

반응 스택의 웹

버전 5.3.16

목차

1. 스프링 웹플렉스.....	2
1.1. 개요.....	2
1.1.1. "반응성"을 정의 합니다.....	2
1.1.2. 반응형 API	삼
1.1.3. 프로그래밍 모델.....	삼
1.1.4. 적용 가능성	4
1.1.5. 서버	5
1.1.6. 성능.....	5
1.1.7. 동시성 모델	6
1.2. 반응 코어	7
1.2.1. HttpHandler	7
1.2.2. 웹 핸들러 API	10
특별한 콩 종류.....	11
양식 데이터	12
멀티파트 데이터	12
전달된 헤더	12
1.2.3. 필터	13
CORS.....	13
1.2.4. 예외.....	13
1.2.5. 코덱	14
잭슨 JSON	14
양식 데이터	15
멀티파트.....	15
한계	16
스트리밍	16
데이터 버퍼	16
1.2.6. 로깅	17
로그 ID	17
민감한 데이터	17
어펜더.....	18
사용자 정의 코덱	18
1.3. 디스패처 핸들러	19
1.3.1. 특별한 콩 종류	20
1.3.2. WebFlux 구성	20
1.3.3. 처리 중	21
1.3.4. 결과 처리	21
1.3.5. 예외.....	21
1.3.6. 해상도 보기.....	22

취급	22
리디렉션	23
콘텐츠 협상	23
1.4. 주석이 달린 컨트롤러	23
1.4.1. @컨트롤러	24
1.4.2. 매팅을 요청합니다.....	25
URI 패턴	26
패턴 비교	29
소모품 미디어 유형	29
생산 가능한 미디어 유형	30
매개변수 및 헤더.....	30
HTTP 헤드, 옵션	31
맞춤 주석	32
명시적 등록	32
1.4.3. 핸들러 메소드	34
메서드 인수	34
반환 값.....	36
유형 변환	37
행렬 변수	37
@RequestParam	40
@RequestHeader	42
@쿠키값	43
@모델 속성	44
@세션 속성	47
@세션 속성	48
@요청 속성	49
멀티파트 콘텐츠	50
@RequestBody	53
HttpEntity	54
@ResponseBody	55
ResponseEntity	56
잭슨 JSON	56
1.4.4. 모델	58
1.4.5. 데이터 바인더	61
1.4.6. 예외 관리.....	63
REST API 예외.....	64
1.4.7. 컨트롤러 조언	64
1.5. 기능적 끝점	65
1.5.1. 개요.....	65
1.5.2. 핸들러 함수.....	67
서버 요청	68

서버 응답	69
핸들러 클래스	70
검증	72
1.5.3. 라우터 기능	74
술어	75
경로	75
종첩 경로	76
1.5.4. 서버 실행	78
1.5.5. 핸들러 기능 필터링 ..	80
1.6. URI 링크	83
1.6.1. 우리컴포넌트	83
1.6.2. 우리빌더	85
1.6.3. URI 인코딩	86
1.7. CORS	89
1.7.1. 소개	89
1.7.2. 처리 중	90
1.7.3. @CrossOrigin	90
1.7.4. 전역 구성	93
1.7.5. CORS 웹 필터	95
1.8. 웹 보안	96
1.9. 기술 보기	97
1.9.1. 백리향	97
1.9.2. 프리마커	97
구성 보기	97
FreeMarker 구성	98
양식 처리	99
1.9.3. 스크립트 보기 ...	100
요구 사항	101
스크립트 템플릿 ..	101
1.9.4. JSON과 XML 1.10.	104
HTTP 캐싱	104
1.10.1. 캐시 컨트롤 ..	105
1.10.2. 컨트롤러 ...	105
1.10.3. 정적 리소스	107
1.11. WebFlux 구성...	108
1.11.1. WebFlux 구성 활성화	108
1.11.2. WebFlux 구성 API 1.11.3. 변환, 형식화	108
1.11.4. 검증	111
1.11.5. 콘텐츠 유형 해석기	112
1.11.6. HTTP 메시지 코덱	113

1.11.7. 확인자 보기	114
1.11.8. 정적 리소스	117
1.11.9. 경로 일치	119
1.11.10. 웹소켓 서비스	120
1.11.11. 고급 구성 모드	121
1.12. HTTP/2	122
2. 웹 클라이언트	123
2.1. 구성	123
2.1.1. MaxInMemorySize	124
2.1.2. 원자로 네트 ..	125
자원	125
시간 초과	127
2.1.3. 부두	129
2.1.4. HttpComponents	130
2.2. 검색하다()	131
2.3. 교환	133
2.4. 요청 본문	134
2.4.1. 양식 데이터	135
2.4.2. 멀티파트 데이터	136
2.5. 필터	138
2.6. 속성	141
2.7. 컨텍스트	142
2.8. 동기 사용 ..	143
2.9. 테스트	144
3. 웹소켓	145
3.1. 웹소켓 소개	145
3.1.1. HTTP 대 WebSocket	146
3.1.2. WebSocket을 사용하는 경우	146
3.2. 웹소켓 API	146
3.2.1. 서버	147
3.2.2. 웹소켓 핸들러	149
3.2.3. 데이터 버퍼	153
3.2.4. 악수	153
3.2.5. 서버 구성	154
3.2.6. CORS	155
3.2.7. 클라이언트	155
4. 테스트	156
5. R소켓	157
5.1. 개요	157
5.1.1. 프로토콜	157
5.1.2. 자바 구현	158

5.1.3. 스프링 지원.....	159
5.2. RSocket요청자	159
5.2.1. 클라이언트 요청자	159
연결 설정	160
전략.....	160
클라이언트 응답자	161
고급	163
5.2.2. 서버 요청자.....	163
5.2.3. 요청	164
5.3. 주석이 달린 응답자	166
5.3.1. 서버 응답자	166
5.3.2. 클라이언트 응답자	168
5.3.3. @메시지 매핑.....	168
5.3.4. @ConnectMapping.....	170
5.4. 메타데이터 추출기	170
6. 리액티브 라이브러리	173

문서의 이 부분은 [Reactive Streams](#)에 구축된 반응 스택 웹 애플리케이션에 대한 지원을 다룹니다. Netty, Undertow 및 Servlet 3.1+ 컨테이너와 같은 비차단 서버에서 실행되는 API입니다. 개별 장에서는 [Spring WebFlux](#)를 다룹니다. 프레임워크, 반응형 [WebClient](#), [테스트](#) 지원 및 [반응형 라이브러리](#). Servlet-stack 웹 애플리케이션의 경우 [Web on Servlet Stack](#)을 참조하세요.

1장. 스프링 웹플럭스

Spring Framework에 포함된 원래의 웹 프레임워크인 Spring Web MVC는 Servlet API 및 Servlet 컨테이너를 위해 특별히 제작되었습니다. 반응 스택 웹 프레임워크인 Spring WebFlux는 나중에 버전 5.0에 추가되었습니다. 완전히 차단되지 않으며 [반응 스트림](#)을 지원합니다. 역할이며 Netty, Undertow 및 Servlet 3.1+ 컨테이너와 같은 서버에서 실행됩니다.

두 웹 프레임워크는 소스 모듈의 이름을 미러링합니다 ([spring-webmvc](#) 및 [스프링 웹플럭스](#)) Spring Framework에서 나란히 공존합니다. 각 모듈은 선택 사항입니다. 애플리케이션은 반응형 [WebClient](#)가 있는 Spring MVC 컨트롤러와 같이 하나 또는 다른 모듈 또는 경우에 따라 둘 다 사용할 수 있습니다.

1.1. 개요

Spring WebFlux는 왜 만들어졌나요?

답의 일부는 적은 수의 스레드로 동시성을 처리하고 더 적은 하드웨어 리소스로 확장할 수 있는 비차단 웹 스택이 필요하다는 것입니다. Servlet 3.1은 비차단 I/O를 위한 API를 제공했습니다. 그러나 이를 사용하면 계약이 동기 ([Filter, Servlet](#)) 이거나 차단 ([getParameter, getPart](#)) 인 나머지 Servlet API에서 멀어집니다. 이것은 새로운 공통 API가 모든 비차단 런타임에서 기반 역할을하게 된 동기였습니다. 이는 비동기 비차단 공간에 잘 구축된 서버(예: Netty) 때문에 중요합니다.

대답의 다른 부분은 함수형 프로그래밍입니다. Java 5에서 주석을 추가하면 기회(예: 주석이 달린 REST 컨트롤러 또는 단위 테스트)가 생성된 것처럼 Java 8에서 람다 표현식을 추가하면 Java에서 기능적 API에 대한 기회가 생성되었습니다. 이것은 비차단 애플리케이션 및 연속 스타일 API([CompletableFuture](#) 및 [ReactiveX에 의해 대중화됨](#))에 대한 이점입니다. 비동기 논리의 선언적 구성을 허용합니다. 프로그래밍 모델 수준에서 Java 8은 Spring WebFlux가 주석이 달린 컨트롤러와 함께 기능적인 웹 엔드포인트를 제공할 수 있도록 했습니다.

1.1.1. "반응성" 정의

우리는 "비차단"과 "기능적"에 대해 다루었지만 반응적이란 무엇을 의미합니까?

"반응형"이라는 용어는 I/O 이벤트에 반응하는 네트워크 구성 요소, 마우스 이벤트에 반응하는 UI 컨트롤러 등 변화에 반응하는 것을 중심으로 구축된 프로그래밍 모델을 나타냅니다. 그런 의미에서 비차단은 반응적입니다. 차단되는 대신 이제 작업이 완료되거나 데이터를 사용할 수 있게 되면 알림에 반응하는 모드에 있기 때문입니다.

Spring 팀에서 우리가 "반응성"과 연관시키는 또 다른 중요한 메커니즘이 있는데, 그것은 비차단 배압입니다. 동기식 명령형 코드에서 차단 호출은 호출자가 대기하도록 하는 자연스러운 형태의 역할 역할을 합니다. 비차단 코드에서는 빠른 생산자가 대상을 압도하지 않도록 이벤트 속도를 제어하는 것이 중요해집니다.

Reactive Streams는 [작은 사양](#)입니다. (또한 [채택](#) Java 9)에서 역할이 있는 비동기 구성 요소 간의 상호 작용을 정의합니다. 예를 들어 데이터 리포지토리([게시자 역할](#)) HTTP 서버([가입자 역할](#))가 데이터를 생성할 수 있습니다. 그러면 응답에 쓸 수 있습니다. 그만큼

Reactive Streams의 주요 목적은 구독자가 게시자가 데이터를 생성하는 속도 또는 속도를 제어할 수 있도록 하는 것입니다.

일반적인 질문: 게시자가 속도를 늦출 수 없으면 어떻게 합니까?



Reactive Streams의 목적은 메커니즘과 경계를 설정하는 것입니다. 게시자가 속도를 늦출 수 없으면 버퍼링, 삭제 또는 실패 여부를 결정해야 합니다.

1.1.2. 반응형 API

Reactive Streams는 상호 운용성을 위해 중요한 역할을 합니다. 라이브러리 및 인프라 구성 요소에 관심이 있지만 너무 낮은 수준이기 때문에 응용 프로그램 API로 덜 유용합니다.

애플리케이션은 비동기 로직을 구성하기 위해 더 높은 수준의 더 풍부하고 기능적인 API가 필요합니다. 이는 Java 8 **Stream** API와 유사하지만 컬렉션용이 아닙니다. 이것이 리액티브 라이브러리가 하는 역할입니다.

[원자로](#) Spring WebFlux에서 선택한 반응 라이브러리입니다. [모노](#) 를 제공합니다 및 [플렉스](#) 연산자의 [ReactiveX 어휘](#) 와 정렬된 풍부한 연산자 세트를 통해 0..1 ([모노](#)) 및 0..N ([플렉스](#)) 의 데이터 시퀀스에서 작동하는 API 유형 . Reactor는 Reactive Streams 라이브러리이므로 모든 오퍼레이터는 비차단 배압을 지원합니다. Reactor는 서버 측 Java에 중점을 둡니다. Spring과 긴밀히 협력하여 개발되었습니다.

WebFlux는 핵심 종속성으로 Reactor가 필요하지만 Reactive Streams를 통해 다른 반응 라이브러리와 상호 운용 가능합니다. 일반적으로 WebFlux API는 일반 [게시자](#) 를 입력으로 받아 내부적으로 Reactor 유형에 맞게 조정하고 이를 사용하고 [Flux](#) 또는 [Mono](#) 를 출력으로 반환합니다. 따라서 모든 [게시자](#) 를 입력으로 전달할 수 있고 출력에 작업을 적용할 수 있지만 다른 반응 라이브러리와 함께 사용하려면 출력을 조정해야 합니다. 가능할 때마다(예: 주석이 달린 컨트롤러) WebFlux는 RxJava 또는 다른 반응 라이브러리의 사용에 투명하게 적응합니다. 자세한 내용은 [반응형 라이브러리](#) 를 참조하세요.



Reactive API 외에도 WebFlux는 [Coroutine](#) 과 함께 사용할 수도 있습니다. 보다 명령적인 스타일의 프로그래밍을 제공하는 Kotlin의 API. 다음 Kotlin 코드 샘플은 Coroutines API와 함께 제공됩니다.

1.1.3. 프로그래밍 모델

[spring-web](#) 모듈에는 HTTP 추상화, 지원되는 서버용 Reactive Streams 어댑터 , 코덱 및 Servlet API와 비슷하지만 비차단 계약이 있는 핵심 [WebHandler API](#) 를 포함하여 Spring WebFlux의 기반이 되는 반응 기반이 포함되어 있습니다 .

이를 기반으로 Spring WebFlux는 두 가지 프로그래밍 모델 중 하나를 선택할 수 있습니다.

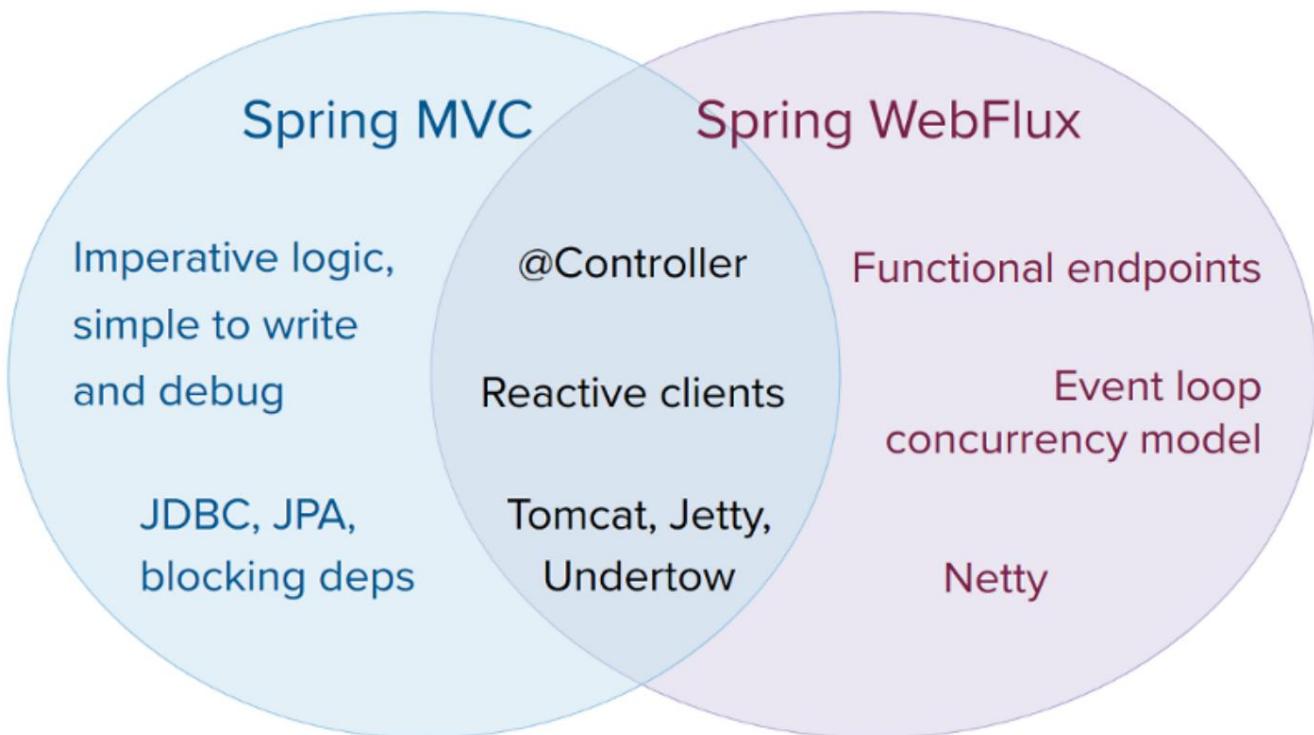
- [주석이 달린 컨트롤러](#): Spring MVC와 일치하고 [spring-web](#) 모듈의 동일한 주석을 기반으로 합니다. Spring MVC와 WebFlux 컨트롤러는 모두 반응형(Reactor 및 RxJava) 반환 유형을 지원하므로 결과적으로 구분하기가 쉽지 않습니다. 한 가지 주목할만한 차이점은 WebFlux가 반응형 [@RequestBody](#) 인수도 지원한다는 것입니다.
- [기능적 엔드포인트](#): Lambda 기반의 경량 기능적 프로그래밍 모델입니다. 이것을 애플리케이션이 요청을 라우팅하고 처리하는 데 사용할 수 있는 작은 라이브러리 또는 유ти리티 세트로 생각할 수 있습니다. 주석이 달린 컨트롤러와의 가장 큰 차이점은 애플리케이션이

주석을 통해 의도를 선언하고 다시 호출하는 것과 비교하여 처음부터 끝까지 요청 처리.

1.1.4. 적용 가능성

Spring MVC 또는 WebFlux?

당연한 질문이지만 불건전한 이분법을 설정하는 질문입니다. 실제로 둘 다 함께 작동하여 사용 가능한 옵션의 범위를 확장합니다. 이 둘은 서로의 연속성과 일관성을 위해 설계되었으며 나란히 사용할 수 있으며 양쪽의 피드백은 양쪽 모두에 도움이 됩니다. 다음 다이어그램은 이 둘의 관계, 공통점 및 각각이 고유하게 지원하는 것을 보여줍니다.



다음과 같은 특정 사항을 고려하는 것이 좋습니다.

- 잘 작동하는 Spring MVC 애플리케이션이 있다면 변경할 필요가 없다. 명령형 프로그래밍은 코드를 작성, 이해 및 디버그하는 가장 쉬운 방법입니다. 역사적으로 대부분이 차단되어 있기 때문에 라이브러리를 최대한 선택할 수 있습니다.
- 이미 비차단 웹 스택을 구매하고 있다면 Spring WebFlux는 이 분야의 다른 것과 동일한 실행 모델 이점을 제공하고 서버(Netty, Tomcat, Jetty, Undertow 및 Servlet 3.1+ 컨테이너)를 선택할 수도 있습니다. 프로그래밍 모델(주석이 있는 컨트롤러 및 기능적 웹 끝점) 선택 및 반응 라이브러리(Reactor, RxJava 또는 기타) 선택.
- Java 8 람다 또는 Kotlin과 함께 사용하기 위한 가볍고 기능적인 웹 프레임워크에 관심이 있는 경우 Spring WebFlux 기능적 웹 엔드포인트를 사용할 수 있습니다. 이는 더 큰 투명성과 제어의 이점을 얻을 수 있는 덜 복잡한 요구 사항을 가진 소규모 애플리케이션 또는 마이크로서비스에도 좋은 선택이 될 수 있습니다.
- 마이크로서비스 아키텍처에서는 Spring MVC 또는 Spring WebFlux 컨트롤러 또는 Spring WebFlux 기능 엔드포인트와 함께 애플리케이션을 혼합할 수 있습니다. 예 대한 지원

두 프레임워크에서 동일한 주석 기반 프로그래밍 모델을 사용하면 지식을 쉽게 재사용할 수 있으며 올바른 작업에 적합한 도구를 선택할 수 있습니다.

- 응용 프로그램을 평가하는 간단한 방법은 종속성을 확인하는 것입니다. 차단 지속성 API(JPA, JDBC) 또는 네트워킹 API를 사용할 경우 최소한 일반적인 아키텍처에는 Spring MVC가 최선의 선택입니다. Reactor와 RxJava 모두에서 별도의 스레드에서 차단 호출을 수행하는 것은 기술적으로 가능하지만 비차단 웹 스택을 최대한 활용하지 못할 것입니다.

- 원격 서비스에 대한 호출이 있는 Spring MVC 애플리케이션이 있는 경우 반응형 [WebClient를 사용해 보십시오](#).

Spring MVC 컨트롤러 메소드에서 직접 반응형(Reactor, RxJava [또는 기타](#))을 반환할 수 있습니다. 호출당 지역 시간 또는 호출 간의 상호 의존성이 클수록 이점이 더 극적입니다. Spring MVC 컨트롤러는 다른 반응형 컴포넌트도 호출할 수 있습니다.

- 대규모 팀이 있는 경우 비차단, 기능 및 선언적 프로그래밍으로의 전환에 있어 가파른 학습 곡선을 염두에 두십시오. 완전한 전환 없이 시작하는 실용적인 방법은 반응형 [WebClient를 사용하는 것입니다](#). 그 외에도 작게 시작하여 이점을 측정하십시오. 광범위한 응용 분야에서 전환이 필요하지 않을 것으로 예상합니다. 어떤 이점을 찾아야 하는지 확실하지 않은 경우 비차단 I/O의 작동 방식(예: 단일 스레드 Node.js의 동시성)과 그 효과에 대해 알아보십시오.

1.1.5. 서버

Spring WebFlux는 Tomcat, Jetty, Servlet 3.1+ 컨테이너와 Netty 및 Undertow와 같은 비서블릿 런타임에서 지원됩니다. 모든 서버는 하위 수준의 [공통 API](#)에 맞게 조정되어 서버 전체에서 상위 수준 [프로그래밍 모델](#)을 지원할 수 있습니다.

Spring WebFlux에는 서버를 시작하거나 중지하는 내장 지원이 없습니다. 그러나 Spring 구성 및 [WebFlux 인프라](#)에서 애플리케이션을 [조립](#)하고 몇 줄의 코드로 실행 [하는](#) 것은 쉽습니다.

Spring Boot에는 이러한 단계를 자동화하는 WebFlux 스타터가 있습니다. 기본적으로 스타터는 Netty를 사용하지만 Maven 또는 Gradle 종속성을 변경하여 Tomcat, Jetty 또는 Undertow로 쉽게 전환할 수 있습니다.

Spring Boot는 기본적으로 Netty를 사용하는데, 이는 비동기, 비차단 공간에서 더 널리 사용되며 클라이언트와 서버가 리소스를 공유할 수 있기 때문입니다.

Tomcat 및 Jetty는 Spring MVC 및 WebFlux와 함께 사용할 수 있습니다. 그러나 그것들이 사용되는 방식은 매우 다르다는 것을 명심하십시오. Spring MVC는 Servlet 차단 I/O에 의존하고 애플리케이션이 필요한 경우 Servlet API를 직접 사용할 수 있도록 합니다. Spring WebFlux는 Servlet 3.1 비차단 I/O에 의존하고 저수준 어댑터 뒤에서 Servlet API를 사용합니다. 직접 사용을 위해 노출되지 않습니다.

Undertow의 경우 Spring WebFlux는 Servlet API 없이 직접 Undertow API를 사용합니다.

1.1.6. 성능

성능에는 많은 특성과 의미가 있습니다. 반응 및 비차단은 일반적으로 애플리케이션을 더 빠르게 실행하지 않습니다. 어떤 경우에는 가능합니다(예: [WebClient](#)를 사용하여 원격 호출을 병렬로 실행하는 경우). 전체적으로 비차단 방식으로 작업을 수행하려면 더 많은 작업이 필요하며 필요한 처리 시간이 약간 늘어날 수 있습니다.

반응 및 비차단의 주요 예상 이점은 작고 고정된

스레드 수와 메모리가 적습니다. 이는 애플리케이션이 보다 예측 가능한 방식으로 확장되기 때문에 부하 시 애플리케이션을 더욱 탄력적으로 만듭니다. 그러나 이러한 이점을 관찰하려면 약간의 대기 시간이 필요합니다(느리고 예측할 수 없는 네트워크 I/O의 혼합 포함). 이것이 바로 반응성 스택이 강점을 보이기 시작하는 지점이며, 그 차이는 극적일 수 있습니다.

1.1.7. 동시성 모델

Spring MVC와 Spring WebFlux는 모두 주석이 달린 컨트롤러를 지원하지만 동시성 모델과 차단 및 스레드에 대한 기본 가정에 중요한 차이점이 있습니다.

Spring MVC(및 일반적으로 서블릿 애플리케이션)에서는 애플리케이션이 현재 스레드를 차단할 수 있다고 가정합니다(예: 원격 호출의 경우). 이러한 이유로 서블릿 컨테이너는 요청 처리 중에 잠재적인 차단을 흡수하기 위해 큰 스레드 풀을 사용합니다.

Spring WebFlux(및 일반적으로 비차단 서버)에서는 애플리케이션이 차단되지 않는다고 가정합니다. 따라서 비차단 서버는 작은 고정 크기 스레드 풀(이벤트 루프 작업자)을 사용하여 요청을 처리합니다.



"크기 조정"과 "작은 스레드 수"가 모순되게 들릴 수 있지만 현재 스레드를 차단하지 않고 대신 콜백에 의존한다는 것은 흡수할 차단 호출이 없기 때문에 추가 스레드가 필요하지 않다는 것을 의미합니다.

차단 API 호출

차단 라이브러리를 사용해야 하는 경우 어떻게 합니까? Reactor와 RxJava는 모두 다른 스레드에서 계속 처리할 수 있도록 `publishOn` 연산자를 제공합니다. 그것은 쉬운 탈출 해치가 있다는 것을 의미합니다.

그러나 API 차단은 이 동시성 모델에 적합하지 않음을 명심하십시오.

변경 가능한 상태

Reactor 및 RxJava에서는 연산자를 통해 논리를 선언합니다. 런타임 시 개별 단계에서 데이터가 순차적으로 처리되는 반응 파이프라인이 형성됩니다. 이것의 주요 이점은 해당 파이프라인 내의 애플리케이션 코드가 동시에 호출되지 않기 때문에 애플리케이션이 변경 가능한 상태를 보호하지 않아도 된다는 것입니다.

스레딩 모델

Spring WebFlux로 실행되는 서버에서 어떤 스레드를 볼 것으로 예상해야 합니까?

- "바닐라" Spring WebFlux 서버(예: 데이터 액세스 또는 기타 선택적 종속성 없음)에서 서버용 스레드 하나와 요청 처리용 스레드 여러 개(일반적으로 CPU 코어 수만큼)를 기대할 수 있습니다. 그러나 서블릿 컨테이너는 서블릿(차단) I/O와 서블릿 3.1(비차단) I/O 사용을 모두 지원하기 위해 더 많은 스레드(예: Tomcat의 경우 10개)로 시작할 수 있습니다.
 - 반응형 **WebClient**는 이벤트 루프 스타일로 작동합니다. 따라서 이와 관련된 고정된 수의 처리 스레드를 볼 수 있습니다(예: Reactor Netty 커넥터가 있는 **reactor-http-nio-**). 그러나 Reactor Netty가 클라이언트와 서버 모두에 사용되는 경우 두 가지는 기본적으로 이벤트 루프 리소스를 공유합니다.
 - Reactor 및 RxJava는 다른 스레드 풀로 처리를 전환하는 데 사용되는 **publishOn** 연산자 와 함께 사용하기 위해 스케줄러라고 하는 스레드 풀 추상화를 제공합니다. 스케줄러에는 특정 동시성 전략을 제안하는 이름이 있습니다. 예를 들어 "병렬"(CPU의 경우

제한된 수의 스레드가 있는 바인딩된 작업) 또는 "탄력적"(많은 수의 스레드가 있는 I/O 바인딩된 작업의 경우). 이러한 스레드가 표시되면 일부 코드가 특정 스레드 풀 [스케줄러](#) 전략을 사용하고 있음을 의미합니다.

- 데이터 액세스 라이브러리 및 기타 타사 종속성도 다음의 스레드를 생성하고 사용할 수 있습니다.
자신의.

구성

Spring Framework는 [서버](#) 시작 및 중지를 지원하지 않습니다. 서버에 대한 스레딩 모델을 설정하려면 서버별 설정 API를 사용해야 하며, 스프링 부트를 사용하는 경우 서버별로 스프링 부트 설정 옵션을 확인해야 한다. [WebClient](#)를 직접 [구성](#) 할 수 있습니다. 다른 모든 라이브러리에 대해서는 해당 문서를 참조하십시오.

1.2. 반응 코어

[spring-web](#) 모듈에는 반응형 웹 애플리케이션에 대한 다음과 같은 기본 지원이 포함되어 있습니다.

- 서버 요청 처리에는 두 가지 수준의 지원이 있습니다.
 - [HttpHandler](#): Reactor Netty, Undertow, Tomcat, Jetty 및 모든 Servlet 3.1+ 컨테이너용 어댑터와 함께 비차단 I/O 및 Reactive Streams 역암을 사용하여 HTTP 요청 처리를 위한 기본 계약입니다.
 - [WebHandler API](#): 주석이 달린 컨트롤러 및 기능적 끝점과 같은 구체적인 프로그래밍 모델을 기반으로 요청 처리를 위한 약간 더 높은 수준의 범용 웹 API입니다.
- 클라이언트 측의 경우 [Reactor Netty](#) 용 어댑터와 함께 비차단 I/O 및 Reactive Streams 역암으로 HTTP 요청을 수행하기 위한 기본 [ClientHttpConnector](#) 계약이 있습니다. 반응형 [부두 HttpClient](#) 및 [Apache HttpComponents](#). 응용 프로그램에서 사용되는 더 높은 수준의 [WebClient](#)는 이 기본 계약을 기반으로 합니다.
- 클라이언트 및 서버의 경우 HTTP 요청 및 응답의 직렬화 및 역직렬화를 위한 [코덱](#) 콘텐츠.

1.2.1. [HttpHandler](#)

[HttpHandler](#) 요청과 응답을 처리하는 단일 방법을 사용하는 간단한 계약입니다. 이것은 의도적으로 최소한이며, 주요 목적은 다른 HTTP 서버 API에 대한 최소한의 추상화입니다.

다음 표에서는 지원되는 서버 API에 대해 설명합니다.

서버 이름	사용된 서버 API	반응 스트림 지원
네티	네티 API	원자로 네티
언더토우	언더토우 API	spring-web: 반응을 위한 Undertow 스트림 브리지
수코양이	서블릿 3.1 비차단 I/O ByteBuffers 대 byte[]를 읽고 쓰는 Tomcat API	spring-web: Reactive Streams 브리지에 대한 Servlet 3.1 비차단 I/O

서버 이름	사용된 서버 API	반응 스트림 지원
독	서블릿 3.1 비차단 I/O 부두 API ByteBuffers 대 byte[] 쓰기	spring-web: 서블릿 3.1 비차단 Reactive Streams 브리지에 대한 I/O
서블릿 3.1 컨테이너	서블릿 3.1 비차단 I/O	spring-web: 서블릿 3.1 비차단 Reactive Streams 브리지에 대한 I/O

다음 표에서는 서버 종속성을 설명합니다([지원되는 버전도 참조](#)).

서버 이름	그룹 아이디	아티팩트 이름
원자로 네티	io.projectreactor.netty	원자로 네티
언더토우	i.undertow	언더 토우 코어
수코양이	org.apache.tomcat.embed	바람등이 포함 코어
독	org.eclipse.jetty	부두 서버, 부두 서블릿

아래 코드 조각은 각 서버 API와 함께 **HttpHandler** 어댑터를 사용하는 방법을 보여줍니다.

원자로 네티

자바

```
HttpHandler 핸들러 = ...
ReactorHttpHandlerAdapter 어댑터 = new ReactorHttpHandlerAdapter(handler);
HttpServer.create().host(호스트).port(포트).handle(어댑터).bind().block();
```

코틀린

```
val 처리기: HttpHandler = val 어댑 ...
터 = ReactorHttpHandlerAdapter(처리기)
HttpServer.create().host(호스트).port(포트).handle(어댑터).bind().block()
```

언더토우

자바

```
HttpHandler 핸들러 = ...
UndertowHttpHandlerAdapter 어댑터 = new UndertowHttpHandlerAdapter(handler);
Undertow 서버 = Undertow.builder().addHttpListener(포트,
호스트).setHandler(어댑터).빌드();
server.start();
```

코틀린

```
발 처리기: HttpHandler = 발 어댑 ...  
터 = UndertowHttpHandlerAdapter(처리기) 발 서버 =  
Undertow.builder().addHttpListener(포트, 호스트).setHandler(어  
댑터).build() server.start()
```

수코양이

자바

```
HttpHandler 핸들러 = ...  
서블릿 서블릿 = 새로운 TomcatHttpHandlerAdapter(핸들러);  
  
톰캣 서버 = 새로운 톰캣(); 파일 기본 =  
new File(System.getProperty("java.io.tmpdir")); 컨텍스트 rootContext  
= server.addContext("", base.getAbsolutePath()); Tomcat.addServlet(rootContext,  
"메인", 서블릿); rootContext.addServletMappingDecoded("/", "메인");  
server.setHost(호스트); 서버.setPort(포트); server.start();
```

코틀린

```
val 핸들러: HttpHandler = val 서 ...  
블릿 = TomcatHttpHandlerAdapter(핸들러)  
  
val server = Tomcat() val  
base = File(System.getProperty("java.io.tmpdir")) val  
rootContext = server.addContext("", base.getAbsolutePath())  
Tomcat.addServlet(rootContext, "main", servlet)  
rootContext.addServletMappingDecoded("/", "main")  
server.host = 호스트 server.setPort(포트) server.start()
```

둘

자바

```
HttpHandler 핸들러 = ...
서블릿 서블릿 = 새로운 JettyHttpHandlerAdapter(핸들러);

서버 서버 = new Server();
ServletContextHandler 컨텍스트 핸들러 = 새로운 ServletContextHandler(서버, "");
contextHandler.addServlet(새로운 ServletHolder(서블릿), "/"); 컨텍스트 핸들러.start();

ServerConnector 커넥터 = new ServerConnector(서버); 커넥터.setHost(호
스트); 커넥터.setPort(포트); server.addConnector(커넥터); server.start();
```

코틀린

```
val 핸들러: HttpHandler = val 서블 ...  
릿 = JettyHttpHandlerAdapter(핸들러)

val 서버 = 서버() val
contextHandler = ServletContextHandler(서버, "")  
contextHandler.addServlet(ServletHolder(서블릿), "/")  
contextHandler.start();

val 커넥터 = ServerConnector(서버) connector.host  
= 호스트 connector.port = 포트 server.addConnector(커
넥터) server.start()
```

서블릿 3.1+ 컨테이너

모든 Servlet 3.1+ 컨테이너에 WAR로 배포하려면 [AbstractReactiveWebInitializer](#) 를 확장하고 포함할 수 있습니다. 전쟁에서. 해당 클래스는 [ServletHttpHandlerAdapter](#) 로 [HttpHandler](#) 를 래핑하고 이를 [서블릿](#) 으로 등록합니다 .

1.2.2. 웹 핸들러 API

`org.springframework.web.server` 패키지 는 여러 [WebExceptionHandler](#) 의 체인을 통해 요청을 처리하기 위한 범용 웹 API를 제공하기 위해 [HttpHandler](#) 계약을 기반으로 합니다 . 다중 [WebFilter](#), 및 단일 [WebHandler](#) 요소, 구성 요소가 [자동으로 감지](#) 되는 Spring [ApplicationContext](#) 를 가리키거나 빌더에 구성 요소를 등록하여 체인을 [WebHttpHandlerBuilder](#) 와 함께 사용할 수 있습니다 .

[HttpHandler](#) 는 다른 HTTP 서버의 사용을 추상화하는 단순한 목표를 가지고 있지만 [WebHandler](#) API는 다음과 같은 웹 애플리케이션에서 일반적으로 사용되는 광범위한 기능 세트를 제공하는 것을 목표로 합니다.

- 속성이 있는 사용자 세션.
- 요청 속성.
- 요청에 대한 확인된 [로케일](#) 또는 [주체](#).
- 구문 분석 및 캐시된 양식 데이터에 대한 액세스.
- 멀티파트 데이터에 대한 추상화.
- 그리고 더..

특별한 콩 종류

아래 표는 [WebHttpHandlerBuilder](#) 가 Spring에서 자동 감지할 수 있는 구성 요소를 나열합니다.

ApplicationContext 또는 직접 등록할 수 있습니다.

콩 이름	콩 종류	세타	설명
<모든>	웹 예외 처리기	0..N	예외 처리 제공 WebFilter 체인에서 인스턴스 및 대상 WebHandler . 자세한 내용은 예외를 참조하십시오 .
<모든>	웹필터	0..N	가로채기 스타일 논리 적용 나머지 필터 전후 체인 및 대상 WebHandler . 을위한 자세한 내용은 필터를 참조하십시오 .
웹 핸들러	웹 핸들러	1	요청에 대한 핸들러입니다.
웹세션매니저	웹세션매니저	0..1	WebSession 의 관리자 통해 노출된 인스턴스 ServerWebExchange 의 메소드 . 기본적으로 DefaultWebSessionManager 입니다.
serverCodecConfigurer	ServerCodecConfigurer	0..1	HttpMessageReader 에 액세스하려면 양식 데이터 구문 분석을 위한 인스턴스 및 노출되는 멀티파트 데이터 방법을 통해 서버웹 익스체인지 . ServerCodecConfigurer.create() 기본.
localeContextResolver	LocaleContextResolver	0..1	LocaleContext 에 대한 해석기 방법을 통해 노출 서버웹 익스체인지 . AcceptHeaderLocaleContextResolver 기본적으로.
forwardedHeaderTransfo 르메르	ForwardedHeaderTransfo 르메르	0..1	전달 유형 처리용 헤더를 추출하고 제거하거나 제거하여 그들만. 기본적으로 사용되지 않습니다.

양식 데이터

`ServerWebExchange` 는 다음과 같은 양식 데이터 액세스 방법을 제공합니다.

자바

```
Mono<MultiValueMap<문자열, 문자열>> formData();
```

코틀린

```
일시 중단 재미 formData(): MultiValueMap<문자열, 문자열>
```

`DefaultServerWebExchange` 는 구성된 `HttpMessageReader` 를 사용하여 양식 데이터 (`application/x-www-form-urlencoded`) 를 `MultiValueMap` 으로 구문 분석합니다. 기본적으로 `FormHttpMessageReader` 는 `ServerCodecConfigurer` 빈에서 사용하도록 구성됩니다(웹 핸들러 API 참조).

멀티파트 데이터

웹 MVC

`ServerWebExchange` 는 멀티파트 데이터에 액세스하기 위해 다음과 같은 방법을 제공합니다.

자바

```
Mono<MultiValueMap<문자열, 부분>> multipartData();
```

코틀린

```
일시 중단 재미 multipartData(): MultiValueMap<문자열, 부분>
```

`DefaultServerWebExchange` 는 구성된 `HttpMessageReader<MultiValueMap<String, Part>>` 를 사용하여 `multipart/form-data` 콘텐츠를 `MultiValueMap` 으로 구문 분석 합니다. 기본적으로 이것은 타사 종속성이 없는 `DefaultPartHttpMessageReader` 입니다. 또는 `Synchronous NIO Multipart` 를 기반으로 하는 `SynchronousPartHttpMessageReader` 를 사용할 수 있습니다. 도서관. 둘 다 `ServerCodecConfigurer` 빈을 통해 구성됩니다(웹 핸들러 API 참조).

스트리밍 방식으로 멀티파트 데이터를 구문 분석하려면 `HttpMessageReader<Part>` 에서 반환된 `Flux<Part>` 를 대신 사용할 수 있습니다. 예를 들어, 주석이 달린 컨트롤러에서 `@RequestPart` 를 사용하면 이름별로 개별 부품에 대한 맵과 같은 액세스를 의미 하므로 멀티파트 데이터 전체를 구문 분석해야 합니다. 대조적으로 `@RequestBody` 를 사용하여 `MultiValueMap` 에 수집하지 않고 `Flux<Part>` 로 콘텐츠를 디코딩 할 수 있습니다.

전달된 헤더

웹 MVC

요청이 프록시(예: 로드 밸런서)를 통과할 때 호스트, 포트 및 체계가 변경될 수 있습니다.

따라서 클라이언트 관점에서 올바른 호스트, 포트 및 체계를 가리키는 링크를 만드는 것이 어렵습니다.

RFC 7239 프록시가 원래 요청에 대한 정보를 제공하는 데 사용할 수 있는 전달 된 HTTP 헤더를 정의합니다 . `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto`, `X-Forwarded-Ssl` 및 `X-Forwarded-Prefix` 를 비롯한 다른 비표준 헤더도 있습니다 .

`ForwardedHeaderTransformer` 는 전달된 헤더를 기반으로 요청의 호스트, 포트 및 체계를 수정한 다음 해당 헤더를 제거하는 구성 요소입니다 . `forwardedHeaderTransformer`라는 이름의 Bean으로 선언하면 감지 `되어` 사용됩니다 .

헤더가 의도한 대로 프록시에 의해 추가되었는지 또는 악의적인 클라이언트에 의해 추가되었는지 애플리케이션이 알 수 없기 때문에 전달된 헤더에 대한 보안 고려 사항이 있습니다 . 이것이 외부에서 들어오는 신뢰할 수 없는 전달 트래픽을 제거하도록 신뢰 경계에 있는 프록시를 구성해야 하는 이유입니다 . 또한 `RemoveOnly=true`로 `ForwardedHeaderTransformer` 를 구성할 수도 있습니다 . 이 경우 헤더는 제거하지만 사용하지 않습니다 .



5.1 에서 `ForwardedHeaderFilter` 는 더 이상 사용되지 않으며 `ForwardedHeaderTransformer` 로 대체되어 교환이 생성되기 전에 전달된 헤더를 더 일찍 처리할 수 있습니다 . 어쨌든 필터가 구성된 경우 필터 목록에서 제거되고 대신 `ForwardedHeaderTransformer` 가 사용됩니다 .

1.2.3. 필터

웹 MVC

`WebHandler API`에서 `WebFilter` 를 사용하여 필터의 나머지 처리 체인과 대상 `WebHandler` 전후에 가로채기 스타일 논리를 적용할 수 있습니다 . `WebFlux Config` 를 사용할 때 `WebFilter` 등록은 Spring 빈으로 선언하고 (선택적으로) 빈 선언에 `@Order` 를 사용하거나 `Ordered` 를 구현 하여 우선 순위를 표현하는 것처럼 간단합니다 .

CORS

웹 MVC

Spring WebFlux는 컨트롤러의 주석을 통해 CORS 구성에 대한 세분화된 지원을 제공합니다 . 그러나 Spring Security와 함께 사용할 때 Spring Security의 필터 체인보다 먼저 주문해야 하는 내장 `CorsFilter` 에 의존하는 것이 좋습니다 .

`CORS` 및 [webflux-cors.pdf](#)에 대한 섹션을 참조하십시오 . 자세한 사항은 .

1.2.4. 예외

웹 MVC

`WebHandler API`에서 `WebExceptionHandler` 를 사용하여 `WebFilter` 인스턴스 체인과 대상 `WebHandler` 의 예외를 처리 할 수 있습니다 . `WebFlux Config` 를 사용할 때 `WebExceptionHandler` 를 등록하는 것은 Spring 빈으로 선언하고 (선택적으로) 빈 선언에서 `@Order` 를 사용하거나 `Ordered` 를 구현 하여 우선 순위를 표현하는 것처럼 간단합니다 .

다음 표에서는 사용 가능한 `WebExceptionHandler` 구현을 설명합니다 .

예외 처리기	설명
ResponseStatusExceptionHandler	ResponseStatusException 유형의 예외 처리를 제공합니다. 예외의 HTTP 상태 코드에 대한 응답을 설정합니다.
WebFluxResponseStatusException 연기	모든 예외에서 <code>@ResponseStatus</code> 주석의 HTTP 상태 코드도 결정할 수 있는 ResponseStatusExceptionHandler 의 확장입니다. 이 핸들러는 WebFlux 구성에서 선언됩니다.

1.2.5. 코덱

웹 MVC

[spring-web](#) 및 [spring-core](#) 모듈은 Reactive Streams 백 프레셔가 있는 비차단 I/O를 통해 상위 수준 개체와 바이트 콘텐츠를 직렬화 및 역직렬화하는 지원을 제공합니다. 다음은 이 지원에 대해 설명합니다.

- **인코더** 및 **디코더** HTTP와 독립적으로 콘텐츠를 인코딩 및 디코딩하는 저수준 계약입니다.
- **HttpMessageReader** 및 **HttpMessageWriter** HTTP 메시지를 인코딩 및 디코딩하는 계약입니다. 콘텐츠.
- **Encoder**는 **EncoderHttpMessageWriter**로 래핑되어 웹에서 사용하도록 조정할 수 있습니다. **Decoder**는 **DecoderHttpMessageReader**로 래핑될 수 있습니다.
- **데이터 버퍼** 다른 바이트 버퍼 표현(예: Netty `ByteBuf`, `java.nio.ByteBuffer` 등)을 추상화하고 모든 코덱에서 작동합니다. [데이터 버퍼 및 코덱](#) 참조 이 주제에 대한 자세한 내용은 "Spring Core" 섹션을 참조하세요.

[spring-core](#) 모듈은 `byte []`, **ByteBuffer**, **DataBuffer**, **Resource**, `String` 인코더 및 디코더 구현을 제공합니다. [spring-web](#) 모듈은 잭슨 JSON, 잭슨 스마일, JAXB2, 프로토콜 버퍼 및 기타 인코더 및 디코더와 함께 양식 데이터, 멀티파트 콘텐츠, 서버 전송 이벤트 등에 대한 웹 전용 HTTP 메시지 판독기 및 작성기 구현을 제공합니다.

ClientCodecConfigurer 및 **ServerCodecConfigurer**는 일반적으로 애플리케이션에서 사용할 코덱을 구성하고 사용자 지정하는 데 사용됩니다. [HTTP 메시지 코덱](#) 구성에 대한 섹션을 참조하십시오.

잭슨 JSON

JSON 및 바이너리 JSON ([스마일](#)) Jackson 라이브러리가 있는 경우 둘 다 지원됩니다.

Jackson2Decoder는 다음과 같이 작동합니다.

- Jackson의 비동기식 비차단 파서는 JSON 개체를 나타내는 각각의 **TokenBuffer**로 바이트 청크 스트림을 집계하는 데 사용됩니다.
- 각 **TokenBuffer**는 더 높은 수준의 객체를 생성하기 위해 Jackson의 **ObjectMapper**로 전달됩니다.
- 단일 값 게시자(예: `Mono`)로 디코딩할 때 하나의 **TokenBuffer**가 있습니다.
- 다중 값 게시자(예: `Flux`)로 디코딩할 때 각 **TokenBuffer**는 완전히 형성된 개체에 대해 충분한 바이트가 수신되는 즉시 **ObjectMapper**에 전달됩니다. 입력 내용

JSON 배열 또는 [줄로 구분된 JSON](#) 일 수 있습니다. NDJSON, JSON 라인 또는 JSON 텍스트 시퀀스와 같은 형식입니다.

[Jackson2Encoder](#) 는 다음과 같이 작동합니다.

- 단일 값 게시자(예: `Mono`)의 경우 단순히 `ObjectMapper` 를 통해 직렬화합니다 .
 - `application/json` 이 있는 다중 값 게시자의 경우 기본적으로 `Flux#collectToList()` 를 사용하여 값을 수집한 다음 결과 컬렉션을 직렬화합니다.
 - `application/x-ndjson` 또는 `application/stream+x-jackson-smile` 과 같은 스트리밍 미디어 유형이 있는 다중 값 게시자의 경우 [줄로 구분된 JSON](#) 을 사용하여 각 값을 개별적으로 인코딩, 작성 및 플러시합니다. 체재. 다른 스트리밍 미디어 유형이 인코더에 등록될 수 있습니다.
 - SSE의 경우 이벤트별로 [Jackson2Encoder](#) 가 호출되고 전달을 보장하기 위해 출력이 플러시됩니다.
- 자체 없이.



기본적으로 [Jackson2Encoder](#) 와 [Jackson2Decoder](#) 는 모두 `String` 유형의 요소를 지원하지 않습니다 . 대신 기본 가정은 문자열 또는 문자열 시퀀스가 `CharSequenceEncoder`에 의해 렌더링되는 직렬화된 JSON 콘텐츠를 나타내는 것입니다.
`Flux<String>` 에서 JSON 배열을 렌더링하는 것이 필요한 경우 `Flux #collectToList()` 를 사용하고 `Mono<List<String>>` 을 인코딩합니다 .

양식 데이터

`FormHttpMessageReader` 및 `FormHttpMessageWriter` 는 `application/x www-form-urlencoded` 콘텐츠 디코딩 및 인코딩을 지원 합니다.

여러 위치에서 양식 콘텐츠에 액세스해야 하는 서버 측에서 `ServerWebExchange` 는 `FormHttpMessageReader` 를 통해 콘텐츠를 구문 분석 한 다음 반복 액세스를 위해 결과를 캐시 하는 전용 `getFormData()` 메서드를 제공합니다. `WebHandler API` 섹션의 [양식 데이터](#) 를 참조하십시오 .

`getFormData()` 가 사용 되면 요청 본문에서 원본 원시 콘텐츠를 더 이상 읽을 수 없습니다.

이러한 이유로 응용 프로그램은 원시 요청 본문에서 읽기와 비교하여 캐시된 양식 데이터에 액세스하기 위해 일관되게 `ServerWebExchange` 를 통과해야 합니다.

멀티파트

`MultipartHttpMessageReader` 및 `MultipartHttpMessageWriter` 는 "multipart/form-data" 콘텐츠 디코딩 및 인코딩을 지원합니다. 차례로 `MultipartHttpMessageReader` 는 `Flux<Part>` 에 대한 실제 구문 분석을 위해 다른 `HttpMessageReader` 에 위임 한 다음 단순히 부분을 `MultiValueMap` 으로 수집합니다. 기본적으로 `DefaultPartHttpMessageReader` 가 사용되지만 `ServerCodecConfigurer` 를 통해 변경할 수 있습니다 . `DefaultPartHttpMessageReader` 에 대한 자세한 내용은 다음의 [javadoc](#) 를 참조하십시오. `DefaultPartHttpMessageReader`.

다중 부분 양식 콘텐츠에 여러 위치에서 액세스해야 하는 서버 측에서 `ServerWebExchange` 는 `MultipartHttpMessageReader` 를 통해 콘텐츠를 구문 분석 한 다음 반복 액세스를 위해 결과를 캐시 하는 전용 `getMultipartData()` 메서드를 제공합니다. `WebHandler API` 섹션의 [멀티파트 데이터](#) 를 참조하십시오 .

`getMultipartData()` 가 사용 되면 요청 본문에서 원본 원시 콘텐츠를 더 이상 읽을 수 없습니다. 이러한 이유로 응용 프로그램은 반복적으로 `getMultipartData()` 를 사용하여 부분에 대한 액세스와 같은 매핑을 수행하거나 `Flux<Part>` 에 대한 일회성 액세스를 위해 `SynchronousPartHttpMessageReader` 에 의존해야 합니다 .

제한

입력 스트림의 일부 또는 전체를 버퍼링하는 [디코더](#) 및 [HttpMessageReader](#) 구현은 메모리에 버퍼링할 최대 바이트 수에 대한 제한을 사용하여 구성 할 수 있습니다. 어떤 경우에는 입력이 짐계되고 단일 개체로 표시되기 때문에 버퍼링이 발생합니다(예: `@RequestBody byte[]`가 있는 컨트롤러 메서드, `x-www-form-urlencoded` 데이터 등). 버퍼링은 입력 스트림을 분할할 때 스트리밍과 함께 발생할 수도 있습니다(예: 구분된 텍스트, JSON 개체 스트림 등). 이러한 스트리밍 사례의 경우 스트림의 한 개체와 연결된 바이트 수에 제한이 적용됩니다.

버퍼 크기를 구성하기 위해 주어진 [Decoder](#) 또는 [HttpMessageReader](#) 가 `maxInMemorySize` 속성을 노출 하는지 확인 하고 그렇다면 Javadoc 에 기본값에 대한 세부 정보가 있을 것입니다. 서버 측에서 [ServerCodecConfigurer](#) 는 모든 코덱을 설정할 수 있는 단일 위치를 제공합니다(HTTP 메시지 코덱 참조). 클라이언트 측에서는 [WebClient.Builder](#)에서 모든 코덱의 제한을 변경할 수 있습니다.

[Multipart](#) 구문 분석 의 경우 `maxInMemorySize` 속성 은 파일이 아닌 부분의 크기를 제한합니다. 파일 파트의 경우 파트가 디스크에 기록되는 임계 값을 결정합니다. 디스크에 기록된 파일 파트 의 경우 파트당 디스크 공간의 양을 제한 하는 추가 `maxDiskUsagePerPart` 속성이 있습니다. 멀티파트 요청에서 전체 파트 수를 제한 하는 `maxParts` 속성 도 있습니다 . WebFlux에서 세 가지 모두를 구성하려면 미리 구성된 [MultipartHttpMessageReader](#) 인스턴스 를 [ServerCodecConfigurer](#)에 제공해야 합니다.

스트리밍

웹 MVC

HTTP 응답(예: `text/event-stream`, `application/x-ndjson`)으로 [스트리밍](#) 할 때 연결이 끊긴 클라이언트를 더 빨리 안정적으로 감지하려면 데이터 를 주기적으로 보내는 것이 중요합니다. 이러한 전송은 주석 전용, 빈 SSE 이벤트 또는 하트비트로 효과적으로 작용할 기타 "무작동" 데이터일 수 있습니다.

데이터 버퍼

[DataBuffer](#) 는 WebFlux의 바이트 버퍼를 나타냅니다. 이 참조의 Spring Core 부분에는 [데이터 버퍼 및 코덱에 대한 섹션에 더 많은 내용이 있습니다](#). 이해해야 할 핵심 사항은 Netty와 같은 일부 서버에서 바이트 버퍼가 폴링되고 참조 카운트되며 메모리 누수를 방지하기 위해 사용 시 해제되어야 한다는 것입니다.

WebFlux 응용 프로그램은 일반적으로 더 높은 수준의 개체로 변환하는 코덱에 의존하거나 사용자 지정 코덱을 만들기로 선택하지 않는 한 데이터 버퍼 를 직접 사용하거나 생성하지 않는 한 이러한 문제에 대해 걱정할 필요가 없습니다. 이러한 경우 [데이터 버퍼 및 코덱](#) 의 정보를 검토하십시오. 특히 [DataBuffer](#) 사용에 대한 섹션을 참조하십시오.

1.2.6. 별채 반출

[웹 MVC](#)

Spring WebFlux의 **DEBUG** 레벨 로깅은 컴팩트하고 최소이며 인간 친화적으로 설계되었습니다. 특정 문제를 디버깅할 때만 유용한 정보와 반복해서 유용한 정보의 가치가 높은 비트에 중점을 둡니다.

TRACE 레벨 로깅은 일반적으로 **DEBUG** 와 동일한 원칙을 따르지만 (예를 들어 Firehose가 아니어야 함) 모든 문제를 디버깅하는 데 사용할 수 있습니다. 또한 일부 로그 메시지는 **TRACE** 대 **DEBUG** 에서 다른 수준의 세부 정보를 표시할 수 있습니다.

좋은 로깅은 로그를 사용한 경험에서 나옵니다. 명시된 목표를 충족하지 못하는 것을 발견하면 저희에게 알려주십시오.

로그 ID

WebFlux에서 단일 요청은 여러 스레드에 걸쳐 실행될 수 있으며 스레드 ID는 특정 요청에 속하는 로그 메시지의 상관 관계를 지정하는 데 유용하지 않습니다. 이것이 WebFlux 로그 메시지에 기본적으로 요청별 ID가 접두사로 붙는 이유입니다.

서버 측에서 로그 ID는 `ServerWebExchange` 속성 (`LOG_ID_ATTRIBUTE`)에 저장되며, 해당 ID를 기반으로 하는 완전히 형식화된 접두사는 `ServerWebExchange#getLogPrefix()`에서 사용할 수 있습니다. WebClient 측에서 로그 ID는 `ClientRequest` 속성 (`LOG_ID_ATTRIBUTE`)에 저장됩니다. 완전히 형식화된 접두사는 `ClientRequest#logPrefix()`에서 사용할 수 있습니다.

민감한 데이터

[웹 MVC](#)

DEBUG 및 **TRACE** 로깅은 민감한 정보를 기록할 수 있습니다. 이것이 양식 매개변수와 헤더가 기본적으로 마스킹되는 이유이며 명시적으로 전체 로깅을 활성화해야 합니다.

다음 예는 서버 측 요청에 대해 그렇게 하는 방법을 보여줍니다.

자바

```
@ 구성
@EnableWebFlux
클래스 MyConfig 는 WebFluxConfigurer 를 구현합니다. {

    @Override 공개
    무효 configureHttpMessageCodecs(ServerCodecConfigurer 구성자)
        { configurer.defaultCodecs().enableLoggingRequestDetails(true);
    }
}
```

코틀린

```

@구성
@EnableWebFlux
클래스 MyConfig : WebFluxConfigurer {

    재정의 재미 configureHttpMessageCodecs(구성자: ServerCodecConfigurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true)
    }
}

```

다음 예는 클라이언트 측 요청에 대해 그렇게 하는 방법을 보여줍니다.

자바

```

소비자<ClientCodecConfigurer> 소비자 = 구성 ->
    configurer.defaultCodecs().enableLoggingRequestDetails(true);

 WebClient webClient = WebClient.builder()
    .exchangeStrategies(전략 -> 전략.코덱(소비자)) .build();

```

코틀린

```

val 소비자: (ClientCodecConfigurer) -> 단위 = { 구성자 ->
    configurer.defaultCodecs().enableLoggingRequestDetails(true) }

val webClient =
    WebClient.builder() .exchangeStrategies({ 전략 -> 전략.코덱(소비자) }) .build()

```

어펜더

SLF4J 및 Log4J 2와 같은 로깅 라이브러리는 차단을 방지하는 비동기 로거를 제공합니다.
로깅을 위해 큐에 넣을 수 없는 잠재적으로 메시지를 삭제하는 것과 같은 자체 단점이 있지만 현재 반응형 비차단 응용 프로그램에서 사용할 수 있는 최상의 옵션입니다.

커스텀 코덱

응용 프로그램은 추가 미디어 유형을 지원하기 위해 사용자 지정 코덱을 등록하거나 기본 코덱에서 지원하지 않는 특정 동작을 등록할 수 있습니다.

개발자가 표현한 일부 구성 옵션은 기본 코덱에 적용됩니다. 사용자 지정 코덱은 [버퍼링 제한을 적용](#)하거나 [민감한 데이터를 기록](#)하는 것과 같은 기본 설정에 맞출 수 있는 기회를 원할 수 있습니다.

다음 예는 클라이언트 측 요청에 대해 그렇게 하는 방법을 보여줍니다.

자바

```
WebClient webClient = WebClient.builder()
    .codecs(구성 -> {
        CustomDecoder 디코더 = new CustomDecoder();
        configurer.customCodecs().registerWithDefaultConfig(디코더);
    })
    .짓다();
```

코틀린

```
val webClient = WebClient.builder()
    .codecs({ 구성 ->
        val 디코더 = CustomDecoder()
        configurer.customCodecs().registerWithDefaultConfig(디코더)
    })
    .짓다()
```

1.3. 디스패처 핸들러

웹 MVC

Spring MVC와 유사하게 Spring WebFlux는 전면 컨트롤러 패턴을 중심으로 설계되었습니다.

중央 [WebHandler](#)인 DispatcherHandler 는 요청 처리를 위한 공유 알고리즘을 제공하는 반면 실제 작업은 구성 가능한 대리자 구성 요소에 의해 수행됩니다. 이 모델은 유연하고 지원합니다. 다양한 워크플로.

DispatcherHandler 는 Spring 구성에서 필요한 대리자 구성 요소를 검색합니다. 그것은 또한 Spring Bean 자체로 설계되었으며 액세스를 위해 ApplicationContextAware 를 구현합니다. 실행되는 컨텍스트입니다. WebHandler 의 빈 이름으로 DispatcherHandler 가 선언 되면 다음과 같습니다 . 턴, WebHttpHandlerBuilder에 의해 발견, 다음과 같이 요청 처리 체인을 구성합니다. WebHandler API 에 설명되어 있습니다.

WebFlux 애플리케이션의 Spring 구성에는 일반적으로 다음이 포함됩니다.

- Bean 이름이 [webHandler](#)인 DispatcherHandler
- [WebFilter](#) 및 [WebExceptionHandler](#) 빈
- DispatcherHandler 특수 빈
- 기타

구성은 다음과 같이 처리 체인을 빌드하기 위해 WebHttpHandlerBuilder 에 제공됩니다.

예는 다음을 보여줍니다.

자바

```
ApplicationContext 컨텍스트 = ...
HttpHandler 핸들러 = WebHttpHandlerBuilder.applicationContext(context).build();
```

코틀린

```
val 컨텍스트: ApplicationContext = val ...  
핸들러 = WebHttpHandlerBuilder.applicationContext(context).build()
```

결과 [HttpHandler](#) 는 [서버 어댑터](#)와 함께 사용할 준비가 되었습니다.

1.3.1. 특별한 콩 종류

[웹 MVC](#)

[DispatcherHandler](#) 는 요청을 처리하고 적절한 응답을 렌더링하기 위해 특수 빈에 위임합니다. "특수 빈"은 [WebFlux](#) 프레임워크 계약을 구현하는 [Spring](#) 관리 [캐시](#) 인스턴스를 의미합니다. 일반적으로 기본 제공 계약과 함께 제공되지만 속성을 사용자 지정하거나 확장하거나 바꿀 수 있습니다.

다음 표는 [DispatcherHandler](#)에서 감지한 특수 Bean을 나열합니다. 낮은 수준에서 감지된 다른 빈도 있다는 점에 유의하십시오(웹 핸들러 API의 [특수 빈 유형](#) 참조).

콩 종류	설명
핸들러 매핑	요청을 처리기에 매핑합니다. 매핑은 주석이 달린 컨트롤러, 간단한 URL 패턴 매핑 등 HandlerMapping 구현에 따라 세부 사항이 다른 몇 가지 기준을 기반으로 합니다.
	주요 HandlerMapping 구현은 @RequestMapping 주석 메서드를 위한 RequestMappingHandlerMapping , 기능적 엔드포인트 경로를 위한 RouterFunctionMapping , URI 경로 패턴 및 WebHandler 인스턴스의 명시적 등록을 위한 SimpleUrlHandlerMapping 입니다.
핸들러 어댑터	핸들러가 실제로 호출되는 방식에 관계없이 요청에 매핑된 핸들러를 호출 하도록 DispatcherHandler 를 돋습니다. 예를 들어 주석이 달린 컨트롤러를 호출하려면 주석을 해결해야 합니다. HandlerAdapter 의 주요 목적은 이러한 세부 사항으로부터 DispatcherHandler 를 보호하는 것 입니다.
핸들러 결과 핸들러	핸들러 호출의 결과를 처리하고 응답. 결과 처리를 참조하십시오 .

1.3.2. WebFlux 구성

[웹 MVC](#)

애플리케이션은 요청을 처리하는 데 필요한 인프라 빈([Web Handler API](#) 및 [DispatcherHandler](#) 아래에 나열됨)을 선언할 수 있습니다. 그러나 대부분의 경우 [WebFlux Config](#) 가 가장 좋은 시작점입니다. 필요한 Bean을 선언하고 이를 사용자 정의하기 위해 상위 레벨 구성 콜백 API를 제공합니다.



Spring Boot는 WebFlux 구성에 의존하여 Spring WebFlux를 구성하고 많은 추가 편리한 옵션을 제공합니다.

1.3.3. 처리

[웹 MVC](#)

DispatcherHandler 는 다음과 같이 요청을 처리합니다.

- 각 **HandlerMapping** 은 일치하는 핸들러를 찾기 위해 요청되며 첫 번째 일치가 사용됩니다.
- 핸들러가 발견되면 적절한 **HandlerAdapter** 를 통해 실행 되어 반환값을 노출합니다.
HandlerResult 로 실행의 값입니다.
- HandlerResult** 는 적절한 **HandlerResultHandler** 에 제공되어 다음을 통해 처리를 완료합니다.
응답에 직접 작성하거나 뷰를 사용하여 렌더링합니다.

1.3.4. 결과 처리

HandlerAdapter 를 통한 처리기 호출의 반환 값은 일부 추가 컨텍스트와 함께 **HandlerResult** 로 래핑되고 이에 대한 지원을 요청 하는 첫 번째 **HandlerResultHandler** 에 전달됩니다. 다음 표는 [WebFlux 구성](#) 에서 선언된 사용 가능한 **HandlerResultHandler** 구현을 보여줍니다.

결과 핸들러 유형 반환 값		기본 주문
ResponseEntityResultHandler	<code> ResponseEntity</code> , 일반적으로 <code>@Controller</code> 인스턴스에서.	0
서버 응답 결과 핸들러	일반적으로 가능한 엔드포인트의 <code> ServerResponse</code> .	0
ResponseBodyResult 핸드 읽기	<code>@ResponseBody</code> 메서드 또는 <code>@RestController</code> 클래 스에서 반환 값을 처리합니다.	100
ViewResolutionResult 핸들러	<code>CharSequence</code> , 보기 , 모델 , 지도 , 또는 다른 모든 객체 는 모델 속성 으로 처리됩니다. 해상도 보기 를 참조하십시오.	정수.MAX_VALUE

1.3.5. 예외

[웹 MVC](#)

HandlerAdapter에서 반환 된 **HandlerResult** 는 일부 핸들러별 메커니즘을 기반으로 하는 오류 처리를 위한 함수를 노출할 수 있습니다. 이 오류 함수는 다음과 같은 경우에 호출됩니다.

- 핸들러(예: `@Controller`) 호출이 실패합니다.
- HandlerResultHandler** 를 통한 핸들러 반환 값의 처리가 실패합니다.

오류 함수는 핸들러에서 반환된 반응 유형이 데이터 항목을 생성하기 전에 오류 신호가 발생하는 한 응답(예: 오류 상태)을 변경할 수 있습니다.

이것이 `@Controller` 클래스의 `@ExceptionHandler` 메소드가 지원되는 방식입니다. 대조적으로 Spring MVC에서 동일한 지원은 `HandlerExceptionResolver`를 기반으로 구축됩니다. 이것은 일반적으로 중요하지 않습니다.

그러나 WebFlux에서는 `@ControllerAdvice`를 사용하여 핸들러가 선택되기 전에 발생하는 예외를 처리할 수 없음을 명심하십시오.

"주석이 있는 컨트롤러" 섹션의 [예외 관리](#) 또는 WebHandler API 섹션의 [예외](#)도 참조 하십시오.

1.3.6. 해상도 보기

웹 MVC

보기 해상도를 사용하면 특정 보기 기술에 얹매이지 않고 HTML 템플릿과 모델을 사용하여 브라우저에 렌더링할 수 있습니다. Spring WebFlux에서는 `ViewResolver` 인스턴스를 사용하여 String(논리적 보기 이름을 나타냄)을 `View` 인스턴스에 맵핑 하는 전용 `HandlerResultHandler`를 통해 보기 확인이 지원됩니다. 그런 다음 [보기](#)를 사용하여 응답을 렌더링합니다.

손질

웹 MVC

`ViewResolutionResultHandler`에 전달 된 `HandlerResult`에는 핸들러의 반환 값과 요청 처리 중에 추가된 속성이 포함된 모델이 포함됩니다. 반환 값은 다음 중 하나로 처리됩니다.

- `String, CharSequence`: 구성된 `ViewResolver` 구현 목록을 통해 `View`로 해석될 논리적 뷰 이름 .
- `void`: 선행 및 후행 슬래시를 뺀 요청 경로를 기반으로 기본 보기 이름을 선택하고 이를 [보기로 해석합니다](#). 뷰 이름이 제공되지 않았거나(예: 모델 속성이 반환됨) 비동기 반환 값(예: `Mono` 가 비어 있음)이 제공되지 않은 경우에도 마찬가지입니다.
- [렌더링](#): 보기 해결 시나리오용 API. 코드를 사용하여 IDE의 옵션 탐색 완성.
- [모델, 맵](#): 요청을 위해 모델에 추가할 추가 모델 속성.
- 기타: 기타 모든 반환 값(`BeanUtils#isSimpleProperty`에 의해 결정된 단순 유형 제외) 모델에 추가할 모델 속성으로 처리됩니다. 속성 이름은 [규칙](#)을 사용하여 클래스 이름에서 파생됩니다. 핸들러 메소드 `@ModelAttribute` 주석이 존재하지 않는 한.

모델은 비동기, 반응 유형(예: Reactor 또는 RxJava에서)을 포함할 수 있습니다. 렌더링하기 전에 `AbstractView`는 이러한 모델 속성을 구체적인 값으로 해석하고 모델을 업데이트합니다.

단일 값 반응 유형은 단일 값 또는 값 없음(비어 있는 경우)으로 확인되는 반면 다중 값 반응 유형(예: `Flux<T>`)은 수집되어 `List<T>`로 확인됩니다.

보기 해상도를 구성하는 것은 Spring 구성에 `ViewResolutionResultHandler` 빈을 추가하는 것만 큼 간단 합니다. `WebFlux Config`는 보기 확인을 위한 전용 구성 API를 제공합니다.

Spring WebFlux와 통합된 [보기 기술에 대한 자세한 내용은 보기 기술](#)을 참조하세요.

[리디렉션](#)[웹 MVC](#)

보기 이름에 특수 [리디렉션](#): 접두사를 사용하면 리디렉션을 수행할 수 있습니다. [UrlBasedViewResolver](#) (및 하위 클래스) 는 이를 리디렉션에 필요하다는 명령으로 인식합니다. 나머지 보기 이름은 리디렉션 URL입니다.

순 효과는 컨트롤러가 [RedirectView](#) 를 반환하거나

[Rendering.redirectTo\("abc"\).build\(\)](#), 하지만 이제 컨트롤러 자체가 논리적 뷰 이름으로 작동할 수 있습니다. [redirect:/some/resource](#) 와 같은 보기 이름은 현재 애플리케이션에 상대적인 반면, [redirect:https://example.com/arbitrary/path](#) 와 같은 보기 이름은 절대 URL로 리디렉션됩니다.

[콘텐츠 협상](#)[웹 MVC](#)

[ViewResolutionResultHandler](#) 는 콘텐츠 협상을 지원합니다. 요청 미디어 유형을 선택한 각 [보기에서 지원하는 미디어 유형과 비교합니다](#). 요청된 미디어 유형을 지원하는 첫 번째 [보기](#) 가 사용됩니다.

Spring WebFlux는 JSON 및 XML과 같은 미디어 유형을 지원하기 위해 [HttpMessageWriter](#) 를 통해 렌더링 되는 특수 [View](#) 인 [HttpMessageWriterView](#)를 제공합니다.

일반적으로 [WebFlux 구성](#) 을 통해 이를 기본 보기로 구성합니다. 기본 보기의 요청된 미디어 유형과 일치하는 경우 항상 선택되고 사용됩니다.

1.4. 주석이 달린 컨트롤러

[웹 MVC](#)

Spring WebFlux는 [@Controller](#) 및 [@RestController](#) 구성 요소가 주석을 사용하여 요청 매핑, 요청 입력, 예외 처리 등을 표현하는 주석 기반 프로그래밍 모델을 제공합니다. 주석이 달린 컨트롤러는 유연한 메서드 서명을 가지고 있으며 기본 클래스를 확장하거나 특정 인터페이스를 구현할 필요가 없습니다.

다음 목록은 기본 예를 보여줍니다.

자바

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String handle() { return
        "Hello WebFlux";
    }
}
```

코틀린

```
@RestController
클래스 HelloController {

    @GetMapping("/hello") 재미
    있는 핸들() = "Hello WebFlux"
}
```

앞의 예에서 메서드는 응답 본문에 쓸 **문자열** 을 반환합니다.

1.4.1. **@제어 장치**

웹 MVC

표준 Spring 빈 정의를 사용하여 컨트롤러 빈을 정의할 수 있습니다. **@Controller** 스테레오타입은 자동 감지를 허용하고 클래스 경로에서 **@Component** 클래스를 감지하고 이에 대한 빈 정의를 자동 등록하기 위한 Spring 일반 지원과 정렬 됩니다. 또한 주석이 달린 클래스에 대한 스테레오타입 역할을 하여 웹 구성 요소로서의 역할을 나타냅니다.

이러한 **@Controller** 빈의 자동 감지를 활성화하려면 다음 예제와 같이 구성 요소 스캔을 Java 구성에 추가할 수 있습니다.

자바

```
@Configuration
@ComponentScan("org.example.web")
① 공개 클래스 WebConfig {

    // ...
}
```

① org.example.web 패키지를 스캔합니다 .

코틀린

```
@Configuration
@ComponentScan("org.example.web")
① 클래스 WebConfig {

    // ...
}
```

① org.example.web 패키지를 스캔합니다 .

@RestController 는 **구성된 주석**입니다. 이는 자체적으로 **@Controller** 및 **@ResponseBody** 로 메타 주석 처리되어 모든 메소드가 유형 수준 **@ResponseBody** 주석을 상속하므로 응답 본문 대 보기 해상도 및 HTML 템플릿으로 렌더링에 직접 씁니다.

1.4.2. 매팅 요청

웹 MVC

@RequestMapping 주석은 요청을 컨트롤러 메서드에 매팅하는 데 사용됩니다. URL, HTTP 메서드, 요청 매개변수, 헤더 및 미디어 유형별로 일치하는 다양한 속성이 있습니다. 클래스 수준에서 이를 사용하여 공유 매팅을 표현하거나 메서드 수준에서 특정 끝점 매팅으로 범위를 좁힐 수 있습니다.

@RequestMapping 의 HTTP 메서드 특정 바로 가기 변형도 있습니다.

- **@GetMapping**
- **@PostMapping**
- **@PutMapping**
- **@DeleteMapping**
- **@PatchMapping**

앞의 주석은 대부분의 컨트롤러 메소드가 기본적으로 모든 HTTP 메소드와 일치하는 **@RequestMapping**을 사용하는 대신 특정 HTTP 메소드에 매팅되어야 하기 때문에 제공되는 **사용자 정의 주석**입니다. 동시에 **@RequestMapping**은 여전히 클래스 수준에서 공유 매팅을 표현하는 데 필요합니다.

다음 예제에서는 유형 및 메서드 수준 매팅을 사용합니다.

자바

```

@RestController
@RequestMapping("/사람") 클래
스 PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) { // ...

    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody 사람 사람) { // ...

    }
}

```

코틀린

```

@RestController
@RequestMapping("/사람") 클래
스 PersonController {

    @GetMapping("/{id}") 재미
    getPerson(@PathVariable id: Long): Person { // ...

}

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED) fun
    add(@RequestBody 사람: 사람) { // ...

}

```

URI 패턴

웹 MVC

glob 패턴과 와일드카드를 사용하여 요청을 매핑할 수 있습니다.

무늬	설명	예시
?	하나의 문자와 일치	"/pages/t?st.html" 은 "/pages/test.html" 및 "/pages/t3st.html"과 일치 합니다.
*	0개 이상 일치 경로 세그먼트 내의 문자	"/resources/*.png" 는 "/ resources/file.png" 와 일치 합니다. "/projects/*/versions" 는 "/projects/ spring/versions" 와 일치 하지만 "/projects/spring/boot/ versions" 와 일치하지 않습니다.
**	경로가 끝날 때까지 0개 이상의 경로 세그 먼트와 일치합니다.	"/resources/**" 는 "/resources/file.png" 및 " resources/images/file.png"와 일치 합니다. "/resources/**/file.png" 는 ** 가 경로 끝에서만 허용되므 로 유효하지 않습니다.
{이름}	경로 세그먼트를 일치시키고 "/ projects/spring/versions" 변수로 캡처하고 "name"이라는 이름의 project=spring 을 캡처합니다.	"/projects/{project}/versions" 일치
{이름: [az] +}	정규 표현식 "[az]+" 를 다음과 같 은 경로 변수와 일치시킵니다. "이름"	"/projects/{project:[az]+}/versions" 는 "/projects/ spring/ versions" 와 일치 하지만 "/projects/spring1/ versions"는 일치하지 않습니다.

무늬	설명	예시
{*길}	경로가 끝날 때까지 0개 이상의 경로 세그먼트를 일치시키고 "경로"라는 변수로 캡처합니다.	"/resources/{*file}" 은 "/resources/images/file.png" 와 일치하고 file=/images/file.png 를 캡처 합니다.

캡처된 URI 변수는 다음 예제와 같이 [@PathVariable](#)을 사용하여 액세스할 수 있습니다.

자바

```
@GetMapping("/owners/{ownerId}/pets/{petId}") 공
개 애완동물 findPet(@PathVariable Long ownerId, @PathVariable Long petId) { // ... }
```

코틀린

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet { // ... }
```

다음 예제와 같이 클래스 및 메서드 수준에서 URI 변수를 선언할 수 있습니다.

자바

```
@제어 장치
@RequestMapping("/owner/{ownerId}") ①
공개 클래스 OwnerController {

    @GetMapping ("/pets / {petId}") ②
    public Pet findPet (@PathVariable Long ownerId, @PathVariable Long petId)
        // ...
}
```

① 클래스 수준 URI 매팅.

② 메서드 수준 URI 매팅.

코틀린

```

@제어 장치
@RequestMapping("/owner/{ownerId}") ①
클래스 OwnerController {

    EtGetMapping("/pets / {petId}") ② fun
    findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet
        // ...
    }
}

```

① 클래스 수준 URI 매팅.

② 메서드 수준 URI 매팅.

URI 변수는 자동으로 적절한 유형으로 변환되거나 **TypeMismatchException**이 발생합니다. 단순 유형 (**int**, **long**, **Date** 등)은 기본적으로 지원되며 다른 모든 데이터 유형에 대한 지원을 등록할 수 있습니다. 유형 **변환** 및 **DataBinder**를 참조하십시오.

URI 변수는 명시적으로 이름을 지정할 수 있지만(예: **@PathVariable("customId")**), 이름이 동일하고 디버깅 정보를 사용하거나 Java 8의 **-parameters** 컴파일러 플래그를 사용하여 코드를 컴파일하는 경우 해당 세부정보를 생략할 수 있습니다..

{*varName} 구문은 0개 이상의 나머지 경로 세그먼트와 일치하는 URI 변수를 선언합니다.

예를 들어 **/resources/{*path}** 는 **/resources/** 아래의 모든 파일과 일치하고 "path" 변수는 **/resources** 아래의 전체 경로를 캡처합니다.

{varName:regex} 구문은 **{varName:regex}** 구문이 있는 정규식을 사용하여 URI 변수를 선언합니다. 예를 들어 URL이 **/spring-web-3.0.5.jar**인 경우 다음 메소드는 이름, 버전 및 파일 확장자를 추출합니다.

자바

```

EtGetMapping("/ {이름: [az -] +} - {버전: \\ d \\. \\ d \\. \\ d} {ext: \\. [Az] +}") 공용 무효 핸들( @PathVariable
문자열 버전, @PathVariable 문자열 ext) // ...

```

코틀린

```

EtGetMapping (" / {이름: [az -] +} - {버전: \\ d \\. \\ d \\. \\ d} {ext: \\. [Az] +}") 재미있는 핸
들( @ PathVariable 버전: 문자열, @PathVariable ext: 문자열) // ...

```

URI 경로 패턴에는 로컬, 시스템, 환경 및 기타 속성 소스에 대해 **PropertyPlaceholderConfigurer**를 통해 시작 시 확인되는 포함된 **\${...}** 자리 표시자가 있을 수도 있습니다. 예를 들어 이를 사용하여 일부 외부 구성은 기본 URL을 매개변수화할 수 있습니다.

 Spring WebFlux는 URI 경로 일치 지원을 위해 **PathPattern** 및 **PathPatternParser** 를 사용합니다. 두 클래스 모두 **spring-web** 에 있으며 런타임 시 많은 수의 URI 경로 패턴이 일치하는 웹 애플리케이션의 HTTP URL 경로와 함께 사용하도록 명시적으로 설계되었습니다.

Spring WebFlux는 **/person** 과 같은 매핑 이 **/person.*** 에도 일치 하는 Spring MVC와 달리 점미사 패턴 일치를 지원하지 않습니다. URL 기반 콘텐츠 협상의 경우 필요한 경우 더 간단하고 명시적이며 URL 경로 기반 악용에 덜 취약한 쿼리 매개변수를 사용하는 것 이 좋습니다.

패턴 비교

웹 MVC

여러 패턴이 URL과 일치하는 경우 최상의 일치를 찾기 위해 비교해야 합니다. 이것은 보다 구체적인 패턴을 찾는 **PathPattern.SPECIFICITY_COMPARATOR** 로 수행됩니다.

모든 패턴에 대해 URI 변수와 와일드카드의 수를 기반으로 점수가 계산되며, 여기서 URI 변수는 와일드카드보다 점수가 낮습니다. 총점 이 낮은 패턴이 승리합니다. 두 패턴의 점수가 같으면 더 긴 패턴이 선택됩니다.

포괄 패턴(예: ****, {*varName}**) 은 점수에서 제외되며 대신 항상 마지막에 정렬됩니다. 두 가지 패턴이 모두 포괄적인 경우 더 긴 패턴 이 선택됩니다.

소모품 미디어 유형

웹 MVC

다음 예제와 같이 요청 의 **Content-Type** 을 기반으로 요청 매핑을 좁힐 수 있습니다.

자바

```
@PostMapping(경로 = "/pets", 소비 = "application/json") public void
addPet(@RequestBody Pet pet) { // ... }
```

코틀린

```
@PostMapping("/pets", 소모 = ["application/json"]) fun
addPet(@RequestBody pet: Pet) { // ... }
```

소모 속성은 부정 표현도 지원합니다. 예를 들어 **!text/plain** 은 **text/plain** 이외의 모든 콘텐츠 유형을 의미합니다 .

클래스 수준에서 공유 **소비** 속성을 선언할 수 있습니다 . 대부분의 다른 요청 매핑과 달리

그러나 속성은 클래스 수준에서 사용될 때 메서드 수준에서 속성 재정의를 **소비** 합니다.
보다 클래스 수준 선언을 확장합니다.

- ☒ **MediaType.APPLICATION_JSON_VALUE** 은 일반적으로 사용되는 미디어 유형에 대한 상수를 제공합니다. 예를 들어,

생산 가능한 미디어 유형

[웹 MVC](#)

다음 예제와 같이 **Accept** 요청 헤더와 컨트롤러 메서드가 생성하는 콘텐츠 유형 목록을 기반으로 요청 매핑을 좁힐 수 있습니다.

자바

```
@GetMapping(경로 = "/pets/{petId}", 생성 = "응용 프로그램/json")
@EsResponseBody
공개 애완 동물 getPet (athPathVariable 문자열 애완 동물
ID) { // ...}
```

코틀린

```
@GetMapping("/pets/{petId}", 생성 = ["application/json"])
@ResponseBody
fun getPet (athPathVariable String petId): Pet { // ...}
```

미디어 유형은 문자 집합을 지정할 수 있습니다. 부정 표현식이 지원됩니다. 예를 들어 **!text/plain** 은 **text/plain** 이외의 모든 콘텐츠 유형을 의미합니다.

클래스 수준에서 공유 **생성** 속성을 선언할 수 있습니다. 그러나 대부분의 다른 요청 매핑 속성과 달리 클래스 수준에서 사용할 때 메서드 수준은 클래스 수준 선언을 확장하는 대신 속성 재정의 **를 생성** 합니다.

- ☒ **MediaType.APPLICATION_JSON_VALUE, APPLICATION_XML_VALUE** 은 일반적으로 사용되는 미디어 유형에 대한 상수를 제공합니다.

매개변수 및 헤더

[웹 MVC](#)

쿼리 매개변수 조건에 따라 요청 매핑을 좁힐 수 있습니다. 쿼리 매개변수의 존재 여부 (**myParam**), 부재 여부 (**!myParam**) 또는 특정 값 (**myParam=myValue**)에 대해 테스트할 수 있습니다. 다음 예는 값이 있는 매개변수를 테스트합니다.

자바

```
@GetMapping(경로 = "/pets/{petId}", params = "myParam=myValue") ①
public void findPet(@PathVariable String petId) { // ... }
```

① myParam 이 myValue 와 같은지 확인합니다 .

코틀린

```
@GetMapping("/pets/{petId}", params = ["myParam=myValue"]) ① 재
미있는 findPet(@PathVariable petId: String) { // ... }
```

① myParam 이 myValue 와 같은지 확인합니다 .

다음 예에서 볼 수 있듯이 요청 헤더 조건에서도 동일한 것을 사용할 수 있습니다.

자바

```
@GetMapping(path = "/pets", headers = "myHeader=myValue") ①
public void findPet(@PathVariable String petId) { // ... }
```

① myHeader 가 myValue 와 같은지 확인합니다 .

코틀린

```
@GetMapping("/pets", headers = ["myHeader=myValue"]) ① 재
미있는 findPet(@PathVariable petId: String) { // ... }
```

① myHeader 가 myValue 와 같은지 확인합니다 .

HTTP 헤드, 옵션

웹 MVC

`@GetMapping` 및 `@RequestMapping(method=RequestMethod.GET)` 은 요청 매핑을 위해 투명하게 HTTP HEAD를 지원합니다. 컨트롤러 메서드는 변경할 필요가 없습니다. `HttpHandler` 서버 어댑터에 적용된 응답 래퍼는 `Content-Length` 헤더가 실제로 응답에 쓰지 않고 쓰여진 바이트 수로 설정되도록 합니다.

기본적으로 HTTP OPTIONS는 일치하는 URL 패턴이 있는 모든 `@RequestMapping` 메서드에 나열된 HTTP 메서드 목록에 `Allow` 응답 헤더를 설정하여 처리됩니다.

HTTP 메서드 선언이 없는 `@RequestMapping`의 경우 `Allow` 헤더가 다음으로 설정됩니다.

GET, HEAD, POST, PUT, 패치, 삭제, 옵션. 컨트롤러 메서드는 항상 지원되는 HTTP 메서드를 선언해야 합니다(예: `@GetMapping`, `@PostMapping` 등)의 HTTP 메서드 특정 변형 사용).

`@RequestMapping` 메서드를 HTTP HEAD 및 HTTP OPTIONS에 명시적으로 매핑할 수 있지만 일반적인 경우에는 필요하지 않습니다.

사용자 정의 주석

웹 MVC

Spring WebFlux는 작성된 주석 사용을 지원합니다. 요청 매핑을 위해. 그것들은 `@RequestMapping`으로 메타 주석이 달렸고 더 쉽고 더 구체적인 목적으로 `@RequestMapping` 속성의 하위 집합(또는 모두)을 다시 선언하도록 구성된 주석입니다.

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PatchMapping`은 합성된 주석의 예이다. 대부분의 컨트롤러 메서드는 기본적으로 모든 HTTP 메서드와 일치하는 `@RequestMapping`을 사용하는 대신 특정 HTTP 메서드에 매핑되어야 하기 때문에 제공됩니다. 작성된 주석의 예가 필요한 경우 주석이 어떻게 선언되는지 확인하십시오.

Spring WebFlux는 또한 사용자 정의 요청 일치 로직으로 사용자 정의 요청 매핑 속성을 지원합니다. 이것은 하위 분류 `RequestMappingHandlerMapping`이 필요하고 `getCustomMethodCondition` 메서드를 재정의 해야 하는 고급 옵션입니다. 여기서 사용자 지정 속성을 확인하고 고유한 `RequestCondition`을 반환할 수 있습니다.

명시적 등록

웹 MVC

동적 등록 또는 다른 URL에 있는 동일한 처리기의 다른 인스턴스와 같은 고급 사례에 사용할 수 있는 처리기 메서드를 프로그래밍 방식으로 등록할 수 있습니다. 다음 예에서는 그렇게 하는 방법을 보여줍니다.

자바

```

@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping 매핑, UserHandler 핸들러) ①
        NoSuchMethodException 발생 {

        RequestMappingInfo 정보 =
            RequestMappingInfo .paths("/user/{id}").methods(RequestMethod.GET).build(); ②

        메소드 메소드 = UserHandler.class.getMethod("getUser", Long.class); ③

        mapping.registerMapping(정보, 처리기, 메서드); ④
    }

}

```

① 대상 핸들러와 컨트롤러에 대한 핸들러 맵핑을 주입합니다.

② 요청 맵핑 메타데이터를 준비합니다.

③ 핸들러 메소드를 가져옵니다.

④ 등록을 추가합니다.

코틀린

```

@구성 클래스
MyConfig {

    @Autowired fun
    setHandlerMapping(mapping: RequestMappingHandlerMapping, 핸들러: UserHandler) ①

        값 정보 =
        RequestMappingInfo.paths("/user/{id}").methods(RequestMethod.GET).build() ②

        val 메소드 = UserHandler::class.java.getMethod("getUser", Long::class.java) ③

        mapping.registerMapping(정보, 핸들러, 메소드) ④
    }
}

```

① 대상 핸들러와 컨트롤러에 대한 핸들러 맵핑을 주입합니다.

② 요청 맵핑 메타데이터를 준비합니다.

③ 핸들러 메소드를 가져옵니다.

④ 등록을 추가합니다.

1.4.3. 핸들러 메소드

웹 MVC

@RequestMapping 핸들러 메서드는 유연한 서명을 가지고 있으며 지원되는 컨트롤러 메서드 인수 및 반환 값 범위에서 선택할 수 있습니다.

메서드 인수

웹 MVC

다음 표는 지원되는 컨트롤러 메서드 인수를 보여줍니다.

Reactive 유형(Reactor, RxJava 또는 기타)은 I/O 차단(예: 요청 본문 읽기)을 해결해야 하는 인수에서 지원됩니다. 설명 열에 표시됩니다.

차단이 필요하지 않은 인수에는 반응형 유형이 필요하지 않습니다.

JDK 1.8의 `java.util.Optional`은 필수 속성(예: @RequestParam, @RequestHeader 등)이 있고 `required=false`에 해당하는 주석과 함께 메서드 인수로 지원됩니다.

컨트롤러 메서드 인수 설명	
서버웹 익스체인지	전체 <code>ServerWebExchange</code> 에 대한 액세스 — HTTP 요청 및 응답, 요청 및 세션 속성, <code>checkNotModified</code> 메서드 등을 위한 컨테이너입니다.
서버Http요청, 서버HttpResponse	HTTP 요청 또는 응답에 대한 액세스.
웹세션	세션에 대한 액세스. 이것은 새로운 시작을 강요하지 않습니다 속성이 추가되지 않는 한 세션. 반응형을 지원합니다.
java.security.Principal	현재 인증된 사용자 — 알려진 경우 특정 <code>Principal</code> 구현 클래스일 수 있습니다. 반응형을 지원합니다.
org.springframework.http.HttpM 방식	요청의 HTTP 메서드입니다.
java.util.로케일	사용 가능한 가장 구체적인 <code>LocaleResolver</code> 에 의해 결정되는 현재 요청 로케일 — 사실상 구성된 <code>LocaleResolver/LocaleContextResolver</code> 입니다.
java.util.TimeZone + java.time.ZoneId	<code>LocaleContextResolver</code> 에 의해 결정된 현재 요청과 관련된 시간대 .
경로변수	URI 템플릿 변수에 액세스합니다. URI 패턴을 참조하세요 .
@MatrixVariable	URI 경로 세그먼트의 이름-값 쌍에 액세스합니다. 행렬 변수를 참조하십시오 .

컨트롤러 메서드 인수 설명	
@RequestParam	Servlet 요청 매개변수에 액세스합니다. 매개변수 값은 선언된 메소드 인수 유형으로 변환됩니다. @RequestParam 을 참조하십시오 .
	@RequestParam 의 사용은 선택 사항입니다(예: 속성 설정). 이 표 뒷부분의 "기타 인수"를 참조하십시오.
@요청헤더	요청 헤더에 액세스합니다. 헤더 값은 선언된 메서드 인수 유형으로 변환됩니다. @RequestHeader 를 참조하십시오 .
@쿠키값	쿠키에 대한 액세스를 위해. 쿠키 값은 선언된 값으로 변환됩니다. 메소드 인수 유형. @CookieValue 를 참조하십시오 .
@RequestBody	HTTP 요청 본문에 액세스합니다. 본문 내용은 HttpMessageReader 인스턴스를 사용하여 선언된 메서드 인수 형식으로 변환됩니다. 반응형을 지원합니다. @RequestBody 를 참조하십시오 .
HttpEntity	요청 헤더 및 본문에 액세스합니다. 본문은 HttpMessageReader 인스턴스로 변환됩니다. 반응형을 지원합니다. HttpEntity 를 참조하세요 .
@RequestPart	multipart/form-data 요청에서 파트에 액세스하기 위한 것 입니다. 반응형을 지원합니다. 멀티파트 콘텐츠 및 멀티파트 데이터 를 참조하십시오 .
java.util.Map, org.springframework.ui.Model 및 org.springframework.ui.ModelMa 피.	HTML 컨트롤러에서 사용되는 모델에 액세스하기 위해 뷰 렌더링의 일부로 템플릿에 노출됩니다.
@모델속성	데이터 바인딩 및 유효성 검사가 적용된 모델의 기존 속성(없는 경우 인스턴스화됨)에 액세스 합니다. @ModelAttribute 와 Model 및 DataBinder 를 참조하십시오 .
	@ModelAttribute 의 사용은 선택 사항입니다(예: 속성 설정). 이 표 뒷부분의 "기타 인수"를 참조하십시오.
오류, BindingResult	@ModelAttribute 인수 와 같은 명령 개체에 대한 유효성 검사 및 데이터 바인딩 오류에 대한 액세스용입니다. Errors 또는 BindingResult 인수는 유효성이 검사된 메서드 인수 직후에 선언되어야 합니다.
SessionStatus + 클래스 수준 @세션 속성	양식 처리 완료로 표시하기 위해 클래스 수준 @SessionAttributes 주석을 통해 선언된 세션 속성 정리를 트리거합니다. 자세한 내용은 @SessionAttributes 를 참조하세요.
	세부.
UriComponentsBuilder	현재 요청의 호스트, 포트, 체계 및 컨텍스트 경로와 관련된 URL을 준비합니다. URI 링크 를 참조하십시오 .

컨트롤러 메서드 인수 설명	
@세션 속성	모든 세션 속성에 대한 액세스 - 클래스 수준의 결과로 세션에 저장된 모델 속성과 대조적으로 @SessionAttributes 선언. 자세한 내용은 @SessionAttribute를 참조하세요. 세부.
@RequestAttribute	요청 속성에 액세스합니다. 자세한 내용은 @RequestAttribute를 참조하세요.
기타 인수	메소드 인수가 위의 어느 것과도 일치하지 않으면 BeanUtils#isSimpleProperty에 의해 결정된 대로 단순 유형이면 기본적으로 @RequestParam으로 해결됩니다. 또는 @ModelAttribute로, 그렇지 않으면.

반환 값

웹 MVC

다음 표는 지원되는 컨트롤러 메서드 반환 값을 보여줍니다. Reactor, RxJava 또는 기타와 같은 라이브러리의 반응 유형은 일반적으로 모든 반환 값에 대해 지원됩니다.

컨트롤러 메서드 반환 값	설명
@ResponseBody	반환 값은 HttpMessageWriter 인스턴스를 통해 인코딩되고 응답에 기록됩니다. @ResponseBody를 참조하십시오.
HttpEntity, ResponseEntity	반환 값은 HTTP 헤더를 포함한 전체 응답을 지정하고 본문은 HttpMessageWriter 인스턴스를 통해 인코딩되고 응답에 기록됩니다. ResponseEntity 를 참조하십시오.
HttpHeaders	헤더가 있고 본문이 없는 응답을 반환합니다.
끈	ViewResolver 인스턴스로 확인되고 사용되는 뷰 이름 암시적 모델과 함께 — 명령 개체 및 @ModelAttribute 메서드를 통해 결정됩니다. 핸들러 메서드는 이전에 설명한 Model 인수를 선언하여 프로그래밍 방식으로 모델을 강화할 수도 있습니다.
보다	암시적 모델과 함께 렌더링하는 데 사용할 View 인스턴스 — 명령 개체 및 @ModelAttribute 메서드를 통해 결정됩니다. 핸들러 메서드는 이전에 설명한 Model 인수를 선언하여 프로그래밍 방식으로 모델을 강화할 수도 있습니다.
java.util.Map, org.springframework.ui.Model	암시적 모델에 추가할 속성으로, 요청 경로를 기반으로 암시적으로 결정된 보기 이름을 사용합니다.
@모델속성	요청 경로를 기반으로 묵시적으로 결정된 보기 이름을 사용하여 모델에 추가할 속성입니다. @ModelAttribute는 선택 사항입니다. 이 표 뒷부분의 "기타 반환 값"을 참조하십시오.

컨트롤러 메서드 반환 값	설명
표현	모델 및 뷰 렌더링 시나리오용 API입니다.
무효의	<code>void</code> , 비동기식(예: <code>Mono<Void></code>), 반환 유형(또는 <code>null</code> 반환 값)이 있는 메서드는 <code>ServerHttpResponse</code> , <code>ServerWebExchange</code> 인수 또는 <code>@ResponseStatus</code> 도 있는 경우 응답을 완전히 처리한 것으로 간주됩니다. 주석. 컨트롤러가 긍정적인 <code>ETag</code> 또는 <code>lastModified</code> 타임스탬프 검사 를 수행한 경우에도 마찬가지 입니다. // TODO: 자세한 내용은 컨트롤러 를 참조하세요.
	위의 어느 것도 해당되지 않는 경우 <code>void</code> 반환 유형은 REST 컨트롤러의 경우 "응답 본문 없음"을 나타내거나 HTML 컨트롤러의 경우 기본 보기 이름 선택을 나타낼 수도 있습니다.
플렉스<서버센트 이벤트>, <code>Observable<ServerSentEvent></code> 또는 기타 반응 유형	서버에서 보낸 이벤트를 내보냅니다. <code>ServerSentEvent</code> 래퍼 는 데이터만 작성 해야 하는 경우 생략할 수 있습니다(단, <code>텍스트/이벤트 스트림</code> 은 <code>생성</code> 속성을 통해 매핑에서 요청하거나 선언해야 함).
기타 반환 값	반환 값이 위의 어느 것과도 일치하지 않으면 기본적으로 보기 이름으로 처리되고 <code>문자열</code> 또는 <code>void</code> (기본 보기 이름 선택 적용)인 경우 모델에 추가될 모델 속성으로 처리됩니다. , <code>BeanUtils#isSimpleProperty</code> 의해 결정된 단순 유형이 아닌 한, 이 경우 해결되지 않은 상태로 남아 있습니다.

유형 변환

웹 MVC

문자열 기반 요청 입력을 나타내는 주석이 달린 일부 컨트롤러 메서드 인수(예: `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, `@CookieValue`) 는 인수가 `문자열` 이 아닌 다른 것으로 선언된 경우 유형 변환이 필요할 수 있습니다 .

이러한 경우 구성한 변환기에 따라 자동으로 형식 변환이 적용됩니다. 기본적으로 단순 유형(예: `int`, `long`, `Date` 및 기타)이 지원됩니다. 형식 변환은 `WebDataBinder` (`DataBinder` 참조) 를 통해 또는 `FormattingConversionService` 에 `Formatter` 를 등록 하여 (`Spring Field Formatting` 참조) 사용자 정의할 수 있습니다.

유형 변환의 실제 문제는 빈 문자열 소스 값의 처리입니다. 이러한 값은 유형 변환의 결과로 `null` 이 되면 누락된 것으로 처리됩니다 . `Long`, `UUID` 및 기타 대상 유형 의 경우일 수 있습니다 . `null` 이 주입되도록 하려면 인수 주석에 필요한 플래그를 사용 하거나 `인수` 를 `@Nullable`로 선언하십시오.

행렬 변수

웹 MVC

[RFC 3986](#) 경로 세그먼트의 이름-값 쌍에 대해 설명합니다. Spring WebFlux에서는 다음과 같이 참조합니다.

Tim Berners-Lee의 "old post" 를 기반으로 하는 "매트릭스 변수" 이지만 URI 경로 매개변수라고도 할 수 있습니다.

행렬 변수는 모든 경로 세그먼트에 나타날 수 있으며 각 변수는 세미콜론으로 구분되고 여러 값은 쉼표로 구분됩니다(예: "/cars;color=red,green;year=2012"). 반복되는 변수 이름을 통해 여러 값을 지정할 수도 있습니다(예: "color=red;color=green;color=blue").

Spring MVC와 달리 WebFlux에서는 URL에 행렬 변수의 유무가 요청 매팅에 영향을 미치지 않습니다. 즉, URI 변수를 사용하여 변수 내용을 마스킹할 필요가 없습니다. 즉, 컨트롤러 메서드에서 행렬 변수에 액세스하려면 행렬 변수가 예상되는 경로 세그먼트에 URI 변수를 추가해야 합니다. 다음 예에서는 그렇게 하는 방법을 보여줍니다.

자바

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
개 무효 findPet(@PathVariable 문자열 petId, @MatrixVariable int q) {
    // 애완 동물 ID == 42 //
    q == 11 }
```

코틀린

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
재 미 findPet(@PathVariable petId: 문자열, @MatrixVariable q: Int) {
    // 애완 동물 ID == 42 //
    q == 11 }
```

모든 경로 세그먼트에 행렬 변수가 포함될 수 있다는 점을 감안할 때 다음 예와 같이 행렬 변수가 포함될 것으로 예상되는 경로 변수를 명확하게 해야 할 수도 있습니다.

자바

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}") 공
개 무효 findPet(
    AtMatrixVariable(이름 = "q", pathVar = "ownerId") int q1,
    AtMatrixVariable(이름 = "q", pathVar = "petId") int q2)

// q1 == 11 // q2
== 22}
```

코틀린

```
@GetMapping("/owner/{ownerId}/pets/{petId}") 재미
있는 findPet(
    AtMatrixVariable(이름 = "q", pathVar = "ownerId") q1: Int,
    AtMatrixVariable(이름 = "q", pathVar = "petId") q2: Int)

// q1 == 11 // q2
== 22}
```

다음 예제와 같이 행렬 변수를 선택 사항으로 정의하고 기본값을 지정할 수 있습니다.

자바

```
// GET /pets/42

@GetMapping("/pets/{petId}") 공
개 무효 findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

// q == 1 }
```

코틀린

```
// GET /pets/42

@GetMapping("/pets/{petId}")
fun findPet(@MatrixVariable(required = false, defaultValue = "1") q: Int) {

// q == 1 }
```

모든 행렬 변수를 가져오려면 다음 예제와 같이 [MultiValueMap](#)을 사용합니다.

자바

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}") 공
개 무효 findPet(
    @MatrixVariable MultiValueMap<문자열, 문자열> matrixVars,
    AtMatrixVariable(pathVar = "petId") MultiValueMap <문자열, 문자열> petMatrixVars)
{

// matrixVars: ["q": [11,22], "r": 12, "s": 23] // petMatrixVars:
["q": 22, "s": 23]
```

코틀린

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owner/{ownerId}/pets/{petId}") 재미
있는 findPet(
    @MatrixVariable matrixVars: MultiValueMap<문자열, 문자열>,
    AtMatrixVariable(pathVar = "petId") petMatrixVars: MultiValueMap <문자열, 문자열>)
{

// matrixVars: ["q": [11,22], "r": 12, "s": 23] // petMatrixVars:
["q": 22, "s": 23]
```

@RequestParam

웹 MVC

@RequestParam 주석을 사용하여 쿼리 매개변수를 컨트롤러의 메서드 인수에 바인딩할 수 있습니다. 다음 코드 스니펫은 사용법을 보여줍니다.

자바

```

@제어 장치
@RequestMapping("/pets")
공개 클래스 EditPetForm {

// ...

@GetMapping
public String setupForm(@RequestParam("petId") int petId, 모델 모델) { ①
    애완 동물 = this.clinic.loadPet(petId);
    model.addAttribute("펫", 펫); "petForm"을
    반환합니다.
}

// ...

```

① `@RequestParam` 사용 .

코틀린

```

org.springframework.ui.set 가져오기

@제어 장치
@RequestMapping("/pets")
클래스 EditPetForm {

// ...

@GetMapping
fun setupForm(@RequestParam("petId") petId: Int, model: Model): String { ① val pet =
    Clinic.loadPet(petId) model["pet"] = 애완동물 반환 "petForm"

}

// ...

```

① `@RequestParam` 사용 .



Servlet API "요청 매개변수" 개념은 쿼리 매개변수, 양식 데이터 및 멀티파트를 하나로 통합합니다. 그러나 WebFlux에서는 각각 ServerWebExchange를 통해 개별적으로 액세스됩니다. `@RequestParam` 은 쿼리 매개변수에만 바인딩 되지만 데이터 바인딩을 사용하여 쿼리 매개변수, 양식 데이터 및 멀티파트를 **명령 캐치에 적용**할 수 있습니다.

`@RequestParam` 어노테이션 을 사용하는 메소드 매개변수는 기본적으로 필수이지만 `@RequestParam` 의 필수 플래그를 `false`로 설정하여 메소드 매개변수가 선택사항임을 지정할 수 있습니다.

또는 `java.util.Optional` 래퍼를 사용하여 인수를 선언합니다.

대상 메소드 매개변수 유형이 **문자열** 이 아닌 경우 유형 변환이 자동으로 적용됩니다 . 보다 유형 변환.

`@RequestParam` 주석이 `Map<String, String>` 또는 `MultiValueMap<String>`에 선언된 경우 , `String>` 인수를 사용하면 맵이 모든 쿼리 매개변수로 채워집니다.

`@RequestParam` 의 사용은 선택 사항입니다(예: 속성 설정). 기본적으로 모든 단순 값 유형인 인수(`BeanUtils#isSimpleProperty`에 의해 결정됨) 그리고 아니다 다른 인수로 해결된 resolver는 `@RequestParam` 으로 주석이 달린 것처럼 처리됩니다 .

@요청헤더

웹 MVC

`@RequestHeader` 주석을 사용하여 요청 헤더를 메서드 인수에 바인딩할 수 있습니다 . 제어 장치.

다음 예는 헤더가 있는 요청을 보여줍니다.

주인	로컬 호스트:8080
수용하다	<code>text/html,application/xhtml+xml,application/xml;q=0.9</code>
허용 언어	<code>fr, en-gb, q = 0.7, en, q = 0.3</code>
인코딩 수락	<code>gzip, 수축</code>
수락-문자 집합	<code>ISO-8859-1,utf-8;q=0.7,*;q=0.7</code>
살아 유지	<code>300</code>

다음 예는 `Accept-Encoding` 및 `Keep-Alive` 헤더 값을 가져옵니다.

자바

```
@GetMapping("/데모")
public 무효 핸들(
    @RequestHeader("Accept-Encoding") 문자열 인코딩, ①
    @RequestHeader("Keep-Alive") 긴 keepAlive) { ②
//...
}
```

① `Accept-Encoding` 헤더의 값을 가져옵니다.

② `Keep-Alive` 헤더의 값을 가져옵니다 .

코틀린

```
@GetMapping("/demo")
fun
    handle( @RequestHeader("Accept-Encoding") 인코딩: String, ①
            @RequestHeader("Keep-Alive") keepAlive: Long) { ② //... }
```

① **Accept-Encoding** 헤더의 값을 가져옵니다.

② **Keep-Alive** 헤더의 값을 가져옵니다.

대상 메소드 매개변수 유형이 [문자열](#) 이 아닌 경우 유형 변환이 자동으로 적용됩니다. [유형 변환을](#) 참조하십시오.

@RequestHeader 주석이 `Map <String, String>`, `MultiValueMap<String, String>` 또는 `HttpHeaders` 인수에 사용되는 경우 맵은 모든 헤더 값으로 채워집니다.



기본 제공 지원은 쉼표로 구분된 문자열을 유형 변환 시스템에 알려진 문자열 또는 기타 유형의 배열이나 컬렉션으로 변환하는 데 사용할 수 있습니다. 예를 들어 `@RequestHeader("Accept")` 주석이 달린 메서드 매개변수는 `String` 유형일 수 있지만 `String[]` 또는 `List<String>`일 수도 있습니다.

@쿠키값

[웹 MVC](#)

@CookieValue 주석을 사용하여 HTTP 쿠키 값을 컨트롤러의 메서드 인수에 바인딩할 수 있습니다.

다음 예는 쿠키가 있는 요청을 보여줍니다.

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

다음 코드 샘플은 쿠키 값을 가져오는 방법을 보여줍니다.

자바

```
@GetMapping("/demo")
public void 핸들(@CookieValue("JSESSIONID") 문자열 쿠키) { ① //... }
```

① 쿠키 값을 가져옵니다.

코틀린

```
@GetMapping("/")
demo") 재미 핸들(@CookieValue("JSESSIONID") 쿠키: 문자열) { ① //... }
```

① 쿠키 값을 가져옵니다.

대상 메소드 매개변수 유형이 **문자열** 이 아닌 경우 유형 변환이 자동으로 적용됩니다. [유형 변환을 참조하십시오](#).

@모델속성

웹 MVC

메소드 인수에 **@ModelAttribute** 주석을 사용하여 모델의 속성에 액세스하거나 존재하지 않는 경우 인스턴스화할 수 있습니다. model 속성은 또한 필드 이름과 이름이 일치하는 쿼리 매개변수 및 양식 필드의 값으로 오버레이됩니다. 이를 데이터 바인딩이라고 하며 개별 쿼리 매개변수 및 양식 필드의 구문 분석 및 변환을 처리하지 않아도 됩니다. 다음 예제에서는 **Pet** 인스턴스를 바인딩합니다.

자바

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute 애완동물 애완동물) {} ①
```

① **Pet** 의 인스턴스를 바인딩합니다.

코틀린

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute pet: Pet): 문자열 {} ①
```

① **Pet** 의 인스턴스를 바인딩합니다.

앞의 예에서 **Pet** 인스턴스는 다음과 같이 해결 됩니다.

- 모델을 통해 이미 추가된 경우 **모델에서**.
- **@SessionAttributes** 를 통한 HTTP 세션에서 .
- 기본 생성자의 호출에서.
- 쿼리 매개변수 또는 양식 필드와 일치하는 인수가 있는 "기본 생성자" 호출에서. 인수 이름은 JavaBeans **@ConstructorProperties** 를 통해 또는 바이트 코드에서 런타임에 유지되는 매개변수 이름을 통해 결정됩니다.

모델 속성 인스턴스를 얻은 후 데이터 바인딩이 적용됩니다. **WebExchangeDataBinder** 클래스는 쿼리 매개변수 및 양식 필드의 이름을 대상 **개체의 필드 이름과 일치시킵니다**.

필요한 경우 유형 변환이 적용된 후 일치하는 필드가 채워집니다. 데이터 바인딩(및 유효성 검사)에 대한 자세한 내용은 [유효성 검사를 참조하세요](#). 데이터 바인딩 사용자 지정에 대한 자세한 내용은 [DataBinder를 참조하세요](#).

데이터 바인딩으로 인해 오류가 발생할 수 있습니다. 기본적으로 [WebExchangeBindException](#)이 발생하지만 컨트롤러 메서드에서 이러한 오류를 확인하기 위해 다음 예제와 같이 [@ModelAttribute](#) 바로 옆에 [BindingResult](#) 인수를 추가할 수 있습니다.

자바

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") 애완동물 애완동물, BindingResult 결과) { ① if
(result.hasErrors()) { return "펫폼";
}

} // ... }
```

① [BindingResult](#) 추가.

코틀린

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute("pet") pet: Pet, result: BindingResult): String { ① if
(result.hasErrors()) { "펫폼" 반환

} // ... }
```

① [BindingResult](#) 추가.

[javax.validation.Valid](#) 어노테이션 또는 Spring의 [@Validated](#) 어노테이션을 추가하여 데이터 바인딩 후 유효성 검증을 자동으로 적용할 수 있습니다 ([Bean Validation](#) 참조 및 [스프링 유효성 검사](#)). 다음 예제에서는 [@Valid](#) 주석을 사용합니다.

자바

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") 애완동물 애완동물, BindingResult 결과) { ① if (result.hasErrors()) { "petForm"을 반환합니다.

} // ... }
```

① 모델 속성 인수에 [@Valid](#) 를 사용 합니다.

코틀린

```
@PostMapping("/owner/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") 애완 동물: 애완 동물, 결과: BindingResult):
    문자열 { ①
        if (result.hasErrors()) {
            "petForm" 반환
        }
        // ...
    }
```

① 모델 속성 인수에 `@Valid` 를 사용 합니다.

Spring MVC와 달리 Spring WebFlux는 모델에서 반응형을 지원합니다. 예를 들어,
`Mono<계정>` 또는 `io.reactivex.Single<계정>`. 또는 다음을 사용하여 `@ModelAttribute` 인수를 선언할 수 있습니다.
반응형 래퍼 없이 필요한 경우 실제 값으로 적절하게 해결됩니다.
그러나 `BindingResult` 인수를 사용하려면 `@ModelAttribute` 를 선언해야 합니다.
이전에 표시된 것처럼 반응형 래퍼 없이 그 앞에 인수를 추가합니다. 또는 다음을 처리할 수 있습니다.
다음 예제와 같이 반응 유형을 통한 모든 오류:

자바

```
@PostMapping("/owner/{ownerId}/pets/{petId}/edit")
공개 Mono<String> processSubmit(@Valid @ModelAttribute("pet") Mono<Pet> petMono) {
    반환 petMono
        .flatMap(애완동물 -> {
            // ...
        })
        .onErrorResume(예 -> {
            // ...
        });
}
```

코틀린

```
@PostMapping("/owner/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") petMono: Mono<Pet>): Mono<String> {
    반환 petMono
        .flatMap { 애완동물 ->
            // ...
        }
        .onErrorResume{ 예 ->
            // ...
        }
}
```

`@ModelAttribute` 의 사용 은 선택 사항입니다(예: 속성 설정). 기본적으로 모든
단순 값 유형이 아닌 인수(`BeanUtils#isSimpleProperty`에 의해 결정됨) 그리고 아니다
다른 인수 해석기에 의해 해결된 것은 `@ModelAttribute` 로 주석이 달린 것처럼 처리됩니다.

@세션 속성

웹 MVC

@SessionAttributes 는 요청 사이에 WebSession 에 모델 속성을 저장하는 데 사용됩니다. 특정 컨트롤러에서 사용하는 세션 속성을 선언하는 유형 수준 주석입니다. 여기에는 일반적으로 액세스할 후속 요청을 위해 세션에 투명하게 저장되어야 하는 모델 속성의 이름 또는 모델 속성 유형이 나열됩니다.

다음 예를 고려하십시오.

자바

```
@Controller  
@SessionAttributes("pet") ①  
public class EditPetForm { // ... }
```

① @SessionAttributes 주석 을 사용합니다.

코틀린

```
@Controller  
@SessionAttributes("pet") ① 클  
래스 EditPetForm { // ... }
```

① @SessionAttributes 주석 을 사용합니다.

첫 번째 요청에서 이름이 pet인 모델 속성이 모델에 추가되면 자동으로 WebSession 으로 승격되어 저장됩니다. 다음 예제와 같이 다른 컨트롤러 메서드가 SessionStatus 메서드 인수를 사용하여 저장소를 지울 때까지 그대로 유지됩니다.

자바

```

@제어 장치
@SessionAttributes("펫") ①
공개 클래스 EditPetForm {

// ...

@PostMapping("/pets/{id}")
public String 핸들(애완동물, BindingResult 오류, SessionStatus 상태) { ②
    if (errors.hasErrors()) {
        // ...
    }
    상태.setComplete();
    // ...
}
}
}

```

① `@SessionAttributes` 주석을 사용합니다.

② `SessionStatus` 변수를 사용합니다.

코틀린

```

@제어 장치
@SessionAttributes("펫") ①
클래스 EditPetForm {

// ...

@PostMapping("/pets/{id}")
fun 핸들(pet: Pet, 오류: BindingResult, status: SessionStatus): String { ②
    if (errors.hasErrors()) {
        // ...
    }
    status.setComplete()
    // ...
}
}

```

① `@SessionAttributes` 주석을 사용합니다.

② `SessionStatus` 변수를 사용합니다.

@세션 속성

웹 MVC

전역적으로(즉, 외부에서 관리되는 기존 세션 속성에 액세스해야 하는 경우)
컨트롤러 — 예를 들어 필터에 의해) 존재하거나 존재하지 않을 수 있습니다.
다음 예제와 같이 메서드 매개변수에 대한 `@SessionAttribute` 주석:

자바

```
@GetMapping("/")
public String handle(@SessionAttribute 사용자 사용자) { ① // ... }
```

① **@SessionAttribute** 사용 .

코틀린

```
@GetMapping("/")
fun handle(@SessionAttribute 사용자: 사용자): String { ① // ... }
```

① **@SessionAttribute** 사용 .

세션 속성을 추가하거나 제거해야 하는 사용 사례의 경우 **WebSession** 을 컨트롤러 메서드에 주입하는 것을 고려하십시오.

컨트롤러 워크플로의 일부로 세션에 모델 속성을 임시로 저장하려면 **@SessionAttributes** 에 설명된 대로 **SessionAttributes**를 사용하는 것이 좋습니다.

@요청 속성

웹 MVC

@SessionAttribute와 유사하게 **@RequestAttribute** 주석을 사용하여 다음 예제와 같이 이전에 생성된 기존 요청 속성(예: **WebFilter**에 의해)에 액세스할 수 있습니다.

자바

```
@GetMapping("/")
public String 핸들(@RequestAttribute 클라이언트 클라이언트) { ① // ... }
```

① **@RequestAttribute** 사용 .

코틀린

```
@GetMapping("/")
재미 핸들(@RequestAttribute 클라이언트: 클라이언트): 문자열 { ① // ... }
```

① **@RequestAttribute** 사용 .

멀티파트 콘텐츠

웹 MVC

멀티파트 데이터에서 설명한 대로 `ServerWebExchange`는 멀티파트 콘텐츠에 대한 액세스를 제공합니다. 컨트롤러에서 파일 업로드 양식(예: 브라우저에서)을 처리하는 가장 좋은 방법은 다음 예제와 같이 명령 개체에 대한 데이터 바인딩을 사용하는 것입니다.

자바

```
클래스 마이폼 {
    개인 문자열 이름;
    개인 MultipartFile 파일;
    // ...
}

@Controller
공개 클래스 FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(MyForm 양식, BindingResult 오류) {
        // ...
    }
}
```

코틀린

```
클래스
    MyForm( val 이름: 문자
            열, val 파일: MultipartFile)

@Controller
클래스 FileUploadController {

    @PostMapping("/form") 재미
    핸들폼업로드(폼: MyForm, 오류: BindingResult): 문자열 {
        // ...
    }
}
```

RESTful 서비스 시나리오에서 브라우저가 아닌 클라이언트의 멀티파트 요청을 제출할 수도 있습니다. 다음 예제에서는 JSON과 함께 파일을 사용합니다.

```
POST /someUrl
```

콘텐츠 유형: 멀티파트/혼합

```
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
```

내용 처리: 양식 데이터; 이름 = "메타 데이터"

콘텐츠 유형: application/json; 문자 집합=UTF-8

콘텐츠 전송 인코딩: 8비트

```
{ "이름": "값" } --
```

```
edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp 콘
```

텐츠 처리: 양식 데이터; 이름 = "파일 데이터"; 파일명="파일.속성"

콘텐츠 유형: text/xml 콘텐츠

전송 인코딩: 8비트 ... 파일 데이터 ...

다음 예제와 같이 [@RequestPart](#)를 사용하여 개별 부품에 액세스할 수 있습니다.

자바

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") 파트 메타데이터, ①
                      @RequestPart("파일 데이터") FilePart 파일) { ② // ... }
```

① [@RequestPart](#) 를 사용 하여 메타데이터를 가져옵니다.

② [@RequestPart](#) 를 사용 하여 파일을 가져옵니다.

코틀린

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") 파트 메타데이터, ①
           @RequestPart("file-data") FilePart 파일): 문자열 { ② // ... }
```

① [@RequestPart](#) 를 사용 하여 메타데이터를 가져옵니다.

② [@RequestPart](#) 를 사용 하여 파일을 가져옵니다.

원시 부품 콘텐츠(예: JSON — [@RequestBody](#)와 유사) 를 역직렬화하려면 다음 예제와 같이 [Part](#) 대신 구체적인 대상 [객체](#) 를 선언할 수 있습니다.

자바

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") MetaData 메타데이터) { ① // ... }
```

① `@RequestPart` 를 사용 하여 메타데이터를 가져옵니다.

코틀린

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") 메타데이터: MetaData): String { ① // ... }
```

① `@RequestPart` 를 사용 하여 메타데이터를 가져옵니다.

`@RequestPart` 를 `javax.validation.Valid` 또는 Spring의 `@Validated` 주석과 함께 사용할 수 있습니다 . 그러면 표준 Bean 유효성 검사가 적용됩니다. 유효성 검사 오류로 인해 400(BAD_REQUEST) 응답이 발생 하는 `WebExchangeBindException` 이 발생합니다. 예외는 오류 세부 정보가 있는 `BindingResult` 를 포함하며 비동기 래퍼로 인수를 선언한 다음 오류 관련 연산자를 사용하여 컨트롤러 메서드에서 처리할 수도 있습니다.

자바

```
@PostMapping("/")
public String handle(@Valid @RequestPart("meta-data") Mono<MetaData> metadata) { //  
onError* 연산자 중 하나를 사용... }
```

코틀린

```
@PostMapping("/")
fun handle(@Valid @RequestPart("meta-data") 메타데이터: MetaData): String { // ... }
```

모든 멀티파트 데이터에 `MultiValueMap` 으로 액세스하려면 다음 예제와 같이 `@RequestBody` 를 사용할 수 있습니다 .

자바

```
@PostMapping("/")
public String handle(@RequestBody Mono<MultiValueMap<String, Part>> parts) { ① // ... }
```

① `@RequestBody` 사용 .

코틀린

```
@PostMapping("/")
재미 핸들(@RequestBody 부분: MultiValueMap<문자열, 부분>): 문자열 { ① // ... }
```

① `@RequestBody` 사용 .

멀티파트 데이터에 순차적으로 액세스하려면 스트리밍 방식으로 `@RequestBody` 를 다음 과 함께 사용할 수 있습니다 .

다음 예제와 같이 `Flux<Part>` (또는 Kotlin의 `Flow<Part>`) 대신

자바

```
@PostMapping("/")
public String handle(@RequestBody Flux<Part> parts) { ① // ... }
```

① `@RequestBody` 사용 .

코틀린

```
@PostMapping("/")
fun handle(@RequestBody 부분: Flow<Part>): String { ① // ... }
```

① `@RequestBody` 사용 .

`@RequestBody`

웹 MVC

`@RequestBody` 주석을 사용 하여 요청 본문을 읽고 `HttpMessageReader` 를 통해 `Object` 로 역직렬화하도록 할 수 있습니다. 다음 예제에서는 `@RequestBody` 인수를 사용합니다.

자바

```
@PostMapping("/accounts")
공개 무효 핸들(@RequestBody 계정 계정) { // ... }
```

코틀린

```
@PostMapping("/accounts")
fun handle(@RequestBody 계정: 계정) { // ... }
```

Spring MVC와 달리 WebFlux에서 **@RequestBody** 메소드 인수는 반응형과 완전 비차단 읽기 및 (클라이언트-서버) 스트리밍을 지원합니다.

자바

```
@PostMapping("/accounts")
public void 핸들(@RequestBody Mono<계정> 계정) { // ... }
```

코틀린

```
@PostMapping("/accounts")
fun handle(@RequestBody 계정: Flow<계정>) { // ... }
```

[WebFlux 구성](#) 의 [HTTP 메시지 코덱](#) 옵션을 사용하여 메시지 판독기를 구성하거나 사용자 지정할 수 있습니다.

@RequestBody 를 `javax.validation.Valid` 또는 Spring의 **@Validated** 주석과 함께 사용할 수 있습니다 . 그러면 표준 Bean 유효성 검사가 적용됩니다. 유효성 검사 오류로 인해 `WebExchangeBindException` 이 발생하여 400(BAD_REQUEST) 응답이 발생합니다. 예외에는 오류 세부 정보가 있는 `BindingResult` 가 포함되어 있으며 비동기 래퍼로 인수를 선언한 다음 오류 관련 연산자를 사용하여 컨트롤러 메서드에서 처리할 수 있습니다.

자바

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Mono<Account> account) { //
    onError* 연산자 중 하나를 사용... }
```

코틀린

```
@PostMapping("/accounts")
fun handle(@Valid @RequestBody 계정: Mono<계정>) { // ... }
```

HttpEntity

[웹 MVC](#)

HttpEntity 는 **@RequestBody** 를 사용하는 것과 거의 동일 하지만 요청 헤더와 본문을 노출하는 컨테이너 자체를 기반으로 합니다. 다음 예제에서는 **HttpEntity**를 사용합니다.

자바

```
@PostMapping("/계정") 공개
무효 핸들(HttpEntity<계정> 엔티티) { // ... }
```

코틀린

```
@PostMapping("/accounts")
fun handle(entity: HttpEntity<Account>) { // ... }
```

@ResponseBody

웹 MVC

메서드에 **@ResponseBody** 주석을 사용 하여 [HttpMessageWriter](#) 를 통해 응답 본문에 반환을 직렬화할 수 있습니다. 다음 예에서는 그렇게 하는 방법을 보여줍니다.

자바

```
@GetMapping("/accounts/{id}")
@ResponseBody
공개 계정 핸들() { // ... }
```

코틀린

```
@GetMapping("/accounts/{id}")
@ResponseBody
재미 핸들(): 계정 { // ... }
```

@ResponseBody 는 클래스 수준에서도 지원되며 이 경우 모든 컨트롤러 메서드에서 상속됩니다. 이것은 [@Controller](#) 및 [@ResponseBody](#) 로 표시된 메타 주석에 [불과한 @RestController](#) 의 효과입니다 .

@ResponseBody 는 반응형 유형을 지원합니다. 즉, Reactor 또는 RxJava 유형을 반환하고 생성하는 비동기 값이 응답에 렌더링되도록 할 수 있습니다. 자세한 내용은 [스트리밍](#) 및 [JSON 렌더링](#)을 참조하세요.

@ResponseBody 메소드를 JSON 직렬화 보기와 결합할 수 있습니다. 자세한 내용은 [Jackson JSON](#) 을 참조하세요.

[WebFlux Config](#) 의 [HTTP 메시지 코덱](#) 옵션을 사용 하여 구성하거나 사용자 정의할 수 있습니다 .

메시지 쓰기.

ResponseEntity

[웹 MVC](#)

ResponseEntity 는 @ResponseBody 와 비슷 하지만 상태와 헤더가 있습니다. 예를 들어:

자바

```
@GetMapping("/something")
public ResponseEntity<String> handle() { 문자열
    본문 = 문자열 etag = 반환;
    ResponseEntity.ok().eTag(etag).build(body); }
```

코틀린

```
@GetMapping("/something")
fun handle(): ResponseEntity<String> { val
    body: String = val etag: String = return
    ResponseEntity.ok().eTag(etag).build(body) }
```

WebFlux는 단일 값 반응 유형 을 사용하여 ResponseEntity 를 비동기적으로 생성하고/하거나 본체에 대해 단일 및 다중 값 반응 유형을 지원합니다. 이렇게 하면 다음과 같이 ResponseEntity 를 사용하여 다양한 비동기 응답이 가능 합니다.

- ResponseEntity<Mono<T>> 또는 ResponseEntity<Flux<T>> 는 응답 상태와 헤더를 즉시 알리고 본문은 나중에 비동기적으로 제공합니다. 본문이 0..1개의 값으로 구성된 경우 Mono 를 사용 하고 여러 값을 생성할 수 있는 경우 Flux 를 사용합니다.
- Mono<ResponseEntity<T>> 는 응답 상태, 헤더 및 본문의 세 가지를 모두 나중에 비동기적으로 제공합니다. 이를 통해 응답 상태와 헤더가 비동기식 요청 처리의 결과에 따라 달라질 수 있습니다.
- Mono<ResponseEntity<Mono<T>>> 또는 Mono<ResponseEntity<Flux<T>>> 는 덜 일반적인 대안이지만 또 다른 가능성입니다. 응답 상태와 헤더를 먼저 비동기식으로 제공한 다음 응답 본문을 비동기식으로 제공합니다.

잭슨 JSON

Spring은 Jackson JSON 라이브러리에 대한 지원을 제공합니다.

JSON 보기

[웹 MVC](#)

Spring WebFlux는 Jackson의 직렬화 뷰에 대한 내장 지원을 제공합니다 . **객체** 의 모든 필드의 하위 집합만 렌더링할 수 있습니다 . @ResponseBody 또는 ResponseEntity 컨트롤러 와 함께 사용하려면

메소드에서 다음 예제와 같이 Jackson의 [@JsonView](#) 주석을 사용하여 직렬화 보기 클래스를 활성화할 수 있습니다.

자바

```
@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() { return new User("eric", "7!
        jd#h23");
    }

    public class User {
        public String username;
        public String password;
    }

    public User() {}

    @JsonView(WithoutPasswordView.class)
    public String getUsername() { return
        this.username;
    }

    @JsonView(WithPasswordView.class)
    public String getPassword() { return
        this.password;
    }
}
```

코틀린

```

@RestController
클래스 사용자 컨트롤러 {

    @GetMapping("/사용자")
    @JsonView(User.WithoutPasswordView::class) fun
    getUser(): User { return User("eric", "7ljd#h23")

    } }

클래스 사용자(
    @JsonView(WithoutPasswordView::class) val 사용자 이름: 문자열,
    @JsonView(WithPasswordView::class) val 암호: 문자열
)
{ 인터페이스 WithoutPasswordView 인터페이
스 WithPasswordView : WithoutPasswordView }

```



@JsonView 는 뷰 클래스의 배열을 허용하지만 컨트롤러 메서드당 하나만 지정할 수 있습니다. 여러 보기의 활성화해야 하는 경우 복합 인터페이스를 사용하십시오.

1.4.4. 모델

웹 MVC

@ModelAttribute 주석을 사용할 수 있습니다.

- **@RequestMapping** 메소드의 **메소드 인수**에서 모델에서 객체를 생성하거나 액세스 **WebDataBinder**를 통해 요청에 바인딩합니다.
- **@Controller** 또는 **@ControllerAdvice** 클래스의 메서드 수준 주석으로 초기화를 돕습니다. **@RequestMapping** 메소드 호출 이전의 모델.
- **@RequestMapping** 메서드에서 반환 값을 모델 속성으로 표시합니다.

이 섹션에서는 **@ModelAttribute** 메서드 또는 이전 목록의 두 번째 항목에 대해 설명합니다. 컨트롤러에는 **@ModelAttribute** 메서드가 여러 개 있을 수 있습니다. 이러한 모든 메소드는 동일한 컨트롤러의 **@RequestMapping** 메소드 보다 먼저 호출됩니다. **@ModelAttribute** 메서드는 **@ControllerAdvice**를 통해 컨트롤러 간에 공유할 수도 있습니다. 자세한 내용은 [컨트롤러 조언](#) 섹션을 참조하십시오.

@ModelAttribute 메서드에는 유연한 메서드 서명이 있습니다. **@RequestMapping** 메서드와 동일한 인수를 많이 지원 합니다 (**@ModelAttribute** 자체 및 요청 본문과 관련된 것은 제외).

다음 예제에서는 **@ModelAttribute** 메서드를 사용합니다.

자바

```
@ModelAttribute
public void populateModel(@RequestParam 문자열 번호, 모델 모델)
{ model.addAttribute(accountRepository.findAccount(number)); // 더 추가 ... }
```

코틀린

```
@ModelAttribute
fun populateModel(@RequestParam number: String, model: Model)
{ model.addAttribute(accountRepository.findAccount(number)) // 추가 ... }
```

다음 예에서는 하나의 속성만 추가합니다.

자바

```
@ModelAttribute
공개 계정 addAccount(@RequestParam 문자열 번호) { return
accountRepository.findAccount(숫자); }
```

코틀린

```
@ModelAttribute
재미 addAccount(@RequestParam 번호: 문자열): 계정 { return
accountRepository.findAccount(숫자); }
```



이름이 명시적으로 지정되지 않은 경우 [규칙에 대한 javadoc](#)에 설명된 대로 유형에 따라 기본 이름이 선택됩니다.

오버로드된 `addAttribute` 메서드를 사용하거나 `@ModelAttribute` 의 `name` 속성을 통해 (반환 값의 경우)
항상 명시적 이름을 할당할 수 있습니다 .

Spring WebFlux는 Spring MVC와 달리 모델에서 명시적으로 반응 유형을 지원합니다(예: `Mono<Account>` 또는
`io.reactivex.Single<Account>`). 다음 예제와 같이 `@ModelAttribute` 인수가 래퍼 없이 선언 된 경우 이러한 비동기 모델 속성은
`@RequestMapping` 호출 시 실제 값으로 투명하게 확인(및 모델 업데이트)될 수 있습니다 .

자바

```

@ModelAttribute
public void addAccount(@RequestParam 문자열 번호) { Mono<계정>
    > accountMono = accountRepository.findAccount(숫자); model.addAttribute("계
    정", accountMono); }

@PostMapping("/accounts")
public 문자열 핸들(@ModelAttribute 계정 계정, BindingResult 오류) { // ... }

```

코틀린

```

org.springframework.ui.set 가져오기

@ModelAttribute
fun addAccount(@RequestParam 번호: 문자열) { val
accountMono: Mono<계정> = accountRepository.findAccount(숫자) 모델["계정"] =
accountMono }

@PostMapping("/accounts")
재미 핸들(@ModelAttribute 계정: 계정, 오류: BindingResult): 문자열 { // ... }

```

또한 반응형 래퍼가 있는 모든 모델 속성은 뷰 렌더링 직전에 실제 값으로 해석되고 모델이 업데이트됩니다.

@ModelAttribute 를 **@RequestMapping** 메서드에서 메서드 수준 주석으로 사용할 수도 있습니다 . 이 경우 **@RequestMapping** 메서드의 반환 값은 모델 속성으로 해석됩니다. 반환 값이 보기 이름으로 해석되는 **문자열** 이 아닌 한 HTML 컨트롤러의 기본 동작이므로 일반적으로 필요하지 않습니다 . **@ModelAttribute** 는 다음 예제와 같이 모델 속성 이름을 사용자 지정하는 데도 도움이 될 수 있습니다.

자바

```

@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
public Account handle() { // ... 계
정 반환; }

```

코틀린

```
@GetMapping("/accounts/{id}")
@ResponseBody("myAccount") 재미 핸
들(): 계정 { // ...
    계정 반환
}
```

1.4.5. 데이터 바인더

웹 MVC

`@Controller` 또는 `@ControllerAdvice` 클래스는 `WebDataBinder`의 인스턴스를 초기화하기 위해 `@InitBinder` 메소드를 가질 수 있습니다. 이들은 차례로 다음을 수행하는 데 사용됩니다.

- 요청 매개변수(즉, 양식 데이터 또는 쿼리)를 모델 개체에 바인딩합니다.
- 문자열 기반 요청 값(요청 매개변수, 경로 변수, 헤더, 쿠키 등)을 컨트롤러 메서드 인수의 대상 유형으로 변환합니다.
- HTML 양식을 렌더링할 때 모델 개체 값의 형식을 문자열 값으로 지정합니다.

`@InitBinder` 메소드는 컨트롤러별 `java.beans.PropertyEditor` 또는 Spring `Converter` 및 `Formatter` 구성요소를 등록할 수 있습니다. 또한 [WebFlux Java 구성](#)을 사용하여 전 세계적으로 공유되는 `FormattingConversionService`에 `Converter` 및 `Formatter` 유형을 등록 할 수 있습니다.

`@InitBinder` 메서드는 `@ModelAttribute` (명령 개체) 인수를 제외하고 `@RequestMapping` 메서드와 동일한 많은 인수를 지원합니다. 일반적으로 등록을 위해 `WebDataBinder` 인수와 `void` 반환 값으로 선언됩니다. 다음 예제에서는 `@InitBinder` 주석을 사용합니다.

자바

```
@Controller
public class FormController {

    @InitBinder ① public
    void initBinder(WebDataBinder 바인더) { SimpleDateFormat
        dateFormat = new SimpleDateFormat("yyyy-MM-dd"); dateFormat.setLenient(거
       짓); 바인더.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat,
        false)); }

    // ...
}
```

① `@InitBinder` 주석 사용 .

코틀린

```
@Controller
클래스 FormController {

    @InitBinder ① 재미
    initBinder(바인더: WebDataBinder) { val dateFormat
        = SimpleDateFormat("yyyy-MM-dd") dateFormat.isLenient
        = false 바인더.registerCustomEditor(Date::class.java,
        CustomDateEditor(dateFormat,
        거짓)) }

    // ...
}
```

또는 공유 [FormattingConversionService](#) 를 통해 [포맷터 기반](#) 설정을 사용할 때 다음 예제와 같이 동일한 접근 방식을 재사용하고 컨트롤러별 [포맷터](#) 인스턴스를 등록할 수 있습니다.

자바

```
@Controller
공개 클래스 FormController {

    @InitBinder 보호
    무효 initBinder(WebDataBinder 바인더) { 바인
        더.addCustomFormatter(new DateFormatter("yyyy-MM-dd")); ①
    }

    // ...
}
```

① 사용자 지정 포맷터(이 경우 [DateFormatter](#)) 를 추가합니다.

코틀린

```
@Controller
클래스 FormController {

    @InitBinder 재미
    initBinder(바인더: WebDataBinder) { 바인
        더.addCustomFormatter(DateFormatter("yyyy-MM-dd")) ①
    }

    // ...
}
```

① 사용자 지정 포맷터(이 경우 [DateFormatter](#)) 를 추가합니다.

1.4.6. 예외 관리

웹 MVC

@Controller 및 @ControllerAdvice 클래스는 컨트롤러 메서드의 예외를 처리하기 위해 @ExceptionHandler 메서드를 가질 수 있습니다. 다음 예제에는 이러한 처리기 메서드가 포함되어 있습니다.

자바

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler(①)
    public ResponseEntity<String> handle(IOException ex) { // ...
        ...
    }
}
```

① @ExceptionHandler 선언.

코틀린

```
@Controller
class SimpleController {

    // ...

    @ExceptionHandler(①) 재미
    있는 핸들(ex: IOException): ResponseEntity<String> { // ...
        ...
    }
}
```

① @ExceptionHandler 선언.

예외는 전파되는 최상위 예외(즉, 직접 IOException 발생) 또는 최상위 래퍼 예외 내의 즉각적인 원인(예: IllegalStateException 내부에 래핑된 IOException)과 일치할 수 있습니다.

일치하는 예외 유형의 경우 앞의 예와 같이 대상 예외를 메서드 인수로 선언하는 것이 좋습니다. 또는 주석 선언은 일치하도록 예외 유형을 줍힐 수 있습니다. 일반적으로 인수 서명에서 가능한 한 구체적이고 해당 순서로 우선 순위가 지정된 @ControllerAdvice 에 대한 기본 루트 예외 매핑을 선언하는 것이 좋습니다. [MVC 섹션](#) 참조 자세한 내용은.



WebFlux의 @ExceptionHandler 메서드는 요청 본문 및 @ModelAttribute 관련 메서드 인수를 제외하고 @RequestMapping 메서드와 동일한 메서드 인수 및 반환 값을 지원합니다.

Spring WebFlux에서 [@ExceptionHandler](#) 메소드에 대한 지원은 [@RequestMapping](#) 메소드 용 [HandlerAdapter](#)에 의해 제공됩니다 . 자세한 내용은 [DispatcherHandler](#)를 참조하세요.

REST API 예외

웹 MVC

REST 서비스에 대한 일반적인 요구 사항은 응답 본문에 오류 세부 정보를 포함하는 것입니다.

Spring Framework는 자동으로 그렇게 하지 않습니다. 응답 본문의 오류 세부 정보 표현은 애플리케이션에 따라 다르기 때문입니다. 그러나 [@RestController](#)는 [ResponseEntity](#) 반환 값과 함께 [@ExceptionHandler](#) 메서드를 사용하여 응답의 상태와 본문을 설정할 수 있습니다. 이러한 메서드는 [@ControllerAdvice](#) 클래스에서 선언하여 전역적으로 적용할 수도 있습니다 .



Spring WebFlux에는 Spring MVC [ResponseEntityExceptionHandler](#)에 해당하는 것이 없습니다. [ResponseStatusException](#) (또는 그 하위 클래스) 만 있고 HTTP 상태 코드로 변환할 필요가 없기 때문입니다. 웹플러스 인상

1.4.7. 컨트롤러 조언

웹 MVC

일반적으로 [@ExceptionHandler](#), [@InitBinder](#) 및 [@ModelAttribute](#) 메서드는 선언된 [@Controller](#) 클래스(또는 클래스 계층) 내에서 적용됩니다. 이러한 메서드가 더 전역적으로(컨트롤러 전반에 걸쳐) 적용되도록 하려면 [@ControllerAdvice](#) 또는 [@RestControllerAdvice](#) 주석이 달린 클래스에서 선언할 수 있습니다 .

[@ControllerAdvice](#)는 [@Component](#)로 주석이 달려 있는데, 이는 [컴포넌트 스캐닝](#)을 통해 이러한 클래스를 Spring Bean으로 등록할 수 있음을 의미합니다 . [@RestControllerAdvice](#)는 [@ControllerAdvice](#)와 [@ResponseBody](#) 모두 주석이 달린 합성된 주석으로 , 본질적으로 [@ExceptionHandler](#) 메소드가 메시지 변환을 통해 응답 본문에 렌더링된다는 것을 의미합니다(보기 확인 또는 템플릿 렌더링에 비해).

시작 시 [@RequestMapping](#) 및 [@ExceptionHandler](#) 메서드에 대한 인프라 클래스는 [@ControllerAdvice](#)로 주석이 달린 Spring 빈을 감지 한 다음 런타임에 해당 메서드를 적용합니다. [@ControllerAdvice](#)의 전역 [@ExceptionHandler](#) 메서드는 [@Controller](#)의 로컬 메서드 다음에 적용됩니다 . 대조적으로 전역 [@ModelAttribute](#) 및 [@InitBinder](#) 메서드는 로컬 보다 먼저 적용됩니다.

것.

기본적으로 [@ControllerAdvice](#) 메서드는 모든 요청(즉, 모든 컨트롤러)에 적용되지만 다음 예제와 같이 주석의 속성을 사용하여 컨트롤러의 하위 집합으로 범위를 좁힐 수 있습니다.

자바

```
// @RestController로 주석이 달린 모든 컨트롤러를 대상으로 지정
@ControllerAdvice(annotations = RestController.class) public
class ExampleAdvice1 {}

// 특정 패키지 내의 모든 컨트롤러 대상
@ControllerAdvice("org.example.controllers") public class
ExampleAdvice2 {}

// 특정 클래스에 할당 가능한 모든 컨트롤러 대상
@ControllerAdvice(assignableTypes = {ControllerInterface.class,
AbstractController.class}) public class ExampleAdvice3 {}
```

코틀린

```
// @RestController로 주석이 달린 모든 컨트롤러를 대상으로 합니다.
@ControllerAdvice(주석 = [RestController::class]) 공개 클래스
ExampleAdvice1 {}

// 특정 패키지 내의 모든 컨트롤러 대상
@ControllerAdvice("org.example.controllers") public class
ExampleAdvice2 {}

// 특정 클래스에 할당 가능한 모든 컨트롤러 대상
@ControllerAdvice(assignableTypes = [ControllerInterface::class,
AbstractController::class]) 공개
클래스 ExampleAdvice3 {}
```

앞의 예에서 선택기는 런타임에 평가되며 광범위하게 사용되는 경우 성능에 부정적인 영향을 미칠 수 있습니다. [@ControllerAdvice](#) 참조 자세한 내용은 [javadoc](#)을 참조하십시오.

1.5. 기능적 끝점

[웹 MVC](#)

Spring WebFlux에는 요청을 라우팅하고 처리하는 데 함수가 사용되는 경량 기능 프로그래밍 모델인 `WebFlux.fn`이 포함되어 있으며 계약은 불변성을 위해 설계되었습니다. 주석 기반 프로그래밍 모델의 대안이지만 동일한 [Reactive Core](#) 기반에서 실행됩니다.

1.5.1. 개요

[웹 MVC](#)

`WebFlux.fn`에서 HTTP 요청은 [HandlerFunction](#)으로 처리됩니다. 이 함수는 `ServerRequest`를 받고 지연된 `ServerResponse` (즉, `Mono<ServerResponse>`)를 반환합니다. 요청과 응답 객체 모두 HTTP 요청에 대한 JDK 8 친화적인 액세스를 제공하는 변경 불가능한 계약을 가지고 있습니다.

그리고 응답. **HandlerFunction** 은 어노테이션 기반 프로그래밍 모델에서 **@RequestMapping** 메소드의 바디에 해당합니다.

들어오는 요청은 **RouterFunction** 이 있는 핸들러 함수로 라우팅됩니다. 이 함수는 **ServerRequest** 를 사용하고 지연된 **HandlerFunction** (즉, **Mono<HandlerFunction>**) 을 반환하는 함수입니다. 라우터 함수가 일치하면 핸들러 함수가 반환됩니다. 그렇지 않으면 빈 모노입니다. **RouterFunction** 은 **@RequestMapping** 주석과 동일하지만 라우터 기능이 데이터뿐만 아니라 동작도 제공한다는 주요 차이점이 있습니다.

RouterFunctions.route() 는 다음 예제와 같이 라우터 생성을 용이하게 하는 라우터 빌더를 제공합니다.

자바

```
정적 org.springframework.http.MediaType.APPLICATION_JSON 가져오기; 가져오기 정적 org.springframework.web.reactive.function.server.RequestPredicates.*; 정적 org.springframework.web.reactive.function.server.RouterFunctions.route 가져오기;
```

```
PersonRepository 저장소 = ...
PersonHandler 핸들러 = new PersonHandler(저장소);
```

```
RouterFunction<ServerResponse> 경로 = 경로()
    .GET("/사람/{id}", 수락(APPLICATION_JSON), 처리기::getPerson)
    .GET("/사람", 수락(APPLICATION_JSON), 처리기::listPeople)
    .POST("/사람", 핸들러::createPerson) .build();
```

```
공개 클래스 PersonHandler {
```

```
// ...
```

```
공개 Mono<ServerResponse> listPeople(ServerRequest 요청) {
    // ...
}
```

```
공개 Mono<ServerResponse> createPerson(ServerRequest 요청) {
    // ...
}
```

```
공개 Mono<ServerResponse> getPerson(ServerRequest 요청) {
    // ...
}}
```

코틀린

```
val 저장소: PersonRepository = val 처리기 ...  
= PersonHandler(저장소)
```

```
val route = coRouter { ①  
accept(APPLICATION_JSON).nest { GET("/  
    person/{id}", 핸들러::getPerson)  
    GET("/사람", 핸들러::listPeople)  
}  
POST("/사람", 핸들러::createPerson) }
```

```
클래스 PersonHandler(개인 저장소: PersonRepository) {
```

```
// ...
```

```
suspend fun listPeople(요청: ServerRequest): ServerResponse {  
    // ...  
}
```

```
일시 중단 재미 createPerson(요청: ServerRequest): ServerResponse {  
    // ...  
}
```

```
일시 중단 재미 getPerson(요청: ServerRequest): ServerResponse {  
    // ...  
}}
```

① 코루틴 라우터 DSL을 사용하여 라우터를 생성합니다. [라우터 {}](#)를 통해 Reactive 대안도 사용할 수 있습니다 .

[RouterFunction](#) 을 실행하는 한 가지 방법은 이를 [HttpHandler](#)로 변환하고 내장 [서버 어댑터](#) 중 하나를 통해 설치하는 것입니다.

- [RouterFunctions.toHttpHandler\(RouterFunction\)](#)
- [RouterFunctions.toHttpHandler\(RouterFunction, HandlerStrategies\)](#)

대부분의 응용 프로그램은 WebFlux Java 구성을 통해 실행할 수 있습니다 . [서버 실행을 참조하십시오.](#)

1.5.2. 핸들러 함수

[웹 MVC](#)

[ServerRequest](#) 및 [ServerResponse](#) 는 HTTP 요청 및 응답에 대한 JDK 8 친화적인 액세스를 제공하는 변경할 수 없는 인터페이스입니다. 요청과 응답 모두 [Reactive Streams](#) 를 제공합니다. 신체 흐름에 대한 배압. 요청 본문은 [Reactor Flux](#) 또는 [Mono](#)로 표시됩니다. 응답 본문은 [Flux](#) 및 [Mono](#) 를 포함한 모든 Reactive Streams [Publisher](#) 로 표시됩니다. 이에 대한 자세한 내용은 [반응형 라이브러리를 참조하세요.](#)

서버요청

ServerRequest 는 HTTP 메서드, URI, 헤더 및 쿼리 매개변수에 대한 액세스를 제공하는 반면 본문에 대한 액세스는 **본문** 메서드를 통해 제공됩니다.

다음 예제는 요청 본문을 **Mono<String>**으로 추출합니다.

자바

```
Mono<String> 문자열 = request.bodyToMono(String.class);
```

코틀린

```
값 문자열 = 요청.awaitBody<문자열>()
```

다음 예제는 본문을 **Flux<Person>** (또는 Kotlin 의 **Flow<Person>**)으로 추출합니다. 여기서 **Person** 객체는 JSON 또는 XML과 같은 직렬화된 형식에서 디코딩됩니다.

자바

```
Flux<Person> 사람 = request.bodyToFlux(Person.class);
```

코틀린

```
발 사람 = request.bodyToFlow<사람>()
```

앞의 예는 **BodyExtractor** 기능 전략 인터페이스를 하용하는 보다 일반적인 **ServerRequest.body(BodyExtractor)**입니다.

유ти리티 클래스 **BodyExtractors** 는 여러 인스턴스에 대한 액세스를 제공합니다. 예를 들어 앞의 예제는 다음과 같이 작성할 수도 있습니다.

자바

```
Mono<String> 문자열 = request.body(BodyExtractors.toMono(String.class));
Flux<Person> 사람 = request.body(BodyExtractors.toFlux(Person.class));
```

코틀린

```
val 문자열 = request.body(BodyExtractors.toMono(String::class.java)).awaitSingle()
val 사람 = request.body(BodyExtractors.toFlux(Person::class.java)).asFlow()
```

다음 예에서는 양식 데이터에 액세스하는 방법을 보여줍니다.

자바

```
Mono<MultiValueMap<문자열, 문자열>> 맵 = request.formData();
```

코틀린

```
발 맵 = request.awaitFormData()
```

다음 예는 멀티파트 데이터에 맵으로 액세스하는 방법을 보여줍니다.

자바

```
Mono<MultiValueMap<문자열, 파트>> 맵 = request.multipartData();
```

코틀린

```
발 맵 = request.awaitMultipartData()
```

다음 예는 스트리밍 방식으로 한 번에 하나씩 멀티파트에 액세스하는 방법을 보여줍니다.

자바

```
Flux<Part> 부품 = request.body(BodyExtractors.toParts());
```

코틀린

```
val 부품 = request.body(BodyExtractors.toParts()).asFlow()
```

서버 응답

ServerResponse 는 HTTP 응답에 대한 액세스를 제공하며 변경할 수 없으므로 **빌드** 메서드를 사용하여 생성할 수 있습니다. 빌더를 사용하여 응답 상태를 설정하거나 응답 헤더를 추가하거나 본문을 제공할 수 있습니다. 다음 예에서는 JSON 콘텐츠가 포함된 200(OK) 응답을 생성합니다.

자바

```
Mono<Person> 사람 = ...  
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(사람,  
Person.class);
```

코틀린

```
val 사람: 사람 = ...  
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyValue(사람)
```

다음 예는 **Location** 헤더가 있고 본문이 없는 201(CREATED) 응답을 작성하는 방법을 보여줍니다.

자바

```
URI 위치 = ...
ServerResponse.created(위치).build();
```

코틀린

```
값 위치: URI =
...
ServerResponse.created(location).build()
```

사용된 코덱에 따라 힌트 매개변수를 전달하여 본문을 직렬화 또는 역직렬화하는 방법을 사용자 지정할 수 있습니다. 예를 들어, [Jackson JSON 보기](#)를 지정하려면:

자바

```
ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT,
MyJacksonView.class).body(...);
```

코틀린

```
ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT,
MyJacksonView::class.java).body(...)
```

핸들러 클래스

다음 예제와 같이 핸들러 함수를 람다로 작성할 수 있습니다.

자바

```
HandlerFunction<ServerResponse> helloWorld = 요
청 -> ServerResponse.ok().bodyValue("Hello World");
```

코틀린

```
val helloWorld = HandlerFunction<ServerResponse>
{ ServerResponse.ok().bodyValue("Hello World") }
```

그것은 편리하지만 응용 프로그램에서 여러 기능이 필요하고 여러 인라인 람다가 지저분해질 수 있습니다. 따라서 주석 기반 응용 프로그램에서 **@Controller** 와 유사한 역할을 하는 핸들러 클래스로 관련 핸들러 기능을 그룹화하는 것이 유용합니다. 예를 들어 다음 클래스는 반응적인 **Person** 리포지토리를 노출합니다.

자바

```
정적 org.springframework.http.MediaType.APPLICATION_JSON 가져오기;
정적 org.springframework.web.reactive.function.server.ServerResponse.ok 가져오기;

공개 클래스 PersonHandler {

    개인 최종 PersonRepository 저장소;

    공개 PersonHandler(PersonRepository 저장소) {
        this.repository = 저장소;
    }

    public Mono<ServerResponse> listPeople(ServerRequest 요청) { ①
        Flux<Person> people = repository.allPeople();
        반환 ok().contentType(APPLICATION_JSON).body(사람, Person.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest 요청) { ②
        Mono<Person> 사람 = request.bodyToMono(Person.class);
        반환 ok().build(repository.savePerson(사람));
    }

    public Mono<ServerResponse> getPerson(ServerRequest 요청) { ③
        int personId = Integer.valueOf(request.pathVariable("id"));
        반환 repository.getPerson(personId)
            .flatMap(사람 -> ok().contentType(APPLICATION_JSON).bodyValue(사람))
            .switchIfEmpty(ServerResponse.notFound().build());
    }
}
```

① `listPeople` 은 저장소에 있는 모든 `Person` 객체를 JSON으로 반환하는 핸들러 함수이다.

② `createPerson` 은 요청 본문에 포함된 새로운 `Person` 을 저장하는 핸들러 함수입니다. 메모

`PersonRepository.savePerson(Person)` 은 `Mono <Void>` 를 반환 합니다.

사람이 요청에서 읽고 저장되면 완료 신호. 그래서 우리는 사용

해당 완료 신호가 수신될 때 응답을 보내는 `build(Publisher<Void>)` 메서드(

즉, `사람` 이 저장되었을 때).

③ `getPerson` 은 `id` 경로 변수로 식별되는 한 사람을 반환하는 핸들러 함수입니다.

저장소에서 해당 `Person` 을 검색하고 찾은 경우 JSON 응답을 생성합니다. 그렇지 않은 경우

발견되면 `switchIfEmpty(Mono<T>)` 를 사용하여 404 Not Found 응답을 반환합니다.

코틀린

```

클래스 PersonHandler(개인 저장소: PersonRepository) {

    suspend fun listPeople(요청: ServerRequest): ServerResponse { ①
        val people: Flow<Person> = repository.allPeople()
        반환 ok().contentType(APPLICATION_JSON).bodyAndAwait(사람);
    }

    suspend fun createPerson(요청: ServerRequest): ServerResponse { ②
        발 사람 = request.awaitBody<사람>()
        repository.savePerson(사람)
        ok().buildAndAwait() 반환
    }

    suspend fun getPerson(요청: ServerRequest): ServerResponse { ③
        발 personId = request.pathVariable("id").toInt()
        반환 repository.getPerson(personId)?.let {
            ok().contentType(APPLICATION_JSON).bodyValueAndAwait(it) }
            ?: ServerResponse.notFound().buildAndAwait()
        }
    }
}

```

① `listPeople` 은 저장소에 있는 모든 `Person` 객체를 JSON으로 반환하는 핸들러 함수이다 .

② `createPerson` 은 요청 본문에 포함된 새로운 `Person` 을 저장하는 핸들러 함수입니다 . 메모
`PersonRepository.savePerson (Person)` 은 반환 유형이 없는 일시 중단 함수입니다.

③ `getPerson` 은 `id` 경로 변수로 식별되는 한 사람을 반환하는 핸들러 함수입니다 .
저장소에서 해당 `Person` 을 검색하고 찾은 경우 JSON 응답을 생성합니다. 그렇지 않은 경우
발견되면 404 Not Found 응답을 반환합니다.

확인

기능적 끝점은 Spring의 [유효성 검사](#) 기능을 사용할 수 있습니다. 요청 본문에 유효성 검사를 적용합니다.
예를 들어 사용자 정의 Spring [Validator](#) 가 주어지면 사람에 대한 구현 :

자바

```

    공개 클래스 PersonHandler {
        개인 최종 유효성 검사기 = new PersonValidator(); ①
        // ...

        공개 Mono<ServerResponse> createPerson(ServerRequest 요청) {
            모노 <사람> 사람 =
            request.bodyToMono(Person.class).doOnNext(this::validate); ②
                반환 ok().build(repository.savePerson(사람));
        }

        private void validate(개인 명) {
            오류 오류 = new BeanPropertyBindingResult(사람, "사람");
            validator.validate(사람, 오류);
            if (errors.hasErrors()) {
                새로운 ServerWebInputException(errors.toString()) 던지기; ③
            }
        }
    }

```

① Validator 인스턴스를 생성 합니다.

② 유효성 검사를 적용합니다.

③ 400 응답에 대해 예외를 발생시킵니다.

코틀린

```

클래스 PersonHandler(개인 저장소: PersonRepository) {

    private val validator = PersonValidator() ①

    // ...

    suspend fun createPerson(요청: ServerRequest): ServerResponse { val person =
        request.awaitBody<Person>() validate(person) ②
        repository.savePerson(person) return ok().buildAndAwait()

    }

    private fun validate(사람: 사람) {
        val 오류: 오류 = BeanPropertyBindingResult(사람, "사람"); validator.validate(사
        람, 오류); if (errors.hasErrors()) {

            ServerWebInputException(errors.toString()) 던지기 ③
        }
    }
}

```

① Validator 인스턴스를 생성 합니다.

② 유효성 검사를 적용합니다.

③ 400 응답에 대해 예외를 발생시킵니다.

핸들러는 LocalValidatorFactoryBean 을 기반으로 하는 전역 Validator 인스턴스 를 생성하고 주입하여 표준 빈 유효성 검사 API(JSR-303)를 사용할 수도 있습니다 . [스프링 유효성 검사를](#) 참조하십시오 .

1.5.3. 라우터 기능

[웹 MVC](#)

라우터 기능은 요청을 해당 HandlerFunction으로 라우팅하는 데 사용됩니다. 일반적으로 라우터 기능을 직접 작성하지 않고 RouterFunctions 유ти리티 클래스의 메서드를 사용하여 만듭니다. RouterFunctions.route() (매개변수 없음)는 라우터 기능을 생성하기 위한 유창한 빌더를 제공하는 반면, RouterFunctions.route(RequestPredicate, HandlerFunction) 는 라우터를 생성하는 직접적인 방법을 제공합니다.

일반적으로 route() 빌더는 찾기 어려운 정적 가져오기 없이 일반적인 매핑 시나리오에 대한 편리한 바로 가기를 제공하므로 사용하는 것이 좋습니다 . 예를 들어, 라우터 기능 빌더는 GET 요청에 대한 매핑을 생성하기 위해 [GET\(String, HandlerFunction\)](#) 메서드를 제 공합니다. 및 [POST에 대한 POST\(String, HandlerFunction\)](#) .

HTTP 메서드 기반 매핑 외에도 라우트 빌더는 요청에 매핑할 때 추가 조건자를 도입하는 방법을 제공합니다. 각 HTTP 메서드에는 RequestPredicate 를 매개변수로 사용하는 오버로드된 변형이 있지만 추가 제약 조건을 표현할 수 있습니다.

술어

자신만의 `RequestPredicate` 를 작성할 수 있지만 `RequestPredicates` 유ти리티 클래스는 요청 경로, HTTP 메서드, 콘텐츠 유형 등을 기반으로 일반적으로 사용되는 구현을 제공합니다. 다음 예제에서는 요청 조건자를 사용하여 `Accept` 헤더를 기반으로 제약 조건을 생성합니다.

자바

```
RouterFunction<ServerResponse> 경로 = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN), 요청 ->
        ServerResponse.ok().bodyValue("Hello World")).build();
```

코틀린

```
val route = coRouter
{ GET("/hello-world", accept(TEXT_PLAIN))
    { ServerResponse.ok().bodyValueAndAwait("Hello World") }
}
```

다음을 사용하여 여러 요청 술어를 함께 작성할 수 있습니다.

- `RequestPredicate.and(RequestPredicate)` — 둘 다 일치해야 합니다.
- `RequestPredicate.or(RequestPredicate)` — 둘 중 하나가 일치할 수 있습니다.

`RequestPredicates` 의 많은 술어가 구성됩니다. 예를 들어 `RequestPredicates.GET(String)` 은 `RequestPredicates.method(HttpMethod)` 와 `RequestPredicates.path(String)`로 구성된다. 빌더가 내부적으로 `RequestPredicates.GET` 을 사용하고 이를 `수락` 조건부로 구성 하므로 위에 표시된 예에서도 두 개의 요청 조건부를 사용합니다.

경로

라우터 기능은 순서대로 평가됩니다. 첫 번째 경로가 일치하지 않으면 두 번째 경로가 평가되는 식입니다. 따라서 일반적인 경로보다 더 구체적인 경로를 선언하는 것이 좋습니다. 이것은 나중에 설명하겠지만 라우터 기능을 Spring Bean으로 등록할 때도 중요합니다. 이 동작은 "가장 구체적인" 컨트롤러 메서드가 자동으로 선택되는 주석 기반 프로그래밍 모델과 다릅니다.

라우터 함수 빌더를 사용할 때 정의된 모든 경로는 `build()` 에서 반환되는 하나의 `RouterFunction` 으로 구성됩니다. 여러 라우터 기능을 함께 구성하는 다른 방법도 있습니다.

- `RouterFunctions.route()` 빌더에서 `add(RouterFunction)`
- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)` — 바로 가기
중첩된 `RouterFunctions.route()` 가 있는 `RouterFunction.and()`.

~을위한

다음 예는 4가지 경로의 구성을 보여줍니다.

자바

```
정적 org.springframework.http.MediaType.APPLICATION_JSON 가져오기; 가져오
기 정적 org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository 저장소 = ...
PersonHandler 핸들러 = new PersonHandler(저장소);

RouterFunction<ServerResponse> otherRoute = ...

RouterFunction<ServerResponse> 경로 = 경로()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ① .GET("/")
    person", accept(APPLICATION_JSON), handler::listPeople) ②
    .POST("/사람", 핸들러::createPerson) ③ .add(otherRoute)
    ④ .build();
```

- ① JSON과 일치 하는 Accept 헤더가 있는 GET /person/{id} 는 PersonHandler.getPerson 으로 라우팅됩니다.
- ② JSON과 일치 하는 Accept 헤더가 있는 GET /person 은 PersonHandler.listPeople 로 라우팅됩니다.
- ③ 추가 술어가 없는 POST /person 은 PersonHandler.createPerson에 매핑되고 ,
- ④ otherRoute 는 다른 곳에서 생성되어 빌드된 경로에 추가되는 라우터 기능입니다.

코틀린

```
org.springframework.http.MediaType.APPLICATION_JSON 가져오기

val 저장소: PersonRepository = val 처리기 ...
    = PersonHandler(저장소);

val otherRoute: RouterFunction<ServerResponse> = coRouter { }

val route = coRouter
    { GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ① GET("/")
    person", accept(APPLICATION_JSON), handler::listPeople) ②
    POST("/사람", 핸들러::createPerson) ③ }.and(otherRoute)
    ④
```

- ① JSON과 일치 하는 Accept 헤더가 있는 GET /person/{id} 는 PersonHandler.getPerson 으로 라우팅됩니다.
- ② JSON과 일치 하는 Accept 헤더가 있는 GET /person 은 PersonHandler.listPeople 로 라우팅됩니다.
- ③ 추가 술어가 없는 POST /person 은 PersonHandler.createPerson에 매핑되고 ,
- ④ otherRoute 는 다른 곳에서 생성되어 빌드된 경로에 추가되는 라우터 기능입니다.

중첩 경로

라우터 기능 그룹이 공유 경로와 같은 공유 술어를 갖는 것은 일반적입니다.

위의 예에서 공유 조건 자는 3개의 경로 [에서 사용되는 /person](#) 과 일치하는 경로 조건자가 됩니다. 주석을 사용할 때 /person 에 매핑 되는 유형 수준 [@RequestMapping](#) 주석을 사용하여 이 중복을 제거합니다. WebFlux.fn에서 경로 술어는 다음과 같을 수 있습니다.

라우터 기능 빌더의 경로 방법을 통해 공유됩니다 . 예를 들어, 마지막 몇 줄은 위의 예는 중첩 경로를 사용하여 다음과 같이 개선할 수 있습니다.

자바

```
RouterFunction<ServerResponse> 경로 = 경로()
    .path("/사람", builder -> builder ①
        .GET("/{id}", 수락(APPLICATION_JSON), 처리기::getPerson)
        .GET(수락(APPLICATION_JSON), 핸들러::listPeople)
        .POST("/사람", 핸들러::createPerson))
    .짓다();
```

① 경로 의 두 번째 매개변수 는 라우터 빌더를 사용하는 소비자입니다.

코틀린

```
발 경로 = coRouter {
    "/사람".nest {
        GET("/{id}", 수락(APPLICATION_JSON), 핸들러::getPerson)
        GET(수락(APPLICATION_JSON), 핸들러::listPeople)
        POST("/사람", 핸들러::createPerson)
    }
}
```

경로 기본 중첩이 가장 일반적이지만 다음을 사용하여 모든 종류의 술어에 중첩할 수 있습니다.

빌더의 nest 메소드. 위의 내용에는 여전히 공유 형식의 일부 중복 항목이 포함되어 있습니다.

수락 헤더 술어. 다음과 같이 accept 와 함께 nest 메소드 를 사용하여 더 개선할 수 있습니다 .

자바

```
RouterFunction<ServerResponse> 경로 = 경로()
    .path("/사람", b1 -> b1
        .nest(수락(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", 핸들러::getPerson)
            .GET(핸들러::listPeople)
            .POST("/사람", 핸들러::createPerson))
    .짓다();
```

코틀린

```
발 경로 = coRouter {
    "/사람".nest {
        수락(APPLICATION_JSON).nest {
            GET("/{id}", 핸들러::getPerson)
            GET(핸들러::listPeople)
            POST("/사람", 핸들러::createPerson)
        }
    }
}
```

1.5.4. 서버 실행

웹 MVC

HTTP 서버에서 라우터 기능을 어떻게 실행합니까? 간단한 옵션은 다음 중 하나를 사용하여 라우터 기능을 [HttpHandler](#)로 변환하는 것입니다.

- [RouterFunctions.toHttpHandler\(RouterFunction\)](#)
- [RouterFunctions.toHttpHandler\(RouterFunction, HandlerStrategies\)](#)

그런 다음 서버별 지침에 대한 [HttpHandler](#)에 따라 여러 서버 어댑터와 함께 반환된 [HttpHandler](#)를 사용할 수 있습니다.

Spring Boot에서도 사용되는 보다 일반적인 옵션은 Spring 구성을 사용하여 요청을 처리하는 데 필요한 구성 요소를 선언하는 [WebFlux 구성](#)을 통해 [DispatcherHandler 기본](#) 설정으로 실행하는 것입니다. WebFlux Java 구성은 기능적 끝점을 지원하기 위해 다음 인프라 구성 요소를 선언합니다.

- [RouterFunctionMapping](#): Spring 구성에서 하나 이상의 [RouterFunction<?>](#) 빈을 감지하고 [주문합니다](#). [RouterFunction.andOther](#)를 통해 이들을 결합하고 결과로 구성된 [RouterFunction](#)으로 요청을 라우팅합니다.
- [HandlerFunctionAdapter](#): [DispatcherHandler](#)가 요청에 매핑 된 [HandlerFunction](#)을 호출 할 수 있도록 하는 간단한 어댑터입니다 .
- [ServerResponseResultHandler](#): [HandlerFunction](#) 호출 결과를 다음과 같이 처리합니다 . [ServerResponse](#)의 [writeTo](#) 메소드를 호출합니다 .

앞의 구성 요소는 기능적 엔드포인트가 [DispatcherHandler](#) 요청 처리 수명 주기에 맞도록 하고 선언된 경우 주석이 달린 컨트롤러와 (잠재적으로) 나란히 실행할 수 있습니다. Spring Boot WebFlux 스타터가 기능적 엔드포인트를 활성화하는 방법이기도 합니다.

다음 예제는 WebFlux Java 구성의 보여줍니다(실행 방법은 [DispatcherHandler](#) 참조).

자바

```
@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Bean 공
개 RouterFunction<?> routerFunctionA() { // ...

}

@Bean 공
개 RouterFunction<?> routerFunctionB() { // ...

}

// ...

@Override
public void configureHttpMessageCodecs(ServerCodecConfigurer 구성자) {
    // 메시지 변환 구성...
}

@Override
public void addCorsMappings(CorsRegistry registry) { // CORS 구
    성...
}

@Override
public void configureViewResolvers(ViewResolverRegistry 레지스트리) {
    // HTML 렌더링을 위한 뷰 해상도 구성...
}}
```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    @Bean 재
    미 routerFunctionA(): RouterFunction<*> {
        // ...
    }

    @Bean 재
    미 routerFunctionB(): RouterFunction<*> {
        // ...
    }

    // ...

    재정의 재미 configureHttpMessageCodecs(구성자: ServerCodecConfigurer) {
        // 메시지 변환 구성...
    }

    재미있는 addCorsMappings(레지스트리: CorsRegistry) 재정의 {
        // CORS 구성 ...
    }

    재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) {
        // HTML 렌더링을 위한 뷰 해상도 구성...
    }
}

```

1.5.5. 핸들러 함수 필터링

웹 MVC

라우팅 기능 빌더에서 전, 후 또는 필터 메소드를 사용하여 핸들러 기능을 필터링할 수 있습니다. 주석을 사용하면 **@ControllerAdvice**, **ServletFilter** 또는 둘 다를 사용하여 유사한 기능을 얻을 수 있습니다. 필터는 빌더가 구축한 모든 경로에 적용됩니다. 즉, 중첩 경로에 정의된 필터는 "최상위" 경로에 적용되지 않습니다. 예를 들어 다음 예를 고려하십시오.

자바

```
RouterFunction<ServerResponse> 경로 = 경로()
    .path("/사람", b1 -> b1
        .nest(수락(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", 핸들러::getPerson)
            .GET(핸들러::listPeople)
            .before(요청 -> ServerRequest.from(요청) ①
                .header("X-RequestHeader", "값")
                .짓다())))
        .POST("/사람", 핸들러::createPerson))
    .after((요청, 응답) -> logResponse(응답)) ②
    .짓다();
```

① 사용자 정의 요청 헤더를 추가하는 **before** 필터는 두 개의 GET 경로에만 적용됩니다.

② 응답을 기록 하는 **After** 필터는 중첩된 경로를 포함한 모든 경로에 적용됩니다.

코틀린

```
발 경로 = 라우터 {
    "/사람".nest {
        GET("/{id}", 핸들러::getPerson)
        GET("", 핸들러::listPeople)
        { ① 전에
            ServerRequest.from(it)
            .header("X-RequestHeader", "값").build()
        }
        POST("/사람", 핸들러::createPerson)
        { _, 응답 -> ② 이후
            logResponse(응답)
        }
    }
}
```

① 사용자 정의 요청 헤더를 추가하는 **before** 필터는 두 개의 GET 경로에만 적용됩니다.

② 응답을 기록 하는 **After** 필터는 중첩된 경로를 포함한 모든 경로에 적용됩니다.

라우터 빌더 의 **필터** 메서드는 **HandlerFilterFunction** 을 사용합니다.

ServerRequest 및 **HandlerFunction** 을 호출하고 **ServerResponse** 를 반환합니다 . 핸들러 함수 매개변수
체인의 다음 요소를 나타냅니다. 이것은 일반적으로 라우팅되는 핸들러이지만
여러 개를 적용하면 또 다른 필터가 됩니다.

이제 **SecurityManager** 가 있다고 가정하고 경로에 간단한 보안 필터를 추가할 수 있습니다.

특정 경로가 허용되는지 여부를 결정할 수 있습니다. 다음 예에서는 그렇게 하는 방법을 보여줍니다.

자바

```
SecurityManager securityManager = ...  
  
RouterFunction<ServerResponse> 경로 = 경로()  
.path("/사람", b1 -> b1  
    .nest(수락(APPLICATION_JSON), b2 -> b2  
        .GET("/{id}", 핸들러::getPerson)  
        .GET(핸들러::listPeople))  
        .POST("/사람", 핸들러::createPerson))  
.filter((요청, 다음) -> {  
    if (securityManager.allowAccessTo(request.path())) {  
        return next.handle(요청);  
    }  
    또 다른 {  
        반환 ServerResponse.status(UNAUTHORIZED).build();  
    }  
})  
.짓다();
```

코틀린

```
val securityManager: SecurityManager = ...  
  
발 경로 = 라우터 {  
    ("/사람" 및 수락(APPLICATION_JSON)).nest {  
        GET("/{id}", 핸들러::getPerson)  
        GET("", 핸들러::listPeople)  
        POST("/사람", 핸들러::createPerson)  
        필터 { 요청, 다음 ->  
            if (securityManager.allowAccessTo(request.path())) {  
                다음(요청)  
            }  
            또 다른 {  
                상태(UNAUTHORIZED).빌드();  
            }  
        }  
    }  
}
```

앞의 예는 [next.handle\(ServerRequest\)](#) 호출 이 선택 사항임을 보여줍니다. 우리 액세스가 허용될 때만 핸들러 함수가 실행되도록 하십시오.

라우터 기능 빌더에서 [필터](#) 방법을 사용하는 것 외에도 필터를 적용할 수 있습니다. [RouterFunction.filter\(HandlerFilterFunction\)](#) 를 통해 기존 라우터 기능 .

가능점 끝점에 대한 CORS 지원은 전용

CorsWeb 필터

1.6. URI 링크

웹 MVC

이 섹션에서는 URI를 준비하기 위해 Spring Framework에서 사용할 수 있는 다양한 옵션에 대해 설명합니다.

1.6.1. 우리컴포넌트

Spring MVC와 Spring WebFlux

UriComponentsBuilder는 다음 예제와 같이 변수가 있는 URI 템플릿에서 URI를 빌드하는 데 도움이 됩니다.

자바

```
UriComponents uriComponents =
    UriComponentsBuilder.fromUriString("https://example.com/
hotels/{hotel}") ① .queryParam("q", "{q}") ② .encode()
③ .build(); ④
```

```
URI uri = uriComponents.expand("웨스틴", "123").ToUri(); ??
```

- ① URI 템플릿이 있는 정적 팩토리 메서드.
- ② URI 구성요소를 추가하거나 교체합니다.
- ③ URI 템플릿과 URI 변수를 인코딩하도록 요청합니다.
- ④ UriComponents를 빌드합니다.
- ⑤ 변수를 확장하여 URI를 얻습니다.

코틀린

```
val uriComponents = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{hotel}")
    ① .queryParam("q", "{q}") ② .encode() ③ .build() ④
```

```
val uri = uriComponents.expand("웨스틴", "123").toUri() ⑤
```

- ① URI 템플릿이 있는 정적 팩토리 메서드.
- ② URI 구성요소를 추가하거나 교체합니다.
- ③ URI 템플릿과 URI 변수를 인코딩하도록 요청합니다.
- ④ UriComponents를 빌드합니다.
- ⑤ 변수를 확장하여 URI를 얻습니다.

앞의 예제는 다음 예제와 같이 하나의 체인으로 통합되고 **buildAndExpand**로 단축될 수 있습니다.

자바

```
URI URI = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{호텔}")
    .queryParam("q", "{q}")
    .인코딩()
    .buildAndExpand("웨스틴", "123")
    .toUri();
```

코틀린

```
웨이브 uri = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{호텔}")
    .queryParam("q", "{q}")
    .인코딩()
    .buildAndExpand("웨스틴", "123")
    .투우리()
```

다음과 같이 URI(인코딩을 의미함)로 직접 이동하여 더 줄일 수 있습니다.
예는 다음을 보여줍니다.

자바

```
URI URI = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{호텔}")
    .queryParam("q", "{q}")
    .build("웨스틴", "123");
```

코틀린

```
웨이브 uri = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{호텔}")
    .queryParam("q", "{q}")
    .build("웨스틴", "123")
```

다음 예제와 같이 전체 URI 템플릿을 사용하여 더 줄일 수 있습니다.

자바

```
URI URI = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{호텔}?q={q}")
    .build("웨스틴", "123");
```

코틀린

```
val uri =
    UriComponentsBuilder .fromUriString("https://example.com/hotels/
    {호텔}?q={q}") .build("웨스틴", "123")
```

1.6.2. 우리빌더

Spring MVC와 Spring WebFlux

UriComponentsBuilder 는 **UriBuilder**를 구현 합니다. **UriBuilderFactory**를 사용하여 차례로 **UriBuilder** 를 만들 수 있습니다. **UriBuilderFactory** 와 **UriBuilder** 는 함께 기본 URL, 인코딩 기본 설정 및 기타 세부 정보와 같은 공유 구성을 기반으로 URI 템플릿에서 URI를 빌드하는 플러그형 메커니즘을 제공합니다.

UriBuilderFactory 를 사용하여 **RestTemplate** 및 **WebClient** 를 구성 하여 URI 준비를 사용자 지정할 수 있습니다. **DefaultUriBuilderFactory** 는 내부적으로 **UriComponentsBuilder** 를 사용 하고 공유 구성 옵션을 노출 하는 **UriBuilderFactory** 의 기본 구현입니다 .

다음 예에서는 **RestTemplate**을 구성하는 방법을 보여줍니다.

자바

```
// org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode 가져오기;

문자열 baseUrl = "https://example.org";
DefaultUriBuilderFactory 팩토리 = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

RestTemplate restTemplate = 새로운 RestTemplate();
restTemplate.setUriTemplateHandler(공장);
```

코틀린

```
// org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode 가져오기

val baseUrl = "https://example.org" val 팩
토리 = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val restTemplate = RestTemplate()
restTemplate.uriTemplateHandler = 공장
```

다음 예에서는 **WebClient**를 구성합니다.

자바

```
// org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode 가져오기;

문자열 baseUrl = "https://example.org";
DefaultUriBuilderFactory 팩토리 = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

WebClient 클라이언트 = WebClient.builder().uriBuilderFactory(factory).build();
```

코틀린

```
// org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode 가져오기

val baseUrl = "https://example.org" val 팩
토리 = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val 클라이언트 = WebClient.builder().uriBuilderFactory(공장).build()
```

또한 **DefaultUriBuilderFactory** 를 직접 사용할 수도 있습니다 . **UriComponentsBuilder** 를 사용하는 것과 비슷 하지만 정직 팩토리 메서드 대신 다음 예제와 같이 구성 및 기본 설정을 보유하는 실제 인스턴스입니다.

자바

```
문자열 baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = 새로운 DefaultUriBuilderFactory(baseUrl);

URI URI = uriBuilderFactory.uriString("/호텔/{호텔}").queryParam("q",
    "{q}").build("웨스틴", "123");
```

코틀린

```
val baseUrl = "https://example.com" val
uriBuilderFactory = DefaultUriBuilderFactory(baseUrl)

val uri = uriBuilderFactory.uriString("/호텔/{호텔}").queryParam("q",
    "{q}").build("웨스틴", "123")
```

1.6.3. URI 인코딩

Spring MVC와 Spring WebFlux

UriComponentsBuilder 는 두 가지 수준에서 인코딩 옵션을 제공합니다.

- **UriComponentsBuilder#encode()**: URI 템플릿을 먼저 사전 인코딩한 다음 엄격하게 인코딩합니다.

확장 시 URI 변수입니다.

- [UriComponents#encode\(\)](#): URI 변수가 확장된 후 URI 구성 요소를 인코딩합니다 .

두 옵션 모두 ASCII가 아닌 문자와 잘못된 문자를 이스케이프된 옥텟으로 바꿉니다. 그러나 첫 번째 옵션은 또한 URI 변수에 나타나는 예약된 의미로 문자를 바꿉니다.



경로에서는 유효하지만 예약된 의미를 갖는 ";"을 고려하십시오. 첫 번째 옵션 ";"를 대체합니다. URI 변수에는 "%3B"가 있지만 URI 템플릿에는 없습니다. 대조적으로, 두 번째 옵션은 경로의 유효한 문자이므로 ";"을 대체하지 않습니다.

대부분의 경우 첫 번째 옵션은 URI 변수를 불투명한 데이터는 완전히 인코딩되지만 두 번째 옵션은 URI 변수가 의도적으로 인코딩되는 경우에 유용합니다. 예약 문자를 포함합니다. 두 번째 옵션은 URI 변수를 확장하지 않을 때도 유용합니다. 우발적으로 URI 변수처럼 보이는 모든 것도 인코딩할 것이기 때문입니다.

다음 예에서는 첫 번째 옵션을 사용합니다.

자바

```
URI URI = UriComponentsBuilder.fromPath("/호텔 목록/{도시}")
    .queryParam("q", "{q}")
    .인코딩()
    .buildAndExpand("뉴욕", "foo+bar")
    .toUri();

// 결과는 "/hotel%20list/New%20York?q=foo%2Bbar"입니다.
```

코틀린

```
val uri = UriComponentsBuilder.fromPath("/호텔 목록/{도시}")
    .queryParam("q", "{q}")
    .인코딩()
    .buildAndExpand("뉴욕", "foo+bar")
    .투우리()

// 결과는 "/hotel%20list/New%20York?q=foo%2Bbar"입니다.
```

다음과 같이 URI(인코딩을 의미함)로 직접 이동하여 앞의 예를 줄일 수 있습니다.

다음 예는 다음을 보여줍니다.

자바

```
URI URI = UriComponentsBuilder.fromPath("/호텔 목록/{도시}")
    .queryParam("q", "{q}")
    .build("뉴욕", "foo+bar");
```

코틀린

```
val uri = UriComponentsBuilder.fromPath("/호텔 목록/{도시}") .queryParam("q",
    "{q}") .build("뉴욕", "foo+bar")
```

다음 예제와 같이 전체 URI 템플릿을 사용하여 더 줄일 수 있습니다.

자바

```
URI URI = UriComponentsBuilder.fromUriString("/호텔 목록/{도시}?q={q}")
    .build("뉴욕", "foo+bar");
```

코틀린

```
val uri = UriComponentsBuilder.fromUriString("/호텔 목록/{도시}?q={q}") .build("뉴욕", "foo+bar")
```

[WebClient](#) 및 [RestTemplate](#) 은 [UriBuilderFactory](#) 전략을 통해 내부적으로 URI 템플릿을 확장하고 인코딩합니다. 다음 예제와 같이 둘 다 사용자 지정 전략으로 구성할 수 있습니다.

자바

```
문자열 baseUrl = "https://example.com";
DefaultUriBuilderFactory 팩토리 = 새로운 DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// RestTemplate을 사용자 정의합니다..
RestTemplate restTemplate = 새로운 RestTemplate();
restTemplate.setUriTemplateHandler(공장);

// 웹클라이언트 커스터마이징..
WebClient 클라이언트 = WebClient.builder().uriBuilderFactory(factory).build();
```

코틀린

```

val baseUrl = "https://example.com" val 팩토리 =
DefaultUriBuilderFactory(baseUrl).apply {encodingMode =
EncodingMode.TEMPLATE_AND_VALUES}

// RestTemplate을 사용자 정의합니다.. val
restTemplate = RestTemplate().apply { uriTemplateHandler =
factory }

// WebClient를 사용자 지정합니다.. val client
= WebClient.builder().uriBuilderFactory(factory).build()

```

DefaultUriBuilderFactory 구현은 내부적 으로 **UriComponentsBuilder** 를 사용 하여 URI 템플릿을 확장하고 인코딩합니다. 공장으로서 아래 인코딩 모드 중 하나를 기반으로 인코딩 접근 방식을 구성할 수 있는 단일 위치를 제공합니다.

- **TEMPLATE_AND_VALUES**: 이전 목록의 첫 번째 옵션에 해당하는 **UriComponentsBuilder#encode()**를 사용 하여 URI 템플릿을 미리 인코딩하고 확장 시 URI 변수를 엄격하게 인코딩합니다.
- **VALUES_ONLY**: URI 템플릿을 인코딩하지 않고 대신 템플릿 으로 확장하기 전에 **UriUtils#encodeUriVariables** 를 통해 URI 변수에 엄격한 인코딩을 적용합니다.
- **URI_COMPONENT**: 앞 의 두 번째 옵션에 해당하는 **UriComponents#encode()**를 사용합니다 . 목록, URI 변수가 확장된 후 URI 구성 요소 값을 인코딩합니다 .
- **NONE**: 인코딩이 적용되지 않습니다.

RestTemplate 은 역사적인 이유와 이전 버전과의 호환성을 위해 **EncodingMode.URI_COMPONENT** 로 설정 됩니다. **WebClient** 는 5.0.x의 **EncodingMode.URI_COMPONENT** 에서 5.1 의 **EncodingMode.TEMPLATE_AND_VALUES** 로 변경된 **DefaultUriBuilderFactory** 의 기본값에 의존합니다 .

1.7. CORS

웹 MVC

Spring WebFlux를 사용하면 CORS(Cross-Origin Resource Sharing)를 처리할 수 있습니다. 이 섹션에서는 그렇게 하는 방법에 대해 설명 합니다.

1.7.1. 소개

웹 MVC

보안상의 이유로 브라우저는 현재 출처 외부의 리소스에 대한 AJAX 호출을 금지합니다. 예를 들어 한 탭에는 은행 계좌가 있고 다른 탭에는 evil.com이 있을 수 있습니다. evil.com의 스크립트는 귀하의 자격 증명을 사용하여 은행 API에 AJAX 요청을 할 수 없어야 합니다. 예를 들어 귀하의 계정에서 돈을 인출하는 것입니다!

CORS(Cross-Origin Resource Sharing)는 [W3C 사양](#)입니다. [대부분의 브라우저](#)에서 구현 IFRAME 또는 JSONP를 기반으로 하는 딜 안전하고 딜 강력한 해결 방법을 사용하는 대신 어떤 종류의 도메인 간 요청이 승인되는지 지정할 수 있습니다.

1.7.2. 처리

[웹 MVC](#)

CORS 사양은 실행 전, 단순 및 실제 요청을 구분합니다. CORS가 어떻게 작동하는지 알아보려면 [이 기사](#)를 읽으십시오. 다른 많은 것들 중에서 또는 자세한 내용은 사양을 참조하십시오.

Spring WebFlux [HandlerMapping](#) 구현은 CORS에 대한 내장 지원을 제공합니다. 요청을 처리기에 성공적으로 매팅한 후 [HandlerMapping](#)은 지정된 요청 및 처리기에 대한 CORS 구성은 확인하고 추가 작업을 수행합니다. 실행 전 요청은 직접 처리되는 반면 단순하고 실제적인 CORS 요청은 가로채고 유효성이 검사되며 필요한 CORS 응답 헤더가 설정됩니다.

교차 출처 요청(즉, [Origin](#) 헤더가 있고 요청 호스트와 다름)을 활성화하려면 명시적으로 선언된 일부 CORS 구성이 필요합니다. 일치하는 CORS 구성이 없으면 실행 전 요청이 거부됩니다. 단순하고 실제적인 CORS 요청의 응답에는 CORS 헤더가 추가되지 않으므로 브라우저는 이를 거부합니다.

각 [HandlerMapping](#)을 구성 할 수 있습니다. URL 패턴 기반 [CorsConfiguration](#) 매팅을 사용하여 개별적으로. 대부분의 경우 응용 프로그램은 WebFlux Java 구성은 사용하여 이러한 매팅을 선언하고 모든 [HandlerMapping](#) 구현에 단일 전역 맵이 전달됩니다.

[HandlerMapping](#) 수준의 전역 CORS 구성은 보다 세분화된 핸들러 수준 CORS 구성과 결합할 수 있습니다. 예를 들어 주석이 달린 컨트롤러는 클래스 또는 메서드 수준 [@CrossOrigin](#) 주석을 사용할 수 있습니다(다른 핸들러는 [CorsConfigurationSource를 구현할 수 있음](#)).

전역 및 로컬 구성을 결합하는 규칙은 일반적으로 추가적입니다(예: 모든 전역 및 모든 로컬 출처). [allowCredentials](#) 및 [maxAge](#)와 같이 단일 값만 허용할 수 있는 속성의 경우 로컬이 전역 값을 재정의합니다. [CorsConfiguration#combine\(CorsConfiguration\)](#) 참조 자세한 사항은.

그만큼

소스에서 자세히 알아보거나 고급 사용자 지정을 수행하려면 다음을 참조하세요.



- [Cors 구성](#)
- [CorsProcessor 및 DefaultCorsProcessor](#)
- [추상 핸들러 매팅](#)

1.7.3. [@CrossOrigin](#)

[웹 MVC](#)

[@CrossOrigin](#) _ 다음 예제와 같이 주석은 주석이 달린 컨트롤러 메서드에 대한 원본 간 요청을 활성화합니다.

자바

```

@RestController
@RequestMapping("/계정") 공
개 클래스 AccountController {

    @CrossOrigin
    @GetMapping("/{id}") 공개
    Mono<계정> 검색(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}") public
    Mono<Void> remove(@PathVariable Long id) { // ...

}

```

코틀린

```

@RestController
@RequestMapping("/계정") 클래
스 AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): 계정 {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) { // ...

}

```

기본적으로 [@CrossOrigin](#) 은 다음을 허용합니다.

- 모든 출처.
- 모든 헤더.
- 컨트롤러 메서드가 매핑되는 모든 HTTP 메서드.

[allowCredentials](#) 는 기본적으로 활성화되어 있지 않습니다. 쿠키 및 CSRF 토큰과 같은 민감한 사용자별 정보를 노출하는 신뢰 수준을 설정하고 적절한 경우에만 사용해야 하기 때문입니다. 활성화되면 [allowOrigins](#) 를 하나 이상의 특정 도메인으로 설정해야 합니다(특수 값 "["](#)는 아님). 또는 [allowOriginPatterns](#) 속성을 사용하여 동적 원본 집합과 일치시킬 수 있습니다.

[maxAge](#) 는 30분으로 설정됩니다.

[@CrossOrigin](#) 은 클래스 수준에서도 지원되며 모든 메서드에서 상속됩니다. 다음 예에서는 특정 도메인을 지정하고 `maxAge` 를 1시간으로 설정합니다.

자바

```
@CrossOrigin(origins = "https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/계정") 공
개 클래스 AccountController {

    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) { // ...

    }

    @DeleteMapping("/{id}") public
    Mono<Void> remove(@PathVariable Long id) { // ...

    }
}
```

코틀린

```
@CrossOrigin("https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/계정") 클래
스 AccountController {

    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): 계정 { // ...

    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) { // ...

    }
}
```

다음 예제와 같이 클래스와 메서드 수준 모두에서 [@CrossOrigin](#) 을 사용할 수 있습니다 .

자바

```

@CrossOrigin(maxAge = 3600) ①
@RestController
@RequestMapping("/계정") 공
개 클래스 AccountController {

    @CrossOrigin("https://domain2.com") ②
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) { // ...

}

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) { // ...

}

```

① 클래스 수준에서 `@CrossOrigin` 사용 .

② 메소드 수준에서 `@CrossOrigin` 사용 .

코틀린

```

@CrossOrigin(maxAge = 3600) ①
@RestController
@RequestMapping("/계정") 클래
스 AccountController {

    @CrossOrigin("https://domain2.com") ②
    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): 계정 {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) { // ...

}

```

① 클래스 수준에서 `@CrossOrigin` 사용 .

② 메소드 수준에서 `@CrossOrigin` 사용 .

1.7.4. 전역 구성

[웹 MVC](#)

세분화된 컨트롤러 메서드 수준 구성 외에도 일부 전역 CORS 구성도 정의하고 싶을 것입니다. URL 기반 `CorsConfiguration` 매팅을 설정할 수 있습니다.

모든 HandlerMapping에서 개별적으로 . 그러나 대부분의 응용 프로그램은 WebFlux Java 구성은 사용하여 이를 수행합니다.

기본적으로 전역 구성은 다음을 활성화합니다.

- 모든 출처.
- 모든 헤더.
- GET, HEAD 및 POST 메서드.

allowedCredentials는 기본적으로 활성화되어 있지 않습니다. 쿠키 및 CSRF 토큰과 같은 민감한 사용자별 정보를 노출하는 신뢰 수준을 설정하고 적절한 경우에만 사용해야 하기 때문입니다. 활성화되면 allowOrigins를 하나 이상의 특정 도메인으로 설정해야 합니다(특수 값 "*"는 아님). 또는 allowOriginPatterns 속성을 사용하여 동적 원본 집합과 일치시킬 수 있습니다.

maxAge는 30분으로 설정됩니다.

WebFlux Java 구성에서 CORS를 활성화하려면 다음 예제와 같이 CorsRegistry 콜백을 사용할 수 있습니다.

자바

```
@ 구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override
public void addCorsMappings(CorsRegistry 레지스트리) {

    Registry.addMapping("/api/
        **") .allowedOrigins("https://
        domain2.com") .allowedMethods("PUT",
        "DELETE") .allowedHeaders("header1", "header2",
        "header3" ) .exposedHeaders("헤더1", "헤더
        2") .allowCredentials(true).maxAge(3600);

    // 매핑 추가...
}
}
```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재미있는 addCorsMappings(레지스트리: CorsRegistry) 재정의 {

        Registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("헤더1", "헤더2", "헤더3")
            .exposedHeaders("헤더1", "헤더2")
            .allowCredentials(true).maxAge(3600)

        // 매핑 추가...
    }
}

```

1.7.5. CORS 웹 필터

[웹 MVC](#)

내장 [CorsWebFilter](#)를 통해 CORS 지원을 적용할 수 있으며, [기능성](#)과 잘 어울리는 끝점.



Spring Security와 함께 [CorsFilter](#)를 사용하려는 경우 Spring 보안에는 [기본 제공 지원](#)이 있습니다. CORS용.

필터를 구성하려면 [CorsWebFilter](#) 빈을 선언하고 [CorsConfigurationSource](#)를 다음 예제와 같이 생성자:

자바

```

@콩
CorsWebFilter corsFilter() {

CorsConfiguration 구성 = 새로운 CorsConfiguration();

// 아마도... //
config.applyPermitDefaultValues()

config.setAllowCredentials(true);
config.addAllowedOrigin("https://domain1.com");
config.addAllowedHeader("*"); config.addAllowedMethod("*");

UrlBasedCorsConfigurationSource 소스 = 새로운 UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", 구성);

새로운 CorsWebFilter(소스)를 반환합니다. }

```

코틀린

```

@Bean 재미 corsFilter(): CorsWebFilter {

값 구성 = CorsConfiguration()

// 아마도... //
config.applyPermitDefaultValues()

config.allowCredentials = true
config.addAllowedOrigin("https://domain1.com")
config.addAllowedHeader("*") config.addAllowedMethod("*")

값 소스 = UrlBasedCorsConfigurationSource().apply
    { registerCorsConfiguration("/**", 구성)
}

CorsWebFilter(소스) 반환 }

```

1.8. 웹 보안

[웹 MVC](#)

스프링 [시큐리티](#) 프로젝트는 악의적인 악용으로부터 웹 애플리케이션을 보호하기 위한 지원을 제공합니다. 다음을 포함한 Spring Security 참조 문서를 참조하십시오.

- [WebFlux 보안](#)
- [WebFlux 테스트 지원](#)
- [CSRF 보호](#)
- [보안 대응 헤더](#)

1.9. 기술 보기

[웹 MVC](#)

Spring WebFlux에서 보기 기술의 사용은 플러그 가능합니다. Thymeleaf, FreeMarker 또는 기타 보기 기술을 사용하기로 결정했는지 여부는 주로 구성 변경의 문제입니다. 이 장에서는 Spring WebFlux와 통합된 보기 기술을 다룹니다. 우리는 당신이 이미 [View Resolution](#)에 익숙하다고 가정합니다.

1.9.1. 백리향

[웹 MVC](#)

Thymeleaf는 두 번 클릭하여 브라우저에서 미리 볼 수 있는 자연스러운 HTML 템플릿을 강조하는 최신 서버 측 Java 템플릿 엔진으로, 실행중인 서버. Thymeleaf는 광범위한 기능 세트를 제공하며 적극적으로 개발 및 유지 관리하고 있습니다. 더 완전한 소개는 [Thymeleaf](#)를 참조하십시오. 프로젝트 홈페이지.

Spring WebFlux와의 Thymeleaf 통합은 Thymeleaf 프로젝트에서 관리합니다. 구성에는 [SpringResourceTemplateResolver](#), [SpringWebFluxTemplateEngine](#) 및 [ThymeleafReactiveViewResolver](#)와 같은 몇 가지 빈 선언이 포함됩니다. 자세한 내용은 [Thymeleaf+Spring](#) 참조 WebFlux 통합 [발표](#).

1.9.2. 프리마커

[웹 MVC](#)

[아파치 프리마커](#) HTML에서 이메일 및 기타로 모든 종류의 텍스트 출력을 생성하기 위한 템플릿 엔진입니다. Spring Framework에는 FreeMarker 템플릿과 함께 Spring WebFlux를 사용하기 위한 내장 통합이 있습니다.

[구성 보기](#)

[웹 MVC](#)

다음 예는 FreeMarker를 보기 기술로 구성하는 방법을 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override 공개
무효 configureViewResolvers(ViewResolverRegistry 레지스트리) { registry.freeMarker();

}

// FreeMarker 구성...

@Bean 공
개 FreeMarkerConfigurer freeMarkerConfigurer() {
    FreeMarkerConfigurer 구성자 = new FreeMarkerConfigurer();
    configurer.setTemplateLoaderPath("클래스 경로:/템플릿/프리마커"); 반환 구성자;

}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) { registry.freeMarker()

}

// FreeMarker 구성...

@Bean 재
미 freeMarkerConfigurer() = FreeMarkerConfigurer().apply
    { setTemplateLoaderPath("classpath:/templates/freemarker")
}

```

템플릿은 앞의 예와 같이 **FreeMarkerConfigurer**에서 지정한 디렉터리에 저장해야 합니다. 이전 구성에서 컨트롤러가 보기 이름 **welcome**을 반환하면 해석기는 **classpath:/templates/freemarker/welcome.ftl** 템플릿을 찾습니다.

FreeMarker 구성

[웹 MVC](#)

FreeMarkerConfigurer 빈에서 적절한 빈 속성을 설정하여 FreeMarker '설정' 및 'SharedVariables'를 **FreeMarker Configuration** 객체(Spring에서 관리)에 직접 전달할 수 있습니다. **freemarkerSettings** 속성에는 **java.util.Properties** 객체가 필요합니다.

`freemarkerVariables` 속성에는 `java.util.Map` 이 필요합니다. 다음 예는 `FreeMarkerConfigurer`를 사용하는 방법을 보여줍니다.

자바

```
@ 구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

// ...

@Bean 공
개 FreeMarkerConfigurer freeMarkerConfigurer() { Map<String,
Object> 변수 = new HashMap<>();
variables.put("xml_escape", new XmlEscape());

FreeMarkerConfigurer 구성 = new FreeMarkerConfigurer();
configure.setTemplateLoaderPath("클래스 경로:/템플릿");
configure.setFreemarkerVariables(변수); 반환 구성;

}}
```

코틀린

```
@ 구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

// ...

@Bean 재
미 freeMarkerConfigurer() = FreeMarkerConfigurer().apply
    { setTemplateLoaderPath("classpath:/templates")
        setFreemarkerVariables(mapOf("xml_escape" to XmlEscape()))
    }}
```

`Configuration` 개체에 적용되는 설정 및 변수에 대한 자세한 내용은 `FreeMarker` 설명서를 참조하십시오 .

양식 처리

웹 MVC

Spring은 무엇보다도 `<spring:bind/>` 요소를 포함하는 JSP에서 사용하기 위한 태그 라이브러리를 제공합니다 . 이 요소는 주로 양식이 양식 지원 개체의 값을 표시하고 웹 또는 비즈니스 계층의 `Validator`에서 실패한 유효성 검사 결과를 표시하도록 합니다. Spring은 또한 양식 입력 요소 자체를 생성하기 위한 추가 편의 매크로와 함께 `FreeMarker`에서 동일한 기능을 지원합니다.

바인드 매크로

웹 MVC

표준 매크로 세트는 FreeMarker용 [spring-webflux.jar](#) 파일 내에서 유지되므로 적절하게 구성된 애플리케이션에서 항상 사용할 수 있습니다.

Spring 템플릿 라이브러리에 정의된 일부 매크로는 내부(비공개)로 간주되지만 매크로 정의에는 그러한 범위가 없으므로 호출 코드 및 사용자 템플릿에서 모든 매크로를 볼 수 있습니다. 다음 섹션에서는 템플릿 내에서 직접 호출해야 하는 매크로에만 집중합니다. 매크로 코드를 직접 보려면 파일 이름이 [spring.ftl](#)이고 [org.springframework.web.reactive.result.view.freemarker](#) 패키지에 있습니다.

바인딩 지원에 대한 자세한 내용은 [단순 바인딩](#)을 참조하세요. 스프링 MVC용.

양식 매크로

FreeMarker 템플릿에 대한 Spring의 양식 매크로 지원에 대한 자세한 내용은 Spring MVC 문서의 다음 섹션을 참조하십시오.

- [입력 매크로](#)
- [입력 필드](#)
- [선택 필드](#)
- [HTML 이스케이프](#)

1.9.3. 스크립트 보기

웹 MVC

Spring Framework에는 [JSR-223](#) 위에서 실행할 수 있는 템플릿 라이브러리와 함께 Spring WebFlux를 사용하기 위한 내장 통합이 있습니다. 자바 스크립팅 엔진. 다음 표는 다양한 스크립트 엔진에서 테스트한 템플릿 라이브러리를 보여줍니다.

스크립팅 라이브러리	스크립팅 엔진
핸들바	코뿔소
수염	코뿔소
반응	코뿔소
EJS	코뿔소
ERB	JRuby
문자열 템플릿	자이언
Kotlin 스크립트 템플릿	코틀린

☒ 다른 스크립트 엔진을 통합하기 위해 기본 규칙은 인터페이스.

[요구 사항](#)[웹 MVC](#)

클래스 경로에 스크립트 엔진이 있어야 하며 자세한 내용은 스크립트 엔진에 따라 다릅니다.

- [나스Hon](#) JavaScript 엔진은 Java 8+와 함께 제공됩니다. 최신 업데이트 릴리스 사용

사용할 수 있는 것이 좋습니다.

- [JRuby](#) Ruby 지원에 대한 종속성으로 추가되어야 합니다.

- [자이썬](#) Python 지원에 대한 종속성으로 추가해야 합니다.

[org.jetbrains.kotlin:kotlin-script-util](#)

종속성 파일

그리고

메타

[org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory](#) 라인을 포함하는 INF/
services/javax.script.ScriptEngineFactory 는 다음과 같아야 합니다.

Kotlin 스크립트 지원을 위해 추가되었습니다. [이 예를](#) 참조하십시오 자세한 내용은

스크립트 템플릿 라이브러리가 있어야 합니다. JavaScript에 대해 그렇게 하는 한 가지 방법은

[WebJars](#).

[스크립트 템플릿](#)

[웹 MVC](#)

사용할 스크립트 엔진, 스크립트 파일을 지정하기 위해 [ScriptTemplateConfigurer](#) 빈을 선언할 수 있습니다.

로드할 것인지, 템플릿을 렌더링하기 위해 호출할 함수 등. 다음 예에서는 Mustache를 사용합니다.

템플릿 및 Nashorn JavaScript 엔진:

[자바](#)

```
@ 구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {
```

```
@ 우세하다
공개 무효 configureViewResolvers(ViewResolverRegistry 레지스트리) {
    레지스트리.스크립트 템플릿();
}
```

```
@ 콩
공개 ScriptTemplateConfigure 구성() {
    ScriptTemplateConfigurer 구성자 = new ScriptTemplateConfigurer();
    configurer.setEngineName("rhino");
    configurer.setScripts("mustache.js");
    configure.setRenderObject("콧수염");
    configure.setRenderFunction("렌더링");
    반환 구성;
}
}
```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) {
        레지스트리.스크립트 템플릿()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply { engineName
        = "nashorn" setScripts("mustache.js") renderObject = "콧
        수염" renderFunction = "렌더링"
    }

}

```

`render` 함수는 다음 매개변수와 함께 호출됩니다.

- **문자열 템플릿:** 템플릿 내용
- **지도 모델:** 보기 모델
- **RenderingContext renderingContext : RenderingContext** 애플리케이션 컨텍스트, 로케일, 템플릿 로더 및 URL(5.0 이후)에 대한 액세스를 제공합니다.

`Mustache.render()`는 기본적으로 이 서명과 호환되므로 직접 호출할 수 있습니다.

템플릿 기술에 일부 사용자 지정이 필요한 경우 사용자 지정 렌더링 기능을 구현하는 스크립트를 제공할 수 있습니다. 예를 들어, [핸들러 바](#) 템플릿을 사용하기 전에 컴파일해야 하고 [폴리필](#) 이 필요합니다. 서버 측 스크립트 엔진에서 사용할 수 없는 일부 브라우저 기능을 에뮬레이트하기 위해. 다음 예제에서는 사용자 지정 렌더링 기능을 설정하는 방법을 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@우세하다
공개 무효 configureViewResolvers(ViewResolverRegistry 레지스트리) {
    레지스트리.스크립트 템플릿();
}

@콩
공개 ScriptTemplateConfigure 구성() {
    ScriptTemplateConfigurer 구성자 = new ScriptTemplateConfigurer();
    configurer.setEngineName("rhino");
    configurer.setScripts("polyfill.js", "handlebars.js", "render.js");
    configure.setRenderFunction("렌더링");
    configurer.setSharedEngine(거짓);
    반환 구성;
}
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) {
    레지스트리.스크립트 템플릿()
}

@콩
재미있는 구성자() = ScriptTemplateConfigurer().apply {
    engineName = "코뿔소"
    setScripts("polyfill.js", "handlebars.js", "render.js")
    renderFunction = "렌더링"
    isSharedEngine = 거짓
}
}

```



non-thread-safe를 사용하는 경우 **sharedEngine** 속성을 **false**로 설정해야 합니다.

다음과 같이 동시성을 위해 설계되지 않은 템플릿 라이브러리가 있는 스크립트 엔진

핸들바 또는 Nashorn에서 실행되는 React. 이 경우 Java SE 8 업데이트 60은

[이 버그](#)로 인해 필요합니다. 그러나 일반적으로 최신 Java SE를 사용하는 것이 좋습니다.

어떤 경우에도 패치 릴리스.

polyfill.js는 다음과 같이 핸들바가 제대로 실행되는 데 필요한 **창** 객체만 정의합니다.

스니펫은 다음을 보여줍니다.

```
var 창 = {};
```

이 기본 [render.js](#) 구현은 템플릿을 사용하기 전에 컴파일합니다. 프로덕션 준비 구현은 캐시된 템플릿 또는 미리 컴파일된 템플릿도 저장하고 재사용해야 합니다. 이것은 스크립트 속에서 수행할 수 있을 뿐만 아니라 필요한 모든 사용자 정의(예: 템플릿 엔진 구성 관리)에서 수행할 수 있습니다. 다음 예제에서는 템플릿을 컴파일하는 방법을 보여줍니다.

```
함수 렌더(템플릿, 모델) { var 컴파일된 템플릿 =
  핸들바.컴파일(템플릿); 컴파일된 템플릿(모델)을 반환합니다. }
```

Spring Framework 단위 테스트, [Java](#), 그리고 [자원](#), 더 많은 구성 예를 보려면

1.9.4. JSON 및 XML

[웹 MVC](#)

[콘텐츠 협상](#)을 위해 클라이언트가 요청한 콘텐츠 유형에 따라 HTML 템플릿을 사용하여 모델을 렌더링하거나 다른 형식(예: JSON 또는 XML)으로 모델을 렌더링할 수 있는 것이 유용합니다. 이를 지원하기 위해 Spring WebFlux는 [Jackson2JsonEncoder](#), [Jackson2SmileEncoder](#) 또는 [Jaxb2XmlEncoder](#)와 같은 [spring-web](#)에서 사용 가능한 [코드](#)를 연결하는 데 사용할 수 있는 [HttpMessageWriterView](#)를 제공합니다.

다른 보기 기술과 달리 [HttpMessageWriterView](#)는 [ViewResolver](#)가 필요하지 않지만 대신 기본 보기로 [구성](#) 됩니다. 다른 [HttpMessageWriter](#) 인스턴스 또는 [Encoder](#) 인스턴스를 래핑하여 이러한 기본 보기 하나 이상 구성할 수 있습니다. 요청된 콘텐츠 유형과 일치하는 콘텐츠가 런타임에 사용됩니다.

대부분의 경우 모델에는 여러 속성이 포함됩니다. 직렬화할 것을 결정하기 위해 렌더링에 사용할 모델 속성의 이름으로 [HttpMessageWriterView](#)를 구성할 수 있습니다. 모델에 하나의 속성만 포함된 경우 해당 속성이 사용됩니다.

1.10. HTTP 캐싱

[웹 MVC](#)

HTTP 캐싱은 웹 애플리케이션의 성능을 크게 향상시킬 수 있습니다. HTTP 캐싱은 [Cache-Control](#) 응답 헤더와 [Last-Modified](#) 및 [ETag](#)와 같은 후속 조건부 요청 헤더를 중심으로 이루어집니다. [Cache-Control](#)은 개인(예: 브라우저) 및 공용(예: 프록시) 캐시에 응답을 캐시하고 재사용하는 방법을 조언합니다. [ETag](#) 헤더는 내용이 변경되지 않은 경우 본문 없이 304(NOT_MODIFIED)가 발생할 수 있는 조건부 요청을 수행하는 데 사용됩니다. [ETag](#)는 [Last-Modified](#) 헤더의 보다 정교한 후계자로 볼 수 있습니다.

이 섹션에서는 Spring WebFlux에서 사용할 수 있는 HTTP 캐싱 관련 옵션에 대해 설명합니다.

1.10.1. 캐시 컨트롤

[웹 MVC](#)

캐시 컨트롤 [Cache-Control](#) 헤더와 관련된 설정 구성을 지원하며 여러 곳에서 인수로 허용됩니다.

- [컨트롤러](#)

- [정적 리소스](#)

RFC 7234 동안 [Cache-Control](#) 응답 헤더에 대한 모든 가능한 지시문을 설명합니다.

[CacheControl](#) 유형은 다음 예제와 같이 일반적인 시나리오에 중점을 둔 사용 사례 중심 접근 방식을 취합니다.

자바

```
// 한 시간 동안 캐시 - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// 캐싱 방지 - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// 공개 및 비공개 캐시에서 10일 동안 캐시, // 공개 캐시는 응답을 변환하지 않아야 합니다. // "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic();
```

코틀린

```
// 한 시간 동안 캐시 - "Cache-Control: max-age=3600" val
ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS)

// 캐싱 방지 - "Cache-Control: no-store" val ccNoStore =
CacheControl.noStore()

// 공개 및 비공개 캐시에서 10일 동안 캐시, // 공개 캐시는 응답을 변환하지 않아야 합니다. // "Cache-Control: max-age=864000, public, no-transform" val ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic()
```

1.10.2. 컨트롤러

[웹 MVC](#)

컨트롤러는 HTTP 캐싱에 대한 명시적 지원을 추가할 수 있습니다. 리소스에 대한 [lastModified](#) 또는 [ETag](#) 값을 조건부 요청 헤더와 비교하려면 먼저 계산해야 하므로 그렇게 하는 것이 좋습니다. 컨트롤러는 다음 예제와 같이 [ResponseEntity](#)에 [ETag](#) 및 [Cache-Control](#) 설정을 추가할 수 있습니다.

자바

```
@GetMapping("/책/{id}")
public ResponseEntity<책> showBook(@PathVariable 긴 ID) {

    책 책 = findBook(id);
    문자열 버전 = book.getVersion();

    ResponseEntity 반환
        .좋아요()
        .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
        .eTag(version) // lastModified도 사용할 수 있습니다.
        .body(책);
}
```

코틀린

```
@GetMapping("/책/{id}")
fun showBook(@PathVariable id: Long): ResponseEntity<Book> {

    val book = findBook(id)
    val 버전 = book.getVersion()

    ResponseEntity 반환
        .좋아요()
        .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
        .eTag(version) // lastModified도 사용할 수 있습니다.
        .body(책)
}
```

앞의 예는 다음과 같은 경우 빈 본문과 함께 304(NOT_MODIFIED) 응답을 보냅니다.

조건부 요청 헤더와 비교하면 내용이 변경되지 않았음을 나타냅니다. 그렇지 않으면,

ETag 및 **Cache-Control** 헤더가 응답에 추가됩니다.

다음과 같이 컨트롤러의 조건부 요청 헤더에 대해 확인할 수도 있습니다.

예는 다음을 보여줍니다.

자바

```

@RequestMapping
public String myHandleMethod(ServerWebExchange 교환, 모델 모델) {

    긴 eTag = ... ①

    if (exchange.checkNotModified(eTag)) {
        null을 반환합니다. ②
    }

    model.addAttribute(...); ③ "myViewName"을
    반환합니다. }

```

① 애플리케이션별 계산.

② 응답은 304(NOT_MODIFIED)로 설정되었습니다. 추가 처리가 없습니다.

③ 요청 처리를 계속합니다.

코틀린

```

@RequestMapping
fun myHandleMethod(exchange: ServerWebExchange, 모델: Model): String? {

    val eTag: Long = ... ①

    if (exchange.checkNotModified(eTag)) { null ② 반환

    }

    model.addAttribute(...); ③
    "myViewName" 반환}

```

① 애플리케이션별 계산.

② 응답은 304(NOT_MODIFIED)로 설정되었습니다. 추가 처리가 없습니다.

③ 요청 처리를 계속합니다.

eTag 값, **lastModified** 값 또는 둘 다에 대해 조건부 요청을 확인하기 위한 세 가지 변형이 있습니다. 조건부 **GET** 및 **HEAD** 요청의 경우 응답을 304(NOT_MODIFIED)로 설정할 수 있습니다.

조건부 **POST**, **PUT** 및 **DELETE**의 경우 대신 응답을 412(PRECONDITION_FAILED)로 설정하여 동시 수정을 방지할 수 있습니다.

1.10.3. 정적 리소스

[웹 MVC](#)

최적의 성능을 위해 **Cache-Control** 및 조건부 응답 헤더와 함께 정적 리소스를 제공해야 합니다. [정적 리소스](#) 구성에 대한 섹션을 참조하세요.

1.11. WebFlux 구성

[웹 MVC](#)

WebFlux Java 구성은 주석이 달린 컨트롤러 또는 기능적 끝점으로 요청을 처리하는 데 필요한 구성 요소를 선언하고 구성을 사용자 지정하는 API를 제공합니다. 즉, Java 구성에 의해 생성된 기본 빈을 이해할 필요가 없습니다. 그러나 그것들을 이해하고 싶다면 [WebFluxConfigurationSupport](#)에서 그것들을 보거나 [Special Bean Types](#)에서 그것들이 무엇인지에 대해 더 읽을 수 있습니다.

구성 API에서 사용할 수 없는 고급 사용자 지정의 경우 [고급 구성 모드](#)를 통해 구성은 완전히 제어할 수 있습니다.

1.11.1. WebFlux 구성 활성화

[웹 MVC](#)

다음 예제와 같이 Java 구성에서 [@EnableWebFlux](#) 주석을 사용할 수 있습니다.

자바

```
@구성
@EnableWebFlux
공개 클래스 WebConfig {}
```

코틀린

```
@구성
@EnableWebFlux
클래스 WebConfig
```

앞의 예제는 여러 Spring WebFlux [인프라 빈](#)을 등록하고 JSON, XML 등의 클래스 경로에서 사용 가능한 종속성에 적응합니다.

1.11.2. WebFlux 구성 API

[웹 MVC](#)

다음 예제와 같이 Java 구성에서 [WebFluxConfigurer](#) 인터페이스를 구현할 수 있습니다.

자바

```
@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

// 구성 메소드 구현... }
```

코틀린

```
@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

// 구성 메소드 구현... }
```

1.11.3. 변환, 서식

[웹 MVC](#)

기본적으로 필드에서 `@NumberFormat` 및 `@DateTimeFormat`을 통한 사용자 정의 지원과 함께 다양한 숫자 및 날짜 유형에 대한 포맷터가 설치 됩니다.

Java 구성에서 사용자 정의 포맷터 및 변환기를 등록하려면 다음을 사용하십시오.

자바

```
@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override
public void addFormatters(FormatterRegistry 레지스트리) { // ...

}
```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 addFormatters(레지스트리: FormatterRegistry) {
        // ...
    }
}

```

기본적으로 Spring WebFlux는 날짜 값을 구문 분석하고 형식을 지정할 때 요청 로케일을 고려합니다.

이것은 날짜가 "입력" 양식 필드가 있는 문자열로 표시되는 양식에서 작동합니다. 그러나 "날짜" 및 "시간" 양식 필드의 경우 브라우저는 HTML 사양에 정의된 고정 형식을 사용합니다. 이러한 경우 날짜 및 시간 형식을 다음과 같이 사용자 지정할 수 있습니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

    @Override
    public void addFormatters(FormatterRegistry Registry)
        { DateTimeFormatterRegistrar 등록자 = new DateTimeFormatterRegistrar(); 등록
          자.setUsesIsoFormat(true); registrar.registerFormatters(레지스트리);

    }
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 addFormatters(레지스트리: FormatterRegistry) { val registrar =
        DateTimeFormatterRegistrar() registrar.setUsesIsoFormat(true)
        registrar.registerFormatters(registry)

    }
}

```

 [FormatterRegistrar](#) 참조 [FormatterRegistrar](#) 구현 및 [FormattingConversionServiceFactoryBean](#) –

1.11.4. 확인

웹 MVC

기본적으로 Bean 유효성 검사 가 클래스 경로(예: Hibernate Validator)에 존재하는 경우 LocalValidatorFactoryBean 은 전역 유효성 검사기 로 등록됩니다. @Controller 메서드 인수에서 @Valid 및 @Validated 와 함께 사용합니다.

Java 구성에서 다음 예제와 같이 전역 Validator 인스턴스를 사용자 정의할 수 있습니다.

자바

```
@ 구성
@EnableWebFlux
공개 클래스 WebConfig 는 WebFluxConfigurer 를 구현합니다. {

@Override
public Validator getValidator() { // ...

}

}
```

코틀린

```
@ 구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 getValidator(): Validator { // ...

}

}
```

다음 예제와 같이 Validator 구현을 로컬로 등록할 수도 있습니다.

자바

```
@Controller
공개 클래스 MyController {

@InitBinder 보호
무효 initBinder(WebDataBinder 바인더) { 바인더.addValidators(new FooValidator());
}

}
```

코틀린

```
@Controller
클래스 MyController {

    @InitBinder 보호
    재미 initBinder(바인더: WebDataBinder) { 바인
        더.addValidators(FooValidator())
    }
}
```



어딘가에 **LocalValidatorFactoryBean** 을 주입 해야 하는 경우 MVC 구성에 선언된 것과 충돌을 피하기 위해
빈을 만들고 **@Primary** 로 표시합니다.

1.11.5. 콘텐츠 유형 해석기

웹 MVC

Spring WebFlux가 요청에서 **@Controller** 인스턴스에 대해 요청된 미디어 유형을 결정하는 방법을 구성할 수 있습니다. 기본적으로 **Accept** 헤더만 선택되지만 쿼리 매개변수 기반 전략을 활성화할 수도 있습니다.

다음 예에서는 요청된 콘텐츠 유형 해상도를 사용자 지정하는 방법을 보여줍니다.

자바

```
@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

    @Override
    public void configureContentTypeResolver(RequestedContentTypeResolverBuilder 빌더) { // ...
}

}
```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 configureContentTypeResolver(빌더:
RequestedContentTypeResolverBuilder) { // ...

}}

```

1.11.6. HTTP 메시지 코덱

[웹 MVC](#)

다음 예에서는 요청 및 응답 본문을 읽고 쓰는 방법을 사용자 지정하는 방법을 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override
public void configureHttpMessageCodecs(ServerCodecConfigurer 구성자) {
    configure.defaultCodecs().maxInMemorySize(512*1024);
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 configureHttpMessageCodecs(구성자: ServerCodecConfigurer) {
    // ...
}

```

ServerCodecConfigurer 는 기본 판독기 및 작성기 세트를 제공합니다. 이를 사용하여 더 많은 판독기 및 작성기를 추가하거나, 기본 항목을 사용자 지정하거나, 기본 항목을 완전히 교체할 수 있습니다.

Jackson JSON 및 XML의 경우 **Jackson2ObjectMapperBuilder** 사용을 고려하십시오. 사용자 정의 다음과 같은 Jackson의 기본 속성:

- **DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES** 비활성화됩니다.

- `MapperFeature.DEFAULT_VIEW_INCLUSION` 비활성화됩니다.

또한 다음과 같은 잘 알려진 모듈이 클래스 경로에서 감지되면 자동으로 등록합니다.

- `jackson-datatype-joda`: Joda-Time 유형을 지원합니다.
- `jackson-datatype-jsr310`: Java 8 날짜 및 시간 API 유형을 지원합니다.
- `jackson-datatype-jdk8`: `Optional` 과 같은 다른 Java 8 유형에 대한 지원 .
- `jackson-module-kotlin`: Kotlin 클래스 및 데이터 클래스를 지원합니다.

1.11.7. 확인자 보기

[웹 MVC](#)

다음 예에서는 보기 해상도를 구성하는 방법을 보여줍니다.

자바

```
@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override
public void configureViewResolvers(ViewResolverRegistry 레지스트리) {
    // ...
}}
```

코틀린

```
@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) {
        // ...
}}
```

`ViewResolverRegistry`에는 Spring Framework가 통합하는 보기 기술에 대한 바로 가기가 있습니다 . 다음 예제에서는 `FreeMarker`를 사용합니다(기본 `FreeMarker` 보기 기술 구성도 필요함).

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override 공개
무효 configureViewResolvers(ViewResolverRegistry 레지스트리) { registry.freeMarker();

}

// 프리마커 구성...

@Bean 공
개 FreeMarkerConfigurer freeMarkerConfigurer() {
    FreeMarkerConfigurer 구성 = new FreeMarkerConfigurer();
    configure.setTemplateLoaderPath("클래스 경로:/템플릿"); 반환 구성;
}

}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) { registry.freeMarker()

}

// 프리마커 구성...

@Bean 재
미 freeMarkerConfigurer() = FreeMarkerConfigurer().apply
    { setTemplateLoaderPath("classpath:/templates")
}

}

```

다음 예제와 같이 모든 **ViewResolver** 구현을 연결할 수도 있습니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override
public void configureViewResolvers(ViewResolverRegistry 레지스트리) {
    ViewResolver 리졸버 = ... ;
    Registry.viewResolver(리졸버);
}
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) { val resolver:
    ViewResolver = ...
    Registry.viewResolver(리졸버
}
}

```

콘텐츠 협상 을 지원 하고 보기 확인(HTML 제외)을 통해 다른 형식을 렌더링하려면 [spring-web](#) 에서 사용 가능한 코덱 을 허용하는 [HttpMessageWriterView](#) 구현을 기반으로 하나 이상의 기본 보기 cấu성을 구성할 수 있습니다 . 다음 예에서는 그렇게 하는 방법을 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override 공개
무효 configureViewResolvers(ViewResolverRegistry 레지스트리) { registry.freeMarker();

    Jackson2JsonEncoder 인코더 = new Jackson2JsonEncoder();
    Registry.defaultViews(새로운 HttpMessageWriterView(인코더));
}

// ...
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 configureViewResolvers(레지스트리: ViewResolverRegistry) { registry.freeMarker()

        val 인코더 = Jackson2JsonEncoder()
        registry.defaultViews(HttpMessageWriterView(인코더))
    }

    // ...
}

```

Spring WebFlux와 통합된 [보기 기술에 대한 자세한 내용은 보기 기술](#) 을 참조하세요.

1.11.8. 정적 리소스

[웹 MVC](#)

이 옵션은 [리소스 기본](#) 목록에서 정적 리소스를 제공하는 편리한 방법을 제공합니다. 위치.

다음 예에서 `/resources`로 시작하는 요청이 주어지면 상대 경로는 클래스 경로의 `/static`에 상대적인 정적 리소스를 찾고 제공하는 데 사용됩니다. 리소스는 브라우저 캐시의 최대 사용과 브라우저의 HTTP 요청 감소를 보장하기 위해 1년 만료로 제공됩니다. [Last-Modified](#) 헤더도 평가되며 존재하는 경우 304 상태 코드가 반환됩니다. 다음 목록은 예를 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
        { registry.addResourceHandler("/resources/**") .addResourceLocations("/")
            public", "classpath:/"
            static/" .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS));
    }

}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 addResourceHandlers(registry: ResourceHandlerRegistry)
        { registry.addResourceHandler("/resources/**") .addResourceLocations("/")
            public", "classpath:/static/") .setCacheControl(CacheControl.maxAge(365,
            TimeUnit.DAYS) )
    }
}

```

리소스 핸들러는 [ResourceResolver](#) 체인도 지원합니다. 구현 및 [ResourceTransformer](#) 최적화된 리소스로 작업하기 위한 도구 체인을 만드는 데 사용할 수 있는 구현입니다.

MD5 해시를 기반으로 버전이 지정된 리소스 URL에 [VersionResourceResolver](#) 를 사용할 수 있습니다.

콘텐츠, 고정 애플리케이션 버전 또는 기타 정보에서 계산됩니다. [ContentVersionStrategy](#) (MD5 해시)는 몇 가지 주목할만한 예외(예: 모듈 로더와 함께 사용되는 JavaScript 리소스)를 제외하고는 좋은 선택입니다.

다음 예는 Java 구성에서 [VersionResourceResolver](#) 를 사용하는 방법을 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
        { registry.addResourceHandler("/resources/**") .addResourceLocations("/")
            public/) .resourceChain(true) .addResolver(new

                VersionResourceResolver().addContentVersionStrategy("/**")); }

}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

    재정의 재미 addResourceHandlers(registry: ResourceHandlerRegistry) { registry.addResourceHandler("/resources/
        **") .addResourceLocations("/public/") .resourceChain(true)

        .addResolver(VersionResourceResolver().addContentVersionStrategy("/**")) }

}

```

[ResourceUrlProvider](#) 를 사용 하여 URL을 다시 작성하고 해석기 및 변환기의 전체 체인을 적용할 수 있습니다(예: 버전 삽입). WebFlux 구성은 [ResourceUrlProvider](#) 를 제공 하므로 다른 사람에게 주입할 수 있습니다.

Spring MVC와 달리 WebFlux에서는 현재로서는 정적 리소스 URL을 투명하게 다시 작성할 수 있는 방법이 없습니다. 왜냐하면 resolver와 변환기의 non-blocking chain을 사용할 수 있는 보기 기술이 없기 때문입니다. 로컬 리소스만 제공할 때 해결 방법은 [ResourceUrlProvider](#) 를 직접 사용하고(예: 사용자 지정 요소를 통해) 차단하는 것입니다.

[EncodedResourceResolver](#) (예: Gzip, Brotli로 인코딩됨)와 [VersionedResourceResolver](#) 를 모두 사용할 때 콘텐츠 기반 버전이 인코딩되지 않은 파일을 기반으로 항상 안정적으로 계산되도록 하려면 이 순서대로 등록해야 합니다.

[WebJars org.webjars:webjars-locator-core](#) 라이브러리가 클래스 경로에 있을 때 자동으로 등록되는 [WebJarsResourceResolver](#) 를 통해서도 지원됩니다 . 해석기는 jar 버전을 포함하도록 URL을 다시 작성할 수 있으며 버전 없이 들어오는 URL과도 일치할 수 있습니다(예: /jquery/jquery.min.js 에서 /jquery/1.2.0/jquery.min.js로).



[ResourceHandlerRegistry](#) 를 기반으로 하는 Java 구성은 최종 수정 동작 및 최적화된 리소스 확인과 같은 세분화된 제어를 위한 추가 옵션을 제공합니다.

1.11.9. 경로 일치

[웹 MVC](#)

경로 일치와 관련된 옵션을 사용자 지정할 수 있습니다. 개별 옵션에 대한 자세한 내용은 [PathMatchConfigurer](#) 를 참조하십시오. 자바도. 다음 예에서는 [PathMatchConfigurer](#)를 사용하는 방법을 보여줍니다.

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override 공개
무효 configurePathMatch(PathMatchConfigurer 구성자) { 구성
    자 .setUseCaseSensitiveMatch(true) .setUseTrailingSlashMatch(false) .addPathPrefix("/api",
        HandlerTypePredicate.forName(RestController.class));

}
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

@Override 재미
configurePathMatch(구성: PathMatchConfigure) { 구
    성 .setUseCaseSensitiveMatch(true) .setUseTrailingSlashMatch(false) .addPathPrefix("/api",
        HandlerTypePredicate.forName(RestController :: class.java))
}
}

```

Spring WebFlux 는 세마콜론 내용이 제거된(즉, 경로 또는 행렬 변수) 디코딩된 경로 세그먼트 값에 액세스하기 위해 **RequestPath** 라는 요청 경로의 구문 분석된 표현에 의존합니다. 즉, Spring MVC와 달리 요청 경로를 디코딩하지 경로 일치를 위해 세마콜론 내용을 제거할지 여부를 지정할 필요가 없습니다.



Spring WebFlux는 또한 우리가 **권장** 하는 Spring MVC와 달리 접미사 패턴 일치를 지원하지 않습니다. 그것에 대한 의존에서 멀어집니다.

1.11.10. 웹소켓 서비스

WebFlux Java 구성은 WebSocket 핸들러 호출을 지원 하는 **WebSocketHandlerAdapter** 빈을 선언합니다. 처리하기 위해 해야 할 모든 작업을 의미합니다.

WebSocket 핸드셰이크 요청은 **SimpleUrlHandlerMapping** 을 통해 **WebSocketHandler** 를 URL 에 매핑하는 것입니다.

어떤 경우에는 WebSocket 서버 속성을 구성할 수 있는 제공된 **WebSocketService** 서비스로 **WebSocketHandlerAdapter** 빈을 생성해야 할 수도 있습니다. 예를 들어:

자바

```

@구성
@EnableWebFlux
공개 클래스 WebConfig는 WebFluxConfigurer를 구현합니다. {

@Override
public WebSocketService getWebSocketService()
    { TomcatRequestUpgradeStrategy 전략 = new TomcatRequestUpgradeStrategy(); 전
    략.setMaxSessionIdleTimeout(0L); 새로운 HandshakeWebSocketService(전략)를 반환합니다.

}
}

```

코틀린

```

@구성
@EnableWebFlux
클래스 WebConfig : WebFluxConfigurer {

@Override fun
webSocketService(): WebSocketService { val 전략 =
    TomcatRequestUpgradeStrategy().apply {
        setMaxSessionIdleTimeout(0L)

    } HandshakeWebSocketService(전략) 반환
}
}

```

1.11.11. 고급 구성 모드

[웹 MVC](#)

@EnableWebFlux 는 다음을 수행하는 DelegatingWebFluxConfiguration 을 가져옵니다 .

- WebFlux 애플리케이션을 위한 기본 Spring 구성 제공
- 해당 구성은 사용자 정의하기 위해 WebFluxConfigurer 구현을 감지하고 위임 합니다.

고급 모드의 경우 @EnableWebFlux 를 제거하고 다음에서 직접 확장할 수 있습니다.

다음 예제와 같이 WebFluxConfigurer 를 구현하는 대신 WebFluxConfiguration 을 위임 합니다.

자바

```

@Configuration
공개 클래스 WebConfig 확장 DelegatingWebFluxConfiguration {

// ...
}

```

코틀린

```
@Configuration  
클래스 WebConfig : DelegatingWebFluxConfiguration {  
// ... }
```

WebConfig에서 기존 메소드를 유지할 수 있지만 이제 기본 클래스에서 빈 선언을 재정의할 수 있으며 여전히 클래스 경로에 다른 **WebMvcConfigurer** 구현을 얼마든지 가질 수 있습니다.

1.12. HTTP/2

[웹 MVC](#)

HTTP/2는 Reactor Netty, Tomcat, Jetty 및 Undertow에서 지원됩니다. 그러나 서버 구성과 관련된 고려 사항이 있습니다. 자세한 내용은 [HTTP/2 위키 페이지를 참조하십시오.](#)

2장. 웹 클라이언트

Spring WebFlux에는 HTTP 요청을 수행할 클라이언트가 포함되어 있습니다. [WebClient](#)에는 Reactor를 기반으로 하는 기능적이고 유창한 API가 있습니다. [Reactive Libraries](#)를 참조하세요. 이 라이브러리는 스레드나 동시성을 처리할 필요 없이 비동기 논리의 선언적 구성 을 가능하게 합니다. 완전히 비차단적이며 스트리밍을 지원하며 서버 측에서 요청 및 응답 콘텐츠를 인코딩 및 디코딩하는 데에도 사용되는 동일한 [코덱](#)에 의존합니다.

[WebClient](#)는 요청을 수행할 HTTP 클라이언트 라이브러리가 필요합니다. 다음에 대한 기본 제공 지원이 있습니다.

- [원자로 네티](#)
- [제티 리액티브 HttpClient](#)
- [아파치 HttpComponents](#)
- 다른 것들은 [ClientHttpConnector](#)를 통해 연결될 수 있습니다.

2.1. 구성

[WebClient](#)를 만드는 가장 간단한 방법은 정적 팩토리 메서드 중 하나를 사용하는 것입니다.

- [WebClient.create\(\)](#)
- [WebClient.create\(문자열 baseUrl\)](#)

추가 옵션과 함께 [WebClient.builder\(\)](#)를 사용할 수도 있습니다.

- [uriBuilderFactory](#): 기본 URL로 사용할 사용자 정의 된 [UriBuilderFactory](#).
- [defaultUriVariables](#): URI 템플릿을 확장할 때 사용할 기본값입니다.
- [defaultHeader](#): 모든 요청에 대한 헤더입니다.
- [defaultCookie](#): 모든 요청에 대한 쿠키입니다.
- [defaultRequest](#): 모든 요청을 사용자 정의하는 [소비자](#).
- [필터](#): 모든 요청에 대한 클라이언트 필터입니다.
- [exchangeStrategies](#): HTTP 메시지 판독기/작성기 사용자 지정.
- [clientConnector](#): HTTP 클라이언트 라이브러리 설정.

예를 들어:

자바

```
WebClient 클라이언트 =
    WebClient.builder() .codecs(구성
        -> ... ) .build();
```

코틀린

```
val webClient = WebClient.builder()
    .codecs { 구성 -> ... }
    .짓다()
```

일단 구축되면 **WebClient**는 변경할 수 없습니다. 그러나 이를 복제하고 다음과 같이 수정된 사본을 작성할 수 있습니다. 다음과 같다:

자바

```
WebClient client1 = WebClient.builder()
    .필터(필터A).필터(필터B).빌드();

WebClient client2 = client1.mutate()
    .필터(필터C).필터(필터D).빌드();

// client1에는 filterA, filterB가 있습니다.

// client2에는 filterA, filterB, filterC, filterD가 있습니다.
```

코틀린

```
val client1 = WebClient.builder()
    .필터(필터A).필터(필터B).빌드()

val client2 = client1.mutate()
    .filter(filterC).filter(filterD).build()

// client1에는 filterA, filterB가 있습니다.

// client2에는 filterA, filterB, filterC, filterD가 있습니다.
```

2.1.1. 최대 메모리 크기

코덱은 애플리케이션 메모리 문제를 피하기 위해 메모리에 데이터를 버퍼링하는 데 제한이 있습니다. 기본적으로 그들은 256KB로 설정됩니다. 충분하지 않으면 다음 오류가 발생합니다.

```
org.springframework.core.io.buffer.DataBufferLimitException: 최대 한도 초과
버퍼할 바이트
```

기본 코덱의 제한을 변경하려면 다음을 사용하십시오.

자바

```
WebClient webClient = WebClient.builder()
    .codecs(구성 -> 구성.defaultCodecs().maxInMemorySize(2 * 1024 *
1024))
    .짓다();
```

코틀린

```
val webClient = WebClient.builder() .codecs
{ 구성자 -> 구성자.defaultCodecs().maxInMemorySize(2 * 1024 *
1024) }
.짓다()
```

2.1.2. 원자로 네티

Reactor Netty 설정을 사용자 정의하려면 사전 구성된 [HttpClient](#)를 제공하십시오.

자바

```
HttpClient httpClient = HttpClient.create().secure(sslSpec -> ...);

WebClient webClient = WebClient.builder()
    .clientConnector(새로운 ReactorClientHttpConnector(httpClient)) .build();
```

코틀린

```
val httpClient = HttpClient.create().secure { ... }

val webClient =
WebClient.builder() .clientConnector(ReactorClientHttpConnector(httpClient)) .build()
```

자원

기본적으로 [HttpClient](#)는 이벤트 루프 스레드 및 연결 풀을 포함하여 [reactor.netty.http.HttpResources](#)에 있는 전역 Reactor Netty 리소스에 참여 합니다. 이벤트 루프 동시성을 위해 고정된 공유 리소스가 선호되기 때문에 권장되는 모드입니다. 이 모드에서 전역 자원은 프로세스가 종료될 때까지 활성 상태를 유지합니다.

서버가 프로세스와 시간을 맞추면 일반적으로 명시적인 종료가 필요하지 않습니다. 그러나 서버가 in-process(예: WAR로 배포된 Spring MVC 애플리케이션)을 시작하거나 중지할 수 있는 경우, Reactor 가 Netty 전역 리소스는 다음 예제와 같이 Spring [ApplicationContext](#)가 닫힐 때 종료됩니다.

자바

```
@Bean 공개 ReactorResourceFactory ReactorResourceFactory()
{ return new ReactorResourceFactory(); }
```

코틀린

```
@Bean 재미있는 ReactorResourceFactory() = ReactorResourceFactory()
```

또한 글로벌 Reactor Netty 리소스에 참여하지 않도록 선택할 수도 있습니다. 그러나 이 모드에서는 다음 예제와 같이 모든 Reactor Netty 클라이언트 및 서버 인스턴스가 공유 리소스를 사용하도록 해야 하는 부담이 있습니다.

자바

```
@Bean 공개 ReactorResourceFactory resourceFactory()
{ ReactorResourceFactory 팩토리 = new ReactorResourceFactory();
factory.setUseGlobalResources(거짓); ① 반환 공장; }
```

```
@Bean 공개 WebClient webClient() {

Function<HttpClient, HttpClient> mapper = client -> { // 추가 사용자
정의...
};

ClientHttpConnector 커넥터 = new
ReactorClientHttpConnector(resourceFactory(), 매퍼); ②

반환 WebClient.builder().clientConnector(connector).build(); ③ }
```

① 글로벌 자원과 독립적인 자원을 창출한다.

② Resource Factory와 함께 **ReactorClientHttpConnector** 생성자를 사용합니다 .

③ 커넥터를 **WebClient.Builder**에 연결합니다.

코틀린

```
@Bean fun resourceFactory() = ReactorResourceFactory() 적용
{isUseGlobalResources = false ① }
```

```
@Bean 재미 webClient(): WebClient {
```

```
    val 매퍼: (HttpClient) -> HttpClient = {
        // 추가 사용자 정의...
    }
```

```
    val 커넥터 = ReactorClientHttpConnector(resourceFactory(), 매퍼) ②
```

```
    반환 WebClient.builder().clientConnector(connector).build() ③ }
```

① 글로벌 자원과 독립적인 자원을 창출한다.

② Resource Factory와 함께 `ReactorClientHttpConnector` 생성자를 사용합니다 .

③ 커넥터를 `WebClient.Builder`에 연결합니다.

시간 초과

연결 시간 초과를 구성하려면:

자바

```
io.netty.channel.ChannelOption 가져오기;

HttpClient httpClient =
    HttpClient.create() .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000);

WebClient webClient = WebClient.builder()
    .clientConnector(새로운 ReactorClientHttpConnector(httpClient)) .build();
```

코틀린

```
io.netty.channel.ChannelOption 가져오기

val httpClient =
    HttpClient.create() .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000);

val webClient =
    WebClient.builder() .clientConnector(new
        ReactorClientHttpConnector(httpClient)) .build();
```

읽기 또는 쓰기 시간 초과를 구성하려면:

자바

```
가져오기 io.netty.handler.timeout.ReadTimeoutHandler;
io.netty.handler.timeout.WriteTimeoutHandler 가져오기;

HttpClient httpClient = HttpClient.create()
    .doOnConnected(연결 -> 연결
        .addHandlerLast(새로운 ReadTimeoutHandler(10))
        .addHandlerLast(새로운 WriteTimeoutHandler(10)));

// 웹 클라이언트 생성...
```

코틀린

```
io.netty.handler.timeout.ReadTimeoutHandler 가져오기
io.netty.handler.timeout.WriteTimeoutHandler 가져오기

val httpClient = HttpClient.create()
    .doOnConnected { 연결 -> 연결
        .addHandlerLast(새로운 ReadTimeoutHandler(10))
        .addHandlerLast(새로운 WriteTimeoutHandler(10))
    }

// 웹 클라이언트 생성...
```

모든 요청에 대한 응답 시간 초과를 구성하려면:

자바

```
HttpClient httpClient = HttpClient.create()
    .responseTimeout(Duration.ofSeconds(2));

// 웹 클라이언트 생성...
```

코틀린

```
val httpClient = HttpClient.create()
    .responseTimeout(Duration.ofSeconds(2));

// 웹 클라이언트 생성...
```

특정 요청에 대한 응답 시간 초과를 구성하려면:

자바

```
WebClient.create().get()
    .uri("https://example.org/path")
    .httpRequest(httpRequest -> {
        HttpClientRequest ReactorRequest = httpRequest.getNativeRequest();
        ReactorRequest.responseTimeout(Duration.ofSeconds(2));
    })
    .검색하다()
    .bodyToMono(String.class);
```

코틀린

```
WebClient.create().get()
    .uri("https://example.org/path")
    .httpRequest { httpRequest: ClientHttpRequest ->
        val ReactorRequest = httpRequest.getNativeRequest<HttpClientRequest>()
        ReactorRequest.responseTimeout(Duration.ofSeconds(2))
    }
    .검색하다()
    .bodyToMono(문자열::class.java)
```

2.1.3. 둑

다음 예는 Jetty [HttpClient](#) 설정을 사용자 정의하는 방법을 보여줍니다.

자바

```
HttpClient httpClient = 새로운 HttpClient();
httpClient.setCookieStore(...);

WebClient webClient = WebClient.builder()
    .clientConnector(새로운 JettyClientHttpConnector(httpClient))
    .짓다();
```

코틀린

```
val httpClient = HttpClient()
httpClient.cookieStore = ...
...
val webClient = WebClient.builder()
    .clientConnector(새로운 JettyClientHttpConnector(httpClient))
    .짓다();
```

기본적으로 [HttpClient](#) 는 자체 리소스 ([Executor](#), [ByteBufferPool](#), [Scheduler](#))를 생성하며, 프로세스가 종료되거나 [stop\(\)](#) 이 호출될 때까지 활성화됩니다.

Jetty 클라이언트(및 서버)의 여러 인스턴스 간에 리소스를 공유하고 다음을 보장할 수 있습니다.

다음 예제와 같이 **JettyResourceFactory** 유형의 Spring 관리 Bean을 선언하여 Spring **ApplicationContext** 가 닫힐 때 자원이 종료됩니다 .

자바

```
@Bean
공개 JettyResourceFactory resourceFactory() { return new
JettyResourceFactory(); }

@Bean
공개 WebClient webClient() {

HttpClient httpClient = 새로운 HttpClient(); // 추가 사용자 정의...

ClientHttpConnector 커넥터 =
    새로운 JettyClientHttpConnector(httpClient, resourceFactory()); ①

반환 WebClient.builder().clientConnector(connector).build(); ② }
```

① Resource Factory와 함께 **JettyClientHttpConnector** 생성자를 사용합니다 .

② **WebClient.Builder** 에 커넥터를 연결합니다 .

코틀린

```
@Bean 재미있는 resourceFactory() = JettyResourceFactory()

@Bean
재미 webClient(): WebClient {

값 httpClient = HttpClient()
// 추가 사용자 정의...

val 커넥터 = JettyClientHttpConnector(httpClient, resourceFactory()) ①

반환 WebClient.builder().clientConnector(connector).build() ② }
```

① Resource Factory와 함께 **JettyClientHttpConnector** 생성자를 사용합니다 .

② **WebClient.Builder** 에 커넥터를 연결합니다 .

2.1.4. HttpComponents

다음 예제에서는 Apache HttpComponents **HttpClient** 설정을 사용자 지정하는 방법을 보여줍니다.

자바

```
HttpAsyncClientBuilder clientBuilder = HttpAsyncClients.custom();
clientBuilder.setDefaultRequestConfig(...); CloseableHttpAsyncClient 클라이언트 = clientBuilder.build(); ClientHttpConnector 커넥터 = 새로운 HttpComponentsClientHttpConnector(클라이언트);
```

```
WebClient webClient = WebClient.builder().clientConnector(connector).build();
```

코틀린

```
val 클라이언트 = HttpAsyncClients.custom().apply
{setDefaultRequestConfig(...)}.build() val 커넥터 =
HttpComponentsClientHttpConnector(클라이언트) val
webClient = WebClient.builder().clientConnector(connector).build()
```

2.2. 검색하다()

[retrieve\(\)](#) 메서드는 응답을 추출하는 방법을 선언하는 데 사용할 수 있습니다. 예를 들어:

자바

```
WebClient 클라이언트 = WebClient.create("https://example.org");

Mono<ResponseEntity<Person>> 결과 = client.get() .uri("/")
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON) .retrieve() .toEntity(Person.class);
```

코틀린

```
val 클라이언트 = WebClient.create("https://example.org")

값 결과 = client.get()
    .uri("/사람/{id}", id).accept(MediaType.APPLICATION_JSON) .retrieve() .toEntity<
    사람>().awaitSingle()
```

또는 본체만 가져오려면 다음을 수행합니다.

자바

```
 WebClient 클라이언트 = WebClient.create("https://example.org");

Mono<사람> 결과 = client.get()
    .uri("/person/{id}", id).accept(MediaType.APPLICATION_JSON)
    .검색하다()
    .bodyToMono(Person.class);
```

코틀린

```
val 클라이언트 = WebClient.create("https://example.org")

값 결과 = client.get()
    .uri("/person/{id}", id).accept(MediaType.APPLICATION_JSON)
    .검색하다()
    .awaitBody<사람>()
```

디코딩된 객체 스트림을 가져오려면 다음을 수행합니다.

자바

```
Flux<Quote> 결과 = client.get()
    .uri("/따옴표").accept(MediaType.TEXT_EVENT_STREAM)
    .검색하다()
    .bodyToFlux(Quote.class);
```

코틀린

```
값 결과 = client.get()
    .uri("/따옴표").accept(MediaType.TEXT_EVENT_STREAM)
    .검색하다()
    .bodyToFlow<인용>()
```

기본적으로 4xx 또는 5xx 응답은 다음을 위한 하위 클래스를 포함하여 [WebClientResponseException](#)을 발생시킵니다.

특정 HTTP 상태 코드. 오류 응답 처리를 사용자 정의하려면 [onStatus](#) 핸들러를 다음과 같이 사용하십시오.

다음과 같다:

자바

```
Mono<사람> 결과 = client.get()
    .uri("/person/{id}", id).accept(MediaType.APPLICATION_JSON)
    .검색하다()
    .onStatus(HttpStatus::is4xxClientError, 응답 -> ...)
    .onStatus(HttpStatus::is5xxServerError, 응답 -> ...)
    .bodyToMono(Person.class);
```

코틀린

```
값 결과 = client.get()
    .uri("/person/{id}", id).accept(MediaType.APPLICATION_JSON)
    .검색하다()
    .onStatus(HttpStatus::is4xxClientError) { ... }
    .onStatus(HttpStatus::is5xxServerError) { ... }
    .awaitBody<사람>()
```

2.3. 교환

`exchangeToMono ()` 및 `exchangeToFlux()` 메서드(또는 `awaitExchange {}` 및 `exchangeToFlow {}`

Kotlin에서) 디코딩과 같이 더 많은 제어가 필요한 고급 사례에 유용합니다.

응답 상태에 따라 다르게 응답:

자바

```
Mono<Person> entityMono = client.get()
    .uri ("/ 명 / 1")
    .accept(MediaType.APPLICATION_JSON)
    .exchangeToMono(응답 -> {
        if (response.statusCode().equals(HttpStatus.OK)) {
            반환 응답.bodyToMono(Person.class);
        }
        또 다른 {
            // 오류로 전환
            return response.createException().flatMap(모노::오류);
        }
    });
});
```

코틀린

```
val 엔터티 = client.get()
    .uri ("/ 명 / 1")
    .accept(MediaType.APPLICATION_JSON)
    .awaitExchange {
        if (response.statusCode() == HttpStatus.OK) {
            response.awaitBody<Person>() 반환
        }
        또 다른 {
            response.createExceptionAndAwait() 던지기
        }
    }
});
```

위를 사용할 때 반환된 **Mono** 또는 **Flux** 가 완료된 후 응답 본문을 확인하고 소비되지 않으면 메모리 및 연결 누수를 방지하기 위해 해제됩니다. 따라서 응답 더 다운스트림으로 디코딩할 수 없습니다. 디코딩 방법을 선언하는 것은 제공된 함수에 달려 있습니다. 필요한 경우 응답.

2.4. 요청 본문

요청 본문은 [ReactiveAdapterRegistry에서](#) 처리하는 모든 비동기 유형에서 인코딩할 수 있습니다.

[Mono](#) 또는 Kotlin Coroutines [Deferred](#) 와 같은 예는 다음과 같습니다.

자바

```
모노<사람> personMono = ...;

Mono<Void> 결과 = client.post()
    .uri ("/ 명 / {id}", id)
    .contentType(MediaType.APPLICATION_JSON)
    .body(personMono, Person.class)
    .검색하다()
    .bodyToMono(Void.class);
```

코틀린

```
val personDeferred: Deferred<Person> = ...

client.post()
    .uri ("/ 명 / {id}", id)
    .contentType(MediaType.APPLICATION_JSON)
    .body<Person>(personDeferred)
    .검색하다()
    .awaitBody<단위>()
```

다음 예제와 같이 객체 스트림을 인코딩할 수도 있습니다.

자바

```
Flux<사람> personFlux = ...;

Mono<Void> 결과 = client.post()
    .uri ("/ 명 / {id}", id)
    .contentType(MediaType.APPLICATION_STREAM_JSON)
    .body(personFlux, Person.class)
    .검색하다()
    .bodyToMono(Void.class);
```

코틀린

```
val 사람: 흐름<사람> = ...
client.post()
    .uri ("/ 명 / {id}", id)
    .contentType(MediaType.APPLICATION_JSON)
    .body(사람)
    .검색하다()
    .awaitBody<단위>()
```

또는 실제 값이 있는 경우 **bodyValue** 바로 가기 메서드를 다음과 같이 사용할 수 있습니다.

다음 예는 다음을 보여줍니다.

자바

```
사람 사람 = ...
...
Mono<Void> 결과 = client.post()
    .uri ("/ 명 / {id}", id)
    .contentType(MediaType.APPLICATION_JSON)
    .bodyValue(사람)
    .검색하다()
    .bodyToMono(Void.class);
```

코틀린

```
발 사람: 사람 = ...
...
client.post()
    .uri ("/ 명 / {id}", id)
    .contentType(MediaType.APPLICATION_JSON)
    .bodyValue(사람)
    .검색하다()
    .awaitBody<단위>()
```

2.4.1. 양식 데이터

양식 데이터를 보내기 위해 **MultiValueMap<String, String>** 을 본문으로 제공할 수 있습니다. 참고

내용은 **FormHttpMessageWriter** 에 의해 자동으로 **application/x-www-form-urlencoded** 로 설정됩니다. 그만큼

다음 예제에서는 **MultiValueMap<String, String>**을 사용하는 방법을 보여줍니다.

자바

```
MultiValueMap<문자열, 문자열> formData = ...;

Mono<Void> 결과 = client.post()
    .uri("/경로", 아이디)
    .bodyValue(formData)
    .검색하다()
    .bodyToMono(Void.class);
```

코틀린

```
val formData: MultiValueMap<문자열, 문자열> = ...
client.post()
    .uri("/경로", 아이디)
    .bodyValue(formData)
    .검색하다()
    .awaitBody<단위>()
```

다음 예제와 같이 **BodyInserters** 를 사용하여 인라인으로 양식 데이터를 제공할 수도 있습니다.

자바

```
정적 org.springframework.web.reactive.function.BodyInserters.* 가져오기;

Mono<Void> 결과 = client.post()
    .uri("/경로", 아이디)
    .body(fromFormData("k1", "v1").with("k2", "v2"))
    .검색하다()
    .bodyToMono(Void.class);
```

코틀린

```
org.springframework.web.reactive.function.BodyInserters.* 가져오기

client.post()
    .uri("/경로", 아이디)
    .body(fromFormData("k1", "v1").with("k2", "v2"))
    .검색하다()
    .awaitBody<단위>()
```

2.4.2. 멀티파트 데이터

멀티파트 데이터를 보내려면 값이 다음 중 하나인 **MultiValueMap<String, ?>** 을 제공해야 합니다.

부분 콘텐츠를 나타내는 **개체** 인스턴스 또는 콘텐츠를 나타내는 **HttpEntity** 인스턴스 및

부품에 대한 헤더. **MultipartBodyBuilder** 는 멀티파트 요청을 준비하는 편리한 API를 제공합니다.

다음 예는 **MultiValueMap<String, ?>**을 생성하는 방법을 보여줍니다.

자바

```
MultipartBodyBuilder 빌더 = new MultipartBodyBuilder();
builder.part("필드파트", "필드값"); builder.part("filePart1", new
FileSystemResource("...logo.png")); builder.part("jsonPart", new Person("제
이슨")); builder.part("myPart", 부분); // 서버 요청의 일부
```

```
MultiValueMap<문자열, HttpEntity<?>> 부분 = builder.build();
```

코틀린

```
val builder = MultipartBodyBuilder().apply
{ part("fieldPart", "fieldValue") part("filePart1", new
FileSystemResource("...logo.png")) part("jsonPart", new Person("제이
슨")) } part("myPart", part) // 서버 요청의 일부 }
```

```
val 부품 = builder.build()
```

대부분의 경우 각 부분에 대해 **Content-Type** 을 지정할 필요가 없습니다 . 콘텐츠 유형은 직렬화하도록 선택한 **HttpMessageWriter** 에 따라 자동으로 결정 되거나 **리소스** 의 경우 파일 확장자를 기반으로 결정됩니다. 필요한 경우 오버로드된 빌더 **부분** 메서드 중 하나 를 통해 각 부분에 사용할 **MediaType** 을 명시적으로 제공할 수 있습니다 .

MultiValueMap 이 준비 되면 **WebClient** 에 전달하는 가장 쉬운 방법 은 다음 예제와 같이 **body** 메서드를 사용하는 것입니다.

자바

```
MultipartBodyBuilder 빌더 = ...;
Mono<Void> 결과 = client.post() .uri("/")
경로",
id) .body(builder.build()) .retrieve() .bodyToMono(Void.class);
```

코틀린

```
val 빌더: MultipartBodyBuilder =
...
client.post() .uri("/")
경로",
id) .body(builder.build()) .retrieve() .awaitBody<단위>()
```

MultiValueMap에 문자열이 아닌 값이 하나 이상 포함되어 있으면 일반 형식도 나타낼 수 있습니다.

데이터(즉, application/x-www-form-urlencoded)가 있는 경우 **Content-Type**을 다음으로 설정할 필요가 없습니다.

멀티파트/폼 데이터. **MultipartBodyBuilder**를 사용할 때 항상 그렇습니다.

HttpEntity 래퍼.

MultipartBodyBuilder의 대안으로 멀티파트 콘텐츠, 인라인 스타일,

다음 예제와 같이 내장 **BodyInserters**를 통해

자바

```
정적 org.springframework.web.reactive.function.BodyInserters.* 가져오기;
```

```
Mono<Void> 결과 = client.post()
    .uri("/경로", 아이디)
    .body(fromMultipartData("fieldPart", "value").with("filePart", 리소스))
    .검색하다()
    .bodyToMono(Void.class);
```

코틀린

```
org.springframework.web.reactive.function.BodyInserters.* 가져오기
```

```
client.post()
    .uri("/경로", 아이디)
    .body(fromMultipartData("fieldPart", "value").with("filePart", 리소스))
    .검색하다()
    .awaitBody<단위>()
```

2.5. 필터

WebClient.Builder를 통해 클라이언트 필터 (**ExchangeFilterFunction**)를 등록하여 다음을 수행할 수 있습니다.

다음 예제와 같이 요청을 가로채고 수정합니다.

자바

```
WebClient 클라이언트 = WebClient.builder()
    .filter((요청, 다음) -> {

        필터링된 ClientRequest = ClientRequest.from(요청)
            .header("푸", "바")
            .짓다();

        return next.exchange(필터링됨);
    })
    .짓다();
```

코틀린

```
val 클라이언트 = WebClient.builder()
    .filter { 요청, 다음 ->

        필터링된 값 = ClientRequest.from(요청)
            .header("푸", "바")
            .진다()

        next.exchange(필터링됨)
    }
    .진다()
```

이는 인증과 같은 교차 문제에 사용할 수 있습니다. 다음 예제에서는 정적 팩토리 메소드를 통한 기본 인증을 위한 필터:

자바

정적 가져오기
`org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
양이온;`

```
WebClient 클라이언트 = WebClient.builder()
    .filter(basicAuthentication("사용자", "비밀번호"))
    .진다();
```

코틀린

수입
`org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
양이온`

```
val 클라이언트 = WebClient.builder()
    .filter(basicAuthentication("사용자", "비밀번호"))
    .진다()
```

기존 **WebClient** 인스턴스 를 변경하여 필터를 추가하거나 제거할 수 있습니다.
 원래 인스턴스에 영향을 주지 않는 **WebClient** 인스턴스입니다. 예를 들어:

자바

정적 가져오기
 org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
 양이온;

```
 WebClient 클라이언트 = webClient.mutate()  

    .filters(필터 목록 -> {  

        filterList.add(0, basicAuthentication("사용자", "비밀번호"));  

    })  

    .짓다();
```

코틀린

```
val 클라이언트 = webClient.mutate()  

    .filters { it.add(0, basicAuthentication("사용자", "비밀번호")) }  

    .짓다()
```

WebClient 는 **ExchangeFunction**이 뒤따르는 필터 체인 주변의 얇은 외관입니다. 그것은 제공합니다

요청을 하고, 상위 수준의 개체와 주고받는 인코딩을 위한 워크플로, 그리고

응답 콘텐츠는 항상 소비됩니다. 필터가 어떤 식으로든 응답을 처리할 때 각별히 주의하십시오.

항상 콘텐츠를 소비하거나 그렇지 않으면 콘텐츠를 다운스트림으로 전파해야 합니다.

동일한 것을 보장하는 **WebClient**. 아래는 **UNAUTHORIZED** 상태 코드를 처리하는 필터 이지만

예상 여부에 관계없이 모든 응답 콘텐츠가 릴리스되도록 합니다.

자바

```
공개 ExchangeFilterFunction 갱신 토큰 필터() {  

    반환(요청, 다음) -> next.exchange(요청).flatMap(응답 -> {  

        if (response.statusCode().value() == HttpStatus.UNAUTHORIZED.value()) {  

            반환 응답.releaseBody()  

            .then(renewToken())  

            .flatMap(토큰 -> {  

                클라이언트 요청 새로운 요청 =  

                ClientRequest.from(요청).build();  

                return next.exchange(newRequest);  

            });
        } 또 다른 {  

            반환 Mono.just(응답);  

        }
    });
}
```

코틀린

```

재미 갱신TokenFilter(): ExchangeFilterFunction? {
    return ExchangeFilterFunction { 요청: ClientRequest?, 다음: ExchangeFunction ->
        next.exchange(요청!!).flatMap { 응답: ClientResponse ->
            if (response.statusCode().value() == HttpStatus.UNAUTHORIZED.value()) {
                return@flatMap 응답.releaseBody()
                    .then(renewToken())
                    .flatMap { 토큰: 문자열? ->
                        val newRequest = ClientRequest.from(요청).빌드()
                        next.exchange(newRequest)
                    }
            } 또 다른 {
                return@flatMap Mono.just(응답)
            }
        }
    }
}

```

2.6. 속성

요청에 속성을 추가할 수 있습니다. 를 통해 정보를 전달하려는 경우 편리합니다.
필터 체인을 만들고 주어진 요청에 대한 필터의 동작에 영향을 줍니다. 예를 들어:

자바

```

WebClient 클라이언트 = WebClient.builder()
    .filter((요청, 다음) -> {
        선택적<객체> usr = request.attribute("myAttribute");
        // ...
    })
    .짓다();

client.get().uri("https://example.org/")
    .attribute("myAttribute", "...")
    .검색하다()
    .bodyToMono(Void.class);

}

```

코틀린

```

val 클라이언트 = WebClient.builder()
    .filter { 요청, _ ->
        request.attributes()["myAttribute"];
        // ...
    }
    .짓다()

client.get().uri("https://example.org/")
    .attribute("myAttribute", "...")
    .검색하다()
    .awaitBody<단위 }()

```

`WebClient.Builder` 수준에서 전역적으로 `defaultRequest` 콜백을 구성할 수 있습니다.

예를 들어 Spring MVC에서 사용할 수 있는 모든 요청에 속성을 삽입할 수 있습니다.

`ThreadLocal` 데이터를 기반으로 요청 속성을 채우는 애플리케이션.

2.7. 문맥

속성은 필터 체인에 정보를 전달하는 편리한 방법을 제공하지만 영향을 미칠 뿐입니다.

현재 요청. 추가 요청에 전파되는 정보를 전달하려는 경우

예를 들어 `flatMap`을 통해 중첩되거나, 예를 들어 `concatMap`을 통해 실행된 후에는 Reactor를 사용해야 합니다.
문맥.

리액터 컨텍스트는 모든 리액티브 체인에 적용하기 위해 리액티브 체인의 끝에 채워져야 합니다.

작업. 예를 들어:

자바

```

WebClient 클라이언트 = WebClient.builder()
    .filter((요청, 다음) ->
        Mono.deferContextual(contextView -> {
            문자열 값 = contextView.get("foo");
            // ...
        }))
    .짓다();

client.get().uri("https://example.org/")
    .검색하다()
    .bodyToMono(String.class)
    .flatMap(본체 -> {
        // 중첩 요청 수행(컨텍스트가 자동으로 전파됨)...
    })
    .contextWrite(context -> context.put("foo", ...));

```

2.8. 동기 사용

WebClient 는 결과에 대해 끝에서 차단하여 동기식 스타일로 사용할 수 있습니다.

자바

```
사람 사람 = client.get().uri("/사람/{id}", i).retrieve()
    .bodyToMono(Person.class)
    .차단하다();

List<Person> people = client.get().uri("/persons").retrieve()
    .bodyToFlux(Person.class)
    .collectList()
    .차단하다();
```

코틀린

```
발 사람 = runBlocking {
    client.get().uri("/사람/{id}", i).retrieve()
        .awaitBody<사람>()
}

발 명 = runBlocking {
    client.get().uri("/사람").retrieve()
        .bodyToFlow<사람>()
        .toList()
}
```

그러나 여러 번 호출해야 하는 경우 각 응답에 대해 차단하지 않는 것이 더 효율적입니다.

개별적으로, 대신 결합된 결과를 기다립니다.

자바

```
Mono<Person> personMono = client.get().uri("/person/{id}", personId)
    .retrieve().bodyToMono(Person.class);

Mono<목록<취미>> hobbiesMono = client.get().uri("/person/{id}/hobbies", personId)
    .retrieve().bodyToFlux(Hobby.class).collectList();

Map<String, Object> data = Mono.zip(personMono, hobbiesMono, (사람, 취미) -> {
    Map<문자열, 문자열> map = new LinkedHashMap<>();
    map.put("사람", 사람);
    map.put("취미", 취미);
    리턴 맵;
})
    .차단하다();
```

코틀린

```

val 데이터 = runBlocking {
    발 personDeferred = 비동기 {
        client.get().uri("/사람/{id}", personId)
            .retrieve().awaitBody<사람>()
    }

    val hobbiesDeferred = 비동기 {
        client.get().uri("/사람/{id}/취미", personId)
            .retrieve().bodyToFlow<취미>().toList()
    }

    mapOf("사람" to personDeferred.await(), "취미" to
    취미Deferred.await())
}

```

위의 내용은 하나의 예일 뿐입니다. 퍼팅을 위한 다른 패턴과 연산자가 많이 있습니다.

많은 원격 호출(잠재적으로 일부 중첩, 상호 의존적)을 끝까지 차단하지 않고 만드는 반응 파이프라인을 함께 사용합니다.



[Flux](#) 또는 [Mono](#)를 사용 하면 Spring MVC 또는 Spring에서 차단할 필요가 없습니다.

WebFlux 컨트롤러. 컨트롤러에서 결과 반응 유형을 반환하기만 하면 됩니다.

방법. Kotlin Coroutine 및 Spring WebFlux에도 동일한 원칙이 적용됩니다.

일시 중단 기능을 사용하거나 컨트롤러 메서드에서 [Flow](#) 를 반환합니다.

2.9. 테스트

[WebClient](#) 를 사용하는 코드를 테스트하려면 [OkHttp](#) 와 같은 모의 웹 서버를 사용할 수 있습니다.

[목웹서버](#). 사용 예를 보려면 [WebClientIntegrationTests](#) 를 확인하십시오. 봄에

프레임워크 테스트 스위트 또는 [정적 서버](#) OkHttp 저장소의 샘플.

3장. 웹소켓

[서블릿 스택과 동일](#)

참조 문서의 이 부분에서는 반응 스택 WebSocket 메시징에 대한 지원을 다룹니다.

3.1. 웹소켓 소개

WebSocket 프로토콜, [RFC 6455](#), 단일 TCP 연결을 통해 클라이언트와 서버 간에 전이중 양방향 통신 채널을 설정하는 표준화된 방법을 제공합니다. HTTP와 다른 TCP 프로토콜이지만 포트 80 및 443을 사용하고 기존 방화벽 규칙을 다시 사용할 수 있도록 HTTP를 통해 작동하도록 설계되었습니다.

WebSocket 상호 작용은 HTTP [업그레이드](#) 헤더를 사용하여 업그레이드하거나 이 경우 WebSocket 프로토콜로 전환 하는 HTTP 요청으로 시작됩니다. 다음 예에서는 이러한 상호 작용을 보여줍니다.

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1 호스트:  
localhost:8080 업그레이드: websocket ① 연결: 업그레이드 ②
```

```
Sec-WebSocket-Key: Uc9l9TMkWGhFHD2qnFHltg==  
Sec-WebSocket-Protocol: v10.stomp, v11.stomp Sec-  
WebSocket-Version: 13 출처: http://localhost:8080
```

① [업그레이드](#) 헤더 .

② [업그레이드](#) 연결 을 사용합니다 .

일반적인 200 상태 코드 대신 WebSocket을 지원하는 서버는 다음과 유사한 출력을 반환합니다.

```
HTTP/1.1 101 스위칭 프로토콜 ① 업그레이드:  
websocket 연결: 업그레이드 Sec-WebSocket-  
Accept: 1qVdfYHU9hP0l4JYYNxF623Gzn0=  
Sec-WebSocket-Protocol: v10.stomp
```

① 프로토콜 스위치

성공적인 핸드셰이크 후 HTTP 업그레이드 요청의 기반이 되는 TCP 소켓은 클라이언트와 서버 모두 계속해서 메시지를 보내고 받을 수 있도록 열려 있습니다.

WebSocket의 작동 방식에 대한 완전한 소개는 이 문서의 범위를 벗어납니다. RFC 6455, HTML5의 WebSocket 장 또는 웹에 있는 많은 소개 및 자습서를 참조하십시오.

WebSocket 서버가 웹 서버(예: nginx) 뒤에서 실행 중인 경우 WebSocket 업그레이드 요청을 WebSocket 서버로 전달하도록 구성해야 할 수 있습니다. 마찬가지로, 만약

응용 프로그램이 클라우드 환경에서 실행되는 경우 WebSocket 지원과 관련된 클라우드 공급자의 지침을 확인하십시오.

3.1.1. HTTP 대 WebSocket

WebSocket이 HTTP와 호환되도록 설계되고 HTTP 요청으로 시작하더라도 두 프로토콜이 매우 다른 아키텍처와 애플리케이션 프로그래밍 모델로 이어진다는 점을 이해하는 것이 중요합니다.

HTTP 및 REST에서 애플리케이션은 많은 URL로 모델링됩니다. 애플리케이션과 상호 작용하기 위해 클라이언트는 요청-응답 스타일의 해당 URL에 액세스합니다. 서버는 HTTP URL, 메서드 및 헤더를 기반으로 요청을 적절한 처리기로 라우팅합니다.

대조적으로 WebSocket에는 일반적으로 초기 연결을 위한 URL이 하나만 있습니다. 결과적으로 모든 응용 프로그램 메시지는 동일한 TCP 연결에서 흐릅니다. 이것은 완전히 다른 비동기식 이벤트 기반 메시징 아키텍처를 나타냅니다.

WebSocket은 HTTP와 달리 메시지 내용에 의미 체계를 지정하지 않는 저수준 전송 프로토콜이기도 합니다. 즉, 클라이언트와 서버가 메시지 의미 체계에 동의하지 않는 한 메시지를 라우팅하거나 처리할 방법이 없습니다.

WebSocket 클라이언트와 서버는 HTTP 핸드셰이크 요청의 **Sec-WebSocket-Protocol** 헤더를 통해 더 높은 수준의 메시징 프로토콜(예: STOMP) 사용을 협상할 수 있습니다. 그것이 없다면 그들은 그들 자신의 관례를 마련해야 합니다.

3.1.2. WebSocket을 사용해야 하는 경우

WebSocket은 웹 페이지를 동적이고 대화식으로 만들 수 있습니다. 그러나 많은 경우 Ajax와 HTTP 스트리밍 또는 긴 폴링의 조합이 간단하고 효과적인 솔루션을 제공할 수 있습니다.

예를 들어, 뉴스, 메일 및 소셜 피드는 동적으로 업데이트되어야 하지만 몇 분마다 업데이트하는 것이 완벽할 수 있습니다. 반면에 협업, 게임 및 금융 앱은 실시간에 훨씬 더 가까워야 합니다.

대기 시간만으로는 결정 요인이 아닙니다. 메시지 양이 비교적 적은 경우(예: 네트워크 오류 모니터링) HTTP 스트리밍 또는 폴링은 효과적인 솔루션을 제공할 수 있습니다. WebSocket을 사용하는 가장 좋은 사례는 낮은 대기 시간, 높은 빈도 및 높은 볼륨의 조합입니다.

또한 인터넷을 통해 제어할 수 없는 제한적인 프록시는 **업그레이드** 헤더를 전달하도록 구성되지 않았거나 유휴 상태로 나타나는 오래 지속되는 연결을 닫기 때문에 WebSocket 상호 작용을 방해할 수 있습니다. 이는 방화벽 내에서 내부 응용 프로그램에 WebSocket을 사용하는 것이 공개 응용 프로그램에 사용하는 것보다 더 간단한 결정임을 의미합니다.

3.2. 웹소켓 API

[서블릿 스택과 동일](#)

Spring Framework는 WebSocket 메시지를 처리하는 클라이언트 및 서버 측 애플리케이션을 작성하는 데 사용할 수 있는 WebSocket API를 제공합니다.

3.2.1. 섬기는 사람

서블릿 스택과 동일

WebSocket 서버를 생성하려면 먼저 `WebSocketHandler`를 생성할 수 있습니다. 다음 예에서는 그렇게 하는 방법을 보여줍니다.

자바

```
org.springframework.web.reactive.socket.WebSocketHandler 가져오기;
org.springframework.web.reactive.socket.WebSocketSession 가져오기;

공개 클래스 MyWebSocketHandler는 WebSocketHandler를 구현합니다. {

@Override
public Mono<Void> handle(WebSocketSession session) { // ...

}}
```

코틀린

```
org.springframework.web.reactive.socket.WebSocketHandler 가져오기
org.springframework.web.reactive.socket.WebSocketSession 가져오기

클래스 MyWebSocketHandler : WebSocketHandler {

재미있는 핸들 재정의(세션: WebSocketSession): Mono<Void> {
    // ...
}}
```

그런 다음 URL에 매핑할 수 있습니다.

자바

```
@Configuration
클래스 WebConfig {

    @Bean
    public HandlerMapping handlerMapping()
    { Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/경로", 새로운 MyWebSocketHandler()); 정수 순서 =
        -1; // 주석이 달린 컨트롤러 앞

        return new SimpleUrlHandlerMapping(지도, 순서);
    }
}
```

코틀린

```
@Configuration
클래스 WebConfig {

    @Bean
    fun handlerMapping(): HandlerMapping { val map
        = mapOf("/path" to MyWebSocketHandler()) val order =
        -1 // 주석이 달린 컨트롤러 이전

        SimpleUrlHandlerMapping(지도, 순서) 반환
    }
}
```

[WebFlux 구성](#) 을 사용하는 경우 더 이상 할 일이 없으며 WebFlux 구성을 사용하지 않는 경우 아래와 같이 [WebSocketHandlerAdapter](#) 를 선언해야 합니다.

자바

```
@Configuration
클래스 WebConfig {

    // ...

    @Bean 공
    개 WebSocketHandlerAdapter handlerAdapter() { return new
        WebSocketHandlerAdapter();
    }
}
```

코틀린

```

@Configuration
클래스 WebConfig {

// ...

@Bean 재
미있는 handlerAdapter() = WebSocketHandlerAdapter() }

```

3.2.2. 웹소켓 핸들러

`WebSocketHandler`의 `핸들` 메서드는 `WebSocketSession`을 취하고 `Mono<Void>`를 반환 하여 다음 을 나타냅니다.

세션의 애플리케이션 처리가 완료될 때. 세션은 인바운드 및 아웃바운드 메시지에 대한 두 가지 스트림을 통해 처리됩니다. 다음 표에서는 스트림을 처리하는 두 가지 방법을 설명합니다.

WebSocketSession 메서드	설명
<code>Flux<WebSocketMessage> 수신()</code>	인바운드 메시지 스트림에 대한 액세스를 제공하고 연결이 닫힐 때 완료됩니다.
<code>Mono<Void> send(Publisher<WebSocketMessage>)</code> 나가는	메시지에 대한 소스를 가져오고 메시지를 작성 하고 소스가 완료되고 쓰기가 완료되면 완료 되는 <code>Mono<Void></code> 를 반환합니다.

`WebSocketHandler`는 인바운드 및 아웃바운드 스트림을 통합 흐름으로 구성하고 해당 흐름의 완료를 반영 하는 `Mono<Void>`를 반환해야 합니다. 애플리케이션 요구 사항에 따라 다음과 같은 경우 통합 흐름이 완료됩니다.

- 인바운드 또는 아웃바운드 메시지 스트림이 완료됩니다.
- 인바운드 스트림은 완료되고(즉, 연결이 닫힘) 아웃바운드 스트림은 무한.
- 선택한 지점에서 `WebSocketSession`의 `close` 메소드를 통해.

인바운드 및 아웃바운드 메시지 스트림이 함께 구성되면 Reactive Streams가 활동 종료 신호를 보내기 때문에 연결이 열려 있는지 확인할 필요가 없습니다. 인바운드 스트림은 완료 또는 오류 신호를 수신하고 아웃바운드 스트림은 취소 신호를 수신합니다.

핸들러의 가장 기본적인 구현은 인바운드 스트림을 처리하는 것입니다. 다음 예에서는 이러한 구현을 보여줍니다.

자바

```
클래스 ExampleHandler는 WebSocketHandler를 구현합니다. {
```

@우세하다

```
공개 Mono<Void> 핸들(WebSocketSession 세션) {
    session.receive() .doOnNext(메시지) ①
        지 -> {
            // ...
        } ②
        .concatMap(메시지 -> {
            // ...
        }) ③
        .그 다음에(); ④
}
```

- ① 수신 메시지 스트림에 액세스합니다.
- ② 각 메시지에 대해 무언가를 하십시오.
- ③ 메시지 내용을 사용하는 중첩된 비동기 작업을 수행합니다.
- ④ 수신 완료 시 완료 되는 **Mono<Void>** 를 반환합니다 .

코틀린

```
클래스 예제 핸들러 : 웹소켓 핸들러 {
```

```
재미있는 핸들 재정의(세션: WebSocketSession): Mono<Void> {
    반환 session.receive() .doOnNext ①
        {
            // ...
        } ②
        .concat맵 {
            // ...
        } ③
        .그 다음에() ④
}
```

- ① 수신 메시지 스트림에 액세스합니다.
- ② 각 메시지에 대해 무언가를 하십시오.
- ③ 메시지 내용을 사용하는 중첩된 비동기 작업을 수행합니다.
- ④ 수신 완료 시 완료 되는 **Mono<Void>** 를 반환합니다 .



중첩된 비동기 작업의 경우 다음에서 **message.retain()** 을 호출해야 할 수 있습니다.

풀링된 데이터 버퍼를 사용하는 기본 서버(예: Netty). 그렇지 않으면,
데이터 버퍼는 데이터를 읽을 기회가 있기 전에 해제될 수 있습니다. 을위한
더 많은 배경 지식은 [데이터 버퍼 및 코덱을 참조하세요.](#)

다음 구현은 인바운드 및 아웃바운드 스트림을 결합합니다.

자바

```
클래스 ExampleHandler는 WebSocketHandler를 구현합니다. {

    @우세하다

    공개 Mono<Void> 핸들(WebSocketSession 세션) {

        Flux<WebSocketMessage> 출력 =
            session.receive() .doOnNext(메시지 -> {
                // ...
            })
            .concatMap(메시지 -> {
                // ...
            })
            .map(값 -> session.textMessage("에코 " + 값)); ②

        반환 session.send(출력); ③
    }
}
```

- ① 인바운드 메시지 스트림을 처리합니다.
- ② 아웃바운드 메시지를 생성하여 결합된 흐름을 생성합니다.
- ③ 계속 받는 동안 완료되지 않은 **Mono<Void>** 를 반환합니다.

코틀린

```
클래스 예제 핸들러 : 웹소켓 핸들러 {

    재미있는 핸들 재정의(세션: WebSocketSession): Mono<Void> {

        값 출력 = session.receive() .doOnNext
            {
                // ...
            }
            .concat맵 {
                // ...
            }
            .map { session.textMessage("에코 $it") } ②

        session.send 반환(출력)
    }
}
```

- ① 인바운드 메시지 스트림을 처리합니다.
- ② 아웃바운드 메시지를 생성하여 결합된 흐름을 생성합니다.
- ③ 계속 받는 동안 완료되지 않은 **Mono<Void>** 를 반환합니다.

인바운드 및 아웃바운드 스트림은 독립적일 수 있으며 완료를 위해서만 결합됩니다.
다음 예는 다음을 보여줍니다.

자바

```
클래스 ExampleHandler는 WebSocketHandler를 구현합니다. {
```

```
@우세하다
```

```
공개 Mono<Void> 핸들(WebSocketSession 세션) {
```

```
    Mono<Void> 입력 =
        session.receive() .doOnNext(메시지 -> {
            // ...
        })
        .concatMap(메시지 -> {
            // ...
        })
        .그 다음에();
```

①

```
    Flux<String> 소스 = ... ;
    Mono<Void> 출력 = session.send(source.map(session::textMessage)); ②
```

```
    반환 Mono.zip(입력, 출력).then();
```

③

```
}
```

```
}
```

① 인바운드 메시지 스트림을 처리합니다.

② 발신 메시지를 보냅니다.

③ 스트림에 합류하고 스트림 이 끝날 때 완료 되는 **Mono<Void>** 를 반환합니다.

코틀린

```

클래스 예제 핸들러 : 웹소켓 핸들러 {

    재미있는 핸들 재정의(세션: WebSocketSession): Mono<Void> {

        값 입력 = session.receive() .doOnNext
            {
                // ...
            }
        .concat맵 {
            // ...
        }
        .그 다음에()

        val 소스: Flux<String> = val 출 ...  

        력 = session.send(source.map(session::textMessage)) ②

        반환 Mono.zip(입력, 출력).then()
    }
}

```

① 인바운드 메시지 스트림을 처리합니다.

② 발신 메시지를 보냅니다.

③ 스트림에 합류하고 스트림 이 끝날 때 완료 되는 **Mono<Void>** 를 반환합니다.

3.2.3. 데이터 버퍼

DataBuffer 는 WebFlux의 바이트 버퍼를 나타냅니다. 참조의 Spring Core 부분

[데이터 버퍼 및 코덱](#) 에 대한 섹션에서 이에 대해 자세히 설명합니다. 이해해야 할 핵심 포인트는 Netty와 같은 일부 서버에서는 바이트 버퍼가 폴링되고 참조 카운트되며 다음과 같은 경우 해제되어야 합니다. 메모리 누수를 피하기 위해 소비됩니다.

Netty에서 실행할 때 애플리케이션은 **DataBufferUtils.retain(dataBuffer)** 을 사용해야 합니다.

해제되지 않았는지 확인하기 위해 입력 데이터 버퍼를 유지하고 이후에 사용

버퍼가 소모될 때 **DataBufferUtils.release(dataBuffer)** .

3.2.4. 악수

[서블릿 스택과 동일](#)

WebSocketHandlerAdapter 는 WebSocketService에 위임합니다. 기본적으로 다음 인스턴스입니다.

WebSocket 요청에 대한 기본 검사를 수행한 다음 사용하는 **HandshakeWebSocketService**

사용 중인 서버에 대한 **RequestUpgradeStrategy** 입니다. 현재 Reactor Netty에 대한 지원이 내장되어 있으며, Tomcat, Jetty 및 Undertow.

HandshakeWebSocketService 는 설정을 허용하는 **sessionAttributePredicate** 속성을 노출합니다 .

WebSession 에서 속성을 추출하여 속성에 삽입하는 **Predicate<String>**

웹소켓세션.

3.2.5. 서버 구성

서블릿 스택과 동일

각 서버에 대한 `RequestUpgradeStrategy` 는 기본 WebSocket 서버 엔진에 특정한 구성을 노출합니다. WebFlux Java 구성은 사용할 때 `WebFlux 구성` 의 해당 섹션에 표시된 대로 이러한 속성을 사용자 지정할 수 있습니다. 그렇지 않으면 WebFlux 구성은 사용하지 않는 경우 아래를 사용합니다.

자바

```
@Configuration
클래스 WebConfig {

    @Bean 공
    개 WebSocketHandlerAdapter handlerAdapter() {
        반환 새로운 WebSocketHandlerAdapter(webSocketService());
    }

    @Bean 공
    개 WebSocketService webSocketService() {
        TomcatRequestUpgradeStrategy 전략 = 새로운 TomcatRequestUpgradeStrategy(); 전
        략.setMaxSessionIdleTimeout(0L); 새로운 HandshakeWebSocketService(전략)를 반환합니다.

    }
}
```

코틀린

```
@Configuration
클래스 WebConfig {

    @Bean 재
    미있는 handlerAdapter() =
        WebSocketHandlerAdapter(webSocketService())

    @Bean 재
    미| webSocketService(): WebSocketService { 발 전략 =
        TomcatRequestUpgradeStrategy().apply {
            setMaxSessionIdleTimeout(0L)

        } HandshakeWebSocketService(전략) 반환
    }
}
```

사용 가능한 옵션을 보려면 서버의 업그레이드 전략을 확인하십시오. 현재는 Tomcat과 Jetty만이 이러한 옵션을 제공합니다.

3.2.6. CORS

서블릿 스택과 동일

CORS를 구성하고 WebSocket 끝점에 대한 액세스를 제한하는 가장 쉬운 방법은 [WebSocketHandler](#)는 [CorsConfigurationSource](#)를 구현하고 허용된 CorsConfiguration을 반환 합니다. 출처, 헤더 및 기타 세부 사항. 그렇게 할 수 없으면 [corsConfigurations](#)를 설정할 수도 있습니다. [SimpleUrlHandler](#)의 속성을 사용하여 URL 패턴으로 CORS 설정을 지정합니다. 둘 다 지정하면 [CorsConfiguration](#)에서 결합 방법을 사용하여 결합됩니다.

3.2.7. 고객

Spring WebFlux는 Reactor Netty에 대한 구현으로 [WebSocketClient](#) 추상화를 제공합니다. Tomcat, Jetty, Undertow 및 표준 Java(즉, JSR-356).



Tomcat 클라이언트는 효과적으로 표준 Java 클라이언트의 확장입니다. [WebSocketSession](#) 처리의 추가 기능을 활용하여 역할에 대한 메시지 수신을 일시 중단하는 Tomcat 전용 API.

WebSocket 세션을 시작하려면 클라이언트의 인스턴스를 만들고 [실행](#) 메서드를 사용할 수 있습니다.

자바

```
WebSocketClient 클라이언트 = 새로운 ReactorNettyWebSocketClient();
URI URL = 새로운 URI("ws://localhost:8080/경로");
client.execute(URL, 세션 ->
    session.receive()
        .doOnNext(System.out::println)
        .그 다음에());
```

코틀린

```
val 클라이언트 = ReactorNettyWebSocketClient()
val URL = URI("ws://localhost:8080/경로")
client.execute(url) { 세션 ->
    session.receive()
        .doOnNext(::println)
        .그 다음에()
}
```

Jetty와 같은 일부 클라이언트는 [수명 주기](#)를 구현하며 다음을 수행하기 전에 중지 및 시작해야 합니다. 그것을 써. 모든 클라이언트에는 기본 WebSocket 구성과 관련된 생성자 옵션이 있습니다. 고객.

4장. 테스트

[Spring MVC에서도 동일](#)

spring-test 모듈은 `ServerHttpRequest`, `ServerHttpResponse` 및 `ServerWebExchange` 의 모의 구현을 제공합니다 .
[Spring Web Reactive](#) 참조 모의 객체에 대한 토론을 위해.

웹 테스트 클라이언트 HTTP 서버 없이 `WebFlux` 애플리케이션 테스트를 지원하기 위해 이러한 모의 요청 및 응답 개체를 기반으로 합니다. 종단 간 통합 테스트에도 `WebTestClient` 를 사용할 수 있습니다 .

5장. RSocket

이 섹션에서는 RSocket 프로토콜에 대한 Spring Framework의 지원에 대해 설명합니다.

5.1. 개요

RSocket은 다음 상호 작용 모델 중 하나를 사용하여 TCP, WebSocket 및 기타 바이트 스트림 전송을 통한 다중 이중 통신을 위한 응용 프로그램 프로토콜입니다.

- **요청-응답** — 하나의 메시지를 보내고 하나를 다시 받습니다.
- **Request-Stream** — 하나의 메시지를 보내고 메시지 스트림을 다시 받습니다.
- **채널** — 양방향으로 메시지 스트림을 보냅니다.
- **Fire-and-Forget** — 단방향 메시지를 보냅니다.

초기 연결이 이루어지면 "클라이언트"와 "서버"의 구분은 양쪽이 대칭이 되고 양쪽이 위의 상호 작용 중 하나를 시작할 수 있기 때문에 손실됩니다. 이것이 프로토콜에서 참여 측을 "요청자" 및 "응답자"라고 부르는 반면 위의 상호 작용을 "요청 스트림" 또는 단순히 "요청"이라고 부르는 이유입니다.

다음은 RSocket 프로토콜의 주요 기능 및 이점입니다.

- **반응 스트림** 네트워크 경계를 가로지르는 의미 체계 - **Request-Stream** 및 **Channel** 과 같은 스트리밍 요청의 경우 역압 신호가 요청자와 응답자 사이를 이동하여 요청자가 소스에서 응답자의 속도를 늦추도록 하여 네트워크 계층 혼잡 제어에 대한 의존도를 줄입니다. 네트워크 수준 또는 모든 수준에서 버퍼링.
- **요청 제한** - 이 기능은 **LEASE** 프레임의 이름을 따서 "임대"로 명명되어 주어진 시간 동안 상대방이 허용하는 총 요청 수를 제한하기 위해 각 끝에서 보낼 수 있습니다. 임대는 주기적으로 갱신됩니다.
- **세션 재개** - 이것은 연결 손실을 위해 설계되었으며 일부 상태를 유지해야 합니다. 상태 관리는 응용 프로그램에 대해 투명하며 가능한 경우 생산자를 중지하고 필요한 상태의 양을 줄일 수 있는 역압과 함께 잘 작동합니다.
- 대용량 메시지의 단편화 및 재조립.
- Keepalive(하트비트).

RSocket에는 **구현**이 있습니다. 여러 언어로 자바 [라이브러리 프로젝트 리액터](#)를 기반으로 하며, 및 [원자로 네티](#) 운송을 위해 즉, 애 플리케이션에 있는 Reactive Streams Publishers의 신호가 네트워크를 통해 RSocket을 통해 투명하게 전파됩니다.

5.1.1. 프로토콜

RSocket의 장점 중 하나는 와이어에 대해 잘 정의된 동작과 읽기 쉬운 **사양**이 있다는 것입니다. 일부 프로토콜 **확장**과 함께 따라서 언어 구현 및 상위 수준 프레임워크 API와 상관없이 사양을 읽는 것이 좋습니다. 이 섹션은 다음을 제공합니다.

일부 컨텍스트를 설정하기 위한 간결한 개요.

연결 종

처음에 클라이언트는 TCP 또는 WebSocket과 같은 일부 저수준 스트리밍 전송을 통해 서버에 연결하고 **SETUP** 프레임을 서버에 전송하여 연결 매개변수를 설정합니다.

서버는 **SETUP** 프레임을 거부할 수 있지만 일반적으로 **SETUP** 이 요청 수를 제한하기 위해 임대 의미론의 사용을 나타내지 않는 한 일반적으로 전송(클라이언트의 경우) 및 수신(서버의 경우) 후에 요청을 시작할 수 있습니다. 이 경우 양측은 요청을 허용하기 위해 다른 쪽 끝에서 **LEASE** 프레임을 기다려야 합니다.

요구하다

연결이 설정되면 양측은 **REQUEST_RESPONSE**, **REQUEST_STREAM**, **REQUEST_CHANNEL** 또는 **REQUEST_FNF** 프레임 중 하나를 통해 요청을 시작할 수 있습니다. 각 프레임은 요청자에서 응답자에게 하나의 메시지를 전달합니다.

그러면 응답자는 응답 메시지와 함께 **PAYOUT** 프레임을 반환할 수 있으며 **REQUEST_CHANNEL** 의 경우 요청자는 더 많은 요청 메시지와 함께 **PAYOUT** 프레임을 보낼 수도 있습니다.

요청이 **Request-Stream** 및 **Channel** 과 같은 메시지 스트림을 포함하는 경우 응답자는 요청자의 요구 신호를 존중해야 합니다. 수요는 메시지의 수로 표현됩니다.

초기 수요는 **REQUEST_STREAM** 및 **REQUEST_CHANNEL** 프레임에 지정됩니다. 후속 요구는 **REQUEST_N** 프레임을 통해 신호됩니다.

각 측에서는 **METADATA_PUSH** 프레임을 통해 개별 요청이 아니라 전체 연결과 관련된 메타데이터 알림을 보낼 수도 있습니다.

메시지 형식

RSocket 메시지에는 데이터와 메타데이터가 포함됩니다. 메타데이터는 경로, 보안 토큰 등을 보내는 데 사용할 수 있습니다. 데이터와 메타데이터는 서로 다른 형식을 지정할 수 있습니다. 각각에 대한 MIME 유형은 **SETUP** 프레임에서 선언되고 지정된 연결의 모든 요청에 적용됩니다.

모든 메시지가 메타데이터를 가질 수 있지만 일반적으로 경로와 같은 메타데이터는 요청당이므로 요청의 첫 번째 메시지에만 포함됩니다(예: **REQUEST_RESPONSE**, **REQUEST_STREAM**, **REQUEST_CHANNEL** 또는 **REQUEST_FNF** 프레임 중 하나).

프로토콜 확장은 애플리케이션에서 사용할 공통 메타데이터 형식을 정의합니다.

- **복합 메타데이터**-- 여러 개의 독립적으로 형식이 지정된 메타데이터 항목.
- **라우팅** — 요청에 대한 경로.

5.1.2. 자바 구현

자바 구현 RSocket용은 [Project Reactor](#)를 기반으로 합니다. TCP 및 WebSocket의 전송은 [Reactor Netty](#)를 기반으로 합니다. Reactive Streams 라이브러리인 Reactor는 프로토콜 구현 작업을 단순화합니다. 애플리케이션의 경우 선언적 연산자 및 투명한 배업 지원과 함께 **Flux** 및 **Mono** 를 사용하는 것이 자연스럽습니다.

RSocket Java의 API는 의도적으로 최소한의 기본입니다. 프로토콜 기능에 중점을 두고 애플리케이션 프로그래밍 모델(예: RPC codegen 대 기타)을 더 높은 수준의 독립적인 문제로 남겨둡니다.

주요 계약 [io.rsocket.RSocket](#) 단일 메시지에 대한 약속을 나타내는 [Mono](#), 메시지 스트림을 나타내는 [Flux](#), 바이트 버퍼로 데이터 및 메타데이터에 액세스하여 실제 메시지를 [io.rsocket.Payload](#)로 네 가지 요청 상호 작용 유형을 모델링합니다. RSocket 계약은 대칭적으로 사용됩니다. 요청을 위해 애플리케이션에는 요청을 수행할 [RSocket](#)이 제공됩니다. 응답을 위해 애플리케이션은 요청을 처리하기 위해 [RSocket](#)을 구현합니다.

이것은 철저한 소개를 의미하지 않습니다. 대부분의 경우 Spring 애플리케이션은 API를 직접 사용할 필요가 없습니다. 그러나 Spring과 독립적인 RSocket을 보거나 실험하는 것이 중요할 수 있습니다. RSocket Java 리포지토리에는 여러 [샘플 앱](#)이 포함되어 있습니다. API 및 프로토콜 기능을 보여줍니다.

5.1.3. 스프링 지원

[spring-messaging](#) 모듈은 다음을 포함합니다 :

- [RSocketRequester](#) — 데이터 및 메타데이터 인코딩/디코딩을 사용하여 [io.rsocket.RSocket](#)을 통해 요청을 만드는 유창한 API입니다.
- [Annotated Responders](#) — [@MessageMapping](#) 응답을 위한 주석 처리기 메서드.

[spring-web](#) 모듈에는 RSocket 애플리케이션에 필요할 것 같은 Jackson CBOR/JSON 및 Protobuf와 같은 [인코더](#) 및 [디코더](#) 구현이 포함되어 있습니다. 또한 효율적인 경로 일치를 위해 연결할 수 있는 [PathPatternParser](#)도 포함합니다.

Spring Boot 2.2는 WebFlux 서버에서 WebSocket을 통해 RSocket을 노출하는 옵션을 포함하여 TCP 또는 WebSocket을 통해 RSocket 서버를 지원합니다. [RSocketRequester.Builder](#) 및 [RSocketStrategies](#)에 대한 클라이언트 지원 및 자동 구성도 있습니다. [RSocket 섹션](#)을 참조하십시오. 자세한 내용은 Spring Boot 참조를 참조하세요.

Spring Security 5.2는 RSocket 지원을 제공합니다.

Spring Integration 5.2는 RSocket 클라이언트 및 서버와 상호 작용하기 위한 인바운드 및 아웃바운드 게이트웨이를 제공합니다. 자세한 내용은 Spring 통합 참조 매뉴얼을 참조하십시오.

Spring Cloud Gateway는 RSocket 연결을 지원합니다.

5.2. RSocket 요청자

[RSocketRequester](#)는 RSocket 요청을 수행하는 유창한 API를 제공하여 저수준 데이터 버퍼 대신 데이터 및 메타데이터에 대한 개체를 수락하고 반환합니다. 클라이언트에서 요청을 만들고 서버에서 요청을 만들기 위해 대칭적으로 사용할 수 있습니다.

5.2.1. 클라이언트 요청자

클라이언트 측에서 [RSocketRequester](#)를 얻으려면 연결 설정과 함께 RSocket SETUP 프레임을 보내는 것과 관련된 서버에 연결해야 합니다. [RSocketRequester](#)는 다음을 지원하는 빌더를 제공합니다.

SETUP 프레임에 대한 연결 설정을 포함하여 `io.rsocket.core.RSocketConnector`를 준비합니다.

다음은 기본 설정으로 연결하는 가장 기본적인 방법입니다.

자바

```
RSocketRequester 요청자 = RSocketRequester.builder().tcp("localhost", 7000);

URI URL = URI.create("https://example.org:8080/rsocket");
RSocketRequester 요청자 = RSocketRequester.builder().webSocket(url);
```

코틀린

```
발 요청자 = RSocketRequester.builder().tcp("localhost", 7000)

URI URL = URI.create("https://example.org:8080/rsocket"); val 요청
자 = RSocketRequester.builder().webSocket(url)
```

위의 항목은 즉시 연결되지 않습니다. 요청이 있으면 공유 연결이 투명하게 설정되어 사용됩니다.

연결 설정

`RSocketRequester.Builder`는 초기 **SETUP** 프레임을 사용자 지정하기 위해 다음을 제공합니다.

- `dataMimeType(MimeType)` — 연결의 데이터에 대한 MIME 유형을 설정합니다.
- `metadataMimeType(MimeType)` — 연결의 메타데이터에 대한 MIME 유형을 설정합니다.
- `setupData(Object)` — SETUP에 포함할 데이터입니다.
- `setupRoute(String, Object…)` — SETUP에 포함할 메타데이터의 경로입니다.
- `setupMetadata(Object, MimeType)` — SETUP에 포함할 기타 메타데이터 .

데이터의 경우 기본 MIME 유형은 첫 번째로 구성된 [디코더](#)에서 파생됩니다. 메타데이터의 경우 기본 MIME 유형은 [복합 메타데이터](#)입니다. 요청당 여러 메타데이터 값과 MIME 유형 쌍을 허용합니다. 일반적으로 둘 다 변경할 필요가 없습니다.

SETUP 프레임의 데이터 및 메타데이터는 선택 사항입니다. 서버 측에서는 [@ConnectMapping](#) 메서드를 사용하여 연결 시작과 **SETUP** 프레임의 내용을 처리할 수 있습니다. 메타데이터는 연결 수준 보안에 사용될 수 있습니다.

전략

`RSocketRequester.Builder`는 요청자 를 구성하기 위해 `RSocketStrategies`를 수락 합니다. 데이터 및 메타데이터 값의 (역)직렬화를 위한 인코더 및 디코더를 제공하려면 이것을 사용해야 합니다. 기본적으로 `String`, `byte[]`, `ByteBuffer`에 대한 [spring-core](#)의 기본 코덱만 등록됩니다. [spring-web](#)을 추가하면 다음과 같이 등록할 수 있는 더 많은 항목에 액세스할 수 있습니다.

자바

```
RSocketStrategies 전략 = RSocketStrategies.builder() .encoders(encoders
-> encoders.add(new Jackson2CborEncoder())).decoders(decoders ->
decoders.add(new Jackson2CborDecoder())).build();
```

```
RSocketRequester 요청자 = RSocketRequester.builder() .rsocketStrategies(전
략) .tcp("localhost", 7000);
```

코틀린

```
val 전략 = RSocketStrategies.builder()
    .encoders { it.add(Jackson2CborEncoder()) } .decoders
    { it.add(Jackson2CborDecoder()) } .build()
```

```
발 요청자 = RSocketRequester.builder() .rsocketStrategies(전
략) .tcp("localhost", 7000)
```

RSocketStrategies 는 재사용을 위해 설계되었습니다. 일부 시나리오(예: 동일한 애플리케이션의 클라이언트 및 서버)에서는 이를 Spring 구성에서 선언하는 것이 바람직할 수 있습니다.

클라이언트 응답자

RSocketRequester.Builder 를 사용하여 서버의 요청에 대한 응답자를 구성할 수 있습니다.

서버에서 사용되지만 다음과 같이 프로그래밍 방식으로 등록된 동일한 인프라를 기반으로 클라이언트 측 응답에 주석 처리기를 사용할 수 있습니다.

자바

```
RSocketStrategies 전략 = RSocketStrategies.builder() .routeMatcher(new
PathPatternRouteMatcher()) ① .build();
```

```
SocketAcceptor 응답자 =
RSocketMessageHandler.responder(전략, new ClientHandler()); ②
```

```
RSocketRequester 요청자 =
RSocketRequester.builder() .rsocketConnector(connector ->
connector.acceptor(responder)) ③ .tcp("localhost", 7000);
```

① 효율적인 경로 매칭을 위해 **spring-web** 이 있는 경우 **PathPatternRouteMatcher**를 사용 한다.

② **@MessageMapping** 및/또는 **@ConnectMapping** 메소드를 사용하여 클래스에서 응답자를 생성 합니다.

③ 응답자를 등록합니다.

코틀린

```
val 전략 = RSocketStrategies.builder()
    .routeMatcher(PathPatternRouteMatcher()) ① .build()
```

```
발 응답자 =
RSocketMessageHandler.responder(전략, 새로운 ClientHandler()); ②
```

```
val 요청자 = RSocketRequester.builder() .rsocketConnector
    { it.acceptor(responder) } ③ .tcp("localhost", 7000)
```

① 효율적인 경로 매칭을 위해 `spring-web` 이 있는 경우 `PathPatternRouteMatcher`를 사용 한다.

② `@MessageMapping` 및/또는 `@ConnectMapping` 메소드 를 사용하여 클래스에서 응답자를 생성 합니다.

③ 응답자를 등록합니다.

위의 내용은 클라이언트 응답자의 프로그래밍 방식 등록을 위해 설계된 바로 가기일 뿐입니다. 클라이언트 응답자가 Spring 구성에 있는 대체 시나리오의 경우 여전히 `RSocketMessageHandler` 를 Spring bean으로 선언하고 다음과 같이 적용할 수 있습니다.

자바

```
ApplicationContext 컨텍스트 = ... ;
```

```
RSocketMessageHandler 핸들러 = context.getBean(RSocketMessageHandler.class);
```

```
RSocketRequester 요청자 =
```

```
RSocketRequester.builder() .rsocketConnector(connector ->
    connector.acceptor(handler.responder())) .tcp("localhost", 7000);
```

코틀린

```
org.springframework.beans.factory.getBean 가져오기
```

```
val 컨텍스트: ApplicationContext = val 핸 ...
```

```
들러 = context.getBean<RSocketMessageHandler>()
```

```
발 요청자 = RSocketRequester.builder() .rsocketConnector
```

```
{ it.acceptor(handler.responder()) } .tcp("localhost", 7000)
```

위의 경우 `RSocketMessageHandler` 에서 `setHandlerPredicate` 를 사용하여 클라이언트 응답자를 감지하는 다른 전략으로 전환해야 할 수도 있습니다. 예를 들어 `@RSocketClientResponder` 대 기본 `@Controller` 와 같은 사용자 정의 주석을 기반으로 합니다. 이는 클라이언트와 서버가 있거나 동일한 응용 프로그램에 여러 클라이언트가 있는 시나리오에서 필요합니다.

프로그래밍 모델에 대한 자세한 내용은 주석이 [달린 응답자](#) 도 참조하세요 .

고급의

기본 [RSocketRequesterBuilder](#) 제공 콜백 예게 폭로하다 그만큼

연결 유지 간격, 세션에 대한 추가 구성 옵션을 위한 [io.rsocket.core.RSocketConnector](#)
재개, 인터셉터 등. 다음과 같이 해당 수준에서 옵션을 구성할 수 있습니다.

자바

```
RSocketRequester 요청자 = RSocketRequester.builder()
    .rsocketConnector(커넥터 -> {
        // ...
    })
    .tcp("로컬호스트", 7000);
```

코틀린

```
val 요청자 = RSocketRequester.builder()
    .rsocket커넥터 {
        //...
    }
    .tcp("로컬호스트", 7000)
```

5.2.2. 서버 요청자

서버에서 연결된 클라이언트로 요청하는 것은 요청자를 얻는 문제입니다.

서버에서 연결된 클라이언트.

주석 이 달린 응답자에서 [@ConnectMapping](#) 및 [@MessageMapping](#) 메서드는 다음을 지원합니다.

[RSocketRequester](#) 인수. 연결 요청자에 액세스하는 데 사용합니다. 명심하십시오

[@ConnectMapping](#) 메서드는 본질적으로 이전에 처리되어야 하는 [SETUP](#) 프레임의 핸들러입니다.

요청을 시작할 수 있습니다. 따라서 처음에 요청을 처리에서 분리해야 합니다. 을위한

예시:

자바

```
@ConnectMapping
Mono<Void> 핸들(RSocketRequester 요청자) {
    requester.route("상태").data("5")
        .retrieveFlux(StatusReport.class)
        .subscribe(바 -> { ①
            // ...
        });
    반환 ... ②
}
```

① 처리와 무관하게 비동기적으로 요청을 시작합니다.

② 처리 및 반환 완료 [Mono<Void>](#)를 수행합니다.

코틀린

```
@ConnectMapping
일시 중단 재미 핸들(요청자: RSocketRequester) { GlobalScope.launch {

    requester.route("status").data("5").retrieveFlow<StatusReport>().collect { ①
        // ...
    }
} // ...
② }
```

① 처리와 무관하게 비동기적으로 요청을 시작합니다.

② suspending 기능에서 처리합니다.

5.2.3. 요청

[클라이언트](#) 또는 [서버](#) 요청 자가 있으면 다음과 같이 요청할 수 있습니다.

자바

```
뷰박스 뷰박스 =
    ...
    Flux<AirportLocation> 위치 = requester.route("locate.radars.within") ①
        .data(viewBox)
        ② .retrieveFlux(AirportLocation.class); ③
```

① 요청 메시지의 메타데이터에 포함할 경로를 지정합니다.

② 요청 메시지에 대한 데이터를 제공합니다.

③ 예상 응답을 선언합니다.

코틀린

```
val viewBox: ViewBox =
    ...
    발 위치 = requester.route("locate.radars.within") ①
        .data(viewBox)
        ② .retrieveFlow<공항위치>() ③
```

① 요청 메시지의 메타데이터에 포함할 경로를 지정합니다.

② 요청 메시지에 대한 데이터를 제공합니다.

③ 예상 응답을 선언합니다.

상호 작용 유형은 입력 및 출력의 카디널리티에서 암시적으로 결정됩니다. 위의 예는 하나의 값이 전송되고 값의 스트림이 수신되기 때문에 **Request-Stream**입니다. 대부분의 경우 입력 및 출력 선택이 RSocket 상호 작용 유형과 응답자가 예상하는 입력 및 출력 유형과 일치하는 한 이에 대해 생각할 필요가 없습니다. 유효하지 않은 조합의 유일한 예는 다대일입니다.

data(Object) 메서드는 또한 **Flux** 및 **Mono**를 포함한 모든 Reactive Streams **Publisher** 와 **ReactiveAdapterRegistry**에 등록된 다른 값 생산자를 허용합니다 . 동일한 유형의 값을 생성하는 **Flux** 와 같은 다중 값 **게시자** 의 경우 오버로드 된 데이터 방법 중 하나를 사용하여 모든 요소에 대한 유형 검사 및 **인코더** 조회를 방지하는 것이 좋습니다.

```
data(객체 생성자, Class<?> elementClass); data(객체 생성자, ParameterizedTypeReference<?> elementTypeRef);
```

data (Object) 단계는 선택 사항입니다. 데이터를 보내지 않는 요청은 건너뛰세요.

자바

```
Mono<AirportLocation> 위치 =  
requester.route("find.radar.EWR") .retrieveMono(AirportLocation.class);
```

코틀린

```
org.springframework.messaging.rsocket.retrieveAndAwait 가져오기
```

```
발 위치 = requester.route("find.radar.EWR") .retrieveAndAwait<  
공항 위치>()
```

복합 메타데이터 를 사용하는 경우 추가 메타데이터 값을 추가할 수 있습니다. (기본값) 및 값이 등록된 **인코더에서 지원되는지 여부**. 예를 들어:

자바

```
문자열 securityToken = ... ;  
ViewBox viewBox = ... ;  
MimeType mimeType =MimeType.valueOf("message/x.rsocket.authentication.bearer.v0");  
  
Flux<AirportLocation> 위치 = requester.route("locate.radars.within")  
.metadata(securityToken,  
  
mimeType) .data(viewBox) .retrieveFlux(AirportLocation.class);
```

코틀린

```
org.springframework.messaging.rsocket.retrieveFlow 가져오기

val 요청자: RSocketRequester = ...
...
val securityToken: 문자열 = val ...
viewBox: ViewBox = val ...
MimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0")

발 위치 = requester.route("locate.radars.within")
    .metadata(securityToken, mimeType)
    .data(뷰박스)
    .retrieveFlow<공항 위치>()
```

Fire-and-Forget 의 경우 **Mono<Void>** 를 반환 하는 **send()** 메서드를 사용 합니다. **모노** 는 _ 메시지가 성공적으로 전송되었다는 것이지 처리되었다는 것은 아닙니다.

Metadata-Push 의 경우 **Mono<Void>** 반환 값 과 함께 **sendMetadata()** 메서드를 사용 합니다.

5.3. 주석이 달린 응답자

RSocket 응답자는 **@MessageMapping** 및 **@ConnectMapping** 메서드로 구현될 수 있습니다.
@MessageMapping 메서드는 개별 요청을 처리하는 반면 **@ConnectMapping** 메서드는 처리합니다 .
연결 수준 이벤트(설정 및 메타데이터 푸시). 주석이 달린 응답자가 지원됩니다.
대칭적으로, 서버 측에서 응답하고 클라이언트 측에서 응답합니다.

5.3.1. 서버 응답자

서버 측에서 주석이 달린 응답자를 사용하려면 Spring 에 **RSocketMessageHandler** 를 추가하십시오.
@MessageMapping 및 **@ConnectMapping** 메소드를 사용하여 **@Controller** 빈을 감지하는 구성 :

자바

```
@구성
정적 클래스 ServerConfig {
    ...
}

@콩
공개 RSocketMessageHandler rscketMessageHandler() {
    RSocketMessageHandler 핸들러 = 새로운 RSocketMessageHandler();
    핸들러.routeMatcher(새로운 PathPatternRouteMatcher());
    리턴 핸들러;
}
```

코틀린

```
@Configuration
클래스 ServerConfig {

    @Bean 재
    미 rsocketMessageHandler() = RSocketMessageHandler().apply { routeMatcher
        = PathPatternRouteMatcher()
    }
}
```

그런 다음 Java RSocket API를 통해 RSocket 서버를 시작하고 다음과 같이 응답자에 대한 **RSocketMessageHandler**를 연결합니다.

자바

```
ApplicationContext 컨텍스트 = ... ;
RSocketMessageHandler 핸들러 = context.getBean(RSocketMessageHandler.class);

CloseableChannel 서버 =

RSocketServer.create(handler.responder()) .bind(TcpServerTransport.create("localhost", 7000)) .block();
```

코틀린

```
org.springframework.beans.factory.getBean 가져오기

val 컨텍스트: ApplicationContext = val 핸 ...
들러 = context.getBean<RSocketMessageHandler>()

val 서버 = RSocketServer.create(handler.responder())
    .bind(TcpServerTransport.create("localhost",
    7000)) .awaitSingle()
```

RSocketMessageHandler는 **합성**을 지원합니다. 및 **라우팅** 기본적으로 메타데이터. 다른 MIME 유형으로 전환하거나 추가 메타데이터 MIME 유형을 등록해야 하는 경우 **MetadataExtractor**를 설정할 수 있습니다.

지원하는 메타데이터 및 데이터 형식에 필요한 **인코더** 및 **디코더** 인스턴스를 설정해야 합니다. 코덱 구현을 위해 **spring-web** 모듈이 필요할 것 입니다.

기본적으로 **SimpleRouteMatcher**는 AntPathMatcher를 통해 경로를 일치시키는 데 사용됩니다. 효율적인 경로 일치를 위해 **spring-web**에서 **PathPatternRouteMatcher**를 연결하는 것이 좋습니다. RSocket 경로는 계층적일 수 있지만 URL 경로는 아닙니다. 두 경로 일치자는 모두 "."를 사용하도록 구성됩니다. 기본적으로 구분 기호로 사용되며 HTTP URL과 같이 URL 디코딩이 없습니다.

RSocketMessageHandler는 동일한 프로세스에서 클라이언트와 서버 간에 구성을 공유해야 하는 경우에 유용할 수 있는 **RSocketStrategies**를 통해 구성할 수 있습니다.

자바

```

@구성
정적 클래스 ServerConfig {

    @콩
    공개 RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler 핸들러 = 새로운 RSocketMessageHandler();
        handler.setRSocketStrategies(rsocketStrategies());
        리턴 핸들러;
    }

    @콩
    공개 RSocketStrategies rsocketStrategies() {
        RSocketStrategies.builder() 반환
            .encoders(인코더 -> 인코더.add(new Jackson2CborEncoder()))
            .decoders(디코더 -> 디코더.add(new Jackson2CborDecoder()))
            .routeMatcher(새로운 PathPatternRouteMatcher())
            .짓다();
    }
}

```

코틀린

```

@구성
클래스 서버 구성 {

    @콩
    재미 rsocketMessageHandler() = RSocketMessageHandler().apply {
        rSocketStrategies = rsocketStrategies()
    }

    @콩
    재미있는 rsocketStrategies() = RSocketStrategies.builder()
        .인코더 { it.add(Jackson2CborEncoder()) }
        .decoders { it.add(Jackson2CborDecoder()) }
        .routeMatcher(PathPatternRouteMatcher())
        .짓다()
}

```

5.3.2. 클라이언트 응답자

클라이언트 측의 주석이 달린 응답자는 [RSocketRequester.Builder](#)에서 구성해야 합니다. 을위한 자세한 내용은 [클라이언트 응답자를 참조하십시오.](#)

5.3.3. @MessageMapping

[서버](#) 또는 [클라이언트](#) 응답자 구성이 설정 되면 [@MessageMapping](#) 메서드를 다음과 같이 사용할 수 있습니다.
다음과 같다:

자바

```
@Controller
public 클래스 RadarsController {

    @MessageMapping("locate.radars.within") public
    Flux<AirportLocation> 레이더(MapRequest 요청) {
        // ...
    }
}
```

코틀린

```
@Controller
클래스 RadarsController {

    @MessageMapping("locate.radars.within") 재미있
    는 레이더(요청: MapRequest): Flow<AirportLocation> {
        // ...
    }
}
```

위의 `@MessageMapping` 메서드는 "locate.radars.within" 경로가 있는 Request-Stream 상호 작용에 응답합니다. 다음 메서드 인수를 사용하는 옵션과 함께 유연한 메서드 서명을 지원합니다.

메서드 인수	설명
<code>@유효 탐색량</code>	요청의 페이로드입니다. 이것은 <code>Mono</code> 또는 <code>Flux</code> 와 같은 비동기 유형의 구체적인 값일 수 있습니다. 참고: 주석 사용은 선택 사항입니다. 단순 유형이 아니고 지원되는 다른 인수가 아닌 메소드 인수는 예상 페이로드로 간주됩니다.
<code>RSocket요청자</code>	원격 종단에 요청하기 위한 요청자입니다.
<code>@DestinationVariable</code> 매팅의 변수를 기반으로 경로에서 추출한 값 패턴, 예: <code>@MessageMapping("find.radar.{id}")</code> .	
<code>@머리글</code>	MetadataExtractor에 설명된 대로 추출을 위해 등록된 메타 데이터 값 입니다 .
<code>@헤더 맵<문자열, 개체></code>	MetadataExtractor에 설명된 대로 추출을 위해 등록된 모든 메타 데이터 값 입니다 .

반환 값은 응답 페이로드로 직렬화될 하나 이상의 객체여야 합니다. `Mono` 또는 `Flux` 와 같은 비동기 유형, 구체적인 값 또는 `void` 또는 `Mono<Void>` 와 같은 값이 없는 비동기 유형이 될 수 있습니다.

`@MessageMapping` 메서드가 지원 하는 RSocket 상호 작용 유형 은

입력(예: `@Payload` 인수) 및 출력의 카디널리티, 여기서 카디널리티는 다음을 의미합니다.

수행원:

카디널스 타이	설명
1	명시적 값 또는 <code>Mono<T></code> 와 같은 단일 값 비동기 형식입니다.
많은 <code>Flux<T></code>	와 같은 다중 값 비동기 형식입니다.
0	입력의 경우 이는 메소드에 <code>@Payload</code> 인수가 없음을 의미합니다.
출력의 경우 이것은 <code>void</code> 또는 <code>Mono<Void></code> 와 같은 값이 없는 비동기 유형입니다.	

아래 표는 모든 입력 및 출력 카디널리티 조합과 해당하는

상호작용 유형:

입력 카디널리티	산출 카디널리티	상호작용 유형
0, 1	0	화재 후 망각, 요청-응답
0, 1	1	요청-응답
0, 1	많은	요청 스트림
많은	0, 1, 많은	요청 채널

5.3.4. @ConnectMapping

`@ConnectMapping` 은 RSocket 연결이 시작될 때 `SETUP` 프레임을 처리하고 이후의 모든 `METADATA_PUSH` 프레임을 통한 `메타` 데이터 푸시 알림, 즉 `io.rsocket.RSocket`.

`@ConnectMapping` 메소드는 `@MessageMapping` 과 동일한 인수를 지원 하지만 다음을 기반으로 합니다. `SETUP` 및 `METADATA_PUSH` 프레임의 메타데이터 및 데이터. `@ConnectMapping` 은 다음과 같은 패턴을 가질 수 있습니다. 메타데이터에 경로가 있거나 패턴이 없는 경우 특정 연결에 대한 좁은 처리 선언되면 모든 연결이 일치합니다.

`@ConnectMapping` 메서드는 데이터를 반환할 수 없으며 `void` 또는 `Mono<Void>` 를 사용하여 선언해야 합니다. 반환 값. 처리 시 새 연결에 대한 오류가 반환되면 연결이 거부됩니다. 연결을 위해 `RSocketRequester` 에 요청하기 위해 처리를 보류 해서는 안 됩니다. 보다 자세한 내용은 [서버 요청자](#).

5.4. 메타데이터 추출기

응답자는 메타데이터를 해석해야 합니다. [복합 메타데이터](#) 독립적으로 포맷 가능 각각 고유한 MIME 유형이 있는 메타데이터 값(예: 라우팅, 보안, 추적용). 응용 프로그램 필요 지원하기 위해 메타데이터 MIME 유형을 구성하는 방법 및 추출된 값에 액세스하는 방법.

`MetadataExtractor` 는 직렬화된 메타데이터를 가져오고 디코딩된 이름-값 쌍을 반환하는 계약입니다. 그런 다음 주석 처리기의 `@Header` 를 통해 이름으로 헤더처럼 액세스할 수 있습니다.

행동 양식.

`DefaultMetadataExtractor` 는 메타데이터를 디코딩하기 위해 `Decoder` 인스턴스를 제공할 수 있습니다 . 기본적으로 "message/x.rsocket.routing.v0" 에 대한 지원이 내장되어 있습니다 . `문자열` 로 디코딩 하고 "경로" 키 아래에 저장합니다. 다른 모든 MIME 유형의 경우 `디코더` 를 제공하고 다음과 같이 MIME 유형을 등록해야 합니다.

자바

```
DefaultMetadataExtractor 추출기 = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(fooMimeType, Foo.class, "foo");
```

코틀린

```
org.springframework.messaging.rsocket.metadataToExtract 가져오기

val 추출기 = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Foo>(fooMimeType, "foo")
```

복합 메타데이터는 독립적인 메타데이터 값을 결합하는 데 적합합니다. 그러나 요청자가 복합 메타데이터를 지원하지 않거나 사용하지 않도록 선택 할 수 있습니다. 이를 위해 `DefaultMetadataExtractor` 는 디코딩된 값을 출력 맵에 매핑하기 위한 사용자 정의 로직이 필요할 수 있습니다.

다음은 메타데이터에 JSON을 사용하는 예입니다.

자바

```
DefaultMetadataExtractor 추출기 = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract( MimeType.valueOf("application/vnd.myapp.metadata+json"), new
ParameterizedTypeReference<Map<String, String>>() {}, (jsonMap, outputMap) ->
{ outputMap.putAll(jsonMap) ;

});
```

코틀린

```
org.springframework.messaging.rsocket.metadataToExtract 가져오기

val 추출기 = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Map<String, String>>(MimeType.valueOf("application/vnd.myapp.metadata+json")) { jsonMap, outputMap
->
outputMap.putAll(jsonMap)
```

`RSocketStrategies` 를 통해 `MetadataExtractor` 를 구성할 때 `RSocketStrategies.Builder` 가 구성된 디코더로 추출기를 생성하도록 할 수 있으며 콜백을 사용하여 다음과 같이 등록을 사용자 정의할 수 있습니다 .

자바

```
RSocketStrategies 전략 = RSocketStrategies.builder()
    .metadataExtractorRegistry(레지스트리 -> {
        Registry.metadataToExtract(fooMimeType, Foo.class, "foo");
        // ...
    })
    .짓다();
```

코틀린

org.springframework.messaging.rsocket.metadataToExtract 가져오기

```
val 전략 = RSocketStrategies.builder()
    .metadataExtractorRegistry { 레지스트리: MetadataExtractorRegistry ->
        Registry.metadataToExtract<Foo>(fooMimeType, "foo")
        // ...
    }
    .짓다()
```

6장. 리액티브 라이브러리

[spring-webflux](#)는 [Reactor-core](#)에 의존하며 비동기 로직을 구성하고 Reactive Streams 지원을 제공하기 위해 내부적으로 사용합니다. 일반적으로 WebFlux API는 [Flux](#) 또는 [Mono](#) (내부적으로 사용되기 때문에)를 반환하고 모든 Reactive Streams [Publisher](#) 구현을 입력으로 관대하게 수락합니다. [Flux](#)와 [Mono](#)의 사용은 카디널리티를 표현하는 데 도움이 되기 때문에 중요합니다. 예를 들어 단일 또는 다중 비동기 값이 예상되는지 여부와 결정을 내리는 데 필수적일 수 있습니다(예: HTTP 메시지 인코딩 또는 디코딩).

주석이 달린 컨트롤러의 경우 WebFlux는 애플리케이션에서 선택한 반응 라이브러리에 투명하게 적응합니다. 이것은 [ReactiveAdapterRegistry](#)의 도움으로 수행됩니다. 반응 라이브러리 및 기타 비동기 유형에 대한 플러그형 지원을 제공합니다. 레지스트리에는 RxJava 3, Kotlin 코루틴 및 SmallRye Mutiny에 대한 지원이 내장되어 있지만 다른 타사 어댑터도 등록할 수 있습니다.

- ☒ Spring Framework 5.3.11부터 RxJava 1 및 2에 대한 지원은 다음과 같이 더 이상 사용되지 않습니다. RxJava의 자체 EOL 조언 및 RxJava 3에 대한 업그레이드 권장 사항.

기능적 API(예: [Functional Endpoints](#), [WebClient](#) 및 기타)의 경우 WebFlux API에 대한 일반 규칙이 적용됩니다. [Flux](#) 및 [Mono](#)를 반환 값으로, Reactive Streams [Publisher](#)를 입력으로 사용합니다. [게시자](#)가 사용자 지정 또는 다른 반응 라이브러리에서 제공되는 경우 알 수 없는 의미 체계(0..N)가 있는 스트림으로만 처리될 수 있습니다. 그러나 의미 체계가 알려진 경우 원시 Publisher를 전달하는 대신 [Flux](#) 또는 [Mono.from\(Publisher\)](#)으로 래핑할 수 있습니다.

예를 들어 [Mono](#)가 아닌 [게시자](#)가 있는 경우 Jackson JSON 메시지 작성자는 여러 값을 예상합니다. 미디어 유형이 무한 스트림(예: [application/json+stream](#))을 의미하는 경우 값이 개별적으로 기록되고 플러시됩니다. 그렇지 않으면 값이 목록으로 버퍼링되고 JSON 배열로 렌더링됩니다.