Spring Boot & Kotlin

Pain or Gain?

by Urs Peter

LinkedIn: bit.ly/urs-peter-linked-in

Email: upeter@xebia.com

Blog: xebia.com/blog/

Kotlin Training

Certified by JetBrains

About Me:

Kotlin Training

Certified by JetBrains

Xebia Academy

KotlinConf

Kotlin DEV DAY

KKON

DEVOXX United Kingdom

TEQNATION CODE | INNOVATE | CREATE

Kotlin

Scala

Java

Xebia

Nice to you, too!

Hi I’m **Jadev**, a seasoned Java developer. I don’t take anything for granted!

Jadev

Nice to meet:

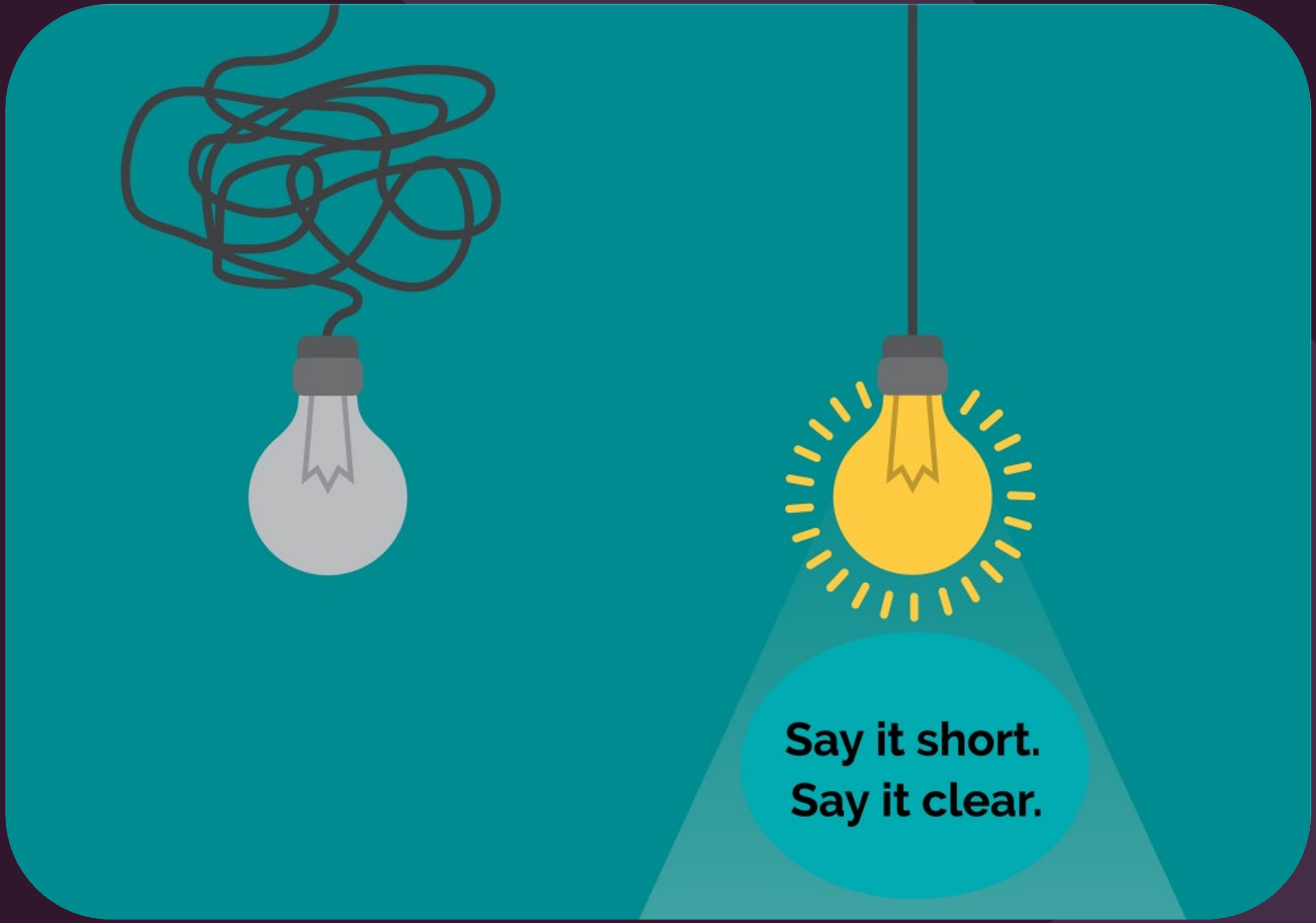
Y😃U

Xebia

Why should you choose for Kotlin rather than modern Java?

I’m using Java for quite some time and works for me.
*What’s the **GAIN** for me* when using Kotlin?

Jadev



Kotlin is more *concise* than Java

Xebia

Kotlin’s all-in-one class, field, getter/setter declaration

```java
public class User {
   ~~private final String email;~~
   ~~private final Optional<URL> avatarUrl;~~
   ~~private boolean emailVerified;~~

   public User(String email, boolean emailVerified, Optional<URL>
                                                   avatarUrl) {
      this.email = email;
      this.emailVerified = emailVerified;
      this.avatarUrl = avatarUrl;
   }

   public User(String email, boolean emailVerified) {
      this.email = email;
      this.emailVerified = emailVerified;
      this.avatarUrl = Optional.empty();
   }

   public User(String email, Optional<URL> avatarUrl) {
      this.email = email;
      this.emailVerified = false;
      this.avatarUrl = avatarUrl;
   }

   public User(String email) {
      this.email = email;
      this.emailVerified = false;
      this.avatarUrl = Optional.empty();

   }

   public void sayHi() {
              System.out.println(STR."Hi from \{email}");
   }

   ~~public String getEmail() {~~
      ~~return email;~~
   ~~}~~

   ~~public boolean isEmailVerified() {~~
      ~~return emailVerified;~~
   ~~}~~

   ~~public void setEmailVerified(boolean emailVerified) {~~
      ~~this.emailVerified = emailVerified;~~
   ~~}~~

   ~~public Optional<URL> getAvatarUrl() {~~
      ~~return avatarUrl;~~
   ~~}~~
}
```

```kotlin
class User(val email: String,
           val avatarUrl:URL? = null,
           var emailVerified: Boolean = false) {

   fun sayHi():Unit = println("Hi from $email")

}
```

Xebia

No more overloads with Default Arguments

```java
public class User {
   private final String email;
   private final Optional<URL> avatarUrl;
   private boolean emailVerified;

    public User(String email, boolean emailVerified, Optional<URL>
                                                           avatarUrl) {
       this.email = email;
       this.emailVerified = emailVerified;
       this.avatarUrl = avatarUrl;
    }

    public User(String email, boolean emailVerified) {
       this.email = email;
       this.emailVerified = emailVerified;
       this.avatarUrl = Optional.empty();
    }

    public User(String email, Optional<URL> avatarUrl) {
       this.email = email;
       this.emailVerified = false;
       this.avatarUrl = avatarUrl;
    }

    public User(String email) {
       this.email = email;
       this.emailVerified = false;
       this.avatarUrl = Optional.empty();

    }

    public void sayHi() {
                System.out.println(STR."Hi from \{email}");
    }

    public String getEmail() {
       return email;
    }

    public boolean isEmailVerified() {
       return emailVerified;
    }

    public void setEmailVerified(boolean emailVerified) {
       this.emailVerified = emailVerified;
    }

    public Optional<URL> getAvatarUrl() {
       return avatarUrl;
    }
}
```

```kotlin
class User(val email: String,
           val avatarUrl:URL? = null,
           var emailVerified: Boolean = false){

   fun sayHi():Unit = println("Hi from $email")

}
```

No more overloads with Default Arguments

```java
public class User {
   private final String email;
   private final Optional<URL> avatarUrl;
   private boolean emailVerified;

   public User(String email, boolean emailVerified, Optional<URL>
                                                        avatarUrl) {
      this.email = email;
      this.emailVerified = emailVerified;
      this.avatarUrl = avatarUrl;
   }

   public User(String email, boolean emailVerified) {
      this.email = email;
      this.emailVerified = emailVerified;
      this.avatarUrl = Optional.empty();
   }

   public User(String email, Optional<URL> avatarUrl) {
      this.email = email;
      this.emailVerified = false;
      this.avatarUrl = avatarUrl;
   }

   public User(String email) {
      this.email = email;
      this.emailVerified = false;
      this.avatarUrl = Optional.empty();

   }

   public void sayHi() {
                sayHi("Hi from")
   }

   public void sayHi(String greeting) {
                System.out.println(STR."\{greeting} \{email}");
   }

   public String getEmail() {
      return email;
   }

   public boolean isEmailVerified() {
      return emailVerified;
   }

   public void setEmailVerified(boolean emailVerified) {
      this.emailVerified = emailVerified;
   }
```

```kotlin
class User(val email: String,
           val avatarUrl:URL? = null,
           var emailVerified: Boolean =
false){

   fun sayHi(greeting:String = "Hi from") =
                     println("$greeting $email")
}
```

No more builders with Named Arguments

```java
final var user = UserBuilder
                     .email("spr@ing.io")
                     .avatarUrl("http://url")
                   .build(),
    …)
```

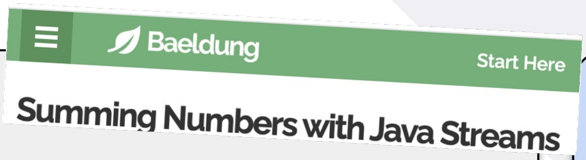
```kotlin
val user = User(
                  email = "spr@ing.io",
                  avatarUrl = "http://url"),
    …)
```

```java
public class UserBuilder {
   private String email;
   private URL avatarUrl;
   private boolean emailVerified = false;

   public UserBuilder() {   }

   public UserBuilder email(String email) {
       this.email = email;
       return this;
   }

   public UserBuilder avatarUrl(URL avatarUrl) {
       this.avatarUrl = avatarUrl;
       return this;
   }

   public UserBuilder verified(boolean verified) {
       this.emailVerified = verified;
       return this;
   }

   public User build() {
       return new User(this.name,
                    Optional.of(this.avatarUrl),
                    this.emailVerified,
       );
   }
}
```

In other words:
Kotlin makes Lombok **obsolete**

~~Project Lombok~~

Kotlin Collections & Conciseness



Summing Numbers with Java Streams

```
List.of(1, 2, 3)
        ~~.stream()~~
        ~~.mapToInt(Integer::valueOf)~~
        .sum();
//6
```

Guide to Java 8 groupingBy Collector

```
users.~~stream()~~
      ~~.collect(Collectors~~
        .groupingBy(User::isEmailVerified));
//[(true=[User(...),], false=[User(...)])]
```

```
 Map<Integer, String> swapped =
   Map.of("Jack", 42, "Sue", 22)
   ~~.entrySet()~~
   ~~.stream()~~
   .collect(
     Collectors.toMap(
       Map.Entry::getValue,
       Map.Entry::getKey)
   );
 //[(42, Jack), (22, Sue)]
```

```
listOf(1,2,3).sum()
```

`sum(), avg()` on numerical Collections only 😃

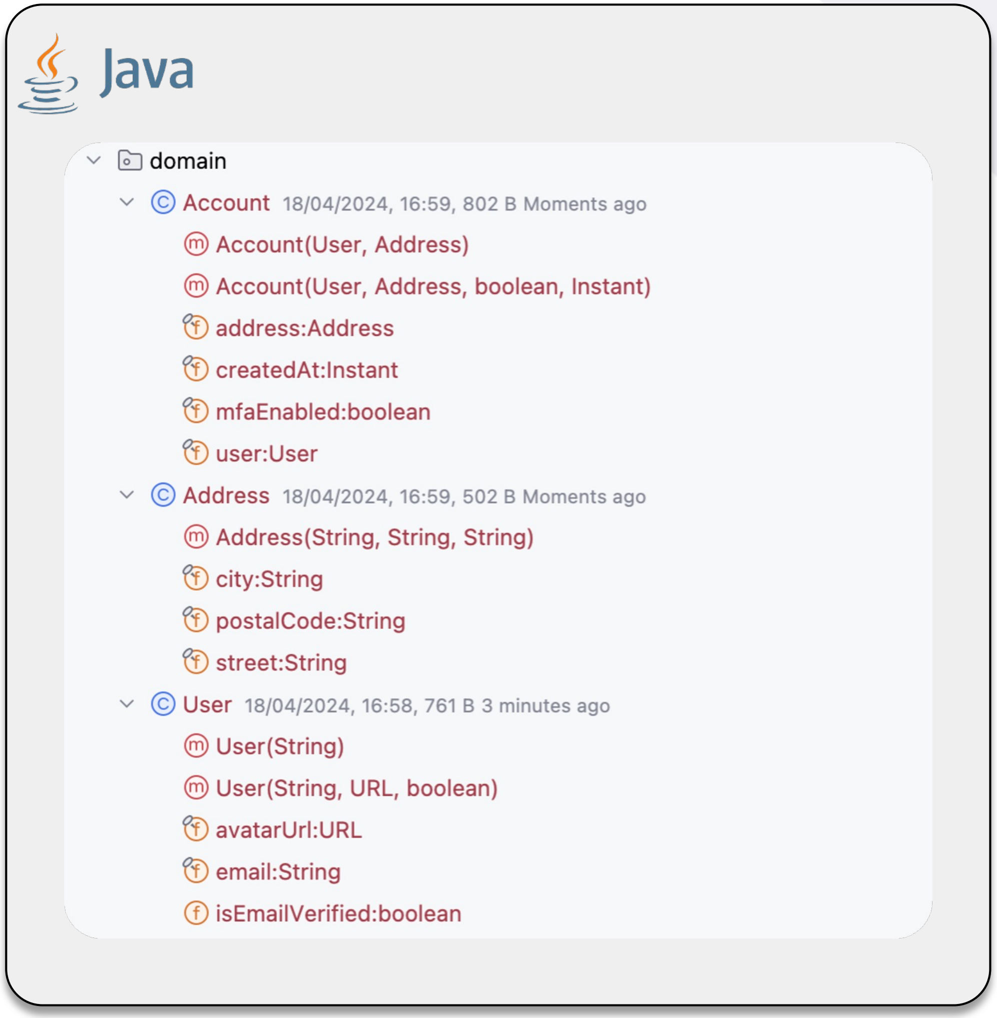
```
users.groupBy{ it.isEmailVerified }
```

```
val swapped =
   mapOf("Jack" to 42, "Sue" to 22)
        .map{ (name, age) ->  age to name}
```

Destructuring 😃

Identical higher-order functions on
`List, Map, Set, Range, Array, String` 😃

First-class Class support


Java

domain
- Account 18/04/2024, 16:59, 802 B Moments ago
  - Account(User, Address)
  - Account(User, Address, boolean, Instant)
  - address:Address
  - createdAt:Instant
  - mfaEnabled:boolean
  - user:User
- Address 18/04/2024, 16:59, 502 B Moments ago
  - Address(String, String, String)
  - city:String
  - postalCode:String
  - street:String
- User 18/04/2024, 16:58, 761 B 3 minutes ago
  - User(String)
  - User(String, URL, boolean)
  - avatarUrl:URL
  - email:String
  - isEmailVerified:boolean

Kotlin Domain.kt

```kotlin
package domain

import java.net.URL
import java.time.Instant

class User(val email: String,
           val avatarUrl: URL? = null,
           var isEmailVerified: Boolean)

class Account(val user:User,
              val address: Address,
              val mfaEnabled:Boolean,
              val createdAt: Instant)

class Address(val street: String,
              val city: String,
              val postalCode: String)
```

Kotlin allows multiple public class/interface declarations in a single file.👍

Kotlin *conciseness delivers:*

30%-40% More Code Clarity / Code Reduction

```java
public User(String email, boolean emailVerified) {
this.email = email; this.emailVerified = emailVerified;
 this.avatarUrl = Optional.empty();}
public User(String email, Optional<URL> avatarUrl) {
 this.email = email; this.emailVerified = false;
this.avatarUrl = avatarUrl;}
public User(String email) {
  this.email = email;  this.emailVerified = false;
this.avatarUrl = Optional.empty();
this.type = Optional.empty();
```

```kotlin
class User(val email: String,
  val avatarUrl:URL? = null,
  var emailVerified: Boolean){
   fun sayHi()  =
      println("Hi $email)

   fun sayHo()  =
      println("Ho $email)
}
```

30%-40%

Source: https://developer.android.com/kotlin/first

Kotlin is *safer* than Java

Kotlin is *safer* than Java: Null safety

```java
public class User {
   private final String email;
   private final Optional<URL> avatarUrl;
   private boolean emailVerified;

  public User(@NotNull String email,
              Optional<URL> avatarUrl,
              boolean emailVerified) {
    this.email = email;
    this.emailVerified = emailVerified;
    this.avatarUrl = avatarUrl;
  }
  ...
}
```

Compiles, but fails at runtime (IAE, NPE) 👎

```java
 final var user = new User(null, null, null);
```

```java
 Optional<User> userOpt = findById(...);

 userOpt.flatMap(User::getAvatarUrl)
        .flatMap(url ->
           Optional.ofNullable(url.getQuery()))
        .orElse("");
```

```kotlin
class User(val email: String,
           val avatarUrl:URL? = null,
           var emailVerified: Boolean = false)
{
}
```

❤️ Nullability is Kotlin's most loved feature ❤️

Does not compile 👍

```kotlin
 val user = User(null, null, null)
```

```kotlin
 val user:User? = findById(...)

 user?.avatarUrl?.query ?: ""
```

Easily travers nullable object graph with: ? 👍

```kotlin
emptyList<Int>().firstOrNull()

listOf(user1,null).filterNotNull().maxByOrNull{
     it.email.size
}

val user2:User? = "no user!" as? User
```

Kotlin is *safer* than Java: Primary Constructor

```java
public class User {
  private String email;
  private Optional<URL> avatarUrl;
  private boolean emailVerified;

  public User() {}

  public User(@NotNull String email,
              Optional<URL> avatarUrl,
              boolean emailVerified) {
     this.email = email;
     this.emailVerified = emailVerified;
     this.avatarUrl = avatarUrl;
  }
  ...
}
```

```java
final var user = new User();
user.getEmail().contains("@")
```

NPE 👎

```kotlin
class User(val email: String,
           val avatarUrl:URL? = null,
           var emailVerified: Boolean = false)
{
}
```

Primary constructor must be called

…which ensures the instance is initialized correctly 👍

```kotlin
val user = User("spr@ing.io")
```

Kotlin is *safer* than Java: Smart Casts also for Nullable Types

```java
public void process(Object obj) {
 if(obj instanceof User user &&
   user.getAvatarUrl().isPresent()) {
     System.out.println("Avatar path: " +
      user.getAvatarUrl().get().getPath());
   }
}
```

Smart casts 👍

```kotlin
fun process(any:Any) {
  if(any is User && any.avatarUrl != null) {
     println("Avatar path:
${any.avatarUrl.path}")
  }
}
```

Xebia

Kotlin is *safer* than Java: Smart Casts

```java
public void process(Object obj) {
  if(obj instanceof User user) //&&
    //((User)obj).getAvatarUrl().isPresent()) {
      System.out.println(
        "Avatar path: " +
       user.getAvatarUrl().get().getPath());
    }
}
```

Runtime Exception 👎

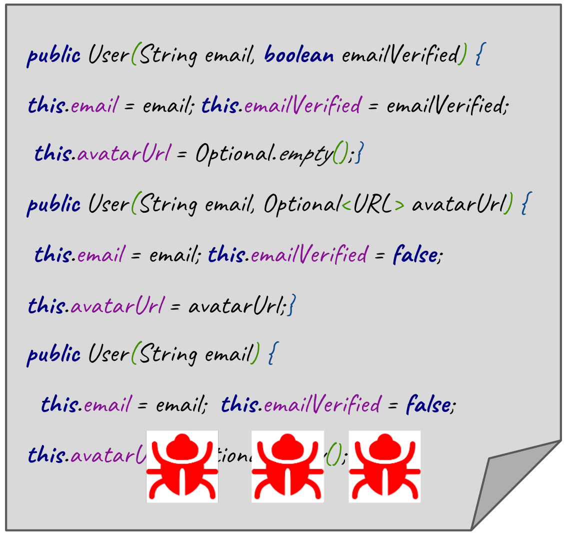
```kotlin
fun process(any:Any) {
   if(any is User) { //&& any.avatarUrl != null)
{
      println("Avatar path:
${any.avatarUrl
    }
}
```

Does not compile 👍

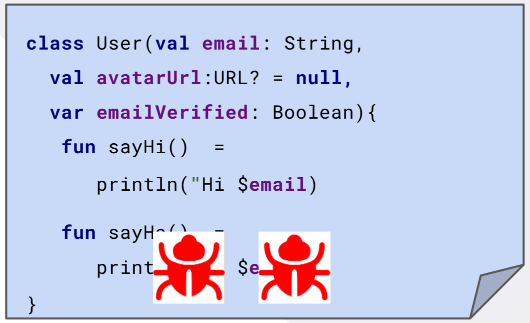
Xebia

Kotlin *safety features deliver:*

~ 30% Less Bugs due to Safety Features



```
public User(String email, boolean emailVerified) {
this.email = email; this.emailVerified = emailVerified;
 this.avatarUrl = Optional.empty();}
public User(String email, Optional<URL> avatarUrl) {
 this.email = email; this.emailVerified = false;
this.avatarUrl = avatarUrl;}
public User(String email) {
  this.email = email;  this.emailVerified = false;
this.avatarU      tion.        ();
```



```
class User(val email: String,
  val avatarUrl:URL? = null,
  var emailVerified: Boolean){
   fun sayHi()  =
      println("Hi $email)

   fun sayH-()  =
      print        $e
}
```

for <u>all</u> JVM versions

Source: https://developer.android.com/kotlin/first

Kotlin favors *immutability* more than Java

Java records vs Kotlin data classes

```java
public record User(
                   String email,
                   Optional<URL> avatarUrl,
                   boolean emailVerified) {
 public User(...) {...} x 4

 public void sayHi() {
   System.out.println(STR."Hi from \{email}");
 }
}
```

```kotlin
data class User(
           val email: String,
           val avatarUrl:URL? = null,
           var emailVerified: Boolean = false){

    fun sayHi() = println("Hi from $email")
}
```

Mutability possible 👍

```java
final var jack = new User("spr@ing.io");
jack: User[email=Jack, avatarUrl=...]

jack.equals(new User("spr@ing.io"));
res1: true

final var fred = new User("info@ing.io",
                       jack.emailVerified(),
                       jack.avatarUrl());
                  @ing.io, avatarUrl=...)
```

Requires copying all arguments 👎

~~JPA~~ ~~Java Persistence API~~
Not compatible

```kotlin
  val jack = User("spr@ing.io")
  jack: User(email=spr@ing.io, avatarUrl=...)

  jack == User("spr@ing.io")
  res1: true

  val fred = jack.copy(email = "info@ing.io")
  fred: User(email=info@ing.io, avatarUrl=...)
```

convenient copy method with named arguments 👍

JPA
Java Persistence API
Compatible

Immutable Collections

```java
final var users = List.of(user1, user2);

users.add(user3);
```

Throws `UnsupportedOperationException` 👎

```java
static <T> List<T> appendAnElement(
                List<T> immList, T element) {
   List<T> tmpList = new ArrayList<>(immList);
   tmpList.add(element);
   return Collections.unmodifiableList(tmpList);
}
```

Add One Element to an Immutable List in Java

Immutable by default

Returns new collection 👍

```kotlin
val users = listOf(user1, user2)

val newUsers = users + user3

val fromJava = List.of(user1, user2) + user3
```

No exception here 👍

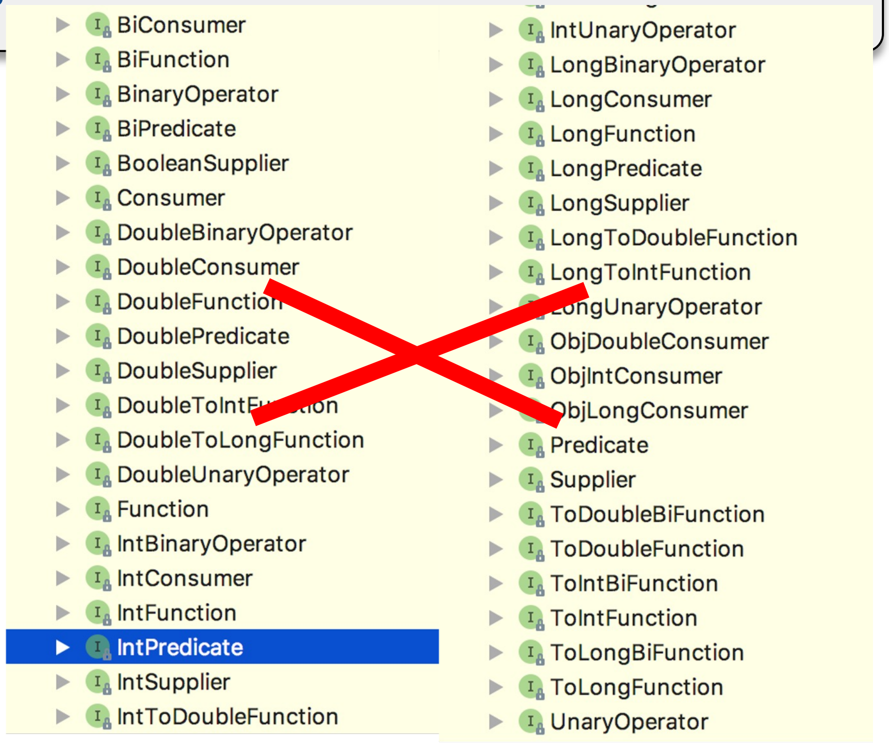
Xebia

Kotlin is more *Functional* than Java

Functions as First Class Citizens

Java

```java
public void doWithImage(
     URL url,
     BiConsumer<String, BufferedImage> f)
                    throws IOException {

  f.accept(url.getFile(), ImageIO.read(url));

}
```



```
BiConsumer                 IntUnaryOperator
BiFunction                 LongBinaryOperator
BinaryOperator             LongConsumer
BiPredicate                LongFunction
BooleanSupplier            LongPredicate
Consumer                   LongSupplier
DoubleBinaryOperator       LongToDoubleFunction
DoubleConsumer             LongToIntFunction
DoubleFunction             LongUnaryOperator
DoublePredicate            ObjDoubleConsumer
DoubleSupplier             ObjIntConsumer
DoubleToIntFunction        ObjLongConsumer
DoubleToLongFunction       Predicate
DoubleUnaryOperator        Supplier
Function                   ToDoubleBiFunction
IntBinaryOperator          ToDoubleFunction
IntConsumer                ToIntBiFunction
IntFunction                ToIntFunction
IntPredicate               ToLongBiFunction
IntSupplier                ToLongFunction
IntToDoubleFunction        UnaryOperator
```

Kot

In Kotlin Functions are *first class citizens* with syntax support for *declaring* functions 👍

```kotlin
fun doWithImage(
  url: URL,
  f:(String, BufferedImage) -> ()) =
 f(url.file, ImageIO.read(url))
```

Xebia

Kotlin is more *flexible* than Java

Kotlin is more flexible than Java: Extensions

Kotlin offers *extension functions* that allow extending existing types

```java
public static UserDto toDto(user:User) {
  return new UserDto(
                  email,
                  avatarUrl,
                  emailVerified);
}
```

```kotlin
fun User.toDto():UserDto =
   UserDto(email, avatarUrl, emailVerified)
```

If in scope (same package or imported), User now has a `toDto()` method.

```java
final var userDto = toDto(user)
```

```kotlin
val userDto = user.toDto()
```

Extensions great because they:
- lead to fluent code 👍
- show up in code completion 👍
- can be scoped 👍

Kotlin is more flexible than Java: Extensions in APIs

For interoperability, Kotlin relies on Java APIs that are 'pimped' with Extensions.

Standard library extensions:

```
"hi".reversed()
"5".toIntOrNull()

URL("...")
    .openStream()
    .copyTo(File("out.txt").outputStream())
```

3rd party extensions
(Spring, Jackson etc.):

```
jdbcTemplate.queryForObject<User>("select …")


objectMapper.readValue<List<User>>("[...]")
```

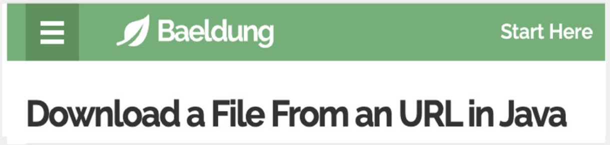
Testing!

```
"wow" shouldBe "wow".reversed()
```

Baeldung Start Here

How to Reverse a String in Java

```
new StringBuilder("hi").reverse().toString();
```



Baeldung Start Here

Download a File From an URL in Java

```
jdbcTemplate.queryForObject(
          "select ~~...", User.class~~);

objectMapper.readValue("[...]",
    ~~new TypeReference<List<User>>(){ }~~);
```

Xebia

Kotlin is more flexible than Java: DSLs

```
9 should beLessThan(10)
shouldThrow<NumberFormatException> { "Nan".toInt() }

"Kotlin" should startWith("K")
"12:30" should match("""\d{2}:\d{2}""")

listOf(1,2) should containExactlyInAnyOrder(listOf(2,1))
listOf(1,2,3) should beSorted()
```

So what are the Gains of using Kotlin?

Kotlin is more *concise* than Java

~30%-40% better code clarity / reduction

Kotlin is *safer* than Java

~30% less bugs

Kotlin favors *immutability* more than Java

Thread-safe, easier to reason and work with immutable domains

Kotlin is more *Functional* than Java

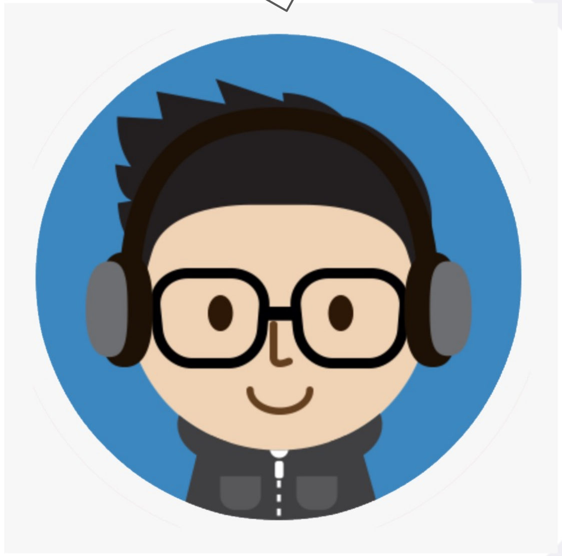
Functions are First-Class-Citizens

Kotlin is more *flexible* than Java

More fluent and richer APIs through Extensions

Java is not standing still

Java is catching up quickly,
so why not simply wait?

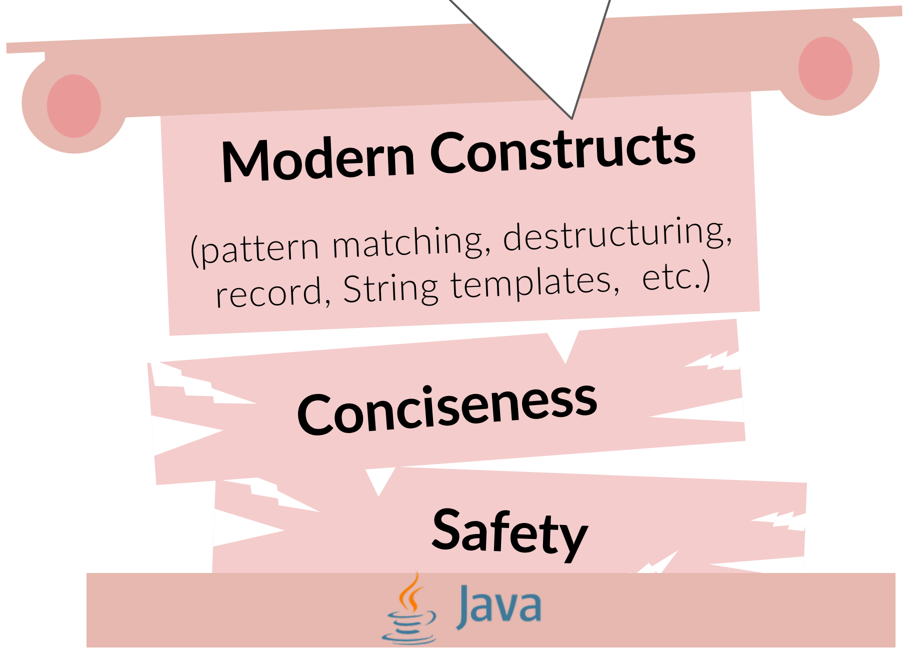


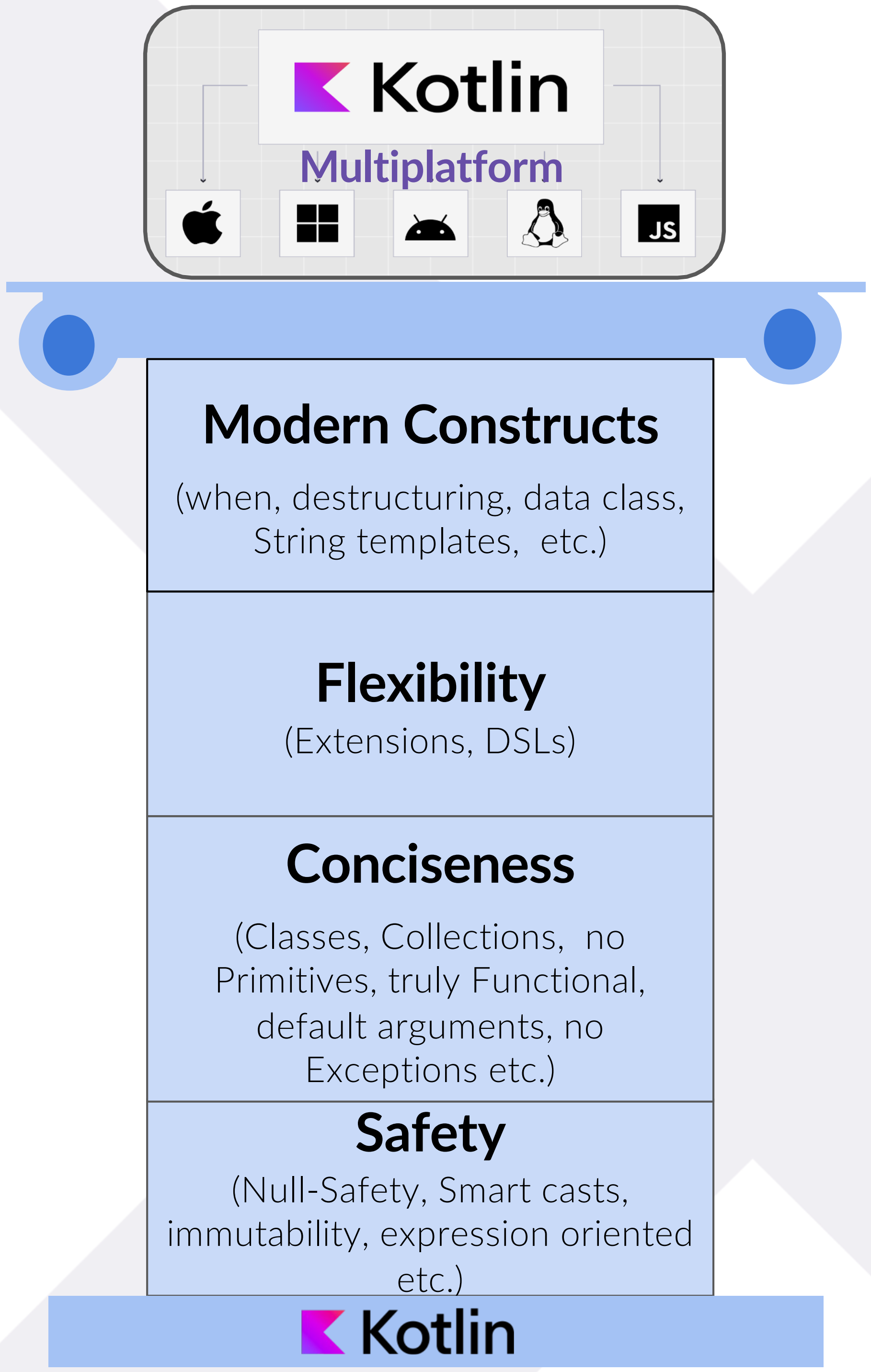
Jadev

True, ‘but’…

…and with every new feature *ambiguity* increases.

Records vs Classes?
Structured Concurrency vs Reactive?
var vs type vs Lombok?


Modern Constructs
(pattern matching, destructuring, record, String templates, etc.)
Conciseness
Safety
Java


Kotlin
Multiplatform
JS
Modern Constructs
(when, destructuring, data class, String templates, etc.)
Flexibility
(Extensions, DSLs)
Conciseness
(Classes, Collections, no Primitives, truly Functional, default arguments, no Exceptions etc.)
Safety
(Null-Safety, Smart casts, immutability, expression oriented etc.)
Kotlin

Java vs Kotlin at one glance

Java

Kotlin

However, for companies it's harder to find Kotlin developers than Java developers.

Xebia

Kotlin & Spring Boot

Sounds all nice and dandy, **but**:
“how well are all these features supported in Spring Boot?”

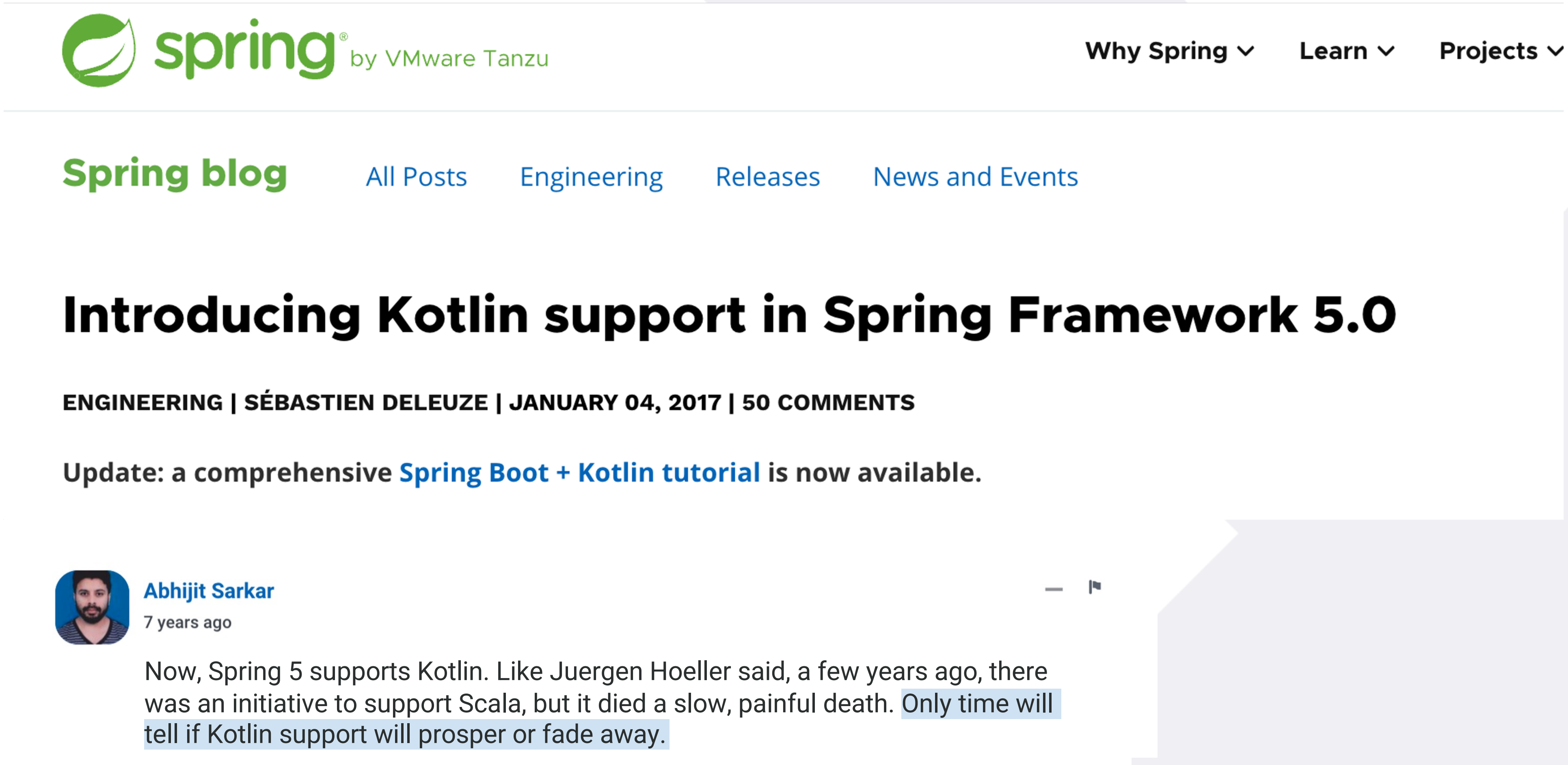
Jadev

Kotlin

spring
BOOT

Kotlin & Spring Boot Web

Spring Boot & Kotlin: First-Class Citizen?



Spring blog | All Posts | Engineering | Releases | News and Events

Introducing Kotlin support in Spring Framework 5.0

ENGINEERING | SÉBASTIEN DELEUZE | JANUARY 04, 2017 | 50 COMMENTS

Update: a comprehensive Spring Boot + Kotlin tutorial is now available.

Abhijit Sarkar
7 years ago

Now, Spring 5 supports Kotlin. Like Juergen Hoeller said, a few years ago, there was an initiative to support Scala, but it died a slow, painful death. Only time will tell if Kotlin support will prosper or fade away.

A Simple Spring Boot application in Kotlin

```kotlin
@Configuration
open class Configuration {
   @Bean
   open fun restTemplate(): RestTemplate = RestTemplate()
}

@JsonIgnoreProperties
data class Joke(val joke:String, val lang:String)

@RestController
class JokeController(val restTemplate: RestTemplate,
                    @Value("\${root.uri}") val rootUri:String) {

 @GetMapping("/jokes")
 @ResponseBody
 fun randomJoke(@RequestParam("category") category: String?):Joke? =
    restTemplate.getForEntity<Joke>("$rootUri/${category ?: "Programming"}?type=single").body
}

@SpringBootApplication
open class JokesApplication

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

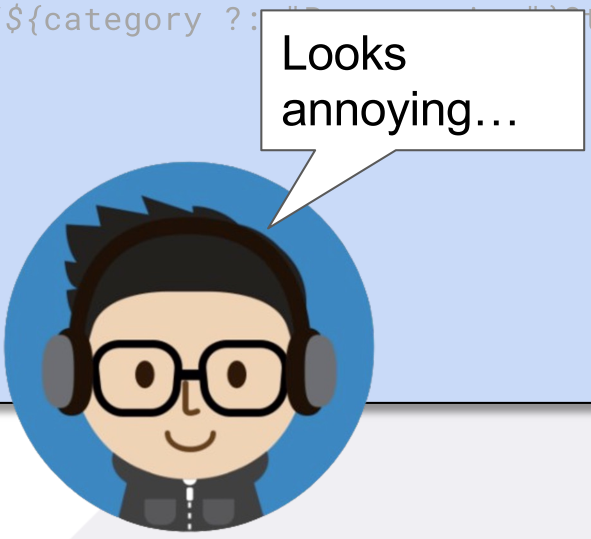
curl http://localhost:8080/jokes

```
{
   "joke": "Why do programmers prefer dark mode? Because light attracts bugs.",
   "lang": "en",
}
```

Dealing with open restriction I

```
@Configuration
open class Configuration {
   @Bean
   open fun restTemplate(): RestTemplate = RestTemplate()
}

@JsonIgnoreProperties
d                                                    ring)

@
c                                                    emplate,
                                                    val rootUri:String) {

 @GetMapping("/jokes")
 @ResponseBody
 fun randomJoke(@RequestParam("category") category: String?):Joke? =
    restTemplate.getForEntity<Joke>("$rootUri/${category ?:              type=single").body
}

@SpringBootApplication
open class JokesApplication

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

Proxied beans with CGLIB require open keyword, since Kotlin classes are `final` by default



Dealing with open restriction II

To avoid declaring all applicable beans as open the `kotlin-maven-allopen` `spring` compiler plugin can be used, which is recommended.

```
@Configuration
~~open~~ class Configuration {
   @Bean
   ~~open~~ fun restTemplate(): RestTemplate = RestTemplate()
}
```

pom.xml

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-plugin</artifactId>
      <configuration>
        <compilerPlugins>
          <plugin>spring</plugin>
        </compilerPlugins>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>org.jetbrains.kotlin</groupId>
          <artifactId>kotlin-maven-allopen</artifactId>
          <version>1.9.23</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
...
```

build.gradle

```
...
plugins {
    kotlin("plugin.allopen") version "1.9.23"
}
```

Xebia

Dependency Injection

```
@Configuration
class Configuration {
   @Bean
   fun restTemplate(): RestTemplate = RestTemplate()
}

@JsonIgnoreProperties
data class Joke(val joke:String, val lang:String)

@RestController
class JokeController(val restTemplate: RestTemplate,
                     @Value("\${root.uri}") val rootUri:String) {

 @GetMapping("/jokes")
 @ResponseBody
 fun randomJoke(@RequestParam("category") catego
    restTemplate.getForEntity<Joke>("$rootUri/${category ?: "Programming"}?type=single").body
}

@SpringBootApplication
class JokesApplication

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

Classes having only their primary constructor, the `@Autowire constructor` can be omitted.

Inject property values via the well-known `@Value` annotation.

Kotlin

Null-Safety

```kotlin
@Configuration
class Configuration {
   @Bean
   fun restTemplate(): RestTemplate = RestTemplate()
}

@JsonIgnoreProperties
data class Joke(val joke:String, val lang:String)

@RestController
class JokeController(val restTemplate: RestTemplate,
                    @Value("\${root.uri}") val rootUri:Stri

 @GetMapping("/jokes")
 @ResponseBody
 fun randomJoke(@RequestParam("category") category: String?):Joke? =
    restTemplate.getForEntity<Joke>("$rootUri/${category ?: "Programming"}?type=single").body
}

@Spring BootApplication
class JokesApplication

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

Null-Safety is fully supported. E.g. Nullable request params are not required. Conversely, non-nullable parameters are.

Extensions

Spring Boot automatically serializes data classes when the *Jackson Kotlin module* is on the classpath

```
@Configuratio
open class Co
   @Bean
   open fun restTemplate(): RestTemplate = RestTemplate()
}

@JsonIgnoreProperties
data class Joke(val joke:String, val lang:String)

@RestController
class JokeController(val restTemplate: RestTemplate,
                     @Value("\${root.uri}") val rootUri:String) {

 @GetMapping("/jokes")
 @ResponseBody
 fun randomJoke(@RequestParam("category") category: String?):Joke? =
    restTemplate.getForEntity<Joke>("$rootUri/${category ?: "Programming"}?type=single").body
}

@SpringBootApplication
class JokesApplication

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

Spring Boot offers a variety of handy extensions. See this list for all extensions.

Main class & method

```
@Configuration
class Configuration {
   @Bean
   fun restTemplate(): RestTemplate = RestTemplate()
}

@JsonIgnoreProperties
data class Joke(val joke:String, val lang:String)

@RestController
class JokeController(val restTemplate: RestTemplate,
                    @Value("\${root.uri}") val rootUri:String) {

 @GetMapping("/jokes")
 @ResponseBody
 fun randomJoke(@RequestParam("category") catego
    restTemplate.getForEntity<Joke>("$rootUri/${c                                    ody
}

@SpringBootApplication
class JokesApplication

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

The main method and Application class needs to be defined as follows:

Kotlin

Xebia

No more Spring annotations - a good idea?

```kotlin
@JsonIgnoreProperties
data class Joke(val joke: String, val lang: String)

class JokeHandler(val restTemplate: RestTemplate, val rootUri:String) {
    fun get(req: ServerRequest): ServerResponse {
        val category = req.param("category").orElse("Programming")
        return restTemplate.getForEntity<Joke>("$rootUri/$category?type=single").body?.let {
            ok().body(it)
        } ?: notFound().build()
    }
}

val appBeans = beans {
    bean<RestTemplate>()
    bean {
        val jokeHandler =  JokeHandler(ref(), env["root.uri"]!!)
        router {
            GET("/jokes", jokeHandler::get)
        }
    }
}

@SpringBootApplication
class JokesApplicationNg

fun main(args: Array<String>) {
    runApplication<JokesApplicationNg>(*args) {
        addInitializers(appBeans)
    }
}
```

Instead of an annotated controller we only declare (a) handler function(s) `ServerRequest -> ServerResponse`

Next we declare all beans manually using Kotlin's *bean definition DSL.*

`ref()` is used to refer to dependencies

All http routes are programmatically declared, passing a reference to the corresponding handler

Finally, the resulting beans are passed to the `addInitializer` method

Kotlin

ONLY ia

Make it persistent

```kotlin
@Entity
data class Joke(@Id @GeneratedValue val id:Long? = null, val joke:String, val rating:Int)

@Repository
class JokeRepository:JpaRepository<Joke, Long> {

     fun findAllByCategory(category:String): List<Joke>
}

@RestController
@RequestMapping("/jokes")
class JokeController(val jokeRepository: JokeRepository) {

     @GetMapping("{id}")
     @ResponseBody
     fun jokeById(@PathVariable("id") id:Long): Joke? =
        jokeRepository.findByIdOrNull(id)

     @PostMapping
     @ResponseBody
     fun insertJoke(@RequestBody joke: Joke): Joke =
        jokeRepository.save(joke.copy(rating = 0))
}

@SpringBootApplication
class JokesApplication

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

No-argument constructor

Jakarta Persistence Entities require a *no-argument constructor…* 🤔

```
@Entity
data class Joke(@Id @GeneratedValue val id:Long? = null, val joke:String, val rating:Int)

@Repository
class JokeRepository:JpaRepository<Joke, Long> {

     fun findAllByCategory(category:String): List<Joke>
}

@RestController
@RequestMapping("/jokes")
class JokeController(val jokeRepository: JokeRepository) {

     @GetMapping("{id}")
     @ResponseBody
     fun jokeById(@PathVariable("id") id:Long): Joke? =
        jokeRepository.findByIdOrNull(id)

     @PostMapping
     @ResponseBody
     fun insertJoke(@RequestBody joke: Joke): Joke =
        jokeRepository.save(joke.copy(rating = 0))
}
```

Kotlin

Xebia

Dealing with default constructor restriction

By using the `kotlin-maven-no-arg` `jpa` compiler plugin, a no-argument constructor will be generated automatically.

pom.xml

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-plugin</artifactId>
      <configuration>
        <compilerPlugins>
          <plugin>jpa</plugin>
        </compilerPlugins>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>org.jetbrains.kotlin</groupId>
          <artifactId>kotlin-maven-noarg</artifactId>
          <version>1.9.23</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
...
```

build.gradle

```
...
plugins {
    kotlin("plugin.noarg") version "1.9.23"
}
```

Repositories

```
@Entity
data class Joke(@Id @GeneratedValue val id:Long? = null, val joke:String, val rating:Int)

@Repository
class JokeRepository:JpaRepository<Joke, Long> {

     fun findFirstByCategory(category:String): Joke?
}

@RestController
@RequestMapping("/jok
class JokeController(

     @GetMapping("{id}")
     @ResponseBody
     fun jokeById(@PathVariable("id") id:Long): Joke? =
        jokeRepository.findByIdOrNull(id)

     @PostMapping
     @ResponseBody
     fun insertJoke(@RequestBody joke: Joke): Joke =
        jokeRepository.save(joke.copy(rating = 0))
}
```

On top of supporting the *CamelCase-to-Query* syntax, *Nullability* is supported too.

Repositories & Nullability

```
@Entity
data class Joke(@Id @GeneratedValue val id:Long? = null, val joke:String, val rating:Int)

@Repository
class JokeRepository:JpaRepository<Joke, Long> {

     fun findFirstByCategory(category:String): Joke?
}

@RestController
@RequestMapping("/jokes")
class JokeController(val jokeRepository: JokeRepository) {

     @GetMapping("{id}")
     @ResponseBody
     fun jokeById(@PathVariable("id") id:Long): Joke? =
        jokeRepository.findByIdOrNull(id)

     @PostMapping
     @ResponseBody
     fun insertJoke(@RequestBody joke: Joke): Joke =
        jokeRepository.save(joke.copy(rating = 0))
}
```

Default repository query, all have a …`OrNull()` version, especially for Kotlin!

Immutable Entities

```kotlin
@Entity
data class Joke(@Id @GeneratedValue val id:Long? = null, val joke:String, val rating:Int)

@Repository
class JokeRepository:

    fun findFirstByC
}

@RestController
@RequestMapping("/jokes")
class JokeController(val jokeRepository: JokeRepository) {

    @GetMapping("{id}")
    @ResponseBody
    fun jokeById(@PathVariable("id") id:Long): Joke? =
       jokeRepository.findByIdOrNull(id)

    @PostMapping
    @ResponseBody
    fun insertJoke(@RequestBody joke: Joke): Joke =
       jokeRepository.save(joke.copy(rating = 0))
}
```

Using `data` classes for entities are controversial, since certain corner-cases with linked entities can cause problems in generated `equals`, `hashCode` and `toString` method.

So, no convenient `copy(...)` method available? 🤔

Kotlin

Xebia

Mutable Entities

```
@Entity
class Joke(@Id @GeneratedValue val id:Long? = null, var joke:String, var rating:Int)

@Repository
class JokeRepository:JpaRepository<Joke, Long> {

     fun findFirstByCategory(category:String): Joke?
}

@RestController
@RequestMapping("/jokes")
class JokeController(val jokeRepository: JokeRepository) {

     @GetMapping("{id}")
     @ResponseBody
     fun jokeById(@PathVariable("id") id:Long): Joke? =
        jokeRepository.findByIdOrNull(id)

     @PostMapping
     @ResponseBody
     fun insertJoke(@RequestBody joke: Joke): Joke =
        jokeRepository.save(joke.apply{ rating = 0 })
}
```

Mutable entities…

… can still be treated very elegantly with `apply {...}`

Kotlin

Xebia

Testing in Spring Boot

```kotlin
import com.ninjasquad.springmockk.MockkBean

@SpringBootTest
@AutoConfigureMockMvc
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
class JokesControllerTest(val mapper: ObjectMapper,
                          val mockMvc: MockMvc,
                          @MockkBean
                          val restTemplate: RestTemplate) {

  @Test
  fun `should return a joke`() {
    val reply = Joke("A man walks into a bar. Ouch.", "en")
    every { restTemplate.getForEntity<Joke>(any<String>()) } returns ResponseEntity(reply, HttpStatus.OK)
     mockMvc.get("/jokes")
         .andExpect{ status().isOk }
         .andReturn.response.contentAsString.let {
             mapper.readValue<Joke>(it) shouldBe reply
         }
     verify {restTemplate.getForEntity<Joke>(any<String>()) }
  }
}
```

Define all dependencies in the constructor (requires `@TestConstructor` annotation)

Use backticks `` `my test method` `` for easy readable test methods

`Mockk` is a powerful mocking library designed for Kotlin.

Testing Spring Boot Applications

```kotlin
import com.ninjasquad.springmockk.MockkBean

@SpringBootTest
@AutoConfigureMockMvc
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
class JokesControllerTest(val mapper: ObjectMapper,
                          val mockMvc: MockMvc,
                          @MockkBean
                          val restTemplate: RestTemplate) {

 @Test
 fun `should return a joke`() {
  val reply = Joke("A man walks into a bar. Ouch.", "en")
   every { restTemplate.getForEntity<Joke>(any<String>()) } returns ResponseEntity(reply, HttpStatus.OK)
   mockMvc.get("/jokes")
       .andExpect{ status().isOk }
       .andReturn.response.contentAsString.let {
            mapper.readValue<Joke>(it) shouldBe reply
       }
   verify {restTemplate.getForEntity<Joke>(any<String>()) }
  }
}
```

So much code to deserialize the payload. Is there no better way?

Kotlin

Xebia

Testing Spring Boot Applications

```
import com.ninjasquad.springmockk.MockkBean

@SpringBootTest
@AutoConfigureMockMvc
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
class JokesControllerTest(val mapper: ObjectMapper,
                          val mockMvc: MockMvc,
                          @MockkBean
                          val restTemplate: RestTemplate) {

 @Test
 fun `should return a joke`() {
  val reply = Joke("A man walks into a bar. Ouch.", "en")
   every { restTemplate.getForEntity<Joke>(any<String>()) } returns ResponseEntity(reply, HttpStatus.OK)
   mockMvc.get("/jokes")
      .andExpect{ status().isOk }
      .andReturn().bodyAs<Joke>() shouldBe reply
   verify {restTemplate.getForEntity<Joke>(any<String>()) }
  }
}
```

…and off you go

Kotlin

```
inline fun <reified T> MvcResult.bodyAs() =
     mapper.readValue<T>(response.contentAsString)
```

Simply define an Extension…

Kotlin & Spring Boot Webflux

I use Spring Boot Webflux. Is there any **GAIN** when using Webflux with Kotlin rather than Java?

Jadev

Kotlin & Spring Boot Webflux

Xebia

Sequential Programming...

```
public URL randomAvatar() { ... }           //remote blocking method call
public Boolean verifyEmail(String name) { ... } //remote blocking method call
public User save(User user) {... }             //remote blocking db method call
```

```
@PostMapping("/users")
@ResponseBody
public User storeUser(@RequestBody User user) {
   var avatarUrl  = randomAvatar()
   var validEmail = verifyEmail(user.getEmail());
   if(!validEmail) {
       throw new InvalidEmailException("Invalid Email");
   }
   return save(UserBuilder.from(user).withAvatarUrl(avatarUrl).build());
}
```

😝 resource inefficient
😝 can get unresponsive
😝 no parallelism support

...is so easy!

Reactive Programming...

```
public Mono<Boolean> verifyEmail(String name) { ... } //remote async method call
public Mono<URL> randomAvatar() { ... } //remote async method call
public Mono<User> save(User user) {... } //remote async db method call
```

```
@PostMapping("/users")
@ResponseBody
public Mono<User> storeUser(@RequestBody User user) {
  Mono<URL> avatarMono = avatarService.randomAvatar();
  Mono<Boolean> validEmailMono = emailService.verifyEmail(user.getEmail());
  return Mono.zip(avatarMono, validEmailMono).flatMap(tuple ->
   if(!tuple.getT2()) //what is getT2()? It's the validEmail Boolean...
       Mono.error(new InvalidEmailException("Invalid Email"));
   else personRepo.save(UserBuilder.from(user)
                                   .withAvatarUrl(tuple.getT1()));
  );
}
```

😃 resource efficient
😃 supports parallelism
😃 responsive

Reactive Programming: With great *Power* comes great *Pain*

```java
public Mono<Boolean> verifyEmail(String name) { ... } //remote async method call
public Mono<URL> randomAvatar() { ... } //remote async method call
public Mono<User> save(User user) {... } //remote async db method call
```

😝 Every domain object must be wrapped in a reactive building block

⚠️ limited to non-blocking libraries (WebClient, R2DBC)

😝 complex operators everywhere

```java
 @PostMapping("/users")
@ResponseBody
 public Mono<User> storeUser(@RequestBody User user) {
  Mono<URL> avatarMono = avatarService.randomAvatar();
  Mono<Boolean> validEmailMono = emailService.verifyEmail(user.getEmail());
  return Mono.zip(avatarMono, validEmailMono).flatMap(tuple ->
   if(!tuple.getT2()) //what is getT2()? It's the validEmail Boolean...
       Mono.error(new InvalidEmailException("Invalid Email"));
   else personRepo.save(UserBuilder.from(user)
                                  .withAvatarUrl(tuple.getT1()));
  );
}
```

😝 certain standard programming constructs cannot be used - e.g. throwing Exceptions

The *business intent* of my code gets **lost** in all the 'combinator jungle' - and it's hard to learn too! ☹️

Xebia

Reactive Programming to the rescue?

yes and no:

reactive gets the job done but:
***accidental complexity** is enormous*

The real answer? Coroutines & Spring Boot

Kotlin Coroutines to the rescue!

Kotlin has built-in concurrency support that are based on *Coroutines*.

With Coroutines, logic can be expressed *sequentially* whereas the underlying implementation figures out the *asynchrony*.

A method marked `suspend` can be run *within a coroutine* that can suspend it without blocking a Thread

```
suspend fun randomAvatar(): URL = ...
suspend fun verifyEmail(email:String): Boolean = ...
suspend fun save(user:User): Long = …
```

Remote Service Calls with Reactor & Coroutines

Spring's `WebClient` used for remote non-blocking REST calls, is based on `Mono<T>`

```java
@Component
public class AvatarService {

    public Mono<URL> randomAvatar() {
        return WebClient.create("http://<host>")
                                              .get()
                                         .uri("/avatar")
                                         .retrieve()
}
```

```kotlin
import org.springframework.web.reactive.function.client.awaitBody

@Component
class AvatarService {

   suspend fun randomAvatar(): URL = WebClient.create("http://<host>")
                       .get()
                       .uri("/avatar")
                       .retrieve()
                       .awaitBody<URL>()
```

With Coroutines simply mark remote service calls methods with `suspend`.

...and use one of the 'glue methods' `await...` that turn a `Mono<T>` into a suspended call. And gone is the `Mono<T>` abstraction 😃!

ONLY

Database Access with Reactor & Coroutines

Spring's reactive repositories rely on `Mono<T>`'s for single result repository calls.

```
@Repository
interface UserDao extends ReactiveCrudRepository<User, Long> {

    public Mono<User> findByUserName(String userName);
}
```

Java

With Coroutines extend repositories from `org.springframework.data.repository.kotlin.CoroutineCrudRepository`

We can also make return types safer by introducing nullability 👍👍.

```
@Repository
interface UserDao : CoroutineCrudRepository<User, Long> {

  suspend fun findByUserName(userName: String) : User?

}
```

Kotlin ONLY

Mark additional queries with `suspend`.

To define queries use spring-data's common naming syntax or `@Query` annotations

Xebia

Webflux & Coroutines in Action

```
dependencies {
 implementation("org.springframework.boot:spring-boot-starter-webflux:${spring.boot.version}")
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:${kotlinx.version}")
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-reactor:${kotlinx.version}")
}
```

```
@PostMapping("/users")
@ResponseBody
public Mono<User> storeUser(@RequestBody User user) {

  Mono<URL> avatarMono = avatarService.randomAvatar();

  Mono<Boolean> validEmailMono = emailService.verifyEmail(user.getEmail());

  return Mono.zip(avatarMono, validEmailMono).flatMap(tuple ->

   if(!tuple.getT2()) //what is getT2()? It's the validEmail Boolean...
       Mono.error(new InvalidEmailException("Invalid Email"));
   else personRepo.save(UserBuilder.from(user)
                                 .withAvatarUrl(tuple.getT1()));
  );
}
```

IMPROVE
IMPROVE
IMPROVE

Xebia

Kotlin & Spring Boot Webflux

Looks good.
But in Java we now have *VirtualThreads.*
Will they not ***solve all*** these problems?

Jadev

Short answer: No (only one, to be precise)

Long answer: *Watch my JetBrains webinars:*

JET BRAINS
November 15, 2023
15:00 UTC
Headache-Free Reactive Programming with Spring Boot and Kotlin Coroutines
Urs Peter

JET BRAINS
February 07, 2024
16:00–17:00 UTC
Reactive Spring Boot With Kotlin Coroutines: Adding Virtual Threads
Urs Peter
Senior Consultant at Xebia
1:23:41

https://www.youtube.com/watch?v=ahTXElHrV0c

https://www.youtube.com/watch?v=szl3eWA0VRw

Practical Answer:

A) VirtualThreads are on the *JVM*, so *all* JVM languages (Java, Kotlin, Scala etc.) can use VirtualThreads rather than Java only.

B) VirtualThreads will rather *complement* Coroutines (and reactive frameworks in general) than replace them.

Virtual Thread usage in Spring Boot Web/Webflux

Prerequisite:

- Spring Boot 3.2+
- JDK 21+

Configuration:

application.properties/.yaml

```
spring.threads.virtual.enabled=true
```

However, for Webflux applications, this generally won’t make a difference.

Limitations of VirtualThreads

1. When using *async libraries only* (`WebClient`, `R2DBC` etc.) - as you should for reactive applications - `VirtualThreads` won't add any value at all but only overhead.

```
suspend fun randomAvatar(): URL = ... //remote async method call
suspend fun verifyEmail(email:String): Boolean = ... //remote async method call
suspend fun save(user:User): Long = … //remote async db method call
```

spring WebFlux + Kotlin Coroutines

```
@RestController
class PersonController {

   @GetMapping("/users")
   @ResponseBody
   @Transactional
   suspend fun storeUser(@RequestBody user:User): User = coroutineScope {
      val avatarUrl = async { avatarService.randomAvatar() }
      val validEmail = async { emailService.verifyEmail() }
         if(!validEmail.await()) throw InvalidEmailException("Invalid
email")
         personRepo.save(user.copy(avatar = avatarUrl.await()))
}
```

Kotlin ONLY

`VirtualThreads` have no API for *parallelism*. For parallelism, *Structured Concurrency* is required, which Coroutines offer out of the box (`async`, `await` etc.)

Problem: Blocking code & Coroutines/Reactive

However, if you *have to use* a blocking API…

```
fun randomAvatarBlocking(): URL = ... //remote blocking method call
fun verifyEmailBlocking(email:String): Boolean = ... //remote blocking method call
suspend fun save(user:User): Long = … //remote async db method call
```

```
@RestController
class PersonController {

   @GetMapping("/users")
   @ResponseBody
   @Transactional
   suspend fun storeUser(@RequestBody user:User):User = withContext(Dispatchers.IO) {
      val avatarUrl = async { avatarService.randomAvatarBlocking() }
      val validEmail = async { emailService.verifyEmailBlocking() }
         if(!validEmail.await()) throw InvalidEmailException("Invalid email")
         personRepo.save(user.copy(avatar = avatarUrl.await()))
}
```

However, this separate ThreadPool can get *exhausted*, possibly causing *performance degradation* 😬

… you have to have a *separate* ThreadPool (`Dispatchers.IO / Schedulers.boundedElastic()`), with spare Threads that can be blocked.

The winning formula: VirtualThreads *with* Coroutines/Reactive

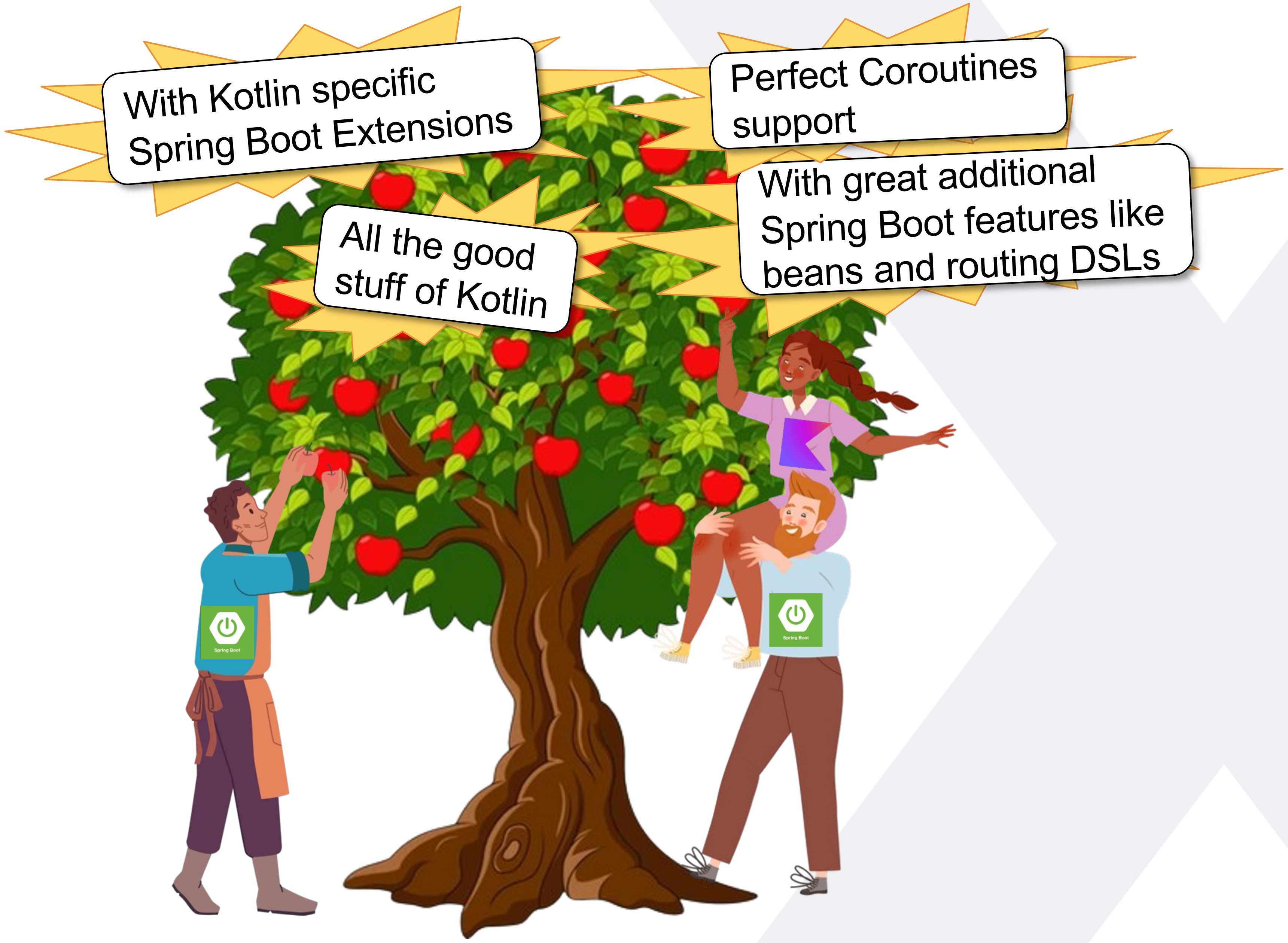
```
fun randomAvatarBlocking(): URL = ... //remote blocking method call
fun verifyEmailBlocking(email:String): Boolean = ... //remote blocking method call
suspend fun save(user:User): Long = … //remote async db method call
```

```
@RestController
class PersonController {

   @GetMapping("/users")
   @ResponseBody
   @Transactional
   suspend fun storeUser(@RequestBody user:User):User = withContext(Dispatchers.VT) {
      val avatarUrl = async { avatarService.randomAvatarBlocking() }
      val validEmail = async { emailService.verifyEmailBlocking() }
         if(!validEmail.await()) throw InvalidEmailException("Invalid email")
         personRepo.save(user.copy(avatar = avatarUrl.await()))
}
```

Using a VirtualThread Dispatcher, *blocking* IO code does not block a PlatformThread, so performance degradation is impossible ✅.

```
val Dispatchers.VT: CoroutineDispatcher
   get() = Executors.newVirtualThreadPerTaskExecutor().asCoroutineDispatcher()
```

Kotlin & Spring Boot: Pain or Gain?

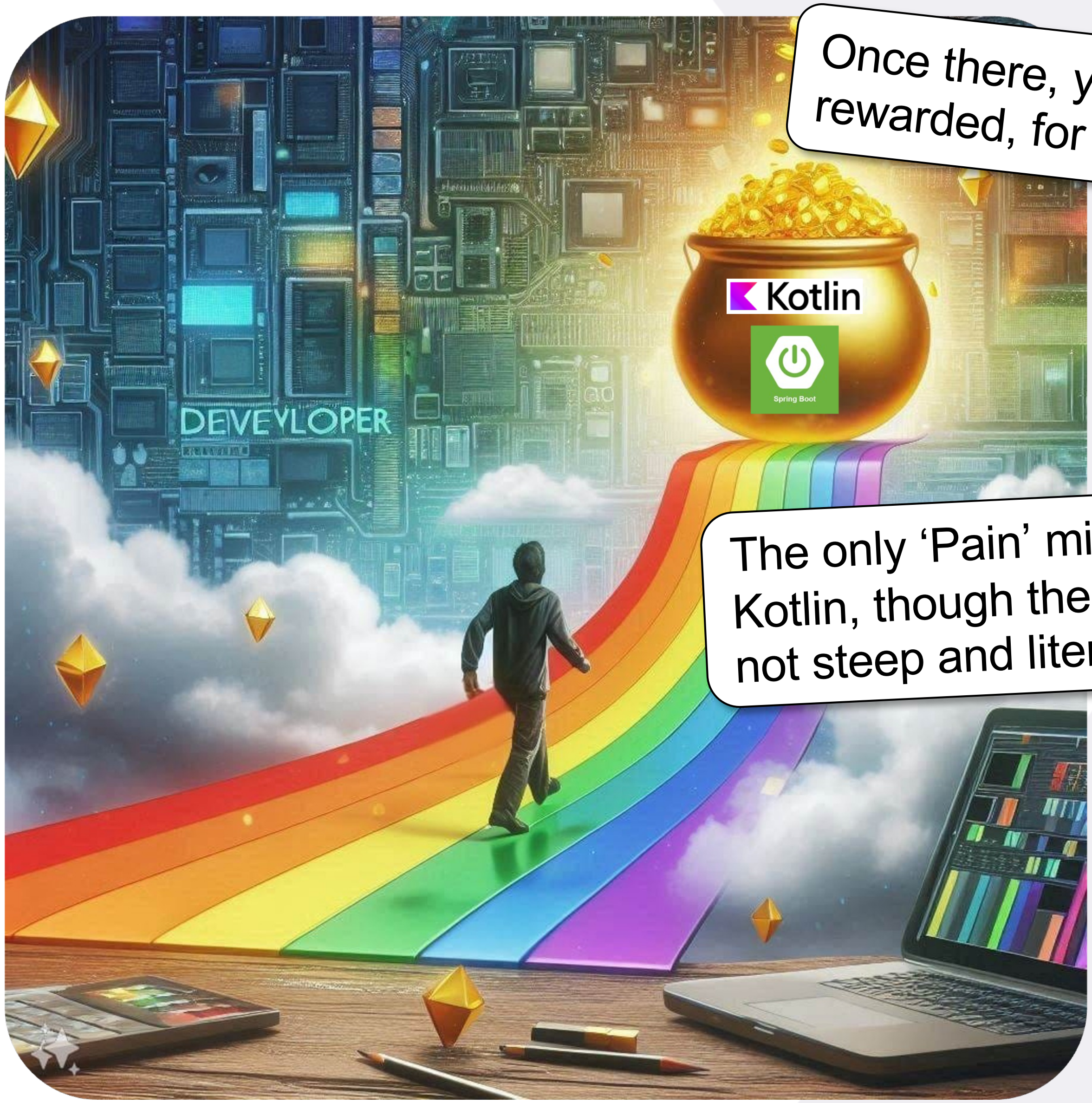

With Kotlin specific Spring Boot Extensions
Perfect Coroutines support
With great additional Spring Boot features like beans and routing DSLs
All the good stuff of Kotlin
Spring Boot
Spring Boot

Kotlin & Spring Boot: Where is the Pain?

Jadev

Yes, ~1-2 years ago, there was still a bit Pain

…but by now, all Kotlin issues are resolved!



Once there, you will be rewarded, for sure 🏆😃🏆!

The only 'Pain' might be learning Kotlin, though the learning curve is not steep and literally `fun`!

Thank you!

!

?

I wish you a lot of **fun** 😃

Urs Peter

LinkedIn:	bit.ly/urs-peter-linked-in
Email:	upeter@xebia.com
Blog:	xebia.com/blog/
Courses:	xebia.com/academy/en/upskilling/kotlin-academy/

Kotlin Training
Certified by JetBrains

Xebia Academy