Spring Social Reference Manual

Craig Walls
Keith Donald

Spring Social Reference Manual

by Craig Walls and Keith Donald

1.0.0.M2

Table of Contents

1. Spring Social Overview

1.1 Introduction

The Spring Social project enables Spring applications to connect with users' profiles on service providers such as Facebook and Twitter and interact with those services on behalf of the user.

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 and checks it off of his viewing list and indicating if he liked the movie or not. Based on his responses, the movie club application can tailor its future suggestions 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 them 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, an 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 service provider framework that models the authorization and connection creation process with a service.
- A connection controller that handles the OAuth exchange between a service provider, consumer, and user.
- APIs for several service providers such as Facebook, Twitter, LinkedIn, TripIt, GitHub, and Gowalla.
- A signin controller that enables a user to authenticate to an application by signing into either Facebook or Twitter.

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 ServiceProvider connect framework and OAuth support.
spring-social-web	Spring Social's `ConnectController` which uses the ServiceProvider framework to manage connections in a web application environment
spring-social-facebook	Includes Spring Social's Facebook API as well as support for signing into an application through Facebook.
spring-social-twitter	Includes Spring Social's Twitter API as well as support for signing into an application via Twitter.
spring-social-linkedin	Includes Spring Social's LinkedIn API.
spring-social-github	Includes Spring Social's GitHub API.
spring-social-gowalla	Includes Spring Social's Gowalla API.
spring-social-tripit	Includes Spring Social's TripIt API.
spring-social-test	Support for testing ServiceProvider 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 access a user's GitHub profile.

If you are developing against a milestone version, such as 1.0.0.M2, 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 Base API

The `ServiceProvider<S>` interface defines the central API for managing connections to an external service provider such as Facebook:

```
public interface ServiceProvider<S> {

    String getId();

    boolean isConnected(Serializable accountId);

    List<ServiceProviderConnection<S>> getConnections(Serializable accountId);

}
```

The <S> parameterized type represents the Java binding to the ServiceProvider's API. For example, the Facebook ServiceProvider implementation is parameterized as ServiceProvider<FacebookApi>, where FacebookApi is the Java interface that may be used to invoke Facebook's graph API on behalf of a user.

Each ServiceProvider is identified by an ID, as returned by the `getId()` method. This id is expected to be unique across all ServiceProviders registered with your application.

A single local user account can have one-to-many connections established with a ServiceProvider, where each connection represents a link between the local user's account and an external account the user has on the provider's system. `isConnected()` checks to see if *any* connections exist between a user account and the service provider. If there are connections, `getConnections()` returns them in rank order.

With a reference to a ServiceProviderConnection you can do the following:

```
public interface ServiceProviderConnection<S> {

    S getServiceApi();

    void disconnect();
}
```

`getServiceApi()` returns a Java binding to the ServiceProvider's API for the external user account associated with the connection. The API can be used to access and update user data on the provider's system.

`disconnect()` may be used to remove the connection with the ServiceProvider, if it is no longer desired.

To put this framework into action, consider Twitter as an example ServiceProvider. Suppose user 'kdonald' of AcmeApp has three Twitter accounts and has connected with each of them:

1. ServiceProvider#getId() would return 'twitter'.
2. ServiceProvider#isConnected("kdonald") would return 'true'.
3. ServiceProvider#getConnections("kdonald") would return a 'connections' List with three elements, one for each Twitter account.
4. connections.get(0) would return the 'connection' to the first Twitter account, and connection.getServiceApi() would return a TwitterApi that can access and update information about that Twitter account.
5. connections.get(1) and connections.get(2) would allow AcmeApp to access and update information about the second and third Twitter accounts, respectively.
6. connection.disconnect() can be called to remove a connection, at which the linked Twitter account is no longer accessible to the application.

2.2 Establishing Connections

So far we have discussed how existing connections are managed using the ServiceProvider framework, but we have not yet discussed how new connections are established. The manner in which connections between local user accounts and external provider accounts 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.

Because each authorization protocol is different, protocol-specific details are kept out of the base ServiceProvider interface. Sub-interfaces have been defined for each protocol, reflecting a distinct ServiceProvider type. In the following sections, we will discuss each type of ServiceProvider supported by the framework. Each section will also describe the protocol-specific flow required to establish a new connection.

OAuth2 Service Providers

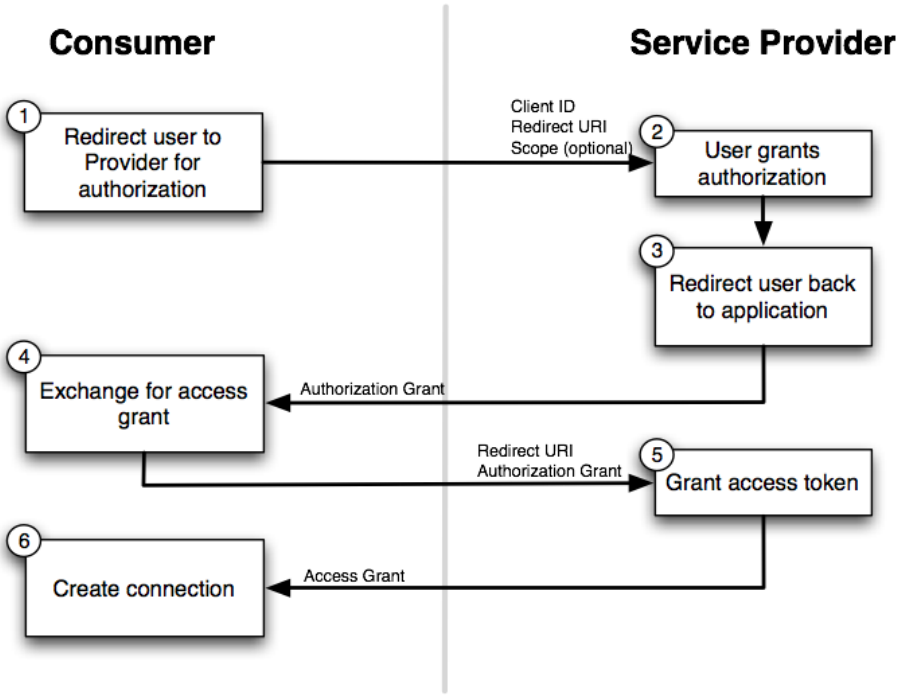
OAuth 2 is rapidly becoming a preferred authorization protocol, and is used by major service providers such as Facebook, Github, Gowalla, and 37signals. In Spring Social, the OAuth2ServiceProvider interface models a service provider based on the OAuth 2 protocol:

```
public interface OAuth2ServiceProvider<S> extends ServiceProvider<S> {

    OAuth2Operations getOAuthOperations();
    ServiceProviderConnection<S> connect(Serializable accountId, 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 `connect`. The OAuth2Operations interface is shown below:

```
public interface OAuth2Operations {

    String buildAuthorizeUrl(String redirectUri, String scope);

    AccessGrant exchangeForAccess(String authorizationGrant, String redirectUri);

}
```

Callers are first expected to call buildAuthorizeUrl(String, String) to construct the URL to redirect the user to for connection authorization. Upon user authorization, the authorizationGrant 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 grant 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.

OAuth1 Service Providers

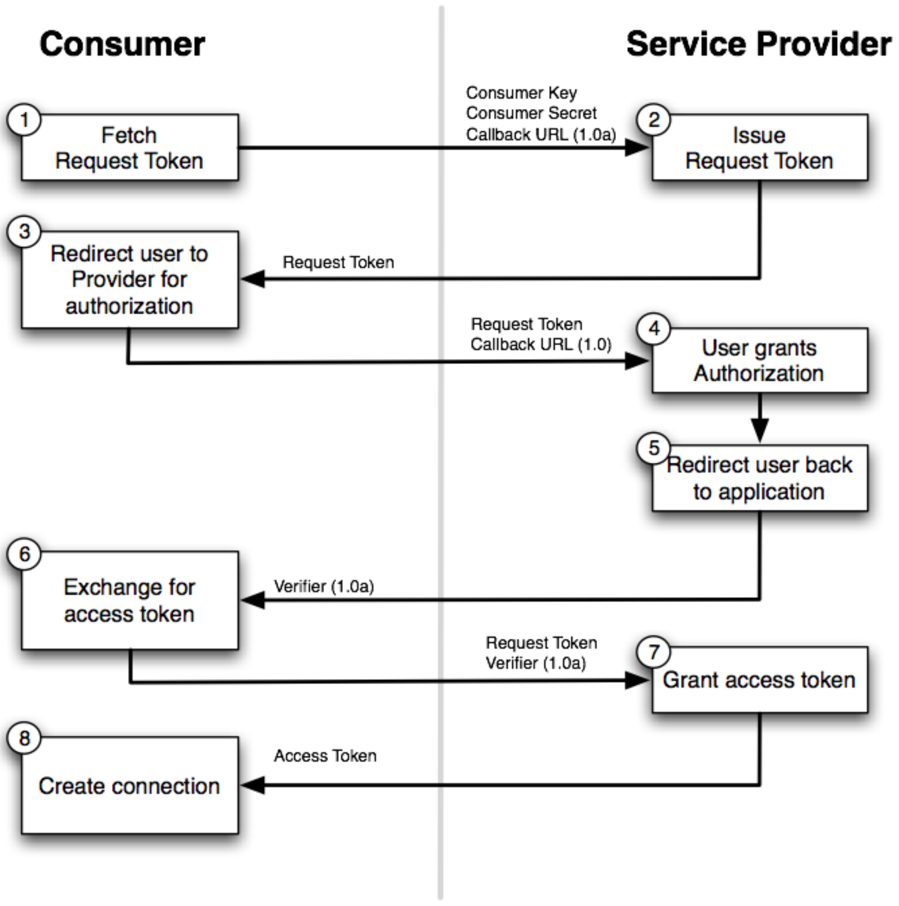
OAuth 1 is the previous version of the OAuth protocol. It is more complex OAuth 2, and sufficiently different that it is supported separately. Twitter, Linked In, and TripIt are some of the well-known ServiceProviders that use OAuth 1. In Spring Social, the OAuth1ServiceProvider interface models a service provider based on the OAuth 1 protocol:

```
public interface OAuth1ServiceProvider<S> extends ServiceProvider<S> {

    OAuth1Operations getOAuthOperations();

    ServiceProviderConnection<S> connect(Serializable accountId, 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 `connect`. The OAuth1Operations interface is shown below:

```
public interface OAuth1Operations {

    OAuthToken fetchNewRequestToken(String callbackUrl);

    String buildAuthorizeUrl(String requestToken);

    OAuthToken exchangeForAccessToken(AuthorizedRequestToken requestToken);

}
```

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) 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.

3. Implementing Service Providers

The spring-social-core module provides support for implementing your own ServiceProviders. This support consists of convenient base classes for the various ServiceProvider types, such as OAuth1 and OAuth2-based providers. A common data access interface is also provided for persisting connection information. In this section, you will learn how to implement ServiceProviders.

3.1 OAuth2 Service Providers

To implement an OAuth2-based ServiceProvider, first extend AbstractOAuth2ServiceProvider. Parameterize <S> to be the Java Binding to the ServiceProvider API. Define a single constructor that accepts an clientId, clientSecret, and ConnectionRepository. Finally, implement getApi(String) to return a new API instance.

See FacebookServiceProvider as an example of an OAuth2-based ServiceProvider:

```
package org.springframework.social.facebook.connect;

import org.springframework.social.connect.oauth2.AbstractOAuth2ServiceProvider;
import org.springframework.social.connect.support.ConnectionRepository;
import org.springframework.social.facebook.FacebookApi;
import org.springframework.social.facebook.FacebookTemplate;
import org.springframework.social.oauth2.OAuth2Template;

public final class FacebookServiceProvider extends AbstractOAuth2ServiceProvider<FacebookApi> {

    public FacebookServiceProvider(String clientId, String clientSecret, ConnectionRepository connectionReposit
        super("facebook", connectionRepository,
            new OAuth2Template(appId, appSecret,
                "https://graph.facebook.com/oauth/authorize?client_id={client_id}&redirect_uri={redirect_uri}&s
                "https://graph.facebook.com/oauth/access_token"));
    }

    @Override
    protected FacebookApi getApi(String accessToken) {
        return new FacebookTemplate(accessToken);
    }

}
```

In the constructor, you should call super, passing up the ID of the ServiceProvider, the connection repository, and a configured OAuth2Template, which 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 Service API implementation, passing it the access token needed to make requests for protected resources. Inside the API implementation, we generally recommend using RestTemplate to make the HTTP calls and add the required Authorization header:

```
public FacebookTemplate(String accessToken) {
    // creates a RestTemplate that adds the OAuth2-draft10 Authorization header to each request before it is ex
    restTemplate = ProtectedResourceClientFactory.draft10(accessToken);
}
```

An example API call with RestTemplate is shown below:

```
public FacebookProfile getUserProfile(String facebookId) {
    return new FacebookProfile(restTemplate.getForObject("https://graph.facebook.com/{facebookId}", Map.class,
}
```

3.2 OAuth1 Service Providers

To implement an OAuth1-based ServiceProvider, first extend AbstractOAuth1ServiceProvider. Parameterize <S> to be the Java Binding to the ServiceProvider API. Define a single constructor that accepts a consumerKey, consumerSecret, and ConnectionRepository. Finally, implement getApi(String, String, String, String) to return a new API instance.

See TwitterServiceProvider as an example of an OAuth1-based ServiceProvider:

```
package org.springframework.social.twitter.connect;

import org.springframework.social.connect.oauth1.AbstractOAuth1ServiceProvider;
import org.springframework.social.connect.support.ConnectionRepository;
import org.springframework.social.oauth1.OAuth1Template;
import org.springframework.social.twitter.TwitterOperations;
import org.springframework.social.twitter.TwitterTemplate;

public final class TwitterServiceProvider extends AbstractOAuth1ServiceProvider<TwitterApi> {

    public TwitterServiceProvider(String consumerKey, String consumerSecret, ConnectionRepository connectionRep
        super("twitter", connectionRepository, consumerKey, consumerSecret,
            new OAuth1Template(consumerKey, consumerSecret,
                "https://twitter.com/oauth/request_token",
                "https://twitter.com/oauth/authorize?oauth_token={requestToken}",
                "https://twitter.com/oauth/access_token"));
    }

    @Override
    protected TwitterApi getApi(String consumerKey, String consumerSecret, String accessToken, String secret) {
        return new TwitterTemplate(consumerKey, consumerSecret, accessToken, secret);
    }

}
```

In the constructor, you should call super, passing up the ID of the ServiceProvider, the connection repository, the consumerKey and secret, and a 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, String, String), you should construct your Service API implementation, passing it the four tokens needed to make requests for protected resources. Inside the API implementation, we generally recommend using RestTemplate to make the HTTP calls and add the required Authorization header:

```
public TwitterTemplate(String consumerKey, String consumerSecret, String accessToken, String accessTokenSecret)
    // creates a RestTemplate that adds the OAuth1 Authorization header to each request before it is executed
    restTemplate = ProtectedResourceClientFactory.create(consumerKey, consumerSecret, accessToken, accessTokenS
}
```

An example API call with RestTemplate is shown below:

```
public TwitterProfile getUserProfile(String screenName) {
    return new TwitterProfile(restTemplate.getForObject("https://api.twitter.com/1/users/show.json?screen_name=
}
```

4. Connecting to Service Providers

4.1 Introduction

In the previous chapter, you learned how Spring Social's *Service Provider 'Connect' Framework* can be used to manage user connections between your application and 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 works with ServiceProviders to coordinate the connection flow. `ConnectController` takes care of redirecting the user to the service provider for authorization and responding to the callback after authorization. At each step, `ConnectController` delegates to a `ServiceProvider` to handle the finer details such as obtaining a request token and creating connections.

4.2 Registering service providers

Because `ConnectController` collaborates with ServiceProviders to establish connections, you'll first need to register one or more `ServiceProvider` implementations as beans in the Spring context. `ConnectController` will discover any bean of type `ServiceProvider` in the Spring context and delegate to it as requested by users of your application.

The following configuration class registers `ServiceProvider` implementations for Twitter, Facebook, and TripIt using Spring's Java configuration style:

```
package org.springframework.social.showcase.config.connect;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.social.facebook.connect.FacebookServiceProvider;
import org.springframework.social.tripit.connect.TripItServiceProvider;
import org.springframework.social.twitter.connect.TwitterServiceProvider;

@Configuration
public class ServiceProviderConfig {

    @Bean
    public TwitterServiceProvider twitter(@Value("${twitter.consumerKey}") String consumerKey,
            @Value("${twitter.consumerSecret}") String consumerSecret, ConnectionRepository connectionReposito
        return new TwitterServiceProvider(consumerKey, consumerSecret, connectionRepository);
    }

    @Bean
    public FacebookServiceProvider facebook(@Value("${facebook.appId}") String appId,
            @Value("${facebook.appSecret}") String appSecret, ConnectionRepository connectionRepository) {
        return new FacebookServiceProvider(appId, appSecret, connectionRepository);
    }

    @Bean
    public TripItServiceProvider tripit(@Value("${tripit.consumerKey}") String consumerKey,
            @Value("${tripit.consumerSecret}") String consumerSecret, ConnectionRepository connectionRepository
        return new TripItServiceProvider(consumerKey, consumerSecret, connectionRepository);
    }
}
```

Each `ServiceProvider` should be configured with the client key and secret that were assigned to it when the application was registered with the service provider. Because the consumer key and secret may be different across environments (e.g., test, production, etc) it is recommended that these values be externalized. Here, the consumer key and secret are provided to the `twitter()` method as placeholder variables to be resolved by Spring's property placeholder support.

ServiceProviders are also given a `ConnectionRepository` at construction. When managing connections, a `ServiceProvider` needs a place to store those connections. Therefore, a `ServiceProvider` delegates to a `ConnectionRepository` for persisting connections. Spring Social supports JDBC-based connection storage with `JdbcConnectionRepository`, which itself is constructed with a `DataSource` and a `TextEncryptor`:.

```

package org.springframework.social.showcase.config.connect;

import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.social.connect.jdbc.JdbcConnectionRepository;
import org.springframework.social.connect.support.ConnectionRepository;
import org.springframework.security.crypto.encrypt.TextEncryptor;

@Configuration
public class ConnectionRepositoryConfig {

    @Bean
    public ConnectionRepository connectionRepository(DataSource dataSource, TextEncryptor textEncryptor) {
        return new JdbcConnectionRepositoy(dataSource, textEncryptor);
    }

}
```

`JdbcConnectionRepository` 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:

```

package org.springframework.social.showcase.config.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.crypto.encrypt.Encryptors;
import org.springframework.security.crypto.encrypt.TextEncryptor;

@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:

```
package org.springframework.social.showcase.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.crypto.encrypt.Encryptors;
import org.springframework.security.crypto.encrypt.TextEncryptor;

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

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

}
```

Configuring service providers in XML

Up to this point, the service provider configuration has been done using Spring's Java-based configuration style. You can configure Spring Social's service providers in either Java configuration or XML. Here's the XML equivalent of the service provider configuration:

```
<?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

    <!-- Configure a Twitter service provider -->
    <bean class="org.springframework.social.twitter.connect.TwitterServiceProvider">
        <constructor-arg value="${twitter.consumerKey}" />
        <constructor-arg value="${twitter.consumerSecret}" />
        <constructor-arg ref="connectionRepository" />
    </bean>

    <!-- Configure a Facebook service provider -->
    <bean class="org.springframework.social.facebook.connect.FacebookServiceProvider">
        <constructor-arg value="${facebook.appId}" />
        <constructor-arg value="${facebook.appSecret}" />
        <constructor-arg ref="connectionRepository" />
    </bean>

    <!-- Configure a TripIt service provider -->
    <bean class="org.springframework.social.tripit.connect.TripItServiceProvider">
        <constructor-arg value="${tripit.consumerKey}" />
        <constructor-arg value="${tripit.consumerSecret}" />
        <constructor-arg ref="connectionRepository" />
    </bean>

    <!-- Configure a connection repository through which account-to-provider connections will be stored -->
    <bean id="connectionRepository" class="org.springframework.social.connect.jdbc.JdbcConnectionRepository">
        <constructor-arg ref="dataSource" />
        <constructor-arg ref="textEncryptor" />
    </bean>

</beans>
```

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

    <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>
```

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

4.3 Creating connections with `ConnectController`

With one or more `ServiceProvider` beans configured, `ConnectController` will be able to coordinate the connection process for those providers. `ConnectController` is a Spring MVC controller and can be configured as a bean in your application's Spring MVC configuration as follows:

```
package org.springframework.social.showcase.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.social.web.connect.ConnectController;
import org.springframework.stereotype.Component;

@Configuration
```

```
public class ConnectControllerConfig {

    @Bean
    public ConnectController connectController(@Value("${application.url}") String applicationUrl) {
        return new ConnectController(applicationUrl);
    }

}
```

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

```
<bean class="org.springframework.social.web.connect.ConnectController">
    <constructor-arg value="${application.url}" />
</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 `ServiceProvider`s to handle the specifics for each protocol. `ConnectController` will discover `ServiceProvider`s as beans in the Spring context. It will select a specific `ServiceProvider` to use by matching the provider'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 `ServiceProvider` whose `getId()` returns "twitter" will be used.

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, the JSP resolved from "connect/twitterConnect" might include the following form:

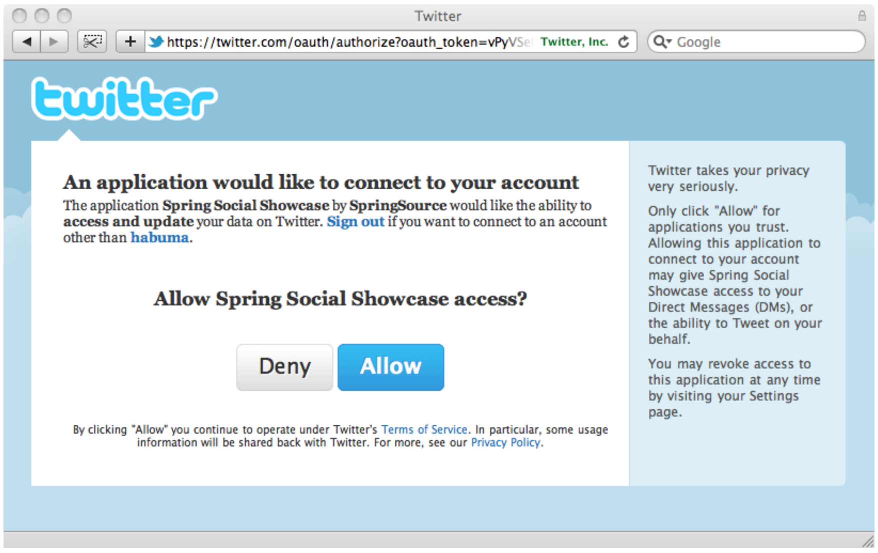
```
<form action="<c:url value="/connect/twitter" />" method="POST">
    <p>You haven't created any connections with Twitter yet. Click the button to create
       a connection between your account and your Twitter profile.
       (You'll be redirected to Twitter where you'll be asked to authorize the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


Twitter

https://twitter.com/oauth/authorize?oauth_token=vPyVSe Twitter, Inc. Google

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.

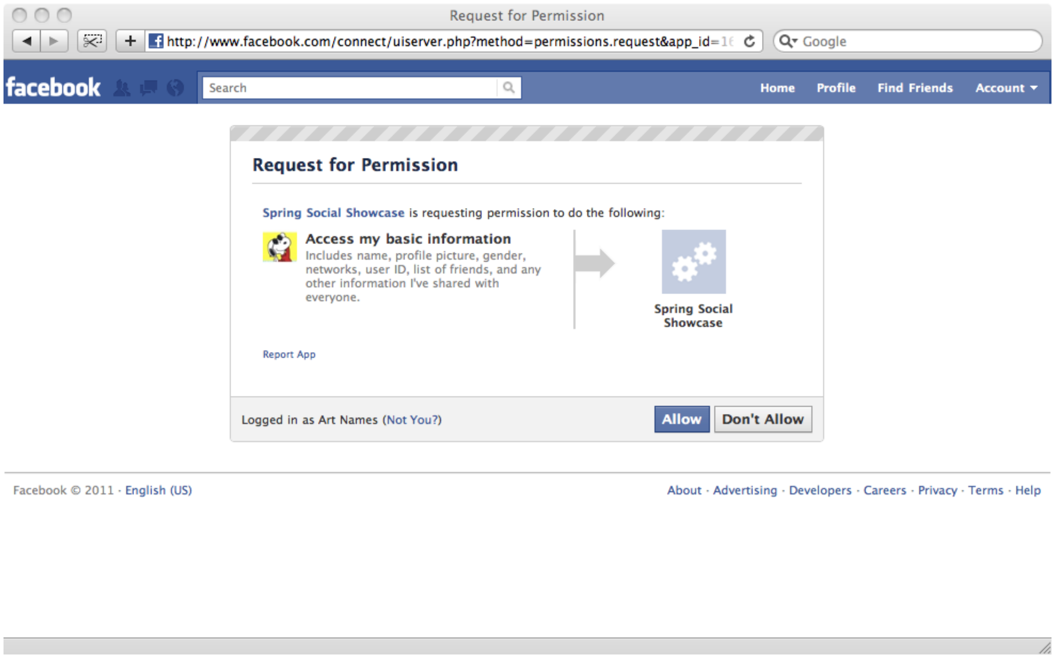
Only click "Allow" for applications you trust. Allowing this application to connect to your account may give Spring Social Showcase access to your Direct Messages (DMs), or the ability to Tweet on your behalf.

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

[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.

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 on 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>
```

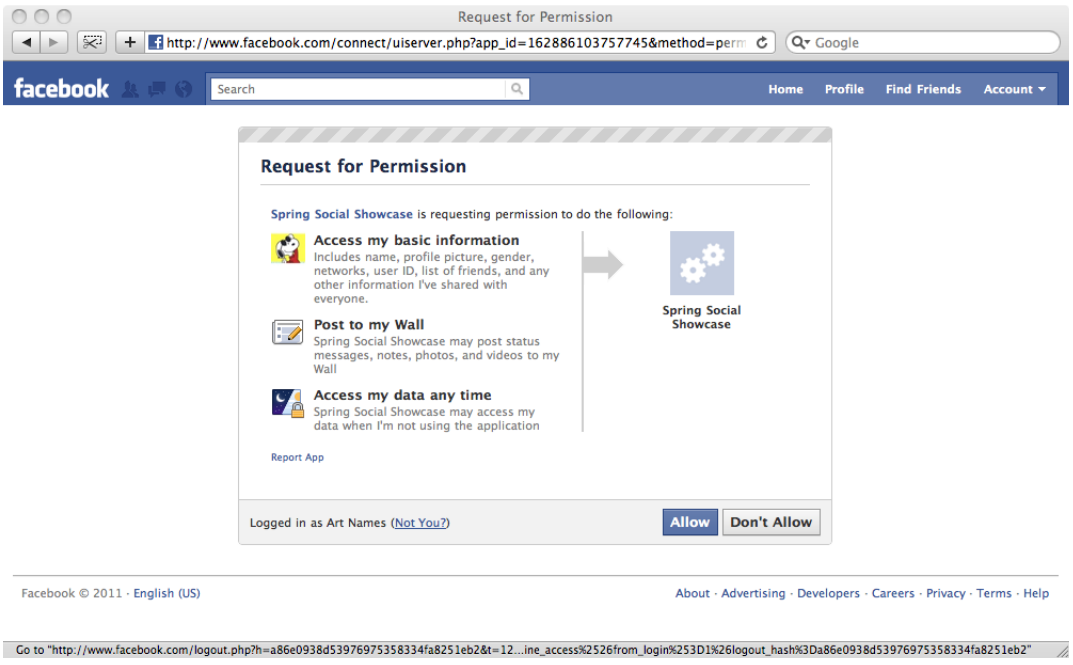
The hidden "scope" field contains the scope values to be passed along to Facebook's authorization URL. In this case, "publish_stream" requests permission to post to a user's wall. In addition, "offline_access" requests permission to access Facebook on behalf of a user even when the user isn't using the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

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 `ServiceProviderConnection`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<S> {

    void preConnect(ServiceProvider<S> provider, WebRequest request);

    void postConnect(ServiceProvider<S> provider, ServiceProviderConnection<S> 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 established between the member account and the provider profile.

For example, suppose that after a connection is made, you want to immediately tweet that connection to the user's Twitter timeline. To accomplish that, you might write the following connection interceptor:

```
package org.springframework.social.showcase.twitter;
import org.springframework.social.connect.ServiceProvider;
import org.springframework.social.connect.ServiceProviderConnection;
import org.springframework.social.twitter.DuplicateTweetException;
import org.springframework.social.twitter.TwitterOperations;
import org.springframework.social.web.connect.ConnectInterceptor;
import org.springframework.util.StringUtils;
import org.springframework.web.context.request.WebRequest;

public class TweetAfterConnectInterceptor implements ConnectInterceptor<TwitterOperations> {

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

    public void postConnect(ServiceProvider<TwitterOperations> provider, ServiceProviderConnection<TwitterOpera
        connection.getServiceApi().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.url}") String applicationUrl) {
    ConnectController controller = new ConnectController(applicationUrl);
    controller.addInterceptor(new TweetAfterConnectInterceptor());
    return controller;
}
```

Or, as configured in XML:

```
<bean class="org.springframework.social.web.connect.ConnectController">
    <constructor-arg value="http://localhost:8080/myapplication" />
    <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 accept service operations type matching the service provider's operations type. In the example given here, only connections made through a service provider whose operation type is `TwitterOperations` will trigger the interceptor's methods.

5. Signing in with Service Provider Accounts

5.1 Introduction

Once a connection has been established between a user's consumer account and their service provider profile, that connection can be used to authenticate them to the consumer application by asking them to sign in to the service provider. Spring Social supports such service provider-based authentication with Twitter and Facebook.

5.2 Sign in with Twitter

Spring Social's `TwitterSigninController` is a Spring MVC controller that processes the "Sign in with Twitter" flow described at http://dev.twitter.com/pages/sign_in_with_twitter. Essentially, this process is an OAuth 1 authorization flow, quite similar to the flow that `ConnectController` processes for connecting an account with Twitter. The key difference is that instead of presenting the user with an authorization page that asks for permission to be granted to the application, the sign in flow presents a simple authentication page that only asks the user to sign in to Twitter.

At the end of process, `TwitterSigninController` attempts to find a previously established connection and uses the connected account to authenticate the user to the application.

To add "Sign in with Twitter" capability to your Spring application, configure `TwitterSigninController` as a bean in your Spring MVC application:

```
<bean class="org.springframework.social.twitter.web.TwitterSigninController">
    <constructor-arg ref="twitterProvider" />
    <constructor-arg ref="connectionRepository" />
    <constructor-arg ref="signinService" />
    <constructor-arg value="http://localhost:8080/myapplication" />
</bean>
```

`TwitterSigninController` is constructed with four arguments:

- A reference to a `TwitterServiceProvider` bean. `TwitterSigninController` will use this to negotiate the connection with Twitter.
- A reference to a connection repository bean. After signing into Twitter, `TwitterSigninController` will use the connection repository to find an account ID that is connected to the Twitter profile.
- A reference to an implementation of `SignInService`, used to perform the actual authentication into the application.
- The application's base URL, used to construct a callback URL for the OAuth flow.

`TwitterSigninController`'s constructor is annotated with `@Inject`, so it's not necessary to explicitly wire any of its arguments except for the application URL. Optimizing the configuration for autowiring, the `TwitterSigninController` bean looks like this:

```
<bean class="org.springframework.social.twitter.web.TwitterSigninController">
    <constructor-arg value="http://localhost:8080/myapplication" />
</bean>
```

`TwitterSigninController` supports the following flow:

- `POST /signin/twitter` - Initiates the "Sign in with Twitter" flow, fetching a request token from Twitter and redirecting to Twitter's authentication page.
- `GET /signin/twitter?oauth_token={request token}&oauth_verifier={verifier}` - Receives the authentication callback from Twitter, 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, `TwitterSigninController` will redirect to a signup URL. The default signup URL is "/signup" (relative to the application root).

`TwitterSigninController` handles the authentication flow with Twitter, but relies on an implementation of `SignInService` to perform the actual authentication into the application. `SignInService` is defined as follows:

```
public interface SignInService<T extends Serializable> {
    void signIn(T accountId);
}
```

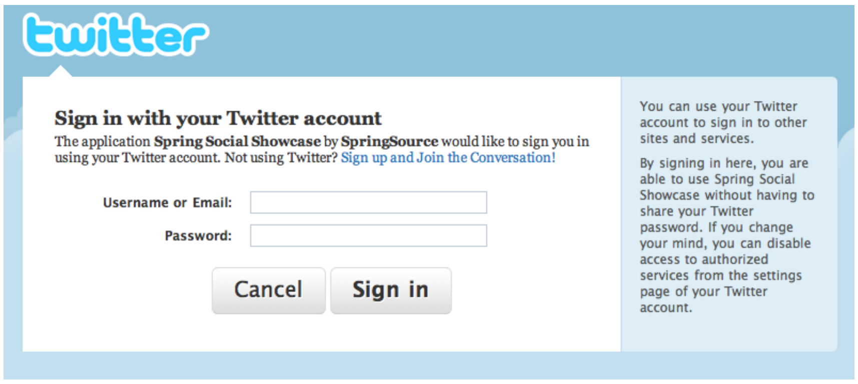
Different applications will implement security differently, so each application must implement `SignInService` in a way that fits its unique security scheme. As an example, suppose that an application's security is based Spring Security and simply uses a user's account ID as their principal. In that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

```
package org.springframework.social.showcase;

import java.io.Serializable;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.social.web.signin.SignInService;

public class AccountIdAsPrincipalSigninService implements SignInService<Long> {
    public void signIn(Long accountId) {
        SecurityContextHolder.getContext().setAuthentication(new UsernamePasswordAuthenticationToken(accountId,
    }
}
```

The last thing to do is to add a "Sign in with Twitter" button to your application:

```
<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"/>" /><
</form>
```

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!

Username or Email:

Password:

Cancel Sign in

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.

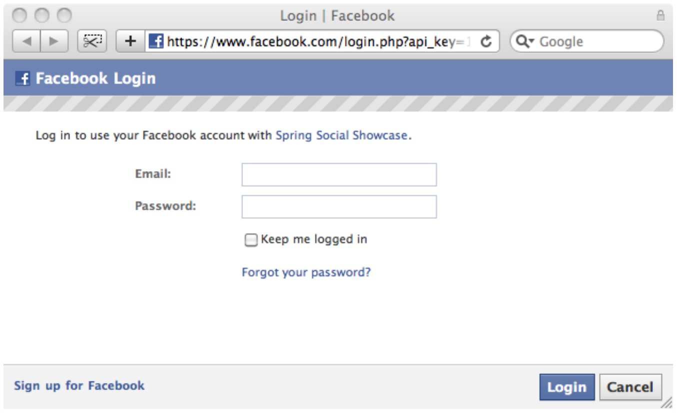
After signing in, the flow will redirect back to the application to complete the sign in process.

If the user has already signed into Twitter prior to clicking the sign in button, Twitter will redirect the flow back to the application without presenting the user with a sign in page.

5.3 Sign in with Facebook

Spring Social's `FacebookSigninController`, when paired with Facebook's `<fb:login-button>` XFBML tag[1], enables a user to authenticate to an application by first signing into Facebook.

Facebook's XFBML tag collection includes `<fb:login-button>`, which displays a login button. When a user clicks on that button, they will be presented with a Facebook Login dialog that looks similar to this:


Login | Facebook

https://www.facebook.com/login.php?api_key=

Facebook Login

Log in to use your Facebook account with Spring Social Showcase.

Email:

Password:

Keep me logged in

Forgot your password?

Sign up for Facebook

Login Cancel

After successfully signing into Facebook, a cookie will be written that contains an access token. If the user has previously established a connection between their account and their Facebook profile, this access token will be the same token that was issued before. `FacebookSigninController` works by extracting that access token from the cookie and using it to lookup a connected account and authenticate to that account.

To enable "Sign in with Facebook" functionality in your application, start by configuring `FacebookSigninController` as a bean in the Spring MVC configuration:

[1]http://developers.facebook.com/docs/reference/plugins/login/

```
<bean class="org.springframework.social.facebook.web.FacebookSigninController">
    <constructor-arg ref="facebookProvider" />
    <constructor-arg ref="connectionRepository" />
    <constructor-arg ref="signinService" />
</bean>
```

As with `TwitterSigninController`, `FacebookSigninController` depends on a connection repository to lookup connections and a sign in service to handle the actual application authentication. It also needs a reference to a `FacebookServiceProvider` bean that it will use to negotiate the connection with Facebook.

`FacebookSigninController`'s constructor is annotated with `@Inject`, so it is not necessary to explicitly wire these dependencies. The `FacebookSigninController` configuration optimized for autowiring takes a simpler form:

```
<bean class="org.springframework.social.facebook.web.FacebookSigninController"/>
```

However, since Facebook's sign in process is not based on an OAuth 1 flow, `FacebookSigninController` does not need the base URL to create a callback URL from. Instead, `FacebookSigninController` needs to know the application's Facebook app id and secret that were issued to the application when it was first registered in Facebook. It uses the app id to find the cookie and the application secret to verify the authenticity of the cookie by calculating and comparing signatures.

With `FacebookSigninController` configured, the next thing to do is to load and initialize Facebook's JavaScript library and XFBML in your application's sign in page. Spring Social provides a convenient JSP tag to handle that work for you. Simply declare the Spring Social Facebook tag library in your JSP:

```
<%@ taglib uri="http://www.springframework.org/spring-social/facebook/tags" prefix="facebook" %>
```

Then use the `<facebook:init>` tag:

```
<facebook:init appId="@facebookProvider.appId" />
```

Note that the tag prefix cannot be "fb", as that would create a conflict between Spring Social's Facebook JSP tag library and Facebook's XFBML tags (which use "fb" as their prefix).

The `<facebook:init>` will initialize the Facebook JavaScript library with your application's App ID. This is the appId assigned to your application when you registered it with Facebook. Here the App ID is specified using the Spring Expression Langauge to be the value of the `appId` property of the bean whose ID is "facebookProvider".

Now that the Facebook JavaScript API is initialized you just need to add the `<fb:login-button>` to the page:

```
<form id="fb_signin" action="<c:url value="/signin/facebook"/>" method="POST">
    <div id="fb-root"></div>
    <p><fb:login-button onlogin="$('#fb_signin').submit();" v="2" length="long">Signin with Facebook</fb:login-
</form>
```

Notice that here `<fb:login-button>` is within a form that submits to "/signin/facebook" (the URL that `FacebookSigninController` handles). Although it's not strictly required that `<fb:login-button>` be within this form, the form itself is necessary to trigger `FacebookSigninController`. `<fb:login-button>` doesn't directly submit this form, so it doesn't matter whether or not it is in the form.

`<fb:login-button>`'s `onlogin` attribute indicates what action should be taken upon successful Facebook login. In this case, `onlogin` is set to use jQuery to submit the form, triggering `FacebookSigninController` to authenticate the user to the application.

5.4 Signing up after a failed sign in

With both `TwitterSigninController` and `FacebookSigninController`, the flow will redirect to a signup page if no connection can be found for the obtained access token. By default, the signup 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 `TwitterSigninController` sets the signup URL to "/newUser":

```
<bean class="org.springframework.social.twitter.web.TwitterSigninController">
    <constructor-arg value="http://localhost:8080/myapplication" />
    <property name="signupUrl" value="/newUser" />
</bean>
```

After the user has successfully signed up in your application, you can complete the connection between the provider and the newly created account by calling `ProviderSignInUtils.handleConnectPostSignUp()`:

```
ProviderSignInUtils.handleConnectPostSignUp(accountId, request);
```

6. Working with Service 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

Twitter's social offering is rather simple: Enable users to post whatever they're thinking, 140 characters at a time.

Spring Social's `TwitterTemplate` (which implements `TwitterOperations`) offers several options for applications to integrate with Twitter.

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 apiKey = "..."; // The application's API/Consumer key
String apiSecret = "..."; // The application's API/Consumer secret
String accessToken = "..."; // The access token granted after OAuth authorization
String accessTokenSecret = "..."; // The access token secret granted after OAuth authorization
TwitterOperations twitter = new TwitterTemplate(apiKey, apiSecret, accessToken, accessTokenSecret);
```

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

```
TwitterOperations twitter = new TwitterTemplate();
```

When constructed with the default constructor, `TwitterTemplate` will allow a few simple operations that do not require authorization, such as searching. Other operations, such as tweeting will fail with an `AccountNotConnectedException` 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 `TwitterOperations` by calling the `getServiceApi()` method on one of the connections given by `TwitterServiceProvider`'s `getConnections()` method. For instance:

```
TwitterOperations twitter = twitterProvider.getConnections(accountId).get(0).getServiceApi();
```

Here, `TwitterServiceProvider` is being asked for a `TwitterOperations` that was created using connection details established previously via the service provider's `connect()` method or through `ConnectController`.

Once you have `TwitterOperations`, you can perform a variety of operations against Twitter.

Retrieving a user's Twitter profile data

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

```
TwitterProfile profile = twitter.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 = twitter.getUserProfile("habuma");
```

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

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

Tweeting

To post a message to Twitter using `TwitterTemplate` the simplest thing to do is to pass the message to the `updateStatus()` method:

```
twitter.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);
twitter.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);
twitter.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();
twitter.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.

`TwitterTemplate` 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 = twitter.getPublicTimeline();
```

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

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

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

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

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

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

Searching Twitter

`TwitterTemplate` 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 = twitter.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 = twitter.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 = twitter.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. `TwitterTemplate`'s `sendDirectMessage()` method can be used to send a direct message to another user:

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

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

```
List<DirectMessage> twitter.getDirectMessagesReceived();
```

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

6.3 Facebook

Spring Social's `FacebookOperations` 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
FacebookOperations facebook = new FacebookTemplate(accessToken);
```

Or, if you are using the service provider framework described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `FacebookTemplate` by calling the `getServiceApi()` method on one of the connections given by `FacebookServiceProvider`'s `getConnections()` method:

```
FacebookOperations facebook = facebookProvider.getConnections(accountId).get(0).getServiceApi();
```

Here, `FacebookServiceProvider` is being asked to give back a `FacebookOperations` created using the connection details for the given account ID. The connection should have been created beforehand via the service provider's `connect()` method or using `ConnectController`.

With a `FacebookOperations` in hand, there are several ways you can use it to interact with Facebook 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 Facebook profile data using `FacebookOperations`' `getUserProfile()` method:

```
FacebookProfile profile = facebook.getUserProfile();
```

The `FacebookProfile` object will contain basic profile information about the authenticating user, including their first and last name, their email address, and their Facebook ID.

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

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

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

```
String profileUrl = facebook.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:

```
List<String> friendIds = facebook.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 = facebook.getUserProfile(friendIds.get(0));
```

Posting to a user's wall

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

```
facebook.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.");
facebook.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.

Publishing to Facebook

Facebook's Graph API allows authenticated users to publish data to several of its object types. `FacebookOperations` enables arbitrary publication via the Graph API with its `publish()` method.

For example, in the previous section you saw how to post a message to the user's Facebook wall using `updateStatus()`. Alternatively you could have accomplished the same thing using `publish()` like this:

```
MultiValueMap<String, String> data = new LinkedMultiValueMap<String, String>();
data.set("message", message);
publish("me", "feed", data);
```

The first argument to the `publish()` method is the object to publish to--in this case "me" indicates the authenticated user. The second argument is the connection associated with the object--"feed" indicates that it is the user's Facebook wall. Finally, the third argument is a `MultiValueMap` containing data to be published. In this case, it only contains a "message" to be posted to the user's wall.

You can read more about what graph API objects and connections Facebook supports for publishing at http://developers.facebook.com/docs/api#editing

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 `LinkedInOperations` 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 apiKey = "..."; // The application's API/Consumer key
String apiSecret = "..."; // The application's API/Consumer secret
String accessToken = "..."; // The access token granted after OAuth authorization
String accessTokenSecret = "..."; // The access token secret granted after OAuth authorization
LinkedInOperations linkedin = new LinkedInTemplate(apiKey, apiSecret, accessToken, accessTokenSecret);
```

If you're using the service provider framework described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `LinkedInTemplate` by calling the `getServiceApi()` method on one of the connections given by `LinkedInServiceProvider`'s `getConnections()` method. For example:

```
LinkedInOperations linkedin = linkedinProvider.getConnections(accountId).get(0).getServiceApi();
```

In this case, the `getServiceOperations()` is asked to return a `LinkedInOperations` instance created using connection details established using the service provider's `connect()` method or via `ConnectController`.

Once you have a `LinkedInOperations` 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 `TripItOperations` 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 apiKey = "..."; // The application's API/Consumer key
String apiSecret = "..."; // The application's API/Consumer secret
String accessToken = "..."; // The access token granted after OAuth authorization
String accessTokenSecret = "..."; // The access token secret granted after OAuth authorization
TripItOperations tripit = new TripItTemplate(apiKey, apiSecret, accessToken, accessTokenSecret);
```

If you're using Spring Social's service provider framework (as described in Chapter 2, *Service Provider 'Connect' Framework*), you can get a `TripItOperations` by calling the `getServiceApi()` method on one of the connections given by `TripItServiceProvider`'s `getConnections()` method:

```
TripItOperations tripit = tripitProvider.getConnections(accountId).get(0).getServiceApi();
```

In this case, `TripItServiceProvider` is being asked to give a `TripItOperations` constructed using connection data established beforehand using the service provider's `connect()` method or via `ConnectController`.

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

Retrieving a user's TripIt profile data

`TripItOperations`' `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 `GitHubOperations` 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
GitHubOperations github = new GitHubTemplate(accessToken);
```

If you are using the service provider framework described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `GitHubTemplate` by calling the `getServiceApi()` method on one of the connections given by `GitHubServiceProvider`'s `getConnections()` method:

```
GitHubOperations github = githubProvider.getConnections(accountId).get(0).getServiceApi();
```

Here, `GitHubServiceProvider` is being asked for a `GitHubOperations` that was created using the connection details for the given account ID. The connection should have been created beforehand via the service provider's `connect()` method or using `ConnectController`.

With a `GitHubOperations` 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 `GitHubOperations`'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 `GowallaOperations` 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
```

```
GowallaOperations gowalla = new GowallaTemplate(accessToken);
```

If you are using the service provider framework described in Chapter 2, *Service Provider 'Connect' Framework*, you can get an instance of `GowallaTemplate` by calling the `getServiceApi()` method on one of the connections given by `GowallaServiceProvider`'s `getConnections()` method:

```
GowallaOperations gowalla = gowallaProvider.getConnections(accountId).get(0).getServiceApi();
```

Here, `GowallaServiceProvider` is being asked for a `GowallaOperations` that was created using the connection details for the given account ID. The connection should have been created beforehand via the service provider's `connect()` method or using `ConnectController`.

With a `GowallaOperations` 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

`GowallaOperations` 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.