Spring 5 WebClient

Thursday, September 23, 2021

2:08 PM

**WebClient is an interface which represents an entry point to performing web requests.**

**WebClient is reactive , non-blocking solution that works with http/1.1 protocol.**

**Webclient supports both synchronous and asynchronous operations.**

**MAVEN Dependency :**

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

**Gradle :**

|  |
| --- |
| dependencies {  compile 'org.springframework.boot:spring-boot-starter-webflux'  } |

**Follow below steps :**

**1. Create an instance**

**2. Make a request**

**3. Handle response**

**1. CREATING a WEBClient instance.**

**The first one is creating a WebClient object with default settings:**

**WebClient client = WebClient.create();**

**The second option is to initiate a WebClient instance with a given base URI:**

**WebClient client = WebClient.create("http://localhost:8080");**

**The third option (and the most advanced one) is building a client by using the DefaultWebClientBuilder class, which**

**allows full customization:**

**WebClient client = WebClient.builder()**

**.baseUrl("http://localhost:8080")**

**.defaultCookie("cookieKey", "cookieValue")**

**.defaultHeader(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)**

**.defaultUriVariables(Collections.singletonMap("url", "**[**http://localhost:8080**](http://localhost:8080)**"))**

**.build();**

**Example :**

**public WebClientApi() {**

**LOG.debug("In WebClientApi constructor");**

**this.webClient = WebClient.builder()**

**.baseUrl("http://localhost:8383")**

**.defaultHeader(HttpHeaders.ACCEPT, MediaType.APPLICATION\_JSON\_VALUE)**

**.filter(logRequest())**

**.build();**

**}**

**private ExchangeFilterFunction logRequest() {**

**return (request, nxt) ->{**

**LOG.debug("WEBClient Request is {} and {} ",request.method(),request.url());**

**request.headers().forEach((key,values) -> values.forEach(value -> LOG.debug("{} = {}",key , value)));**

**return nxt.exchange(request);**

**};**

**}**

**1.1 : Creating a WebClient Instance with Timeouts**

**The default HTTP timeouts is 30 seconds which is too slow, to customize this behavior, we can create an HttpClient instance and configure**

**our WebClient to use it.**

**Below are steps we need to add.**

**1. set the connection timeout via the ChannelOption.CONNECT\_TIMEOUT\_MILLIS option**

**2. set the read and write timeouts using a ReadTimeoutHandler and a WriteTimeoutHandler, respectively**

**3. configure a response timeout using the responseTimeout directive**

**As we said, all these have to be specified in the HttpClient instance we'll configure:**

**HttpClient httpClient = HttpClient.create()**

**.option(ChannelOption.CONNECT\_TIMEOUT\_MILLIS, 5000)**

**.responseTimeout(Duration.ofMillis(5000))**

**.doOnConnected(conn -> conn.addHandlerLast(**

**new ReadTimeoutHandler(5000, TimeUnit.MILLISECONDS))**

**.addHandlerLast(new WriteTimeoutHandler(5000, TimeUnit.MILLISECONDS)));**

**WebClient client = WebClient.builder()**

**.clientConnector(new ReactorClientHttpConnector(httpClient))**

**.build();**

**NOTE we can call timeout on our client request as well,**

**this is a signal timeout, not an HTTP connection, a read/write, or a response timeout;**

**it's a timeout for the Mono/Flux publisher.**

1. **Define the Method and Request:**

**Example :**

**this.webClient .get()**

**.uri("/mockserver"+getAllRecordsUrl)**

**.retrieve()**

**Methods are : get() , post() , delete() etc.**

**Preparing the request :**

**RequestBodySpec bodySpec = uriSpec.uri("/resource");**

**Using a *UriBuilder Function*:**

**RequestBodySpec bodySpec = uriSpec.uri( uriBuilder -> uriBuilder.pathSegment("/resource").build());**

**Or**

**this.webClient .get()**

**.uri(uriBuilder -> uriBuilder.pathSegment("/mockserver/v1/records/dealer/{dealerid}")**

**.build( ));**

1. **Define the Body :**

**By presenting a *Publisher* (and the type of elements that will be published) to the *body*method:**

**this.webClient.patch().uri(urlPath)**

**.header(key,value)**

**.contentType(MediaType.APPLICATION\_JSON)**

**.accept(MediaType.APPLICATION\_JSON)**

**.body(Mono.just(new Foo("name")), Foo.class)**

**.retrieve().bodyToMono(Foo.class));**

**BodyInserters.fromPublisher(Mono.just("data")), String.class);**

**This class also offers other intuitive functions to cover more advanced scenarios. For instance, in case we have to send multipart requests:**

**LinkedMultiValueMap map = new LinkedMultiValueMap();  
 map.add("key1", "value1");  
 map.add("key2", "value2");  
 RequestHeadersSpec<?> headersSpec = bodySpec.body(  
 BodyInserters.fromMultipartData(map));**

**All these methods create a *BodyInserter* instance that we can then present as the *body* of the request.**

**The *BodyInserter* is an interface responsible for populating a *ReactiveHttpOutputMessage* body with a given output message and a context used during the insertion.**

**A *Publisher* is a reactive component in charge of providing a potentially unbounded number of sequenced elements. It is an interface too, and the most popular implementations are *Mono* and *Flux.***

1. **Retrieve Response :**

**.retrieve()**

**.onStatus(HttpStatus::is4xxClientError, response -> Mono.error(new BadRequestException("Error in Get Records")))**

**.bodyToMono(DealerResponse.class)**

***bodyToMono* method, which will throw a *WebClientException* if the status code is *4xx* (client error) or *5xx* (server error).**

**Using the exchange() method to retrieve the response**

**Note : The retrieve() method is the simplest way to get the response body. However, If you want to have more control over the response, then you can use the exchange() method which has access to the entire ClientResponse including all the headers and the body.**

**public Flux<GithubRepo> listGithubRepositories(String username, String token) {  
 return webClient.get()  
 .uri("/user/repos")  
 .header("Authorization", "Basic " + Base64Utils  
 .encodeToString((username + ":" + token).getBytes(UTF\_8)))  
 .exchange()  
 .flatMapMany(clientResponse -> clientResponse.bodyToFlux(GithubRepo.class));  
 }**

1. **WebClient Filters :**

**A filter can intercept, examine, and modify a client request (or response).**

**The filter function has two parameters: the *ClientRequest* to modify and the next *ExchangeFilterFunction*.**

**By using the *WebClient.builder()* we're able to add filters:**

**WebClient webClient = WebClient.builder()  
 .filter(filterFunction)  
 .build();**

**A Custom Filter**

**The filter examines the request method and increases a “global” counter in case of a GET request:**

**ExchangeFilterFunction countingFunction = (clientRequest, nextFilter) -> {  
 HttpMethod httpMethod = clientRequest.method();  
 if (httpMethod == HttpMethod.GET) {  
 getCounter.incrementAndGet();  
 }  
 return nextFilter.exchange(clientRequest);  
 };**

**The second filter we'll define appends a version number to the request URL path.**

**We utilize the *ClientRequest.from()* method to create a new request object from the current one and set the modified URL.**

**Subsequently, we continue executing the filter chain with the new modified request object:**

**ExchangeFilterFunction urlModifyingFilter = (clientRequest, nextFilter) -> {  
 String oldUrl = clientRequest.url().toString();  
 URI newUrl = URI.create(oldUrl + "/" + version);  
 ClientRequest filteredRequest = ClientRequest.from(clientRequest)  
 .url(newUrl)  
 .build();  
 return nextFilter.exchange(filteredRequest);  
 };**

**Next, let's define a filter to log the methods of sent requests along with their URLs. These details are available in the request object.**

**All we have to do then is to print to some output stream**

**ExchangeFilterFunction loggingFilter = (clientRequest, nextFilter) -> {  
 printStream.print("Sending request " + clientRequest.method() + " " + clientRequest.url());  
 return nextFilter.exchange(clientRequest);  
};**

**private ExchangeFilterFunction logResposneStatus() {  
 return ExchangeFilterFunction.ofResponseProcessor(clientResponse -> {  
 logger.info("Response Status {}", clientResponse.statusCode());  
 return Mono.just(clientResponse);  
 });  
}**

1. **Simultaneous Spring WebClient Calls**

**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.**

**In this quick tutorial, we’ll take a look at several approaches to see how we can accomplish this by making parallel service calls using the webclient.**

**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(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***](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)**and** [***Mono***](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html)**in the examples as well.**

|  |  |
| --- | --- |
| C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image001.png  MONO | C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image002.png  FLUX |

**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<User> fetchUsers(List<Integer> userIds) {  
 return Flux.fromIterable(userIds)  
 .parallel()  
 .runOn(Schedulers.elastic())  
 .flatMap(this::getUser)  
 .ordered((u1, u2) -> u2.id() - u1.id());  
}**

**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.**
  + **We then call the *parallel* method which creates a *ParallelFlux –* this indicates the simultaneous character of the execution**
  + **In this example, we've decided to use the *elastic*** [**scheduler**](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#elastic--) **to run the call on, but feel free to choose another configuration**
  + **Next, we invoke *flatMap*to run the *getUser* method we created previously, which returns *ParallelFlux***
  + **Then, we need to specify how to convert *ParallelFlux* to simple *Flux*. We'll use *ordered* method with own comparator**

**It's worth noting, that since operations are happening in parallel, we don't know the resulting order, hence the API provides the *ordered* method.**

**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<User> fetchUserAndOtherUser(int id) {  
 return Flux.merge(getUser(id), getOtherUser(id))  
 .parallel()  
 .runOn(Schedulers.elastic())  
 .ordered((u1, u2) -> u2.id() - u1.id());  
}**

**NOTE :**

**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(int userId, int itemId) {  
 Mono<User> user = getUser(userId).subscribeOn(Schedulers.elastic());  
 Mono<Item> item = getItem(itemId).subscribeOn(Schedulers.elastic());  
  
 return Mono.zip(user, item, UserWithItem::new);  
}**

**Note :**

**However, the *subscribeOn* method does not subscribe to the *Mono*.**

**It specifies what kind of *Scheduler* to use when the subscribe call happens. Again we're using the elastic scheduler in this example which ensures each subscription happens on a dedicated single thread.**

**The last step is to call the *zip* method which 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.**

1. **Test Cases :**

**Using *MockWebServer* is** [**recommended by the Spring Team**](https://github.com/spring-projects/spring-framework/issues/19852#issuecomment-453452354) **for writing integration tests*.***

**public class EmployeeService {  
  
 public EmployeeService(String baseUrl) {  
 this.webClient = WebClient.create(baseUrl);  
 }**

**public Mono<Employee> getEmployeeById(Integer employeeId) {  
 return webClient  
 .get()  
 .uri("http://localhost:8080/employee/{id}", employeeId)  
 .retrieve()  
 .bodyToMono(Employee.class);  
 }  
}**

**4.1. *MockWebServer* Dependencies**

**To use *MockWebServer*, we need to add Maven dependencies for both** [**okhttp**](https://search.maven.org/search?q=g:com.squareup.okhttp3%20AND%20a:okhttp&core=gav) **and** [**mockwebserver**](https://search.maven.org/search?q=g:com.squareup.okhttp3%20AND%20a:mockwebserver&core=gav) **to our pom.xml:**

**<dependency>  
 <groupId>com.squareup.okhttp3</groupId>  
 <artifactId>okhttp</artifactId>  
 <version>4.0.1</version>  
 <scope>test</scope>  
</dependency>  
<dependency>  
 <groupId>com.squareup.okhttp3</groupId>  
 <artifactId>mockwebserver</artifactId>  
 <version>4.0.1</version>  
 <scope>test</scope>  
</dependency>**

**4.2. Adding *MockWebServer* to Our Test**

**Let's test our *EmployeeService* with *MockWebServer*:**

**public class EmployeeServiceMockWebServerTest {  
  
 public static MockWebServer mockBackEnd;  
  
 @BeforeAll  
 static void setUp() throws IOException {  
 mockBackEnd = new MockWebServer();  
 mockBackEnd.start();  
 }  
  
 @AfterAll  
 static void tearDown() throws IOException {  
 mockBackEnd.shutdown();  
 }  
}**

**In the above JUnit Test class, the *setUp* and *tearDown* method takes care of creating and shutting down the *MockWebServer.***

**The next step is to map the port of the actual REST service call to the *MockWebServer's* port.**

**@BeforeEach  
void initialize() {  
 String baseUrl = String.format("http://localhost:%s", mockBackEnd.getPort());  
 employeeService = new EmployeeService(baseUrl);  
}**

**Now it's time to create a stub so that the *MockWebServer* can respond to an *HttpRequest*.**

**4.3. Stubbing a Response**

**Let's use *MockWebServer's* handy *enqueue* method to queue a test response on the webserver:**

**@Test  
void getEmployeeById() throws Exception {  
 Employee mockEmployee = new Employee(100, "Adam", "Sandler", 32, Role.LEAD\_ENGINEER);  
 mockBackEnd.enqueue(new MockResponse()  
 .setBody(objectMapper.writeValueAsString(mockEmployee))  
 .addHeader("Content-Type", "application/json"));  
  
 Mono<Employee> employeeMono = employeeService.getEmployeeById(100);  
  
 StepVerifier.create(employeeMono)  
 .expectNextMatches(employee -> employee.getRole()  
 .equals(Role.LEAD\_ENGINEER))  
 .verifyComplete();  
}**

**When the actual API call is made from the *getEmployeeById(Integer employeeId)* method in our *EmployeeService*class, *MockWebServer* will respond with the queued stub.**

**4.4. Checking a Request**

**We may also want to make sure that the *MockWebServer* was sent the correct *HttpRequest*.**

***MockWebServer* has a handy method named *takeRequest* that returns an instance of *RecordedRequest*:**

**RecordedRequest recordedRequest = mockBackEnd.takeRequest();  
   
assertEquals("GET", recordedRequest.getMethod());  
assertEquals("/employee/100", recordedRequest.getPath());**

***Flux<T>* is useful when we need to handle zero to many or potentially infinite results. We can think of a Twitter feed as an example.**

**When we know that the results are returned all at once – as in our use case – we can use *Mono<T>***

1. **Uploading a File from a Resource**

**To start with, we want to declare our URL:**

**URI url = UriComponentsBuilder.fromHttpUrl(EXTERNAL\_UPLOAD\_URL).build().toUri();**

**Let's say in this example we want to upload a PDF.**

**We'll use *MediaType.APPLICATION\_PDF* as our *ContentType*.**

**Our upload endpoint returns an *HttpStatus.*Since we're expecting only one result, we'll wrap it in a *Mono*:**

**Mono<HttpStatus> httpStatusMono = webClient.post()  
 .uri(url)  
 .contentType(MediaType.APPLICATION\_PDF)  
 .body(BodyInserters.fromResource(resource))  
 .exchangeToMono(response -> {  
 if (response.statusCode().equals(HttpStatus.OK)) {  
 return response.bodyToMono(HttpStatus.class).thenReturn(response.statusCode());  
 } else {  
 throw new ServiceException("Error uploading file");  
 }  
 });**

**The method consuming this method can also return a *Mono*, and we can continue until we actually need to access the result. Once we're ready, we can call the *block()* method on the *Mono* object.**

**The *fromResource()* method uses the *InputStream* of the passed resource to write to the output message.**

**Uploading a File from Multipart Resource**

**If our external upload endpoint takes a Multipart form data, we can use the *MultiPartBodyBuilder* to take care of the parts:**

**MultipartBodyBuilder builder = new MultipartBodyBuilder();  
builder.part("file", multipartFile.getResource());**

**Here, we could be adding various parts according to our requirements. The value in the map can be an *Object* or an *HttpEntity.***

**When we call *WebClient*, we use *BodyInsterter.fromMultipartData* and build the object:**

**.body(BodyInserters.fromMultipartData(builder.build()))**

**We update the content type to *MediaType.MULTIPART\_FORM\_DATA* to reflect the changes.**

**Let's look at the entire call:**

**Mono<HttpStatus> httpStatusMono = webClient.post()  
 .uri(url)  
 .contentType(MediaType.MULTIPART\_FORM\_DATA)  
 .body(BodyInserters.fromMultipartData(builder.build()))  
 .exchangeToMono(response -> {  
 if (response.statusCode().equals(HttpStatus.OK)) {  
 return response.bodyToMono(HttpStatus.class).thenReturn(response.statusCode());  
 } else {  
 throw new ServiceException("Error uploading file");  
 }  
 });**

1. **URI Path Component**

**A path component consists of a sequence of path segments separated by a slash ( / ).**

**First, let's start with a simple case when a URI doesn't have any variable segments */products*:**

**this.webClient.get()  
 .uri("/products")  
 .retrieve();  
verifyCalledUrl("/products");**

**For that case, we can just pass a *String* as an argument.**

**Next, let's take the */products/{id}* endpoint and build the corresponding URI:**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/{id}")  
 .build(2))  
 .retrieve();  
verifyCalledUrl("/products/2");**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/{id}/attributes/{attributeId}")  
 .build(2, 13))  
 .retrieve();  
verifyCalledUrl("/products/2/attributes/13");**

1. **URI Query Parameters**

**Single Value Parameters**

**Let's start with single value parameters and take the */products/?name={name}&deliveryDate={deliveryDate}&color={color}* endpoint.**

**To set a query parameter we call the *queryParam()* method of the *UriBuilder* interface:**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/")  
 .queryParam("name", "AndroidPhone")  
 .queryParam("color", "black")  
 .queryParam("deliveryDate", "13/04/2019")  
 .build())  
 .retrieve();  
verifyCalledUrl("/products/?name=AndroidPhone&color=black&deliveryDate=13/04/2019");**

**Here we've added three query parameters and assigned actual values immediately. Additionally, it's also possible to leave placeholders instead of exact values:**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/")  
 .queryParam("name", "{title}")  
 .queryParam("color", "{authorId}")  
 .queryParam("deliveryDate", "{date}")  
 .build("AndroidPhone", "black", "13/04/2019"))  
 .retrieve();  
verifyCalledUrl("/products/?name=AndroidPhone&color=black&deliveryDate=13%2F04%2F2019");**

**Especially, this might be helpful when passing a builder object further in a chain.**

**Please note one important difference between the two code snippets above.**

**With attention to the expected URIs, we can see that they were encoded differently. Particularly, the slash character ( / ) was escaped in the last example. Generally speaking,** [**RFC3986**](https://www.ietf.org/rfc/rfc3986.txt) **doesn't require encoding of slashes in the query.**

**Array Parameters**

**Likewise, we may need to pass an array of values. Still, there are no strict rules for passing arrays in a query string. Therefore, an array representation in a query string differs from project to project and usually depends on underlying frameworks. We'll cover the most widely used formats.**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/")  
 .queryParam("tag[]", "Snapdragon", "NFC")  
 .build())  
 .retrieve();**

**As we can see, the final URI contains multiple tag parameters followed by encoded square brackets.**

**The *queryParam()* method accepts variable arguments as values, so there is no need to call the method several times.**

**Alternatively, we can omit square brackets and just pass multiple query parameters with the same key, but different values – */products/?category={category1}&category={category2}*:**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/")  
 .queryParam("category", "Phones", "Tablets")  
 .build())  
 .retrieve();  
verifyCalledUrl("/products/?category=Phones&category=Tablets");**

**To conclude, there is one more extensively-used method to encode an array is to pass comma-separated values. Let's transform our previous example into comma-separated values:**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/")  
 .queryParam("category", String.join(",", "Phones", "Tablets"))  
 .build())  
 .retrieve();  
verifyCalledUrl("/products/?category=Phones,Tablets");**

**Thus, we are just using the *join()* method of the *String* class to create a comma-separated string.**

**Sure, we can use any other delimiter that is expected by the application.**

1. **Encoding Mode**

**Remember how we mentioned URL encoding earlier.**

**If the default behavior doesn't fit our requirements, we can change it. We need to provide a *UriBuilderFactory* implementation while building a *WebClient* instance.**

**In this case, we'll use the *DefaultUriBuilderFactory* class. To set encoding call the *setEncodingMode()* method.**

**The following modes are available:**

* + **TEMPLATE\_AND\_VALUES: Pre-encode the URI template and strictly encode URI variables when expanded**
  + **VALUES\_ONLY: Do not encode the URI template, but strictly encode URI variables after expanding them into the template**
  + **URI\_COMPONENTS: Encode URI component value after expending URI variables**
  + **NONE: No encoding will be applied**

**The default value is TEMPLATE\_AND\_VALUES. Let's set the mode to URI\_COMPONENTS:**

**DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(BASE\_URL);  
factory.setEncodingMode(DefaultUriBuilderFactory.EncodingMode.URI\_COMPONENT);  
this.webClient = WebClient  
 .builder()  
 .uriBuilderFactory(factory)  
 .baseUrl(BASE\_URL)  
 .exchangeFunction(exchangeFunction)  
 .build();**

**As a result, the following assertion will succeed:**

**this.webClient.get()  
 .uri(uriBuilder - > uriBuilder  
 .path("/products/")  
 .queryParam("name", "AndroidPhone")  
 .queryParam("color", "black")  
 .queryParam("deliveryDate", "13/04/2019")  
 .build())  
 .retrieve();  
verifyCalledUrl("/products/?name=AndroidPhone&color=black&deliveryDate=13/04/2019");**

**And, of course, we can provide a completely custom *UriBuilderFactory* implementation to handle URI creation manually.**

1. **Retry in Spring WebFlux**

**When we're building applications in a distributed cloud environment, we need to design for failure. This often involves retries.**

**Spring WebFlux offers us a few tools for retrying failed operations.**

**In this tutorial, we'll look at how to add and configure retries to our Spring WebFlux applications.**

**Use Case**

**For our example, we'll use** [***MockWebServer***](https://www.baeldung.com/spring-mocking-webclient#mockwebserver) **and simulate an external system being temporarily unavailable and then becoming available.**

**Let's create a simple test for a component connecting to this REST service:**

**@Test  
void givenExternalServiceReturnsError\_whenGettingData\_thenRetryAndReturnResponse() {  
  
 mockExternalService.enqueue(new MockResponse().setResponseCode(SERVICE\_UNAVAILABLE.code()));  
 mockExternalService.enqueue(new MockResponse().setResponseCode(SERVICE\_UNAVAILABLE.code()));  
 mockExternalService.enqueue(new MockResponse().setResponseCode(SERVICE\_UNAVAILABLE.code()));  
 mockExternalService.enqueue(new MockResponse().setBody("stock data"));  
  
 StepVerifier.create(externalConnector.getData("ABC"))  
 .expectNextMatches(response -> response.equals("stock data"))  
 .verifyComplete();  
  
 verifyNumberOfGetRequests(4);  
}**

**Adding Retries**

**There are two key retry operators built into the *Mono* and *Flux* APIs.**

**9.1. Using *retry***

**First, let's use the *retry* method, which prevents the application from immediately returning an error and re-subscribes a specified number of times:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .retrieve()  
 .bodyToMono(String.class)  
 .retry(3);  
}**

**This will retry up to three times, no matter what error comes back from the web client.**

**9.2. Using *retryWhen***

**Next, let's try a configurable strategy using the *retryWhen* method:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .retrieve()  
 .bodyToMono(String.class)  
 .retryWhen(Retry.max(3));  
}**

**This allows us to configure a** [***Retry***](https://projectreactor.io/docs/core/release/api/reactor/util/retry/Retry.html) **object to describe the desired logic.**

**Here, we've used the *max* strategy to retry up to a maximum number of attempts. This is equivalent to our first example but allows us more configuration options. In particular, we should note that in this case, each retry happens as quickly as possible.**

**9.3. Adding Delay**

**The main disadvantage of retrying without any delay is that this does not give the failing service time to recover.**

**It may overwhelm it, making the problem worse and reducing the chance of recovery.**

**9.3.1. Retrying with *fixedDelay***

**We can use the *fixedDelay* strategy to add a delay between each attempt:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .retrieve()  
 .bodyToMono(String.class)  
 .retryWhen(Retry.fixedDelay(3, Duration.ofSeconds(2)));  
}**

**This configuration allows a two-second delay between attempts, which may increase the chances of success.**

**However, if the server is experiencing a longer outage, then we should wait longer.**

**But, if we configure all delays to be a long time, short blips will slow our service down even more.**

**9.3.2. Retrying with *backoff***

**Instead of retrying at fixed intervals, we can use the *backoff* strategy:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .retrieve()  
 .bodyToMono(String.class)  
 .retryWhen(Retry.backoff(3, Duration.ofSeconds(2)));  
}**

**In effect, this adds a progressively increasing delay between attempts — roughly at 2, 4, and then 8-second intervals in our example.**

**This gives the external system a better chance to recover from commonplace connectivity issues or handle the backlog of work.**

**9.3.3. Retrying with *jitter***

**An additional benefit of the *backoff* strategy is that it adds randomness or jitter to the computed delay interval.**

**Consequently, jitter can help to reduce retry-storms where multiple clients retry in lockstep.**

**By default, this value is set to 0.5, which corresponds to a jitter of at most 50% of the computed delay.**

**Let's use the *jitter* method to configure a different value of 0.75 to represent jitter of at most 75% of the computed delay:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .accept(MediaType.APPLICATION\_JSON)  
 .retrieve()  
 .bodyToMono(String.class)  
 .retryWhen(Retry.backoff(3, Duration.ofSeconds(2)).jitter(0.75));  
}**

**We should note that the possible range of values is between 0 (no jitter) and 1 (jitter of at most 100% of the computed delay).**

**9.3.4. Filtering Errors**

**At this point, any errors from the service will lead to a retry attempt, including 4xx errors such as *400:Bad Request* or *401:Unauthorized*.**

**we should not retry on such client errors, as server response is not going to be any different.**

**Therefore, let's see how we can apply the retry strategy only in the case of specific errors.**

**First, let's create an exception to represent the server error:**

**public class ServiceException extends RuntimeException {  
   
 public ServiceException(String message, int statusCode) {  
 super(message);  
 this.statusCode = statusCode;  
 }  
}**

**Next, we'll create an error *Mono* with our exception for the 5xx errors and use the *filter* method to configure our strategy:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .retrieve()  
 .onStatus(HttpStatus::is5xxServerError,   
 response -> Mono.error(new ServiceException("Server error", response.rawStatusCode())))  
 .bodyToMono(String.class)  
 .retryWhen(Retry.backoff(3, Duration.ofSeconds(5))  
 .filter(throwable -> throwable instanceof ServiceException));  
}**

**Now we only retry when a *ServiceException* is thrown in the *WebClient* pipeline.**

**9.3.5. Handling Exhausted Retries**

**Finally, we can account for the possibility that all our retry attempts were unsuccessful.**

**In this case, the default behavior by the strategy is to propagate a *RetryExhaustedException*, wrapping the last error.**

**Instead, let's override this behavior by using the *onRetryExhaustedThrow* method and provide a generator for our *ServiceException*:**

**public Mono<String> getData(String stockId) {  
 return webClient.get()  
 .uri(PATH\_BY\_ID, stockId)  
 .retrieve()  
 .onStatus(HttpStatus::is5xxServerError, response -> Mono.error(new ServiceException("Server error", response.rawStatusCode())))  
 .bodyToMono(String.class)  
 .retryWhen(Retry.backoff(3, Duration.ofSeconds(5))  
 .filter(throwable -> throwable instanceof ServiceException)  
 .onRetryExhaustedThrow((retryBackoffSpec, retrySignal) -> {  
 throw new ServiceException("External Service failed to process after max retries", HttpStatus.SERVICE\_UNAVAILABLE.value());  
 }));  
}**

**Now the request will fail with our *ServiceException* at the end of a failed series of retries.**

**Difference between Spring Async and WebFlux**

**Spring Async supports Servlet 3.0 specifications, but Spring WebFlux supports Servlet 3.1+. It brings a number of differences:**

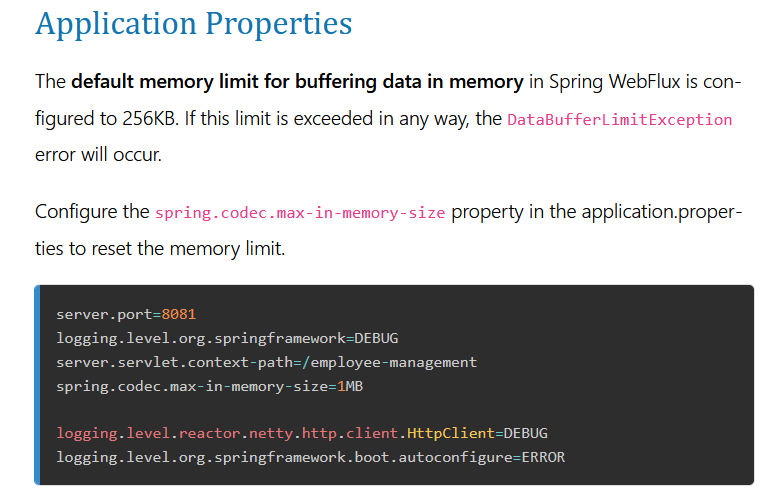
* + **Spring Async I/O model during its communication with the client is blocking. It may cause a performance problem with slow clients. On the other hand, Spring WebFlux provides a non-blocking I/O model.**
  + **Reading the request body or request parts is blocking in Spring Async, whiles it is non-blocking in Spring WebFlux.**
  + **In Spring Async, *Filter*s and *Servlet*s are working synchronously, but Spring WebFlux supports full asynchronous communication.**
  + **Spring WebFlux is compatible with wider ranges of Web/Application servers than Spring Async, like Netty, and Undertow.**

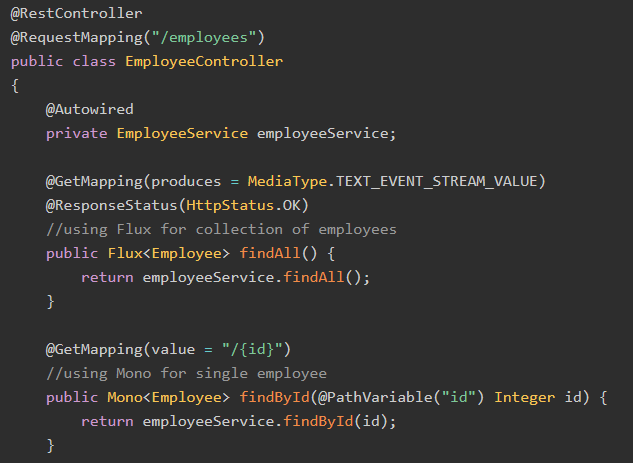
**Moreover, Spring WebFlux supports reactive backpressure, so we have more control over how we should react to fast producers than both Spring MVC Async and Spring MVC.**

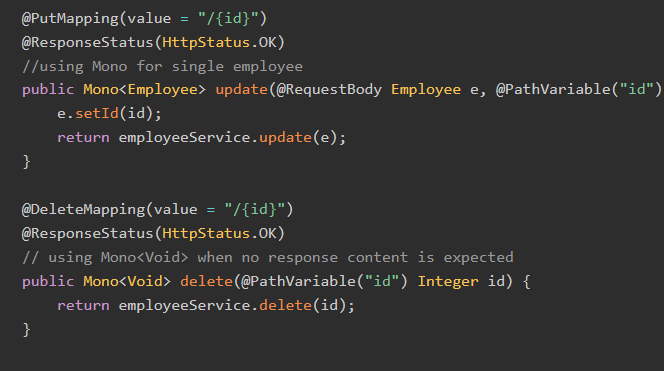
**Spring Flux also has a tangible shift towards functional coding style and declarative API decomposition thanks to Reactor API behind it.**

**Do all of these items lead us to use Spring WebFlux? Well, Spring Async or even Spring MVC might be the right answer to a lot of projects out there, depending on the desired load scalability or availability of the system.**

**Notes :**

****

****

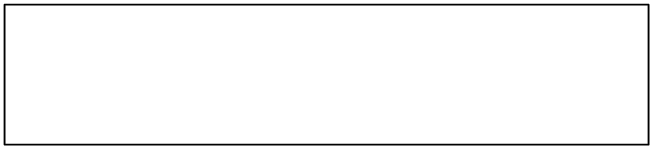
****

**Reference : Baeldung**

C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image006.png

C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image007.png





C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image014.png



C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image016.png

C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image017.png



C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image020.png





C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image023.png

















C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image034.png

C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image036.png











C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image042.png











































C:\B4629085\312E0ECE-3550-48F9-87A3-9A8C1BE8DCBC_files\image064.png