ParallelWebClients

Sunday, January 16, 2022 4:55 PM

Simultaneous Web Clients Requests

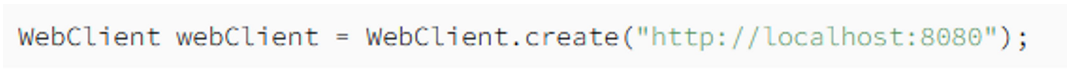
Typically when making HTTP requests in our applications, we execute these calls sequentially. However, there are occasions when we might want to perform these requests simultaneously.
For example, we may want to do this when retrieving data from multiple sources or when we simply want to try giving our application a performance boost.

A Simple User Service

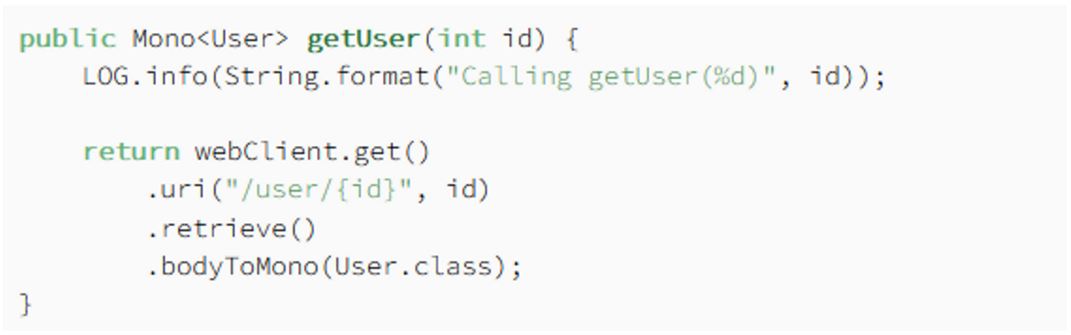
We're going to be using a simple *User* API in our examples.
This API has a GET method that exposes one method *getUser* for retrieving a user using the id as a parameter.

Let's take a look at how to make a single call to retrieve a user for a given id:

```
WebClient webClient = WebClient.create("http://localhost:8080");
Public Mono<User> getUser(intid){
   LOG.info(String.format("Calling getUser(%d)", id));
          return webClient.get()
                              .uri("/user/{id}", id)
                              .retrieve()
                              .bodyToMono(User.class);
}
```



```
WebClient webClient = WebClient.create("http://localhost:8080");
```



```
public Mono<User> getUser(int id) {
    LOG.info(String.format("Calling getUser(%d)", id));

    return webClient.get()
        .uri("/user/{id}", id)
        .retrieve()
        .bodyToMono(User.class);
}
```

Making Simultaneous *WebClient* Calls

In this section, we're going see several examples for calling our *getUser* method concurrently. We'll also take a look at both publisher implementations *Flux* and *Mono* in the examples as well.

4.1. Multiple Calls to the Same Service

Let's now imagine that we want to fetch data about five users simultaneously and return the result as a list of users:

```
Public Flux fetchUsers(List userIds){
   return Flux.fromIterable(userIds)
                 .flatMap(this::getUser);
}
```

Let’s decompose the steps to understand what we've done:

We begin by creating a Flux from our list of *userIds* using the static *fromIterable* method.

Next, we invoke *flatMap* to run the getUser method we created previously.
This reactive operator has a concurrency level of 256 by default, meaning it executes at most 256 *getUser* calls simultaneously.

This number is configurable via method parameter using an overloaded version of *flatMap*.
It's worth noting, that since operations are happening in parallel, we don't know the resulting order. If we need to maintain the input order, we can use ***flatMapSequential*** operator instead.

As Spring WebClient uses a non-blocking HTTP client under the hood, there is no need to define any Scheduler by the user. *WebClient* takes care of scheduling calls and publishing their results on appropriate threads internally, without blocking.

Multiple Calls to Different Services Returning the Same Type

Let's now take a look at how we can call multiple services simultaneously.

In this example, we're going to create another endpoint which returns the same *User* type:

```
Public Mono<User> getOtherUser(int id){
   return webClient.get()
                        .uri("/otheruser/{id}", id)
                        .retrieve()
                        .bodyToMono(User.class);
              }
```

Now, the method to perform two or more calls in parallel becomes:

```
Public Flux fetchUserAndOtherUser(int id){
   return Flux.merge(getUser(id), getOtherUser(id));
}
```

The main difference in this example is that we've used the static method *merge* instead of the *fromIterable* method. Using the merge method, we can combine two or more *Flux*es into one result.

Multiple Calls to Different Services Different Types

The probability of having two services returning the same thing is rather low. **More typically we'll have another service providing a different response type and our goal is to merge two (or more) responses**.

The *Mono* class provides the static zip method which lets us combine two or more results:

```
Public Mono fetchUserAndItem(intuser Id, intitem Id){
   Mono user = getUser(userId);
   Mono item = getItem(itemId);
return Mono.zip(user, item, UserWithItem::new);
}
```

The *zip* method combines the given *user* and *item Mono*s into a new *Mono* with the type *UserWithItem*. This is a simple POJO object which wraps a user and item.

Reference From <https://www.baeldung.com/spring-webclient-simultaneous-calls>