Spring WebFlux

이일민 Toby

소프트웨어 개발업자
Epril
Wirebarley
KSUG

토비의 스프링

toby.epril.com
fb.com/tobyilee
youtube.com/tobyleetv

발표내용

: Spring 5.0 WebFlux 소개와 개발 방법

: M5+

: ~~리액티브 프로그래밍이란 무엇인가?~~

: ~~클라우드에서 리액티브 시스템 구축~~

: ~~Reactor, RxJava 사용법~~

대상

: **중급**

: 스프링 @MVC와 리액티브 프로그래밍의 기본 지식과 적용 기술(RxJava, Reactor), 배경을 알고 있는 또는 사용해본 개발자

http://www.springcamp.io/2017/

: 자바8+ 람다식과 함수형 스타일 개발에 대한 지식이나 경험을 가진 개발자

Spring 5.0의 새로운 모듈

Srpring-WebFlux 개요

Spring 5.0

- 2016년 M1 공개
- 현재 M5->RC1
- 2017년 6월 출시 예정
- 자바8+

Spring-WebFlux

- 구 Spring-Web-Reactive
- 스프링 5의 메인 테마는 원래 JDK9이었는데 이제는 WebFlux으로 바뀌었다 – Jürgen Höller
- 스프링 리액티브 스택의 웹 파트 담당

용도

- 비동기-논블록킹 리액티브 개발에 사용
- 효율적으로 동작하는 고성능 웹 애플리케이션 개발
- 서비스간 호출이 많은 마이크로서비스 아키텍처에 적합
- ~~새로운 방식으로 개발해보고 싶은 호기심 충족~~

2가지 개발방식 지원

- 기존의 @MVC 방식
- 새로운 함수형 모델

2가지 개발방식 지원

- 기준의 @MVC 방식
- 새로운 함수형 모델

  - @Controller, @RestController
  - @RequestMapping

2가지 개발방식 지원

- 기준의 @MVC 방식
- 새로운 함수형 모델

  - RouterFunction
  - HandlerFunction

새로운 요청-응답 모델

- 서블릿 스택과 API에서 탈피
  - 서블릿 API는 리액티브 함수형 스타일에 적합하지 않음
- ~~HttpServletRequest, HttpServletResponse~~
- ServerRequest, ServerResponse

지원 웹 서버/컨테이너

- Servlet 3.1+ (Tomcat, Jetty, ...)
- Netty
- Undertow

지원 웹 서버/컨테이너

- Servlet 3.1+ (Tomcat, Jetty, ...)
- Netty
- Undertow

서블릿 3.1+의 비동기-논블록킹 요청 처리 기능

지원 웹 서버/컨테이너

- Servlet 3.1+ (Tomcat, Jetty, ...)
- Netty
- Undertow

비동기-논블록킹IO 웹 서버

RouterFunction + HandlerFunction

함수형 스타일 WebFlux

스프링이 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

스프링이 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

- 웹 요청을 어느 핸들러에게 보낼지 결정
- URL, 헤더
- @RequestMapping

스프링이 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

- 핸들러에 전달할 웹 요청 준비
- 웹 URL, 헤더, 쿠기, 바디

스프링이 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

- 전달 받은 요청 정보를 이용해 로직을 수행하고 결과를 리턴

스프링이 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

  - 핸들러의 리턴 값으로 웹 응답 생성

```
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

```
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

```java
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

요청 매핑

```java
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

요청 바인딩

```java
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

핸들러 실행

```java
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

핸들러 결과 처리(응답 생성)

RouterFunction

- 함수형 스타일의 요청 매핑

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

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

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

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

WebFlux 버전의 웹 요청

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

웹 플럭스 버전의 웹 응답인 ServerResonse이나 그 서브타입의 Mono 퍼블리셔를 리턴하는 HandlerFunction 의 Mono 타입...

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


ServerResonse를 리턴하는 HandlerFunction

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

웹 응답을 리턴하는 함수

HandlerFunction

- 함수형 스타일의 웹 핸들러(컨트롤러 메소드)
- 웹 요청을 받아 웹 응답을 돌려주는 함수

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
  Mono<T> handle(ServerRequest request);
}
```

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
   Mono<T> handle(ServerRequest request);
}
```

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
  Mono<T> handle(ServerRequest request);
}
```

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
  Mono<T> handle(ServerRequest request);
}
```

함수형 WebFlux가 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

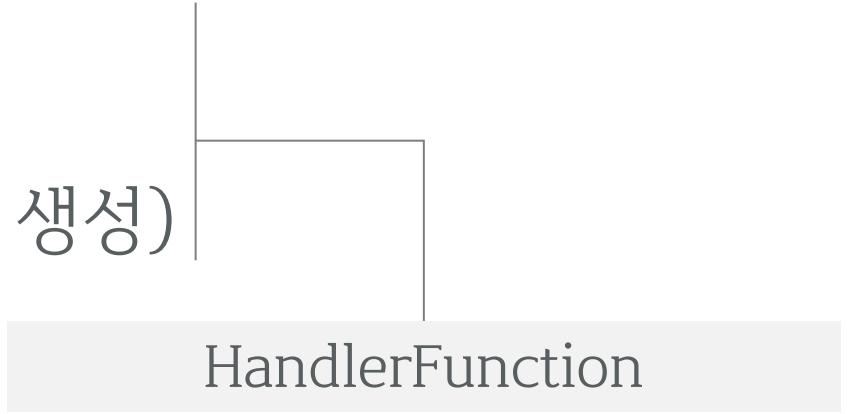
함수형 WebFlux가 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)

RouterFunction

함수형 WebFlux가 웹 요청을 처리하는 방식

- 요청 매핑
- 요청 바인딩
- 핸들러 실행
- 핸들러 결과 처리(응답 생성)



WebFlux 함수형 Hello/{name} 작성

- 함수 2개 만들면 됨
  - HandlerFunction을 먼저 만들고
  - RouterFunction에서 path기준 매핑을 해준다

```java
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

HandlerFunction으로

```
HandlerFunction helloHandler = req -> {
   String name = req.pathVariable("name");
   Mono<String> result = Mono.just("Hello " + name);
   Mono<ServerResponse> res = ServerResponse.ok().body(result,
String.class);
   return res;
};
```

```
HandlerFunction helloHandler = req -> {
   String name = req.pathVariable("name");
   Mono<String> result = Mono.just("Hello " + name);
   Mono<ServerResponse> res = ServerResponse.ok().body(result,
String.class);
   return res;
};
```

HandlerFunction.handle()의 람다식

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

```
HandlerFunction helloHandler = req -> {
  String name = req.pathVariable("name");
  Mono<String> result = Mono.just("Hello " + name);
  Mono<ServerResponse> res = ServerResponse.ok().body(result,
String.class);
  return res;
};
```

ServerRequest.pathVariable()로 {name} 추출

```
HandlerFunction helloHandler = req -> {
  String name = req.pathVariable("name");
  Mono<String> result = Mono.just("Hello " + name);
  Mono<ServerResponse> res = ServerResponse.ok().body(result,
String.class);
  return res;
};
```

로직 적용후 결과 값을 Mono에 담는다

```
HandlerFunction helloHandler = req -> {
  String name = req.pathVariable("name");
  Mono<String> result = Mono.just("Hello " + name);
  Mono<ServerResponse> res = ServerResponse.ok().body(result,
String.class);
  return res;
};
```

- 웹 응답을 ServerResponse로 만든다
- HTTP 응답에는 응답코드, 헤더, 바디가 필요
- ServerResponse의 빌더를 활용
- Mono에 담긴 ServerResponse 타입으로 리턴

```
HandlerFunction helloHandler = req -> {
   String name = req.pathVariable("name");
   Mono<String> result = Mono.just("Hello " + name);
   Mono<ServerResponse> res = ServerResponse.ok().body(result,
String.class);
   return res;
};
```

함수형 스타일의 코드는 간결하게 작성할 수 있다

```
HandlerFunction helloHandler = req ->
        ok().body(fromObject("Hello " + req.pathVariable("name")));
```

```
HandlerFunction helloHandler = req ->
        ok().body(fromObject("Hello " + req.pathVariable("name")));
```

ServerRequest에서 로직에 필요한 데이터 추출 & 바인딩

```
HandlerFunction helloHandler = req ->
        ok().body(fromObject("Hello " + req.pathVariable("name")));
```

- 애플리케이션 로직을 적용하고 결과 값을 만든다
- BodyInserters.fromObject()의 도움을 받아 Mono에 담는다

```
HandlerFunction helloHandler = req ->
        ok().body(fromObject("Hello " + req.pathVariable("name")));
```

- 로직의 결과 값을 바디에 담고 상태코드를 추가해 웹 응답(ServerResponse)으로 만든다
- content-type이나 기타 헤더들은 스프링이 알아서 만들어준다

```
HandlerFunction helloHandler = req ->
        ok().body(fromObject("Hello " + req.pathVariable("name")));
```

함수(람다 식) 형식으로 만들어 둔다

```java
@RestController
public class MyController {
  @GetMapping("/hello/{name}")
  String hello(@PathVariable String name) {
    return "Hello " + name;
  }
}
```

RouterFunction으로

```
RouterFunction router = req ->
    RequestPredicates.path("/hello/{name}").test(req) ?
        Mono.just(helloHandler) : Mono.empty();
```

```
RouterFunction router = req ->
    RequestPredicates.path("/hello/{name}").test(req) ?
        Mono.just(helloHandler) : Mono.empty();
```

웹 요청 정보 중에서 URL 경로 패턴 검사

```
RouterFunction router = req ->
    RequestPredicates.path("/hello/{name}").test(req) ?
        Mono.just(helloHandler) : Mono.empty();
```

조건에 맞으면 핸들러 함수를 Mono에 담아서 반환

```
RouterFunction router = req ->
    RequestPredicates.path("/hello/{name}").test(req) ?
        Mono.just(helloHandler) : Mono.empty();
```

- 조건에 맞지 않으면 빈 Mono 반환
- 함수니까 뭐라도 반환해야 함

```
RouterFunction router = req ->
    RequestPredicates.path("/hello/{name}").test(req) ?
        Mono.just(helloHandler) : Mono.empty();
```

RouterFunction.route()의 람다식

```
Mono<HandlerFunction<T>> route(ServerRequest request);
```

HandlerFunction과 RouterFunction 조합

- RouterFunctions.route(predicate, handler)

```
RouterFunction router =
   RouterFunctions.route(RequestPredicates.path("/hello/{name}"),
        req -> ServerResponse.ok().body(fromObject("Hello " +
                req.pathVariable("name"))));
```

```
RouterFunction router =
  RouterFunctions.route(RequestPredicates.path("/hello/{name}"),
    req -> ServerResponse.ok().body(fromObject("Hello " +
      req.pathVariable("name"))));
```



```
RouterFunction router = req ->
  RequestPredicates.path("/hello/{name}").test(req) ?
    Mono.just(helloHandler) : Mono.empty();
```

RouterFunction의 매핑 조건을 체크하는 로직만 발췌

```
RouterFunction router =
  RouterFunctions.route(RequestPredicates.path("/hello/{name}"),
      req -> ServerResponse.ok().body(fromObject("Hello " +
              req.pathVariable("name"))));
```



```
HandlerFunction helloHandler = req ->
        ok().body(fromObject("Hello " + req.pathVariable("name")));
```

HandlerFunction은 그대로

RouterFunction의 등록

- RouterFunction 타입의 @Bean으로 만든다

```
@Bean
RouterFunction helloPathVarRouter() {
   return route(RequestPredicates.path("/hello/{name}"),
         req -> ok().body(fromObject("Hello " +
                                    req.pathVariable("name"))));
}
```

핸들러 내부 로직이 복잡하다면 분리한다

- 핸들러 코드만 람다 식을 따로 선언하거나
- 메소드를 정의하고 메소드 참조로 가져온다

```
HandlerFunction handler = req -> {
   String res = myService.hello(req.pathVariable("name"));
   return ok().body(fromObject(res));
};

return route(path("/hello/{name}"), handler);
```

```
HandlerFunction handler = req -> {
  String res = myService.hello(req.pathVariable("name"));
  return ok().body(fromObject(res));
};

return route(path("/hello/{name}"), handler);
```

다른 bean 호출을 포함한 복잡한 로직을 담은 람다 식

```
@Component
public class HelloHandler {
  @Autowired MyService myService;

  Mono<ServerResponse> hello(ServerRequest req) {
    String res = myService.hello(req.pathVariable("name"));
    return ok().body(fromObject(res));
  }
  …
}
```

```
@Component
public class HelloHandler {
  @Autowired MyService myService;

  Mono<ServerResponse> hello(ServerRequest req) {
    String res = myService.hello(req.pathVariable("name"));
    return ok().body(fromObject(res));
  }
  …
}
```

앞의 람다식과 동일한 메소드 타입을 가진 메소드

```
@Bean
RouterFunction helloRouter(@Autowired HelloHandler helloHandler) {
  return route(path("/hello/{name}"), helloHandler::hello);
}
```

```
@Bean
RouterFunction helloRouter(@Autowired HelloHandler helloHandler) {
  return route(path("/hello/{name}"), helloHandler::hello);
}
```

핸들러 메소드가 정의된 빈을 주입

```
@Bean
RouterFunction helloRouter(@Autowired HelloHandler helloHandler) {
  return route(path("/hello/{name}"), helloHandler::hello);
}
```

빈의 핸들러 메소드 레퍼런스

RouterFunction의 조합

- 핸들러 하나에 @Bean하나씩 만들어야 하나?
- RouterFunction의 and(), andRoute() 등으로 하나의 @Bean에 n개의 RouterFunction을 선언할 수 있다

RouterFunction의 매핑 조건의 중복

- 타입 레벨 – 메소드 레벨의 @RequestMapping처럼 공통의 조건을 정의하는 것 가능
- RouterFunction.nest()

```
public RouterFunction<?> routingFunction() {
  return nest(pathPrefix("/person"),
    nest(accept(APPLICATION_JSON),
        route(GET("/{id}"), handler::getPerson)
        .andRoute(method(HttpMethod.GET), handler::listPeople)
    ).andRoute(POST("/").and(contentType(APPLICATION_JSON)),
handler::createPerson));
}
```

```
public RouterFunction<?> routingFunction() {
  return nest(pathPrefix("/person"),
    nest(accept(APPLICATION_JSON),
        route(GET("/{id}"), handler::getPerson)
        .andRoute(method(HttpMethod.GET), handler::listPeople)
    ).andRoute(POST("/").and(contentType(APPLICATION_JSON)),
handler::createPerson));
}
```

URL이 /preson으로 시작하는 조건을 공통으로

```
public RouterFunction<?> routingFunction() {
  return nest(pathPrefix("/person"),
    nest(accept(APPLICATION_JSON),
        route(GET("/{id}"), handler::getPerson)
        .andRoute(method(HttpMethod.GET), handler::listPeople)
    ).andRoute(POST("/").and(contentType(APPLICATION_JSON)),
handler::createPerson));
}
```

APPLICATION_JSON을 accept하는 공통조건 중첩

```
public RouterFunction<?> routingFunction() {
  return nest(pathPrefix("/person"),
    nest(accept(APPLICATION_JSON),
      route(GET("/{id}"), handler::getPerson)
      .andRoute(method(HttpMethod.GET), handler::listPeople)
    ).andRoute(POST("/").and(contentType(APPLICATION_JSON)),
handler::createPerson));
}
```

/person/{id} 경로의 GET이면 getPerson 핸들러로 매핑

```
public RouterFunction<?> routingFunction() {
  return nest(pathPrefix("/person"),
    nest(accept(APPLICATION_JSON),
      route(GET("/{id}"), handler::getPerson)
      .andRoute(method(HttpMethod.GET), handler::listPeople)
    ).andRoute(POST("/").and(contentType(APPLICATION_JSON)),
handler::createPerson));
}
```

/person 경로에 GET이면 listPeople 핸들러로

```
public RouterFunction<?> routingFunction() {
  return nest(pathPrefix("/person"),
    nest(accept(APPLICATION_JSON),
      route(GET("/{id}"), handler::getPerson)
      .andRoute(method(HttpMethod.GET), handler::listPeople)
    ).andRoute(POST("/").and(contentType(APPLICATION_JSON)),
handler::createPerson));
}
```

/person 경로에 POST이고 contentType이 APPLICATION_JSON이면 createPerosn 핸들러로

WebFlux 함수형 스타일의 장점

- 모든 웹 요청 처리 작업을 명시적인 코드로 작성
  - 메소드 시그니처 관례와 타입 체크가 불가능한 애노테이션에 의존하는 @MVC 스타일보다 명확
  - 정확한 타입 체크 가능
- 함수 조합을 통한 편리한 구성, 추상화에 유리
- 테스트 작성의 편리함
  - 핸들러 로직은 물론이고 요청 매핑과 린턴 값 처리까지 단위테스트로 작성 가능

WebFlux 함수형 스타일의 단점

- 함수형 스타일의 코드 작성이 편하지 않으면 코드 작성과 이해 모두 어려움
- 익숙한 방식으로도 가능한데 뭐하러

@Controller + @RequestMapping

@MVC WebFlux

@MVC WebFlux

- 애노테이션과 메소드 형식의 관례를 이용하는 @MVC 방식과 유사
- 비동기 + 논블록킹 리액티브 스타일로 작성

ServerRequest, ServerResponse

- WebFlux의 기본 요청, 응답 인터페이스 사용
- 함수형 WebFlux의 HandlerFunction을 메소드로 만들었을 때와 유사
- 매핑만 애노테이션 방식을 이용

```
@RestController
public static class MyController {
  @RequestMapping("/hello/{name}")
  Mono<ServerResponse> hello(ServerRequest req) {
    return ok().body(fromObject(req.pathVariable("name")));
  }
```

```
@RestController
public static class MyController {
  @RequestMapping("/hello/{name}")
  Mono<ServerResponse> hello(ServerRequest req) {
    return ok().body(fromObject(req.pathVariable("name")));
  }
```

메소드로 재정의된 HandlerFunction

```java
@RestController
public static class MyController {
  @RequestMapping("/hello/{name}")
  Mono<ServerResponse> hello(ServerRequest req) {
    return ok().body(fromObject(req.pathVariable("name")));
  }
```

요청 매핑과 등록은 기존 @MVC 방식으로

@MVC 요청 바인딩과 Mono/Flux 리턴 값

- 가장 대표적인 @MVC WebFlux 작성 방식
- 파라미터 바인딩은 @MVC 방식 그대로
- 핸들러 로직 코드의 결과를 Mono/Flux 타입으로 리턴

@MVC와 동일한 바인딩

- 경로 변수
- 커맨드 오브젝트
- 폼 오브젝트, 모델 애트리뷰트

```
@GetMapping("/hello/{name}")
Mono<String> hello(@PathVariable String name) {
  return Mono.just("Hello " + name);
}
```

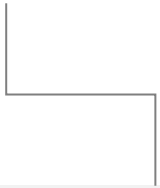
```
@GetMapping("/hello/{name}")
Mono<String> hello(@PathVariable String name) {
  return Mono.just("Hello " + name);
}
```

@MVC 스타일 매핑

```
@GetMapping("/hello/{name}")
Mono<String> hello(@PathVariable String name) {
  return Mono.just("Hello " + name);
}
```

@MVC에서 사용하던 요청 바인딩 그대로

```
@GetMapping("/hello/{name}")
Mono<String> hello(@PathVariable String name) {
  return Mono.just("Hello " + name);
}
```



리턴 값은 Mono/Flux로

```
@RequestMapping("/hello")
Mono<String> hello(User user) {
  return Mono.just("Hello " + user.getName());
}
```

```
@RequestMapping("/hello")
Mono<String> hello(User user) {
  return Mono.just("Hello " + user.getName());
}
```

- 커맨드 오브젝트, 모델 오브젝트 바인딩
- URL 파라미터 또는 form-data

@RequestBody 바인딩 (JSON, XML)

- T
- Mono<T>
- Flux<T>

```
@RequestMapping("/hello")
Mono<String> hello(@RequestBody User user) {
  return Mono.just("Hello " + user.getName());
}
```

```
@RequestMapping("/hello")
Mono<String> hello(@RequestBody User user) {
  return Mono.just("Hello " + user.getName());
}
```

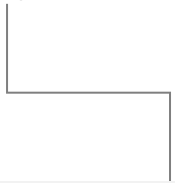
- 웹 요청의 body를 MessageConverter에서 바인딩
- @MVC와 동일

```
@RequestMapping("/hello")
Mono<String> hello(@RequestBody Mono<User> user) {
  return user.map(u -> "Hello " + u.getName());
}
```

```
@RequestMapping("/hello")
Mono<String> hello(@RequestBody Mono<User> user) {
  return user.map(u -> "Hello " + u.getName());
}
```

변환된 오브젝트를 Mono에 담아서 전달

```
@RequestMapping("/hello")
Mono<String> hello(@RequestBody Mono<User> user) {
  return user.map(u -> "Hello " + u.getName());
}
```



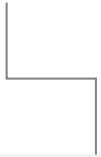
Mono의 연산자를 사용해서 로직을 수행하고 Mono로 리턴

```
@PostMapping(value = "/hello")
Flux<String> hello(@RequestBody Flux<User> users) {
  return users.map(u -> "Hello " + u.getName());
}
```

```
@PostMapping(value = "/hello")
Flux<String> hello(@RequestBody Flux<User> users) {
  return users.map(u -> "Hello " + u.getName());
}
```

User의 스트림 형태로 요청을 전달

```
@PostMapping(value = "/hello")
Flux<String> hello(@RequestBody Flux<User> users) {
  return users.map(u -> "Hello " + u.getName());
}
```



User의 스트림 형태로 로직 수행

@ResponseBody 리턴 값 타입

- T
- Mono<T>
- Flux<T>
- Flux<ServerSideEvent>
- void
- Mono<Void>

WebClient + Reactive Data

WebFlux와 리액티브 기술

WebFlux만으로 성능이 좋아질까?

- 비동기-논블록킹 구조의 장점은 블록킹 IO를 제거하는 데서 나온다
- HTTP 서버에서 논블록킹 IO는 오래 전부터 지원
- 뭘 개선해야 하나?

개선할 블록킹 IO

- 데이터 액세스 리포지토리
- HTTP API 호출
- 기타 네트워크를 이용하는 서비스

JPA – JDBC 기반 RDB 연결

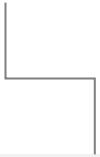
- 현재는 답이 없다
- 블로킹 메소드로 점철된 JDBC API
- 일부 DB는 논블록킹 드라이버가 존재하지만...
- @Async 비동기 호출과 CFuture를 리액티브로 연결하고 쓰레드풀 관리를 통해서 웹 연결 자원을 효율적으로 사용하도록 만드는 정도
- JDK 10에서 Async JDBC가 등장할 수도

Spring Data JPA의 비동기 쿼리 결과 방식

- 리포지토리 메소드의 리턴 값을 @Async 메소드처럼 작성

```
@Async
CompletableFuture<User> findOneByFirstname(String firstname);
```

```
@Async
CompletableFuture<User> findOneByFirstname(String firstname);
```



- 리포지토리 메소드를 비동기로 실행하고 결과를 CompletableFuture로 돌려 받는다
- @Async이므로 비동기 실행과 동시에 메소드 리턴

```
@GetMapping
Mono<User> findUser(String name) {
  return
Mono.fromCompletionStage(myRepository.findOneByFirstName(name));
}
```

```
@GetMapping
Mono<User> findUser(String name) {
  return
Mono.fromCompletionStage(myRepository.findOneByFirstName(name));
}
```

CompletableFuture<User> 리턴

```
@GetMapping
Mono<User> findUser(String name) {
  return
Mono.fromCompletionStage(myRepository.findOneByFirstName(name));
}
```

- CompletableFuture의 비동기 결과는 Mono로 변환
- 리포지토리에서 Stream<T>로 리턴한다면 Flux로도 변환 가능하나 블록킹된다

본격 리액티브 데이터 액세스 기술

- 스프링 데이터의 리액티브 리포지토리 이용
  - MongoDB
  - Cassandra
  - Redis
- ReactiveCrudRepository 확장

```
public interface ReactivePersonRepository extends
                                        ReactiveCrudRepository<Person, String> {
  Flux<Person> findByLastname(Mono<String> lastname);
  @Query("{ 'firstname': ?0, 'lastname': ?1}")
  Mono<Person>  findByFirstnameAndLastname(String firstname,
String lastname); }
```

```
public interface ReactivePersonRepository extends
                                    ReactiveCrudRepository<Person, String> {
  Flux<Person> findByLastname(Mono<String> lastname);
  @Query("{ 'firstname': ?0, 'lastname': ?1}")
  Mono<Person>  findByFirstnameAndLastname(String firstname,
String lastname); }
```

리액티브 방식의 CRUD 메소드 지원

```
public interface ReactivePersonRepository extends
                                    ReactiveCrudRepository<Person, String> {
  Flux<Person> findByLastname(Mono<String> lastname);
  @Query("{ 'firstname': ?0, 'lastname': ?1}")
  Mono<Person>  findByFirstnameAndLastname(String firstname,
String lastname); }
```

0~n개의 결과를 비동기-논블록킹 리액티브 방식으로
조회하도록 Flux<T>로 리턴

```
public interface ReactivePersonRepository extends
                                    ReactiveCrudRepository<Person, String> {
  Flux<Person> findByLastname(Mono<String> lastname);
  @Query("{ 'firstname': ?0, 'lastname': ?1}")
  Mono<Person>  findByFirstnameAndLastname(String firstname,
String lastname); }
```

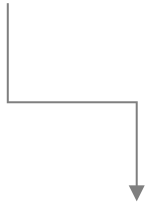
0 또는 1개의 결과를 비동기-논블록킹 리액티브로 조회하도록 결과를 Mono<T>로 리턴

```
@Autowired UserRepository userRepository;

@GetMapping
Flux<User> users() {
  return userRepository.findAll();
}
```

```
@Autowired UserRepository userRepository;

@GetMapping
Flux<User> users() {
  return userRepository.findAll();
}
```



```
Flux<T> findAll();
```

DB부터 웹까지 논블록킹 방식으로 동작

```
@GetMapping
Flux<String> users() {
  return userRepository.findAll().map(u -> "Hello " + u.getName());
}
```

리포지토리 결과 Flux에 대해 추가 로직 적용

```
public Mono<ServerResponse> getPerson(ServerRequest request) {
  int personId = Integer.valueOf(request.pathVariable("id"));
  Mono<ServerResponse> notFound = ServerResponse.notFound().build();
  Mono<Person> personMono = this.repository.getPerson(personId);
  return personMono.flatMap(person ->
ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(
person))).switchIfEmpty(notFound);
}
```

```
public Mono<ServerResponse> getPerson(ServerRequest request) {
  int personId = Integer.valueOf(request.pathVariable("id"));
  Mono<ServerResponse> notFound = ServerResponse.notFound().build();
  Mono<Person> personMono = this.repository.getPerson(personId);
  return personMono.flatMap(person ->
ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(
person))).switchIfEmpty(notFound);
}
```

```
public Mono<ServerResponse> getPerson(ServerRequest request) {
  int personId = Integer.valueOf(request.pathVariable("id"));
  Mono<ServerResponse> notFound = ServerResponse.notFound().build();
  Mono<Person> personMono = this.repository.getPerson(personId);
  return personMono.flatMap(person ->
ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(
person))).switchIfEmpty(notFound);
}
```

```
public Mono<ServerResponse> getPerson(ServerRequest request) {
  int personId = Integer.valueOf(request.pathVariable("id"));
  Mono<ServerResponse> notFound = ServerResponse.notFound().build();
  Mono<Person> personMono = this.repository.getPerson(personId);
  return personMono.flatMap(person ->
ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(
person))).switchIfEmpty(notFound);
}
```

```
public Mono<ServerResponse> getPerson(ServerRequest request) {
  int personId = Integer.valueOf(request.pathVariable("id"));
  Mono<ServerResponse> notFound = ServerResponse.notFound().build();
  Mono<Person> personMono = this.repository.getPerson(personId);
  return personMono.flatMap(person ->
ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(
person))).switchIfEmpty(notFound);
}
```

```
public Mono<ServerResponse> getPerson(ServerRequest request) {
  int personId = Integer.valueOf(request.pathVariable("id"));
  Mono<ServerResponse> notFound = ServerResponse.notFound().build();
  Mono<Person> personMono = this.repository.getPerson(personId);
  return personMono.flatMap(person ->
ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(
person))).switchIfEmpty(notFound);
}
```

논블록킹 API 호출은 WebClient

- AsyncRestTemplate의 리액티브 버전
- 요청을 Mono/Flux로 전달할 수 있고
- 응답을 Mono/Flux형태로 가져온다

```
@GetMapping("/webclient")
Mono<String> webclien() {
  return WebClient.create("http://localhost:8080")
        .get()
        .uri("/hello/{name}", "Spring")
        .accept(MediaType.TEXT_PLAIN)
        .exchange()
        .flatMap(r -> r.bodyToMono(String.class))
        .map(d -> d.toUpperCase())
        .flatMap(d -> helloRepository.save(d));
}
```

함수형 스타일의 코드가 읽기 어렵다면

- 각 단계의 타입이 보이지 않기 때문이다
- 타입이 보이도록 코드를 재구성하고 익숙해지도록 연습이 필요하다

```
@GetMapping("/webclient")
Mono<String> webclient() {
  WebClient wc = WebClient.create("http://localhost:8080");
  UriSpec<RequestHeadersSpec<?>> uriSpec = wc.get();
  RequestHeadersSpec<?> headerSpec = uriSpec.uri("/hello/{name}",
                                                 "Spring");
  RequestHeadersSpec<?> headerSpec2 =
                                 headerSpec.accept(MediaType.TEXT_PLAIN);
  Mono<ClientResponse> res = headerSpec2.exchange();
  Mono<String> data = res.flatMap(r -> r.bodyToMono(String.class));
  Mono<String> upperData = data.map(d -> d.toUpperCase());
  return upperData.flatMap(d -> helloRepository.save(d));
}
```

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

기준 URL을 넣어 WebClient 생성

```
UriSpec<RequestHeadersSpec<?>> uriSpec = wc.get();
```

HTTP 메소드 결정

다음 단계로 Uri 설정 준비

```
RequestHeadersSpec<?> headerSpec =
                                    uriSpec.uri("/hello/{name}", "Spring");
```

URI 패턴과 파라미터로 URI 설정

URI 패턴과 파라미터로 URI 설정

```
RequestHeadersSpec<?> headerSpec2 =
                        headerSpec.accept(MediaType.TEXT_PLAIN);
```

헤더 설정

```
Mono<ClientResponse> res = headerSpec2.exchange();
```

요청을 응답으로 교환!

ServerResponse와 유사한 구조의 응답 정보

```
Mono<String> data = res.flatMap(r -> r.bodyToMono(String.class));
```

요청 바디를 String 타입으로 변환해서
Mono에 담아 리턴하는 함수

Mono 데이터에 적용한 함수의 결과가 Mono타입이기 때문에
flatMap을 적용해야 한다. 아니면 Mono<Mono<String>>이 됨

```
Mono<String> upperData = data.map(d -> d.toUpperCase());
```

데이터에 함수를 적용해서 변환

```
return upperData.flatMap(d -> helloRepository.save(d));
```

데이터를 리포지토리에 저장하고 결과를 리턴

```
@GetMapping("/webclient")
Mono<String> webclien() {
  return WebClient.create("http://localhost:8080")
        .get()
        .uri("/hello/{name}", "Spring")
        .accept(MediaType.TEXT_PLAIN)
        .exchange()
        .flatMap(r -> r.bodyToMono(String.class))
        .map(d -> d.toUpperCase())
        .flatMap(d -> helloRepository.save(d));
}
```

비동기-논블록킹 리액티브 웹 애플리케이션의 효과를 얻으려면

- WebFlux + 리액티브 리포지토리
  + 리액티브 원격 API 호출
  + 리액티브 지원 외부 서비스
  + @Async 블록킹 IO
- 코드에서 블록킹 작업이 발생하지 않도록 Flux 스트림 또는 Mono에 데이터를 넣어서 전달

리액티브 함수형은 꼭 성능 때문만?

- 함수형 스타일 코드를 이용해 간결하고 읽기 좋고 조합하기 편한 코드 작성
- 데이터 흐름에 다양한 오퍼레이터 적용
- 연산을 조합해서 만든 동시성 정보가 노출되지 않는 추상화된 코드 작성
  - 동기,비동기,블록킹,논블록킹 등을 유연하게 적용
- 데이터의 흐름의 속도를 제어할 수 있는 메커니즘 제공

논블록킹 IO에만 효과가 있나?

- 시스템 외부에서 발생하는 이벤트에도 유용
- 클라이언트로부터의 이벤트에도 활용 가능

ReactiveStreams

- WebFlux가 사용하는 Reactor외에 RxJava2를 비롯한 다양한 리액티브 기술에 적용된 표준 인터페이스
- 다양한 기술, 서비스 간의 상호 호환성에 유리
- 자바9에 Flow API로 포함

뭘 공부해야 하나

- 자바 8+함수형 프로그래밍에 익숙해질 것
- CompletableFuture와 같이 비동기 작업의 조합, 결합에 뛰어난 툴의 사용법을 익힐 것
- ReactorCore 학습
  - Mono/Flux, 오퍼레이터, 스케줄러
- WebFlux와 스프링의 리액티브 스택 공부
- 비동기 논블록킹 성능과 관련된 벤치마킹, 모니터링, 디버깅 연구
- 테스트

감사합니다

스프링 리액티브 라이브 코딩은 youtube.com/tobyleetv