< ALL GUIDES

Consuming a RESTful Web Service

This guide walks you through the process of creating an application that consumes a RESTful web service.

What You Will Build

You will build an application that uses Spring's `RestTemplate` to retrieve a random Spring Boot quotation at https://gturnquist-quoters.cfapps.io/api/random.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- You can also import the code straight into your IDE:
    - Spring Tool Suite (STS)
    - IntelliJ IDEA

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Starting with Spring Initializr.

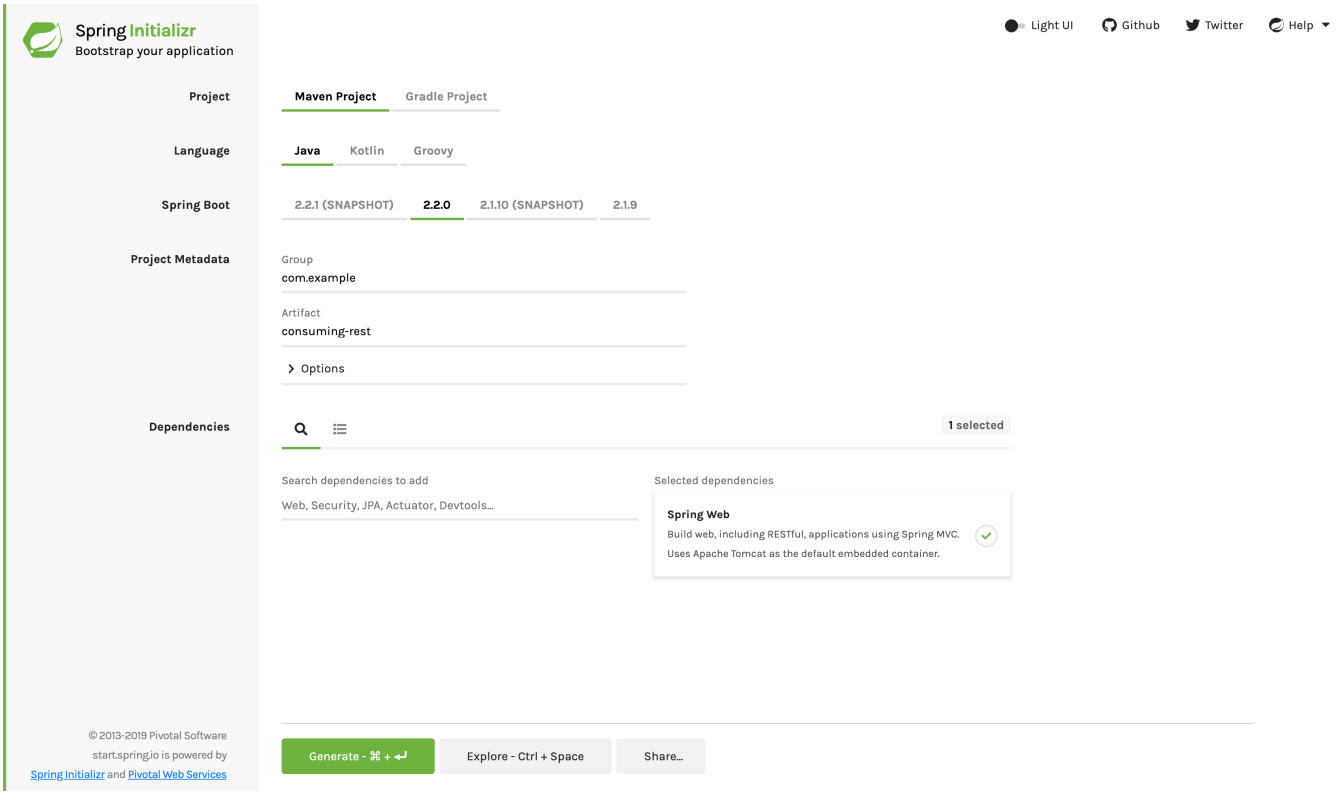
To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-consuming-rest.git`
- cd into `gs-consuming-rest/initial`
- Jump ahead to Fetching a REST Resource.

When you finish, you can check your results against the code in `gs-consuming-rest/complete`.

Starting with Spring Initializr

For all Spring applications, you can start with the Spring Initializr. The Initializr offers a fast way to pull in all the dependencies you need for an application and does a lot of the set up for you. Because this example needs to be nothing more than a web application, you need to include only the Web dependency. The following image shows the Initializr set up for this sample project:



The preceding image shows the Initializr with Maven chosen as the build tool. You can also use Gradle. It also shows values of `com.example` and `consuming-rest` as the Group and Artifact, respectively. You will use those values throughout the rest of this sample.

The following listing shows the `pom.xml` file created when you choose Maven:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apach
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.1.7.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>consuming-rest</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>consuming-rest</name>
        <description>Demo project for Spring Boot</description>

        <properties>
                <java.version>1.8</java.version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

The following listing shows the `build.gradle` file created when you choose Gradle:

```
plugins {
        id 'org.springframework.boot' version '2.1.7.RELEASE'
        id 'io.spring.dependency-management' version '1.0.8.RELEASE'
        id 'java'
}
```

```
group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-web'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

These build files can be this simple because `spring-boot-starter-web` includes everything you need to build a web application, including the Jackson classes you need to work with JSON.

Fetching a REST Resource

With project setup complete, you can create a simple application that consumes a RESTful service.

A RESTful service has been stood up at https://gturnquist-quoters.cfapps.io/api/random. It randomly fetches quotations about Spring Boot and returns them as JSON documents.

If you request that URL through a web browser or curl, you receive a JSON document that looks something like this:

COPY

```
{
   type: "success",
   value: {
      id: 10,
      quote: "Really loving Spring Boot, makes stand alone Spring apps easy."
   }
}
```

That is easy enough but not terribly useful when fetched through a browser or through curl.

A more useful way to consume a REST web service is programmatically. To help you with that task, Spring provides a convenient template class called `RestTemplate`. `RestTemplate` makes interacting with most RESTful services a one-line incantation. And it can even bind that data to custom domain types.

First, you need to create a domain class to contain the data that you need. The following listing shows the `Quote` class, which you can use as your domain class:

src/main/java/com/example/consumingrest/Quote.java

COPY

```
package com.example.consumingrest;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class Quote {

  private String type;
  private Value value;

  public Quote() {
  }

  public String getType() {
    return type;
  }

  public void setType(String type) {
    this.type = type;
  }

  public Value getValue() {
    return value;
  }

  public void setValue(Value value) {
    this.value = value;
  }

  @Override
  public String toString() {
    return "Quote{" +
        "type='" + type + '\'' +
        ", value=" + value +
        '}';
  }
}
```

This simple Java class has a handful of properties and matching getter methods. It is annotated with `@JsonIgnoreProperties` from the Jackson JSON processing library to indicate that any properties not bound in this type should be ignored.

To directly bind your data to your custom types, you need to specify the variable name to be exactly the same as the key in the JSON document returned from the API. In case your variable name and key in JSON doc do not match, you can use `@JsonProperty` annotation to specify the exact key of the JSON document. (This example matches each variable name to a JSON key, so you do not need that annotation here.)

You also need an additional class, to embed the inner quotation itself. The `Value` class fills that need and is shown in the following listing (at `src/main/java/com/example/consumingrest/Value.java`):

COPY

```
package com.example.consumingrest;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class Value {

  private Long id;
  private String quote;

  public Value() {
  }

  public Long getId() {
    return this.id;
  }

  public String getQuote() {
    return this.quote;
  }

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

  public void setQuote(String quote) {
    this.quote = quote;
  }

  @Override
  public String toString() {
    return "Value{" +
        "id=" + id +
        ", quote='" + quote + '\'' +
        '}';
  }
}
```

This uses the same annotations but maps onto other data fields.

Finishing the Application

The Initalizr creates a class with a `main()` method. The following listing shows the class the Initializr creates (at `src/main/java/com/example/consumingrest/ConsumingRestApplication.java`):

```
package com.example.consumingrest;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ConsumingRestApplication {

        public static void main(String[] args) {
                SpringApplication.run(ConsumingRestApplication.class, args);
        }

}
```

Now you need to add a few other things to the `ConsumingRestApplication` class to get it to show quotations from our RESTful source. You need to add:

- A logger, to send output to the log (the console, in this example).
- A `RestTemplate`, which uses the Jackson JSON processing library to process the incoming data.
- A `CommandLineRunner` that runs the `RestTemplate` (and, consequently, fetches our quotation) on startup.

The following listing shows the finished `ConsumingRestApplication` class (at `src/main/java/com/example/consumingrest/ConsumingRestApplication.java`):

```
package com.example.consumingrest;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class ConsumingRestApplication {

        private static final Logger log =
LoggerFactory.getLogger(ConsumingRestApplication.class);

        public static void main(String[] args) {
                SpringApplication.run(ConsumingRestApplication.class, args);
        }

        @Bean
        public RestTemplate restTemplate(RestTemplateBuilder builder) {
                return builder.build();
```

```
        }

        @Bean
        public CommandLineRunner run(RestTemplate restTemplate) throws
Exception {
                return args -> {
                        Quote quote = restTemplate.getForObject(
                                        "https://gturnquist-
quoters.cfapps.io/api/random", Quote.class);
                        log.info(quote.toString());
                };
        }
}
```

Running the Application

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-consuming-rest-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-consuming-rest-0.1.0.jar
```

> The steps described here create a runnable JAR. You can also build a classic WAR file.

You should see output similar to the following but with a random quotation:

```
2019-08-22 14:06:46.506  INFO 42940 --- [           main] c.e.c.ConsumingRestAppl
```

If you see an error that reads,
`Could not extract response: no suitable HttpMessageConverter found for respons`
it is possible that you are in an environment that cannot connect to the backend service
are behind a corporate proxy. Try setting the `http.proxyHost` and `http.proxyPort` sy
environment.

Summary

Congratulations! You have just developed a simple REST client by using Spring Boot.

See Also

The following guides may also be helpful:

- Building a RESTful Web Service
- Consuming a RESTful Web Service with AngularJS
- Consuming a RESTful Web Service with jQuery
- Consuming a RESTful Web Service with rest.js
- Accessing GemFire Data with REST
- Accessing MongoDB Data with REST
- Accessing data with MySQL
- Accessing JPA Data with REST
- Accessing Neo4j Data with REST
- Securing a Web Application
- Building an Application with Spring Boot
- Creating API Documentation with Restdocs
- Enabling Cross Origin Requests for a RESTful Web Service
- Building a Hypermedia-Driven RESTful Web Service

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.