Spring Web flux using non-blocking model to perfro msignificantly better

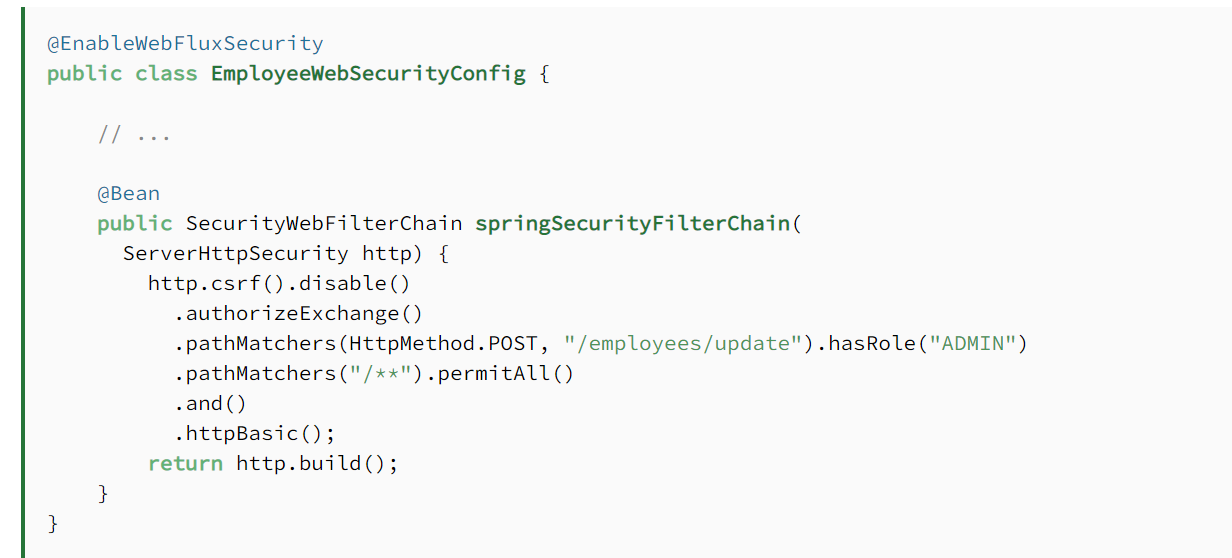
Webflux uses Netty as embedded server where as Spring MVC uses tomcat server .

Reacrtive programming support for web application

Webflux uses Project reactor and its publisher implementations Flux and Mono

New framework supporting 2 programming models

* Annotation based reactive components
* Functional routing and handling



Graphical user interface, text, application

Description automatically generated

**2. *doOnNext***

***Mono*‘s *doOnNext()* allows us to attach a listener that will be triggered when the data is emitted.**For the code examples in this article, we'll use the *PaymentService* class. In this case, we'll call the *processPayment* method only when the *paymentMono* emits the data, using *doOnNext()*:

@Test

**void** **givenAPaymentMono\_whenCallingServiceOnNext\_thenCallServiceWithPayment**() {

**Payment** paymentOf100 = **new** **Payment**(100);

Mono<Payment> paymentMono = Mono.just(paymentOf100);

paymentMono.doOnNext(paymentService::processPayment)

.block();

verify(paymentService).processPayment(paymentOf100);

}

**However, an empty *Mono* will not emit any data, and *doOnNext* will not be triggered.** Consequently, if we repeat the test using *Mono.empty()*, the *processPayment* method should no longer be called:

@Test

**void** **givenAnEmptyMono\_whenCallingServiceOnNext\_thenDoNotCallService**() {

Mono<Payment> emptyMono = Mono.empty();

emptyMono.doOnNext(paymentService::processPayment)

.block();

verify(paymentService, never()).processPayment(any());

}

**3. *doOnSuccess***

**We can use *doOnSuccess* to attach a listener that will be triggered when the *Mono* completes successfully.**Let's repeat the test, but using *doOnSuccess* this time:

@Test

**void** **givenAPaymentMono\_whenCallingServiceOnSuccess\_thenCallServiceWithPayment**() {

**Payment** paymentOf100 = **new** **Payment**(100);

Mono<Payment> paymentMono = Mono.just(paymentOf100);

paymentMono.doOnSuccess(paymentService::processPayment)

.block();

verify(paymentService).processPayment(paymentOf100);

}

**Though, we should note that a *Mono* is considered to be completed successfully even if no data is emitted.** As a result, for an empty *Mono*, the code above will call the *processPayment* method with a *null* *Payment*:

Test

**void** **givenAnEmptyMono\_whenCallingServiceOnSuccess\_thenCallServiceWithNull**() {

Mono<Payment> emptyMono = Mono.empty();

emptyMono.doOnSuccess(paymentService::processPayment)

.block();

verify(paymentService).processPayment(null);

}

**4. Conclusion**

In this short article, we learned the difference between a *Mono*‘s *doOnNext* and *doOnSuccess* listeners. We saw that we can use *doOnNext* if we want to react to the data received. On the other hand, we should use *doOnSuccess* if we want the method call to happen when the *Mono* completes successfully, regardless of whether it emits data or not.