~~Think Reactive.~~

Simply, flow of data.

content:

Reactive vs. Imperative programming

Project Reactor by Netflix

Web Flux asynchronous, non-blocking

deprecate RestTemplate, what next ? WebClient

Persisting reactively R2DBC

RSocket network protocol

required knowledge: Java core, Stream API, synchronous vs asynchronous, Spring Boot, Spring Web/MVC/Rest, Spring Data, basic CRUD operations, Unit/Integration testing

level: intermediate Java developers

duration: 30 min

REACTIVE PROGRAMMING - describe set of steps as pipeline or stream trough which data flows, reactive stream process data as it becomes available, data can be endless.
// example: national geographic subscription

Imperative - set of tasks, each running one at a time, one after another. Data is processed in bulk, and can't be handed to next task until previous has completed.
vs.
Reactive - set of tasks to process data, that can run in parallel. Each task can process subsets of data, handing it to next task, while continue work on another subset of data. Advantage in scalability and performance, than imperative
// example: water ballon/garden hose

REACTIVE STREAM - late 2013 initiative, Netflix, Light-bend and Pivotal - standard for asynchronous (perform task in parallel) stream processing with nonblocking back-pressure (consumers of data establish limits on how much data can process).

Java streams - typically synchronous, work with finite set of data, like iterating over a collection with functions.
vs.
Reactive streams - support processing of any size datasets, incl infinite. Process data in realtime, as it becomes available, with back-pressure to avoid consumer blocking.

REACTIVE STREAM SPECIFICATION SUMMED BY FOUR DEFINITIONS

`Publisher` - produces data, send to Subscriber per Subscription, provide single method through which Subscriber can subscribe.

```
public interface Publisher<T> {
    void subscribe(Subscriber<? extends T> subscriber); //start data flow
}
```

`Subscriber` - once subscribed, receives events from Publisher, via methods.

```
public interface Subscriber<T> {
    void onSubscribe(Subscription sub); // 1st event, pass Subscription
    void onNext(T item); // for every item published, to be delivered
    void onError(Throwable ex); // if there are any errors
    void onComplete(); // informs that publisher finish the data
}
```

`Subscription` - Subscriber manage with it the subscription

```
public interface Subscription {
    void request(long n); // request data, n back-pressure limit of items
    void cancel(); // cancel the subscription, stop the data flow
}
```

`Processor` - combination of Subscriber and Publisher, subscribe to data flow and then publish the results…

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

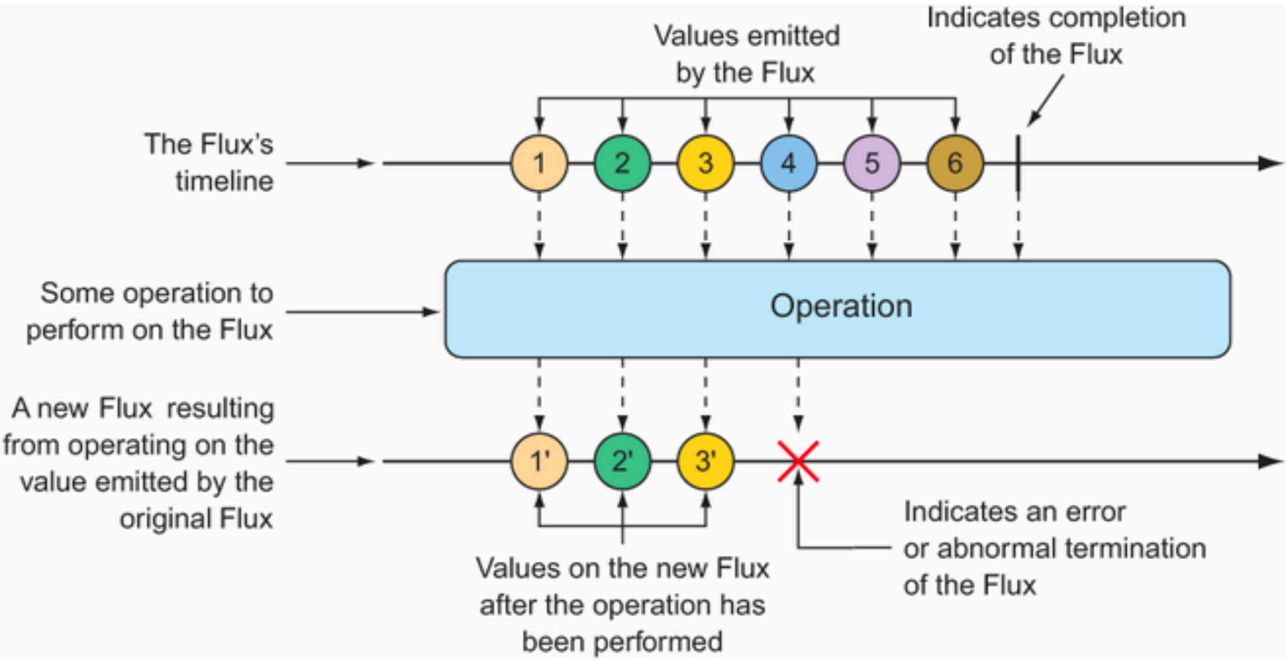
PROJECT REACTOR - implementation of Reactive Streams specification that provide functional API for composing Reactive Streams, foundation for Spring's reactive programming.

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>

<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
```

`Mono` and `Flux` are core Reactor's types, implementation of Reactive Streams `Publisher`.
`Flux` represents pipeline of zero, one or many(infinite) data items.
`Mono` optimised for datasets having zero or one items.
// example Reactor (`Mono, Flux`) vs RxJava/ReactiveX (`Single, Observable`)

```
String name = "Ivan"; // perform steps, one after another
String capitalName = name.toUpperCase();
String text = "Hello " + capitalName;
System.out.println(text); // 100% all on a single thread
vs.
Mono.just("Ivan") // publisher (Mono impl) that emits value
        .map(n -> name.toUpperCase()) // get value and publish it, Mono 2
        .map(cn -> "Hello " + cn) // get value and publish it, Mono 3
        .subscribe( //finally subscribe to Mono (publisher), receive data
                System.out::println); //maybe single thread or maybe not?
```


Values emitted by the Flux
Indicates completion of the Flux
The Flux's timeline
1
2
3
4
5
6
Some operation to perform on the Flux
Operation
A new Flux resulting from operating on the value emitted by the original Flux
1'
2'
3'
Indicates an error or abnormal termination of the Flux
Values on the new Flux after the operation has been performed

COMMON REACTIVE OPERATIONS - more than 500 operations

```
@Test
void create(){
    Flux<String> flux = Flux.just("A", "B", "C");

    StepVerifier.create(flux) // subscribe to Reactive type
            .expectNext("A") // apply assertion to each data, as it flows
            .expectNext(“B”, "C")
            .verifyComplete(); // verify the stream completed
}
```

// Creation operations - usually get a Flux/Mono from a Service or Repo…

```
.just("A", "B", “C"); // create reactive type, from single data items
.fromArray(new String[]{"A", "B", “C"}); // create from Array
.fromIterable(List.of("A", "B", “C")); // create from Iterable
.fromStream(Stream.of("A", "B", “C")); // create from Stream
.range(1, 5) // create range 1,2,3,4,5
.interval(Duration.ofSeconds(1)); // create infinite
```

// Combining and Splitting operations

```
first.mergeWith(second); // merge to single data stream
.zip(names, ages); // combine to Tuple2 pairs
.zip(names, ages, (a, b) -> a + b); // combine to Objects
```

// Filtering and Transforming operations

```
.skip(2); // skip first n elements “C”
.take(2); // take first n elements “A”, “B”
.filter(el -> el.contains(“B")); // filter based on predicate
.distinct(); // only unique values will be emitted
.map(el -> new User(el)); // transform elements, synchronously!
.flatMap(el -> // transform elements, synchronously!
    Mono.just(el).map(User::new)
            .subscribeOn(Schedulers.parallel())
); // each subscription take place in parallel thread
.buffer(2); // buffer to collections List.of(“A”, “B”)
.collectList(); // collect emitted elements to Iterable
.collectMap(el -> el.charAt(0)); // collect to Map, key from function
```

// Logic operations

```
all(el -> el.length() == 1); // all data elements, meet criteria
.any(el -> el.equalsIgnoreCase(“c")); // at least one el, meet criteria
```

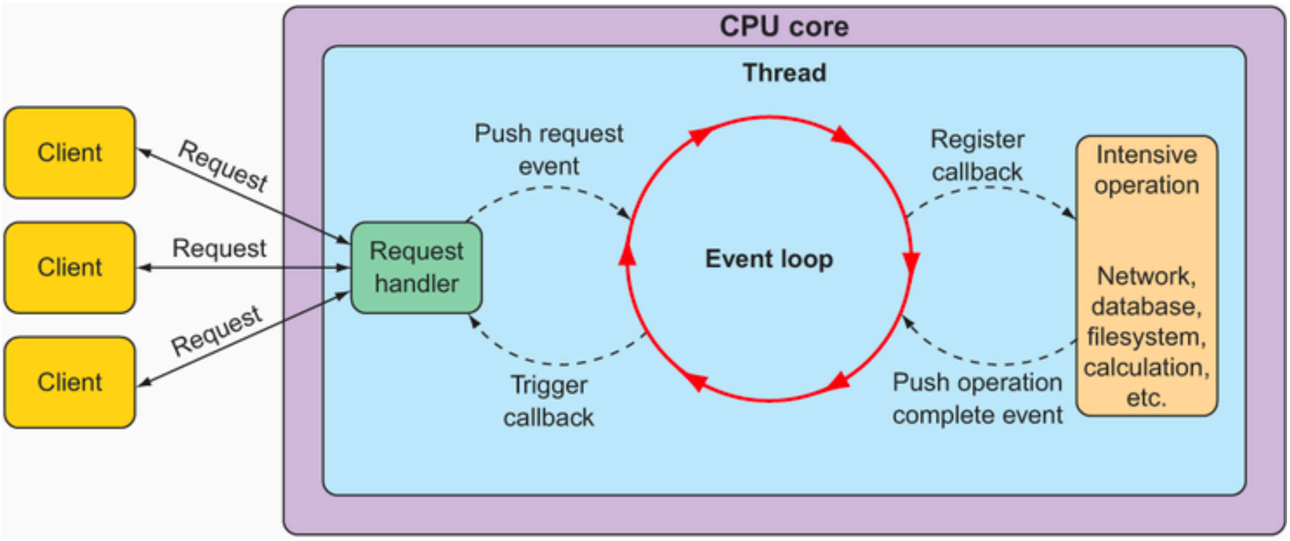
Schedulers concurrency models, for executing the subscription:

```
.immediate() // in the current thread
.single() // in a single, reusable thread for all callers
.newSingle() // in a per-call dedicated thread
.elastic() // in a worker from unbounded pool, create as needed, idle 60s
.parallel() //in worker from fix-size pool, to the number CPU cores
```

WEBFLUX - from v.5, Spring's reactive web framework, asynchronous and nonblocking, based on Project Reactor.

Servlet web frameworks - (Spring MVC) are blocking and multithreaded, using single thread per connection, as request is handled, a worker thread is pulled from the pool to process the data, while request thread is blocked until notified by worker that is finished.
// this is how most web apps are developed, things change from occasionally to frequently consuming content, IoT exchanging data with web APIs

Asynchronous web frameworks - (WebFlux) achieve higher scalability with fewer threads (one per CPU core), by Event Looping can be handled many requests per thread.
// everything is handled as event, when costly operation (db, network) is needed, event loop register a callback for it to be performed in parallel, while it moves on to handle other events. When operation is complete, it's treated as event and pass data as response.



Both framework share many common components (Annotations),
Spring MVC is based on Servlet API, which requires Servlet Container (default Tomcat) to execute on, while
WebFlux builds on top Reactive HTTP API and embedded server is Netty, which is asynchronous, non-blocking and event-driven, making it natural fit for reactive web framework.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
```

// WebFlux controller methods accept and return reactive types (Flux/Mono), instead of domain types, and also can work with ReactiveX (Single, Observable, Completable)

// Reactive Spring MVC? can also work with Mono/Flux, however it is servlet-based, relaying on multithreading to handle requests.

WRITE REACTIVE REPOS, SERVICES and CONTROLLERS. *// show/remind usual REST Controller*

```
@RequiredArgsConstructor
@RestController
@RequestMapping("/reactive")
public class ReactiveController {

    private UserRepo reactiveUserRepo;

    @GetMapping("/all")
    public Flux<User> getAllUsers() { // if RxJava can return Observable
        return reactiveUserRepo.findAll().take(5);
    }

    @GetMapping("/{id}")
    public Mono<User> getById(@PathVariable("id") Long id) {
        return reactiveUserRepo.findById(id); // if RxJava return Single
    }

    @PostMapping(consumes = "application/json")
    @ResponseStatus(HttpStatus.CREATED)
    public Mono<User> saveUser(@RequestBody Mono<User> mono) {
        return mono.flatMap(user -> reactiveUserRepo.save(user));
//        return reactiveUserRepo.saveAll(mono).next();
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> deleteById(@PathVariable("id") Long id){
        return reactiveUserRepo.deleteById(id); // RxJava - Completable
    }
}


@Repository
public interface UserRepo extends ReactiveCrudRepository<User, Long> {
// the methods from the repository return reactive types: Flux<User> ..
   @Override
   Flux<User> findAll();

   @Override
   Mono<User> findById(Long id);
}
```

SUMMARY:

- WebFLux is reactive web framework that mirror Spring MVC and share many of the same annotations.
- Spring functional programming module as alternative on the annotation-based
- Testing Reactive controllers with `WebTestClient`
- `WebClient` is an analog and potential successor of `RestTemplate` in the inter-application exchange and micro-service architecture.
- Spring Security 5 support both reactive and non-reactive programing model for securing web apps.

WEB CLIENT - the old-timer `RestTemplate`, introduced in Spring 3.0 work with nonreactive domain types and collections, and can't work with Reactive Types (`Flux/Mono`).
The alternative is `WebClient` - send/receive Reactive types when making requests to external APIs, has fluent builder style interface describe and send requests.

1. create instance of `WebClient` (or inject it as bean)
2. Specify HHTP method of the request
3. Specify URI and Headers (optional)
4. Submit the request
5. Consume the responce

```
@Bean // in any @Configuration class
public WebClient webClient() {
//  return WebClient.create();
    return WebClient.builder()
            .baseUrl("https://jsonplaceholder.typicode.com)
            .defaultHeader(HttpHeaders.CONTENT_TYPE,
                           MediaType.APPLICATION_JSON_VALUE)
            .build();
}
```

```
@RestController
@RequestMapping(path = "/webclient", produces = "application/json")
public class WebClientController {

    private static final String SERVER_ROUTE =
                                 "https://jsonplaceholder.typicode.com";
    @Autowired
    private WebClient webClient;

    @GetMapping("/all")
    public Flux<User> getAll() {
        return WebClient.create()// create basic WebClient here
            .get() // GET http method
            .uri(SERVER_ROUTE + "/users") // server uri, concatenated
            .retrieve() // return simple obj ResponseSpec
            .bodyToFlux(User.class) // subscribe to body, as Flux<User>
            .timeout(Duration.ofMillis(2000));
   } // to apply additional operations to Mono subscribe to it
     // mono.subscribe(element -> {..});
```

```
curl -X GET http://localhost:8080/webcleint/all
```

```
   @GetMapping("/{id}")
   public Mono<String> getById(@PathVariable("id") String id) {
       return webClient // WebClient from Bean in @Configuration
            .method(HttpMethod.GET)
            .uri("/users/{id}", id) // server uri, with argument
            .retrieve()
            .onStatus(status -> status == HttpStatus.NOT_FOUND,
                response -> Mono.just(new NoSuchElementException(“”)))
            .bodyToMono(String.class);
   }
```

```
curl -X GET http://localhost:8080/webcleint/2
```

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Mono<User> createUser(@RequestBody User user) {
    return webClient
            .post()
            .uri("/users")
            .header(HttpHeaders.CONTENT_TYPE,
                    MediaType.APPLICATION_JSON_VALUE)
            .accept(MediaType.APPLICATION_JSON)
            .bodyValue(user) // or .body(Mono.just(user), User.class)
            .retrieve() // return simple ResponseSpec obj
            .bodyToMono(User.class);
}
```

```
curl -X POST -H 'Content-Type: application/json' -d '{}' http://
localhost:8080/users/
```

`.retrieve()` method return `ResponceSpec` (wrapping the response), on which we can apply `.bodyToMono()`, `.bodyToFlux()`, `.onStatus()`, ect..

`.exchangeToMono()` or `.exchangeToFlux()` methods return `ClientResponse` (again wrapping the response), on which we can use all data like payload(body), headers and cookies.

```
@PutMapping
public Mono<User> updateUser(@RequestParam("id") String id) {
    return webClient
            .put()
            .uri("/users/{id}", id)
            .exchangeToMono(response -> { // return full ClientResponse
                if (response.headers().header("X_Auth").isEmpty()
                        && response.cookies().containsKey("A")) {
                    return Mono.empty();
                }
                return Mono.just(response); // get Mono<ClientResponse>
            })
            .flatMap(cr -> cr.bodyToMono(User.class));
}
```

```
curl -X PUT -H ‘X_Auth: 1234' -d '{}' http://localhost:8080/users/
```

```
@DeleteMapping(path = "/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public Mono<Void> deleteById(@PathVariable("id") String id) {
    return webClient
            .delete()
            .uri("/users/{id}", id)
            .retrieve()
            .onStatus(HttpStatus::is4xxClientError,
                    response -> Mono.just(new NoSuchElementException("Bad
client request")))
            .bodyToMono(Void.class); // response body is empty
}
```

```
curl -X DELETE http://localhost:8080/users/5
```

R2DBC (Reactive Relational Database Connectivity) - enable nonblocking persistence using reactive types (`Flux/Mono`) to relational db, like (MySQL, PostgeSQL, Oracle, H2). Spring Data R2DBC auto repository support, similar to Spring Data JDBC.

```
<dependency> <!-- connectivity tool -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>

<dependency> <!– connectivity db driver -->
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>

<dependency> <!-- database -->
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
```

DIFFERENCES

- Required setter's methods on properties.
- On save, obj non-null ID will be update.
- Collection referencing by ids (not objects), direct relationship is not supported (currently)

```
@Data // incl. @ReqArgsCtor
@NoArgsConstructor
@EqualsAndHashCode(exclude = "id")
public class Player {

    @Id
    private Long id; // when saving, obj with non-null ID will be update!

    // from lombok, instead final to enforce it in ReqArgsCtor
    private @NonNull String name; // R2DBC require setters on properties!
}

create table Player (
    id identity,
    name varchar(10) not null
)

@Data
@NoArgsConstructor
public class Team {

    @Id
    private Long id;

    private @NonNull String name;

    private Set<Long> playerIds = new HashSet<>();
}   // references to related obj IDs


create table Team (
    id identity,
    name varchar(10) not null,
    player_ids array -- integer[] for PostgreSQL
)
```

RSOCKET - protocol for binary asynchronous inter application communication, based on Reactive Streams, it is alternative on the blocking HTTP based communication.
Offer 4 distinct communication models.
// example: old style letter in an envelope, send and long wait for answer..
much like request-response model HTTP and REST.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-rsocket</artifactId>

@MessageMapping(“user/{id}") // handle incoming msg on this route
@DestinationVariable("name") String name // placeholder, to extract value
RSocketRequester.Builder // bean to send a request,
                           Spring Boot auto create it in App context
```

REQUEST-RESPONSE - mimic the HTTP, the client issue single request (`Mono<Something>`) and the server responded with single response (`Mono<Anything>`).
Look similar to HTTP model, however it is nonblocking and based on reactive types, making more efficient use of the threads.

```
@Controller // SERVER
public class RsocketServerController {

    @MessageMapping("user/{id}") // handle incoming msg on this route
    public Mono<User> getUserMsg( // payload is received as Mono
                 @DestinationVariable("id") Long id, Mono<String> name) {
        log.info("User {} with id {} received.", name, id);
        return Mono.just(new User(id, name.toString()));
    }
}
spring.rsocket.server.port=3000
// application/properties: enable server and specify port to listen to



@Configuration // CLIENT
public class RSocketClientConfig {

    @Bean
    public ApplicationRunner sender(RSocketRequester.Builder
requestBuilder) {
        return args -> {
            RSocketRequester tcp = requestBuilder.tcp("localhost", 3000);

            tcp.route("user/{id}", 5) // route to be sent to
                    .data("John") // with msg payload, in example String
                    .retrieveMono(User.class)
                    .subscribe(res -> log.info("Response: {}", res));
        }; // subscribe to received Mono<User> and handle payload
    }
}
```

REQUEST-STREAM - the client issue single request (`Mono<Something>`) and the server responded with stream of zero, one or many values in a stream (`Flux<Anything>`).

```
@MessageMapping("allByName/{name}") // handle incoming msg on this route
public Flux<User> getAllUsersByName( // no msg payload!
                              @DestinationVariable("name") String name) {
    return Flux.interval(Duration.ofSeconds(1)), // response with Stream
             .map(el -> new User(new Random().nextLong(), name));
}

tcp.route("allByName/{name}", "John") // route to be sent to
         .retrieveFlux(User.class) // subscribe to received Flux<User>
         .doOnNext(el -> log.info("Id{} name{}", el.getId(), el.getName())
         .subscribe(); // handle payload
```

FIRE-AND-FORGET - the client issue single request (`Mono<Something>`) and the server do NOT respond, return empty `Mono<Void>`.

```
@MessageMapping("display")
public Mono<Void> displayUser(Mono<User> userMono) { // msg with payload
    return userMono.doOnNext(user -> log.info(user.getName()))
             .thenEmpty(Mono.empty()); // simply NO response
}

  tcp.route("display") // route to be sent to
             .data(new User()) // some data payload
             .send() // instead of retrieveMono/Flux, simply send
             .subscribe();
};
```

CHANNEL - the client open bidirectional channel, the client send stream of values `Flux<Something>` and server respond with `Flux<Anything>`, both can exchange data at any time.

```
@MessageMapping("team") // msg with Flux payload
public Flux<Team> exchangeData(Flux<Player> playerFlux) {
    return playerFlux.doOnNext(el -> log.info("Player {}", el.getName()))
             .map(el -> new Team()); // get the Team from repo
}   // multiple responses as Flux

tcp.route("team") // route to be sent to
         .data(Flux.fromIterable(List.of(new Player(), new Player())))
         .retrieveFlux(Team.class) // opens bidirectional channel
         .subscribe(team -> log.info("Team {}", team.getName()));
```

RSOCKET OVER WEBSOCKET - default is over tcp, but client might be JS in browser or some firewall restrictions over tcp do not allow specific port… WebSocket works over a route (unlike tcp over a port).

In client: `requesterBuilder.websocket(URI.create(“ws://localhost:8080/abc"))`

```
# RSocket trasport over WebSocket
spring.rsocket.server.transport=websocket
spring.rsocket.server.mapping-path=/abc
```

resources:

https://github.com/Ivan-Tashev/ProjectReactor

further reading:

Securing reactive web API

```
@EnableWebFluxSecurity,
@Bean SecurityWebFilterChain, @Bean ReactiveUserDetailsServ
```

Functional request handlers

```
@Bean public RouterFunction<?> hello {
   return route(GET ”/hello”), request -> ok().body(just(“Hello”));
```

Web Test Client

```
WebTestClient testClient = WebTestClient.bindToController(
      new SomeController(mockedRepo)).build();
```

Thank you.