


---

I released five new sample lessons from my Test With Spring course: [Introduction to Spock Framework](#)

---

## Spring Batch Tutorial: Reading Information From a REST API

 Petri Kainulainen  March 13, 2016

 14 comments

 [Spring Batch](#), [Spring Framework](#)

Spring Batch has a good support for reading input data from different data sources such as [files](#) and [databases](#).

However, it doesn't have a built-in support for reading input data from a REST API. This means that we have to [create a custom \*ItemReader\*](#).

This blog post describes how we can create a custom *ItemReader* that reads the input data of our batch job by using the *RestTemplate* class.

Let's get started.

If you are not familiar with Spring Batch, **you should read** the following blog posts before you continue reading this blog post:

- [Spring Batch Tutorial: Introduction](#) specifies the term batch job, explains why you should use Spring Batch, and identifies the basic building blocks of a Spring Batch job.
- [Spring Batch Tutorial: Getting the Required Dependencies With Maven](#) describes how you can get Spring Batch dependencies with Maven.
- [Spring Batch Tutorial: Getting the Required Dependencies With Gradle](#) describes how you can get Spring Batch dependencies with Gradle.
- [Spring Batch Tutorial: Reading Information From a File](#) describes how you can read information from CSV and XML files.
- [Spring Batch Tutorial: Reading Information From a Database](#) describes how you can read input data from a database by using database cursors and pagination.
- [Spring Batch Tutorial: Creating a Custom \*ItemReader\*](#) describes how you can create a custom *ItemReader*.

During this tutorial we will implement several Spring Batch jobs that processes the student information of an online course. This time we have to implement an *ItemReader* that fetches the student information by sending a GET request to an external REST API. This API returns always the following JSON:

```

1  [
2    {
3      "emailAddress": "tony.test@gmail.com",
4      "name": "Tony Tester",
5      "purchasedPackage": "master"
6    },
7    {
8      "emailAddress": "nick.newbie@gmail.com",
9      "name": "Nick Newbie",
10     "purchasedPackage": "starter"
11   },
12   {
13     "emailAddress": "ian.intermediate@gmail.com",
14     "name": "Ian Intermediate",
15     "purchasedPackage": "intermediate"
16   }
17 ]

```

We have to transform that JSON into *StudentDTO* objects which are processed by our batch job. The *StudentDTO* class contains the information of a single student, and its source code looks as follows:

```

1  public class StudentDTO {
2
3      private String emailAddress;
4      private String name;
5      private String purchasedPackage;
6
7      public StudentDTO() {}
8
9      public String getEmailAddress() {
10         return emailAddress;
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public String getPurchasedPackage() {
18         return purchasedPackage;
19     }
20
21     public void setEmailAddress(String emailAddress) {
22         this.emailAddress = emailAddress;
23     }
24
25     public void setName(String name) {
26         this.name = name;
27     }
28
29     public void setPurchasedPackage(String purchasedPackage) {
30         this.purchasedPackage = purchasedPackage;
31     }
32 }

```

Let's create a custom *ItemReader* that reads input data from a REST API.

## Creating an ItemReader That Reads Input Data From a REST API

We can create our *ItemReader* by following these steps:

1. Create a *RESTStudentReader* class.
2. Implement the *ItemReader* interface and set the type of the returned object to *StudentDTO*.
3. Add the following *private* fields to the *RESTStudentReader* class:
  - The *final apiUrl* field contains the url of the external REST API.

- The *nextStudentIndex* field contains the index of the next *StudentDTO* object.
  - The *studentData* field contains the list of found *StudentDTO* objects.
4. Add a constructor to the created class and ensure that the constructor has the following constructor arguments:
    - The url of the external REST API.
    - A *RestTemplate* object.
  5. Implement the constructor by storing its constructor arguments into the fields of the created object. Set the value of the *nextStudentIndex* field to 0.
  6. Add a *public read()* method to created class, and specify that the method returns a *StudentDTO* object and can throw an *Exception*.
  7. Implement the *read()* method by following these rules:
    - If the student information has not been fetched, fetch the student information from the external API by using the *RestTemplate* object.
    - If the next student is found, return the found *StudentDTO* object and increase the index of the next student by 1.
    - If the next student is not found, return *null*.

The source code of the *RESTStudentReader* class looks as follows:

```

1  import org.springframework.batch.item.ItemReader;
2  import org.springframework.http.ResponseEntity;
3  import org.springframework.web.client.RestTemplate;
4
5  import java.util.Arrays;
6  import java.util.List;
7
8  class RESTStudentReader implements ItemReader<StudentDTO> {
9
10
11     private final String apiUrl;
12     private final RestTemplate restTemplate;
13
14     private int nextStudentIndex;
15     private List<StudentDTO> studentData;
16
17     RESTStudentReader(String apiUrl, RestTemplate restTemplate) {
18         this.apiUrl = apiUrl;
19         this.restTemplate = restTemplate;
20         nextStudentIndex = 0;
21     }
22
23     @Override
24     public StudentDTO read() throws Exception {
25         if (studentDataIsNotInitialized()) {
26             studentData = fetchStudentDataFromAPI();
27         }
28
29         StudentDTO nextStudent = null;
30
31         if (nextStudentIndex < studentData.size()) {
32             nextStudent = studentData.get(nextStudentIndex);
33             nextStudentIndex++;
34         }
35
36         return nextStudent;
37     }
38
39     private boolean studentDataIsNotInitialized() {
40         return this.studentData == null;
41     }
42
43     private List<StudentDTO> fetchStudentDataFromAPI() {
44         ResponseEntity<StudentDTO[]> response = restTemplate.getForEntity(
45             apiUrl,
46             StudentDTO[].class
47         );

```

[Accept](#)

**Additional Reading:**

- [Spring Framework Reference Documentation: 27.10 Accessing RESTful services on the Client](#)
- [Spring Batch Tutorial: Creating a Custom ItemReader](#)

Before we can use our new *ItemReader*, we have to configure the *RestTemplate* bean. Let's find out how we can configure this bean.

## Configuring the RestTemplate Bean

We can configure the *RestTemplate* bean by following these steps:

1. Add a *restTemplate()* method to our application context configuration class, ensure that the method returns a *RestTemplate* object, and annotate the method with the *@Bean* annotation.
2. Implement the method by returning a new *RestTemplate* object.

If we use Spring Framework, the source code of our application context configuration class looks as follows:

```
1 import org.springframework.context.annotation.Bean;
2 import org.springframework.context.annotation.Configuration;
3 import org.springframework.web.client.RestTemplate;
4
5 @Configuration
6 public class SpringBatchExampleContext {
7
8     @Bean
9     RestTemplate restTemplate() {
10         return new RestTemplate();
11     }
12 }
```

If we use Spring Boot, we can also add the *restTemplate()* method into our *@SpringBootApplication* class. The source code of this class looks as follows:

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.web.client.RestTemplate;
5
6 @SpringBootApplication
7 public class SpringBatchExampleApplication {
8
9     @Bean
10     RestTemplate restTemplate() {
11         return new RestTemplate();
12     }
13
14     public static void main(String[] args) {
15         SpringApplication.run(SpringBatchExampleApplication.class, args);
16     }
17 }
```

We might have to add some additional dependencies into our build script before we can configure the *RestTemplate* bean. These dependencies are described in the following:

- If we are using Spring Framework, we have to add the *spring-webmvc* dependency

After we have configured the *RestTemplate* bean, we can finally configure our new *ItemReader* bean.

## Configuring the ItemReader Bean

We can configure our *ItemReader* bean by following these steps:

1. Create a configuration class and annotate the created class with the `@Configuration` annotation.
2. Add a new method to the created class, ensure that this method returns an *ItemReader*<*StudentDTO*> object, and annotate this method with the `@Bean` annotation.
3. Ensure that the created method takes an *Environment* object and a *RestTemplate* object as method parameters.
4. Implement the method by returning a new *RESTStudentReader* object. When we create a new *RESTStudentReader* object, we have to pass the following objects as constructor arguments:
  - The url of the external REST API. We will fetch this information from a properties file by using the *Environment* object given as a method parameter.
  - The *RestTemplate* object that is used to fetch the student information from the external REST API.

The source code of our configuration class looks as follows:

```
1  import org.springframework.batch.item.ItemReader;
2  import org.springframework.context.annotation.Bean;
3  import org.springframework.context.annotation.Configuration;
4  import org.springframework.core.env.Environment;
5  import org.springframework.web.client.RestTemplate;
6
7  @Configuration
8  public class RESTStudentJobConfig {
9
10     @Bean
11     ItemReader<StudentDTO> restStudentReader(Environment environment,
12                                             RestTemplate restTemplate) {
13         return new RESTStudentReader(
14             environment.getRequiredProperty("rest.api.to.database.job.api.url"),
15             restTemplate
16         );
17     }
18 }
```

Let's summarize what we learned from this blog post.

## Summary

This blog post has taught us two things:

- Spring Batch doesn't have an *ItemReader* that can read information from a REST API.
- If we want to read the input data of our batch job from an external REST API, we have to read this information by using the *RestTemplate* class

**P.S.** You can get the example applications of this blog post from Github: [Spring example](#) and [Spring Boot example](#).



## GET FREE EBOOK

Subscribe my email newsletter **AND** you will get my eBook:  
Writing Integration Tests for Spring Powered Repositories **FOR FREE**.

Email Address...

SUBSCRIBE

I will never rent, sell, or share your email address.

---

### RELATED POSTS

**SPRING FRAMEWORK /**

#### **Screencast: Unit Testing of Spring MVC Controllers – “Normal” Controllers**

**UNCATEGORIZED /**

#### **10 Most Popular Blog Posts of 2015**

**SPRING FRAMEWORK /**

#### **Integration Testing of Spring MVC Applications: Forms**



##### **About the Author**

Petri Kainulainen is passionate about software development and continuous improvement. He is specialized in software development with the Spring Framework and is the author of [Spring Data](#) book.

[About Petri Kainulainen →](#)

##### **Connect With Me**



14 comments... [add one](#)

This was a great tutorial on how to use a REST API as the source of my data for Spring Batch. I am struggling with the best way to unit test it though. Do you have any suggestions or additions for this example?

Thanks!

REPLY



Petri

August 8, 2016, 21:49

Hi,

You could move the logic that reads the input data to a separate component (I will call this component `RestInputReader`) and inject that component to the Spring Batch reader. This gives you the possibility to replace the `RestInputReader` with a stub when you are writing unit tests for the Spring Batch reader.

I wouldn't write unit tests for the `RestInputReader` because it doesn't make much sense to replace `RestTemplate` with a test double because `RestTemplate` does all the work. I would test it by writing integration tests. If the REST API is an external API, I would replace it with a simple stub.

If you have any additional questions, don't hesitate to ask them.

REPLY



Senthil

September 30, 2016, 10:09

I have gone through Spring Batch Tutorial: Reading Information From a REST API post. It is very useful. Can please let me know how to run this project?

REPLY



Petri

September 30, 2016, 12:27

Hi,

Sure. Just clone [this repository](#), select the example you want to run (Spring or Spring Boot), and run the command described in the README.

REPLY

[Facebook](#)[Twitter](#)[Google+](#)[LinkedIn](#)

Hi Petri,

How to read XML file using custom itemreader,itemprocessor write into database using itemwriter.

Pls can u share code for above one

Waiting for your response

Advanced thank

Dandu

8553562402

REPLY



Petri

October 11, 2016, 20:08

Hi,

Unfortunately I cannot cover this topic in a single answer. However, you can find my Spring Batch tutorial and a few other great resources from my [Spring Batch resource page](#). Also, [this blog post](#) explains how you can read data from XML file and write it to a database.

REPLY



Florent

December 25, 2016, 09:12

Hi Petri,

Thank you for this great post. I have approximately the same use case than you. But my problem is that my rest service is using a database containing more than 150.000 entries. So I am afraid that my memory is not big enough. How can I handle that problem?

REPLY



Petri

December 25, 2016, 15:02

Hi Florent,

I agree that it's not a good to load the content of your database into memory. I would probably implement an API that supports pagination and write an ItemReader that reads the data one page at the time.

If you don't know how you can do it, don't hesitate to ask additional questions



[Facebook](#)[Twitter](#)[Google+](#)[LinkedIn](#)**Claudio**

March 31, 2017, 18:46

Thanks Petri, it helped me a lot your example.

[REPLY](#)**Petri**

April 3, 2017, 22:31

You are welcome. I am happy hear that this blog post was useful to you.

[REPLY](#)**Sherin**

May 1, 2017, 17:43

Petri

The `ItemReader` you have is stateful. Do you think it is thread-safe?

~Sherin

[REPLY](#)**Sherin**

May 1, 2017, 17:49

By the way it is a great example

~Sherin K Syriac

[REPLY](#)**Petri**

May 1, 2017, 23:46

Thank you for your kind words. I really appreciate them. About your question:

Yes, my `ItemReader` is stateful, and Spring Batch assumes that this is the case. [The Javadoc of the `ItemReader` interface](#) states that:

Implementations are expected to be stateful and will be called multiple times for each batch, with each call to `read()` returning a different value and finally returning `null` when all input data is exhausted.

[Facebook](#)[Twitter](#)[Google+](#)[LinkedIn](#)[REPLY](#)

---

### Leave a Comment

---

PREVIOUS POST: [JAVA TESTING WEEKLY 10 / 2016](#)

NEXT POST: [JAVA TESTING WEEKLY 11 / 2016](#)

---



## GET FREE EBOOK

Subscribe my email newsletter **AND** you will get my eBook: Writing Integration Tests for Spring Powered Repositories **FOR FREE**.

SUBSCRIBE

*I will never sell, rent, or share your email address.*

### WRITE BETTER TESTS

[Java Testing Weekly](#)

[Spring MVC Test Tutorial](#)