return new SignupForm();
    }
}
```

If `ProviderSignInUtils.getConnection()` returns a connection, that means there was a failed provider sign in attempt that can be completed if the user registers to the application. In that case, a `SignupForm` object is created from the user profile data obtained from the connection's `fetchUserProfile()` method. Within `fromProviderUser()`, the `SignupForm` properties may be set like this:

```
public static SignupForm fromProviderUser(UserProfile providerUser) {
    SignupForm form = new SignupForm();
    form.setFirstName(providerUser.getFirstName());
    form.setLastName(providerUser.getLastName());
    form.setUsername(providerUser.getUsername());
    form.setEmail(providerUser.getEmail());
    return form;
}
```

Here, the `SignupForm` is created with the user's first name, last name, username, and email from the `UserProfile`. In addition, `UserProfile` also has a `getName()` method which will return the user's full name as given by the provider.

The availability of `UserProfile`'s properties will depend on the provider. Twitter, for example, does not provide a user's email address, so the `getEmail()` method will always return null after a sign in attempt with Twitter.

After the user has successfully signed up in your application a connection can be created between the new local user account and their provider account. To complete the connection call `ProviderSignInUtils.handlePostSignUp()`. For example, the following method handles the sign up form submission, creates an account and then calls `ProviderSignInUtils.handlePostSignUp()` to complete the connection:

```
@RequestMapping(value="/signup", method=RequestMethod.POST)
public String signup(@Valid SignupForm form, BindingResult formBinding, WebRequest request) {
    if (formBinding.hasErrors()) {
        return null;
    }
    boolean accountCreated = createAccount(form, formBinding);
    if (accountCreated) {
        ProviderSignInUtils.handlePostSignUp(request);
        return "redirect:/";
    }
    return null;
}
```

6. Working with Service Provider APIs

6.1 Introduction

After a user has granted your application access to their service provider profile, you'll be able to interact with that service provider to update or retrieve the user's data. Your application may, for example, post a Tweet on behalf of a user or review a user's list of contacts to see if any of them have also created connections to your application.

Each service provider exposes their data and functionality through an API. Spring Social provides Java-based access to those APIs via provider-specific templates, each implementing a provider operations interface.

Spring Social comes with six provider API templates/operations for the following service providers:

- Twitter
- Facebook
- LinkedIn
- TripIt
- GitHub
- Gowalla

6.2 Twitter

From a user's perspective, Twitter's function is rather simple: Enable users to post whatever they're thinking, 140 characters at a time. In contrast, Twitter's API is rather rich, enabling applications to interact with Twitter in ways that may not be obvious from the humble tweet box. Spring Social offers interaction with Twitter's service API through the `TwitterApi` interface and its implementation, `TwitterTemplate`.

Creating an instance of `TwitterTemplate` involves invoking its constructor, passing in the application's OAuth credentials and an access token/secret pair authorizing the application to act on a user's behalf. For example:

```
String consumerKey = "..."; // The application's consumer key
String consumerSecret = "..."; // The application's consumer secret
String accessToken = "..."; // The access token granted after OAuth authorization
String accessTokenSecret = "..."; // The access token secret granted after OAuth authorization
TwitterApi twitterApi = new TwitterTemplate(consumerKey, consumerSecret, accessToken, accessTokenSecret);
```

In addition, `TwitterTemplate` has a default constructor that creates an instance without any OAuth credentials:

```
TwitterApi twitterApi = new TwitterTemplate();
```

```

```

When constructed with the default constructor, `TwitterTemplate` will allow a few simple operations that do not require authorization, such as searching. Attempting other operations, such as tweeting will fail with an `IllegalStateException` being thrown.

If you are using Spring Social's service provider framework, as described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `TwitterApi` via a `Connection`. For example, the following snippet calls `getApi()` on a connection to retrieve a `TwitterApi`:

```
Connection<TwitterApi> connection = connectionRepository.findPrimaryConnectionToApi(TwitterApi.class);
TwitterApi twitterApi = connection.getApi();
```

Here, `ConnectionRepository` is being asked for the primary connections that the current user has with Twitter. From that connection, it retrieves a `TwitterApi` instance that is configured with the connection details received when the connection was first established.

Once you have a `TwitterApi`, you can perform a several operations against Twitter. `TwitterApi` is defined as follows:

```
public interface TwitterApi {

    boolean isAuthorizedForUser();

    DirectMessageOperations directMessageOperations();

    FriendOperations friendOperations();

    ListOperations listOperations();

    SearchOperations searchOperations();

    TimelineOperations timelineOperations();

    UserOperations userOperations();

}
```

The `isAuthorizedForUser` helps determine if the `TwitterApi` instance has been created with credentials to perform on behalf of a user. It will return true if it is capable of performing operations requiring authorization; false otherwise.

The remaining six methods return sub-APIs, partitioning the Twitter service API into divisions targeting specific facets of Twitter functionality. These sub-APIs are defined by interfaces described in Table 6.1, “TwitterApi's Sub-APIs”.

Table 6.1. TwitterApi's Sub-APIs

Sub-API Interface	Description
DirectMessageOperations	Reading and sending direct messages.

Sub-API Interface	Description
FriendOperations	Retrieving a user's list of friends and followers and following/ unfollowing users.
ListOperations	Maintaining, subscribing to, and unsubscripting from user lists
SearchOperations	Searching tweets and viewing search trends
TimelineOperations	Reading timelines and posting tweets.
UserOperations	Retrieving user profile data.

What follows is a survey of common tasks you may perform with `TwitterApi` and its sub-APIs. For complete details on the Spring Social's entire Twitter API binding, refer to the JavaDoc.

Retrieving a user's Twitter profile data

To get a user's Twitter profile, call `UserOperations`' `getUserProfile()`:

```
TwitterProfile profile = twitterApi.userOperations().getUserProfile();
```

This returns a `TwitterProfile` object containing profile data for the authenticated user. This profile information includes the user's Twitter screen name, their name, location, description, and the date that they created their Twitter account. Also included is a URL to their profile image.

If you want to retrieve the user profile for a specific user other than the authenticated user, you can so do by passing the user's screen name as a parameter to `getUserProfile()`:

```
TwitterProfile profile = twitterApi.userOperations().getUserProfile("habuma");
```

If all you need is the screen name for the authenticating user, then call `UserOperations.getScreenName()`:

```
String profileId = twitterApi.userOperations().getScreenName();
```

Tweeting

To post a message to Twitter the simplest thing to do is to pass the message to the `updateStatus()` method provided by `TimelineOperations`:

```
twitterApi.timelineOperations().updateStatus("Spring Social is awesome!")
```

Optionally, you may also include metadata about the tweet, such as the location (latitude and longitude) you are tweeting from. For that, pass in a `StatusDetails` object, setting the location property:

```
StatusDetails statusDetails = new StatusDetails().setLocation(51.502f, -0.126f);
twitterApi.timelineOperations().updateStatus("I'm tweeting from London!", statusDetails)
```

To have Twitter display the location in a map (on the Twitter web site) then you should also set the `displayCoordinates` property to `true`:

```
StatusDetails statusDetails = new StatusDetails().setLocation(51.502f, -0.126f).setDisplayCoordinates(true);
twitterApi.timelineOperations().updateStatus("I'm tweeting from London!", statusDetails)
```

If you'd like to retweet another tweet (perhaps one found while searching or reading the Twitter timeline), call the `retweet()` method, passing in the ID of the tweet to be retweeted:

```
long tweetId = tweet.getId();
twitterApi.timelineOperations().retweet(tweetId);
```

Note that Twitter disallows repeated tweets. Attempting to tweet or retweet the same message multiple times will result in a `DuplicateTweetException` being thrown.

Reading Twitter timelines

From a Twitter user's perspective, Twitter organizes tweets into four different timelines:

- User - Includes tweets posted by the user.
- Friends - Includes tweets from the user's timeline and the timeline of anyone that they follow, with the exception of any retweets.
- Home - Includes tweets from the user's timeline and the timeline of anyone that they follow.
- Public - Includes tweets from all Twitter users.

To be clear, the only difference between the home timeline and the friends timeline is that the friends timeline excludes retweets.

`TimelineOperations` also supports reading of tweets from one of the available Twitter timelines. To retrieve the 20 most recent tweets from the public timeline, use the `getPublicTimeline()` method:

```
List<Tweet> tweets = twitterApi.timelineOperations().getPublicTimeline();
```

`getHomeTimeline()` retrieves the 20 most recent tweets from the user's home timeline:

```
List<Tweet> tweets = twitterApi.timelineOperations().getHomeTimeline();
```

Similarly, `getFriendsTimeline()` retrieves the 20 most recent tweets from the user's friends timeline:

```
List<Tweet> tweets = twitterApi.timelineOperations().getFriendsTimeline();
```

To get tweets from the authenticating user's own timeline, call the `getUserTimeline()` method:

```
List<Tweet> tweets = twitterApi.timelineOperations().getUserTimeline();
```

If you'd like to retrieve the 20 most recent tweets from a specific user's timeline (not necessarily the authenticating user's timeline), pass the user's screen name in as a parameter to `getUserTimeline()`:

```
List<Tweet> tweets = twitterApi.timelineOperations().getUserTimeline("rclarkson");
```

In addition to the four Twitter timelines, you may also want to get a list of tweets mentioning the user. The `getMentions()` method returns the 20 most recent tweets that mention the authenticating user:

```
List<Tweet> tweets = twitterApi.timelineOperations().getMentions();
```

Friends and Followers

A key social concept in Twitter is the ability for one user to "follow" another user. The followed user's tweets will appear in the following user's home and friends timelines. To follow a user on behalf of the authenticating user, call the `FriendOperations`' `follow()` method:

```
twitterApi.friendOperations().follow("habuma");
```

Similarly, you may stop following a user using the `unfollow()` method:

```
twitterApi.friendOperations().unfollow("habuma");
```

If you want to see who a particular user is following, use the `getFriends()` method:

```
List<TwitterProfile> friends = twitterApi.friendOperations().getFriends("habuma");
```

On the other hand, you may be interested in seeing who is following a given user. In that case the `getFollowers()` method may be useful:

```
List<TwitterProfile> followers = twitterApi.friendOperations().getFollowers("habuma");
```

Twitter User Lists

In addition to following other users, Twitter provides the ability for users to collect users in lists, regardless of whether or not they are being followed. These lists may be private to the use who created them or may be public for others to read and subscribe to.

To create a new list, use `ListOperations`' `createList()` method:

```
UserList familyList = twitterApi.listOperations().createList(
        "My Family", "Tweets from my immediate family members", false);
```

`createList()` takes three parameters and returns a `UserList` object representing the newly created list. The first parameter is the name of the list. The second parameter is a brief description of the list. The final parameter is a boolean indicating whether or not the list is public. Here, false indicates that the list should be private.

Once the list is created, you may add members to the list by calling the `addToList()` method:

```
twitterApi.listOperations().addToList(familyList.getSlug(), "artnames");
```

The first parameter given to `addToList()` is the list slug (which is readily available from the `UserList` object). The second parameter is the screen name of a user to add to the list.

To remove a member from a list, pass the same parameters to `removedFromList()`:

```
twitterApi.listOperations().removeFromList(familyList.getSlug(), "artnames");
```

You can also subscribe to a list on behalf of the authenticating user. Subscribing to a list has the effect of including tweets from the list's members in the user's home timeline. The `subscribe()` method is used to subscribe to a list:

```
twitterApi.listOperations().subscribe("habuma", "music");
```

Here, `subscribe()` is given the list owner's screen name ("habuma") and the list slug ("music").

Similarly, you may unsubscribe from a list with the `unsubscribe()` method:

```
twitterApi.listOperations().unsubscribe("habuma", "music");
```

Searching Twitter

`SearchOperations` enables you to search the public timeline for tweets containing some text through its `search()` method.

For example, to search for tweets containing "#spring":

```
SearchResults results = twitterApi.searchOperations().search("#spring");
```

The `search()` method will return a `SearchResults` object that includes a list of 50 most recent matching tweets as well as some metadata concerning the result set. The metadata includes the maximum tweet ID in the search results list as well as the ID of a tweet that precedes the resulting tweets. The `sinceId` and `maxId` properties effectively define the boundaries of the result set. Additionally, there's a boolean `lastPage` property that, if `true`, indicates that this result set is the page of results.

To gain better control over the paging of results, you may choose to pass in the page and results per page to `search()`:

```
SearchResults results = twitterApi.searchOperations().search("#spring", 2, 10);
```

Here, we're asking for the 2nd page of results where the pages have 10 tweets per page.

Finally, if you'd like to confine the bounds of the search results to fit between two tweet IDs, you may pass in the since and maximum tweet ID values to `search()`:

```
SearchResults results = twitterApi.searchOperations().search("#spring", 2, 10, 145962, 210112);
```

This ensures that the result set will not contain any tweets posted before the tweet whose ID is 146962 nor any tweets posted after the tweet whose ID is 210112.

Sending and receiving direct messages

In addition to posting tweets to the public timelines, Twitter also supports sending of private messages directly to a given user. `DirectMessageOperations`' `sendDirectMessage()` method can be used to send a direct message to another user:

```
twitterApi.directMessageOperations().sendDirectMessage("kdonald", "You going to the Dolphins game?")
```

`DirectMessageOperations` can also be used to read direct messages received by the authenticating user through its `getDirectMessagesReceived()` method:

```
List<DirectMessage> twitterApi.directMessageOperations().getDirectMessagesReceived();
```

`getDirectMessagesReceived()` will return the 20 most recently received direct messages.

6.3 Facebook

Spring Social's `FacebookApi` and its implementation, `FacebookTemplate` provide the operations needed to interact with Facebook on behalf of a user. Creating an instance of `FacebookTemplate` is as simple as constructing it by passing in an authorized access token to the constructor:

```
String accessToken = "f8FX29g..."; // access token received from Facebook after OAuth authorization
FacebookApi facebook = new FacebookTemplate(accessToken);
```

If you are using Spring Social's service provider framework, as described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `FacebookApi` via a `Connection`. For example, the following snippet calls `getApi()` on a connection to retrieve a `FacebookApi`:

```
Connection<FacebookApi> connection = connectionRepository.findPrimaryConnectionToApi(FacebookApi.class);
FacebookApi facebookApi = connection.getApi();
```

Here, `ConnectionRepository` is being asked for the primary connections that the current user has with Facebook. From that connection, it retrieves a `FacebookApi` instance that is configured with the connection details received when the connection was first established.

With a `FacebookApi` in hand, there are several ways you can use it to interact with Facebook on behalf of the user. Spring Social's Facebook API binding is divided into 9 sub-APIs exposes through the methods of `FacebookApi`:

```
public interface FacebookApi extends GraphApi {

    CommentOperations commentOperations();

    EventOperations eventOperations();

    FeedOperations feedOperations();

    FriendOperations friendOperations();

    GroupOperations groupOperations();

    LikeOperations likeOperations();

    MediaOperations mediaOperations();

    PlacesOperations placesOperations();

    UserOperations userOperations();

}
```

The sub-API interfaces returned from `FacebookApi`'s methods are described in Table 6.2, “FacebookApi's Sub-APIs”.

Table 6.2. FacebookApi's Sub-APIs

Sub-API Interface	Description
CommentOperations	Add, delete, and read comments on Facebook objects.
EventOperations	Create and maintain events and RSVP to event invitations.
FeedOperations	Read and post to a Facebook wall.
FriendOperations	Retrieve a user's friends and maintain friend lists.
GroupOperations	Retrieve group details and members.
LikeOperations	Retrieve a user's interests and likes. Like and unlike objects.
MediaOperations	Maintain albums, photos, and videos.
PlacesOperations	Checkin to location in Facebook Places and retrieve places a user and their friends have checked into.
UserOperations	Retrieve user profile data and profile images.

The following sections will give an overview of common tasks that can be performed via `FacebookApi` and its sub-APIs. For complete details on all of the operations available, refer to the JavaDoc.

Retrieving a user's profile data

You can retrieve a user's Facebook profile data using `FacebookApi`' `getUserProfile()` method:

```
FacebookProfile profile = facebookApi.userOperations().getUserProfile();
```

The `FacebookProfile` object will contain basic profile information about the authenticating user, including their first and last name and their Facebook ID. Depending on what authorization scope has been granted to the application, it may also include additional details about the user such as their email address, birthday, hometown, and religious and political affiliations. For example, `getBirthday()` will return the current user's birthday if the application has been granted "user_birthday" permission; null otherwise. Consult the JavaDoc for `FacebookProfile` for details on which permissions are required for each property.

If all you need is the user's Facebook ID, you can call `getProfileId()` instead:

```
String profileId = facebookApi.userOperations().getProfileId();
```

Or if you want the user's Facebook URL, you can call `getProfileUrl()`:

```
String profileUrl = facebookApi.userOperations().getProfileUrl();
```

Getting a user's Facebook friends

An essential feature of Facebook and other social networks is creating a network of friends or contacts. You can access the user's list of Facebook friends by calling the `getFriendIds()` method from `FriendOperations`:

```
List<String> friendIds = facebookApi.friendOperations().getFriendIds();
```

This returns a list of Facebook IDs belonging to the current user's list of friends. This is just a list of `String` IDs, so to retrieve an individual user's profile data, you can turn around and call the `getUserProfile()`, passing in one of those IDs to retrieve the profile data for an individual user:

```
FacebookProfile firstFriend = facebookApi.userOperations().getUserProfile(friendIds.get(0));
```

Or you can get a list of user's friends as `FacebookProfile`s by calling `getFriendProfiles()`:

```
List<FacebookProfile> friends = facebookApi.friendOperations().getFriendProfiles();
```

Facebook also enables users to organize their friends into friend lists. To retrieve a list of the authenticating user's friend lists, call `getFriendLists()` with no arguments:

```
List<Reference> friends = facebookApi.friendOperations().getFriendLists();
```

You can also retrieve a list of friend lists for a specific user by passing the user ID (or an alias) to `getFriendLists()`:

```
List<Reference> friends = facebookApi.friendOperations().getFriendLists("habuma");
```

`getFriendLists()` returns a list of `Reference` objects that carry the ID and name of each friend list.

To retieve a list of friends who are members of a specific friend list call `getFriendListMembers()`, passing in the ID of the friend list:

```
List<Reference> friends = facebookApi.friendOperations().getFriendListMembers("193839228");
```

`FriendOperations` also support management of friend lists. For example, the `createFriendList()` method will create a new friend list for the user:

```
Reference collegeFriends = facebookApi.friendOperations().createFriendList("College Buddies");
```

`createFriendList()` returns a `Reference` to the newly created friend list.

To add a friend to the friend list, call `addToFriendList()`:

```
facebookApi.friendOperations().addToFriendList(collegeFriends.getId(), "527631174");
```

`addToFriendList()` takes two arguments: The ID of the friend list and the ID (or alias) of a friend to add to the list.

In a similar fashion, you may remove a friend from a list by calling `removeFromFriendList()`:

```
facebookApi.friendOperations().removeFromFriendList(collegeFriends.getId(), "527631174");
```

Posting to and reading feeds

To post a message to the user's Facebook wall, call `FeedOperations`' `updateStatus()` method, passing in the message to be posted:

```
facebookApi.feedOperations().updateStatus("I'm trying out Spring Social!");
```

If you'd like to attach a link to the status message, you can do so by passing in a `FacebookLink` object along with the message:

```
FacebookLink link = new FacebookLink("http://www.springsource.org/spring-social",
        "Spring Social",
        "The Spring Social Project",
        "Spring Social is an extension to Spring to enable applications to connect with service providers.");
facebookApi.feedOperations().updateStatus("I'm trying out Spring Social!", link);
```

When constructing the `FacebookLink` object, the first parameter is the link's URL, the second parameter is the name of the link, the third parameter is a caption, and the fourth is a description of the link.

If you want to read posts from a user's feed, `FeedOperations` has several methods to choose from. The `getFeed()` method retrieves recent posts to a user's wall. When called with no parameters, it retrieves posts from the authenticating user's wall:

```
List<Post> feed = facebookApi.feedOperations().getFeed();
```

Or you can read a specific user's wall by passing their Facebook ID to `getFeed()`:

```
List<Post> feed = facebookApi.feedOperations().getFeed("habuma");
```

In any event, the `getFeed()` method returns a list of `Post` objects. The `Post` class has six subtypes to represent different kinds of posts:

- `CheckinPost` - Reports a user's checkin in Facebook Places.
- `LinkPost` - Shares a link the user has posted.
- `NotePost` - Publicizes a note that the user has written.
- `PhotoPost` - Announces a photo that the user has uploaded.
- `StatusPost` - A simple status.
- `VideoPost` - Announces a video that the user has uploaded.

The `Post`'s `getType()` method identifies the type of `Post`.

6.4 LinkedIn

LinkedIn is a social networking site geared toward professionals. It enables its users to maintain and correspond with a network of contacts they have are professionally linked to.

Spring Social offers integration with LinkedIn via `LinkedInApi` and its implementation, `LinkedInTemplate`.

To create an instance of `LinkedInTemplate`, you may pass in your application's OAuth 1 credentials, along with an access token/secret pair to the constructor:

```
String consumerKey = "..."; // The application's consumer key
String consumerSecret = "..."; // The application's consumer secret
String accessToken = "..."; // The access token granted after OAuth authorization
String accessTokenSecret = "..."; // The access token secret granted after OAuth authorization
LinkedInApi linkedinApi = new LinkedInTemplate(consumerKey, consumerSecret, accessToken, accessTokenSecret);
```

If you are using Spring Social's service provider framework, as described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `LinkedInApi` via a `Connection`. For example, the following snippet calls `getApi()` on a connection to retrieve a `LinkedInApi`:

```
Connection<LinkedInApi> connection = connectionRepository.findPrimaryConnectionToApi(LinkedInApi.class);
LinkedInApi linkedinApi = connection.getApi();
```

Here, `ConnectionRepository` is being asked for the primary connections that the current user has with LinkedIn. From that connection, it retrieves a `LinkedInApi` instance that is configured with the connection details received when the connection was first established.

Once you have a `LinkedInApi` you can use it to interact with LinkedIn on behalf of the user who the access token was granted for.

Retrieving a user's LinkedIn profile data

To retrieve the authenticated user's profile data, call the `getUserProfile()` method:

```
LinkedInProfile profile = linkedin.getUserProfile();
```

The data returned in the `LinkedInProfile` includes the user's LinkedIn ID, first and last names, their "headline", the industry they're in, and URLs for the public and standard profile pages.

If it's only the user's LinkedIn ID you need, then you can get that by calling the `getProfileId()` method:

```
String profileId = linkedin.getProfileId();
```

Or if you only need a URL for the user's public profile page, call `getProfileUrl()`:

```
String profileUrl = linkedin.getProfileUrl();
```

Getting a user's LinkedIn connections

To retrieve a list of LinkedIn users to whom the user is connected, call the `getConnections()` method:

```
List<LinkedInProfile> connections = linkedin.getConnections();
```

This will return a list of `LinkedInProfile` objects for the user's 1st-degree network (those LinkedIn users to whom the user is directly linked--not their extended network).

6.5 TripIt

TripIt is a social network that links together travelers. By connecting with other travelers, you can keep in touch with contacts when your travel plans coincide. Also, aside from its social aspects, TripIt is a rather useful service for managing one's travel information.

Using Spring Social's `TripItApi` and its implementation, `TripItTemplate`, you can develop applications that integrate a user's travel information and network.

To create an instance of `TripItTemplate`, pass in your application's OAuth 1 credentials along with a user's access token/secret pair to the constructor:

```
String consumerKey = "..."; // The application's consumer key
String consumerSecret = "..."; // The application's consumer secret
String accessToken = "..."; // The access token granted after OAuth authorization
String accessTokenSecret = "..."; // The access token secret granted after OAuth authorization
TripItApi tripitApi = new TripItTemplate(consumerKey, consumerSecret, accessToken, accessTokenSecret);
```

If you are using Spring Social's service provider framework, as described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `TripItApi` via a `Connection`. For example, the following snippet calls `getApi()` on a connection to retrieve a `TripItApi`:

```
Connection<TripItApi> connection = connectionRepository.findPrimaryConnectionToApi(TripItApi.class);
TripItApi tripitApi = connection.getApi();
```

Here, `ConnectionRepository` is being asked for the primary connections that the current user has with TripIt. From that connection, it retrieves a `TripItApi` instance that is configured with the connection details received when the connection was first established.

In either event, once you have a `TripItApi`, you can use it to retrieve a user's profile and travel data from TripIt.

Retrieving a user's TripIt profile data

`TripItApi`' `getUserProfile()` method is useful for retrieving the authenticated user's TripIt profile data. For example:

```
TripItProfile userProfile = tripit.getUserProfile();
```

`getUserProfile()` returns a `TripItProfile` object that carries details about the user from TripIt. This includes the user's screen name, their display name, their home city, and their company.

If all you need is the user's TripIt screen name, you can get that by calling `getProfileId()`:

```
String profileId = tripit.getProfileId();
```

Or if you only need a URL to the user's TripIt profile page, then call `getProfileUrl()`:

```
String profileUrl = tripit.getProfileUrl();
```

Getting a user's upcoming trips

If the user has any upcoming trips planned, your application can access the trip information by calling `getUpcomingTrips()`:

```
List<Trip> trips = tripit.getUpcomingTrips();
```

This returns a list of `Trip` objects containing details about each trip, such as the start and end dates for the trip, the primary location, and the trip's display name.

6.6 GitHub

Although many developers think of GitHub as Git-based source code hosting, the tagline in GitHub's logo clearly states that GitHub is about "social coding". GitHub is a social network that links developers together and with the projects they follow and/or contribute to.

Spring Social's `GitHubApi` and its implementation, `GitHubTemplate`, offer integration with GitHub's social platform.

To obtain an instance of `GitHubTemplate`, you can instantiate it by passing an authorized access token to its constructor:

```
String accessToken = "f8FX29g..."; // access token received from GitHub after OAuth authorization
GitHubApi githubApi = new GitHubTemplate(accessToken);
```

If you are using Spring Social's service provider framework, as described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `GitHubApi` via a `Connection`. For example, the following snippet calls `getApi()` on a connection to retrieve a `GitHubApi`:

```
Connection<GitHubApi> connection = connectionRepository.findPrimaryConnectionToApi(GitHubApi.class);
GitHubApi githubApi = connection.getApi();
```

Here, `ConnectionRepository` is being asked for the primary connections that the current user has with GitHub. From that connection, it retrieves a `GitHubApi` instance that is configured with the connection details received when the connection was first established.

With a `GitHubApi` in hand, there are a handful of operations it provides to interact with GitHub on behalf of the user. These will be covered in the following sections.

Retrieving a GitHub user's profile

To get the currently authenticated user's GitHub profile data, call `GitHubApi`'s `getUserProfile()` method:

```
GitHubUserProfile profile = github.getUserProfile();
```

The `GitHubUserProfile` returned from `getUserProfile()` includes several useful pieces of information about the user, including their...

- Name
- Username (ie, login name)
- Company

- Email address
- Location
- Blog URL
- Date they joined GitHub

If all you need is the user's GitHub username, you can get that by calling the `getProfileId()` method:

```
String username = github.getProfileId();
```

And if you need a URL to the user's GitHub profile page, you can use the `getProfileUrl()` method:

```
String profileUrl = github.getProfileUrl();
```

6.7 Gowalla

Gowalla is a location-based social network where users may check in to various locations they visit and earn pins and stamps for having checked in a locations that achieve some goal (for example, a "Lucha Libre" pin may be earned by having checked into 10 different Mexican food restaurants).

Spring Social supports interaction with Gowalla through the `GowallaApi` interface and its implementation, `GowallaTemplate`.

To obtain an instance of `GowallaTemplate`, you can instantiate it by passing an authorized access token to its constructor:

```
String accessToken = "f8FX29g..."; // access token received from Gowalla after OAuth authorization
GowallaApi gowallaApi = new GowallaTemplate(accessToken);
```

If you are using Spring Social's service provider framework, as described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `GowallaApi` via a `Connection`. For example, the following snippet calls `getApi()` on a connection to retrieve a `GowallaApi`:

```
Connection<GowallaApi> connection = connectionRepository.findPrimaryConnectionToApi(GowallaApi.class);
GowallaApi gowallaApi = connection.getApi();
```

Here, `ConnectionRepository` is being asked for the primary connections that the current user has with Gowalla. From that connection, it retrieves a `GowallaApi` instance that is configured with the connection details received when the connection was first established.

With a `GowallaApi` in hand, there are a handful of operations it provides to interact with Gowalla on behalf of the user. These will be covered in the following sections.

Retrieving a user's profile data

You can retrieve a user's Gowalla profile using the `getUserProfile()` method:

```
GowallaProfile profile = gowalla.getUserProfile();
```

This will return the Gowalla profile data for the authenticated user. If you want to retrieve profile data for another user, you can pass the user's profile ID into `getUserProfile()`:

```
GowallaProfile profile = gowalla.getUserProfile("habuma");
```

The `GowallaProfile` object contains basic information about the Gowalla user such as their first and last names, their hometown, and the number of pins and stamps that they have earned.

If all you want is the authenticated user's profile ID, you can get that by calling the `getProfileId()`:

```
String profileId = gowalla.getProfileId();
```

Or if you want the URL to the user's profile page at Gowalla, use the `getProfileUrl()` method:

```
String profileUrl = gowalla.getProfileUrl();
```

Getting a user's checkins

`GowallaApi` also allows you to learn about the user's favorite checkin spots. The `getTopCheckins()` method will provide a list of the top 10 places that the user has visited:

```
List<Checkin> topCheckins = gowalla.getTopCheckins();
```

Each member of the returns list is a `Checkin` object that includes the name of the location as well as the number of times that the user has checked in at that location.