Consuming a RESTful Web Service

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

What you'll do

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

What you'll need

About 15 minutes
A favorite text editor or IDE
JDK 1.8 or later
Gradle 2.3+ or Maven 3.0+
You can also import the code from this guide as well as view the web page directly into Spring Tool Suite (STS) and work your way through it from there.

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 Build with Gradle.
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 Fetch a REST resource.
When you're finished, you can check your results against the code in `gs-consuming-rest/complete`.

Gradle

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Gradle and Maven is included here. If you're not familiar with either, refer to Building Java Projects with Gradle or Building Java Projects with Maven.
Create the directory structure
In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

Create a Gradle build file

Below is the initial Gradle build file.

`build.gradle`

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:1.4.2.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'gs-consuming-rest'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter")
    compile("org.springframework:spring-web")
    compile("com.fasterxml.jackson.core:jackson-databind")
    testCompile("junit:junit")
}
```

The Spring Boot gradle plugin provides many convenient features:

It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.

It searches for the `public static void main()` method to flag as a runnable class.

It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot's chosen set of versions.

Maven

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Maven is included here. If you're not familiar with Maven, refer to Building Java Projects with Maven.

Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

`pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<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</groupId>
    <artifactId>gs-consuming-rest</artifactId>
    <version>0.1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
    </parent>

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

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </dependency>
    </dependencies>


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

```
            </plugin>
        </plugins>
    </build>

</project>
```

The Spring Boot Maven plugin provides many convenient features:
It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
It searches for the `public static void main()` method to flag as a runnable class.
It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot's chosen set of versions.

IDE

- Read how to import this guide straight into Spring Tool Suite.
- Read how to work with this guide in IntelliJ IDEA.

Fetch 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 http://gturnquist-quoters.cfapps.io/api/random. It randomly fetches quotes about Spring Boot and returns them as a JSON document.
If you request that URL through your web browser or curl, you'll receive a JSON document that looks something like this:

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

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, create a domain class to contain the data that you need.

`src/main/java/hello/Quote.java`

```
package hello;

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 +
                '}';
    }
}
```

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

An additional class is needed to embed the inner quotation itself.

`src/main/java/hello/Value.java`

```
package hello;

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 simply maps onto other data fields.

Make the application executable

Although it is possible to package this service as a traditional WAR file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java `main()` method. Along the way, you use Spring's support for embedding the Tomcat servlet container as the HTTP runtime, instead of deploying to an external instance.

Now you can write the `Application` class that uses `RestTemplate` to fetch the data from our Spring Boot quotation service.

`src/main/java/hello/Application.java`

```
public class Application {

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

    public static void main(String args[]) {
        RestTemplate restTemplate = new RestTemplate();
        Quote quote = restTemplate.getForObject("http://gturnquist-
quoters.cfapps.io/api/random", Quote.class);
        log.info(quote.toString());
    }

}
```

Because the Jackson JSON processing library is in the classpath, `RestTemplate` will use it (via a message converter) to convert the incoming JSON data into a `Quote` object. From there, the contents of the `Quote` object will be logged to the console.

Here you've only used `RestTemplate` to make an HTTP `GET` request. But `RestTemplate` also supports other HTTP verbs such as `POST`, `PUT`, and `DELETE`.

Managing the Application Lifecycle with Spring Boot

So far we haven't used Spring Boot in our application, but there are some advantages in doing so, and it isn't hard to do. One of the advantages is that we might want to let Spring Boot manage the message converters in the `RestTemplate`, so that customizations are easy to add declaratively. To do that we use `@SpringBootApplication` on the main class and convert the main method to start it up, like in any Spring Boot application. Finally we move the `RestTemplate` to a `CommandLineRunner` callback so it is executed by Spring Boot on startup:

`src/main/java/hello/Application.java`

```
package hello;

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 Application {

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

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

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

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

The `RestTemplateBuilder` is injected by Spring, and if you use it to create a `RestTemplate` then you will benefit from all the autoconfiguration that happens in Spring Boot with message converters and request factories. We also extract the `RestTemplate` into a `@Bean` to make it easier to test (it can be mocked more easily that way).

Build an executable JAR

You can run the application from the command line with Gradle or Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This 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 are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

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

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

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

The procedure above will create a runnable JAR. You can also opt to build a classic WAR fileinstead.

You should see output like the following, with a random quote:

```
2015-09-23 14:22:26.415  INFO 23613 --- [main] hello.Application  :
Quote{type='success', value=Value{id=12, quote='@springboot with @springframework is
pure productivity! Who said in #java one has to write double the code than in other
langs? #newFavLib'}}
```

If you see the error `Could not extract response: no suitable HttpMessageConverter found for response type [class hello.Quote]` it's possible you are in an environment that cannot connect to the backend service (which sends JSON if you can reach it). Maybe you are behind a corporate proxy? Try setting the standard system properties `http.proxyHost` and `http.proxyPort` to values appropriate for your environment.

Summary

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

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