Links:

**https://projectreactor.io/docs/core/release/reference/**

Reactive Programming manages asynchronous data flows between producers of data and consumers that need to react to that data in a non-blocking manner. So, Reactive Programming is all about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale.

Reactive applications are difficult to build with thread-based frameworks, as there is high complexity involved in scaling out an application based on shared mutable state, threads, and locks.

In a Reactive Programming context, “Everything is a Stream and acts in a non-blocking manner when there is data in the stream.”

**Prerequisites**

Reactor Core runs on Java 8 and above.

**Specifications**

https://github.com/reactive-streams/reactive-streams-jvm

API Components

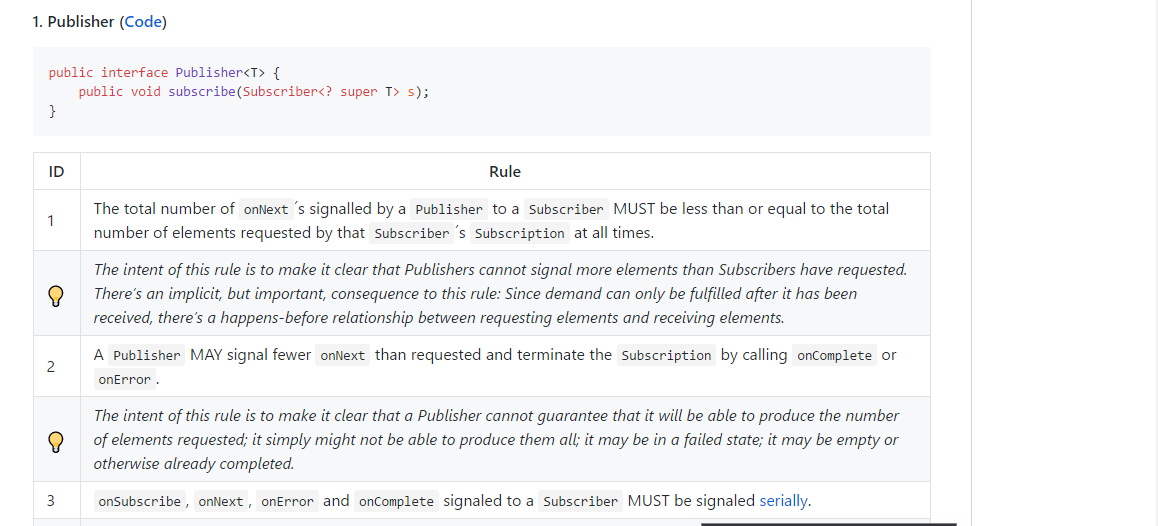
The API consists of the following components that are required to be provided by Reactive Stream implementations:

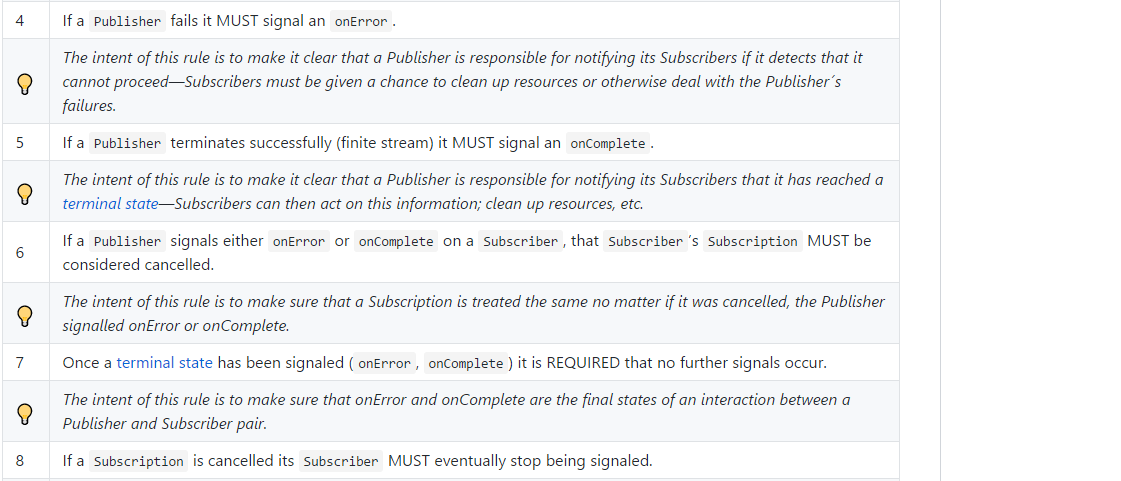
Publisher

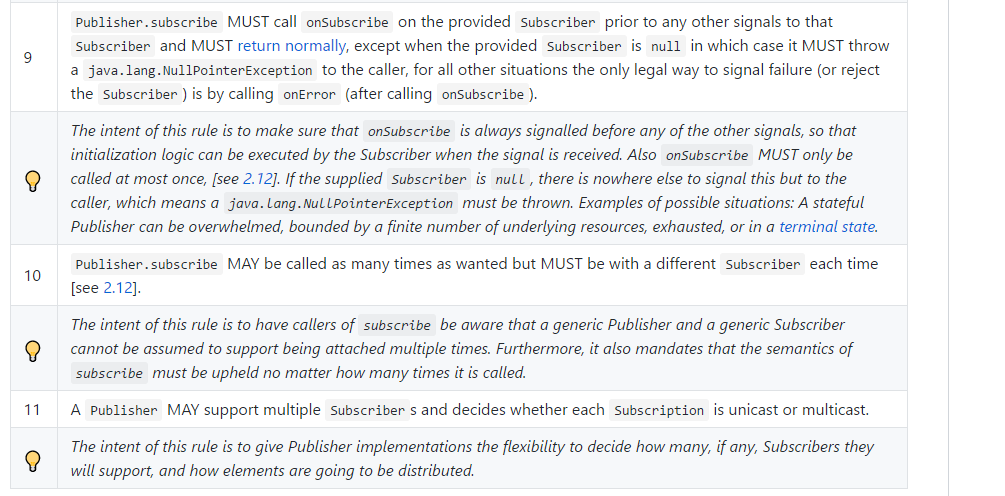
Subscriber

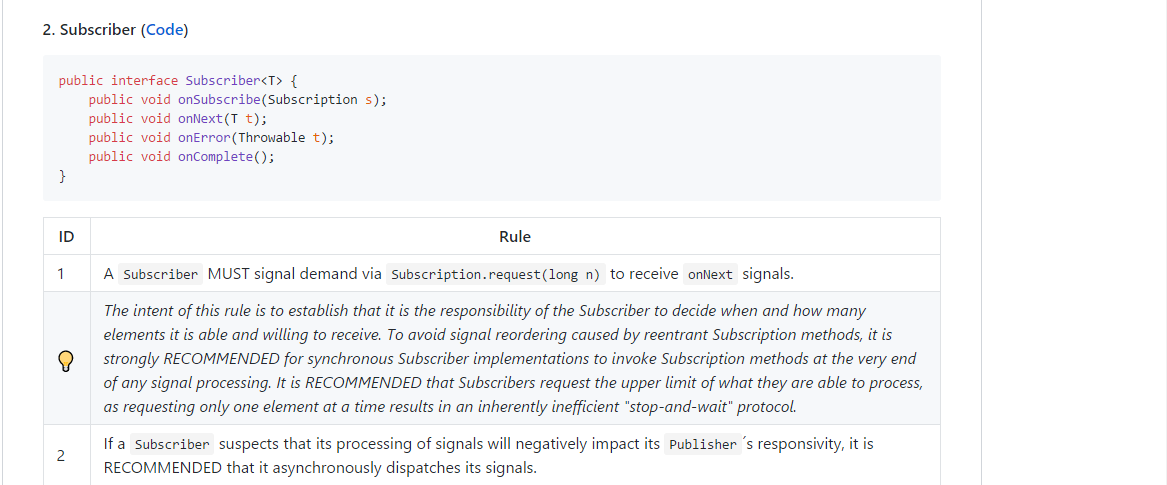
Subscription

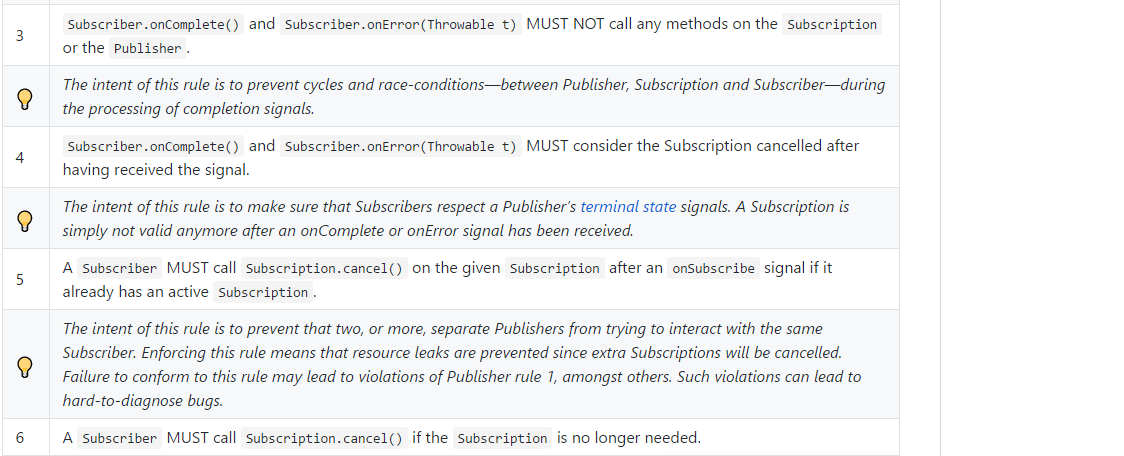
Processor

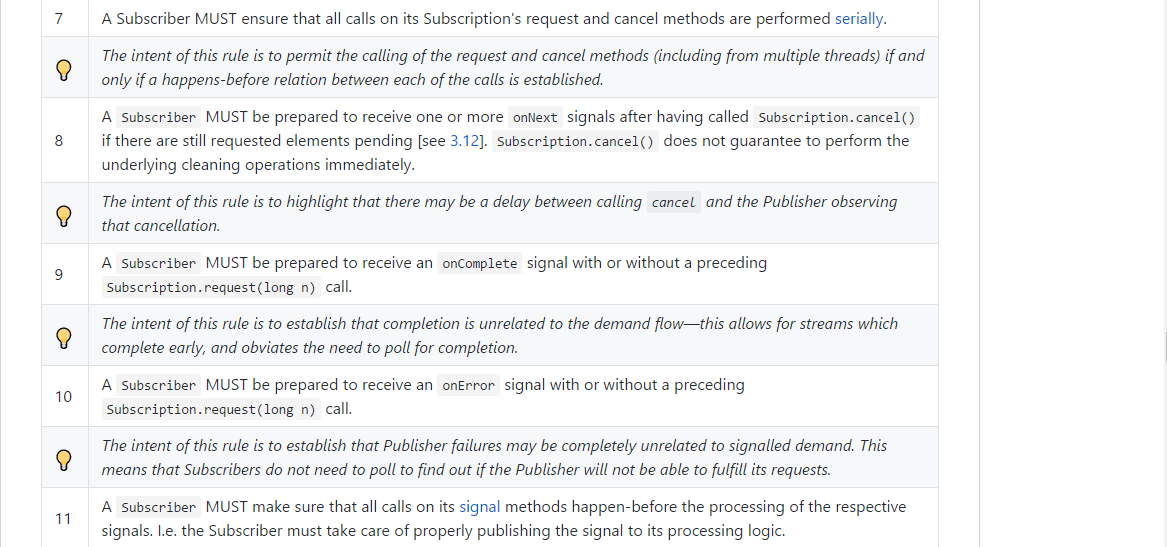


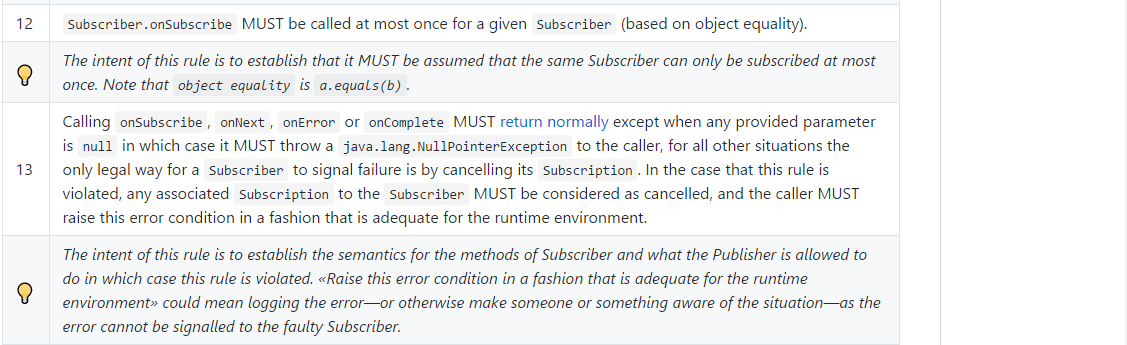


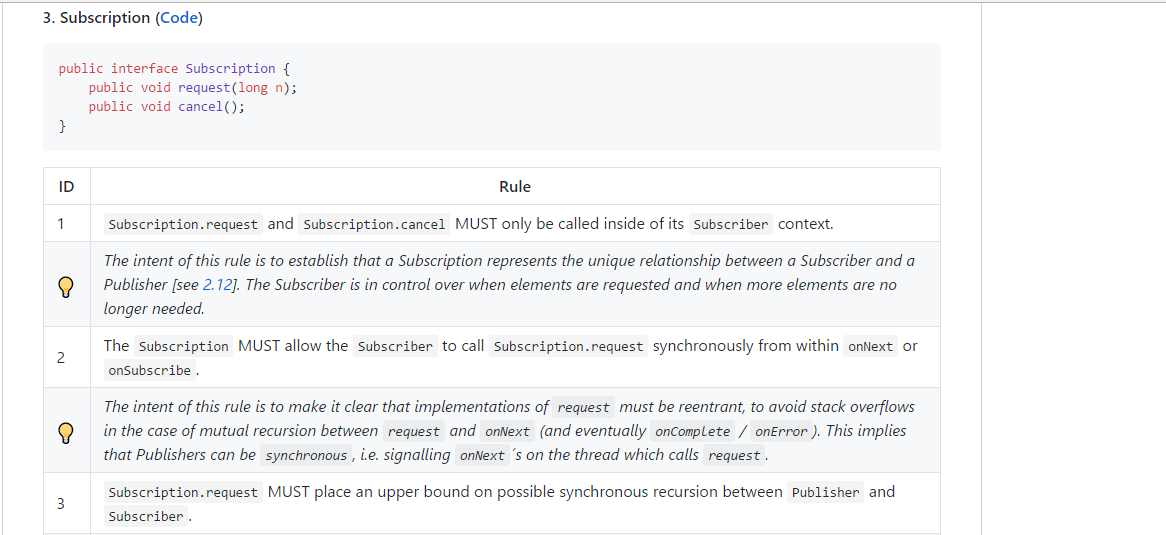


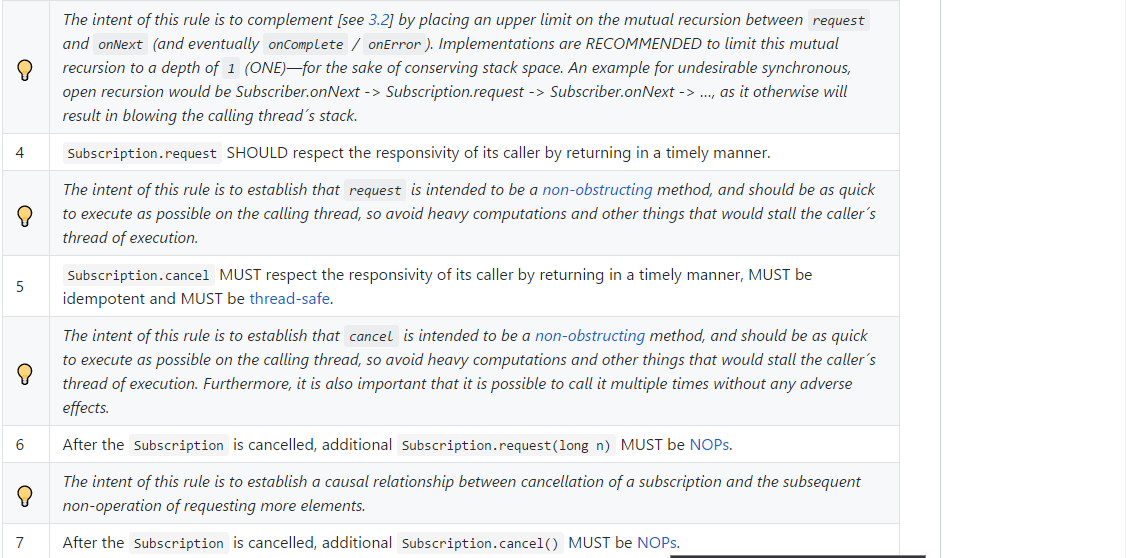


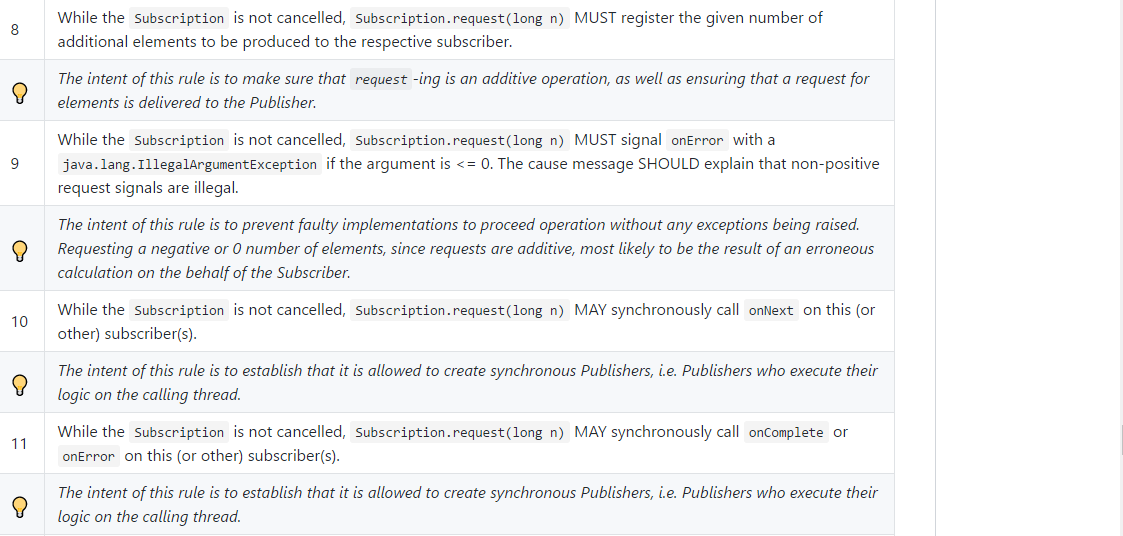


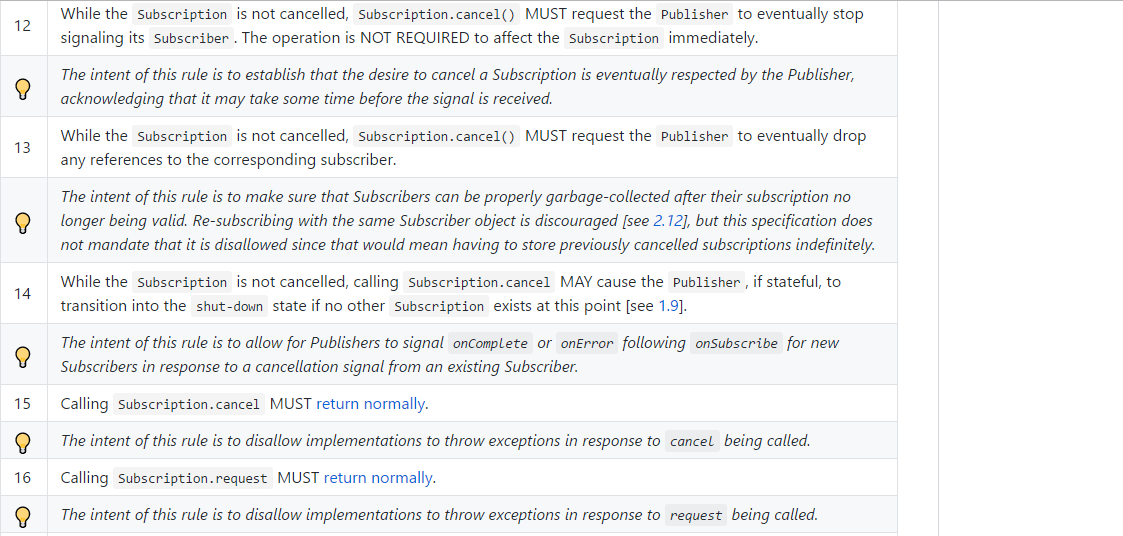


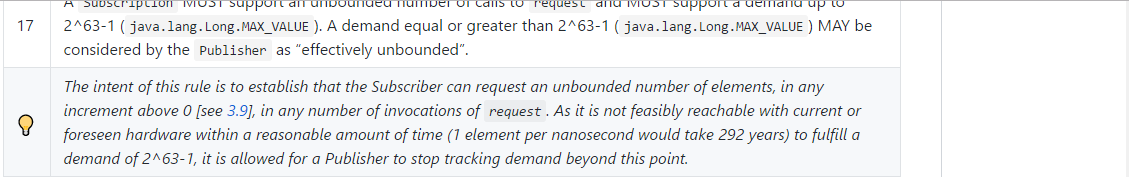


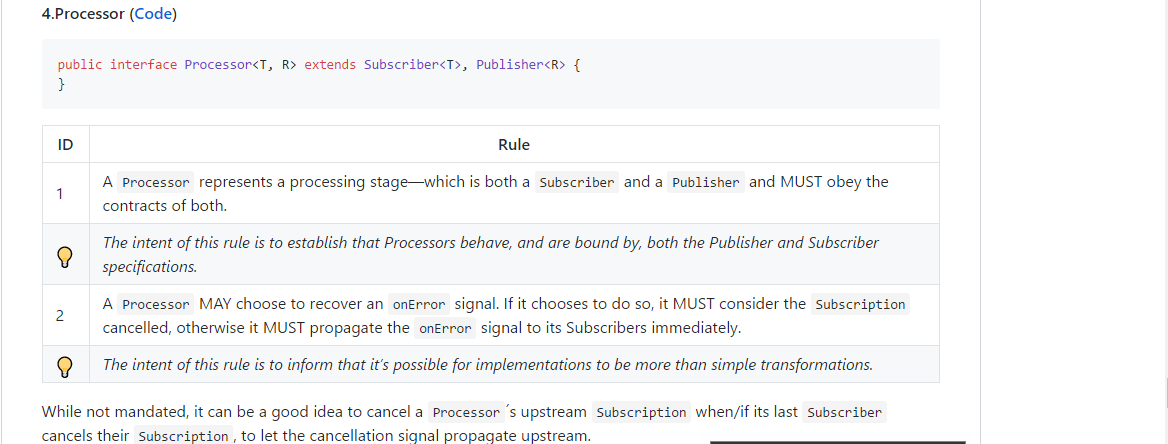




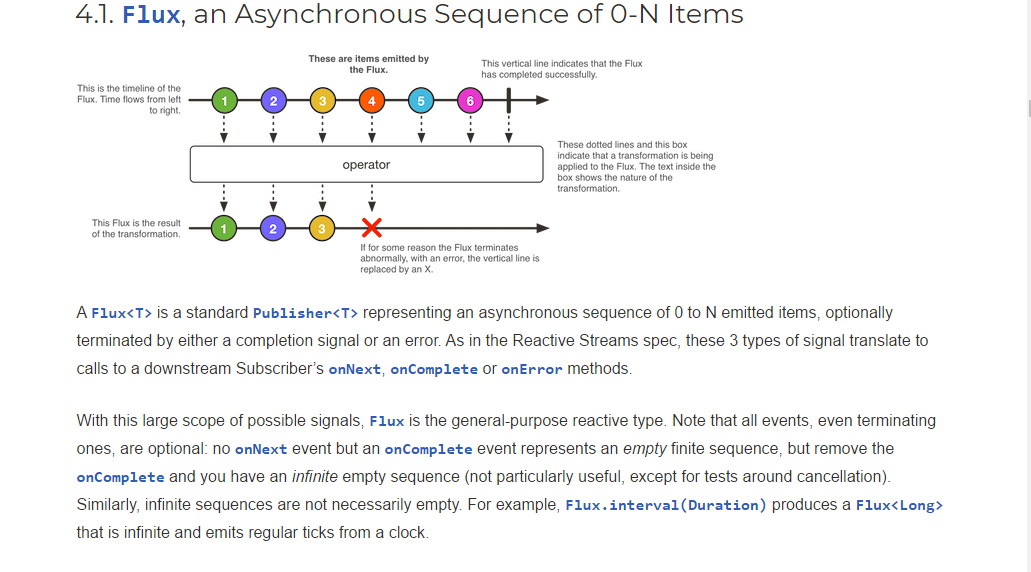








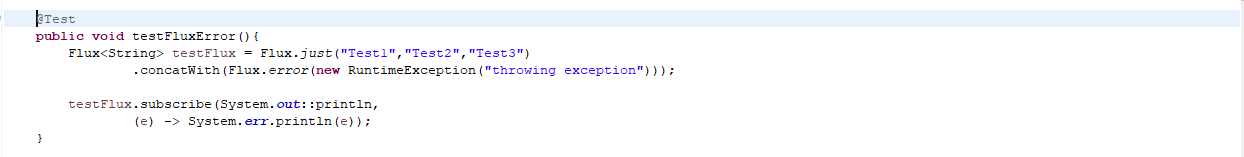
**Flux**



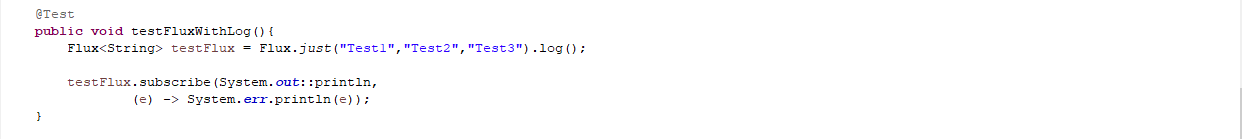
**Simple flux**



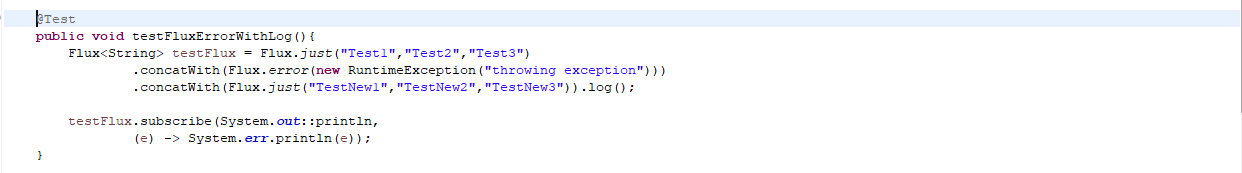
**Flux with error**



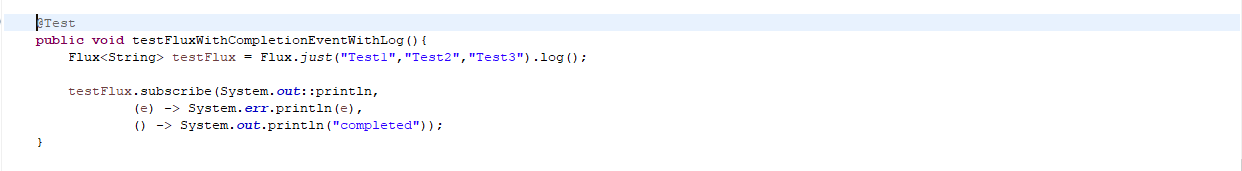
**Logging Flux**



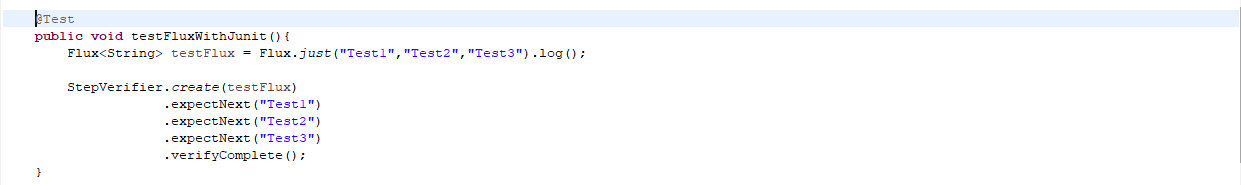
**Flux error with Log**

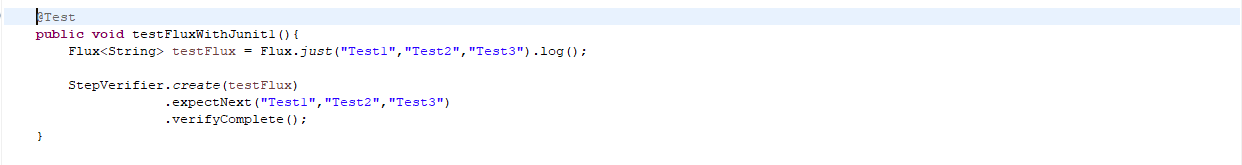


**Flux with completion event**

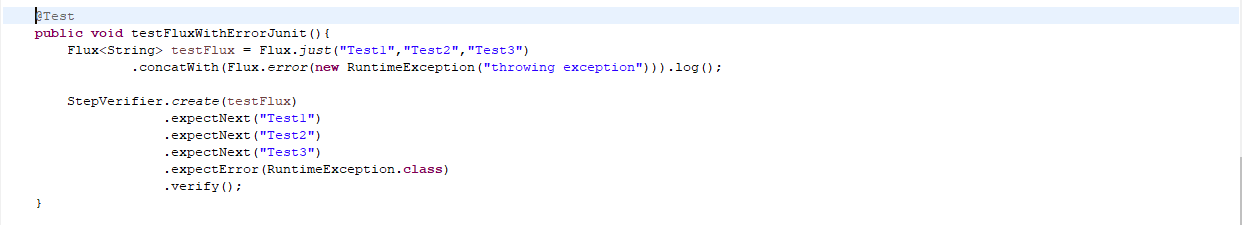


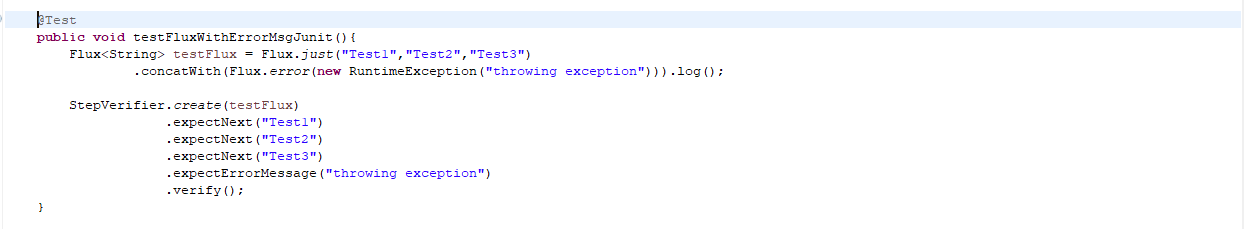
**Junit**

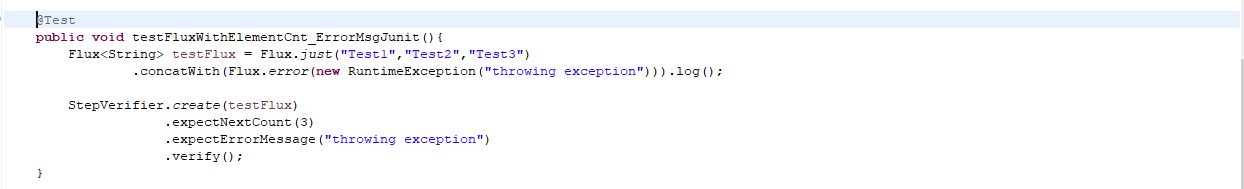




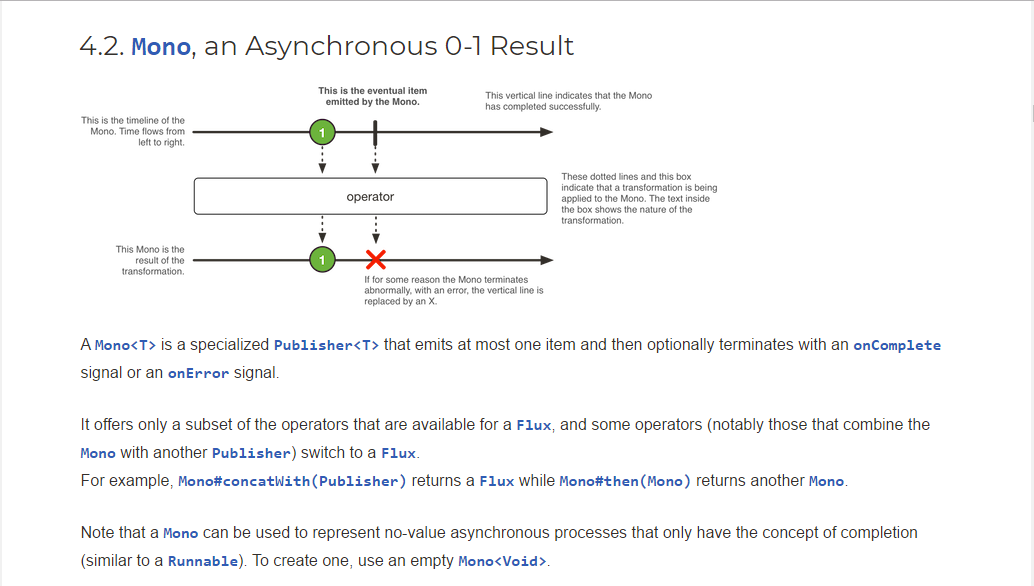
**Junit test with error**



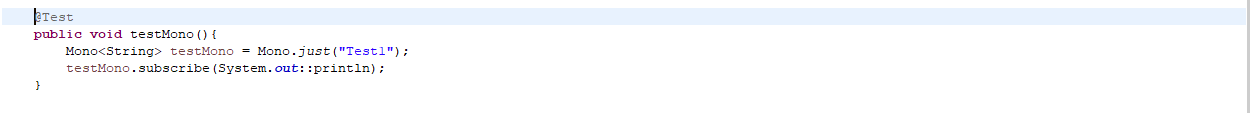




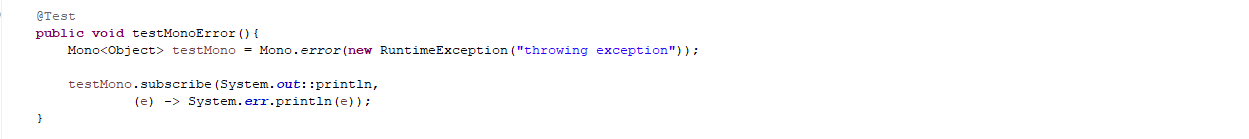
**Mono**



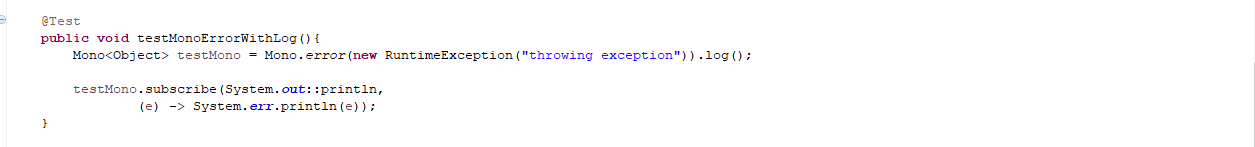
**Simple mono**



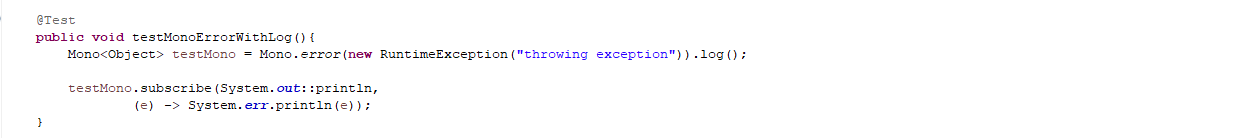
**Mono with error**



**Logging Mono**



**Mono error with Log**



**Mono with completion event**



**Junit**

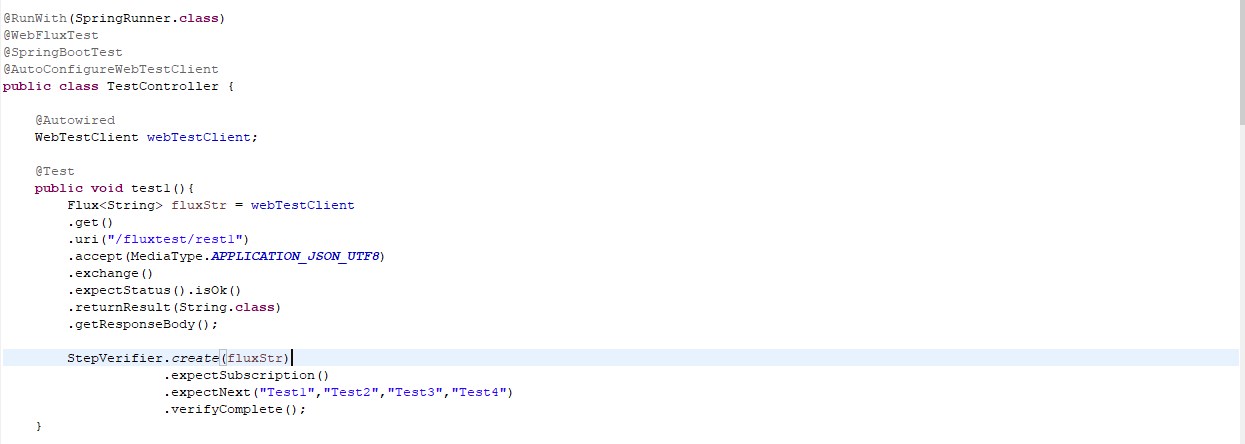


**Junit test with error**



**Junit with WebTestClient**

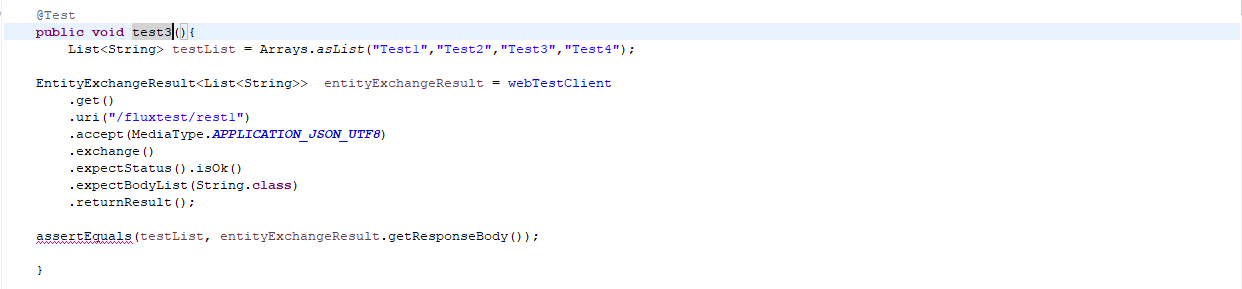
**Step Verifier**

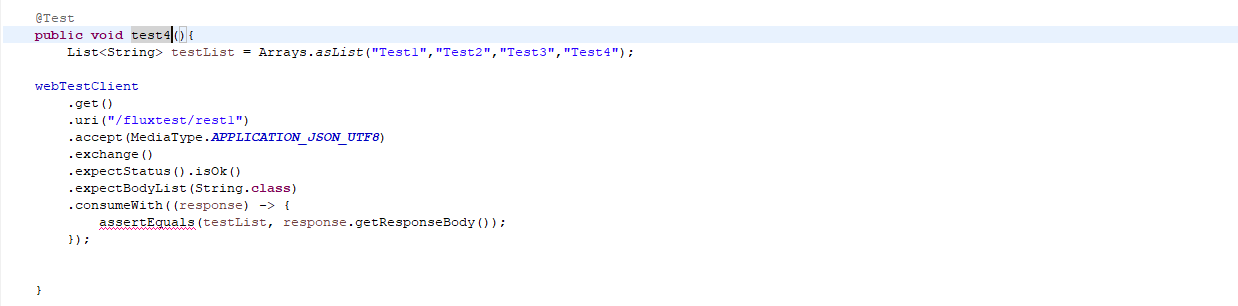


**hasSize**

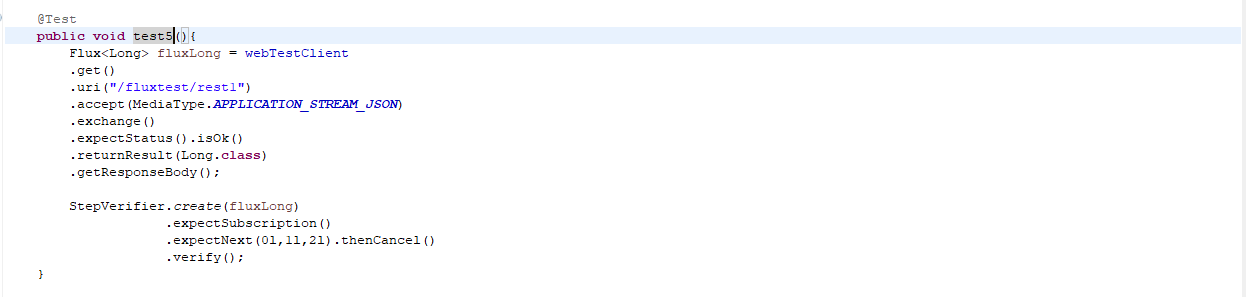


**Assert**





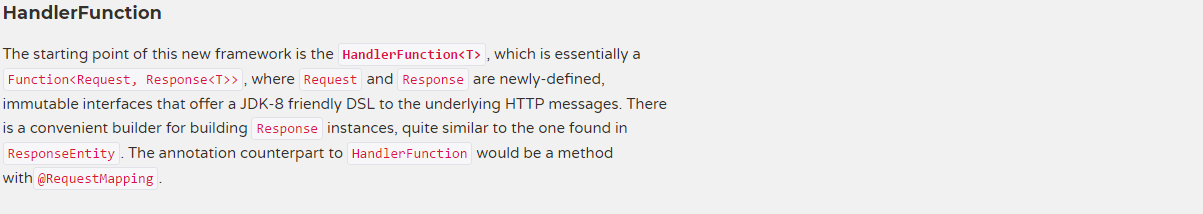
**Test Infinite loop**

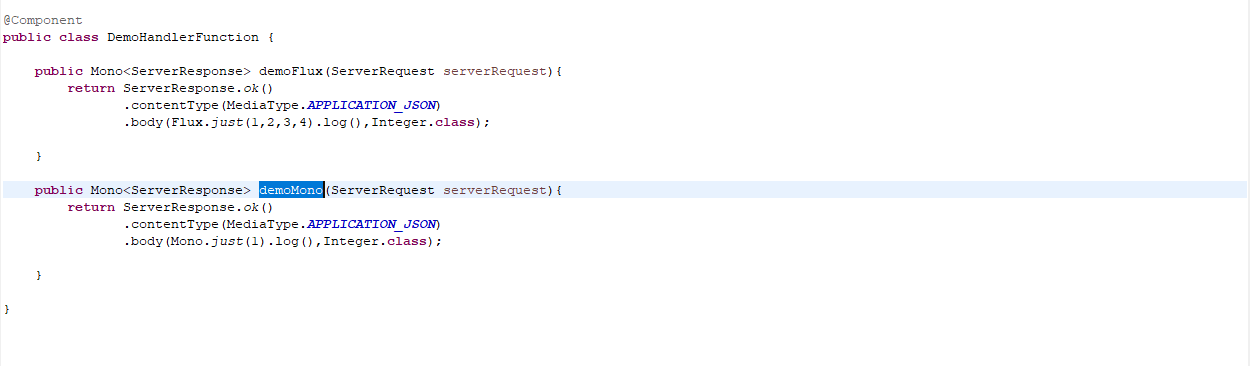


**Functional Web**

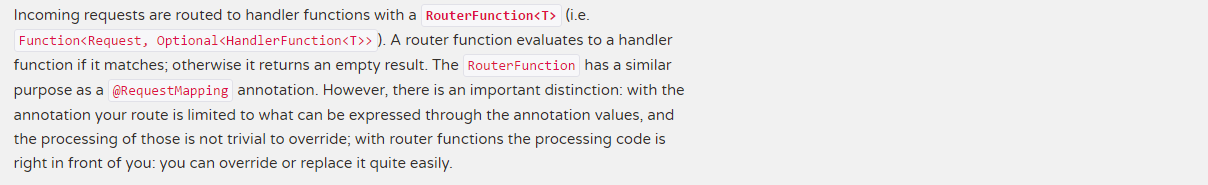
key components: HandlerFunction, RouterFunction, and FilterFunction

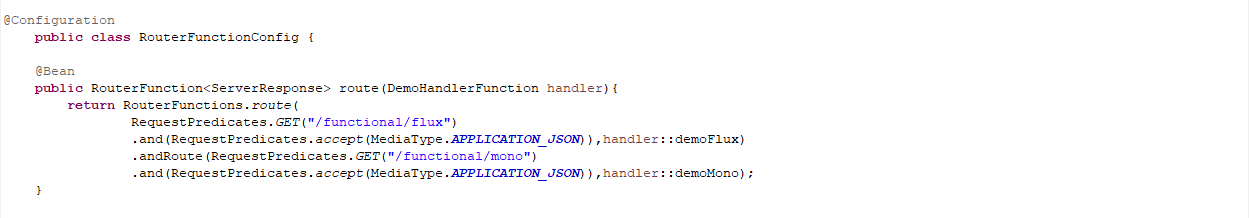
**HandlerFunction**

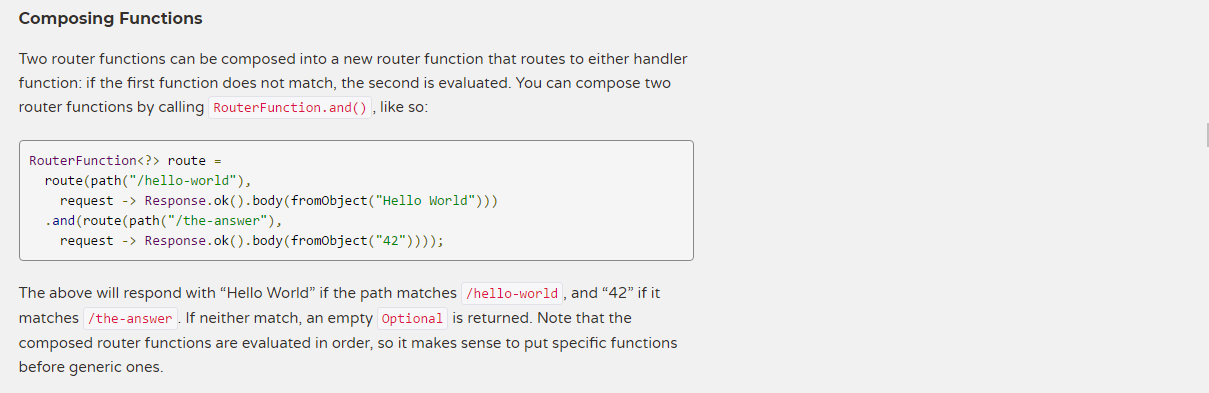


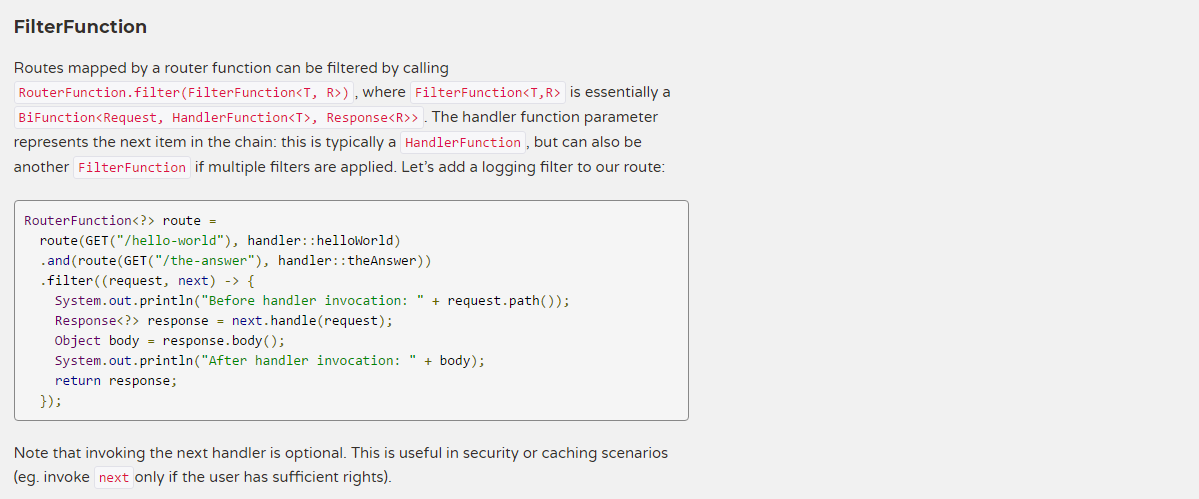


**RouterFunction**









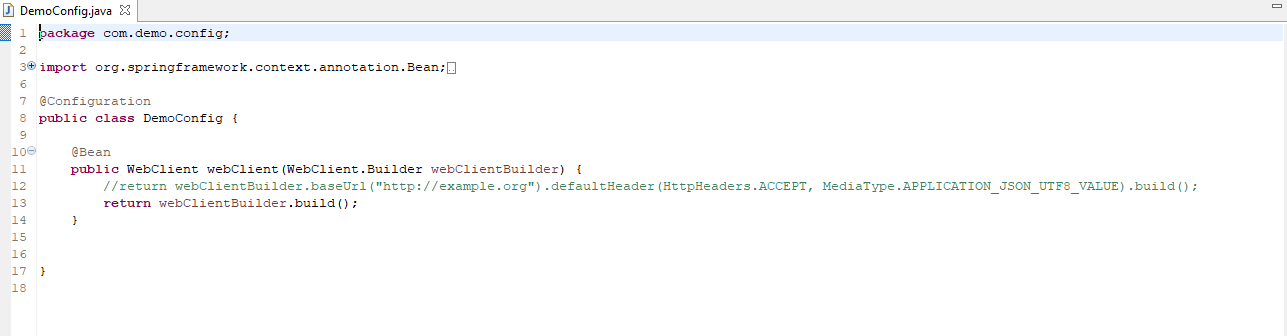
**WebClient**

Spring WebFlux includes a reactive, non-blocking WebClient for HTTP requests. The client has a functional, fluent API with reactive types for declarative composition, see Reactive Libraries. WebFlux client and server rely on the same non-blocking codecs to encode and decode request and response content.

**Configuration**

use WebClient.builder() with further options:

* uriBuilderFactory: Customized UriBuilderFactory to use as a base URL.
* defaultHeader: Headers for every request.
* defaultCookie: Cookies for every request.
* defaultRequest: Consumer to customize every request.
* filter: Client filter for every request.
* exchangeStrategies: HTTP message reader/writer customizations.
* clientConnector: HTTP client library settings.



**retrieve()/exchange()**

When you use exchange(), you must always use any of the body or toEntity methods of ClientResponse to ensure resources are released and to avoid potential issues with HTTP connection pooling. You can use bodyToMono(Void.class) if no response content is expected. However, if the response does have content, the connection is closed and is not placed back in the pool.



