Spring Android Reference Manual

Roy Clarkson

Spring Android Reference Manual

by Roy Clarkson

1.0.0.M4

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 RestTemplate Module

2.1 Introduction

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

2.2 Overview

The `RestTemplate` class is the heart of the Spring Android RestTemplate library. It is conceptually similar to other template classes found in other Spring portfolio projects. RestTemplate's behavior is customized by providing callback methods and configuring the `HttpMessageConverter` used to marshal objects into the HTTP request body and to unmarshal any response back into an object. When you create a new `RestTemplate` instance, the constructor sets up several supporting objects that make up the RestTemplate functionality.

Here is an overview of the functionality supported within `RestTemplate`.

HTTP Client

`RestTemplate` provides an abstraction for making RESTful http requests, and internally, `RestTemplate` utilizes a native Android HTTP client library for those requests. The HttpComponents HttpClient [http://hc.apache.org/httpcomponents-client-ga/index.html] is a native HTTP client available on the Android platform. Within Spring Android RestTemplate the HttpClient is made available through the `HttpComponentsClientHttpRequestFactory`. This class is set as the default `ClientHttpRequestFactory` when you create a new `RestTemplate` instance. The standard J2SE facilities are also available as a native Android library, and are made available through the `SimpleClientHttpRequestFactory`. To utilize the `SimpleClientHttpRequestFactory`, you must either pass a new instance into the `RestTemplate` constructor, or call `setRequestFactory(ClientHttpRequestFactory requestFactory)` on an existing `RestTemplate` instance.

Gzip Compression

`RestTemplate` supports sending and receiving data encoded with gzip compression. The HTTP specification allows for additional values in the `Accept-Encoding` header field, however `RestTemplate` only supports gzip compression at this time.

Object to JSON Marshaling

Object to JSON marshaling in Spring Android RestTemplate requires the use of a third party JSON mapping library. There are two libraries supported in Spring Android, Jackson JSON Processor [http://jackson.codehaus.org], and Google Gson [http://code.google.com/p/google-gson/]. While Jackson is a well known JSON parsing library, the Gson library is smaller, which would result in an smaller Android app when packaged.

Object to XML Marshaling

Object to XML marshaling in Spring Android RestTemplate requires the use of a third party XML mapping library. The Simple XML serializer [http://simple.sourceforge.net] is used to provide this marshaling functionality.

RSS and Atom Support

RSS and Atom feed support in Spring Android RestTemplate 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 to provide this functionality.

2.3 RestTemplate Methods

`RestTemplate` provides higher level methods that correspond to each of the six main HTTP methods. These methods make it easy to invoke many RESTful services and enforce REST best practices.

The names of `RestTemplate` methods follow a naming convention, the first part indicates what HTTP method is being invoked and the second part indicates what is returned. For example, the method `getForObject()` will perform a GET, convert the HTTP response into an object type of your choice and return that object. The method `postForLocation()` will do a POST, converting the given object into a HTTP request and return the response HTTP Location header where the newly created object can be found. In case of an exception processing the HTTP request, an exception of the type `RestClientException` will be thrown. This behavior can be changed by plugging in another `ResponseErrorHandler` implementation into the `RestTemplate.`

For more information on `RestTemplate` and it's associated methods, please refer to the API Javadoc [http://static.springsource.org/spring-android/docs/1.0.x/api/org/springframework/web/client/RestTemplate.html]

HTTP DELETE

```
public void delete(String url, Object... urlVariables) throws RestClientException;

public void delete(String url, Map<String, ?> urlVariables) throws RestClientException;

public void delete(URI url) throws RestClientException;
```

HTTP GET

```
public <T> T getForObject(String url, Class<T> responseType, Object... urlVariables) throws RestClientExceptior

public <T> T getForObject(String url, Class<T> responseType, Map<String, ?> urlVariables) throws RestClientExce

public <T> T getForObject(URI url, Class<T> responseType) throws RestClientException;

public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object... urlVariables);
```

```
public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Map<String, ?> urlVariables);

public <T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType) throws RestClientException;
```

HTTP HEAD

```
public HttpHeaders headForHeaders(String url, Object... urlVariables) throws RestClientException;

public HttpHeaders headForHeaders(String url, Map<String, ?> urlVariables) throws RestClientException;

public HttpHeaders headForHeaders(URI url) throws RestClientException;
```

HTTP OPTIONS

```
public Set<HttpMethod> optionsForAllow(String url, Object... urlVariables) throws RestClientException;

public Set<HttpMethod> optionsForAllow(String url, Map<String, ?> urlVariables) throws RestClientException;

public Set<HttpMethod> optionsForAllow(URI url) throws RestClientException;
```

HTTP POST

```
public URI postForLocation(String url, Object request, Object... urlVariables) throws RestClientException;

public URI postForLocation(String url, Object request, Map<String, ?> urlVariables);

public URI postForLocation(URI url, Object request) throws RestClientException;

public <T> T postForObject(String url, Object request, Class<T> responseType, Object... uriVariables);

public <T> T postForObject(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables);

public <T> T postForObject(URI url, Object request, Class<T> responseType) throws RestClientException;

public <T> ResponseEntity<T> postForEntity(String url, Object request, Class<T> responseType, Object... uriVari

public <T> ResponseEntity<T> postForEntity(String url, Object request, Class<T> responseType, Map<String, ?> ur

public <T> ResponseEntity<T> postForEntity(URI url, Object request, Class<T> responseType) throws RestClientExc
```

HTTP PUT

```
public void put(String url, Object request, Object... urlVariables) throws RestClientException;

public void put(String url, Object request, Map<String, ?> urlVariables) throws RestClientException;

public void put(String url, Object request, Map<String, ?> urlVariables) throws RestClientException;
```

2.4 HTTP Message Conversion

Objects passed to and returned from the methods `getForObject()`, `getForEntity()`, `postForLocation()`, `postForObject()` and `put()` are converted to HTTP requests and from HTTP responses by `HttpMessageConverter` instances. Converters for the main mime types are registered by default, but you can also write your own converter and register it via the `messageConverters()` property.

The default converter instances registered with the template are `ByteArrayHttpMessageConverter`, `StringHttpMessageConverter`, and `ResourceHttpMessageConverter`. If you are app is running on Android 2.2 or later, then `XmlAwareFormHttpMessageConverter` and `SourceHttpMessageConverter` are registered, as these two message converters require the `javax.xml.transform` library. On Android 2.1, this falls back to the `FormHttpMessageConverter` which lacks some of the XML support in the other two.

The `HttpMessageConverter` interface is shown below to give you a better feel for its functionality.

```
public interface HttpMessageConverter<T> {

    // Indicates whether the given class can be read by this converter.
    boolean canRead(Class<?> clazz, MediaType mediaType);

    // Indicates whether the given class can be written by this converter.
    boolean canWrite(Class<?> clazz, MediaType mediaType);

    // Return the list of {@link MediaType} objects supported by this converter.
    List<MediaType> getSupportedMediaTypes();

    // Read an object of the given type form the given input message, and returns it.
    T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
            throws IOException, HttpMessageNotReadableException;

    // Write an given object to the given output message.
    void write(T t, MediaType contentType, HttpOutputMessage outputMessage)
            throws IOException, HttpMessageNotWritableException;

}
```

Concrete implementations for the main media (mime) types are provided in the framework and are registered by default within `RestTemplate`

The following `HttpMessageConverter` implementations are available in Spring Android. For all converters a default media type is used but can be overridden by calling the `setSupportedMediaTypes()` method.

StringHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write Strings from the HTTP request and response. By default, this converter supports all text media types (`text/*`), and writes with a `Content-Type` of `text/plain`.

FormHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write form data from the HTTP request and response. By default, this converter reads and writes the media type `application/x-www-form-urlencoded`. Form data is read from and written into a `MultiValueMap<String, String>`.

ByteArrayMessageConverter

An `HttpMessageConverter` implementation that can read and write byte arrays from the HTTP request and response. By default, this converter supports all media types (`*/*`), and writes with a `Content-Type` of `application/octet-stream`. This can be overridden by setting the supportedMediaTypes property, and overriding `getContentType(byte[])`.

SimpleXmlHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write XML from the HTTP request and response using Simple Framework [http://simple.sourceforge.net]'s `Serializer`. XML mapping can be customized as needed through the use of Simple's provided annotations. When additional control is needed, a custom `Serializer` can be injected through the `Serializer` property. By default, this converter reads and writes the media types `application/xml`, `text/xml`, and `application/*+xml`.

It is important to note that this is not a Spring OXM compatible message converter. It is a standalone implementation that enables XML serialization through Spring Android.

MappingJacksonHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write JSON using Jackson JSON Processor [http://jackson.codehaus.org]'s `ObjectMapper`. JSON mapping can be customized as needed through the use of Jackson's provided annotations. When further control is needed, a custom `ObjectMapper` can be injected through the `ObjectMapper` property for cases where custom JSON serializers/deserializers need to be provided for specific types. By default this converter supports (`application/json`).

GsonHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write JSON using Google Gson [http://code.google.com/p/google-gson/]'s `Gson` class. JSON mapping can be customized as needed through the use of Gson's provided annotations. When further control is needed, a custom `Gson` can be injected through the `Gson` property for cases where custom JSON serializers/deserializers need to be provided for specific types. By default this converter supports (`application/json`).

Please note that this message converter and the `MappingJacksonHttpMessageConverter` support `application/json`. Because of this, only one will automatically be loaded with a new `RestTemplate` instance. If you include Jackson and Gson in your classpath, Jackson will take precedence over Gson.

SourceHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write `javax.xml.transform.Source` from the HTTP request and response. Only `DOMSource`,

`SAXSource`, and `StreamSource` are supported. By default, this converter supports (`text/xml`) and (`application/xml`).

SyndFeedHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write RSS and Atom feeds from the HTTP request and response using Android ROME Feed Reader [http://code.google.com/p/android-rome-feed-reader]. The data is read from and written into a `com.google.code.rome.android.repackaged.com.sun.syndication.feed.synd.SyndFeed`. By default, this converter supports (`application/rss+xml`) and (`application/atom+xml`).

RssChannelHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write RSS feeds from the HTTP request and response. The data is read from and written into a `com.google.code.rome.android.repackaged.com.sun.syndication.feed.rss.Channel`. By default, this converter supports (`application/rss+xml`).

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

AtomFeedHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write Atom feeds from the HTTP request and response. The data is read from and written into a `com.google.code.rome.android.repackaged.com.sun.syndication.feed.atom.Feed`. By default, this converter supports (`application/atom+xml`).

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

2.5 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>

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

```
```

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 RestTemplate supports several optional libraries. These optional libraries are used by different `HttpMessageConverter` instances within `RestTemplate`. 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

Include the following Jackson [http://jackson.codehaus.org] dependencies to your classpath to enable the `MappingJacksonHttpMessageConverter`.

```
<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>
```

Google Gson

Add the following Google Gson [http://code.google.com/p/google-gson/] dependency to your classpath to enable the `GsonHttpMessageConverter`.

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>${gson-version}</version>
</dependency>
```

Simple XML Serializer

Add the following Simple XML Serializer [http://simple.sourceforge.net] dependency to your classpath to enable the `SimpleXmlHttpMessageConverter`.

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

Android ROME Feed Reader

Add the following Android ROME Feed Reader [http://code.google.com/p/android-rome-feed-reader] dependencies to your classpath to enable the `RssChannelHttpMessageConverter`, `AtomFeedHttpMessageConverter`, and `SyndFeedHttpMessageConverter`. 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>
```

The Android ROME Feed Reader is not available through the Maven central repository. When using Maven, you will need to include the following repository in your POM.

```
<!-- 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>
```

2.6 Usage Examples

Using `RestTemplate`, 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 [https://github.com/SpringSource/spring-android-samples]. You can retrieve the source code for the sample app with the following command:

```
$ git clone git://github.com/SpringSource/spring-android-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");
```

Using Gzip Compression

Gzip compression can significantly reduce the size of the response data being returned in a REST request. Gzip must be supported by the web server to which the request is being made. By setting the content coding type of the `Accept-Encoding` header to `gzip`, you are requesting that the server respond using gzip compression. If gzip is available, or enabled on the server, then it should return a compressed response. RestTemplate checks the `Content-Encoding` header in the response to determine if, in fact, the response is gzip compressed. At this time, RestTemplate only supports the gzip content coding type in the `Content-Encoding` header. If the response data is determined to be gzip compressed, then a GZIPInputStream [http://developer.android.com/reference/java/util/zip/GZIPInputStream.html] is used to decompress it.

The following example shows how to request a gzip compressed response from the server.

```
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAcceptEncoding(Collections.singletonList(ContentCodingType.GZIP));

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

RestTemplate restTemplate = new RestTemplate();
ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.GET, requestEntity, String.class);
```

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 REST 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:

```
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAccept(Collections.singletonList(new MediaType("application","json")));

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();
```

Alternatively, you can use the `GsonHttpMessageConverter` for JSON marshaling. The following repeats the same request, utilizing Gson.

```
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAccept(Collections.singletonList(new MediaType("application","json")));

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

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

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

RestTemplate restTemplate = new RestTemplate();
restTemplate.setMessageConverters(messageConverters);

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 REST 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 REST 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 [http://twitter.com], 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 [http://oauth.net/]. 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 [http://oauth.net/] 1.x and 2.0 support in an Android application by utilizing Spring Social [http://www.springsource.org/spring-social]. It includes a SQLite [http://www.sqlite.org/] repository, and Android compatible Spring Security [http://static.springsource.org/spring-security/site/index.html] encryption. The Spring Social project enables your applications to establish Connections with Software-as-a-Service (SaaS) Providers such as Facebook [http://facebook.com] and Twitter [http://twitter.com] 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` [http://static.springsource.org/spring-android/docs/1.0.x/api/org/springframework/social/connect/sqlite/SQLiteConnectionRepository.html] class implements the `ConnectionRepository` [http://static.springsource.org/spring-social/docs/1.0.x/api/org/springframework/social/connect/ConnectionRepository.html] interface from Spring Social. It is used to persist the connection information to a SQLite [http://www.sqlite.org/] 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` [http://static.springsource.org/spring-android/docs/1.0.x/api/index.html?org/springframework/social/connect/sqlite/SQLiteConnectionRepository.html] 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` [http://static.springsource.org/spring-android/docs/1.0.x/api/org/springframework/security/crypto/encrypt/AndroidEncryptors.html] has been provided in Spring Android. This class uses an Android compatible `SecureRandom` [http://developer.android.com/reference/java/security/SecureRandom.html] provider for generating byte array based keys using the SHA1PRNG algorithm.

3.3 How to get

Add the spring-android-auth and the supporting artifacts to your classpath. Maven will handle the dependency management, but the required dependencies are listed here for clarity.

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

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
    <version>3.1.0.RC2.crypto</version>
    <exclusions>
        <!-- Exclude in favor of Spring Android Core -->
        <exclusion>
            <artifactId>spring-core</artifactId>
            <groupId>org.springframework</groupId>
        </exclusion>
    </exclusions>
</dependency>

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

```
</dependency>

<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>
```

To use the Spring Social Twitter provider, you can add it to your classpath. Note the exclusions in this dependency. Commons Logging is built into Android, and many of the Spring Social provider libraries are built with support for Spring Web, which is not needed on Android.

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-twitter</artifactId>
    <version>${spring-social-version}</version>
    <exclusions>
        <exclusion>
            <!-- Provided by Android -->
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

Similarly, you can use the Spring Social Facebook provider by adding it to your classpath. Again note the exclusions.

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-facebook</artifactId>
    <version>${spring-social-version}</version>
    <exclusions>
        <!-- Provided by Android -->
        <exclusion>
            <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 [https://github.com/SpringSource/spring-android-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://github.com/SpringSource/spring-android-samples.git
```

Initializing the SQLite Database

`SQLiteConnectionRepositoryHelper` [http://static.springsource.org/spring-android/docs/1.0.x/api/org/springframework/social/connect/sqlite/support/SQLiteConnectionRepositoryHelper.html] 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` [http://static.springsource.org/spring-social/docs/1.0.x/api/org/springframework/social/connect/support/ConnectionFactoryRegistry.html] stores the different Spring Social connections to be used in the application.

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

You can create a `FacebookConnectionFactory` [http://static.springsource.org/spring-social-facebook/docs/1.0.x/api/org/springframework/social/facebook/connect/FacebookConnectionFactory.html], 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` [http://static.springsource.org/spring-social-twitter/docs/1.0.x/api/org/springframework/social/twitter/connect/TwitterConnectionFactory.html]. 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(Twitter.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(Facebook.class);
```

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 [http://developers.facebook.com/docs/guides/mobile/#web] 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>7</android-platform>
        <android-emulator>7</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>2.1_r1</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 -->
        <java-version>1.6</java-version>
        <spring-android-version>1.0.0.M4</spring-android-version>
        <spring-social-version>1.0.0.RC1</spring-social-version>
        <jackson-version>1.8.3</jackson-version>
        <gson-version>1.7.1</gson-version>
        <simple-version>2.6</simple-version>
    </properties>

    <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.security</groupId>
            <artifactId>spring-security-crypto</artifactId>
            <version>3.1.0.RC2.crypto</version>
            <exclusions>
                <!-- Exclude in favor of Spring Android Core -->
                <exclusion>
                    <artifactId>spring-core</artifactId>
                    <groupId>org.springframework</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.social</groupId>
            <artifactId>spring-social-core</artifactId>
            <version>${spring-social-version}</version>
            <exclusions>
                <!-- Exclude in favor of Spring Android RestTemplate -->
                <exclusion>
                    <artifactId>spring-web</artifactId>
                    <groupId>org.springframework</groupId>
                </exclusion>
                <!-- Provided by Android -->
                <exclusion>
                    <artifactId>commons-logging</artifactId>
                    <groupId>commons-logging</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.social</groupId>
            <artifactId>spring-social-twitter</artifactId>
            <version>${spring-social-version}</version>
            <exclusions>
                <!-- Provided by Android -->
                <exclusion>
                    <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>
                <!-- Provided by Android -->
                <exclusion>
                    <artifactId>commons-logging</artifactId>
                    <groupId>commons-logging</groupId>
```

```
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <!-- Using Jackson for JSON marshaling -->
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-mapper-asl</artifactId>
            <version>${jackson-version}</version>
        </dependency>
        <dependency>
            <!-- Using Gson for JSON marshaling -->
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>${gson-version}</version>
        </dependency>
        <dependency>
            <!-- Using Simple for XML marshaling -->
            <groupId>org.simpleframework</groupId>
            <artifactId>simple-xml</artifactId>
            <version>${simple-version}</version>
            <exclusions>
                <!-- StAX is not available on Android -->
                <exclusion>
                    <artifactId>stax</artifactId>
                    <groupId>stax</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>stax-api</artifactId>
                    <groupId>stax</groupId>
                </exclusion>
                <!-- Provided by Android -->
                <exclusion>
                    <artifactId>xpp3</artifactId>
                    <groupId>xpp3</groupId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

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

```
                <configuration>
                    <source>${java-version}</source>
                    <target>${java-version}</target>
                </configuration>
            </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>

    <repositories>
        <!-- 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
```