Spring Android Reference Manual

Roy Clarkson

Spring Android Reference Manual

by Roy Clarkson

1.0.0.M3

Table of Contents

1. Spring Android Overview

1.1 Introduction

The Spring Android project supports the usage of the Spring Framework in an Android environment. This includes the ability to use RestTemplate as the REST client for your Android applications. Spring Android also provides support for integrating Spring Social functionality into your Android application, which includes a robust OAuth based, authorization client and implementations for popular social web sites, such as Twitter and Facebook.

2. Spring Android Rest Template Module

2.1 Introduction

Spring's RestTemplate is a robust, popular Java-based REST client. The Spring Android Rest Template Module provides a version of RestTemplate that works in an Android environment.

2.2 Overview

The RestTemplate object is the heart of the Spring Android Rest Template library. When you create a new RestTemplate instance, the constructor sets up several supporting objects that make up the RestTemplate functionality.

HttpComponents HttpClient 4.x

The HttpComponents HttpClient [http://hc.apache.org/httpcomponents-client-ga/index.html] is the native http client available on the Android platform. Within Spring Android Rest Template this HttpClient is made available through the HttpComponentsClientHttpRequestFactory. The HttpComponentsClientHttpRequestFactory is set as the default RequestFactory when you create a new RestTemplate instance, so you are not required to set it manually.

Object to JSON Marshaling

Object to JSON marshaling in Spring Android Rest Template requires the use of a third party JSON mapping library. The Jackson JSON Processor [http://jackson.codehaus.org] is used within the MappingJacksonHttpMessageConverter to provide this marshaling functionality. The media type supported by this message converter is "application/json".

The MappingJacksonHttpMessageConverter is conditionally loaded when you create a new RestTemplate instance. If the Jackson dependencies are found in your classpath, the message converter will be automatically added and available for use in REST operations. See the How to Get section for more details on including Jackson in your project. Additionally, the Usage Examples section provides code samples.

Object to XML Marshaling

Object to XML marshaling in Spring Android Rest Template requires the use of a third party XML mapping library. The Simple XML serializer [http://simple.sourceforge.net] is used within the SimpleXmlHttpMessageConverter to provide this marshaling functionality. The media types supported by this message converter are "application/xml", "text/xml", and "application/*+xml".

The SimpleXmlHttpMessageConverter is conditionally loaded when you create a new RestTemplate instance. If the Simple dependency is found in your classpath, the message converter will be automatically added and available for use in REST operations. See the How to Get section for more details on including Simple in your project. Additionally, the Usage Examples section provides code samples.

RSS and Atom Support

RSS and Atom feed support in Spring Android Rest Template requires the use of a third party feed reader library. The Android ROME Feed Reader [http://code.google.com/p/android-rome-feed-reader] is used within the SyndFeedHttpMessageConverter, RssChannelHttpMessageConverter, and the AtomFeedHttpMessageConverter to provide this functionality. The media types supported by these message converters are "application/rss+xml" and "application/atom+xml".

The SyndFeedHttpMessageConverter is conditionally loaded when you create a new RestTemplate instance. If the Android ROME dependencies are found in your classpath, the message converter will be automatically added and available for use in REST operations. See the How to Get section for more details on including Android ROME in your project. Additionally, the Usage Examples section provides code samples.

Because the SyndFeedHttpMessageConverter provides a higher level abstraction around RSS and Atom feeds, the RssChannelHttpMessageConverter, and AtomFeedHttpMessageConverter are not automatically added when you create a new RestTemplate instance. If you prefer to use one of these message converters then you have to manually add it to the RestTemplate instance.

2.3 How to get

Add the spring-android-rest-template artifact to your classpath:

```
<dependency>
    <groupId>org.springframework.android</groupId>
    <artifactId>spring-android-rest-template</artifactId>
    <version>${spring-android-version}</version>
</dependency>
```

Unfortunately, Google's provided Android toolset does not include dependency management support. However, through the use of third party tools, you can use Maven to manage dependencies and build your Android app. See the Spring Android and Maven section for more information.

Spring Android Rest Template supports several optional libraries. These optional libraries are used by different Http Message Converters within Rest Template. If you would like to make use of these Message Converters, then you need to include the corresponding libraries in your classpath.

Jackson JSON Processor

The MappingJacksonHttpMessageConverter is used to marshal Objects to JSON. The Jackson [http://jackson.codehaus.org] library provides this functionality.

Add the following Jackson dependencies to your classpath to enable this Message Converter.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>${jackson-version}</version>
```

```
</dependency>
```

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-core-asl</artifactId>
    <version>${jackson-version}</version>
</dependency>
```

Simple XML Serializer

The SimpleXmlHttpMessageConverter is used to marshal Objects to XML. Simple [http://simple.sourceforge.net] is an XML serialization and configuration framework for Java that is compatible with Android.

Add the following Simple dependency to your classpath to enable this Message Converter.

```
<dependency>
    <groupId>org.simpleframework</groupId>
    <artifactId>simple-xml</artifactId>
    <version>${simple-version}</version>
</dependency>
```

Android ROME Feed Reader

The RssChannelHttpMessageConverter, AtomFeedHttpMessageConverter, and SyndFeedHttpMessageConverter are used to process RSS and Atom feeds. Android ROME Feed Reader [http://code.google.com/p/android-rome-feed-reader] is a port of the popular ROME library that is compatible with Android.

Add the following Android ROME dependencies to your classpath to enable these Message Converters. This library depends on a forked version of JDOM to work on Android 2.1 and earlier. The JDOM library addresses a bug [http://www.jdom.org/pipermail/jdom-interest/2009-July/016345.html] in the Android XML parser.

```
<dependency>
    <groupId>com.google.code.android-rome-feed-reader</groupId>
    <artifactId>android-rome-feed-reader</artifactId>
    <version>${android-rome-version}</version>
</dependency>
```

```
<dependency>
    <groupId>org.jdom</groupId>
    <artifactId>jdom</artifactId>
    <version>${jdom-fork-version}</version>
</dependency>
```

2.4 Usage Examples

Using Rest Template, it's easy to invoke RESTful APIs. Below are several usage examples that illustrate the different methods for making RESTful requests.

All of the following examples are based on a sample Android application [http://git.springsource.org/spring-mobile/samples]. You can retrieve the source code for the sample app with the following command:

```
$ git clone git://git.springsource.org/spring-mobile/samples.git
```

Basic Usage Example

The following example shows a query to google for the search term "SpringSource".

```
RestTemplate restTemplate = new RestTemplate();
String url = "https://ajax.googleapis.com/ajax/services/search/web?v=1.0&q={query}";
String result = restTemplate.getForObject(url, String.class, "SpringSource");
```

Retrieving JSON data via HTTP GET

Suppose you have defined a Java object you wish to populate from a RESTful web request that returns JSON content.

Define your object based on the JSON data being returned from the RESTful request:

```
public class Event {

    private Long id;

    private String title;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public String setTitle(String title) {
        this.title = title;
    }
}
```

Make the RestTemplate request:

```
String url = "http://mypretendservice.com/events";
RestTemplate restTemplate = new RestTemplate();
Event[] events = restTemplate.getForObject(url, Event[].class);
```

You can also set the Accept header for the request:

```
List<MediaType> acceptableMediaTypes = new ArrayList<MediaType>();
acceptableMediaTypes.add(new MediaType("application","json"));

HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAccept(acceptableMediaTypes);

HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

String url = "http://mypretendservice.com/events";

RestTemplate restTemplate = new RestTemplate();
ResponseEntity<Event[]> responseEntity = restTemplate.exchange(url, HttpMethod.GET, requestEntity, Event[].clas
Event[] events = responseEntity.getBody();
```

Retrieving XML data via HTTP GET

Using the same Java object we defined earlier, we can modify the requests to retrieve XML.

Define your object based on the XML data being returned from the RESTful request. Note the annotations used by Simple to marshal the object:

```
@Root
public class Event {

    @Element
    private Long id;

    @Element
    private String title;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public String setTitle(String title) {
        this.title = title;
    }
}
```

To marshal an array of events from xml, we need to define a wrapper class for the list:

```
@Root(name="events")
public class EventList {

   @ElementList(inline=true)
   private List<Event> events;

   public List<Event> getEvents() {
      return events;
   }

   public void setEvents(List<Event> events) {
       this.events = events;
   }
}
```

Make the RestTemplate request:

```
String url = "http://mypretendservice.com/events";
RestTemplate restTemplate = new RestTemplate();
EventList eventList = restTemplate.getForObject(url, EventList.class);
```

You can also specify the Accept header for the request:

```
List<MediaType> acceptableMediaTypes = new ArrayList<MediaType>();
acceptableMediaTypes.add(new MediaType("application","xml"));

HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAccept(acceptableMediaTypes);

HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

String url = "http://mypretendservice.com/events";

RestTemplate restTemplate = new RestTemplate();
ResponseEntity<EventList> responseEntity = restTemplate.exchange(url, HttpMethod.GET, requestEntity, EventList.
EventList eventList = responseEntity.getBody();
```

Send JSON data via HTTP POST

POST a Java object you have defined to a RESTful service that accepts JSON data.

Define your object based on the JSON data expected by the RESTful request:

```
public class Message
{
    private long id;

    private String subject;
```

```
    private String text;

    public void setId(long id) {
        this.id = id;
    }

    public long getId() {
        return id;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getSubject() {
        return subject;
    }

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }
}
```

Make the RestTemplate request. In this example, the request responds with a string value:

```
Message message = new Message();
message.setId(555);
message.setSubject("test subject");
message.setText("test text");

String url = "http://mypretendservice.com/sendmessage";
RestTemplate restTemplate = new RestTemplate();
String response = restTemplate.postForObject(url, message, String.class);
```

You can also specify the Content-Type header in your request:

```
Message message = new Message();
message.setId(555);
message.setSubject("test subject");
message.setText("test text");

HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setContentType(new MediaType("application","json"));

HttpEntity<Message> requestEntity = new HttpEntity<Message>(message, requestHeaders);

String url = "http://mypretendservice.com/sendmessage";

RestTemplate restTemplate = new RestTemplate();
ResponseEntity<String> responseEntity = restTemplate.exchange(url, HttpMethod.POST, requestEntity, String.class
String result = responseEntity.getBody();
```

Retrieve RSS or Atom feed

The following is a basic example of loading an RSS feed:

```
String url = "http://mypretendservice.com/rssfeed";
RestTemplate restTemplate = new RestTemplate();
SyndFeed = restTemplate.getForObject(url, SyndFeed.class);
```

It is possible that you need to adjust the Media Type associated with the SyndFeedHttpMessageConverter. By default, the converter is associated with "application/rss+xml" and "application/atom+xml". An RSS feed might instead have a media type of "text/xml", for example. The following code illustrates how to set the media type.

```
String url = "http://mypretendservice.com/rssfeed";

SyndFeedHttpMessageConverter converter = new SyndFeedHttpMessageConverter();
List<MediaType> mediaTypes = new ArrayList<MediaType>();
mediaTypes.add(new MediaType("text","xml"));
converter.setSupportedMediaTypes(mediaTypes);

List<HttpMessageConverter<?>> messageConverters = new ArrayList<HttpMessageConverter<?>>();
messageConverters.add(converter);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setMessageConverters(messageConverters);
SyndFeed feed = restTemplate.getForObject(url, SyndFeed.class);
```

3. Spring Android Auth Module

3.1 Introduction

Many mobile applications today connect to external web services to access some type of data. These web services may be a third-party data provider, such as Twitter, or it may be an in house service for connecting to a corporate calendar, for example. In many of these cases, to access that data through the web service, you must authenticate and authorize an application on your mobile device. The goal of the spring-android-auth module is to address the need of an Android application to gain authorization to a web service.

There are many types of authorization methods and protocols, some custom and proprietary, while others are open standards. One protocol that is rapidly growing in popularity is OAuth. OAuth is an open protocol that allows users to give permission to a third-party application or web site to access restricted resources on another web site or service. The third-party application receives an access token with which it can make requests to the protected service. By using this access token strategy, a user's login credentials are never stored within an application, and are only required when authenticating to the service.

3.2 Overview

The initial release of the spring-android-auth module provides OAuth 1.x and 2.0 support in an Android application by utilizing Spring Social. It includes a SQLite repository, and Android compatible Spring Security encryption. The Spring Social project enables your applications to establish Connections with Software-as-a-Service (SaaS) Providers such as Facebook and Twitter to invoke Service APIs on behalf of Users. In order to make use of Spring Social on Android the following classes are available.

SQLite Connection Repository

The SQLiteConnectionRepository class implements the ConnectionRepository interface from Spring Social. It is used to persist the connection information to a SQLite database on the Android device. This connection repository is designed for a single user who accesses multiple service providers and may even have multiple accounts on each service provider.

If your device and application are used by multiple people, then a SQLiteUsersConnectionRepository class is available for storing multiple user accounts, where each user account may have multiple connections per provider. This scenario is probably not as typical, however, as many people do not share their phones or devices.

Encryption

The Spring Security Crypto library is not currently supported on Android. To take advantage of the encryption tools in Spring Security, the Android specific class, AndroidEncryptors has been provided in Spring Android. This class uses an Android compatible SecureRandom provider for generating byte array based keys using the SHA1PRNG algorithm.

3.3 How to get

Add the spring-android-auth artifact to your classpath:

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

To use one of the built in Spring Social providers, you can add it to your classpath:

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-twitter</artifactId>
    <version>${spring-social-version}</version>
    <exclusions>
        <exclusion>
            <!-- Exclude in favor of Spring Android RestTemplate -->
            <artifactId>spring-social-web</artifactId>
            <groupId>org.springframework.social</groupId>
        </exclusion>
        <exclusion>
            <!-- Provided by Android -->
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

3.4 Usage Examples

Below are several usage examples that illustrate how to use Spring Android with Spring Social.

The following examples are based on a sample Android application [http://git.springsource.org/spring-mobile/samples], which has Facebook and Twitter examples using Spring Social. You can retrieve the source code for the sample app with Git:

```
$ git clone git://git.springsource.org/spring-mobile/samples.git
```

Initializing the SQLite Database

SQLiteConnectionRepositoryHelper extends SQLiteOpenHelper [http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html]. Create a new instance by passing a context [http://developer.android.com/reference/android/content/Context.html] reference. Depending on your implementation, and to avoid memory leaks [http://developer.android.com/resources/articles/avoiding-memory-leaks.html], you will probably want to use the Application Context when creating a new instance of SQLiteConnectionRepositoryHelper. The name of the database file created is "spring_social_connection_repository.sqlite", and is created the first time the application attempts to open it.

```
Context context = getApplicationContext();
```

```
SQLiteOpenHelper repositoryHelper = new SQLiteConnectionRepositoryHelper(context);
```

Single User App Environment

This example show how to set up the ConnectionRepository for use with multiple connection factories.

To establish a ConnectionRepository, you will need the following objects.

```
ConnectionFactoryRegistry connectionFactoryRegistry;
SQLiteOpenHelper repositoryHelper;
ConnectionRepository connectionRepository;
```

The ConnectionFactoryRegistry stores the different Spring Social connections to be used in the application.

```
connectionFactoryRegistry = new ConnectionFactoryRegistry();
```

You can create a FacebookConnectionFactory, if your application requires Facebook connectivity.

```
// the App ID and App Secret are provided when you register a new Facebook application at facebook.com
String appId = "8ae8f060d81d51e90fadabaab1414a97";
String appSecret = "473e66d79ddc0e360851dc512fe0fb1e";

// Prepare a Facebook connection factory with the App ID and App Secret
FacebookConnectionFactory facebookConnectionFactory;
facebookConnectionFactory = new FacebookConnectionFactory(appId, appSecret);
```

Similarly, you can also create a TwitterConnectionFactory. Spring Social offers several different connection factories to popular services. Additionally, you can create your own connection factory based on the Spring Social framework.

```
// The consumer token and secret are provided when you register a new Twitter application at twitter.com
String consumerToken = "YR571S2JiVBOFyJS5MEg";
String consumerTokenSecret = "Kb8hS0luftwCJX3qVoyiLUMfZDtK1EozFoUkjNLUMx4";

// Prepare a Twitter connection factory with the consumer token and secret
TwitterConnectionFactory twitterConnectionFactory;
twitterConnectionFactory = new TwitterConnectionFactory(consumerToken, consumerTokenSecret)
```

After you create a connection factory, you can add it to the registry. Connection factories may be later retrieved from the registry in order to create new connections to the provider.

```
connectionFactoryRegistry.addConnectionFactory(facebookConnectionFactory);
connectionFactoryRegistry.addConnectionFactory(twitterConnectionFactory);
```

The final step is to prepare the connection repository for storing connections to the different providers.

```
// Create the SQLiteOpenHelper for creating the local database
Context context = getApplicationContext();
SQLiteOpenHelper repositoryHelper = new SQLiteConnectionRepositoryHelper(context);

// The connection repository takes a TextEncryptor as a parameter for encrypting the OAuth information
TextEncryptor textEncryptor = AndroidEncryptors.noOpText();

// Create the connection repository
ConnectionRepository connectionRepository = new SQLiteConnectionRepository(repositoryHelper,
 connectionFactoryRegistry, textEncryptor);
```

Encrypting OAuth Data

Spring Social supports encrypting the user's OAuth connection information within the ConnectionRepository through the use of a Spring Security TextEncryptor. The password and salt values are used to generate the encryptor's secret key. The salt value should be hex-encoded, random, and application-global. While this will encrypt the OAuth credentials stored in the database, it is not an absolute solution. When designing your application, keep in mind that there are already tools available for translating a DEX to a JAR file, and decompiling to source code. Because your application is distributed to a user's device, it is more vulnerable than if it were running on a web server, for example.

```
String password = "password";
String salt = "5c0744940b5c369b";
TextEncryptor textEncryptor = AndroidEncryptors.text(password, salt);
connectionRepository = new SQLiteConnectionRepository(repositoryHelper,
 connectionFactoryRegistry, textEncryptor);
```

During development you may wish to avoid encryption so you can more easily debug your application by viewing the OAuth data being saved to the database. This TextEncryptor performs no encryption.

```
TextEncryptor textEncryptor = AndroidEncryptors.noOpText();
connectionRepository = new SQLiteConnectionRepository(repositoryHelper,
 connectionFactoryRegistry, textEncryptor);
```

Establishing an OAuth 1.0a connection

The following steps illustrate how to establish a connection to Twitter. A working example is provided in the sample application described earlier.

The first step is to retrieve the connection factory from the registry that we created earlier.

```
TwitterConnectionFactory connectionFactory;
connectionFactory = (TwitterConnectionFactory) connectionFactoryRegistry.getConnectionFactory(TwitterApi.class)
```

Fetch a one time use request token. You must save this request token, because it will be needed in a later step.

```
OAuth1Operations oauth = connectionFactory.getOAuthOperations();

// The callback url is used to respond to your application with an OAuth verifier
String callbackUrl = "x-org-springsource-android-showcase://twitter-oauth-response";

// Fetch a one time use Request Token from Twitter
OAuthToken requestToken = oauth.fetchRequestToken(callbackUrl, null);
```

Generate the url for authorizing against Twitter. Once you have the url, you use it in a WebView so the user can login and authorize your application. One method of doing this is provided in the sample application.

```
String authorizeUrl = oauth.buildAuthorizeUrl(requestToken.getValue(), OAuth1Parameters.NONE);
```

Once the user has successfully authenticated and authorized the application, Twitter will call back to your application with the oauth verifier. The following settings from an AndroidManifest illustrate how to associate a callback url with a specific Activity. In this case, when the request is made from Twitter to the callback url, the TwitterActivity will respond.

```
<activity android:name="org.springframework.android.showcase.social.twitter.TwitterActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="x-org-springsource-android-showcase" android:host="twitter-oauth-response" />
  </intent-filter>
</activity>
```

The Activity that responds to the callback url should retrieve the oauth_verifier querystring parameter from the request.

```
Uri uri = getIntent().getData();
String oauthVerifier = uri.getQueryParameter("oauth_verifier");
```

Once you have the oauth_verifier, you can authorize the request token that was saved earlier.

```
AuthorizedRequestToken authorizedRequestToken = new AuthorizedRequestToken(requestToken, verifier);
```

Now exchange the authorized request token for an access token. Once you have the access token, the request token is no longer required, and can be safely discarded.

```
OAuth1Operations oauth = connectionFactory.getOAuthOperations();
OAuthToken accessToken = oauth.exchangeForAccessToken(authorizedRequestToken, null);
```

Finally, we can create a Twitter connection and store it in the repository.

```
Connection<TwitterApi> connection = connectionFactory.createConnection(accessToken);
connectionRepository.addConnection(connection);
```

Establishing an OAuth 2.0 connection

The following steps illustrate how to establish a connection to Facebook. A working example is provided in the sample application described earlier. Keep in mind that each provider's implementation may be different. You may have to adjust these steps when connecting to a different OAuth 2.0 provider.

The first step is to retrieve the connection factory from the registry that we created earlier.

```
FacebookConnectionFactory connectionFactory;
connectionFactory = (FacebookConnectionFactory) connectionFactoryRegistry.getConnectionFactory(FacebookApi.clas
```

Specify the redirect url. In the case of Facebook, we are using the client-side authorization flow. In order to retrieve the access token, Facebook will redirect to a success page that contains the access token in a URI fragment.

```
String redirectUri = "https://www.facebook.com/connect/login_success.html";
```

Define the scope of permissions your app requires.

```
String scope = "publish_stream,offline_access,read_stream,user_about_me";
```

In order to display a mobile formatted web page for Facebook authorization, you must pass an additional parameter in the request. This parameter is not part of the OAuth specification, but the following illustrates how Spring Social supports additional parameters.

```
MultiValueMap<String, String> additionalParameters = new LinkedMultiValueMap<String, String>();
additionalParameters.add("display", "touch");
```

Now we can generate the Facebook authorization url to be used in the browser or web view

```
OAuth2Parameters parameters = new OAuth2Parameters(redirectUri, scope, null, additionalParameters);
OAuth2Operations oauth = connectionFactory.getOAuthOperations();
String authorizeUrl = oauth.buildAuthorizeUrl(GrantType.IMPLICIT_GRANT, parameters);
```

The next step is to load the generated authorization url into a webview within your application. After the user logs in and authorizes your application, the browser will redirect to the url specified earlier. If authentication was successful, the url of the redirected page will now include a URI fragment which contains an access_token

parameter. Retrieve the access token from the URI fragment and use it to create the Facebook connection. One method of doing this is provided in the sample application.

```
AccessGrant accessGrant = new AccessGrant(accessToken);
Connection<FacebookApi> connection = connectionFactory.createConnection(accessGrant);
connectionRepository.addConnection(connection);
```

4. Spring Android Core Module

4.1 Introduction

The spring-android-core module provides common functionality to the other Spring Android modules. It includes a subset of the functionality available in Spring Framework Core.

4.2 How to get

Add the spring-android-core artifact to your classpath:

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

5. Spring Android and Maven

5.1 Introduction

An alternative to downloading the individual library JARs yourself is to use Maven for dependency management. The Maven Android Plugin [http://code.google.com/p/maven-android-plugin] allows developers to utilize Maven's dependency management capabilities within an Android application. Additionally, the Maven Integration for Android Development Tools [http://code.google.com/a/eclipselabs.org/p/m2eclipse-android-integration/] bridges the Maven Android Plugin and the Android Development Tools (ADT) [http://developer.android.com/sdk/eclipse-adt.html] to allow the use of dependency management within Eclipse.

5.2 Example POM

The following Maven POM file [http://maven.apache.org/guides/introduction/introduction-to-the-pom.html] from the Spring Android Showcase sample application, illustrates how to configure the Maven Android Plugin [http://code.google.com/p/maven-android-plugin] and associated dependencies for use with Spring Android and Spring Social.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.springframework.android</groupId>
    <artifactId>spring-android-showcase-client</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
    <packaging>apk</packaging>
    <name>spring-android-showcase-client</name>
    <url>http://www.springsource.org</url>
    <organization>
        <name>SpringSource</name>
        <url>http://www.springsource.org</url>
    </organization>

    <properties>
        <android-platform>10</android-platform>
        <android-emulator>10</android-emulator>
        <maven-android-plugin-version>2.8.4</maven-android-plugin-version>
        <maven-compiler-plugin-version>2.3.2</maven-compiler-plugin-version>
        <maven-eclipse-plugin-version>2.8</maven-eclipse-plugin-version>
        <android-version>2.3.3</android-version>
        <!-- Available Android versions: 1.5_r3, 1.5_r4, 1.6_r2, 2.1.2, 2.1_r1, 2.2.1, 2.3.1, 2.3.3 -->
        <spring-android-version>1.0.0.M3</spring-android-version>
        <spring-social-version>1.0.0.M3</spring-social-version>
        <jackson-version>1.8.0</jackson-version>
        <simple-version>2.4.1</simple-version>
        <android-rome-version>1.0.0-r2</android-rome-version>
    </properties>

    <build>
        <sourceDirectory>src</sourceDirectory>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>com.jayway.maven.plugins.android.generation2</groupId>
```

```
                <artifactId>maven-android-plugin</artifactId>
                <version>${maven-android-plugin-version}</version>
                <configuration>
                    <sdk>
                        <platform>${android-platform}</platform>
                    </sdk>
                    <emulator>
                        <avd>${android-emulator}</avd>
                    </emulator>
                    <deleteConflictingFiles>true</deleteConflictingFiles>
                    <undeployBeforeDeploy>true</undeployBeforeDeploy>
                </configuration>
                <extensions>true</extensions>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven-compiler-plugin-version}</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-eclipse-plugin</artifactId>
                <version>${maven-eclipse-plugin-version}</version>
                <configuration>
                    <downloadSources>true</downloadSources>
                    <downloadJavadocs>true</downloadJavadocs>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>com.google.android</groupId>
            <artifactId>android</artifactId>
            <version>${android-version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.android</groupId>
            <artifactId>spring-android-rest-template</artifactId>
            <version>${spring-android-version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.android</groupId>
            <artifactId>spring-android-auth</artifactId>
            <version>${spring-android-version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.social</groupId>
            <artifactId>spring-social-twitter</artifactId>
            <version>${spring-social-version}</version>
            <exclusions>
                <exclusion>
                    <!-- Exclude in favor of Spring Android RestTemplate -->
                    <artifactId>spring-social-web</artifactId>
                    <groupId>org.springframework.social</groupId>
                </exclusion>
                <exclusion>
                    <!-- Provided by Android -->
                    <artifactId>commons-logging</artifactId>
                    <groupId>commons-logging</groupId>
                </exclusion>
```

```
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.social</groupId>
            <artifactId>spring-social-facebook</artifactId>
            <version>${spring-social-version}</version>
            <exclusions>
                <exclusion>
                    <!-- Exclude in favor of Spring Android RestTemplate -->
                    <artifactId>spring-social-web</artifactId>
                    <groupId>org.springframework.social</groupId>
                </exclusion>
                <exclusion>
                    <!-- Provided by Android -->
                    <artifactId>commons-logging</artifactId>
                    <groupId>commons-logging</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>spring-webmvc</artifactId>
                    <groupId>org.springframework</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <!-- Using Jackson for JSON marshaling -->
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-mapper-asl</artifactId>
            <version>${jackson-version}</version>
        </dependency>
        <dependency>
            <!-- Using Simple for XML marshaling -->
            <groupId>org.simpleframework</groupId>
            <artifactId>simple-xml</artifactId>
            <version>${simple-version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>stax</artifactId>
                    <groupId>stax</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>stax-api</artifactId>
                    <groupId>stax</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <!-- Using ROME for RSS and ATOM feeds -->
            <groupId>com.google.code.android-rome-feed-reader</groupId>
            <artifactId>android-rome-feed-reader</artifactId>
            <version>${android-rome-version}</version>
        </dependency>
    </dependencies>

    <repositories>
        <!-- For developing with Android ROME Feed Reader -->
        <repository>
            <id>android-rome-feed-reader-repository</id>
            <name>Android ROME Feed Reader Repository</name>
            <url>https://android-rome-feed-reader.googlecode.com/svn/maven2/releases</url>
        </repository>
        <!-- For testing against latest Spring snapshots -->
        <repository>
```

```
            <id>org.springframework.maven.snapshot</id>
            <name>Spring Maven Snapshot Repository</name>
            <url>http://maven.springframework.org/snapshot</url>
            <releases><enabled>false</enabled></releases>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <!-- For developing against latest Spring milestones -->
        <repository>
            <id>org.springframework.maven.milestone</id>
            <name>Spring Maven Milestone Repository</name>
            <url>http://maven.springframework.org/milestone</url>
            <snapshots><enabled>false</enabled></snapshots>
        </repository>
    </repositories>

</project>
```

5.3 Maven Commands

Once you have configured a Maven POM in your Android project you can use the following Maven command to clean and assemble your Android APK file. Additional goals [http://maven-android-plugin-m2site.googlecode.com/svn/plugin-info.html] are available for use with the Maven Android Plugin.

```
$ mvn clean install
```

The following command starts the emulator specified in the Maven Android Plugin section of the POM file

```
$ mvn android:emulator-start
```

Deploys the application package to the emulator

```
$ mvn android:deploy
```