Spring Framework 5.0 Themes and Trends

Sébastien Deleuze @sdeleuze

Brian Clozel @bclozel

Spring Framework 5

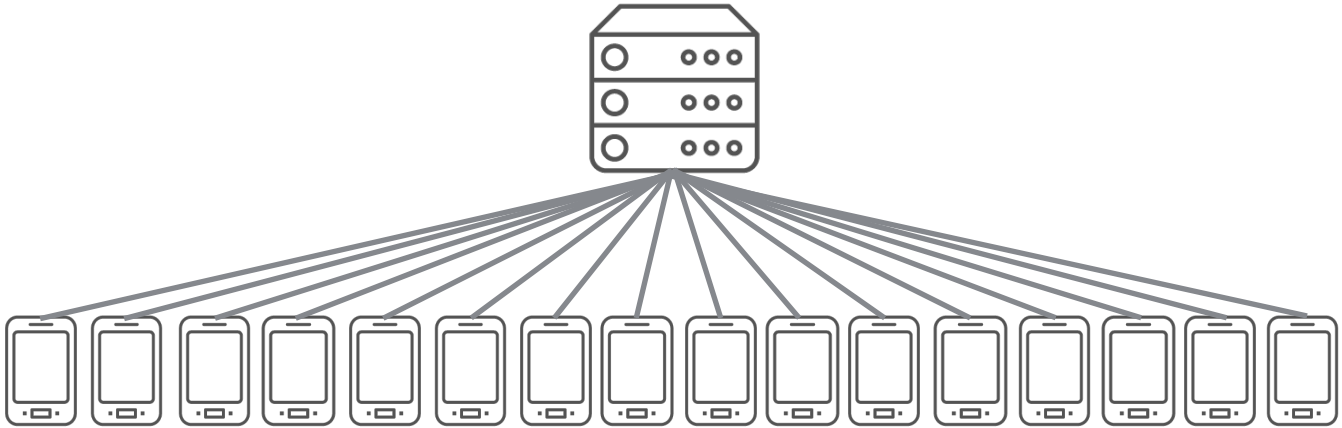
- **Reactive foundations**
- **Spring WebFlux**
- **Kotlin builtin support**
- Functional bean registration
- Baseline upgrade & cleanup
- Java 9 support
- Performance improvements

Reactive foundations

Why going Reactive?

More for **scalability** and
stability than for speed

What issue are we trying to solve?

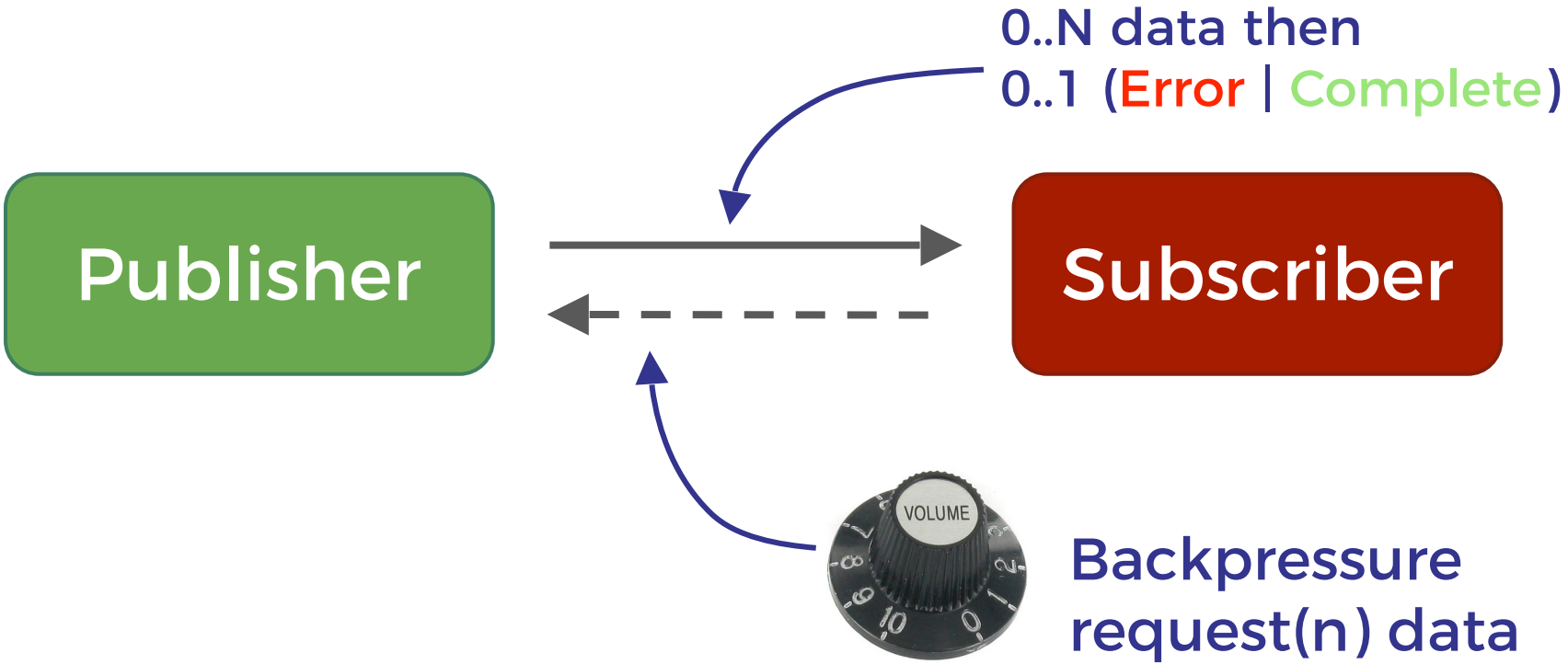


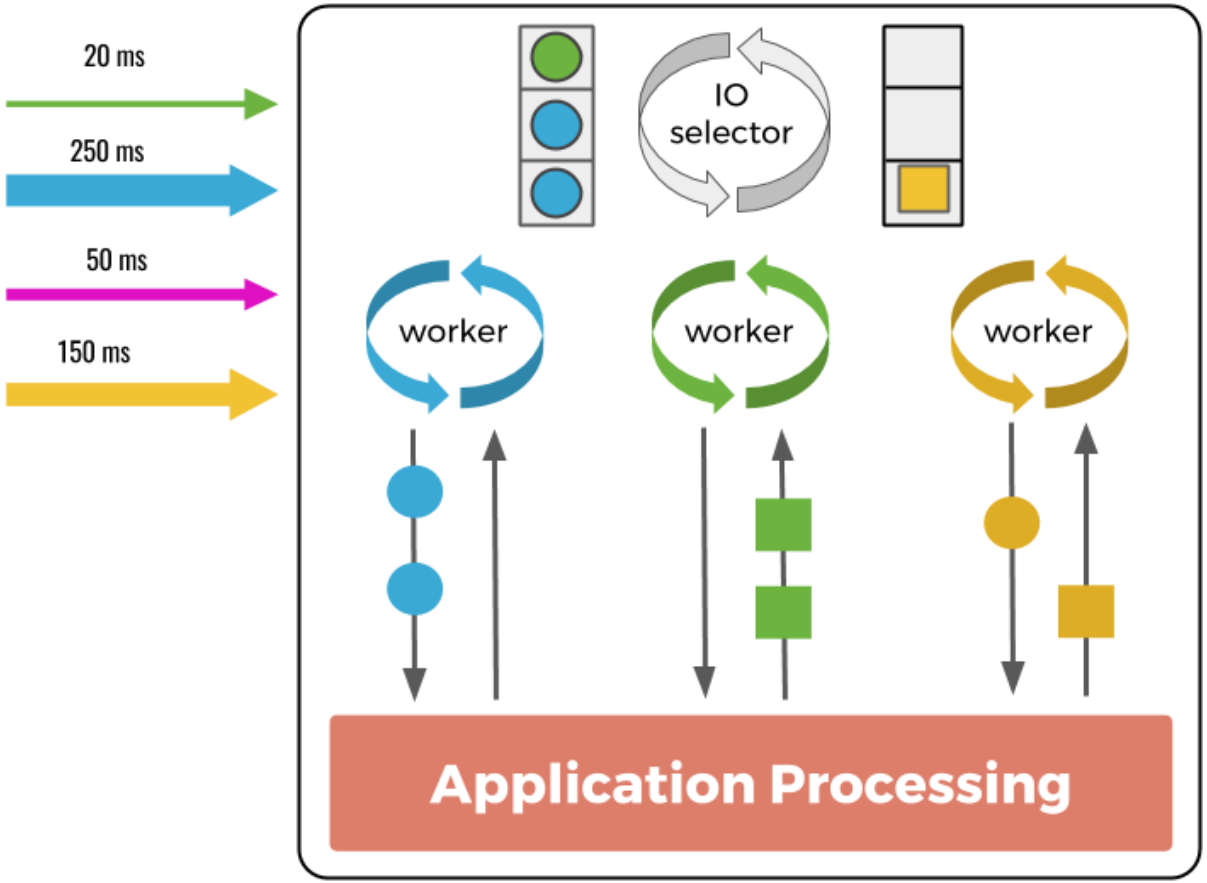
1 request = 1 thread

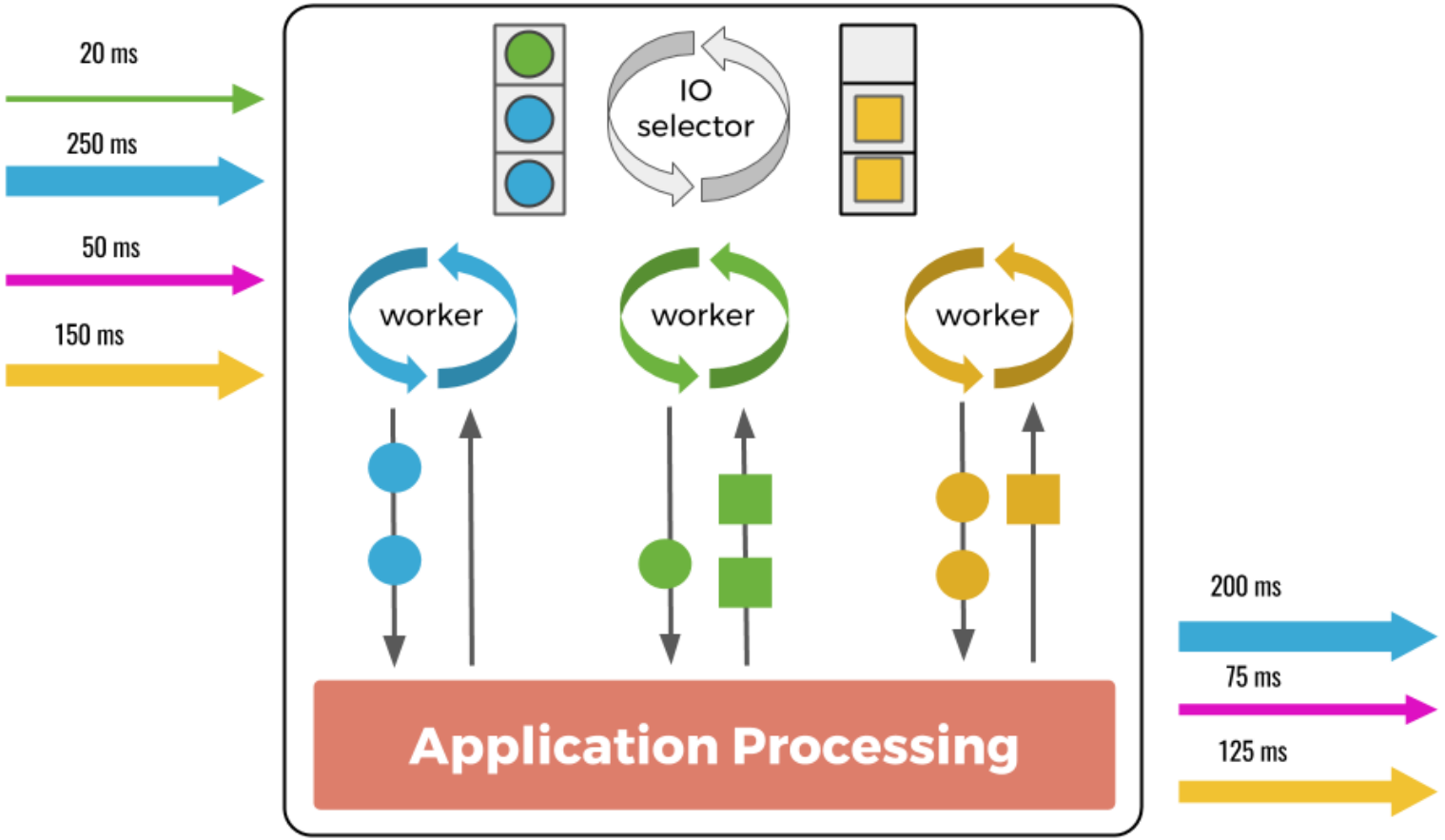
From blocking to event-based

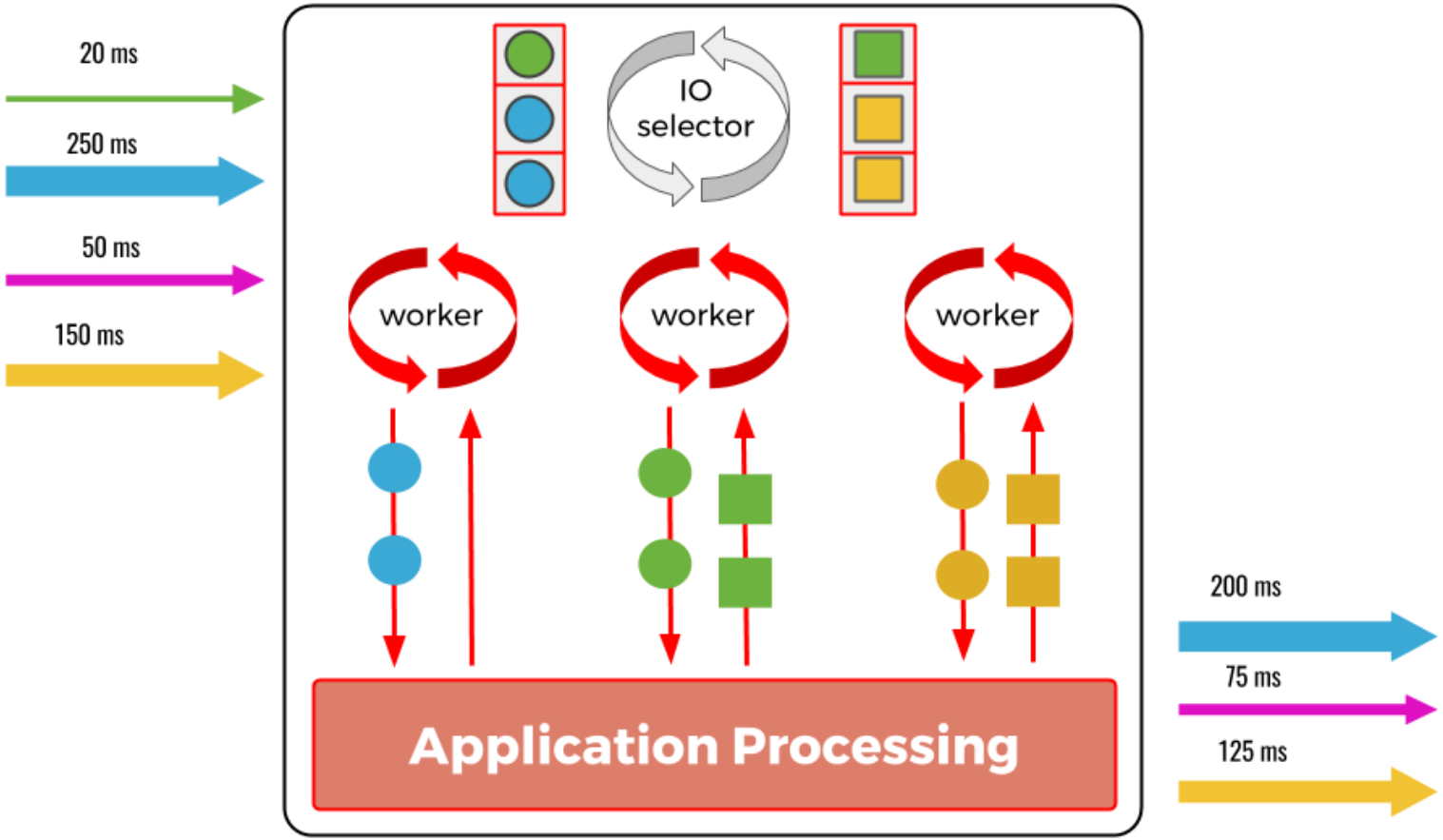
Neutral to latency

Reactive Streams









Reactive Streams is 4 interfaces (and a TCK)

```
public interface Publisher<T> {
  void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
  void onSubscribe(Subscription s);
  void onNext(T t);
  void onError(Throwable t);
  void onComplete();
}

public interface Subscription {
  void request(long n);
  void cancel();
}

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


Reactive libraries

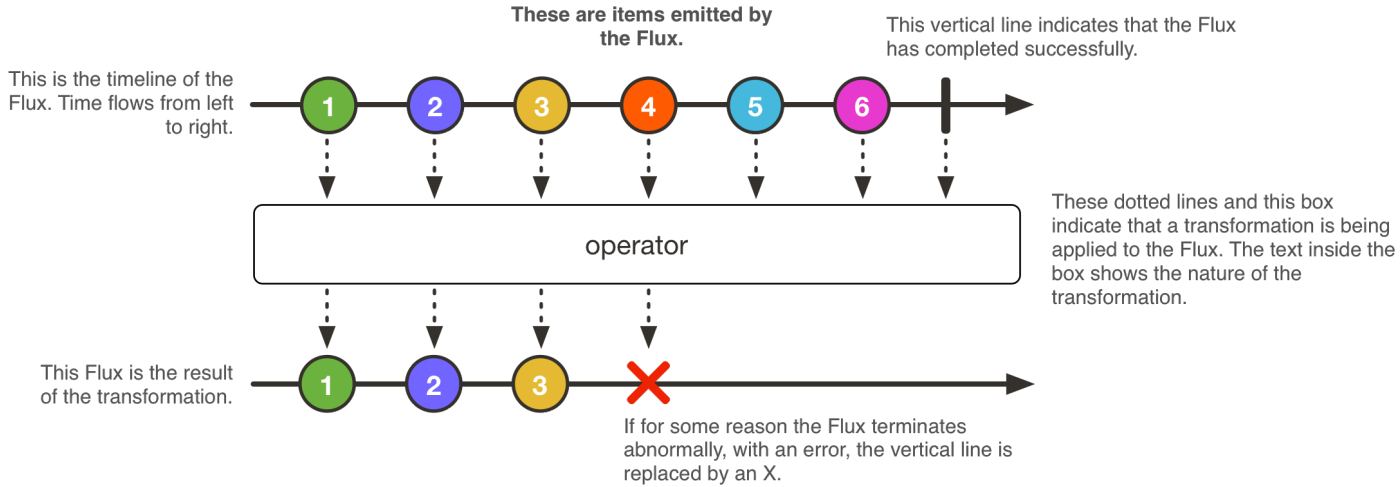
Your web application can use

 Reactor,  RxJava,  Akka Streams, etc.

Spring Framework 5 uses  Reactor for its Reactive foundations

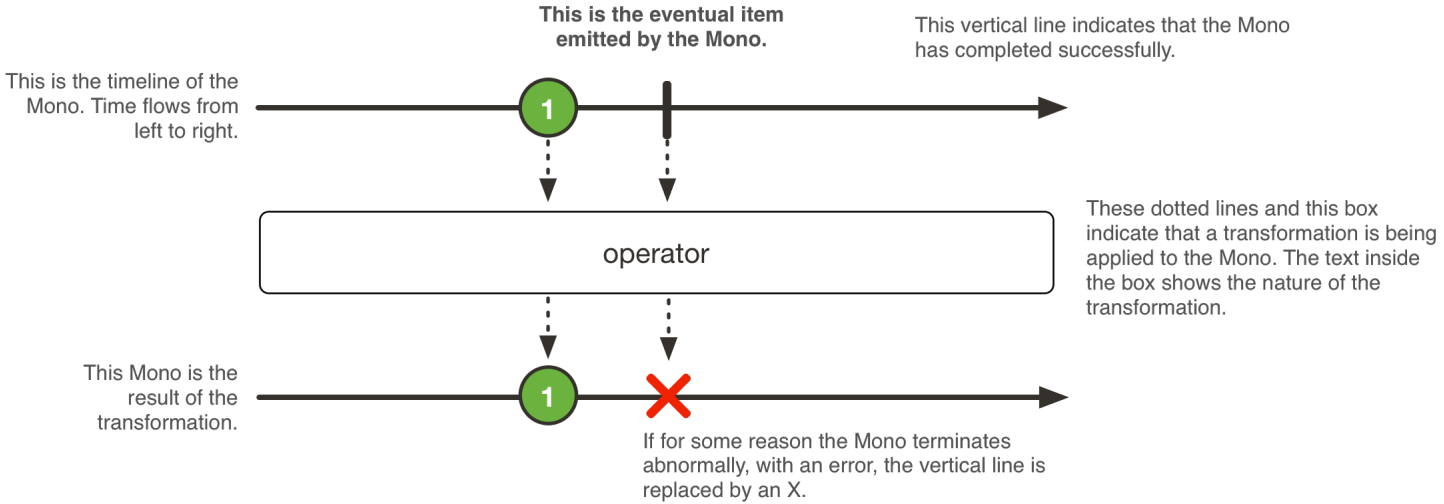
Flux<T>

- Implements Reactive Streams **`Publisher`**
- 0 to n elements
- Operators: `flux.map(…).zip(…).flatMap(…)`



Mono<T>

- Implements Reactive Streams **`Publisher`**
- 0 to 1 element
- Operators: `mono.then(…).otherwise(…)`



StepVerifier

- Designed to test easily Reactive Streams Publishers
- Carefully designed after writing thousands of Reactor and Spring WebFlux tests

```
StepVerifier.create(flux)
    .expectNext("foo", "bar")
    .verifyComplete();
```

Blocking versus Reactive API

Blocking API

```
interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

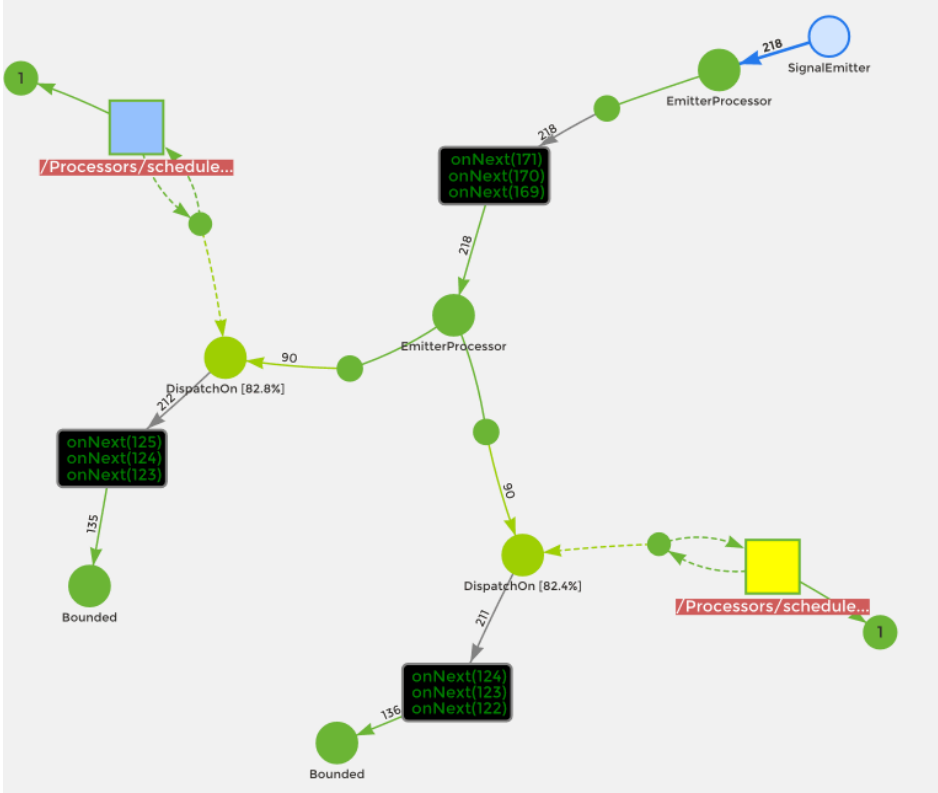
- Method returns when all the data has been received
- Exceptions thrown when an error occurs

Reactive API

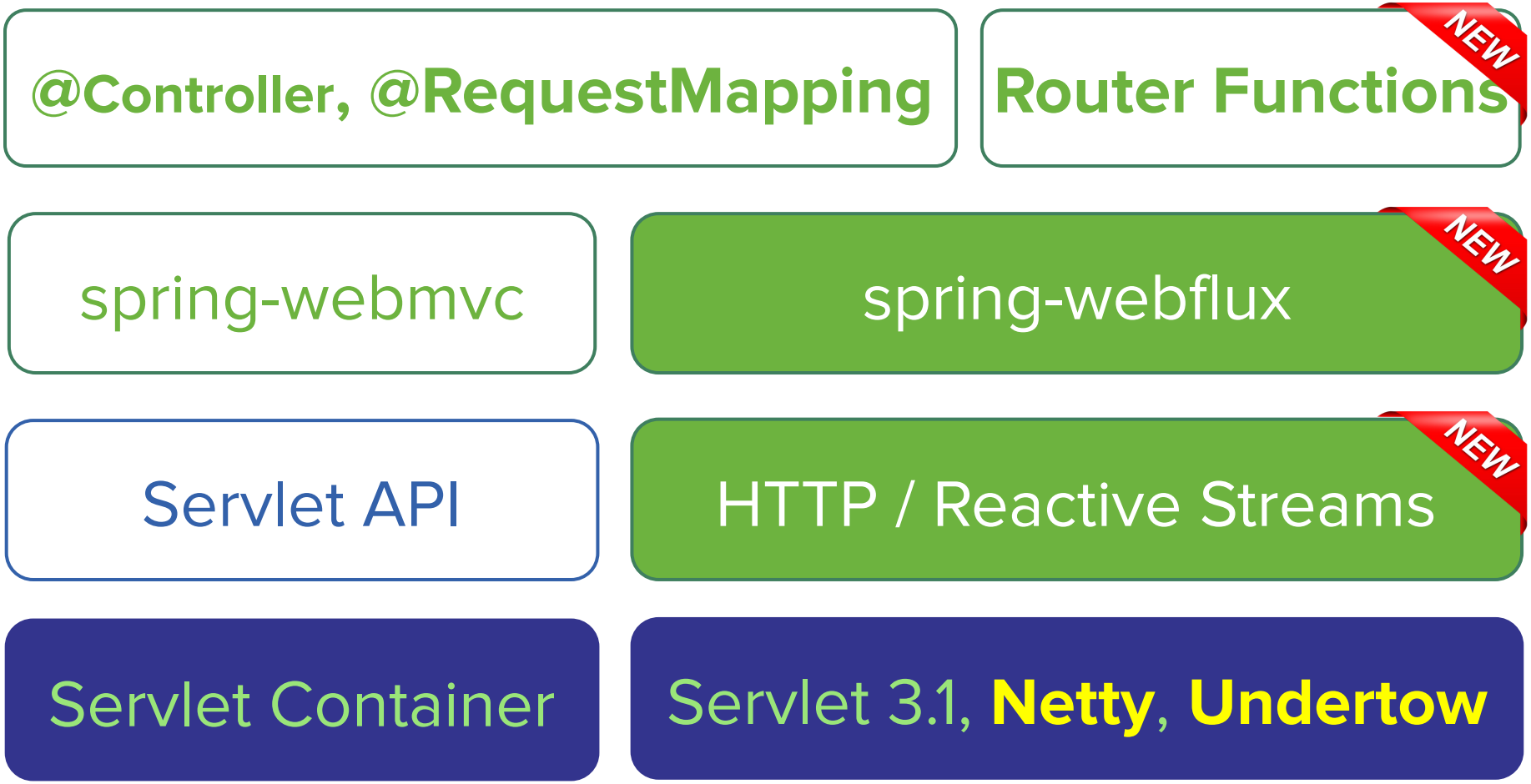
```
interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

- Method returns immediately
- Error event instead of throwing exception
- Complete event instead of method return
- Mono<Void> = error or success, no data
- Data comes as events too the Mono or Flux return value

Upcoming high-level features



Spring WebFlux


@Controller, @RequestMapping
Router Functions
NEW
spring-webmvc
spring-webflux
NEW
Servlet API
HTTP / Reactive Streams
NEW
Servlet Container
Servlet 3.1, Netty, Undertow

Spring MVC


```
@RestController
public class UserController {

    private final UserRepository repo;

    public UserController(UserRepository repo) { this.repo = repo; }

    @GetMapping("/user/{id}")
    public User findOne(@PathVariable String id) { return repo.findOne(id); }

    @GetMapping("/user")
    public List<User> findAll() { return repo.findAll(); }

    @PostMapping("/user")
    public void save(@RequestBody User user) { repo.save(user); }
}

interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

WebFlux annotation-based server


```
@RestController
public class ReactiveUserController {

    private final ReactiveUserRepository repo;

    public ReactiveUserController(ReactiveUserRepository repo) { this.repo = repo; }

    @GetMapping("/user/{id}")
    public Mono<User> findOne(@PathVariable String id) { return repo.findOne(id); }

    @GetMapping("/user")
    public Flux<User> findAll() { return repo.findAll(); }

    @PostMapping("/user")
    public Mono<Void> save(@RequestBody Mono<User> user) { repo.save(user); }
}

interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

Flux: array versus stream

```
@GetMapping("/user")
public Flux<User> findAll()
```

JSON

application/json

```
[
  {"a": "foo", "b": "bar"},
  {"a": "baz", "b": "boo"}
]
```

- Single Jackson invocation
- Doesn't support infinite streams

JSON Streaming

application/stream+json

```
{"a": "foo", "b": "bar"}
{"a": "baz", "b": "boo"}
…
```

Server-Sent Events

text/event-stream

```
data: {"a": "foo", "b": "bar"}
data: {"a": "baz", "b": "boo"}
…
```

- Jackson invocation per element
- Supports infinite streams

WebFlux functional API

- Reactive
- Client and server
- Lightweight
- Flexible
- Can be use with or without dependency injection!
- Simple functional building blocks

WebFlux functional server API

- **RouterFunction**

```
Mono<HandlerFunction<ServerResponse>>
route(ServerRequest request)
```

- **RequestPredicate**

```
boolean test(ServerRequest request)
```

- **HandlerFunction**

```
Mono<ServerResponse> handle(ServerRequest request)
```

- **HandlerFilterFunction**

```
Mono<ServerResponse> filter(ServerRequest request,
HandlerFunction<ServerResponse> next)
```

WebFlux functional server API


```
@Controller
public class UserController  {

  public RouterFunction<ServerResponse> route() {
    return RouterFunctions
        .route(GET("/users"), this::listUsers)
        .andRoute(POST("/users"), this::createUser);
  }

  Mono<ServerResponse> listUsers(ServerRequest request) { ... }
  Mono<ServerResponse> createUser(ServerRequest request) { ... }
}
```

RequestPredicates

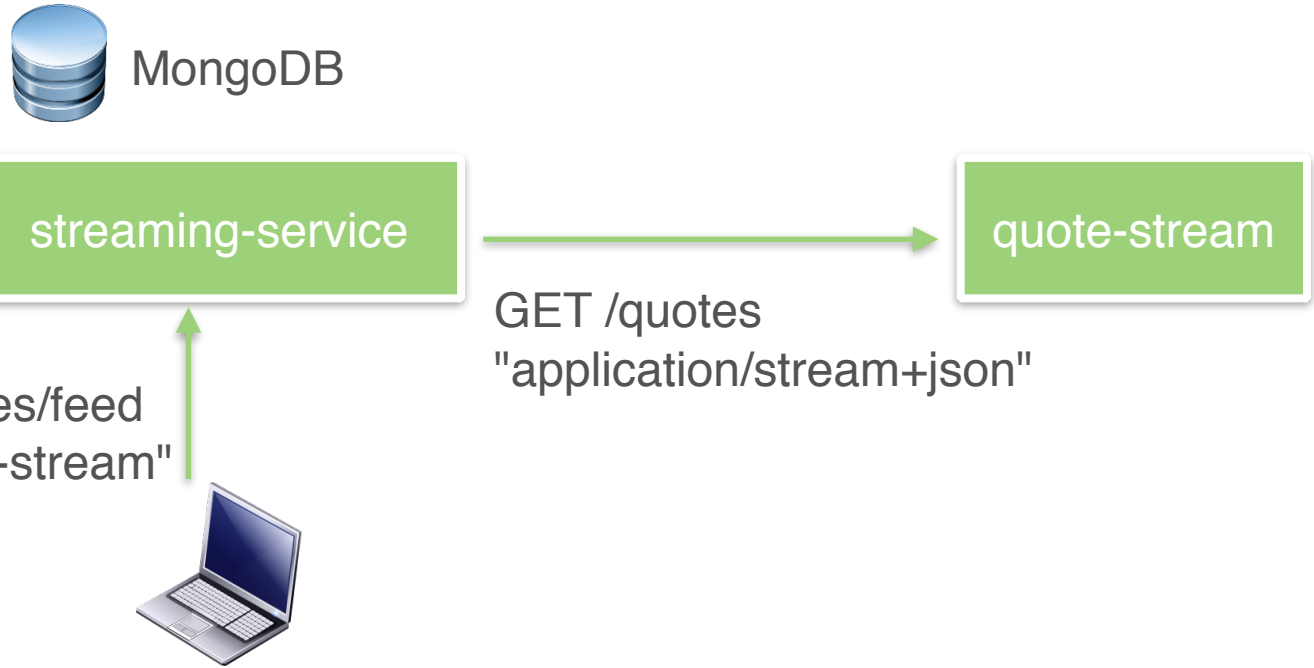
- HTTP method
- Path
- Path prefix
- Path extension
- Headers
- Content type
- Accept
- Query param

New WebClient API


```
WebClient client = WebClient.create();

Mono<GithubUser> githubUser = client
    .get()
    .uri("https://api.github.com/users/{username}", username)
    .exchange()
    .then(response -> response.bodyToMono(GithubUser.class));

Mono<TwitterUser> twitterUser = client
    .get()
    .uri("https://api.twitter.com/1.1/users/show.json?screen_name={username}", username)
    .exchange()
    .then(response -> response.bodyToMono(TwitterUser.class));

return githubUser.and(twitterUser, (github, twitter)-> new AppUser(github, twitter));
```

WebFlux demo

WebFlux Streaming Showcase



https://github.com/bclozel/webflux-streaming-showcase

Kotlin Support

Kotlin

- Overcome Java limitations
- Elegant and pragmatic language
- Concise code
- Simple and easy to learn
- Very good Java interoperability
- Embrace both functional and object oriented programming

How does it compare with ...

Same conciseness and expressive code,
but Kotlin default static typing and null-safety
make a big difference.

"Kotlin is a software engineering language in contrast to Scala which is a computing science language."
Eric Kolotyluk, Electronic Arts, Scala developer

Swift and Kotlin share similar syntax. Swift is coming from native world while Kotlin is coming from JVM world.

Domain model classes

Constructor + properties declaration

Generates equals() and hashcode()

```
data class User(
        val userName:  String,
        val firstName: String?,
        val lastName:  String?,
        val location:  Point = Point(0, 0)
)
```

String? = nullable
String = non-nullable

val = immutable
var = mutable

Optional parameter with default value

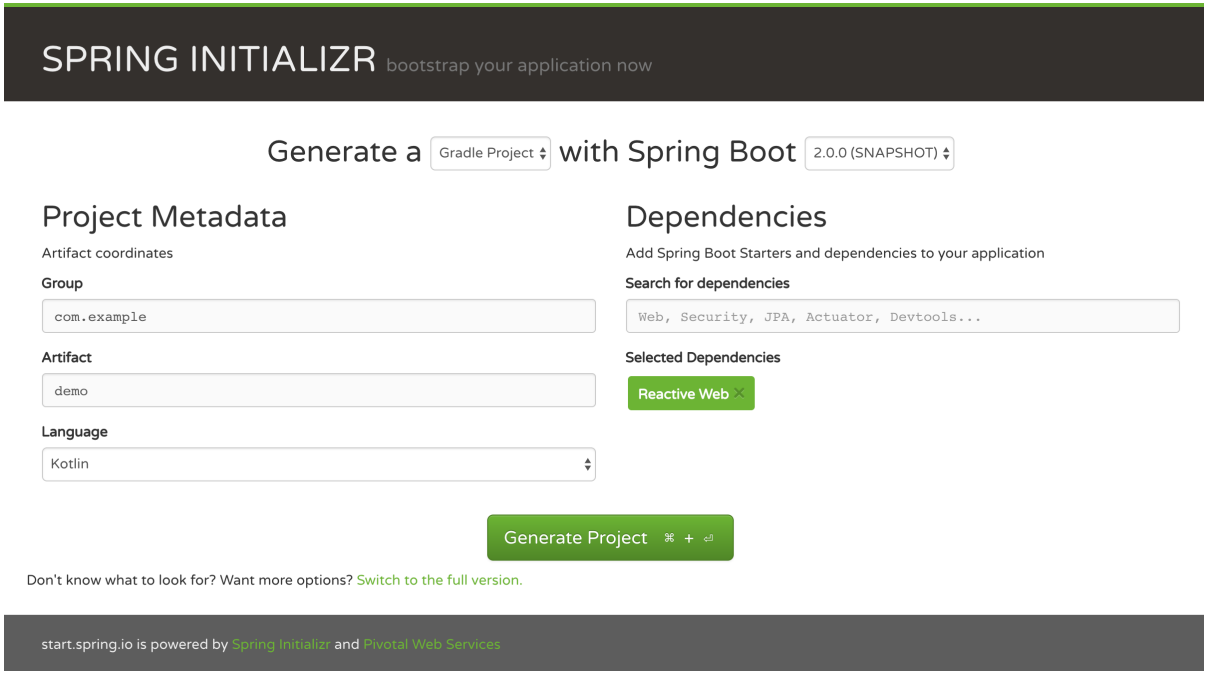
```
public class User {

    private String userName;
    private String firstName;
    private String lastName;
    private Point  location;

    public User(String userName, String firstName, String lastName) {
        this(userName,firstName, lastName, null);
    }

    public User(String userName, String firstName, String lastName, Point location) {
        Assert.notNull(userName);
        Assert.notNull(firstName);
        Assert.notNull(lastName);
        this.userName = userName;
        this.firstName = firstName;
        this.lastName = lastName;
        this.location = location;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getFirstName() {
        return firstName;
    }

    id setFirstName(String firstName) {
    firstName = firstName;


    ring getLastName() {
    n lastName;


    id setLastName(String lastName) {
    lastName = lastName;


    int getLocation() {
    n location;


    public void setLocation(Point location) {
        this.location = location;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        User j = (User) o;

        if (!userName.equals(j.userName)) {
            return false;
        }
        if (!firstName.equals(j.firstName)) {
            return false;
        }
        if (!lastName.equals(j.lastName)) {
            return false;
        }
        return location != null ? location.equals(j.location) : j.location == null;

    }

    @Override
    public int hashCode() {
        int result = userName.hashCode();
        result = 31 * result + firstName.hashCode();
        result = 31 * result + lastName.hashCode();
        result = 31 * result + (location != null ? location.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "User{" +
                "
                "
                "
                "
                "
                '}';
    }
}
```

Kotlin is much more than a "super Java" or than Lombok

My favorite Kotlin features

- Awesome type inference with static typing
- Extensions
- Reified type parameters
- Null safety
- Type aliases
- Allows to write nice DSL
- Smart casts

Spring Initializr



https://start.spring.io/#!language=kotlin

Introducing Kotlin support in Spring Framework 5

Introducing Kotlin support in Spring Framework 5.0

Following the Kotlin support on start.spring.io we introduced a few months ago, we have continued to work to ensure that Spring and Kotlin play well together. One of the key strengths ...

spring.io

RETWEETS 203 J'AIME 227

15:06 - 4 janv. 2017

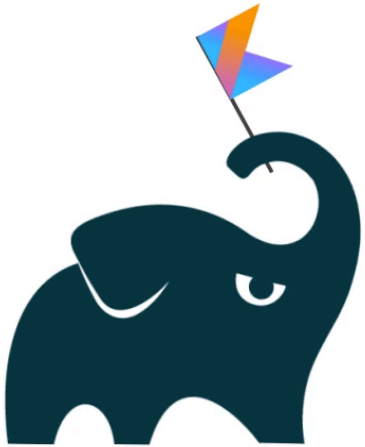
3 203 227

https://goo.gl/gMnhxN

Gradle build files written in Kotlin

Build_gradle 'dep { compile("org.jetbrains.kotlin:kotlin-stdlib-...'

```
43      maven { setUrl("https://repo.spring.io/snapshot") }
44  }
45
46  tasks.withType<KotlinCompile> {
47      kotlinOptions {
48          jvmTarget = "1.8"
49      }
50  }
51
52  configure<NodeExtension> {
53      version = "6.9.4"
54      download = true
55  }
56
57  configure<NoArgExtension> {
58      annotation("org.springframework.data.mongodb.core.mapping.Document")
59  }
60
61  val kotlinVersion: String by extra
62  val springDataVersion = "2.0.0.BUILD-SNAPSHOT"
63
64  dep {
65      dependencies (from getDependencies())          DependencyHandler!
66      depth (from getDepth())                        Int
67      dependencies(p0: Closure<(raw) Any!>!)         Unit
68      depthCompare(p0: Project!)                     Int
69      dependencies {...} (configuration: KotlinDependencyHandle…   Unit
70      deserializeFromPlainText(str: String) (org.jetbrains.kotlin…   T?
71      deserializeFromPlainText(str: String, klass: KClass<T>) (or…   T?
72      Did you know that Quick Documentation View (^J) works in completion lookups as well? >>  π
73
74      compile("com.samskivert:jmustache:1.13")
75      compile("com.atlassian.commonmark:commonmark:0.8.0")
76      compile("com.atlassian.commonmark:commonmark-ext-autolink:0.8.0")
77
```



kotlin-spring Gradle and Maven plugin

Also available as a kotlin-allopen generic plugin

Without kotlin-spring

```
@SpringBootApplication
open class Application {

  @Bean
  open fun foo() = ...

  @Bean
  open fun bar() = ...

}
```

With kotlin-spring

```
@SpringBootApplication
class Application {

  @Bean
  fun foo() = ...

  @Bean
  fun bar() = ...

}
```

kotlin-noarg Gradle and Maven plugin

Also available preconfigured for JPA as kotlin-jpa

```
configure<NoArgExtension> {
    annotation("org.springframework.data.mongodb.core.mapping.Document")
}
```

```
@Document
data class User(
  @Id val login: String,
  val firstname: String,
  val lastname: String,
  val email: String,
  val company: String? = null,
  val description: Map<Language, String> = emptyMap(),
  val logoUrl: String? = null,
  val role: Role = Role.ATTENDEE)
```

Persistence

```
class EventRepository(val template: ReactiveMongoTemplate) {
    fun findAll() = template.find<Event>(Query().with(Sort("year")))
    fun findOne(id: String) = template.findById<Event>(id)
    fun deleteAll() = template.remove<Event>(Query())
    fun save(event: Event) = template.save(event)
}
```

Extensions

```
// Spring Boot extensions
fun run(type: KClass<*>, vararg args: String) = SpringApplication.run(type.java, *args)

// Spring Data extensions
inline fun <reified T : Any> ReactiveMongoOperations.findById(id: Any): Mono<T> =
  findById(id, T::class.java)
inline fun <reified T : Any> ReactiveMongoOperations.find(query: Query): Flux<T> =
  find(query, T::class.java)
inline fun <reified T : Any> ReactiveMongoOperations.findAll(): Flux<T> =
  findAll(T::class.java)
inline fun <reified T : Any> ReactiveMongoOperations.findOne(query: Query): Mono<T> =
  find(query, T::class.java).next()
inline fun <reified T : Any> ReactiveMongoOperations.remove(query: Query): Mono<DeleteResult> =
  remove(query, T::class.java)

// Other extensions
fun ServerRequest.language() =
  Language.findByTag(this.headers().header(HttpHeaders.ACCEPT_LANGUAGE).first())
```

Spring MVC

```
@RestController
class UserController(val repo: UserRepository) {
    @GetMapping("/user/{id}")
    fun findOne(@PathVariable id: String) = repo.findOne(id)

    @GetMapping("/user")
    fun findAll() = repo.findAll()

    @PostMapping("/user")
    fun save(@RequestBody user: User) = repo.save(user)
}

interface UserRepository {
    fun findOne(id: String): User
    fun findAll(): List<User>
    fun save(user: User)
}
```

Classes and functions are public by default

Constructor injection without @Autowired if single constructor

Static typing + type inference

Spring WebFlux annotation-based

```
@RestController
class ReactiveUserController(val repository: ReactiveUserRepository) {
    @GetMapping("/user/{id}")
    fun findOne(@PathVariable id: String) = repository.findOne(id)

    @GetMapping("/user")
    fun findAll() = repository.findAll()

    @PostMapping("/user")
    fun save(@RequestBody user: Mono<User>) = repository.save(user)
}

interface ReactiveUserRepository {
    fun findOne(id: String): Mono<User>
    fun findAll(): Flux<User>
    fun save(user: Mono<User>): Mono<Void>
}
```

Spring WebFlux functional Kotlin DSL

Simple path request predicate

```
router {
  "/blog" { findAllView(it) }
  "/blog/{slug}" { findOneView(it) }
  "/api/blog" { findAll(it) }
  "/api/blog/{id}" { findOne(it) }
}
```

Spring WebFlux functional Kotlin DSL

HTTP method + path predicate

```
router {
  GET("/blog") { findAllView(it) }
  GET("/blog/{slug}") { findOneView(it) }
  GET("/api/blog") { findAll(it) }
  GET("/api/blog/{id}") { findOne(it) }
  POST("/api/blog") { create(it) }
}
```

Spring WebFlux functional Kotlin DSL

Method references

```
router {
  GET("/blog", blogController::findAllView)
  GET("/blog/{slug}", blogController::findOneView)
  GET("/api/blog", blogController::findAll)
  GET("/api/blog/{id}", blogController::findOne)
  POST("/api/blog", blogController::create)
}
```

Spring WebFlux functional Kotlin DSL

Nested routing

```
router {
 "/blog".nest {
    GET("/", blogController::findAllView)
    GET("/{slug}", blogController::findOneView)
  }
  "/api/blog".nest {
    GET("/", blogController::findAll)
    GET("/{id}", blogController::findOne)
    POST("/", blogController::create)
  }
}
```

Spring WebFlux functional Kotlin DSL

Rich set of RequestPredicate

```
router {
 ("/blog" and accept(TEXT_HTML)).nest {
    GET("/", blogController::findAllView)
    GET("/{slug}", blogController::findOneView)
  }
  ("/api/blog" and accept(APPLICATION_JSON)).nest {
    GET("/", blogController::findAll)
    GET("/{id}", blogController::findOne)
    POST("/", blogController::create)
  }
}
```

Leveraging Kotlin nullable information

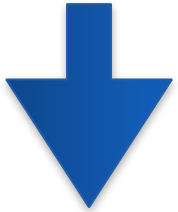
```
// "GET /foo" and "GET /foo?bar=baz" are allowed
@GetMapping("/foo")
fun foo(@RequestParam bar: String?) = ...


// "GET /foo?bar=baz" is allowed and "GET /foo" will return an error
@GetMapping("/foo")
fun foo(@RequestParam bar: String) = ...
```

Spring Kotlin extensions


```
Mono<ServerResponse> findAll(ServerRequest request) {
  return ServerResponse.ok().body(repository.findAll(), User.class);
}
```

ServerResponseExtensions provided in spring-webflux.jar

```
inline fun <reified T : Any> ServerResponse.BodyBuilder.body(publisher: Publisher<T>) =
    body(publisher, T::class.java)
```



No need to specify explicitly types thanks to Kotlin reified type parameters

```
fun findAll(req: ServerRequest) = ok().body(repository.findAll())
```

Spring Framework extensions

- ApplicationContext
- Spring WebFlux
  - Client
  - Server
- Spring MVC
- RestTemplate
- JDBC
- Reactor (via reactor-kotlin)

Kotlin type-safe templates

- Regular Kotlin code, no new dialect to learn
- Extensible, refactoring and auto-complete support
- Need to cache compiled scripts for good performances

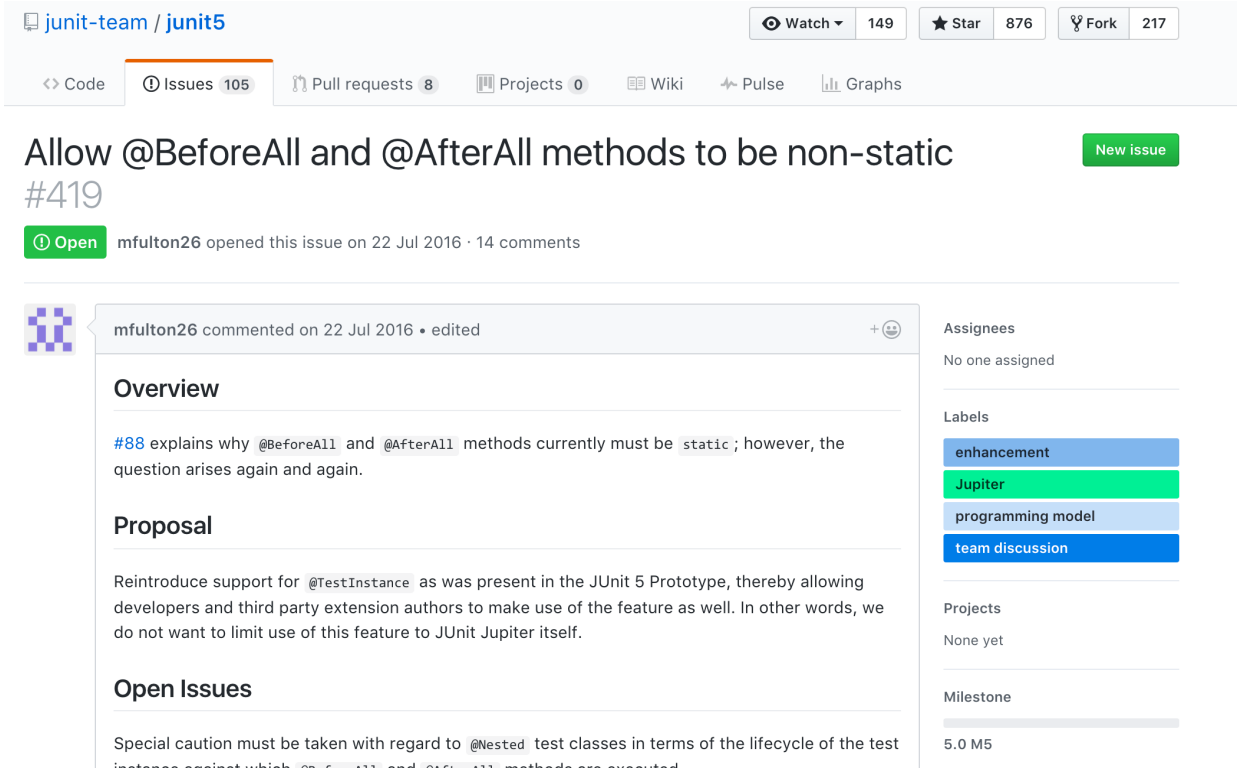
```
import io.spring.demo.*
"""
${include("header")}
<h1>${i18n("title")}</h1>
<ul>
    ${users.joinToLine{ "<li>${i18n("user")} ${it.firstname} ${it.lastname}</li>" }}
</ul>
${include("footer")}
"""
```

See https://github.com/sdeleuze/kotlin-script-templating for more details

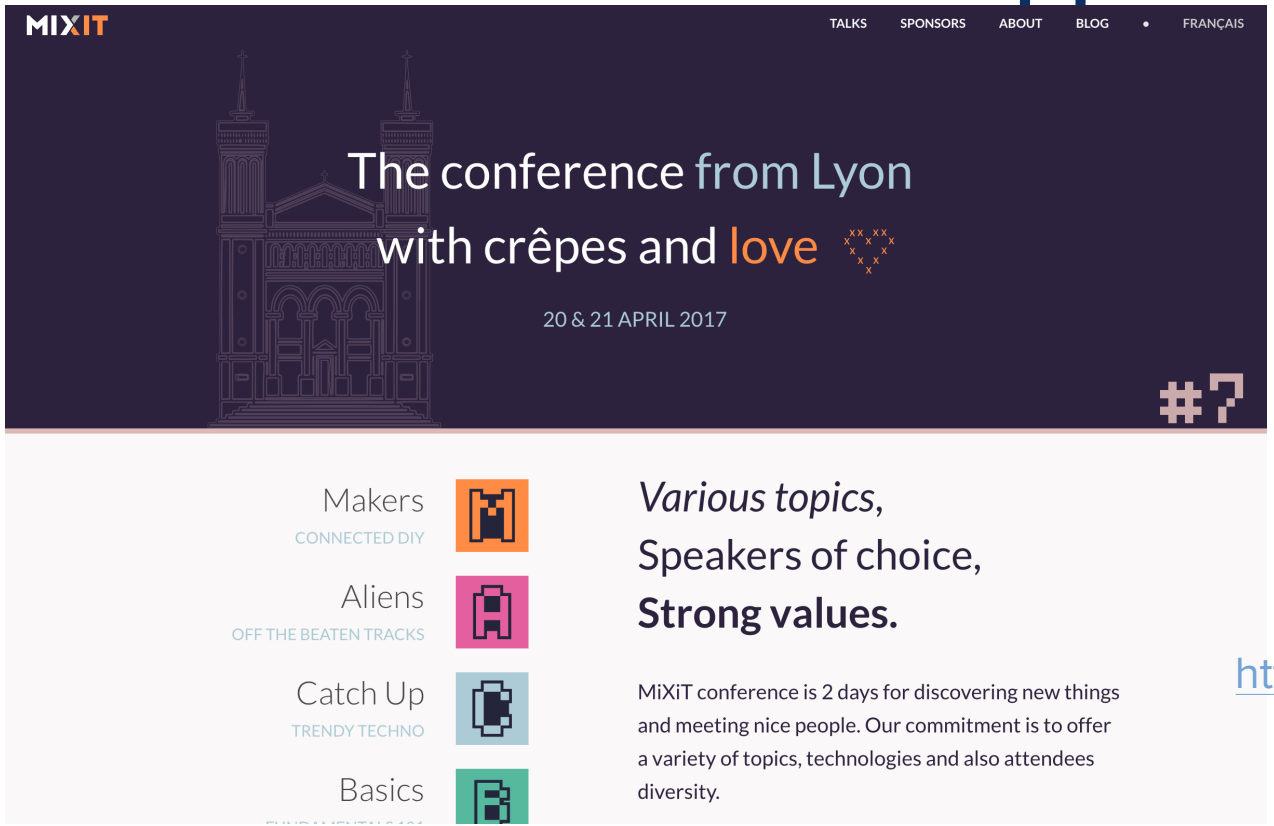
Tests with JUnit

```
class UserIntegrationTests : AbstractIntegrationTests() {
  @Test
  fun `Find Dan North`() {
    val speaker = client.get()
      .uri("/api/speaker/tastapod")
      .accept(APPLICATION_JSON)
      .exchange().then { r -> r.bodyToMono<User>() }

    StepVerifier.create(speaker)
      .consumeNextWith {
        assertEquals("North", it.lastname)
        assertTrue(it.role == Role.SPEAKER)
      }
      .verifyComplete()
  }
}
```

JUnit 5 will improve Kotlin support



MiXiT reference web application



https://github.com/mixitconf/mixit

Functional bean registration

Functional bean registration API

- After XML and JavaConfig, a third major way to register your beans
- Lambda with Supplier act as a FactoryBean
- Very efficient, no reflection, no CGLIB proxies involved
- Usable with pure Spring Framework apps, not yet with Spring Boot

API example


```
GenericApplicationContext context = new GenericApplicationContext();
        context.registerBean(A.class);
        context.registerBean(B.class, () -> new B(context.getBean(A.class)));
```

```
GenericApplicationContext {
    registerBean<A>()
    registerBean { B(it.getBean<A>()) }}
```

Choose the flavour you prefer!

Boot + WebFlux

- Automatic server configuration
- All Spring Boot goodness!
- Bean registration
  - Annotation-based
  - Optimized Classpath scanning
- Reference webapp startup:
  - 3 seconds
  - 20 Mbytes heap size after GC

WebFlux + functional bean registration

- Manual server configuration
- Bean registration
  - Lambda-based
  - No Cglib proxy
  - No need for kotlin-spring plugin
- Reference webapp startup:
  - 1.2 seconds
  - 10 Mbytes heap size after GC

See functional-bean-registration MiXiT branch

```
AnnotationConfigApplicationContext {
  registerBean { ReactiveMongoTemplate(SimpleReactiveMongoDatabaseFactory(
          ConnectionString(it.environment.getProperty("mongo.uri"))))
  }
  registerBean<MarkdownConverter>()
  registerBean<UserRepository>()
  registerBean<EventRepository>()
  registerBean<TalkRepository>()
  registerBean<PostRepository>()
  // ...

}
```

https://goo.gl/iGwzw3

Other topics

Baseline upgrade & cleanups

- Java 8 baseline
- Java 9 support
- Servlet 3.1+, JMS 2.0+, JPA 2.1+, JUnit 5
- Removal of outdated technologies
  - Portlet
  - Velocity

Using Jigsaw with Spring

- **Spring Framework 5 jars are Jigsaw-compliant modules out of the box**
  - Defined as “automatic modules”
  - Required to be able to access your code (classpath scanning, etc)
- **Naming convention based on Maven metadata:**
  - spring-context-5.0.0.RELEASE.jar ==> spring.context
- **Example**

```
module my.app.db {
    requires spring.jdbc;
}
```

Performance improvements

- Functional bean registration is very fast
  - No reflection
  - No proxy
- Alternative to component scanning
  - Pre-computed index at compilation time
  - Extensible using @Indexed
- ASM meta-data cache
- Rewrite of `AntPathMatcher`
- Zero-copy transfer of `org.springframework.core.io.Resource`

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {
```

Roadmap

Spring Framework 5.0 roadmap

- M5 release available
- RC1 April 2017
- GA June 2017

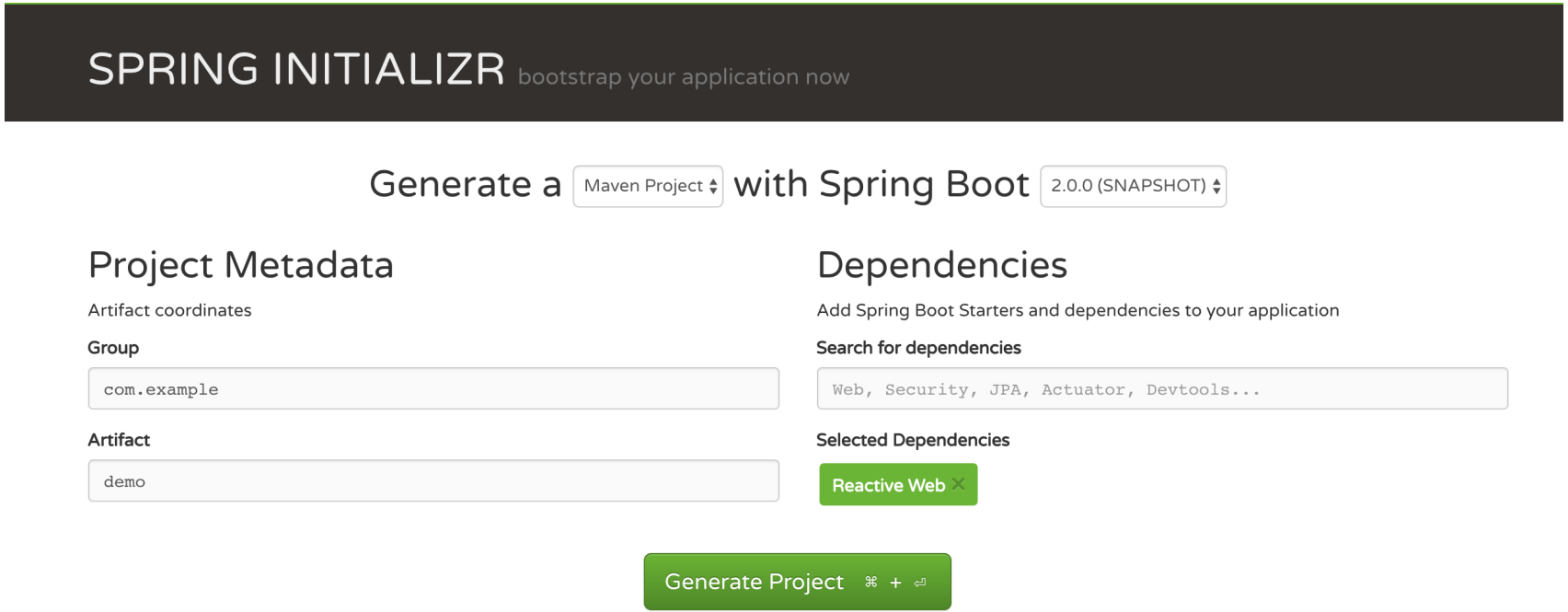
Spring Boot 2.0 (expected November 2017)

- Leverage Spring Framework 5.0
- Spring WebFlux support
- Spring WebFlux Actuators
- Reactive support (data, security...)
- Improved Kotlin support
- ...

Reactive Spring ongoing effort

- Reactor Core, Netty, Adapter, Test are GA
- Reactor Kafka Milestone available
- Reactive support currently introduced in the whole Spring portfolio:
  - Spring Boot
  - Spring Data
  - Spring Security
  - Spring Cloud Stream
  - etc.

Try it: start.spring.io



Thanks


@sdeleuze


@bclozel