Reactive Programming in Java

by Vadym Kazulkin and Rodion Alukhanov, ip.labs GmbH

Conta
ct

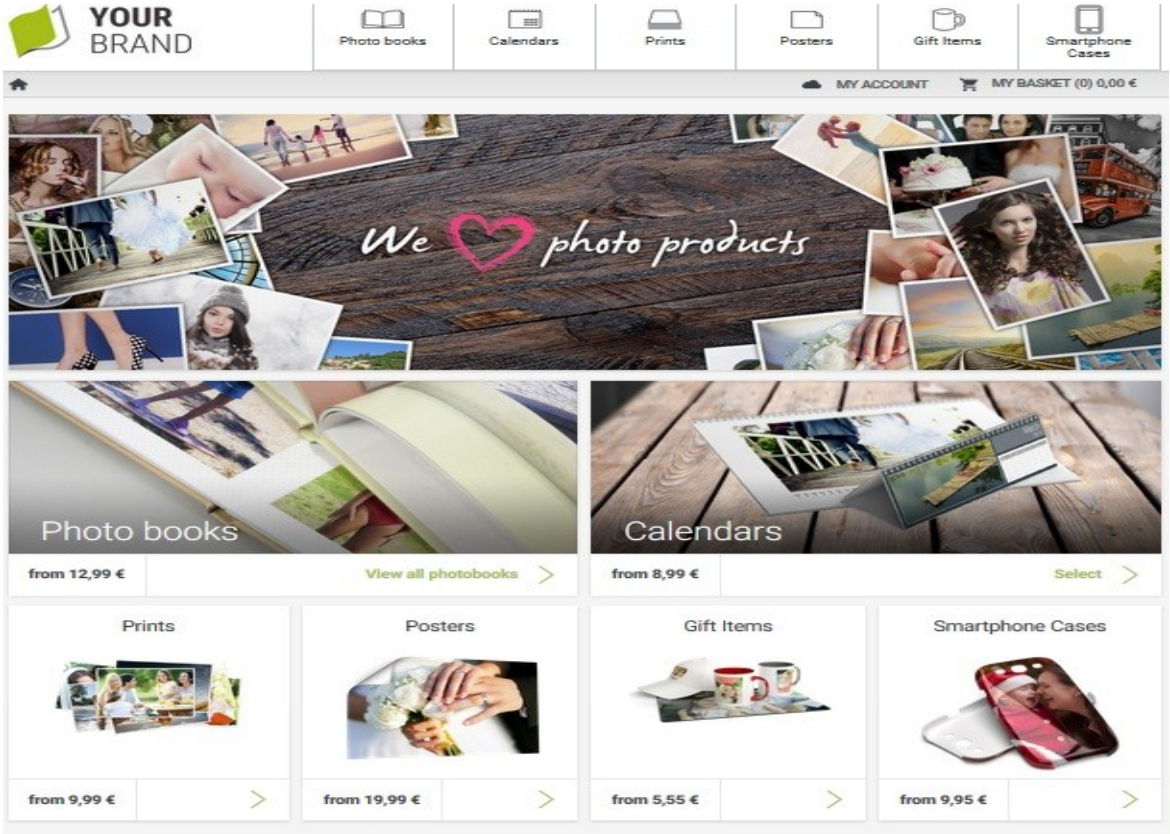
Vadym Kazulkin, ip.labs GmbH:

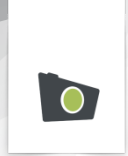
v.kazulkin@iplabs.de
www.xing.com/profile/Vadym_Kazulkin
@VKazulkin

Rodion Alukhanov, ip.labs GmbH :

r.alukhanov@iplabs.de
www.xing.com/profile/Rodion_Alukhanov

ip.labs GmbH

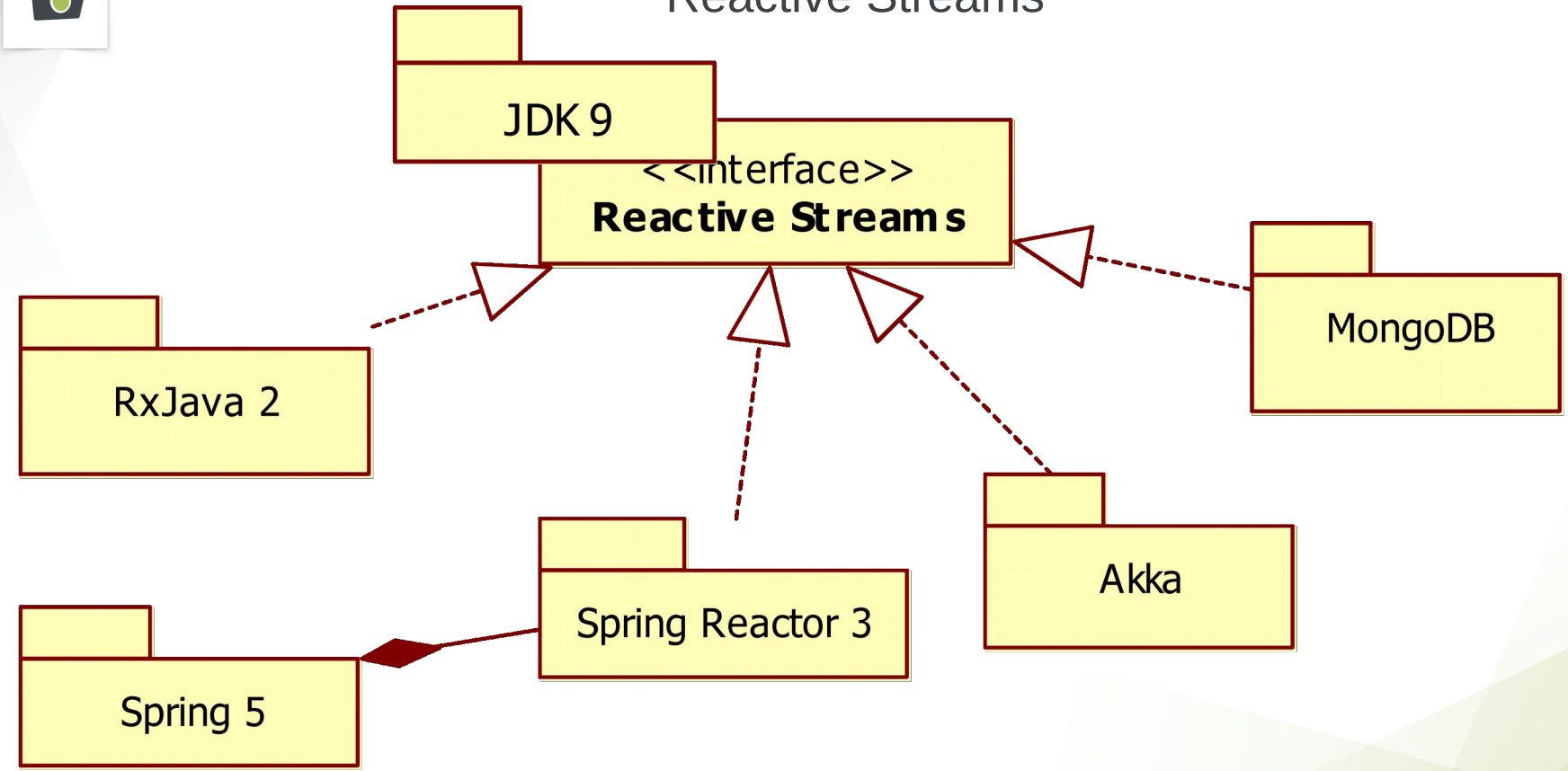

YOUR BRAND
Photo books
Calendars
Prints
Posters
Gift Items
Smartphone Cases
MY ACCOUNT
MY BASKET (0) 0,00 €
We ♥ photo products
Photo books
from 12,99 €
View all photobooks
Calendars
from 8,99 €
Select
Prints
from 9,99 €
Posters
from 19,99 €
Gift Items
from 5,55 €
Smartphone Cases
from 9,95 €



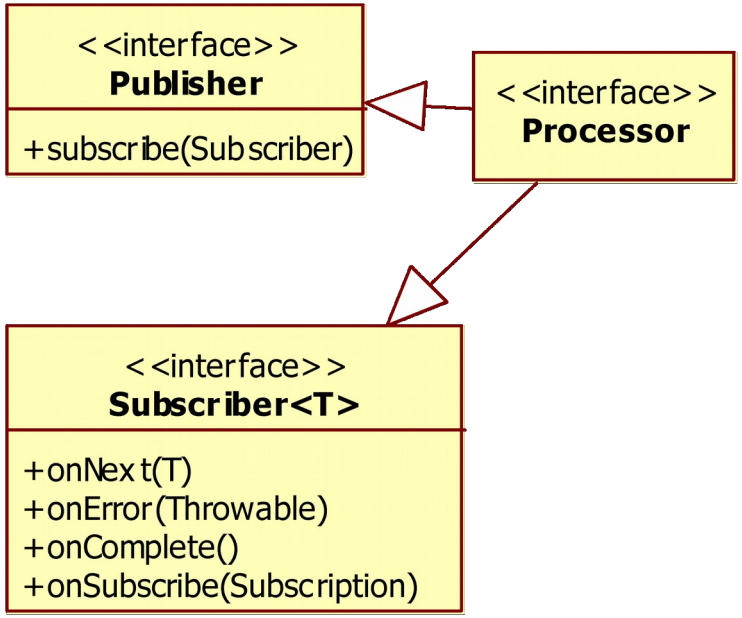
Agenda

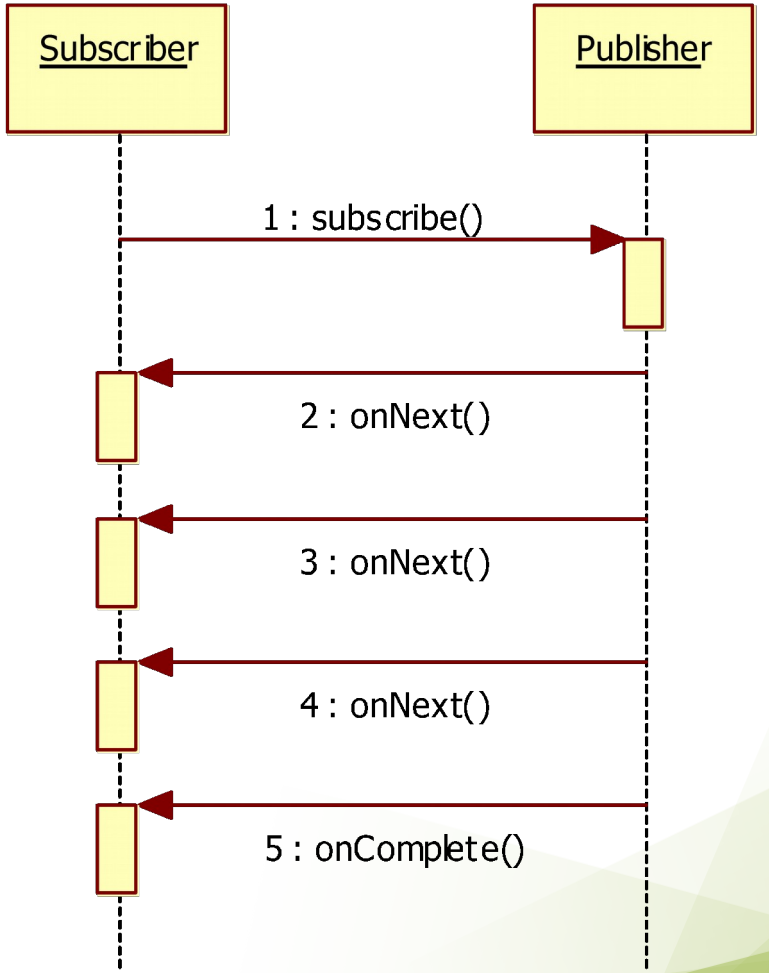
- Reactive Programming in general
- Reactive Streams and JDK 9 Flow API
- RxJava 2
- Spring Reactor 3
- Demo of reactive application with Spring 5, Spring Boot 2, Netty, MongoDB and Thymeleaf technology stack

Reactive Streams

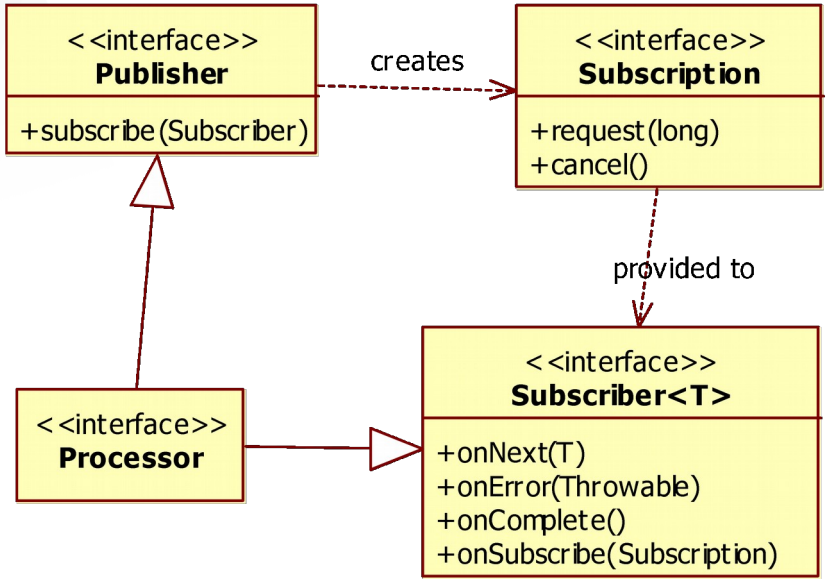


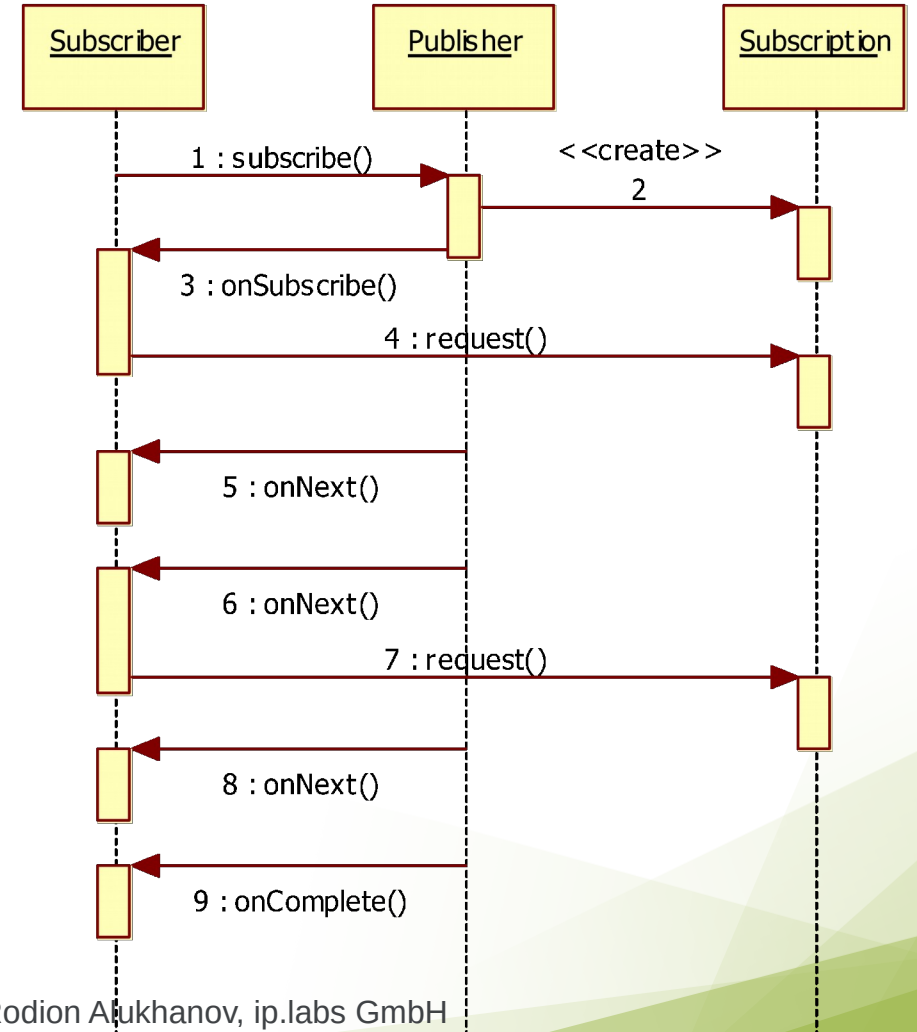
Subscriber/Publisher



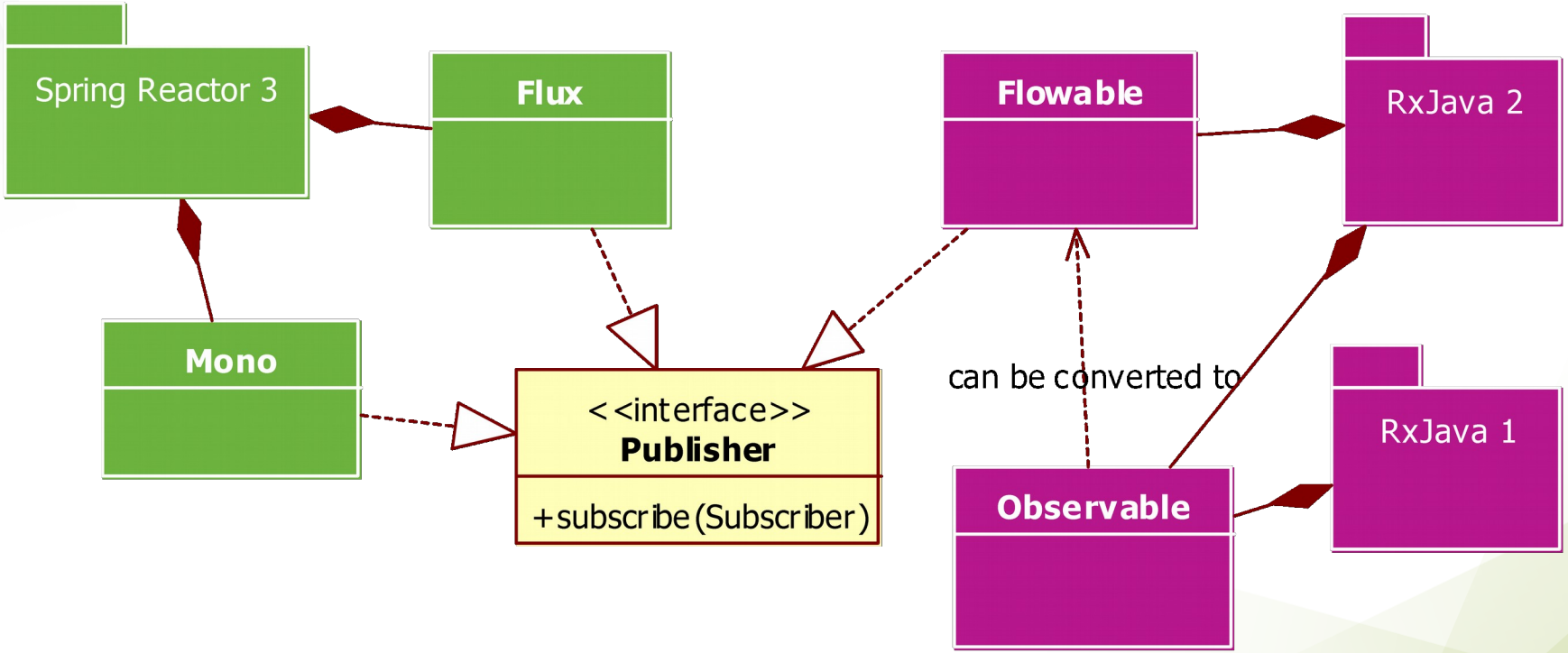


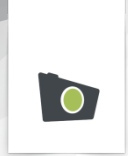
Subscription





Publisher Implementations





RxJava 2

Classic vs. Reactive (Cold Publisher)

```
public List<Photo> getPhotos() {
    List<Photo> result = …
    while (...) {
        result.add(...);
    }
    return result;
}
```

```
for (Item item : getPhoto()) {
    System.out.println(item.toString());
}
```

```
Observable<Photo> publisher = Observable.create(subscriber -> {
    while (...) {
        Photo photo = ...
        subscriber.doNext(photo);
    }
    subscriber.onCompleted();
});
```

```
publisher.subscribe(item->{
    System.out.println(item.toString());
});
```

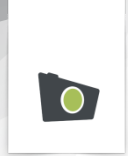
RxJava 2 Basics

```
Observable<Customer> customers = ...
Observable<Customer> adults = customers.filter(c -> c.age > 18);

Observable<Address> addresses = adults.map(c -> c.getDefaultAddress());


Observable<Order> orders = customers.flatMap(c ->
    Observable.fromArray(c.getOrders())
);

Observable<Picture> uploadedPhotos = customers.flatMap(c ->
    Observable.fromArray(dao.loadPhotosByCustomer(c.getId()))
);
```

Example. Print Photo Book

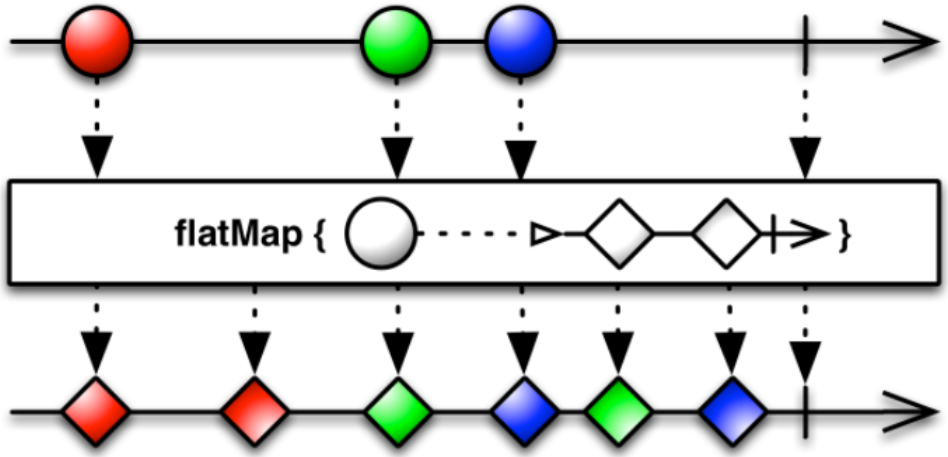
Fetching photo IDs from photo-book	**0 ms**
Loading meta data from database (ID, Dimentions, Source etc.)	**10 x 100ms**
Loading images from server HDD if avaliable	**90 x 50ms**
Loading missing images from source (for example Cloud Service)	**10 x 500ms**
Validating images	**100 x 50ms**
+	
User authentication & authorisation	**max 500ms**

100 images



Loading Meta Data. Buffer.

```
Observable<Integer> ids = Observable.fromIterable(book.getPhotoIDs());

Observable<List<Integer>> idBatches = ids.buffer(10);

Observable<MetaData> metadata =
    idBatches.flatMap(ids -> metadataDao.loadingMetadata(ids));
```



Loading Picture from HDD. FlatMap with at most a single result

```
Observable<PictureFile> fromDisk =
    ids.flatMap(id -> loadingFromDisk(id));


public Observable<PictureFile> loadingFromDisk(Integer id)    {
    Observable<PictureFile> result = Observable.create(s -> {
        try {
            s.onNext(pictureDao.loadFromHdd(id));
        } catch (PictureNotFound e) {
            System.out.println("Not found on disk " + id);
        }
        s.onComplete();
    }
    return result;
}
```

Loading from Cloud. Filter

```
Observable<PictureFile> fromCloud =
        metadata.filter(m->m.isFromCloud())
                .flatMap(id -> loadingFromCloud(id.id));
```

---

metadata 1 metadata 2 metadata 3 metadata 4 metadata 5	disk-file 1 disk-file 2 disk-file 4	cloud-file 3 cloud-file 5

Merging Cloud and HDD

```
Observable<PictureFile> allPictures = fromDisk.mergeWith(fromCloud);
```

---

metadata 1	disk-file 1
metadata 2	cloud-file 3
metadata 3	disk-file 2
metadata 4	disk-file 4
metadata 5	cloud-file 5

Joining Metadata with Picture Files

```
Observable<Pair<MetaData, PictureFile>> pairs =
    metadata.join(allPictures, ..., Pair::of);
```

---

metadata 1 + disk-file 1
metadata 2 + disk-file 1
metadata 3 + disk-file 1
metadata 2 + disk-file 2
metadata 3 + cloud-file 3

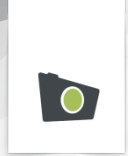
```
pairs = pairs.filter(p->p.fst.id == p.snd.id)
```

Joining Metadata with Picture Files

```
Observable<Picture> pictures = pairs.map(p->new Picture(p.snd, p.fst));
```

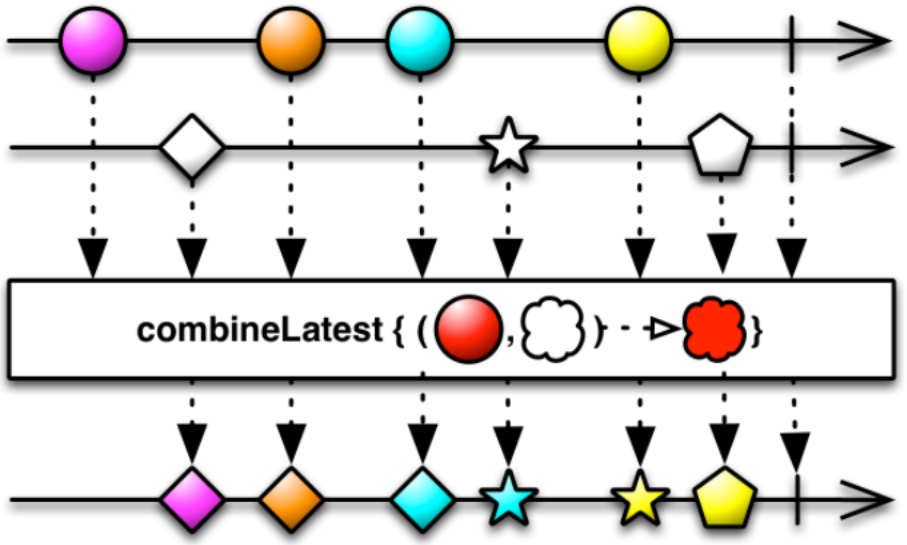
Validating Result

```
Observable<Picture> validated = pictures.flatMap(p→validation(p));

public static Observable<Picture> validation(Picture p) {
    return Observable.<Picture>create(s -> {
        // do the job
        s.onNext(p);
        s.onComplete();
    });
}

Observable<Picture> validated = pictures.map(p→validation2(p));

public static Picture validation2(Picture p) {
    // do the job
    return p;
}
```



Adding Authentication

```
Observable<Boolean> auth = auth();

Observable<Picture> result =
     Observable.combineLatest(auth, validated, (a,p)->p);
```



Back in the „Real World“

```
result.blockingForEach((item)->{
    System.out.println("FINISHED -> " + item.id);
});
```

Reactive Code

```
Observable<Integer> ids =  loadingIds();
Observable<MetaData> metadata = ids.buffer(10).flatMap(id -> loadingMetadata(id));

Observable<PictureFile> fromDisk = ids.flatMap(id -> loadingFromDisk(id));
Observable<PictureFile> fromCloud = metadata.filter(m->m.cloud).
     flatMap(id -> loadingFromCloud(id.id));
Observable<PictureFile> allPictures = fromDisk.mergeWith(fromCloud);

Observable<Long> wait = Observable.never();
Observable<Pair<MetaData, PictureFile>> pairs =
     metadata.join(allPictures, (m)-> wait, (f)-> wait, Pair::of);

Observable<Picture> pictures =
     pairs.filter(p->p.fst.id == p.snd.id).map(p->new Picture(p.snd, p.fst));

Observable<Picture> validated = pictures.flatMap(p->validation(p));

Observable<Boolean> auth = auth();

Observable<Pair<Boolean, Picture>> result =
     Observable.combineLatest(auth, validated,(a,p)->p);
```

Where is Concurrency?

```
private Scheduler scheduler = Schedulers.io();

public Observable<PictureFile> loadingFromDisk(Integer id)     {
    Observable<PictureFile> result = Observable.create(s -> {
        try {
            s.onNext(pictureDao.loadFromHdd(id));
        } catch (PictureNotFound e) {
            System.out.println("Not found on disk " + id);
        }
        s.onComplete();
    }
    return result.subscribeOn(scheduler);
}
```

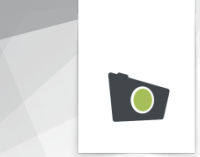
Compare Results

```
Observable<Integer> ids =  loadingIds();
Observable<MetaData> metadata = ids.buffer(10).flatMap(id -> loadingMetadata(id));

Observable<PictureFile> fromDisk = ids.flatMap(id -> loadingFromDisk(id));
Observable<PictureFile> fromCloud = metadata.filter(m->m.cloud).
    flatMap(id -> loadingFromCloud(id.id));
Observable<PictureFile> allPictures = fromDisk.mergeWith(fromCloud);

Observable<Long> wait = Observable.interval(1000, TimeUnit.SECONDS);
Observable<Pair<MetaData, PictureFile>> pairs =
    metadata.join(allPictures, (m)-> wait, (f)-> wait, Pair::of);

Observable<Picture> pictures =
    pairs.filter(p->p.fst.id == p.snd.id).map(p->new Picture(p.snd, p.fst));

Observable<Picture> validated = pictures.flatMap(p->validation(p));

Observable<Boolean> auth = auth();

Observable<Pair<Boolean, Picture>> result =
    Observable.combineLatest(auth, validated, Pair::of);
```

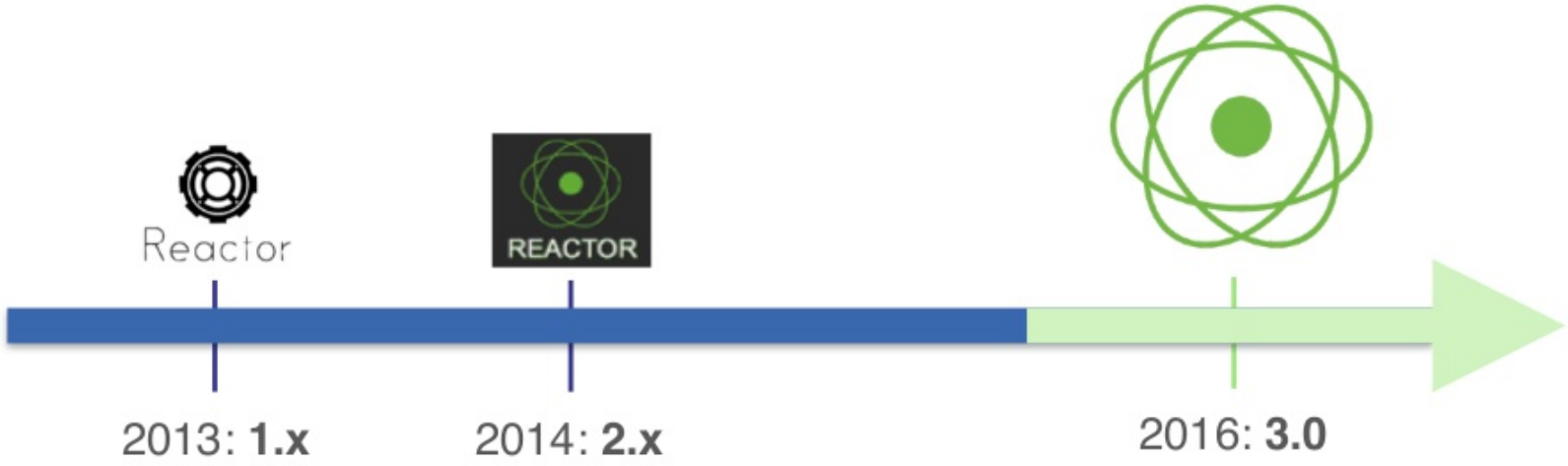
Classic -> 20 sec

Reactor -> 1 sec

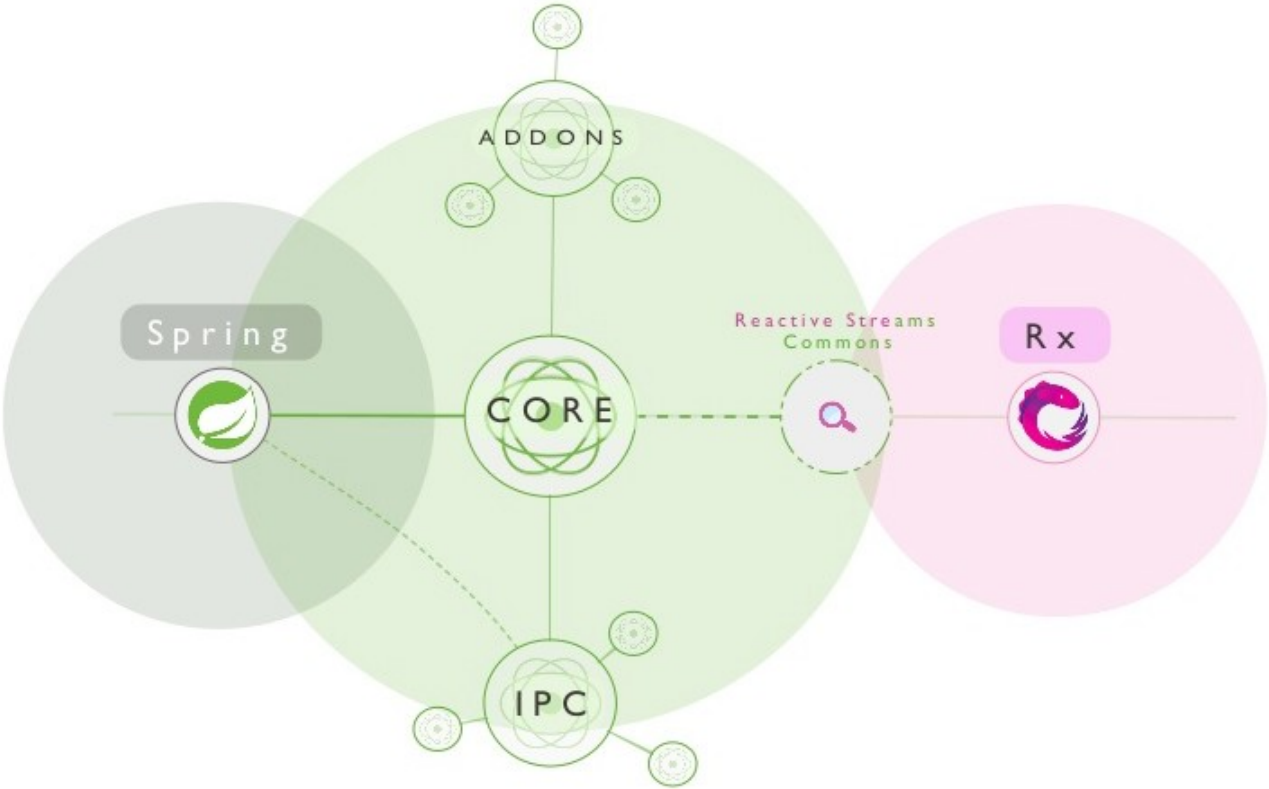
Spring Reactor 3



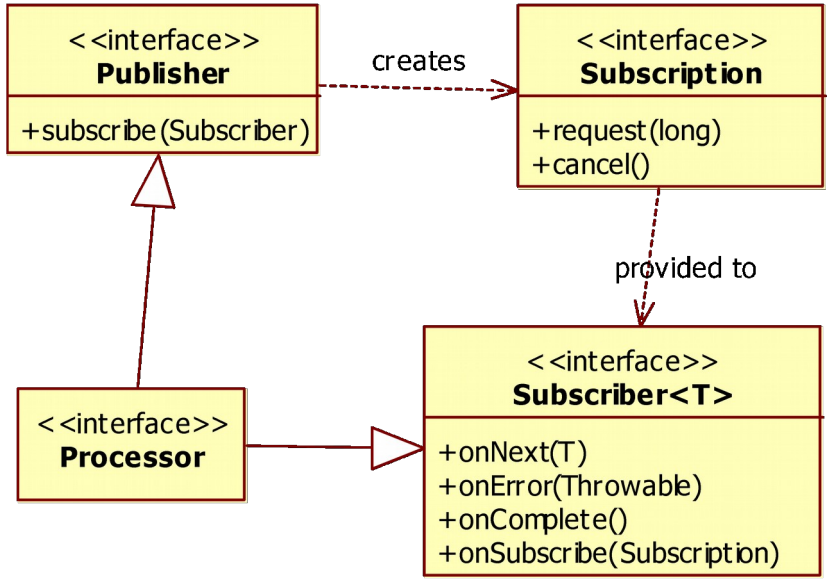
Spring Reactor Timeline

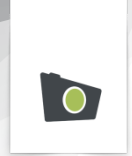


Spring Reactor 3 Big Picture



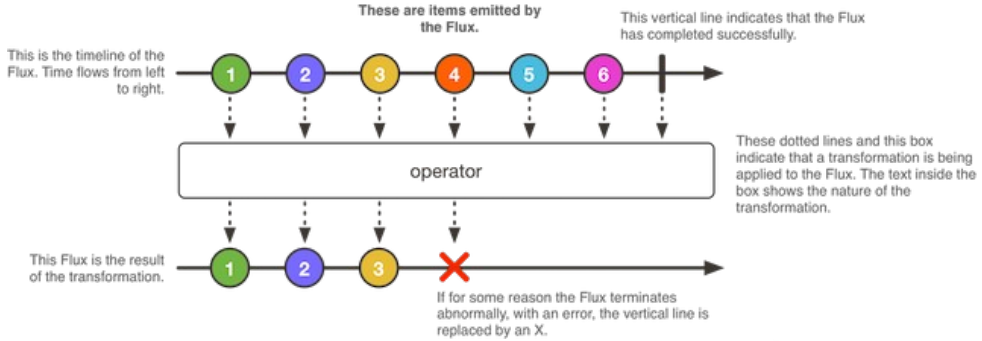
Reactive Stream Interfaces

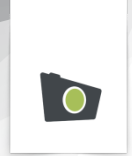




Spring Reactor 3 Main Types

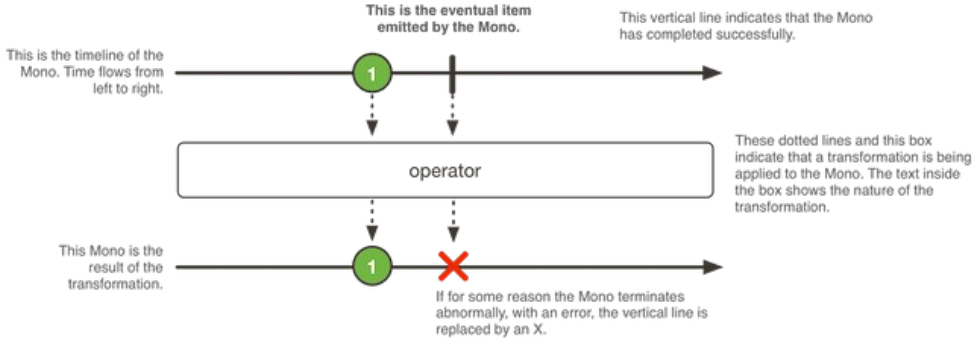
Flux implements Publisher
is capable of emitting of 0 or more items





Spring Reactor 3 Main Types

Mono implements Publisher
can emit **at most** once item



Spring Reactor 3 Examples

Publisher creation

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");
Mono<String> productTitles = Mono.just("Print 9x13");

Flux<String> productTitles = Flux.fromIterable(Arrays.asList("Print 9x13",
"Photobook A4", "Calendar A4"));
Flux<String> productTitles = Flux.fromArray(new String[]{"Print 9x13",
"Photobook A4", "Calendar A4"});
Flux<String> productTitles = Flux.fromStream(Stream.of("Print 9x13",
"Photobook A4", "Calendar A4“));

Mono.fromCallable(), Mono.fromRunnable(), Mono.fromFuture()

Flux.empty(), Mono.empty(), Flux.error(), Mono.error()
```

Spring Reactor 3 Examples

Event subscription

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

productTitles.subscribe(System.out::println);
```

Output:

Print 9x13
Photobook A4
Calendar A4

Spring Reactor 3 Examples

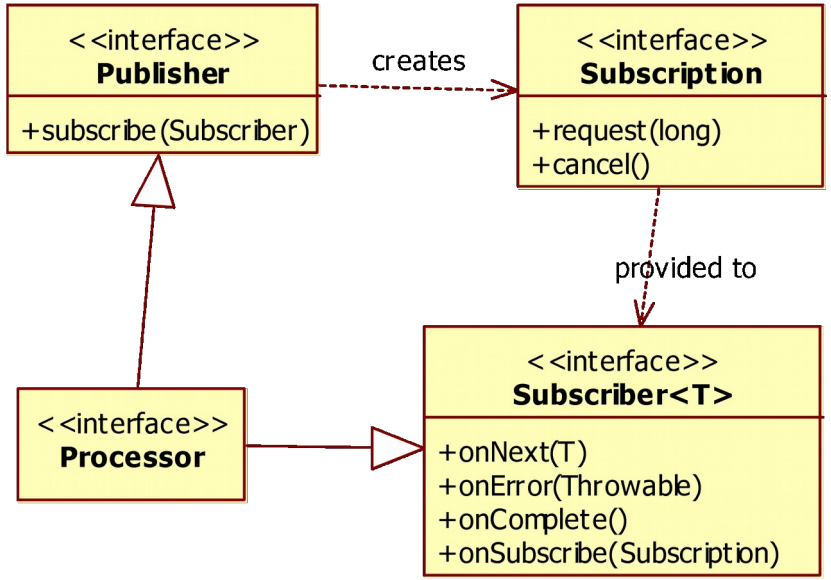
Logging

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");
productTitles.log().subscribe(System.out::println);
```

Output:

```
INFO reactor.Flux.Array.2 - | onSubscribe()
INFO reactor.Flux.Array.2 - | request(unbounded)
INFO reactor.Flux.Array.2 - | onNext(Print 9x13)
Print 9x13
INFO reactor.Flux.Array.2 - | onNext(Photobook A4)
Photobook A4
INFO reactor.Flux.Array.2 - | onNext(Calendar A4)
Calendar A4
INFO reactor.Flux.Array.2 - | onComplete()
```

Reactive Stream Interfaces



Spring Reactor 3 Examples

Event subscription with own Subscriber

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");
productTitles.subscribe(new Subscriber<String>() {

    @Override
    public void onSubscribe(Subscription s) {
        s.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(String t) {
        System.out.println(t);
    }
    @Override
    public void onError(Throwable t) {
    }
    @Override
    public void onComplete() {
    }
});
```

Spring Reactor 3 Examples

Event subscription with custom Subscriber with back-pressure

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");
productTitles.subscribe(new Subscriber<String>() {

    private long count = 0;
    private Subscription subscription;

    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(2);
    }

    public void onNext(String t) {
        count++;
        if (count>=2) {
            count = 0;
            subscription.request(2);
        }
     }
...
```

Spring Reactor 3 Examples

Event subscription with custom Subscriber with back-pressure

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");
productTitles.log().subscribe(new Subscriber<String>()
{subscription.request(2);..}
```

Output:

```
INFO reactor.Flux.Array.2 - | onSubscribe()
INFO reactor.Flux.Array.2 - | request(2)
INFO reactor.Flux.Array.2 - | onNext(Print 9x13)
Print 9x13
INFO reactor.Flux.Array.2 - | onNext(Photobook A4)
Photobook A4
INFO reactor.Flux.Array.2 - | request(2)
INFO reactor.Flux.Array.2 - | onNext(Calendar A4)
Calendar A4
INFO reactor.Flux.Array.2 - | onComplete()
```

Spring Reactor 3 Operations

Transforming (map, scan)
Combining (merge, startWith)
Filtering (last, skip)
Mathematical (count, average, max)
Boolean (every, some, includes)

Source: http://rxmarbles.com

Spring Reactor 3 Examples

Elements filtering

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Flux<String> productTitlesStartingWithP =
productTitles.filter(productTitle-> productTitle.startsWith("P"));

productTitlesStartingWithP.subscribe(System.out::println);
```

Output:

Print 9x13
Photobook A4

Spring Reactor 3 Examples

Elements counter

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Mono<Long> productTitlesCount = productTitles.count();

productTitlesCount.subscribe(System.out::println);
```

Output:

```
3
```

Spring Reactor 3 Examples

Check if all elements match a condition

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Mono<Boolean> allProductTitlesLengthBiggerThan5=
productTitles.all(productTitle-> productTitle.length() > 5);

allProductTitlesLengthBiggerThan5.subscribe(System.out::println);
```

Output:

true

Spring Reactor 3 Examples

Element mapping

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Flux<Integer> productTitlesLength =
productTitles.map(productTitle-> productTitle.length()) ;

productTitlesLength.subscribe(System.out::println);
```

Output:

```
10
12
11
```

Spring Reactor 3 Examples

Zipping of 2 Publishers

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Flux<Double> productPrices = Flux.fromIterable(Arrays.asList(0.09, 29.99, 15.99));

Flux<Tuple2<String, Double>> zippedFlux = Flux.zip(productTitles, productPrices);
zippedFlux.subscribe(System.out::println);
```

Output:

```
[Print 9x13,0.09]
[Photobook A4,29.99]
[Calendar A4,15.99]
```

Spring Reactor 3 Examples

Parallel processing

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");
Flux<Double> productPrices = Flux.fromIterable(Arrays.asList(0.09, 29.99, 15.99));

Flux.zip(productTitles, productPrices)
.parallel()   //returns ParallelFlux, uses all available CPUs or call
parallel(numberOfCPUs)
.runOn(Schedulers.parallel())
.sequential()
.subscribe(System.out::println);
```

Output:

```
[Print 9x13,0.09]
[Photobook A4,29.99]
[Calendar A4,15.99]
```

Spring Reactor 3 Examples

Blocking Publisher

Blocking Flux:

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Iterable<String> blockingConcatProductTitles =
productTitles.concatWith(anotherProductTitle) .toIterable();
```

or

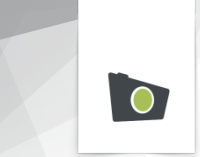
```
Stream<String> blockingConcatProductTitles =
productTitles.concatWith(anotherProductTitle) .toStream();
```

Blocking Mono:

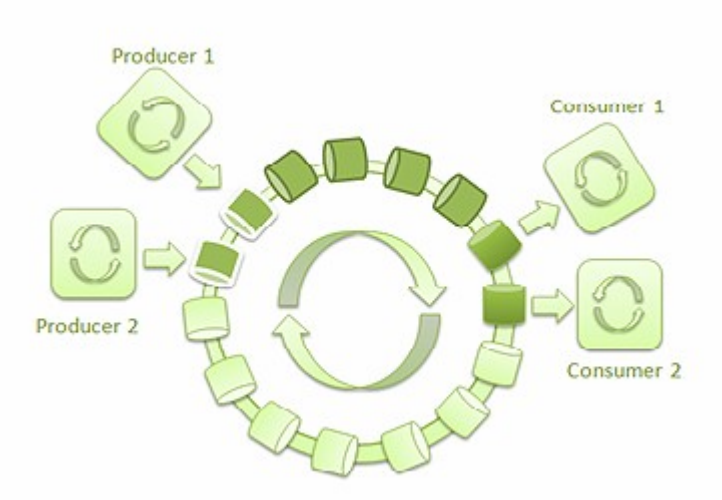
```
String blockingProductTitles = Mono.just("Print 9x13") .block();
```

or

```
CompletableFuture blockingProductTitles = Mono.just("Print 9x13").toFuture();
```



LMAX Disruptor



RingBuffer with multiple producers and consumers

Source: https://github.com/LMAX-Exchange/disruptor/wiki/Introduction

Spring Reactor 3 Examples

Test the publishers

```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

Duration verificationDuration = StepVerifier.create(productTitles).
    expectNextMatches(productTitle -> productTitle.equals("Print 9x13")).
    expectNextMatches(productTitle -> productTitle.equals("Photobook A4")).
    expectNextMatches(productTitle -> productTitle.equals("Calendar A4")).
    expectComplete().
    verify());
```

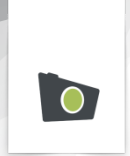
Spring Reactor 3 Examples

Test the publishers

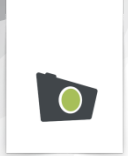
```
Flux<String> productTitles = Flux.just("Print 9x13", "Photobook A4", "Calendar A4");

StepVerifier.create(productTitles).
        expectNextMatches(productTitle -> productTitle.equals("Print 9x13")).
        expectNextMatches(productTitle -> productTitle.equals("Photobook A4")).
        expectNextMatches(productTitle -> productTitle.equals("Calendar A4")).
        expectComplete().
        verify())
```

Output:

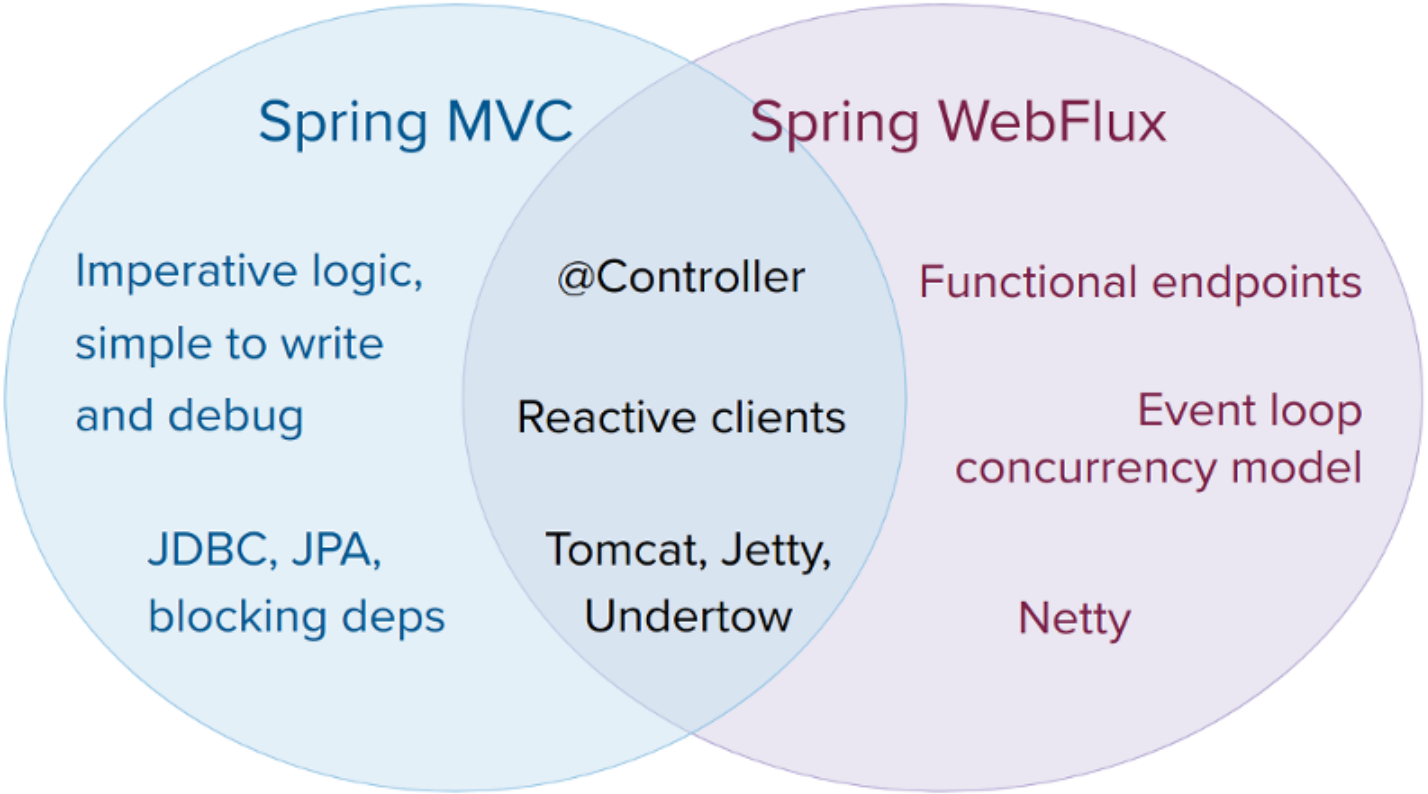
```
Exception in thread "main" java.lang.AssertionError: expectation
"expectComplete" failed (expected: onComplete(); actual: onNext(Calendar
A4));
```

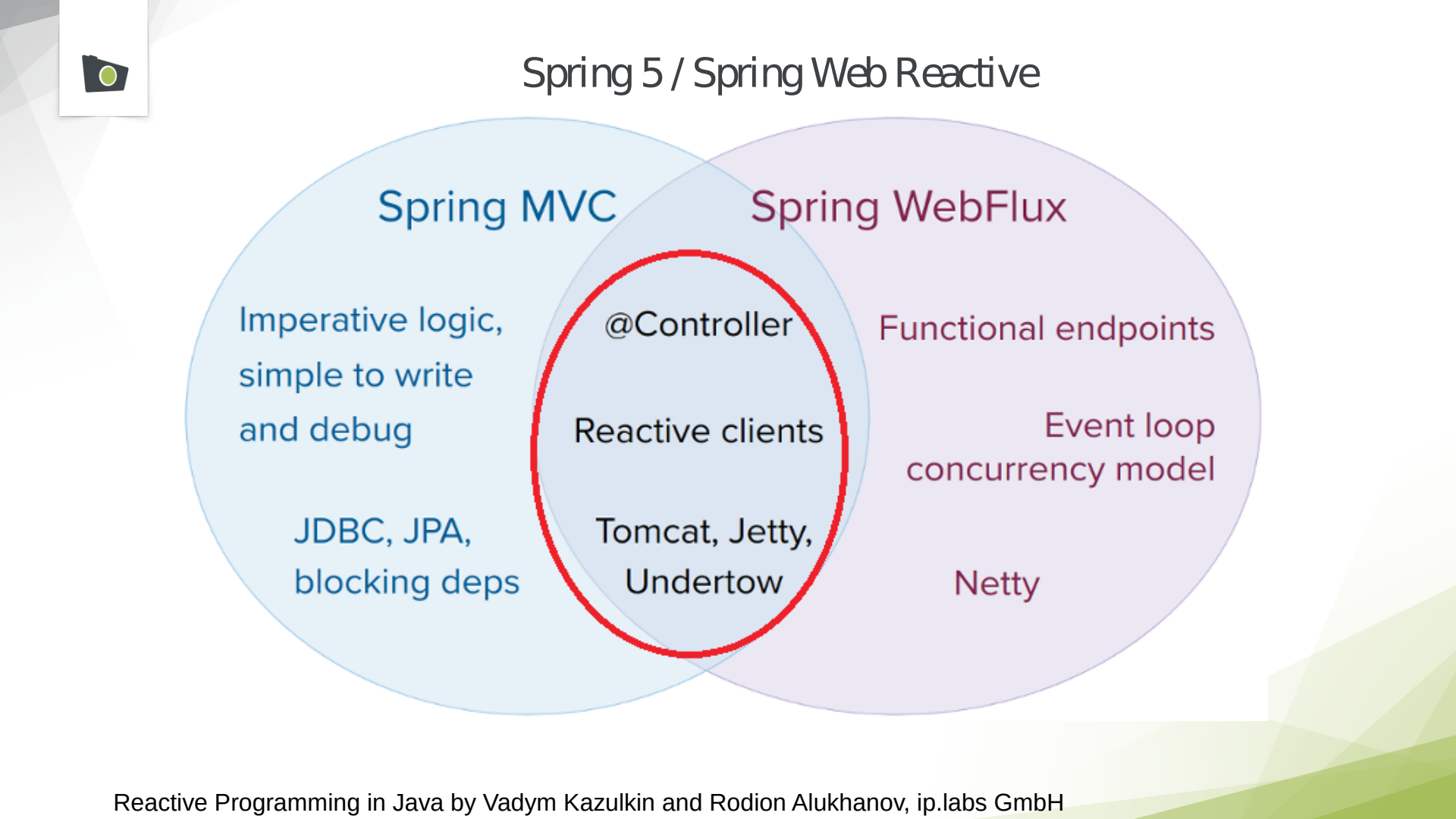


Spring 5 / Spring Boot 2 / Netty / MongoDB / Thymeleaf Demo

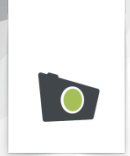


Spring 5 / Spring Web Reactive

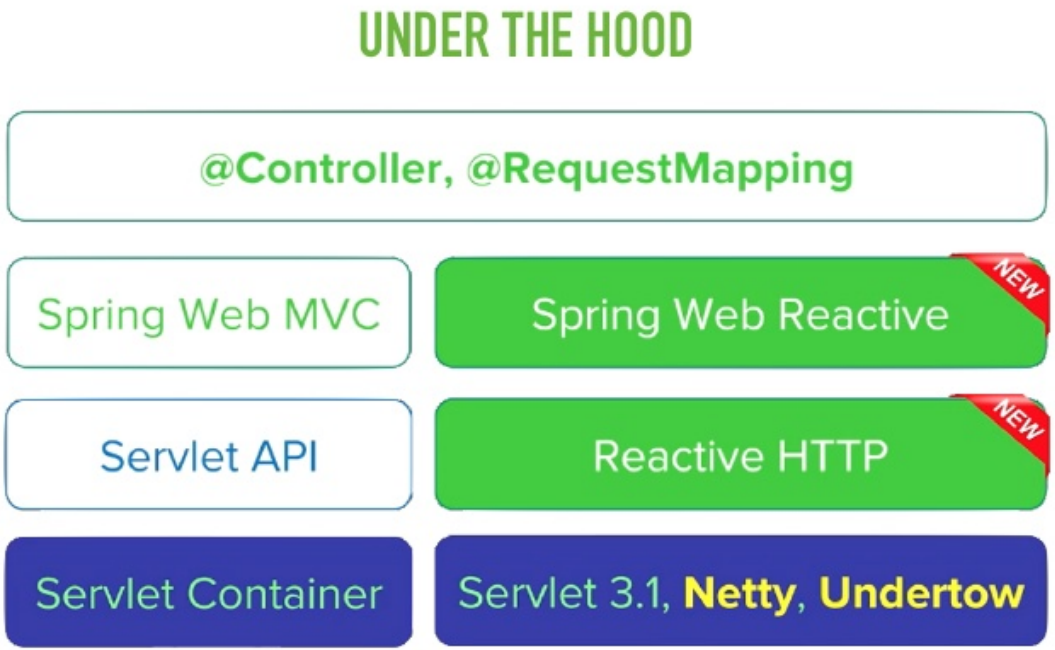




Spring 5 / Spring Web Reactive



Spring 5 / Spring Web Reactive



Dependencies

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <kotlin.version>1.2.21</kotlin.version>
    <kotlin.compiler.jvmTarget>1.8</kotlin.compiler.jvmTarget>
    <kotlin.compiler.jvmSource>1.8</kotlin.compiler.jvmSource>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.module</groupId>
        <artifactId>jackson-module-kotlin</artifactId>
    </dependency>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib-jdk8</artifactId>
    </dependency>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-reflect</artifactId>
    </dependency>
```

Non-blocking Netty Web Client

```
package com.example.services.customer;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class Application {

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

    public @Bean MongoClient mongoClient() {
        return MongoClients.create("mongodb://localhost:27017");
    }

    public @Bean ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoClient(), "customers");
    }

    public @Bean WebClient webClient() {
        return WebClient.builder().clientConnector(new ReactorClientHttpConnector())
                .baseUrl("http://localhost:2222").build();
    }

}
```

MongoDB Reactive Database Template

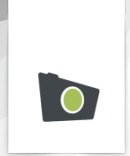
```
package com.example.services.customer;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class Application {

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

    public @Bean MongoClient mongoClient() {
        return MongoClients.create("mongodb://localhost:27017");
    }

    public @Bean ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoClient(), "customers");
    }

    public @Bean WebClient webClient() {
        return WebClient.builder().clientConnector(new ReactorClientHttpConnector())
                .baseUrl("http://localhost:2222").build();
    }

}
```

Model

```
package com.example.services.customer.model;

import org.springframework.data.mongodb.core.mapping.Document;

@Document
public class Customer {

    private String firstName;
    private String lastName;


    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }


    public String getFirstName() {
        return firstName;
    }


    public String getLastName() {
        return lastName;
    }
```

Repository

```
package com.example.services.customer.repository;

import static org.springframework.data.mongodb.core.query.Criteria.where;

@Repository
public class CustomerRepository {

    @Autowired
    private ReactiveMongoTemplate template;


    public Mono<Customer> findByFirstName(String firstName) {
        return template.findOne(query(where("firstName").is(firstName)), Customer.class);
    }

    public Flux<Customer> findAll() {
        return template.findAll(Customer.class);
    }

    public Mono<Customer> save(Mono<Customer> customer) {
        return template.insert(customer);
    }

    public Mono<Void> deleteAllCustomers() {
        return template.dropCollection(Customer.class);
    }

}
```



Thymeleaf Reactive View Resolver

```
package com.example.services.customer;

import org.springframework.context.annotation.Bean;

@Configuration
public class CustomerFluxWebConfig {

    /* ViewResolver for Thymeleaf templates executing in BUFFERED or DATA-DRIVEN mode.
    @Bean
    public ThymeleafReactiveViewResolver thymeleafChunkedAndDataDrivenViewResolver(final ISpringWebFluxTemplateEngine templateEngine){
        final ThymeleafReactiveViewResolver viewResolver = new ThymeleafReactiveViewResolver();
        viewResolver.setTemplateEngine(templateEngine);
        viewResolver.setViewNames(new String[] {"thymeleaf/*chunked*", "thymeleaf/*datadriven*"});
        viewResolver.setResponseMaxChunkSizeBytes(256); // OUTPUT BUFFER size limit
        viewResolver.setOrder(1);
        return viewResolver;
    }
}
```

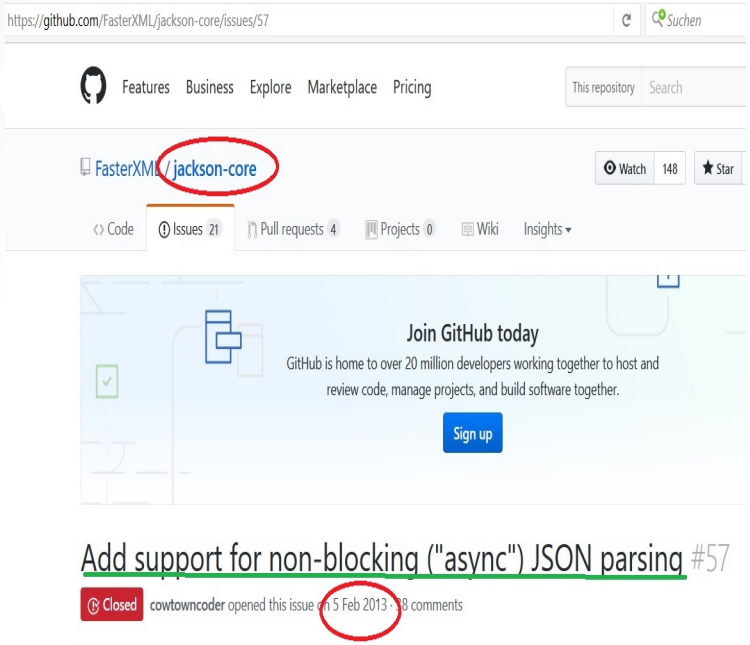
Controller

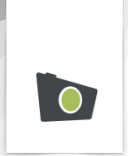
```
@GetMapping("/selectAllCustomersChunked")
public String selectAllCustomersChunked(final Model model) {
    Flux<Customer> customers = repository.findAll();
    model.addAttribute("dataSource", customers); //chunked
    return "thymeleaf/cusomerlist-chunked";
}

@GetMapping("/selectAllCustomersDataDriven")
public String selectAllCustomersDataDriven(final Model model) {
    Flux<Customer> customers = repository.findAll();
    model.addAttribute("dataSource", new ReactiveDataDriverContextVariable(customers, 500)); //data driven
    return "thymeleaf/customerlist-datadriven";

}
```

View

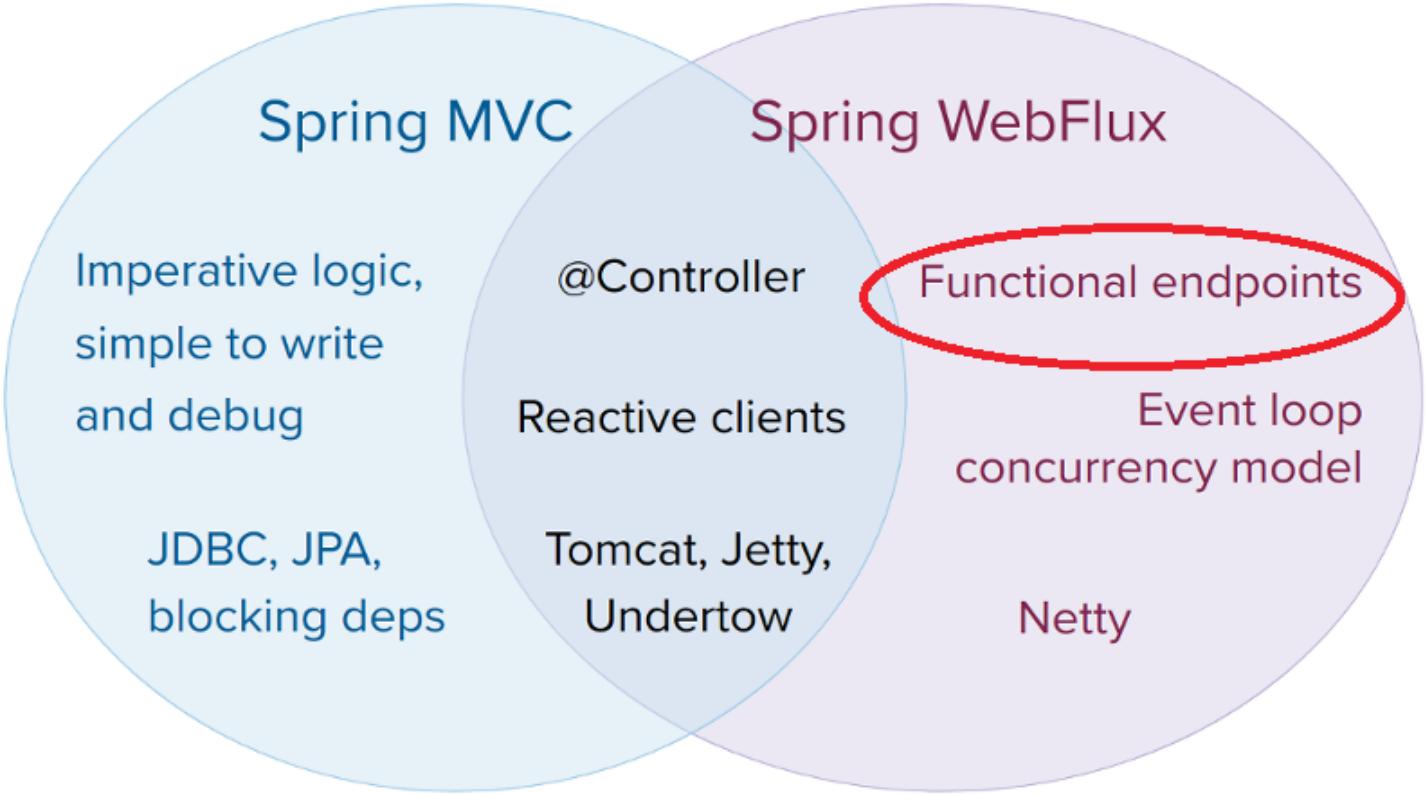
```
<html>

  <head>
    <title>Customer's List  Data Driven</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>

  <body>
    <h3>Mode: <strong>DATA-DRIVEN</strong></h3>

    <table>
      <thead>
        <tr>
          <th>Firstname</th>
          <th>Lastname</th>
        </tr>
      </thead>
      <tbody>
        <tr th:each="e : ${dataSource}">
          <td th:text="${e.firstName}">...</td>
          <td th:text="${e.lastName}">...</td>
        </tr>
      </tbody>
    </table>

  </body>

</html>
```

Non-blocking JSON Parsing with Jackson



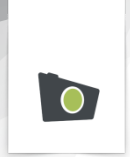


Spring 5 / Spring Web Reactive



Functional Endpoint

- Alternative to the annotated-based programming (uses the same Reactive Web Spring API)
- Lightweight functional programming model
- Functions are used to route and handle requests
- Contracts are designed for immutability



Functional Endpoint

```
@SpringBootApplication
public class CustomerJavaReactiveFEApplication {

    static final String FIRST_NAME_PATH_VARIABLE="firstName";
    @Bean
    RouterFunction<?> router(CustomerHandler handler) {
        return RouterFunctions.route(RequestPredicates.GET("/java/customers"), handler::getAllCustomers).
                and(RouterFunctions.route(RequestPredicates.GET("/java/customers/{"+FIRST_NAME_PATH_VARIABLE+"}"),
                        handler::getCustomerByFirstName));

    }

    public static void main(String[] args) {
        SpringApplication.run(CustomerJavaReactiveFEApplication.class, args);
    }
}

@Component
class CustomerHandler {
    private static final Customer [] CUSTOMERS = {new Customer ("Vadym", "Kazulkin"),
            new Customer("Rodion", "Alukhanov")};

    private CompletableFuture<Customer> findByFirstName(String firstName) {
          return CompletableFuture.supplyAsync(
                  () -> Stream.of(CUSTOMERS).filter(customer -> customer.getFirstName().equals(firstName)).
                  findFirst().get());
    }

    Mono<ServerResponse> getAllCustomers(ServerRequest request) {
        Flux<Customer> customer = Flux.fromStream(Stream.of(CUSTOMERS));
        return ServerResponse.ok().body(BodyInserters.fromPublisher(customer, Customer.class));
    }


    Mono<ServerResponse> getCustomerByFirstName(ServerRequest request) {
        String name = request.pathVariable(CustomerJavaReactiveFEApplication.FIRST_NAME_PATH_VARIABLE);
        return Optional.ofNullable(name).map(this::findByFirstName)
                .map(Mono::fromFuture)
                .map(mono -> ServerResponse.ok().body(BodyInserters.fromPublisher(mono, Customer.class)))
                .orElseThrow(() -> new IllegalStateException("Oops, Error occured!"));
    }
}
```

Benefits of the Reactive Applications

- Efficient resource utilization (spending less money on servers and data centres)
- Processing higher loads with fewer threads

Source: https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape

Uses Cases for Reactive Applications

- External service calls
- Highly concurrent message consumers
- Single page applications with unbounded elements to display

Pitfalls of reactive programming

- For the wrong problem, this makes things worse
- Hard to debug (no control over executing thread)
- Mistakenly blocking a single request leads to increased latency for **all** requests -> blocking all requests brings a server to its knees

Source: https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-server-application

Questions?

Conta
ct

Vadym Kazulkin, ip.labs GmbH:

v.kazulkin@iplabs.de
www.xing.com/profile/Vadym_Kazulkin
@VKazulkin

Rodion Alukhanov, ip.labs GmbH :

r.alukhanov@iplabs.de
www.xing.com/profile/Rodion_Alukhanov

Thank You!