




Pivotal®

Why Spring ❤️ Kotlin

Sébastien Deleuze

 @sdeleuze

Current most popular way to build web applications



**Spring
Boot**



Let's see why and how far we can go with ...



**Spring
Boot**



Kotlin

Introducing Kotlin support in Spring Framework 5.0

🌐 À l'origine en anglais



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 of Kotlin is...

spring.io

RETWEETS

214

J'AIME

231

15:06 - 4 janv. 2017



Migrating a typical Boot application to Kotlin

<https://github.com/sdeleuze/spring-kotlin-deepdive>

SPRING BLOG

Creating a blog using
Spring Boot 2.0, Spring
WebFlux functional API
and Kotlin



RECENT POSTS

Reactor Bismuth is out

By Simon, on September 28th 2017

It is my great pleasure to announce the GA release of **Reactor Bismuth**, which notably encompasses `reactor-core 3.1.0.RELEASE` and `reactor-netty 0.7.0.RELEASE` \uD83C\uDF89

Introducing Kotlin support in Spring Framework 5.0

By Sebastien, on January 4th 2017

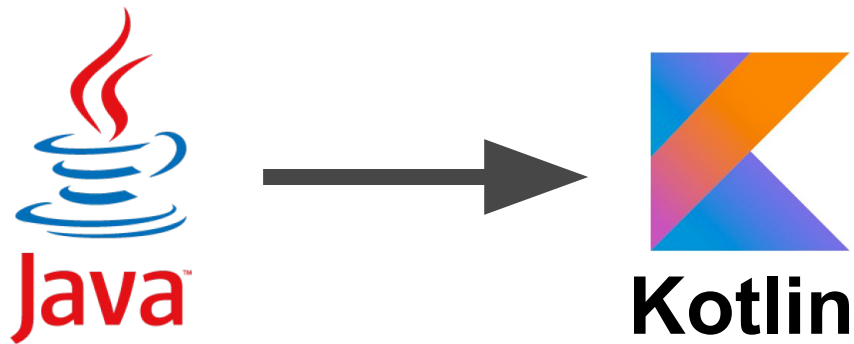
Following the [Kotlin support on start.spring.io](#) we introduced a few months ago, we have continued to work to ensure that Spring and [Kotlin](#) play well together.

Spring Framework 5.0 goes GA

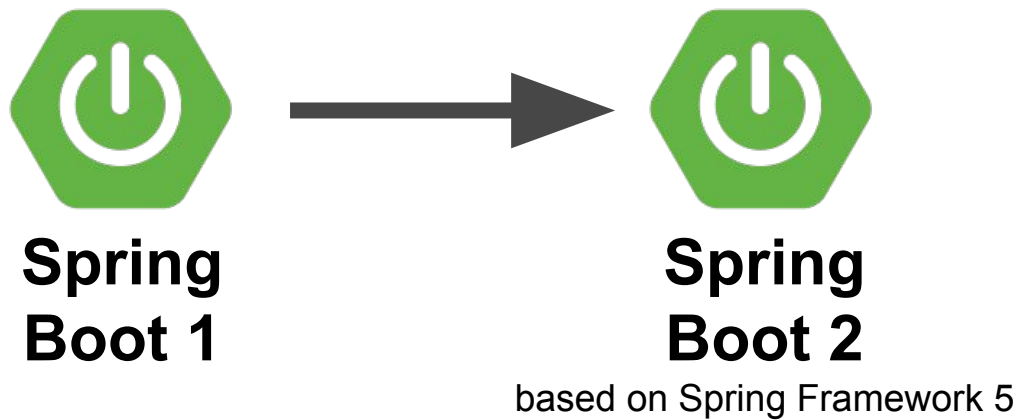
By Juergen, on September 28th 2017

Dear Spring community, It is my pleasure to announce that, after more than a year of milestones and RCs and almost two years of development overall, Spring Framework 5.0 is finally generally available as `5.0.0.RELEASE` from [repo.spring.io](#) and Maven Central!

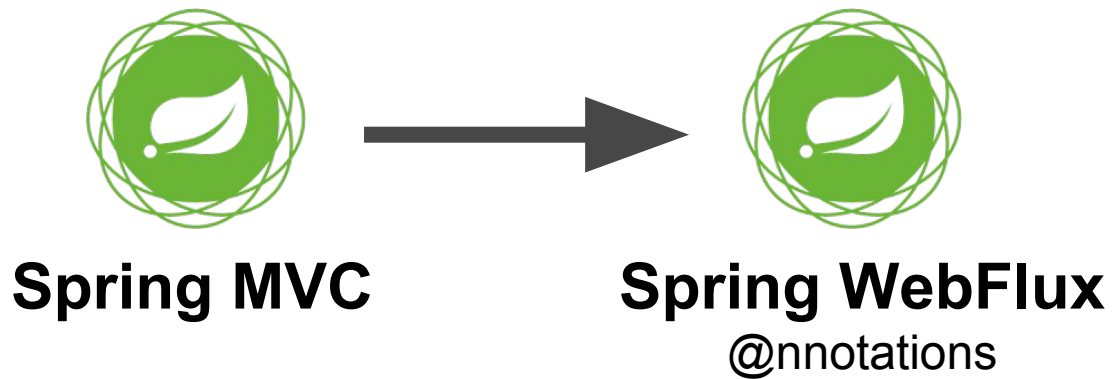
Step 1



Step 2



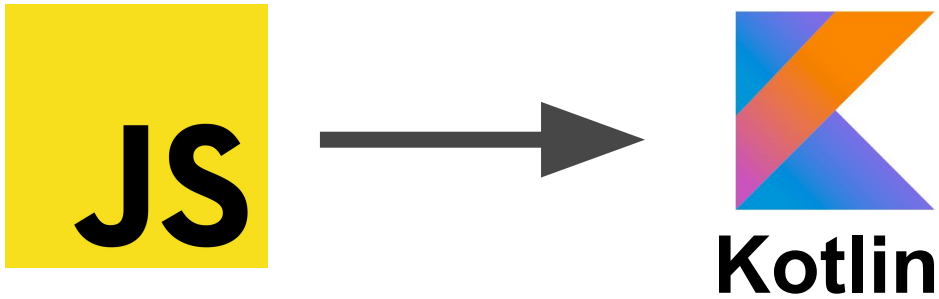
Step 3



Step 4



Step 5



Step 1 Kotlin

Step 2 Boot 2

Step 3 WebFlux @annotations

Step 4 Functional & DSL

Step 5 Kotlin for frontend

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

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.4

- Java
- Java
- Kotlin
- Groovy

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Generate Project alt + ⌘

Don't know what to look for? Want more options? [Switch to the full version.](#)

kotlin-spring compiler plugin

Automatically open Spring annotated classes and methods

Without kotlin-spring plugin

```
@SpringBootApplication
open class Application {

    @Bean
    open fun foo() = Foo()

    @Bean
    open fun bar(foo: Foo) = Bar(foo)

}
```

With kotlin-spring plugin

```
@SpringBootApplication
class Application {

    @Bean
    fun foo() = Foo()

    @Bean
    fun bar(foo: Foo) = Bar(foo)

}
```

Domain model

```
@Document
data class Post(
    @Id val slug: String,
    val title: String,
    val headline: String,
    val content: String,
    @DBRef val author: User,
    val addedAt: LocalDateTime = now())
```

```
@Document
data class User(
    @Id val login: String,
    val firstname: String,
    val lastname: String,
    val description: String? = null)
```

```
@Document
public class Post {

    @Id
    private String slug;

    private String title;

    private LocalDateTime addedAt;

    private String headline;

    private String content;

    @DBRef
    private User author;

    public Post() {
    }

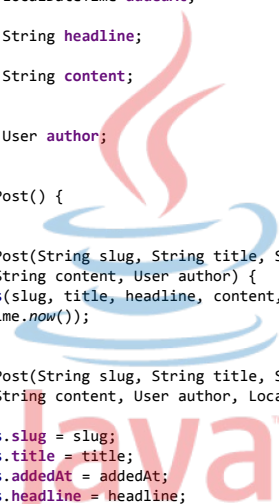
    public Post(String slug, String title, String
headline, String content, User author) {
        this(slug, title, headline, content, author,
LocalDateTime.now());
    }

    public Post(String slug, String title, String
headline, String content, User author, LocalDateTime
addedAt) {
        this.slug = slug;
        this.title = title;
        this.addedAt = addedAt;
        this.headline = headline;
        this.content = content;
        this.author = author;
    }

    public String getSlug() {
        return slug;
    }

    public void setSlug(String slug) {
        this.slug = slug;
    }

    public String getTitle() {
        return title;
    }
}
```



Spring MVC controller written in Java

```
@RestController
public class UserController {

    private final UserRepository userRepository;

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

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

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

    @PostMapping("/user")
    public User save(@RequestBody User user) {
        return userRepository.save(user);
    }
}
```



Spring MVC controller written in Kotlin

```
@RestController
class UserController(private 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)
}
```



Inferred type hints in IDEA

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

    @GetMapping("/user")
    fun findAll() : List<User> = repo.findAll()

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

Settings

Editor

General

Appearance

Show parameter name hints

Select Kotlin

Check “Show function/property/local value return type hints”

Expressive test names with backticks

```
class EmojTests {  
  
    @Test  
    fun `Why Spring ❤️ Kotlin?`() {  
        println("Because I can use emoji in function names \uD83D\uDE09")  
    }  
  
}
```

> Because I can use emoji in function names 😊

Step 1 Kotlin

Step 2 Boot 2

Step 3 WebFlux @annotations

Step 4 Functional & DSL

Step 5 Kotlin for frontend

Spring ❤️ Kotlin

and officially supports it



Spring Framework 5



Reactor Core 3.1

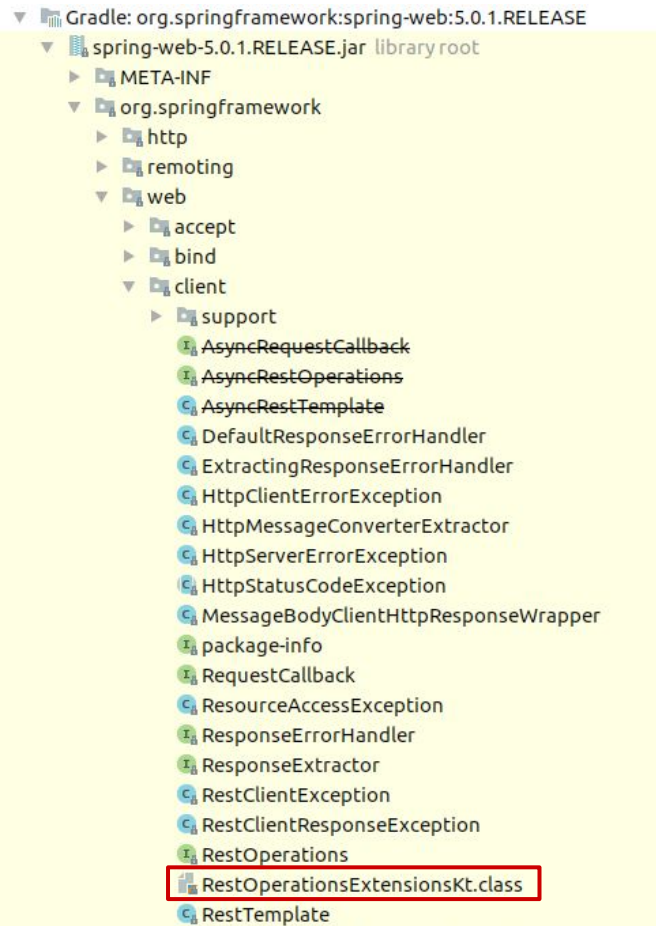


Spring Data Key



Spring Boot 2 (late 2017)

Kotlin bytecode builtin in Spring JARs



Kotlin support reference documentation

<https://goo.gl/uwyjQn>

Table of Contents

[← Back to index](#)

1. Kotlin

- 1.1. Requirements
- 1.2. Extensions
- 1.3. Null-safety
- 1.4. Classes & Interfaces
- 1.5. Annotations
- 1.6. Bean definition DSL
- 1.7. Web
- 1.8. Spring projects in Kotlin
- 1.9. Getting started
- 1.10. Resources
- 2. Apache Groovy
- 3. Dynamic Language Support

Language Support

Version 5.0.1.RELEASE

1. Kotlin

[Kotlin](#) is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing very good [interoperability](#) with existing libraries written in Java.

The Spring Framework provides first-class support for Kotlin that allows developers to write Kotlin applications almost as if the Spring Framework was a native Kotlin framework.

1.1. Requirements

Spring Framework supports Kotlin 1.1+ and requires [kotlin-stdlib](#) (or one of its [kotlin-stdlib-jre7](#) / [kotlin-stdlib-jre8](#) variants) and [kotlin-reflect](#) to be present on the classpath. They are provided by default if one bootstraps a Kotlin project on [start.spring.io](#).

1.2. Extensions

Kotlin [extensions](#) provide the ability to extend existing classes with additional functionality. The Spring Framework Kotlin APIs make use of these extensions to add new Kotlin specific conveniences to existing Spring APIs.

[Spring Framework KDoc API](#) lists and documents all the Kotlin extensions and DSLs available.



Keep in mind that Kotlin extensions need to be imported to be used. This means for example that the `GenericApplicationContext.registerBean` Kotlin extension will only be available if `import org.springframework.context.support.registerBean` is imported. That said, similar to static imports, an IDE should automatically suggest the import in most cases.

For example, [Kotlin reified type parameters](#) provide a workaround for JVM [generics type erasure](#), and Spring Framework provides some extensions to take advantage of this feature. This allows for a better Kotlin API `RestTemplate`, the new `WebClient` from Spring WebFlux and for various other APIs.

Kotlin API documentation

<https://goo.gl/svCLL1>

[spring-framework](#) / [org.springframework.web.reactive.function.server](#)

Package `org.springframework.web.reactive.function.server`

Types

`RouterFunctionDsl`

open class `RouterFunctionDsl`

Provide a `RouterFunction` Kotlin DSL in order to be able to write Idiomatic Kotlin code.

Functions

`body`

`fun <T : Any> BodyBuilder.body(publisher: Publisher<T>): Mono<ServerResponse>`

Extension for `ServerResponse.BodyBuilder.body` providing a `body(Publisher<T>)` variant.

`bodyToFlux`

`fun <T : Any> ServerRequest.bodyToFlux(): Flux<T>`

Extension for `ServerRequest.bodyToFlux` providing a `bodyToFlux<Foo>()` variant leveraging Kotlin reified type parameters.

`bodyToMono`

`fun <T : Any> ServerRequest.bodyToMono(): Mono<T>`

Extension for `ServerRequest.bodyToMono` providing a `bodyToMono<Foo>()` variant leveraging Kotlin reified type parameters.

`bodyToServerSentEvents`

`fun <T : Any> BodyBuilder.bodyToServerSentEvents(publisher: Publisher<T>): Mono<ServerResponse>`

Extension for `ServerResponse.BodyBuilder.body` providing a `bodyToServerSentEvents(Publisher<T>)` variant.

`router`

`fun router(routes: RouterFunctionDsl.() -> Unit): RouterFunction<ServerResponse>`

Allow to create easily a `RouterFunction<ServerResponse>` from a Kotlin router DSL based on the same building blocks than the Java one

Running Spring Boot 1 application with Kotlin

```
@SpringBootApplication
class Application

fun main(args: Array<String>) {
    SpringApplication.run(Application::class.java, *args)
}
```


Running Spring Boot 2 application with Kotlin

```
@SpringBootApplication
class Application

fun main(args: Array<String>) {
    runApplication<FooApplication>(*args)
}
```

Declaring additional beans

```
@SpringBootApplication
class Application {

    @Bean
    fun foo() = Foo()

    @Bean
    fun bar(foo: Foo) = Bar(foo)

}

fun main(args: Array<String>) {
    runApplication<FooApplication>(*args)
}
```

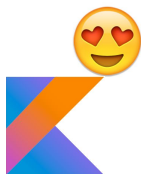
Array-like Kotlin extension for Model



```
operator fun Model.set(attributeName: String, attributeValue: Any) {  
    this.addAttribute(attributeName, attributeValue)  
}
```



```
@GetMapping("/")  
public String blog(Model model) {  
    model.addAttribute("title", "Blog");  
    model.addAttribute("posts", postRepository.findAll());  
    return "blog";  
}
```



```
@GetMapping("/")  
fun blog(model: Model): String {  
    model["title"] = "Blog"  
    model["posts"] = repository.findAll()  
    return "blog"  
}
```

Reified type parameters Kotlin extension

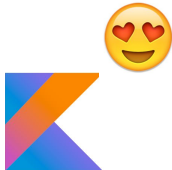
Goodbye type erasure, we are not going to miss you at all!



```
inline fun <reified T: Any> RestOperations.getForObject(url: URI): T? =  
    getForObject(url, T::class.java)
```



```
List<Post> posts = restTemplate.exchange(  
    "/api/post/", HttpMethod.GET, null,  
    new ParameterizedTypeReference<List<Post>>(){}).getBody();
```



```
val posts = restTemplate.getForObject<List<Post>>("/api/post/")
```

Leveraging Kotlin nullable information

To determine `@RequestParam` or `@Autowired` required attribute

```
@Controller // foo is mandatory, bar is optional
class FooController(val foo: Foo, val bar: Bar?) {

    @GetMapping("/") // Equivalent to @RequestParam(required=false)
    fun foo(@RequestParam baz: String?) = ...

}
```

Null safety of Spring APIs

By default, Kotlin consider Java types as platform types (unknown nullability)



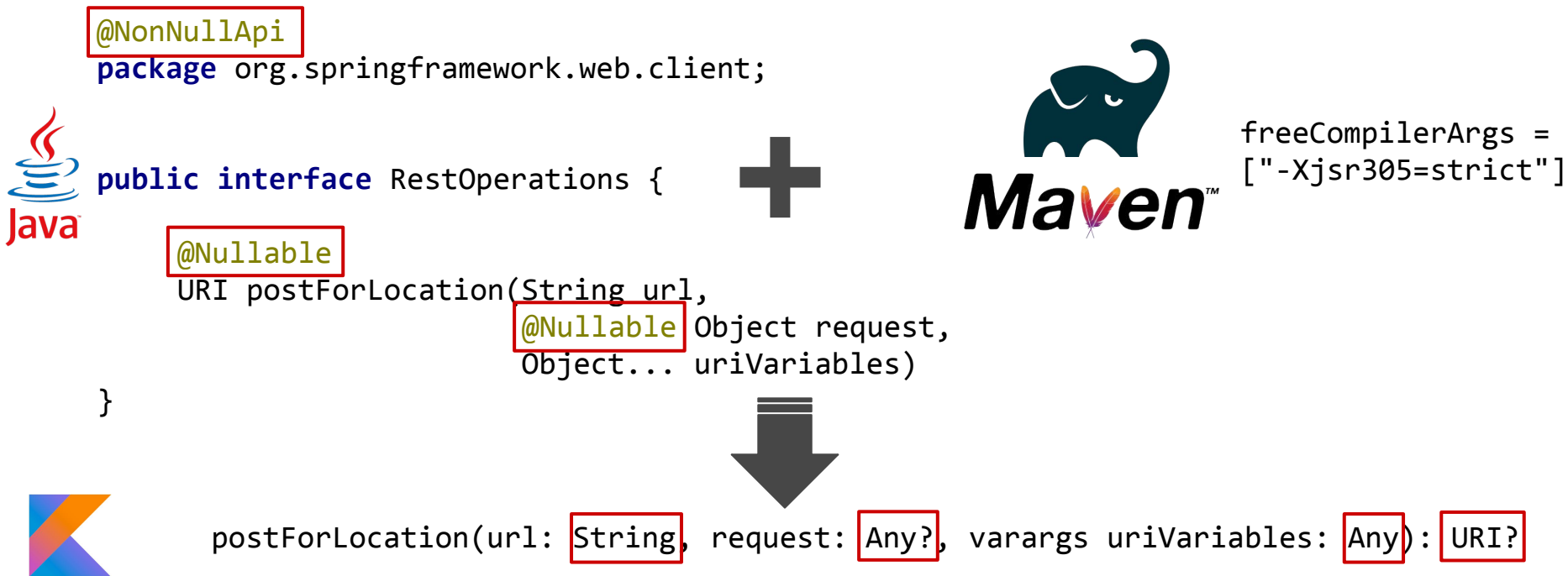
```
public interface RestOperations {  
    URI postForLocation(String url,  
                        Object request,  
                        Object ... uriVariables)  
}
```



```
postForLocation(url: String!, request: Any!, varags uriVariables: Any!): URI!
```

Null safety of Spring APIs

Nullability annotations meta annotated with JSR 305 for generic tooling support



@ConfigurationProperties

Not yet available



Spring Boot 1

```
@ConfigurationProperties("foo")
class FooProperties {
    var baseUri: String? = null
    val admin = Credential()

    class Credential {
        var username: String? = null
        var password: String? = null
    }
}
```



Spring Boot 2

```
@ConfigurationProperties("foo")
interface FooProperties {
    val baseUri: String
    val admin: Credential

    interface Credential {
        val username: String
        val password: String
    }
}
```

See also issue #8762 for discussion about data classes support

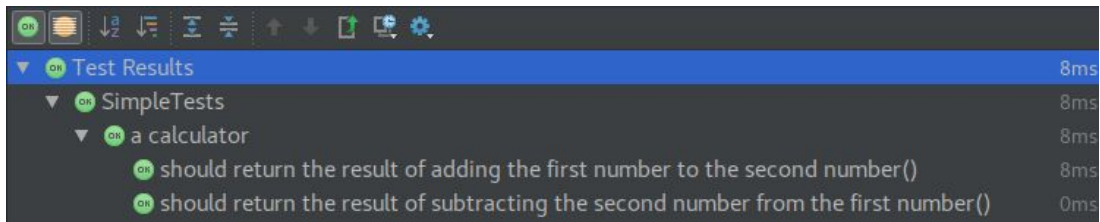
JUnit 5 supports non-static @BeforeAll @AfterAll

With “per class” lifecycle defined via junit-platform.properties or @TestInstance

```
class IntegrationTests {  
  
    private val application = Application(8181)  
    private val client = WebClient.create("http://localhost:8181")  
  
    @BeforeAll  
    fun beforeAll() { application.start() }  
  
    @Test  
    fun test1() { // ... }  
  
    @Test  
    fun test2() { // ... }  
  
    @AfterAll  
    fun afterAll() { application.stop() }  
  
}
```

Specification-like tests with Kotlin and JUnit 5

```
class SimpleTests {  
  
    @Nested  
    @DisplayName("a calculator")  
    inner class Calculator {  
        val calculator = SampleCalculator()  
  
        @Test  
        fun `should return the result of adding the first number to the second number`() {  
            val sum = calculator.sum(2, 4)  
            assertEquals(6, sum)  
        }  
  
        @Test  
        fun `should return the result of subtracting the second number from the first number`() {  
            val subtract = calculator.subtract(4, 2)  
            assertEquals(2, subtract)  
        }  
    }  
}
```



Kotlin type-safe templates

<https://github.com/sdeleuze/kotlin-script-templating>

Experimental

- Available via Spring MVC & WebFlux JSR-223 support
- 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.name}</li>" }}
</ul>
${include("footer")}
"""
```

- Step 1 Kotlin
- Step 2 Boot 2
- Step 3 WebFlux @nnotations**
- Step 4 Functional & DSL
- Step 5 Kotlin for frontend

Spring Framework 5 comes with 2 web stacks

Spring MVC

Blocking

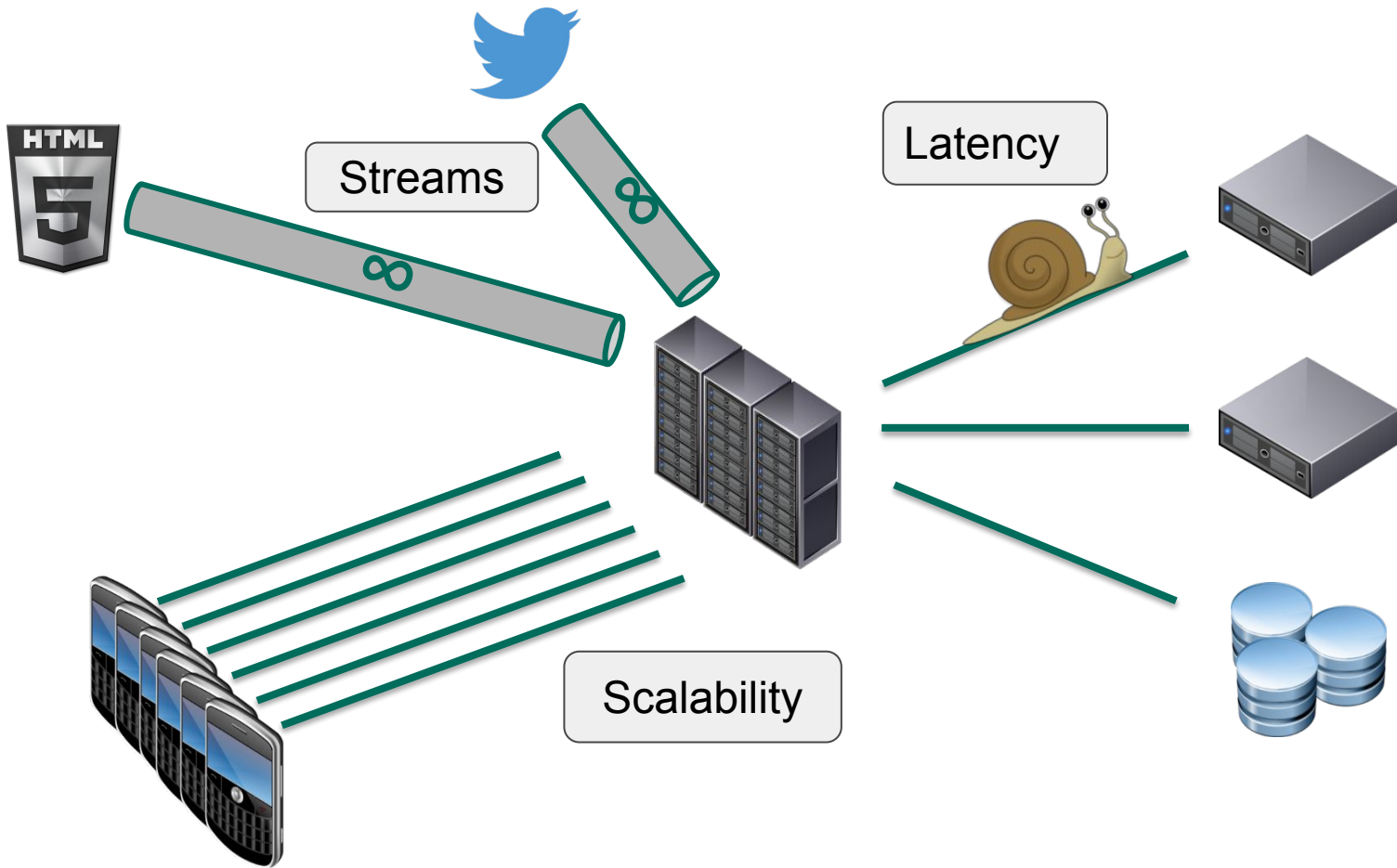
Servlet

Spring WebFlux

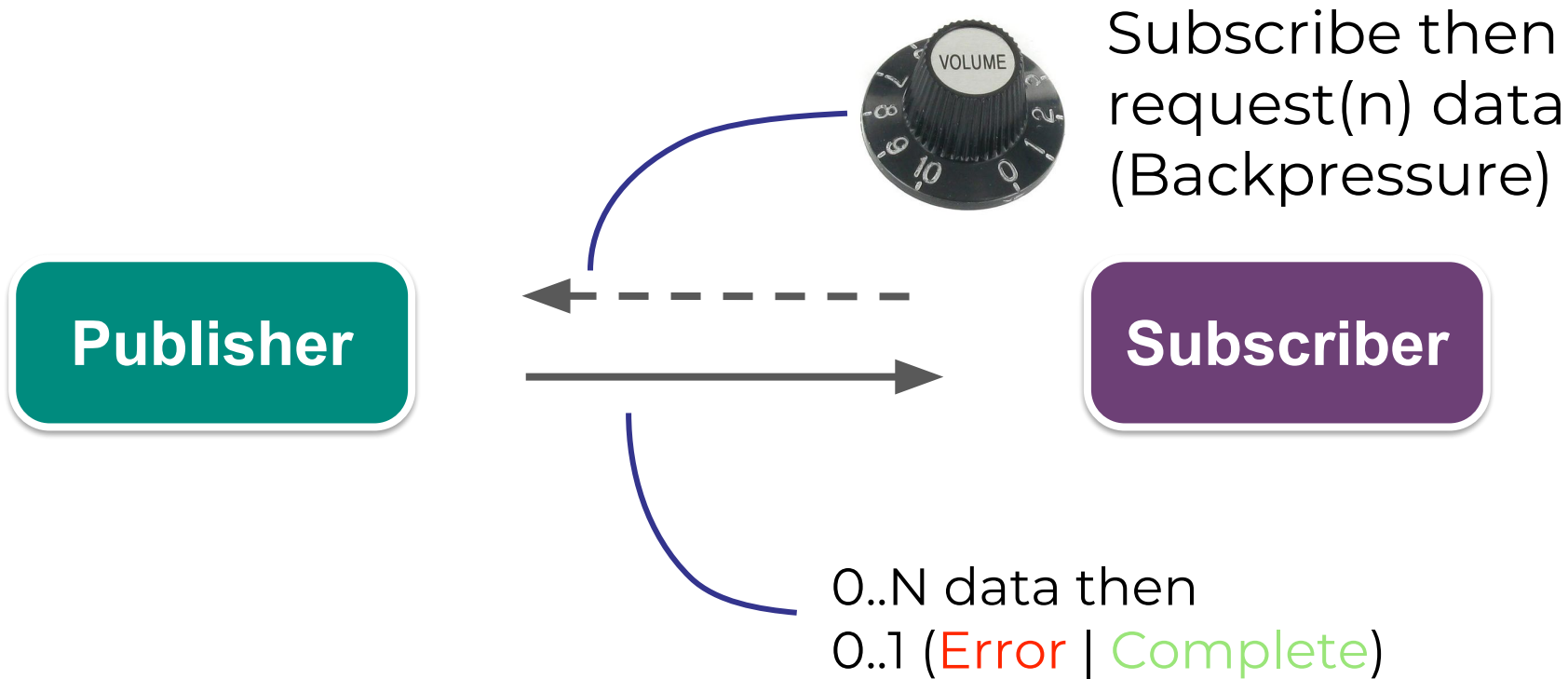
Non-blocking

Reactive Streams

NEW



Reactive Streams



WebFlux supports various async and Reactive API



CompletableFuture
Flow.Publisher



RxJava

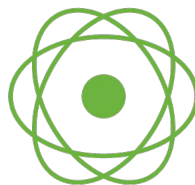


Reactor



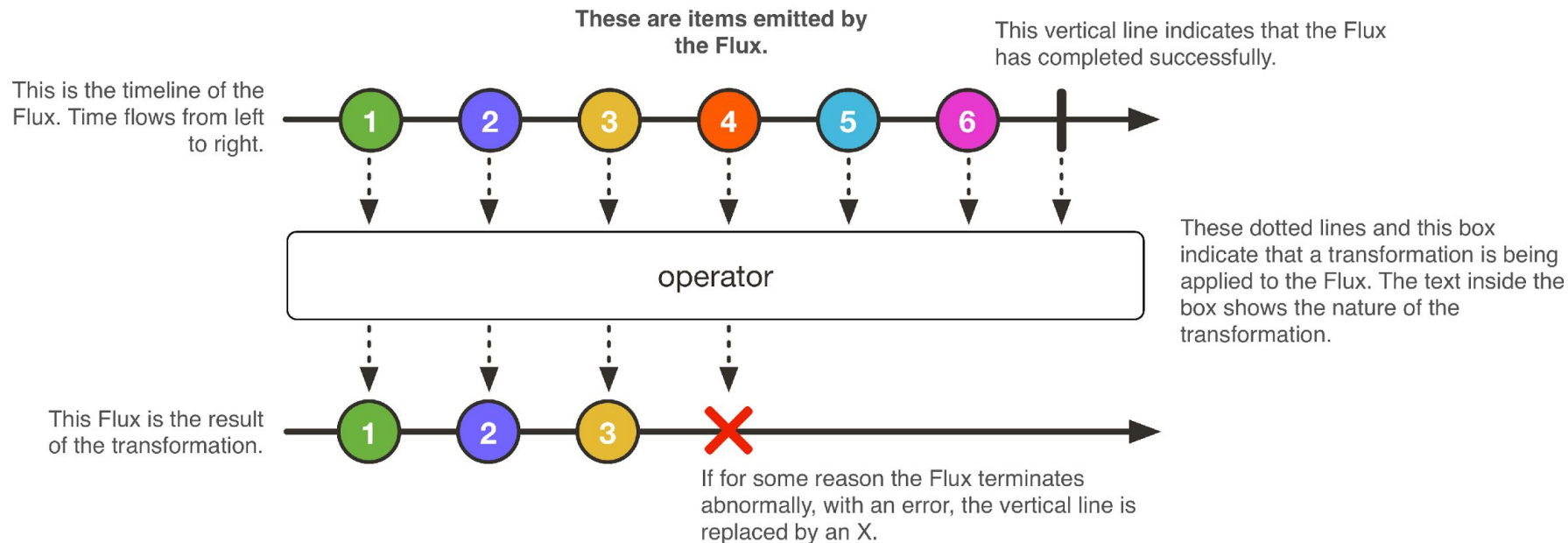
Akka Streams
(via Reactive Streams)

Let's focus on Reactor for now

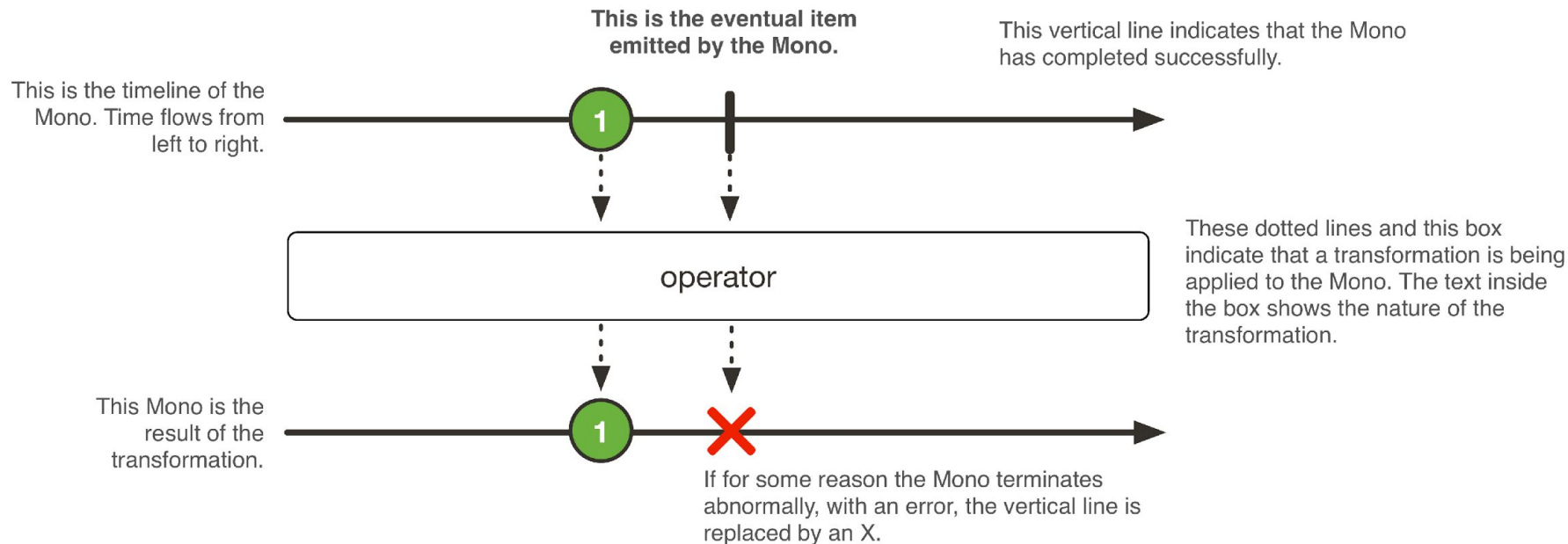


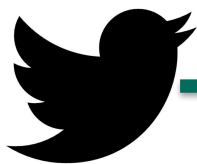
Reactor

Reactor Flux is a Publisher for 0..n elements



Reactor Mono is a Publisher for 0..1 element





Streaming API

WebFlux
client



`Flux.zip(tweets, issues)`



WebFlux
server

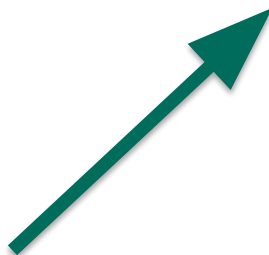


SSE
Websocket



REST API

WebFlux
client

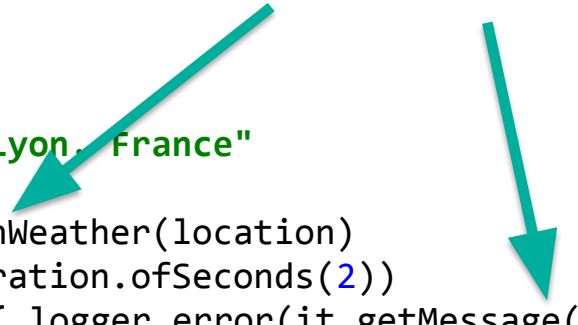


Reactive APIs are functional

```
fun fetchWeather(city: String): Mono<Weather>

val location = "Lyon, France"

mainService.fetchWeather(location)
    .timeout(Duration.ofSeconds(2))
    .doOnError { logger.error(it.getMessage()) }
    .onErrorResume { backupService.fetchWeather(location) }
    .map { "Weather in ${it.getLocation()} is ${it.getDescription()}" }
    .subscribe { logger.info(it) }
```

Two teal arrows originate from the code. One arrow starts at the 'Lyon, France' string in the 'val location' line and points to the 'location' parameter in the 'mainService.fetchWeather' call. The other arrow starts at the 'fetchWeather' function signature and points to the 'mainService.fetchWeather' call.

Reactor Kotlin extensions

Java	Kotlin with extensions
<code>Mono.just("foo")</code>	<code>"foo".toMono()</code>
<code>Flux.fromIterable(list)</code>	<code>list.toFlux()</code>
<code>Mono.error(new RuntimeException())</code>	<code>RuntimeException().toMono()</code>
<code>flux.ofType(User.class)</code>	<code>flux.ofType<User>()</code>
<code>StepVerifier.create(flux).verifyComplete()</code>	<code>flux.test().verifyComplete()</code>
<code>MathFlux.averageDouble(flux)</code>	<code>flux.average()</code> 🥰

Spring WebFlux with annotations

```
@RestController
class ReactiveUserController(val repository: ReactiveUserRepository) {

    @GetMapping("/user/{id}")
    fun findOne(@PathVariable id: String): Mono<User>
        = repository.findOne(id)

    @GetMapping("/user")
    fun findAll(): Flux<User>
        = repository.findAll()

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

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

Spring Data Kay provides
Reactive support for
MongoDB, Redis, Cassandra
and Couchbase

Spring & Kotlin Coroutines

Experimental

- Coroutines are light-weight threads
- Main use cases are
 - Using WebFlux and Spring Data Reactive while keeping imperative programming
 - Creating new operators for Reactor
- kotlinx.coroutines provides Reactive Streams and Reactor support
 - `fun foo(): Mono<T> -> suspend fun foo(): T?`
 - `fun bar(): Flux<T> -> suspend fun bar(): ReceiveChannel<T> or List<T>`
 - `fun baz(): Mono<Void> -> suspend fun baz()`
- Support for Spring MVC, WebFlux and Data Reactive MongoDB is available via <https://github.com/konrad-kaminski/spring-kotlin-coroutine/> (nice work Konrad!)
- **Warning**
 - Coroutine are still experimental
 - No official Spring support yet, see [SPR-15413](#)
 - Ongoing evaluation of performances and back-pressure interoperability

Spring WebFlux with Coroutines

<https://github.com/sdeleuze/spring-kotlin-deepdive/tree/step3-coroutine>

Experimental

```
@RestController
```

```
class CoroutineUserController( val repository: CoroutineUserRepository) {
```

```
    @GetMapping("/user/{id}")
```

```
    suspend fun findOne(@PathVariable id: String): User
        = repository.findOne(id)
```

```
    @GetMapping("/user")
```

```
    suspend fun findAll(): List<User>
        = repository.findAll()
```

```
    @PostMapping("/user")
```

```
    suspend fun save(@RequestBody user: User)
        = repository.save(user)
```

```
}
```

```
interface CoroutineUserRepository {
```

```
    suspend fun findOne(id: String): User
```

```
    suspend fun findAll(): List<User>
```

```
    suspend fun save(user: User)
```

```
}
```

- Step 1 Kotlin
- Step 2 Boot 2
- Step 3 WebFlux @nnotations
- Step 4 Functional & DSL**
- Step 5 Kotlin for frontend

Spring WebFlux server comes in 2 flavors

Annotations

@Controller

@RequestMapping

Functional

RouterFunction

HandlerFunction

NEW

Spring WebFlux functional APIs

RouterFunction

(ServerRequest) -> Mono<HandlerFunction>

HandlerFunction

(ServerRequest) -> Mono<ServerResponse>

WebClient

Provides non-blocking fluent HTTP client API

WebFlux functional API with Kotlin DSL

```
val router = router {  
  
    val users = ...  
  
    accept(TEXT_HTML).nest {  
        "/" { ok().render("index") }  
        "/sse" { ok().render("sse") }  
        "/users" {  
            ok().render("users", mapOf("users" to users.map { it.toDto() })))  
        }  
    }  
    ("/api/users" and accept(APPLICATION_JSON)) {  
        ok().body(users)  
    }  
    ("/api/users" and accept(TEXT_EVENT_STREAM)) {  
        ok().bodyToServerSentEvents(users.repeat().delayElements(ofMillis(100)))  
    }  
}
```

Functional router within Boot

```
@SpringBootApplication()
class Application {

    @Bean
    fun router(htmlHandler: HtmlHandler, userHandler: UserHandler, postHandler: PostHandler) = router {
        accept(APPLICATION_JSON).nest {
            "/api/user".nest {
                GET("/", userHandler::findAll)
                GET("/{login}", userHandler::findOne)
            }
            "/api/post".nest {
                GET("/", postHandler::findAll)
                GET("/{slug}", postHandler::findOne)
                POST("/", postHandler::save)
                DELETE("/{slug}", postHandler::delete)
            }
        }
        (GET("/api/post/notifications") and accept(TEXT_EVENT_STREAM)).invoke(postHandler::notifications)
        accept(TEXT_HTML).nest {
            GET("/", htmlHandler::blog)
            (GET("/{slug}") and !GET("/favicon.ico")).invoke(htmlHandler::post)
        }
    }
}
```

Functional handlers within Boot

@Component

```
class HtmlHandler(private val userRepository: UserRepository,  
                  private val markdownConverter: MarkdownConverter) {  
  
    fun blog(req: ServerRequest) = ok().render("blog", mapOf(  
        "title" to "Blog",  
        "posts" to postRepository.findAll()  
            .flatMap { it.toDto(userRepository, markdownConverter) } ))  
}
```

@Component

```
class PostHandler(private val postRepository: PostRepository,  
                  private val postEventRepository: PostEventRepository) {  
  
    fun findAll(req: ServerRequest) =  
        ok().body(postRepository.findAll())  
  
    fun notifications(req: ServerRequest) =  
        ok().bodyToServerSentEvents(postEventRepository.findWithTailableCursorBy())  
}
```

Spring Framework 5 also introduces Functional bean definition

Very efficient, no reflection, no CGLIB proxy
Lambdas instead of annotations
Both declarative and programmatic

Functional bean definition Kotlin DSL

```
val webContext = beans {  
    bean {  
        val userHandler = ref<UserHandler>()  
        router {  
            accept(APPLICATION_JSON).nest {  
                "/api/user".nest {  
                    GET("/", userHandler::findAll)  
                    GET("/{login}", userHandler::findOne)  
                }  
            }  
            // ...  
        }  
        bean { Mustache.compiler().escapeHTML(false).withLoader(ref()) }  
        bean<HtmlHandler>()  
        bean<PostHandler>()  
        bean<UserHandler>()  
        bean<MarkdownConverter>()  
    }  
}
```

Functional bean definition Kotlin DSL

```
val databaseContext = beans {  
    bean<PostEventListener>()  
    bean<PostEventRepository>()  
    bean<PostRepository>()  
    bean<UserRepository>()  
    environment( { !activeProfiles.contains("cloud") } ) {  
        bean {  
            CommandLineRunner {  
                initializeDatabase(ref(), ref(), ref())  
            }  
        }  
    }  
}  
  
fun initializeDatabase(ops: MongoOperations,  
    userRepository: UserRepository,  
    postRepository: PostRepository) { // ... }
```

Functional bean definition with Spring Boot

This nice syntax currently works only for running the application, not tests

```
@SpringBootApplication
class Application

fun main(args: Array<String>) {
    runApplication<FooApplication>(*args) {
        addInitializers(databaseContext, webContext)
    }
}
```

Functional bean definition with Spring Boot

This more verbose one works for running both application and tests

ContextInitializer

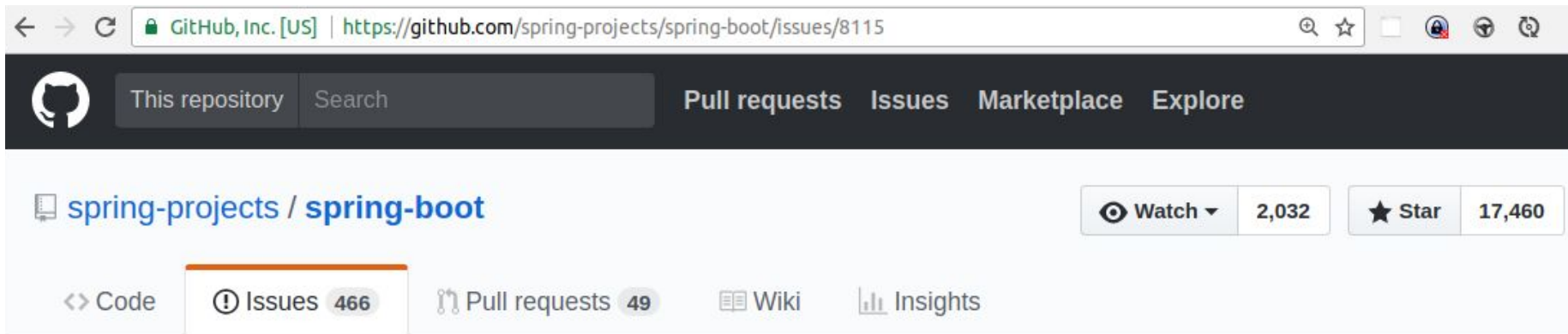
```
class ContextInitializer : ApplicationContextInitializer<GenericApplicationContext> {  
    override fun initialize(context: GenericApplicationContext) {  
        databaseContext.initialize(context)  
        webContext.initialize(context)  
    }  
}
```

application.properties

```
context.initializer.classes=io.spring.deepdive.ContextInitializer
```

Functional bean definition with Spring Boot

Come discussing next steps with us



Expose the functional bean registration API via SpringApplication #8115



wilkinsona opened this issue on Jan 26 · 26 comments

- Step 1 Kotlin
- Step 2 Boot 2
- Step 3 WebFlux @nnotations
- Step 4 Functional & DSL
- Step 5 Kotlin for frontend**

Original JavaScript code

```
if (Notification.permission === "granted") {  
  Notification.requestPermission().then(function(result) {  
    console.log(result);  
  });  
}  
  
let eventSource = new EventSource("/api/post/notifications");  
eventSource.addEventListener("message", function(e) {  
  let post = JSON.parse(e.data);  
  let notification = new Notification(post.title);  
  notification.onclick = function() {  
    window.location.href = "/" + post.slug;  
  };  
});
```

Kotlin to Javascript

Type safety, null safety, only 10 Kbytes with Dead Code Elimination tool

```
data class Post(val slug: String, val title: String)

fun main(args: Array<String>) {
    if (Notification.permission == NotificationPermission.GRANTED) {
        Notification.requestPermission().then { console.log(it) }
    }
    EventSource("/api/post/notifications").addEventListener("message", {
        val post = JSON.parse<Post>(it.data());
        Notification(post.title).addEventListener("click", {
            window.location.href = "/${post.slug}"
        })
    })
}

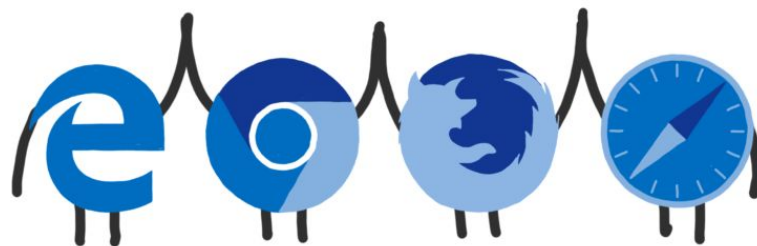
fun Event.data() = (this as MessageEvent).data as String // See KT-20743
```


Kotlin 1.2 allows building multi-platform modules

Sharing code between backend, frontend, mobile ...

WebAssembly

Native platform for the Web

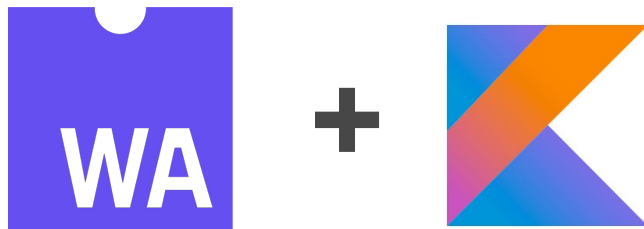


Read “An Abridged Cartoon Introduction To WebAssembly” by Lin Clark for more details

<https://goo.gl/I0kQsC>

Compiling Kotlin to WebAssembly

Experimental



- Kotlin supports WebAssembly via Kotlin Native (LLVM)
- Much better compilation target (long term)
- No DOM and Web API access yet but that's coming ...
- A Kotlin/Native Frontend ecosystem could arise
- Native level performances, low memory consumption
- Fallback via asm.js

Thanks!



<https://speakerdeck.com/sdeleuze/why-spring-loves-kotlin>



Follow me on **@sdeleuze** for fresh Spring + Kotlin news



<https://github.com/mixitconf/mixit>

<https://github.com/sdeleuze/spring-kotlin-fullstack>

<https://github.com/sdeleuze/spring-kotlin-deepdive>

<https://github.com/sdeleuze/spring-kotlin-functional>