1. Spring WebFlux

1.1. Introduction

스프링 프레임 워크에 포함 된 원래의 웹 프레임 워크 인 Spring Web MVC는 Servlet API 및 Servlet 컨테이너 용으로 개발되었습니다. 반응 형 스택 인 웹 프레임 워크 인 Spring WebFlux는 나중에 버전 5.0에 추가되었습니다. 그것은 완전히 non-blocking 이며 Reactive Streams back pressure를 지원하며 Netty, Undertow 및 Servlet 3.1+ 컨테이너와 같은 서버에서 실행됩니다.

두 웹 프레임 워크 spring-webmvc와 spring-webflux의 Spring Framework에서 나란히 공존합니다

모듈은 선택 사항입니다. 응용 프로그램은 하나 또는 다른 모듈을 사용할 수도 있고 어떤 경우에는 둘 다 사용할 수도 있습니다. 예를 들어 MVC 컨트롤러와 Reactive WebClient 처럼.

1.1.1. Motivation

왜 WebFlux가 만들어졌는가?

질문에 대한 일부 대답은 적은 수의 스레드로 동시성을 처리하고 적은 하드웨어 자원으로 확장 할 수있는 논 블로킹 웹 스택이 필요하다는 것입니다. Servlet 3.1은 non-blocking I / O를위한 API를 제공했습니다. 그러나 이를 사용하면 계약이 동기 (필터, 서블릿) 또는 블로킹(getParameter, getPart) 하는 나머지 Servlet API에서 멀어집니다. 이것이 non-blocking 런타임에서 기초가 되는 새로운 공통 API에 대한 동기였습니다.

대답의 다른 부분은 함수형 프로그래밍입니다. 주석이 달린 REST 컨트롤러 또는 단위 테스트에서 Java 8에서 람다식이 추가 됨으로써 Java의 함수형 API에 대한 기회가 생겼다. CompletableFuture 및 ReactiveX에서 널리 사용되는 non-blocking 애플리케이션 및 연속성 API에 대한 이점으로 비동기식 로직의 선언적 설정을 허용합니다. 프로그래밍 모델 레벨에서 Java 8은 Spring WebFlux를 사용하여 주석이 달린 컨트롤러와 함께 기능 웹 엔드 포인트를 제공합니다

1.1.2. Define "reactive"

무엇이 reactive인가??

"reactive"이라는 용어는 I / O 이벤트, 마우스 이벤트에 반응하는 UI 컨트롤러 등과 같은 네트워크 설정 요소에 반응하여 구축되는 프로그래밍 모델을 의미합니다. 이러한 의미에서 non-blocking은 reactive가 아니다. 이제 작업이 완료되거나 데이터를 사용할 수 있게 되면서 알림에 반응하는 모드로 전환됩니다.

또한 Spring 팀이 "reactive"이라고 부르는 또 다른 중요한 메커니즘이 있습니다. 이는 non-blocking back pressure입니다. 동기방식에서 blocking 은 발신자가 대기하도록하는 자연스러운 back pressure를 제공한다. 논 블로킹 코드에서는 이벤트 속도를 제어하여 빠른 제작자가 대상을 압도하지 않도록하는 것이 중요합니다.

reactive 스트림은 Java 9에서도 채택 된 사양입니다. 이것은 back pressure와 비동기 구성 요소 사이의 상호 작용을 정의합니다. data repository(Publisher)는 HTTP 서버(Subscriber)가 응답에 쓸 수 있는 데이터를 생성 할 수 있습니다

reactive 스트림의 주요 목적은 Subscriber 가 Publisher의 데이터를 생성하는 속도 또는 속도를 제어 할 수 있게 하는 것입니다

1.1.3. Reactive API

reactive 스트림은 상호 운용성에 중요한 역할을 합니다.

라이브러리 및 infrastructure components  는 관심의 대상이지만 응용 프로그램 API는 너무 low level 이기 때문에 응용 프로그램 API로 유용하지 않습니다. 애플리케이션에 필요한 것은 비동기 로직을 ​​구성하는 더 높은 수준의 풍부하고 기능적인 API입니다. 이것이 reactive 라이브러리의 역할입니다.

Reactor는 Spring WebFlux에서 선택할 수 있는 reactive 라이브러리입니다

ReactX 어휘와 연계 된 풍부한 연산자 집합을 통해 0..1 및 0..N의 데이터 시퀀스에서 작동하는 Mono 및 Flux API 유형을 제공합니다. Reactor는 Reactive Streams 라이브러리이므로 모든 연산자가 non-blocking back pressure을 지원합니다. Reactor는 서버 측 Java에 중점을 두고 있습니다. Spring과 긴밀히 협력하여 개발되었습니다.

WebFlux는 Reactor를 core dependency 로 필요로 하지만, Reactive Streams를 통해 다른 reactive 라이브러리와 상호 운용됩니다. 일반적으로 WebFlux API는 일반 Publisher를 input으로 받아 들여 내부적으로 Reactor 유형에 적용하고 사용하여 Flux 와 Mono를 output으로 반환합니다. 따라서 Publisher를 input으로 전달할 수 있으며 output에 연산을 적용 할 수 있지만, 출력을 다른 reactive 라이브러리와 함께 사용하도록 조정해야 합니다.

1.1.4. Programming models

spring-web 모듈은 reactive 기초 토대를 포함하고 있습니다.

- 그것들은 HTTP abstractions 를 포함한 **Spring WebFlux** 와

- server, codecs, WebHandler API( Servlet API의 non-blocking 계약과 유사)를 **지원하는 Reactive Streams adapters** 입니다.

스프링 WebFlux는 두 가지 프로그래밍 모델 중에서 선택할 수 있습니다

- Annotated Controllers

Spring MVC와 일치하고, Spring-Web 모듈과 동일한 주석을 기반으로 한다. Spring MVC와 WebFlux 컨트롤러 모두 반응 형 (Reactor, RxJava) 반환 유형을 지원하므로 결과를 구별하는 것이 쉽지 않습니다 주목할만한 차이점 중 하나는 WebFlux가 reactive @RequestBody 인수도 지원한다는 점입니다

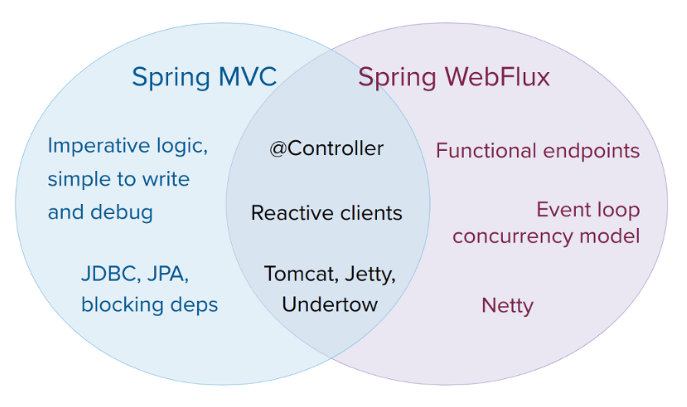
- Functional Endpoints

람다 기반, 가벼운 함수 프로그래밍 모델. 이것을 응용 프로그램이 요청을 라우팅하고 처리하는 데 사용할 수 있는 작은 라이브러리 또는 유틸리티 세트라고 생각하십시오. Annotated Controllers 와의 큰 차이점은 Annotated Controller는 요청 처리의 처음부터 끝까지 담당한다것에 반해, Functional Endpoints 는 애플리케이션이 애노테이션을 통해 intent를 선언하고 콜백된다는 것이다.

1.1.5. Applicability

**Spring MVC or WebFlux?**

당연한 질문이지만 이것은 이분법적인 사고를 하게 만드는 불리한 질문입니다. 실제로 사용 가능한 옵션 범위를 확장하기 위해 Spring MVC와 WebFlux는 함께 노력하고 있습니다. 두 제품은 서로 연속성과 일관성을 유지하도록 설계되었으며 나란히 사용할 수 있으며 각 측면의 feedback 은 양측에 도움이 됩니다. 아래 다이어그램은이 두 가지가 어떻게 관련되어 있는지, 그들이 공통적으로 가지고있는 것, 그리고 각각이 고유하게 지원하는 것을 보여줍니다



다음은 고려해야 할 몇 가지 사항입니다.

* 잘 작동하는 Spring MVC 애플리케이션을 가지고 있다면, 변경할 필요가 없다. Imperative 프로그래밍은 코드를 작성, 이해 및 디버그하는 가장 쉬운 방법입니다. 역사적으로 대부분 blocking 방식이기 때문에 최대한 많은 라이브러리를 선택할 수 있습니다
* 이미 non-blocking web stack을 사용한 경우 , Spring WebFlux는 다른 서버들(Netty, Tomcat, Jetty, Undertow, Servlet 3.1+ containers) 과 같은 동일한 실행 모델 혜택을 제공합니다. 그리고 Reactor, RxJava 와 같은 프로그래밍 모델을 제공한다.
* Java 8 lambda 또는 Kotlin과 함께 사용하기 위한 가볍고 함수형 웹 프레임 워크에 관심이 있다면 Spring WebFlux functional web endpoints를 사용하십시오.
* 마이크로 서비스 아키텍처에서는 Spring MVC 또는 Spring WebFlux 컨트롤러 또는 Spring WebFlux 함수형 endpoints와 함께 다양한 애플리케이션을 사용할 수 있습니다. 두 프레임 워크에서 동일한 주석 기반 프로그래밍 모델을 지원하면 올바른 작업에 적합한 도구를 선택하는 동시에 지식을 쉽게 재사용 할 수 있습니다.
* Spring MVC를 사용할지 WebFlux를 사용할지 판단하는 간단한 방법은 응용 프로그램의 종속성을 확인하는 것입니다. blocking persistence API (JPA, JDBC) 또는 네트워킹 API를 사용하는 경우 Spring MVC는 공통 아키텍처에 가장 적합한 선택입니다. Reactor와 RxJava 모두 별도의 스레드에서 blocking 호출을 수행하는 것이 기술적으로 가능하지만 non-blocking web stack을 사용하는 것은 효율적이지 않다.
* 원격 서비스를 호출하는 스프링 MVC 애플리케이션을 사용하고 있다면 반응적인 WebClient 사용해보십시오. Spring MVC 컨트롤러 메소드에서 직접 reactive type(Reactor, RxJava 등)을 반환 할 수 있습니다. 호출 당 대기 시간 또는 통화 간 상호 의존성이 클수록 더 많은 이점을 얻을 수 있습니다. 스프링 MVC 컨트롤러는 다른 reactive 컴포넌트도 호출 할 수 있다.
* 규모가 큰 팀의 경우 비 차단, 기능 및 선언적 프로그래밍으로 전환하는 과정에서 많은 학습을 해야 할 수 있다. 전체 변경없이 시작하는 실질적인 방법은 사후 대응하는 WebClient를 사용하는 것입니다. 그것 이전에 작게 시작하고 이익을 측정해 보십시오. 우리는 광범위한 응용 분야에서 변화가 필요하지 않을 것으로 예상합니다. 이익을 측정하는 것이 확실하지 않은 경우, non-blocking I/O 가 동작하는 것에 대해서 학습하십시오.

1.1.6. Servers

Spring WebFlux는 Tomcat, Jetty, Servlet 3.1+ 컨테이너뿐만 아니라 non-servlet runtime이 아닌 Netty 및 Undertow 도 지원됩니다. 모든 서버는 low-level 의 공통 API에 맞게 조정되므로 더 높은 수준의 프로그래밍 모델을 여러 서버에서 지원할 수 있습니다.

Spring WebFlux에는 서버를 시작하거나 중지 할 수 있는 내장 지원 기능이 없습니다. 그러나 Spring 구성 및 WebFlux 인프라에서 응용 프로그램을 조립하고 몇 줄의 코드만으로 쉽게 실행할 수 있습니다.

스프링 부트에는 이러한 단계를 자동화하는 WebFlux 스타터가 있습니다. 본적으로 스타터는 Netty를 사용하지만 Maven 또는 Gradle 종속성을 변경하여 Tomcat, Jetty 또는 Undertow로 쉽게 전환 할 수 있습니다. 스프링 부트는 Netty로 기본 설정 되는데, async, non-blocking space에서보다 널리 사용되며 클라이언트와 서버 공유 리소스를 제공하기 때문입니다.

Tomcat과 Jetty는 Spring MVC와 WebFlux와 함께 사용할 수 있습니다. 그러나 그들이 사용되는 방식이 매우 다르다는 것을 명심하십시오. Spring MVC는 Servlet blocking I / O에 의존하며, 필요할 경우 애플리케이션이 직접 Servlet API를 사용할 수 있게 합니다. Spring WebFlux는 Servlet 3.1 non-blocking I / O를 사용한다. 또한 서블릿 API를 사용하지만 직접 사용하지는 않고 low-level adapter 통해서 사용한다.

1.1.7. Performance vs scale

동작은 매우 많은 특징과 의미가 있습니다. Reactive 및 non-blocking 은 일반적으로 응용 프로그램을 더 빠르게 실행하지 않습니다. 경우에 따라 WebClient를 사용하여 원격 호출을 병렬로 실행할 수 있습니다. 전체적으로 non-blocking 방식으로 일을 하기 위해 더 많은 작업이 필요하며 이는 필요한 처리 시간을 약간 늘릴 수 있습니다.

. Reactive 및 non-blocking 의 주요 이점은 적은 수의 고정 된 스레드와 적은 메모리로 확장 할 수 있다는 것입니다. 따라서 응용 프로그램이 예측 가능한 방식으로 확장되므로 응용 프로그램 보다 탄력적입니다. 그러나 이러한 이점을 관찰하려면 느리고 예측할 수 없는 네트워크 I / O를 포함하여 대기 시간이 필요합니다

1.1.8. Concurrency Model

Spring MVC와 Spring WebFlux는 모두 annotated controllers 를 지원하지만 concurrency 모델과 blocking 과 thread에 대한 기본 가정에서 중요한 차이가 있다

Spring MVC와 서블릿 애플리케이션에서는 일반적으로 애플리케이션이 현재 스레드를 차단할 수 있다고 가정한다. 이러한 이유로 요청 처리중 잠재적인 blocking을 처리하기 위해서 서블릿 컨테이너는 큰 스레드 풀을 사용합니다.

일반적으로, Spring WebFlux와 non-blocking 서버에서는 애플리케이션이 차단되지 않으므로 non-blocking 서버는 크기가 작은 고정 크기 스레드 풀 (이벤트 루프 작업자)을 사용하여 요청을 처리합니다

blocking라이브러리를 사용해야 하는 경우 어떻게해야 할까? Reactor와 RxJava는 모두 publishOn 연산자를 사용하여 다른 스레드에서 계속 처리합니다. 즉, 쉽게 빠져나올 수 있는 방법이 있다는 이야기 입니다. 그러나 bloking API는이 concurrency 모델에 적합하지 않다는 것을 명심하십시오.

Spring WebFlux를 실행하는 서버에서 있는 스레드는 무엇입니까?

* “vanilla" Spring WebFlux 서버 에 대한 하나의 스레드를 기대할 수 있습니다. 그리고 요청 처리를 위한 몇 가지 스레드(일반적으로 CPU 코어 수만큼)를 기대할 수 있습니다. 그러나 서블릿 컨테이너는 blocking I/O and servlet 3.1, non-blocking I/O 을 위해 더 많은 스레드로 시작될 수 있습니다.
* reactive WebClient는 이벤트 루프 스타일로 작동합니다. 따라서 이와 관련된 작은 수의 고정 된 처리 스레드를 확인할 수 있습니다. 그러나 Reactor Netty가 클라이언트와 서버 모두에 사용되는 경우, 두 장치는 기본적으로 이벤트 루프 리소스를 공유합니다
* Reactor와 RxJava는 handler를 다른 스레드 풀로 전환하는 데 사용되는 publishOn 연산자와 함께 사용할 Schedulers라는 스레드 풀 추상화 기능을 제공합니다. Schedulers 는 정 동시성 전략을 제안하는 이름을 가지고 있다. 예를 들어 ‘parallel’ 은 제한된 수의 스레드로 CPU 바인딩 작업, ‘elastic’ 은 많은 수의 스레드를 사용하는 I / O 바인딩 작업 . 이러한 스레드가 있다면 일부 코드가 특정 스레드 풀 스케줄러 전략을 사용하고 있음을 의미합니다
* 데이터 액세스 라이브러리 및 기타 타사 종속성은 자체 스레드를 만들고 사용할 수도 있습니다

Spring Framework는 서버 시작 및 중지를 지원하지 않습니다. 서버용 스레딩 모델을 구성하려면 서버 별 구성 API를 사용해야하며, 아니면 Spring Boot를 사용하는 경우 각 서버에 대한 Spring Boot 구성 옵션을 확인하십시오

1.2. Reactive Spring Web

spring-web 모듈은 반응이 적은 웹 응용 프로그램을 빌드하기 위한 저수준 infrastructure와 HTTP 추상화를 제공합니다

서버 지원은 두 개의 레이어로 구성됩니다.

- HttpHandler and server adapters : Reactive Streams back pressure를 사용한 HTTP 요청 처리를 위한 가장 기본적이고 일반적인 API

- WebHandler API : 약간 더 높은 수준이지만, 여전히 일반적인 목적의 필터 체인 스타일 서버 API

1.2.1. HttpHandler

모든 HTTP 서버에는 HTTP 요청 처리를 위한 API가 있습니다. HttpHandler는 요청 및 응답을 처리하는 한 가지 방법을 사용하는 간단한 계약입니다. 그것은 의도적으로 아주작다. 주요 목적은 서로 다른 서버에서 HTTP 요청 처리를 위한 공통 Reactive Stream 기반 API를 제공하는 것입니다.

spring-web 모듈에는 지원되는 모든 서버에 대한 어댑터가 들어 있습니다.

아래 표는 서버 API가 사용되고 Reactive Streams 지원이 제공되는 위치를 보여줍니다

| **Server name** | **Server API used** | **Reactive Streams support** |
| --- | --- | --- |
| Netty | Netty API | [Reactor Netty](https://github.com/reactor/reactor-netty) |
| Undertow | Undertow API | spring-web: Undertow to Reactive Streams bridge |
| Tomcat | Servlet 3.1 non-blocking I/O; Tomcat API to read and write ByteBuffers vs byte[] | spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge |
| Jetty | Servlet 3.1 non-blocking I/O; Jetty API to write ByteBuffers vs byte[] | spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge |
| Servlet 3.1 container | Servlet 3.1 non-blocking I/O | spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge |

다음은 각 서버에 대한 필수 종속성, 지원되는 버전 및 코드 조각입니다

| **erver name** | **Group id** | **Artifact name** |
| --- | --- | --- |
| Reactor Netty | io.projectreactor.ipc | reactor-netty |
| Undertow | io.undertow | undertow-core |
| Tomcat | org.apache.tomcat.embed | tomcat-embed-core |
| Jetty | org.eclipse.jetty | jetty-server, jetty-servlet |

Reactor Netty:

HttpHandler handler = ...

ReactorHttpHandlerAdapter adapter = **new** ReactorHttpHandlerAdapter(handler);

HttpServer.create(host, port).newHandler(adapter).block();

Undertow:

HttpHandler handler = ...

UndertowHttpHandlerAdapter adapter = **new** UndertowHttpHandlerAdapter(handler);

Undertow server = Undertow.builder().addHttpListener(port, host).setHandler(adapter).build();

server.start();

Tomcat:

HttpHandler handler = ...

Servlet servlet = **new** TomcatHttpHandlerAdapter(handler);

Tomcat server = **new** Tomcat();

File base = **new** File(System.getProperty("java.io.tmpdir"));

Context rootContext = server.addContext("", base.getAbsolutePath());

Tomcat.addServlet(rootContext, "main", servlet);

rootContext.addServletMappingDecoded("/", "main");

server.setHost(host);

server.setPort(port);

server.start();

Jetty:

HttpHandler handler = ...

Servlet servlet = **new** JettyHttpHandlerAdapter(handler);

Server server = **new** Server();

ServletContextHandler contextHandler = **new** ServletContextHandler(server, "");

contextHandler.addServlet(**new** ServletHolder(servlet), "/");

contextHandler.start();

ServerConnector connector = **new** ServerConnector(server);

connector.setHost(host);

connector.setPort(port);

server.addConnector(connector);

server.start();

1.2.2. WebHandler API

WebHandler API는 WebExceptionHandler, WebFilter 및 target WebHandler 체인을 통해 요청을 처리하기 위한 범용, 서버, 웹 API입니다. Chain 은 WebHttpHandlerBuilder로 빌더에 구성 요소를 추가하거나 Spring ApplicationContext에서 감지하도록 하여 조립 할 수 있습니다. 빌더는 지원되는 서버에서 실행될 수있는 HttpHandler를 리턴합니다.

HttpHandler는 HTTP 서버에서 가장 작은 계약을 목표로하지만 WebHandler API는 웹 응용 프로그램을 빌드하는 데 일반적으로 사용되는 필수 기능을 제공합니다. 예를 들어, WebHandler API 구성 요소에서 사용할 수 있는 ServerWebExchange는 요청 및 응답뿐만 아니라 요청 및 세션 속성, parsed form data에대한 접근, multipart data 등에 대한 액세스를 제공합니다

Special bean types

| **Bean name** | **Bean type** | **Count** | **Description** |
| --- | --- | --- | --- |
| <any> | WebExceptionHandler | 0..N | WebFilter 및 대상 WebHandler 체인의 예외 처리를 제공합니다. |
| <any> | WebFilter | 0..N | 나머지 필터 체인과 대상 WebHandler에 인터셉터 스타일 논리를 적용한다. |
| "webHandler" | WebHandler | 1 | 요청 처리기. |
| "webSessionManager" | WebSessionManager | 0..1 | WebSession의 관리자는 ServerWebExchange의 메소드를 통해 제공됩니다. 기본 값은 DefaultWebSessionManager |
| "serverCodecConfigurer" | ServerCodecConfigurer | 0..1 | ServerWebExchange 를 통해서 제공된 form data 및 multipart data를 구문 분석하기위한 HttpMessageReader에 액세스하기 위해서 사용. 기본값은 ServerCodecConfigurer.create () |
| "localeContextResolver" | LocaleContextResolver | 0..1 | ServerWebExchange의 메서드를 통해 제공된 LocaleContext resolver. 기본 값은 AcceptHeaderLocaleContextResolver |

Form data

ServerWebExchange는 양식 데이터에 액세스하기 위해 다음 방법을 제공합니다.

Mono<MultiValueMap<String, String>> getFormData();

DefaultServerWebExchange는 form data를 파싱하여 MultiValueMapdp 넣기 위해 설정된 HttpMessageReader를 사용한다. 기본적으로 FormHttpMessageReader는 ServerCodecConfigurer 빈을 통해 사용하도록 설정된다.

Multipart data (Same in Spring MVC)

ServerWebExchange는 Multipart data 에 액세스하기 위해 다음과 같은 방법을 제공합니다

Mono<MultiValueMap<String, Part>> getMultipartData();

DefaultServerWebExchange는 설정된 HttpMessageReader <MultiValueMap <String, Part >>를 사용하여 "multipart / form-data" 컨텐트를 MultiValueMap으로 파싱합니다. 현재 Synchronoss NIO Multipart는 유일하게 지원되는 타사 라이브러리이며 멀티 파트 요청의 넌 블로킹 파싱을 위해 우리가 알고있는 유일한 라이브러리입니다. ServerCodecConfigurer bean을 통해 사용할 수 있습니다 (Web Handler API 참조).

스트리밍 방식으로 멀티 파트 데이터를 구문 분석하려면, HttpMessageReader <Part> 에서 반환된 Flux <Part> 를 사용하십시오.

예를 들어, annotated controller 에서 @RequestPart를 사용하면 개별 파트에 대한 맵과 비슷한 액세스가 이름 별로 수행되므로 다중 파트 데이터를 완전히 구문 분석해야 합니다. 반대로 @RequestBody는 MultiValueMap에 모으지 않고 내용을 Flux <Part>로 디코딩 하는데 사용할 수 있습니다.

1.2.3. Message Codecs (Same in Spring MVC)

Spring-Web 모듈은 Rective Streams Publisher를 통해 HTTP 요청 및 응답 본문을 인코딩 및 디코딩하는 HttpMessageReader 및 HttpMessageWriter 계약을 정의합니다. 이러한 계약는 클라이언트 측에서 사용됩니다(예: WebClient에서, 그리고 서버 측에서, annotated controller 및 functional end point)

Jackson

HTTP Streaming

1.2.4. Filters

Forwarded Headers

CORS

1.2.5. Exceptions

1.3. DispatcherHandler

1.3.1. Special bean types

1.3.2. WebFlux Config

1.3.3. Processing

1.3.4. Result Handling

1.3.5. Exceptions

1.3.6. View Resolution

Handling

Redirecting

Content negotiation

1.4. Annotated Controllers

1.4.1. @Controller

1.4.2. Request Mapping

URI Patterns

Pattern Comparison

Consumable Media Types

Producible Media Types

Parameters and Headers

HTTP HEAD, OPTIONS

Custom Annotations

1.4.3. Handler methods

Method arguments

Return values

Type Conversion

Matrix variables

@RequestParam

@RequestHeader

@CookieValue

@ModelAttribute

@SessionAttributes

@SessionAttribute

@RequestAttribute

Multipart

@RequestBody

HttpEntity

@ResponseBody

ResponseEntity

Jackson JSON

1.4.4. Model

1.4.5. DataBinder

1.4.6. Exceptions

REST API exceptions

1.4.7. Controller Advice

1.5. URI Links

1.5.1. UriComponents

1.5.2. UriBuilder

1.5.3. URI Encoding

1.6. Functional Endpoints

1.6.1. Overview

1.6.2. HandlerFunction

ServerRequest

ServerResponse

Handler Classes

1.6.3. RouterFunction

Predicates

Routes

1.6.4. Running a server

1.6.5. HandlerFilterFunction

1.7. CORS

1.7.1. Introduction

1.7.2. Processing

1.7.3. @CrossOrigin

1.7.4. Global Config

1.7.5. CORS WebFilter

1.8. Web Security

1.9. View Technologies

1.9.1. Thymeleaf

1.9.2. FreeMarker

View config

FreeMarker config

1.9.3. Script Views

Requirements

Script templates

1.9.4. JSON, XML

1.10. HTTP Caching

1.10.1. CacheControl

1.10.2. Controllers

1.10.3. Static resources

1.11. WebFlux Config

1.11.1. Enable WebFlux config

1.11.2. WebFlux config API

1.11.3. Conversion, formatting

1.11.4. Validation

1.11.5. Content type resolvers

1.11.6. HTTP message codecs

1.11.7. View resolvers

1.11.8. Static resources

1.11.9. Path Matching

1.11.10. Advanced config mode

1.12. HTTP/2

2. WebClient

2.1. Retrieve

2.2. Exchange

2.3. Request body

2.3.1. Form data

2.3.2. Multipart data

2.4. Builder options

2.5. Client Filters

2.6. Testing

3. WebSockets

3.1. Introduction

3.1.1. HTTP vs WebSocket

3.1.2. When to use it?

3.2. WebSocket API

3.2.1. Server

3.2.2. WebSocketHandler

3.2.3. Handshake

3.2.4. Server config

3.2.5. CORS

3.2.6. Client

4. Testing

4.1. Threading model

5. Reactive Libraries