Spring for Android Reference Documentation

2.0.0.M2

Roy Clarkson

Table of Contents

1. Spring for Android Overview 1
1.1. Introduction 1
2. RestTemplate Module 2
2.1. Introduction 2
2.2. Overview 2
HTTP Client 2
Gzip Compression 3
Object to JSON Marshaling 3
Object to XML Marshaling 3
2.3. How to get 3
Gradle 3
Maven 3
Ant and Eclipse 4
2.4. RestTemplate Constructors 4
2.5. RestTemplate Methods 4
HTTP DELETE 5
HTTP GET 5
HTTP HEAD 5
HTTP OPTIONS 5
HTTP POST 6
HTTP PUT 6
2.6. HTTP Message Conversion 6
Default Message Converters 7
Available Message Converters 7
ByteArrayHttpMessageConverter 7
FormHttpMessageConverter 7
AllEncompassingFormHttpMessageConverter 7
ResourceHttpMessageConverter 8
SourceHttpMessageConverter 8
StringHttpMessageConverter 8
SimpleXmlHttpMessageConverter 8
MappingJackson2HttpMessageConverter 8
GsonHttpMessageConverter 9
2.7. Usage Examples 10
Basic Usage Example 10
Using Gzip Compression 10
Retrieving JSON data via HTTP GET 11
Retrieving XML data via HTTP GET 12
Send JSON data via HTTP POST 13
HTTP Basic Authentication 14
3. Auth Module 16
3.1. Introduction 16
3.2. Overview 16
SQLite Connection Repository 16
Encryption 16
3.3. How to get 16
Standard Installation 17

Maven Dependencies 17
3.4. Usage Examples 19
Initializing the SQLite Database 19
Single User App Environment 19
Encrypting OAuth Data 21
Establishing an OAuth 1.0a connection 21
Establishing an OAuth 2.0 connection 22
4. Core Module 24
4.1. Introduction 24
4.2. How to get 24
5. Maven Dependency Management 25
5.1. Introduction 25
5.2. Spring Repository 25
5.3. Example build.gradle 26
5.4. Example POM 26
5.5. Maven Commands 28

1. Spring for Android Overview

1.1 Introduction

The Spring for 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 for 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. RestTemplate Module

2.1 Introduction

Spring's RestTemplate is a robust, popular Java-based REST client. The Spring for 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 for 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. There are two native HTTP clients available on Android, the standard J2SE facilities, and the *HttpComponents HttpClient*. The standard JS2SE facilities are made available through `SimpleClientHttpRequestFactory`, while the HttpClient can be utilized via `HttpComponentsAndroidClientHttpRequestFactory`. Support for native HttpClient 4.0 is deprecated in favor of the Android port of HttpClient 4.3. The default `ClientHttpRequestFactory` used when you create a new `RestTemplate` instance differs based on the version of Android on which your application is running.

Google recommends to use the J2SE facilities on *Android 2.3 (Gingerbread)* and newer, while previous versions should use the HttpComponents HttpClient. Based on this recommendation `RestTemplate` checks the version of Android on which your app is running and uses the appropriate `ClientHttpRequestFactory`. To utilize a specific `ClientHttpRequestFactory` you must either pass a new instance into the `RestTemplate` constructor, or call `setRequestFactory(ClientHttpRequestFactory requestFactory)` on an existing `RestTemplate` instance.

Spring for Android also includes support for third-party HTTP client libraries. HttpClient 4.3 for Android is compatible with all versions of Android, and may be utilized as an alternative to the native clients by simply including the dependency in your project. If Spring for Android detects HttpClient 4.3, then it will automatically configure it as the default `ClientHttpRequestFactory`. HttpClient 4.3 has a considerable number of bug fixes and improvements over the native HttpClient 4.0 included within Android.

```
dependencies {
    compile('org.apache.httpcomponents:httpclient-android:$httpClientVersion')
}
```

An additional `ClientHttpRequestFactory` based on *OkHttp* is available as an alternative to the two native clients. `RestTemplate` can be configured to use `OkHttpRequestFactory` through the

`RestTemplate` constructor or by setting the `requestFactory` property. It is supported on Android 2.3 (Gingerbread) and newer, however in order to use it, you must include the OkHttp dependency in your project.

```
dependencies {
    compile('com.squareup.okhttp:okhttp-urlconnection:$okHttpVersion')
}
```

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 for Android RestTemplate requires the use of a third party JSON mapping library. There are two libraries supported in Spring for Android, *Jackson 2.x* and *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 for Android RestTemplate requires the use of a third party XML mapping library. The *Simple XML serializer* is used to provide this marshaling functionality.

2.3 How to get

There are a few methods for including external jars in your Android app. You can use Gradle or Maven for dependency management, or manually download them and include them in your app's `libs/` folder.

Gradle

Android Studio and the New Build System for Android offer a Gradle plugin for building Android apps. Gradle provides built in dependency management, which can be used to include the Spring for Android dependencies in your project.

Add the spring-android-rest-template dependency to your `build.gradle` file:

```
dependencies {
    compile('org.springframework.android:spring-android-rest-template:${version}')
}
```

Maven

Maven can be used to manage dependencies and build your Android app. See the Spring for Android and Maven section for more information. Additional dependencies may be required, depending on which HTTP Message Converters you are using within RestTemplate. See the Message Converters section for more information.

Add the spring-android-rest-template dependency to your `pom.xml` file:

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

Ant and Eclipse

In order to use RestTemplate in your Android application, you must include the following Spring for Android jars in the `libs/` folder.

- `spring-android-rest-template-{version}.jar`
- `spring-android-core-{version}.jar`

If you are building your project with Ant, Ant will automatically include any jars located in the `libs/` folder located in the root of your project. However, in Eclipse you must manually add the jars to the Build Path. Follow these steps to add the jars to your existing Android project in Eclipse.

1. Refresh the project in Eclipse so the `libs/` folder and jars display in the Package Explorer.
2. Right-Click (Command-Click) the first jar.
3. Select the `BuildPath` submenu.
4. Select `Add to Build Path` from the context menu.
5. Repeat these steps for each jar.

2.4 RestTemplate Constructors

The `RestTemplate` constructors are listed below. The default constructor includes a standard set of message body converters. For a list of default converters, see the HTTP Message Conversion section.

```
RestTemplate();

RestTemplate(ClientHttpRequestFactory requestFactory);

RestTemplate(List<HttpMessageConverter<?>> messageConverters);
```

If you would like to specify an alternate `ClientHttpRequestFactory`, such as `OkHttpClientHttpRequestFactory`, then you can do so by passing it in to the `requestFactory` parameter.

```
OkHttpClientHttpRequestFactory requestFactory = new OkHttpClientHttpRequestFactory();
RestTemplate template = new RestTemplate(ClientHttpRequestFactory requestFactory);
```

2.5 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 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
 RestClientException;

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

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...
 uriVariables);

public <T> ResponseEntity<T> postForEntity(String url, Object request, Class<T> responseType,
 Map<String, ?> uriVariables) throws RestClientException;

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

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;
```

2.6 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. 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.

Default Message Converters

The default `RestTemplate` constructor registers a standard set of message converters for the main mime types. 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`, `ResourceHttpMessageConverter`, `SourceHttpMessageConverter` and `AllEncompassingFormHttpMessageConverter`. See the following table for more information.

Table 2.1. Default Message Converters

Message Body Converter	Inclusion Rule
`ByteArrayHttpMessageConverter`	Always included
`StringHttpMessageConverter`	
`ResourceHttpMessageConverter`	
`SourceHttpMessageConverter`	
`AllEncompassingFormHttpMessageConverter`	
`SimpleXmlHttpMessageConverter`	Included if the Simple XML serializer is present.
`MappingJackson2HttpMessageConverter`	Included if the Jackson 2.x JSON processor is present.
`GsonHttpMessageConverter`	Included if Gson is present. Jackson 2.x takes precedence over Gson if both are available on the classpath.

Available Message Converters

The following `HttpMessageConverter` implementations are available in Spring for Android. For all converters a default media type is used but can be overridden through the `supportedMediaTypes` property.

ByteArrayHttpMessageConverter

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[])`.

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>`.

AllEncompassingFormHttpMessageConverter

Extension of `FormHttpMessageConverter`, adding support for XML and JSON-based parts.

ResourceHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write `Resource` Resources. By default, this converter can read all media types. Written resources use `application/octet-stream` for the `Content-Type`.

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`.

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`.

SimpleXmlHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write XML from the HTTP request and response using *Simple Framework*'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 for Android.

Add the following dependency to your classpath to enable the `SimpleXmlHttpMessageConverter`.

Gradle:

```
dependencies {
    compile('org.simpleframework:simple-xml:${version}')
}
```

Maven:

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

MappingJackson2HttpMessageConverter

An `HttpMessageConverter` implementation that can read and write JSON using *Jackson (2.x)*'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`.

Please note that this message converter and the `GsonHttpMessageConverter` both support `application/json` by default. Because of this, you should only add one JSON message converter to a `RestTemplate` instance. `RestTemplate` will use the first converter it finds that matches the specified mime type, so including both could produce unintended results.

Include the following dependencies in your classpath to enable the `MappingJackson2HttpMessageConverter`. Please note that if you are manually copying the jars into your project, you will also need to include the `jackson-annotations` and `jackson-core` jars.

Gradle:

```
dependencies {
    compile('com.fasterxml.jackson.core:jackson-databind:${version}')
}
```

Maven:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>${jackson-version}</version>
</dependency>
```

GsonHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write JSON using *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 `MappingJackson2HttpMessageConverter` both support `application/json` by default. Because of this, you should only add one JSON message converter to a `RestTemplate` instance. `RestTemplate` will use the first converter it finds that matches the specified mime type, so including both could produce unintended results.

Include the following dependency in your classpath to enable the `GsonHttpMessageConverter`.

Gradle:

```
dependencies {
    compile('com.google.code.gson:gson:${version}')
}
```

Maven:

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

2.7 Usage Examples

Using `RestTemplate`, it is 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*. You can retrieve the source code for the sample app with the following command:

```
$ git clone git://github.com/spring-projects/spring-android-samples.git
```

Basic Usage Example

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

```
String url = "https://ajax.googleapis.com/ajax/services/search/web?v=1.0&q={query}";

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response to a String
String result = restTemplate.getForObject(url, String.class, "Spring Framework");
```

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`* is used to decompress it.

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

```
// Add the gzip Accept-Encoding header
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAcceptEncoding(ContentCodingType.GZIP);
HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response to a String
ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.GET, requestEntity,
 String.class);
```

One thing to note, is that when using the J2SE facilities with the `SimpleClientHttpRequestFactory`, Gingerbread and newer automatically set the Accept-Encoding header to request gzip responses. This is built in functionality of newer versions of Android. If you desire to disable gzip, then you must set the `identity` value in the header.

```
// Add the identity Accept-Encoding header
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAcceptEncoding(ContentCodingType.IDENTITY);
HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response to a String
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. Marshaling JSON content requires Jackson or Gson to be available on the classpath.

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:

```
// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response from JSON to an array of Events
Event[] events = restTemplate.getForObject(url, Event[].class);
```

You can also set the `Accept` header for the request:

```
// Set the Accept header
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAccept(Collections.singletonList(new MediaType("application","json")));
HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response from JSON to an array of Events
ResponseEntity<Event[]> responseEntity = restTemplate.exchange(url, HttpMethod.GET, requestEntity,
 Event[].class);
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, you 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:

```
// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response from XML to an EventList object
EventList eventList = restTemplate.getForObject(url, EventList.class);
```

You can also specify the `Accept` header for the request:

```
// Set the Accept header
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAccept(Collections.singletonList(new MediaType("application","xml")));
HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP GET request, marshaling the response from XML to an EventList
ResponseEntity<EventList> responseEntity = restTemplate.exchange(url, HttpMethod.GET, requestEntity,
 EventList.class);
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:

```
// Create and populate a simple object to be used in the request
Message message = new Message();
message.setId(555);
message.setSubject("test subject");
message.setText("test text");

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP POST request, marshaling the request to JSON, and the response to a String
String response = restTemplate.postForObject(url, message, String.class);
```

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

```
// Create and populate a simple object to be used in the request
Message message = new Message();
message.setId(555);
message.setSubject("test subject");
message.setText("test text");

// Set the Content-Type header
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setContentType(new MediaType("application","json"));
HttpEntity<Message> requestEntity = new HttpEntity<Message>(message, requestHeaders);

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

// Make the HTTP POST request, marshaling the request to JSON, and the response to a String
ResponseEntity<String> responseEntity = restTemplate.exchange(url, HttpMethod.POST, requestEntity,
 String.class);
String result = responseEntity.getBody();
```

HTTP Basic Authentication

This example illustrates how to populate the *HTTP Basic Authentication* header with the username and password. If the username and password are accepted, then you will receive the response from the request. If they are not accepted, then the server is supposed to return an *HTTP 401 Unauthorized* response. Internally, RestTemplate handles the response, then throws an `HttpClientErrorException`. By calling `getStatusCode()` on this exception, you can determine the exact cause and handle it appropriately.

```
// Set the username and password for creating a Basic Auth request
HttpAuthentication authHeader = new HttpBasicAuthentication(username, password);
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.setAuthorization(authHeader);
HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);

// Create a new RestTemplate instance
RestTemplate restTemplate = new RestTemplate();

try {
    // Make the HTTP GET request to the Basic Auth protected URL
    ResponseEntity<Message> response = restTemplate.exchange(url, HttpMethod.GET, requestEntity,
 String.class);
    return response.getBody();
} catch (HttpClientErrorException e) {
    Log.e(TAG, e.getLocalizedMessage(), e);
    // Handle 401 Unauthorized response
}
```

3. 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 for Android. This class uses an Android compatible *`SecureRandom`* provider for generating byte array based keys using the SHA1PRNG algorithm.

3.3 How to get

There are a few methods for including external jars in your Android app. One is to manually download them and include them in your app's `libs/` folder. Another option is to use Maven for dependency management.

Standard Installation

In order to use RestTemplate in your Android application, you must include the following Spring jars in the `libs/` folder. These are available from the SpringSource *Community Downloads* page.

- `spring-android-auth-{version}.jar`
- `spring-android-rest-template-{version}.jar`
- `spring-android-core-{version}.jar`
- `spring-security-crypto-{version}.jar`
- `spring-social-core-{version}.jar`

Each *Spring Social* provider may have additional dependencies. For example, to use Spring Social Twitter, the following jars are required.

- `spring-social-twitter-{version}.jar`
- `spring-social-core-{version}.jar`
- `spring-security-crypto-{version}.jar`
- `jackson-mapper-asl-{version}.jar`
- `jackson-core-asl-{version}.jar`

If you are building your project with Ant, Ant will automatically include any jars located in the `libs/` folder located in the root of your project. However, in Eclipse you must manually add the jars to the Build Path. Follow these steps to add the jars to your existing Android project in Eclipse.

1. Refresh the project in Eclipse so the `libs/` folder and jars display in the Package Explorer.
2. Right-Click (Command-Click) the first jar.
3. Select the `BuildPath` submenu.
4. Select `Add to Build Path` from the context menu.
5. Repeat these steps for each jar.

Maven Dependencies

Add the spring-android-auth artifact to your classpath. See the Spring for Android and Maven section for more information.

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

The transitive dependencies are automatically imported by Maven, but they are listed here for clarity.

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

Both the Spring Social Twitter and Facebook libraries transitively depend on the *Jackson JSON processor*. Again, if you are not using Maven, you will need to include these in your `libs/` folder.

```
<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>
```

3.4 Usage Examples

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

The following examples are based on a *sample Android application*, 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/spring-projects/spring-android-samples.git
```

Initializing the SQLite Database

`SQLiteConnectionRepositoryHelper` extends *`SQLiteOpenHelper`*. Create a new instance by passing a *`context`* reference. Depending on your implementation, and to avoid *memory leaks*, 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(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* 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. Core Module

4.1 Introduction

The spring-android-core module provides common functionality to the other Spring for 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:

```
dependencies {
    compile 'org.springframework.android:spring-android-core:$version'
}
```

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

5. Maven Dependency Management

5.1 Introduction

Android Studio and the new Android build system support the use of Maven dependencies through Gradle. Alternatively, the *Android Maven Plugin* can also be used to build an Android application.

5.2 Spring Repository

The following repositories are available for all Spring projects. Much more information is available at the *Spring Repository FAQ*.

Release versions are available through *Maven Central* or via the *Spring Repository*:

```
repositories {
    maven { url "https://repo.spring.io/release" }
}
```

```
<repository>
    <id>spring-repo</id>
    <name>Spring Repository</name>
    <url>https://repo.spring.io/release</url>
</repository>
```

If you are developing against the latest milestone version, you will need to add the following repository in order to resolve the artifact:

```
repositories {
    maven { url "https://repo.spring.io/milestone" }
}
```

```
<repository>
    <id>spring-milestone</id>
    <name>Spring Milestone Repository</name>
    <url>https://repo.spring.io/milestone</url>
</repository>
```

If you are testing with the latest build snapshot, you will need to add the following repository:

```
repositories {
    maven { url "https://repo.spring.io/snapshot" }
}
```

```
<repository>
    <id>spring-snapshot</id>
    <name>Spring Snapshot Repository</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
</repository>
```

5.3 Example build.gradle

The following is an example build.gradle for use with Android Studio and the new Android build system.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 21
    buildToolsVersion '21.1.1'

    defaultConfig {
        applicationId 'org.springframework.demo'
        minSdkVersion 15
        targetSdkVersion 21
        versionCode 1
        versionName '1.0'
    }
    buildTypes {
        release {
            runProguard false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    packagingOptions {
        exclude 'META-INF/ASL2.0'
        exclude 'META-INF/LICENSE'
        exclude 'META-INF/license.txt'
        exclude 'META-INF/NOTICE'
        exclude 'META-INF/notice.txt'
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:support-v4:20.+'
    compile 'org.springframework.android:spring-android-rest-template:2.0.0.M2'
    compile 'org.apache.httpcomponents:httpclient-android:4.3.5'
    compile 'com.fasterxml.jackson.core:jackson-databind:2.4.4'
}
```

5.4 Example POM

The following *Maven POM file* illustrates how to configure the *Maven Android Plugin* and associated dependencies for use with Spring for Android.

```
<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-client</artifactId>
    <version>0.1.0.SNAPSHOT</version>
    <packaging>apk</packaging>
    <name>spring-android-client</name>
    <url>http://projects.spring.io/spring-android</url>
    <inceptionYear>2010</inceptionYear>
    <organization>
        <name>Spring</name>
        <url>http://spring.io</url>
    </organization>

    <properties>
        <android-platform>21</android-platform>
        <android-maven-plugin-version>3.9.0-rc.2</android-maven-plugin-version>
        <maven-compiler-plugin-version>3.1</maven-compiler-plugin-version>
        <java-version>1.6</java-version>
        <com.google.android-version>4.1.1.4</com.google.android-version>
        <org.springframework.android-version>2.0.0.M2</org.springframework.android-version>
  <org.apache.httpcomponents-version>4.3.5</org.apache.httpcomponents-version>
        <com.fasterxml.jackson-version>2.3.4</com.fasterxml.jackson-version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.google.android</groupId>
            <artifactId>android</artifactId>
            <version>${com.google.android-version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.android</groupId>
            <artifactId>spring-android-rest-template</artifactId>
            <version>${org.springframework.android-version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient-android</artifactId>
            <version>${org.apache.httpcomponents-version}</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>${com.fasterxml.jackson-version}</version>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>spring-milestone</id>
            <name>Spring Milestones</name>
            <url>http://repo.spring.io/libs-milestone</url>
        </repository>
    </repositories>

    <build>
        <finalName>${project.artifactId}</finalName>
        <sourceDirectory>src</sourceDirectory>
        <plugins>
            <plugin>
                <groupId>com.jayway.maven.plugins.android.generation2</groupId>
                <artifactId>android-maven-plugin</artifactId>
                <version>${android-maven-plugin-version}</version>
                <configuration>
                    <sdk>
                        <platform>${android-platform}</platform>
                    </sdk>
                    <deleteConflictingFiles>true</deleteConflictingFiles>
                    <undeployBeforeDeploy>true</undeployBeforeDeploy>
                </configuration>
                <extensions>true</extensions>
            </plugin>
            <plugin>
```

5.5 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.

```
$ mvn clean install
```

The Android Maven Plugin provides several *goals* for use in building and deploying your application. You can configure a specific emulator in the plugin configuration, or if you omit the emulator name, the plugin will attempt to execute the specified goal on all available emulators and devices.

The following command starts the emulator specified in the Maven Android Plugin section of the POM file. If no emulator name is configured, then the plugin attempts to start an AVD with the name of `Default`.

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

Deploys the built apk file to the emulator or attached device.

```
$ mvn android:deploy
```

Starts the app on the emulator or attached device.

```
$ mvn android:run
```

Displays a list of help topics for the Android Maven Plugin.

```
$ mvn android:help
```