How to Read JSON Data with WebClient

A guide to **Read JSON Data with Spring 5 WebClient** and mapping JSON Objects to POJOs and Objects using WebFlux Mono and Flux.

JSON Objects

Let's consider we have an Employee Service, which our *WebClient* will consume. The GET employee endpoint on the service returns list of all employees in JSON format.

Service Endpoints
Next are the endpoints, our WebClient will execute and consume the JSON data they return.

- GET /employees
- GET /employees/{id}

Sample JSON Data

The response of the service endpoint that returns an array of JSON objects, will look like this.

```
[
   {
      "id":111,
      "name":"Jon",
      "department":{
         "id":222,
         "name":"Sales"
      }
   },
   {
      "id":112,
      "name":"Mac",
      "department":{
         "id":223,
         "name":"Engineering"
      }
   }
]
```

Setup

We need a minimal setup for this application. First, is to add required maven/gradle dependency and the other is to create model classes to parse the JSON data into.

Dependency

In order to use the *WebClient*, we need to add a dependency on Spring WebFlux. In a Spring Boot project, we can add a starter dependency for WebFlux.

Maven Dependency (pom.xml)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

or, **Gradle Dependency (build.gradle)**

```
implementation group: 'org.springframework.boot', name: 'spring-boot-starte
```

Model Classes

Next, we will create model classes into which, we will transform the JSON objects. Looking at the sample JSON, we can figure out there are employee objects having department objects nested within.

Employee

```java
public class Employee {
    private long id;
    private String name;
    private Department department;

    // ..Constructors and Getter, Setter Methods
}
```

Department

```java
public class Department {
    private long id;
    private String name;

    // ..Constructors and Getter, Setter Methods
}
```

Read a JSON Object with WebClient

Mono is a reactive publisher, that can emit 0 zero or 1 elements. Thus, in order to retrieve a single JSON resource with *WebClient*, we should use *Mono* publisher. We will use *WebClient* to read a JSON object and parse it into POJO.

Example of *WebClient* reading single JSON Object as a POJO with Mono.

```
Mono<Employee> employeeMono = WebClient
    .create(SERVICE_URL + "/employees/111")
    .get()
    .retrieve()
    .bodyToMono(Employee.class);
```

Once, we get the *Mono* of *Employee* POJO object, we can then call *block()* on the *Mono* to get *Employee* POJO instance.

```
Employee employee = employeeMono
    .share().block();
```

Read JSON Object Array with WebClient

A JSON array of objects can contain zero or N objects. Also, a Flux publisher can emit zero or N number of elements that makes it a perfect fit for reading a JSON list/array. However, we can also read complete the

JSON array in the form of an array of POJO. To do that, we can use Mono.

JSON Array as a Flux of POJO

Let's see how can we read a JSON Object using WebClient and create a Flux of POJO. In order to read a Flux of POJO object, we need to use **bodyToFlux()** method.

Example of **reading our JSON Object with *WebClient* as a *Flux* of *Employee* POJO**

```
Flux<Employee> employeeFlux = WebClient
    .create(SERVICE_URL + "/employees")
    .get()
    .retrieve()
    .bodyToFlux(Employee.class);
```

Then, we can use **Java Stream Collector (/java-8-stream-collectors/)** on the flux to collect its elements in a *List*.

```
List<Employee> employees = employeeFlux
                .collect(Collectors.toList())
                .share().block();
```

The *collect()* on the Flux returns a *Mono* of *List*. Thus we are executing *block()* to get a *List* of *Employee* objects.

JSON Array as a Mono of POJO Array

As stated earlier we can also use *WebClient* and read a JSON Array in the form of Array of POJO using *Mono*.
Example of **reading our JSON Array using *WebClient Mono* to create an Array of *Employee* POJO**.

```
Mono<Employee[]> employeesMono =  WebClient
    .create(SERVICE_URL + "/employees")
    .get()
    .retrieve()
    .bodyToMono(Employee[].class);
```

We have used *bodyToMono()* by providing Employee Array class. That returns a *Mono* of *Employee[]*.

```
Employee[] employees = employeesMono
    .share().block();
```

Next, we can use a simple *block()* on the Mono to retrieve the *Employee* POJO Array.

JSON Array as a Mono of Generic Array

In the previous example we are using *WebClient* to parse a JSON response into an *Employee* bean array. Similarly, we can collect a *Mono* of *Object* class.

Example of **reading our JSON Array using *WebClient* to get *Mono* of *Objects***.

```
Mono<Object[]> objectsMono = WebClient
    .create(SERVICE_URL + "/employees")
    .get()
    .retrieve()
    .bodyToMono(Object[].class);
```

Similarly, we can call *block()* on the the *Mono* to get *Object[]*.

```
Object[] objects = objectsMono
    .share().block();
```