|  |  |
| --- | --- |
| ДИСЦИПЛИНА | **Разработка клиент-серверных приложений** |
|  | (полное наименование дисциплины без сокращений) |
| ИНСТИТУТ | **информационных технологий** |
| КАФЕДРА | **Инструментального и прикладного программного обеспечения** |
|  | (полное наименование кафедры) |
| ВИД УЧЕБНОГО | **Материалы для практических/семинарских занятий** |
| МАТЕРИАЛА | (в соответствии с пп.1-11) |
| ПРЕПОДАВАТЕЛЬ | **Куликов Александр Анатольевич**  **Зарипов Евгений Андреевич** |
|  | (фамилия, имя, отчество) |
| СЕМЕСТР | **7, 2023-2024** |
|  | (указать семестр обучения, учебный год) |



# Практическая работа №7

**по дисциплине «Разработка клиент-серверных приложений» для студентов 4 курса. Форма обучения: Очная.**

Тема: Создание реактивного приложения реального времени с использованием Reactor

# Теоретическое введение

1. **Введение в Project Reactor**

Реактивное программирование поддерживается Spring Framework, начиная с версии 5. Эта поддержка построена на основе Project Reactor.

Project Reactor (или просто Reactor) - это библиотека Reactive для создания неблокирующих приложений на JVM, основанная на спецификации Reactive Streams. Reactor - это основа реактивного стека в экосистеме Spring, и он разрабатывается в тесном сотрудничестве со Spring. WebFlux, веб-фреймворк с реактивным стеком Spring, использует Reactor в качестве базовой зависимости.

# Модули Reactor

Проект Reactor состоит из набора модулей, перечисленных в документации Reactor. Модули встраиваемы и совместимы. Основным артефактом является Reactor Core, который содержит реактивные типы Flux и Mono, которые реализуют интерфейс Publisher Reactive Stream (подробности см. в **первом сообщении этой серии**) и набор операторов, которые могут применяться к ним.

Некоторые другие модули:

* Reactor Test - предоставляет некоторые утилиты для тестирования реактивных по- токов
* Reactor Extra - предоставляет некоторые дополнительные операторы Flux
* Reactor Netty - неблокирующие клиенты и серверы TCP, HTTP и UDP с поддержкой обратного давления - на основе инфраструктуры Netty
* Reactor Adapter - адаптер для других реактивных библиотек, таких как RxJava2 и Akka Streams
* Reactor Kafka - реактивный API для Kafka, который позволяет публиковать и полу- чать сообщения в Kafka.

# Настройка проекта

Прежде чем мы продолжим, если вы хотите настроить проект и запустить некоторые из приведенных ниже примеров кода, сгенерируйте новое приложение Spring Boot с помощью **Spring Initializr**. В качестве зависимости выберите Spring Reactive Web. После импорта проекта в вашу среду IDE взгляните на файл POM, и вы увидите, что добавлена зависимость spring-boot-starter-webflux, которая также внесет зависимость ядра-реактора. Также в качестве зависимости добавлен тест-реактор. Теперь вы готовы к запуску следующих примеров кода.

...

# <dependencies>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-

webflux</**artifactId**>

</**dependency**>

# <dependency>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

# <exclusions>

<**exclusion**>

<**groupId**>org.junit.vintage</**groupId**>

<**artifactId**>junit-vintage-

engine</**artifactId**>

</**exclusion**>

</**exclusions**>

</**dependency**>

# <dependency>

<**groupId**>io.projectreactor</**groupId**>

<**artifactId**>reactor-test</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>

# </dependencies>

...

# Возможности Reactor Core

Reactor Core определяет реактивные типы Flux и Mono.

# FLUX и MONO

Flux - это Publisher, который может испускать от 0 до N элементов, а Mono может испускать от 0 до 1 элемента. Оба они завершаются либо сигналом завершения, либо ошибкой, и они вызывают методы onNext, onComplete и onError нижестоящего подписчика. Помимо реализации функций, описанных в спецификации Reactive Streams, Flux и Mono предоставляют набор операторов для поддержки преобразований, фильтрации и обработки ошибок.

В качестве первого упражнения перейдите к классу тестирования, созданному в вашем новом проекте, добавьте следующий пример и запустите его:

@Test

# void simpleFluxExample() {

Flux<String> fluxColors = Flux.just("red", "green", "blue"); fluxColors.subscribe(System.out::println);

}

Метод just создает поток, который испускает предоставленные элементы, а затем завершается. Ничего не передается, пока кто-нибудь на это не подпишется. Чтобы подписаться на него, мы вызываем метод subscribe и в этом случае просто распечатываем отправленные элементы. Создание Mono также может быть выполнено с помощью метода just, с той лишь разницей, что разрешен только один параметр.

# Объединение операторов

Взгляните на Flux API, и вы увидите, что почти все методы возвращают Flux или Mono, что означает, что операторы могут быть связаны. Каждый

оператор добавляет поведение к Publisher (Flux или Mono) и переносит Publisher предыдущего шага в новый экземпляр. Данные поступают от первого издателя и перемещаются по цепочке, трансформируясь каждым оператором. В конце концов, подписчик завершает процесс. Обратите внимание, что ничего не происходит, пока подписчик не подпишется на издателя.

Существует оператор log(), который обеспечивает регистрацию всех сигналов Reactive Streams, происходящих за кулисами. Просто измените последнюю строку приведенного выше примера на

*fluxColors.log().subscribe(System.out::println);*

и перезапустите тест. Теперь вы увидите, что к выходным данным добавляется следующее:

2020-09-12 16:16:39.779 INFO 6252 --- [ main]

reactor.Flux.Array.1 : | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)

2020-09-12 16:16:39.781 INFO 6252 --- [ main]

reactor.Flux.Array.1 : | request(unbounded) 2020-09-12 16:16:39.781 INFO 6252 --- [ main]

reactor.Flux.Array.1 : | onNext(red) red

2020-09-12 16:16:39.781 INFO 6252 --- [ main]

reactor.Flux.Array.1 : | onNext(green) green

2020-09-12 16:16:39.781 INFO 6252 --- [ main]

reactor.Flux.Array.1 : | onNext(blue) blue

2020-09-12 16:16:39.782 INFO 6252 --- [ main]

reactor.Flux.Array.1 : | onComplete()

Теперь, чтобы увидеть, что произойдет, если вы исключите вызов subscribe(), снова измените последнюю строку кода на следующую и повторно запустите тест:

fluxColors.log();

Как вы увидите из выходных данных журнала, сейчас никакие элементы не отправляются - поскольку нет подписчика, инициирующего процесс.

# Поиск подходящего оператора

Reactor предоставляет длинный список операторов, и в качестве помощи в поиске подходящего оператора для конкретного варианта использования есть специальное приложение в справочной документации Reactor. Он разделен на различные категории, как показано в таблице ниже.

|  |  |
| --- | --- |
| **КАТЕГОРИЯ ОПЕРАТОРА** | **ПРИМЕРЫ** |
| Создание новой последовательности | just, fromArray, fromIterable, fromStream |
| Преобразование существующей последовательности | map, flatMap, startWith, concatWith |
| Заглядывать в  последовательность | doOnNext, doOnComplete, doOnError, doOnCancel |
| Фильтрация последовательности | filter, ignoreElements, distinct, elementAt, takeLast |
| Обработка ошибок | onErrorReturn, onErrorResume,  retry |
| Работаем со временем | elapsed, interval, timestamp, timeout |
| Расщепление потока | buffer, groupBy, window |
| Возвращаясь к синхронному  миру | block, blockFirst, blockLast, toIterable, toStream |
| Многоадресная рассылка потока нескольким подписчикам | publish, cache, replay |

Теперь не стесняйтесь создать несколько небольших примеров, в которых используются некоторые из этих операторов, и посмотреть, что произойдет, когда вы их запустите. Например, с помощью оператора map (который преобразует элементы, создаваемые путем применения синхронной функции к каждому элементу):

@Test

**void mapExample**() {

Flux<String> fluxColors = Flux.just("red", "green", "blue"); fluxColors.map(color ->

color.charAt(0)).subscribe(System.out::println);

}

Или оператор zip, который объединяет несколько источников вместе (ожидая, пока все источники испускают один элемент, и объединяет их в кортеж):

@Test

**void zipExample**() {

Flux<String> fluxFruits = Flux.just("apple", "pear", "plum"); Flux<String> fluxColors = Flux.just("red", "green", "blue"); Flux<Integer> fluxAmounts = Flux.just(10, 20, 30); Flux.zip(fluxFruits, fluxColors,

fluxAmounts).subscribe(System.out::println);

}

# Обработка ошибок

Как описано в предыдущем сообщении в блоге, в Reactive Streams ошибки

- это терминальные события. При возникновении ошибки вся последовательность останавливается, и ошибка передается методу onError подписчика, который всегда должен быть определен. Если не определено, onError вызовет исключение UnsupportedOperationException.

Как вы видите, запустив следующий пример, третье значение никогда не генерируется, поскольку второе значение приводит к ошибке:

@Test

# public void onErrorExample() {

Flux<String> fluxCalc = Flux.just(-1, 0, 1)

.map(i -> "10 / " + i + " = " + (10 / i));

value),

}

fluxCalc.subscribe(value -> System.out.println("Next: " + error -> System.err.println("Error: " + error));

Результат будет выглядеть так:

Next: 10 / -1 = -10

Error: java.lang.ArithmeticException: / by zero

Также можно обрабатывать ошибки в середине реактивной цепочки, используя операторы обработки ошибок:

Метод onErrorReturn будет выдавать резервное значение, когда наблюдается ошибка указанного типа. Это можно сравнить с перехватом исключения и возвратом статического запасного значения в императивном программировании. См. Пример ниже:

@Test

# public void onErrorReturnExample() {

Flux<String> fluxCalc = Flux.just(-1, 0, 1)

.map(i -> "10 / " + i + " = " + (10 / i))

.onErrorReturn(ArithmeticException.**class**, "**Division by**

0 **not allowed**");

value),

}

fluxCalc.subscribe(value -> System.out.println("Next: " + error -> System.err.println("Error: " + error));

и результат:

Next: 10 / -1 = -10

Next: Division by 0 not allowed

Как видите, использование оператора обработки ошибок таким образом все еще не позволяет продолжить исходную реактивную последовательность (третье значение здесь также не генерируется), а скорее заменяет ее. Если недостаточно просто вернуть какое-то значение по умолчанию, вы можете использовать этот onErrorResume метод, чтобы подписаться на резервного издателя при возникновении ошибки. Это можно сравнить с перехватом исключения и вызовом резервного метода в императивном программировании. Если, например, вызов внешней службы завершается неудачно, реализация onErrorResume может быть связана с извлечением данных из локального кеша.

# Тестирование

Модуль Reactor Test предоставляет служебные программы, которые могут помочь в тестировании поведения вашего Flux или Mono. В этом помогает API

StepVerifier. Вы создаете StepVerifier и передаете его издателю для тестирования. StepVerifier подписывается на Publisher при вызове метода verify, а затем сравнивает выданные значения с вашими определенными ожиданиями.

См. следующий пример:

@Test

# public void stepVerifierTest() {

Flux<String> fluxCalc = Flux.just(-1, 0, 1)

.map(i -> "10 / " + i + " = " + (10 / i));

StepVerifier.create(fluxCalc)

.expectNextCount(1)

.expectError(ArithmeticException.**class**)

.**verify**();

}

Для объекта создается StepVerifier, fluxCalc и определяются два ожидания: сначала ожидается, что будет выдана одна String, а затем должна быть выдана ошибка с типом ArithmeticException. С помощью вызова verify StepVerifier начинает подписываться на Flux, и инициируется поток.

StepVerifier также имеет другие функции, такие как включение утверждений после выполнения и поддержка виртуального времени, чтобы избежать длительного времени выполнения тестов, связанных с операторами, основанными на времени.

Модуль Reactor Test также предоставляет другой API, TestPublisher который представляет собой Publisher, которым вы можете напрямую управлять, инициируя события onNext, onComplete и onError для целей тестирования.

# Модель параллелизма

Как вы, возможно, уже заметили из вывода журнала simpleFluxExample, до сих пор наш издатель выполнялся в основном потоке так же, как подписчик. Это связано с тем, что Reactor не применяет модель параллелизма. Вместо этого выполнение большинства операторов будет продолжено в том же потоке,

оставляя выбор за разработчиком. Модель выполнения определяется тем Scheduler, что используется.

Есть два способа переключения контекста выполнения в реактивной

цепочке: publishOn и subscribeOn. Отличается следующее:

publishOn(Scheduler scheduler) влияет на выполнение всех последующих операторов (если не указано иное)

subscribeOn(Scheduler scheduler) изменяет поток, из которого подписывается вся цепочка операторов, на основе самого раннего вызова subscribeOn в цепочке. Это не влияет на поведение последующих вызовов publishOn

Класс Schedulers содержит статические методы, чтобы обеспечить контекст выполнения, например:

parallel() - Фиксированный пул воркеров, настроенный для параллельной работы, создавая столько воркеров, сколько ядер ЦП.

single() - Одиночная многоразовая нить. Этот метод повторно использует один и тот же поток для всех вызывающих, пока Планировщик не будет удален. Если вместо этого вам нужен выделенный поток для каждого вызова, вы можете использовать Schedulers.newSingle () для каждого вызова.

boundedElastic() - Динамически создает ограниченное количество рабочих. Он имеет ограничение на количество поддерживающих потоков, которые он может создать, и может ставить задачи в очередь для перепланирования, когда поток становится доступным. Это хороший выбор для обертывания синхронных, блокирующих вызовов.

immediate() - немедленно запускается в исполняемом потоке, не переключая контекст выполнения

fromExecutorService(ExecutorService) - может использоваться для создания Планировщика из любого существующего ExecutorService

Выполните следующий пример и посмотрите на поведение:

A");

@Test

# public void publishSubscribeExample() {

Scheduler schedulerA = Schedulers.newParallel("Scheduler

Scheduler schedulerB = Schedulers.newParallel("Scheduler B"); Scheduler schedulerC = Schedulers.newParallel("Scheduler C");

|  |  |  |  |
| --- | --- | --- | --- |
| Flux.just(1)  .map(i -> {  System.out.println("First | map: | " | + |
| Thread.currentThread().getName()); |  |  |  |
| **return** i;  })  .subscribeOn(schedulerA)  .map(i -> { |  |  |  |
| System.out.println("Second Thread.currentThread().getName()); | map: | " | + |
| **return** i;  })  .publishOn(schedulerB)  .map(i -> { |  |  |  |
| System.out.println("Third Thread.currentThread().getName()); | map: | " | + |
| **return** i;  })  .subscribeOn(schedulerC)  .map(i -> { |  |  |  |
| System.out.println("Fourth Thread.currentThread().getName()); | map: | " | + |
| **return** i;  })  .publishOn(schedulerA)  .map(i -> { |  |  |  |
| System.out.println("Fifth Thread.currentThread().getName()); | map: | " | + |
| **return** i;  })  .blockLast(); |  |  |  |

}

Взглянув на вывод (показан ниже), вы можете увидеть, что первая и вторая операции map выполняются в потоке из планировщика A, поскольку первый subscribeOn в цепочке переключается на этот планировщик, и это влияет на всю цепочку. Перед третьей операцией map выполняется publishOn, переключающий

контекст выполнения на Scheduler B, в результате чего третья и четвертая операции map выполняются в этом контексте (поскольку вторая subscribeOn не будет иметь никакого эффекта). И, наконец, есть новый метод publishOn, который переключает обратно на Планировщик A перед последней операцией map.

First map: Scheduler A-4 Second map: Scheduler A-4 Third map: Scheduler B-3 Fourth map: Scheduler B-3 Fifth map: Scheduler A-1

# BACKPRESSURE (противодавление)

Как вы могли вспомнить из первой части этой серии блогов, противодавление - это способность потребителя сигнализировать производителю, с какой скоростью выброса он может справиться, чтобы он не перегружался.

В приведенном ниже примере показано, как подписчик может контролировать скорость передачи, вызывая request(n) метод в Subscription.

@Test

**public void backpressureExample**() { Flux.range(1,5)

.subscribe(**new** Subscriber<Integer>() {

**private** Subscription s;

**int** counter;

@Override

**public void onSubscribe**(Subscription s) { System.out.println("onSubscribe"); **this**.s = s;

System.out.println("Requesting 2 emissions"); s.request(2);

}

@Override

**public void onNext**(Integer i) {

System.out.println("onNext " + i); counter++;

**if** (counter % 2 == 0) { System.out.println("Requesting 2 emissions"); s.request(2);

}

}

@Override

**public void onError**(Throwable t) { System.err.println("onError");

}

@Override

**public void onComplete**() { System.out.println("onComplete");

}

});

}

Запустите его, и вы увидите, что по запросу одновременно генерируются два значения:

onSubscribe

Requesting 2 emissions

onNext 1

onNext 2

Requesting 2 emissions

onNext 3

onNext 4

Requesting 2 emissions

onNext 5 onComplete

В Subscription также есть cancelметод, позволяющий запросить Издателя остановить эмиссию и очистить ресурсы.

# Холодные и горячие Publisher

Доступны два типа Publisher - cold и hot (холодные и горячие). Пока что мы сосредоточились на холодных Publisher. Как мы заявляли ранее, ничего не происходит, пока мы не подпишемся - но на самом деле это верно только для холодных издателей.

Холодный Publisher генерирует новые данные для каждой подписки. Если подписки нет, данные никогда не генерируются. Напротив, hot издатель не зависит от подписчиков. Он может начать публикацию данных без подписчиков. Если подписчик подписывается после того, как издатель начал передавать значения, он получит только значения, выпущенные после его подписки.

Publisher в Reactor по умолчанию не работают. Один из способов создания горячего Publisher - это вызвать publish() метод в Flux. Это вернет ConnectableFlux<T>, у которого есть метод connect() для запуска передачи значений. Подписчики должны затем подписаться на этот ConnectableFlux вместо исходного Flux.

Давайте посмотрим на простой холодный и горячий Publisher, чтобы увидеть различное поведение. В приведенном ниже примере coldPublisherExample оператор interval используется для создания потока, который генерирует значения long, начинающиеся с 0.

@Test

**public void coldPublisherExample**() **throws** InterruptedException { Flux<Long> intervalFlux =

Flux.interval(Duration.ofSeconds(1)); Thread.sleep(2000);

intervalFlux.subscribe(i ->

System.out.println(String.format("Subscriber A, value: %d", i))); Thread.sleep(2000);

intervalFlux.subscribe(i ->

System.out.println(String.format("Subscriber B, value: %d", i))); Thread.sleep(3000);

}

При запуске будет получен следующий результат:

Subscriber A, value: 0 Subscriber A, value: 1 Subscriber A, value: 2 Subscriber B, value: 0 Subscriber A, value: 3 Subscriber B, value: 1 Subscriber A, value: 4 Subscriber B, value: 2

Теперь вы можете задаться вопросом, почему что-то происходит, когда основной поток спит, но это потому, что оператор интервала по умолчанию выполняется в планировщике Schedulers.parallel(). Как видите, оба подписчика получат значения, начинающиеся с 0.

Теперь давайте посмотрим, что происходит, когда мы используем ConnectableFlux:

@Test

**public void hotPublisherExample**() **throws** InterruptedException { Flux<Long> intervalFlux =

Flux.interval(Duration.ofSeconds(1));

ConnectableFlux<Long> intervalCF = intervalFlux.publish(); intervalCF.connect();

Thread.sleep(2000);

intervalCF.subscribe(i ->

System.out.println(String.format("Subscriber A, value: %d", i))); Thread.sleep(2000);

intervalCF.subscribe(i ->

System.out.println(String.format("Subscriber B, value: %d", i))); Thread.sleep(3000);

}

На этот раз мы получаем следующий результат:

Subscriber A, value: 2 Subscriber A, value: 3 Subscriber A, value: 4 Subscriber B, value: 4 Subscriber A, value: 5 Subscriber B, value: 5 Subscriber A, value: 6 Subscriber B, value: 6

Как мы видим, на этот раз ни один из подписчиков не получает исходные

значения 0 и 1. Они получают значения, которые отправляются после подписки. Вместо того, чтобы вручную запускать публикацию, с помощью этого autoConnect(n)метода также можно настроить ConnectableFlux так, чтобы он запускался после n подписок.

# Прочие возможности

* 1. **Завершение синхронного, блокирующего вызова**

Когда необходимо использовать источник информации, который является синхронным и блокирующим, в Reactor рекомендуется использовать следующий шаблон:

Mono blockingWrapper = Mono.fromCallable(() -> {

**return** /\* make a remote synchronous call \*/

});

blockingWrapper =

blockingWrapper.subscribeOn(Schedulers.boundedElastic());

Метод fromCallable создает Mono, который производит его значение с помощью прилагаемого Callable. Используя Schedulers.boundedElastic(), мы гарантируем, что каждая подписка выполняется на выделенном однопоточном работнике, не влияя на другую неблокирующую обработку.

# Контекст

Иногда возникает необходимость передать некоторые дополнительные, обычно технические данные, через реактивный конвейер. Сравните это с привязкой некоторого состояния к потоку с помощью ThreadLocal в императивном мире.

Reactor имеет функцию, которая в некоторой степени сравнима с ThreadLocal, но может применяться к Flux или Mono вместо Thread, называемая a Context. Это интерфейс, похожий на Map, где вы можете хранить пары ключ- значение и получать значение по его ключу. Контекст прозрачно распространяется по всему реактивному конвейеру и может быть легко доступен в любой момент, вызвав метод Mono.subscriberContext().

Контекст может быть заполнен во время подписки путем добавления вызова

метода subscriberContext(Function) или subscriberContext(Context) ме тода в конце вашего реактивного конвейера, как показано в методе тестирования ниже..

# SINKS

Rector также предлагает возможность создавать Flux или Mono, программно определяя события onNext, onError и onComplete. Для этого предоставляется так называемый API-интерфейс приемника, запускающий события. Существуют несколько различных вариантов раковин, чтобы узнать больше об этом, читайте далее в справочной документации: **Программное создание последовательности**

# Отладка

Отладка реактивного кода может стать проблемой из-за его функционального декларативного стиля, в котором фактическое объявление (или

«assembly ») и обработка сигнала («execution») не происходят одновременно. Обычная трассировка стека Java, генерируемая приложением Reactor, не будет включать никаких ссылок на ассемблерный код, что затрудняет определение фактической основной причины распространенной ошибки.

Чтобы получить более значимую трассировку стека, которая включает информацию о сборке (также называемую трассировкой), вы можете добавить вызов Hooks.onOperatorDebug() в свое приложение. Однако это нельзя использовать в производственной среде, потому что это связано с перемещением тяжелого стека и может отрицательно повлиять на производительность.

Для использования в производственной среде Project Reactor предоставляет отдельный Java-агент, который инструментирует ваш код и добавляет отладочную информацию, не требуя больших ресурсов для захвата трассировки стека при каждом вызове оператора. Чтобы использовать его, вам нужно добавить reactor-tools артефакт в свои зависимости и инициализировать его при запуске приложения Spring Boot:

**public static void main**(String[] args) { ReactorDebugAgent.init(); SpringApplication.run(Application.**class**, **args**);

}

# Метрики

Reactor предоставляет встроенную поддержку для включения и отображения показателей как для планировщиков (Schedulers), так и для издателей (Publishers). Дополнительные сведения см. в разделе «Метрики» Справочного руководства.

# Подводя итог…

В этом сообщении в блоге представлен обзор Project Reactor, в основном сосредоточенный на функциях Reactor Core. **Следующий блог** в этой серии будет о WebFlux - реактивном фреймворке Spring, который использует Reactor в качестве реактивной библиотеки!

1. Введение в Spring WebFlux

Исходный веб-фреймворк для Spring - Spring Web MVC - был построен для Servlet API и контейнеров Servlet.

WebFlux был представлен как часть Spring Framework 5.0. В отличие от Spring MVC, он не требует Servlet API. Он полностью асинхронный и неблокирующий, реализует спецификацию Reactive Streams через проект Reactor. WebFlux требует Reactor в качестве основной зависимости, но он также может взаимодействовать с другими реактивными библиотеками через Reactive

Streams.

Spring WebFlux поддерживает две разные модели программирования: на основе аннотаций и функциональную.

# Аннотированные контроллеры

Если вы работали со Spring MVC, модель на основе аннотаций будет выглядеть довольно знакомой, поскольку в ней используются те же аннотации из веб-модуля Spring, что и в Spring MVC. Основное отличие состоит в том, что теперь методы возвращают реактивные типы Mono и Flux. Пример RestController с использованием модели на основе аннотаций:

@RestController @RequestMapping("/students") **public class StudentController** {

@Autowired

**private** StudentService studentService;

# public StudentController() {

}

id) {

@GetMapping("/{id}")

**public** Mono<ResponseEntity<Student>> getStudent(@PathVariable **long**

**return** studentService.findStudentById(id)

.map(ResponseEntity::ok)

.defaultIfEmpty(ResponseEntity.notFound().build());

}

@GetMapping

**public** Flux<Student> **listStudents**(@RequestParam(name = "name", required = **false**) String name) {

**return** studentService.findStudentsByName(name);

}

@PostMapping

**public** Mono<Student> **addNewStudent**(@RequestBody Student student)

{

**return** studentService.addNewStudent(student);

}

@PutMapping("/{id}")

**public** Mono<ResponseEntity<Student>> updateStudent(@PathVariable **long**

id, @RequestBody Student student) {

**return** studentService.updateStudent(id, student)

.map(ResponseEntity::ok)

.defaultIfEmpty(ResponseEntity.notFound().build());

}

id) {

@DeleteMapping("/{id}")

**public** Mono<ResponseEntity<Void>> deleteStudent(@PathVariable **long**

**return** studentService.findStudentById(id)

.flatMap(s ->

studentService.deleteStudent(s)

.then(Mono.just(**new**

ResponseEntity<Void>(HttpStatus.OK)))

)

.defaultIfEmpty(**new** ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

}

Некоторые пояснения к функциям, использованным в примере:

map функция используется для преобразования элемента, испускаемого Mono, применяя функцию синхронной к нему.

flatMap функция используется для преобразования элемент, испускаемый Mono асинхронно, возвращая значение, излучаемого другим Mono.

defaultIfEmpty функция обеспечивает значение по умолчанию, если Mono завершается без каких - либо данных.

# Функциональные конечные точки

Модель функционального программирования основана на лямбда- выражении и оставляет за приложением полную обработку запроса.

Он основан на концепциях HandlerFunctions и RouterFunctions. HandlerFunctions используются для генерации ответа на данный запрос: @FunctionalInterface

# public interface HandlerFunction<T extends ServerResponse> {

Mono<T> **handle**(ServerRequest request);

}

RouterFunction используется для маршрутизации запросов к HandlerFunctions:

@FunctionalInterface

**public interface RouterFunction**<**T extends ServerResponse**> { Mono<HandlerFunction<T>> route(ServerRequest request);

...

}

Продолжая с тем же примером ученика, мы получим что-то вроде следующего, используя функциональный стиль.

A StudentRouter:

@Configuration

# public class StudentRouter {

@Bean

**public** RouterFunction<ServerResponse> **route**(StudentHandler studentHandler){

**return** RouterFunctions

.route(

GET("/students/{id:[0-9]+}")

.and(accept(APPLICATION\_JSON)), studentHandler::getStudent)

.andRoute(

GET("/students")

.and(accept(APPLICATION\_JSON)), studentHandler::listStudents)

.andRoute(

tudent)

POST("/students")

.and(accept(APPLICATION\_JSON)),studentHandler::addNewS

.andRoute(

PUT("students/{id:[0-9]+}")

.and(accept(APPLICATION\_JSON)),

studentHandler::updateStudent)

.andRoute(

DELETE("/students/{id:[0-9]+}")

.and(accept(APPLICATION\_JSON)), studentHandler::deleteStudent);

}

}

И StudentHandler:

@Component

# public class StudentHandler {

**private** StudentService studentService;

**public StudentHandler**(StudentService studentService) {

**this**.studentService = studentService;

}

**public** Mono<ServerResponse> **getStudent**(ServerRequest serverRequest)

{

Mono<Student> studentMono = studentService.findStudentById( Long.parseLong(serverRequest.pathVariable("id")));

**return** studentMono.flatMap(student -> ServerResponse.ok()

.body(fromValue(student)))

.switchIfEmpty(ServerResponse.notFound().build());

}

**public** Mono<ServerResponse> **listStudents**(ServerRequest serverRequest)

{

String name = serverRequest.queryParam("name").orElse(**null**);

**return** ServerResponse.ok()

.contentType(MediaType.APPLICATION\_JSON)

.body(studentService.findStudentsByName(name),

Student.**class**);

}

**public** Mono<ServerResponse> **addNewStudent**(ServerRequest serverRequest) {

Mono<Student> studentMono = serverRequest.bodyToMono(Student.**class**);

**return** studentMono.flatMap(student -> ServerResponse.status(HttpStatus.OK)

.contentType(MediaType.APPLICATION\_JSON)

.body(studentService.addNewStudent(student),

Student.**class**));

}

**public** Mono<ServerResponse> **updateStudent**(ServerRequest serverRequest) {

**final long** studentId = Long.parseLong(serverRequest.pathVariable("id"));

Mono<Student> studentMono = serverRequest.bodyToMono(Student.**class**);

**return** studentMono.flatMap(student -> ServerResponse.status(HttpStatus.OK)

.contentType(MediaType.APPLICATION\_JSON)

.body(studentService.updateStudent(studentId,

student), Student.**class**));

}

**public** Mono<ServerResponse> **deleteStudent**(ServerRequest serverRequest) {

**final long** studentId = Long.parseLong(serverRequest.pathVariable("id"));

**return** studentService

.findStudentById(studentId)

.flatMap(s ->

ServerResponse.noContent().build(studentService.deleteStudent(s)))

.switchIfEmpty(ServerResponse.notFound().build());

}

}

Некоторые пояснения к функциям, использованным в примере:

switchIfEmpty функция имеет ту же цель, defaultIfEmpty, но вместо того, чтобы обеспечить значение по умолчанию, она используется для обеспечения альтернативного Mono.

Сравнивая две модели, мы видим, что:

Для использования функционального варианта требуется еще немного кода для таких вещей, как получение входных параметров и синтаксический анализ до ожидаемого типа.

Не полагаясь на аннотации, но написание явного кода предлагает некоторую большую гибкость и может быть лучшим выбором, если нам, например, нужно реализовать более сложную маршрутизацию.

* 1. Поддержка сервера

WebFlux работает в средах выполнения, отличных от сервлетов, таких как Netty и Undertow (неблокирующий режим), а также в средах выполнения сервлетов 3.1+, таких как Tomcat и Jetty.

По умолчанию стартер Spring Boot WebFlux использует Netty, но его легко переключить, изменив зависимости Maven или Gradle.

Например, чтобы переключиться на Tomcat, просто исключите spring-boot- starter-netty из зависимости spring-boot-starter-webflux и добавьте spring-boot- starter-tomcat:

# <dependency>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-webflux</**artifactId**>

# <exclusions>

<**exclusion**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-netty</**artifactId**>

</**exclusion**>

</**exclusions**>

</**dependency**>

# <dependency>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-tomcat</**artifactId**>

</**dependency**>

* 1. Конфигурация

Spring Boot обеспечивает автоматическую настройку Spring WebFlux, которая хорошо работает в общих случаях. Если вам нужен полный контроль над конфигурацией WebFlux, можно использовать аннотацию @EnableWebFlux (эта аннотация также потребуется в простом приложении Spring для импорта конфигурации Spring WebFlux).

Если вы хотите сохранить конфигурацию Spring Boot WebFlux и просто добавить дополнительную конфигурацию WebFlux, вы можете добавить свой собственный класс @Configuration типа WebFluxConfigurer (но без @EnableWebFlux).

1. Защита ваших конечных точек

Чтобы получить поддержку Spring Security WebFlux, сначала добавьте в свой проект зависимость spring-boot-starter-security. Теперь вы можете включить

его, добавив @EnableWebFluxSecurity аннотацию в свой класс Configuration (доступно с Spring Security 5.0).

В следующем упрощенном примере будет добавлена поддержка двух пользователей, один с ролью USER, а другой с ролью ADMIN, принудительно применить базовую аутентификацию HTTP и потребовать роль ADMIN для любого доступа к пути /student/admin:

@EnableWebFluxSecurity

# public class SecurityConfig {

@Bean

**public** MapReactiveUserDetailsService **userDetailsService**() {

UserDetails user = User

.withUsername("user")

.password(passwordEncoder().encode("userpwd"))

.roles("USER")

.build();

UserDetails admin = User

.withUsername("admin")

.password(passwordEncoder().encode("adminpwd"))

.roles("ADMIN")

.build();

**return new** MapReactiveUserDetailsService(user, admin);

}

@Bean

**public** SecurityWebFilterChain

**securityWebFilterChain**(ServerHttpSecurity http) {

**return** http.authorizeExchange()

.pathMatchers("/students/admin")

.hasAuthority("ROLE\_ADMIN")

.anyExchange()

.authenticated()

.and().httpBasic()

.and().build();

}

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return new** BCryptPasswordEncoder();

}

}

Также можно защитить метод, а не путь, сначала добавив аннотацию @EnableReactiveMethodSecurity к вашей конфигурации:

@EnableWebFluxSecurity @EnableReactiveMethodSecurity **public class SecurityConfig** {

...

}

А затем добавляем @PreAuthorize аннотацию к защищаемым методам. Например, мы можем захотеть, чтобы наши методы POST, PUT и DELETE были доступны только для роли ADMIN. Затем к этим методам можно применить аннотацию PreAuthorize, например:

@DeleteMapping("/{id}") @PreAuthorize("hasRole('ADMIN')")

**public** Mono<ResponseEntity<Void>> deleteStudent(@PathVariable **long** id) {

...

}

Spring Security предлагает дополнительную поддержку, связанную с приложениями WebFlux, например защиту CSRF, интеграцию OAuth2 и реактивную аутентификацию X.509. Для получения дополнительной информации прочтите следующий раздел в документации Spring Security: Реактивные приложения

1. Веб-клиент

Spring WebFlux также включает реактивный, полностью неблокирующий веб-клиент. У него есть функциональный, свободный API, основанный на Reactor.

Давайте рассмотрим (еще раз) упрощенный пример того, как WebClient можно использовать для запроса нашего StudentController:

# public class StudentWebClient {

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

"userpwd"))

**public** Mono<Student> **get**(**long** id) {

**return** client

.get()

.uri("/students/" + id)

.headers(headers -> headers.setBasicAuth("user",

.retrieve()

.bodyToMono(Student.**class**);

}

"userpwd"))

**public** Flux<Student> **getAll**() {

**return** client.get()

.uri("/students")

.headers(headers -> headers.setBasicAuth("user",

.retrieve()

.bodyToFlux(Student.**class**);

}

"userpwd"))

**public** Flux<Student> **findByName**(String name) {

**return** client.get()

.uri(uriBuilder -> uriBuilder.path("/students")

.queryParam("name", name)

.build())

.headers(headers -> headers.setBasicAuth("user",

.retrieve()

.bodyToFlux(Student.**class**);

}

"adminpwd"))

**public** Mono<Student> **create**(Student s) {

**return** client.post()

.uri("/students")

.headers(headers -> headers.setBasicAuth("admin",

.body(Mono.just(s), Student.**class**)

.**retrieve**()

# .bodyToMono(Student.class);

}

"adminpwd"))

**public** Mono<Student> **update**(Student student) {

**return** client

.put()

.uri("/students/" + student.getId())

.headers(headers -> headers.setBasicAuth("admin",

.body(Mono.just(student), Student.**class**)

.**retrieve**()

# .bodyToMono(Student.class);

}

**public** Mono<Void> **delete**(**long** id) {

**return** client

"adminpwd"))

}

}

.delete()

.uri("/students/" + id)

.headers(headers -> headers.setBasicAuth("admin",

.retrieve()

.bodyToMono(Void.**class**);

1. Тестирование

Для тестирования вашего реактивного веб-приложения WebFlux предлагает WebTestClient, который поставляется с API, аналогичным WebClient.

Давайте посмотрим, как мы можем протестировать наш StudentController с помощью WebTestClient:

@ExtendWith(SpringExtension.**class**)

# @SpringBootTest(webEnvironment =

SpringBootTest.WebEnvironment.RANDOM\_PORT)

# class StudentControllerTest {

@Autowired

WebTestClient webClient;

@Test

@WithMockUser(roles = "USER")

**void test\_getStudents**() { webClient.get().uri("/students")

.header(HttpHeaders.ACCEPT, "application/json")

.exchange()

.expectStatus().isOk()

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

.expectBodyList(Student.**class**);

}

@Test

@WithMockUser(roles = "ADMIN")

# void testAddNewStudent() {

Student newStudent = **new** Student(); newStudent.setName("some name"); newStudent.setAddress("an address");

webClient.post().uri("/students")

.contentType(MediaType.APPLICATION\_JSON)

.accept(MediaType.APPLICATION\_JSON)

.body(Mono.just(newStudent), Student.**class**)

.**exchange**()

)

}

...

}

.**expectStatus**().**isOk**()

# .expectHeader().contentType(MediaType.APPLICATION\_JSON

.**expectBody**()

.**jsonPath**("$.**id**").**isNotEmpty**()

.**jsonPath**("$.**name**").**isEqualTo**(**newStudent**.**getName**())

.**jsonPath**("$.**address**").**isEqualTo**(**newStudent**.**getAddress**());

1. WEBSOCKETS и RSOCKET
   1. Веб-сокеты

В Spring 5 WebSockets также получает дополнительные реактивные возможности. Чтобы создать сервер WebSocket, вы можете создать реализацию WebSocketHandler интерфейса, которая содержит следующий метод:

Mono<Void> **handle**(WebSocketSession session)

Этот метод вызывается при установке нового соединения WebSocket и позволяет обрабатывать сеанс. Он принимает в WebSocketSession качестве входных данных и возвращает Mono <Void>, чтобы сигнализировать о завершении обработки сеанса приложением.

WebSocketSession имеет методы, определенные для обработки входящих и исходящих потоков:

Flux<WebSocketMessage> **receive**()

Mono<Void> **send**(Publisher<WebSocketMessage> messages)

Spring WebFlux также предоставляет WebSocketClient реализации для Reactor Netty, Tomcat, Jetty, Undertow и стандартной Java.

Для получения дополнительной информации прочтите следующую главу в документации Spring's Web on Reactive Stack: WebSockets

* 1. RSOCKET

RSocket - это протокол, моделирующий семантику реактивных потоков по сети. Это двоичный протокол для использования в транспортных потоках байтовых потоков, таких как TCP, WebSockets и Aeron. В качестве введения в эту тему я рекомендую следующий пост в блоге, который написал мой коллега Pär: An introduction to RSocket

А для получения дополнительной информации о поддержке Spring Framework протокола RSocket

1. Подводя итог…

Это сообщение в блоге продемонстрировало, как WebFlux можно использовать для создания реактивного веб-приложения.

# Задание на практическую работу №7

**Задание**

Реализовать сервис, предоставляющий реактивный REST API с использо- ванием Spring WebFlux. Предметная область выбирается студентом самостоя- тельно, но должна быть уникальна в рамках учебной группы.

Требования к сервису:

* Представлено как минимум 5 endpoint’ов;
* Представлены endpoint’в возвращающие как Mono<>, так и Flux<>;
* Продемонстрирована работа с базами данных;
* Продемонстрировано применение операторов преобразования потоков;
* Продемонстрирована работа в формате backpressure;
* Продемонстрирована обработка ошибок в потоках;
* Сервис покрыт unit-тестами.