QCon SÃO PAULO

Zup

Kotlin in Microservices using DDD, Event Sourcing & CQRS

Augusto Branquinho
Mike Shigueru Matsumoto

1. Real Wave

Modules

- Marketing
- Sales
- Care
- Payments
- Insights
- Integration

Technical stack

2. Kotlin

- Designed by Jetbrains
- Kotlin in production: Google, Pivotal, Uber, Atlassian, Coursera, etc
- Statically typed
- Modern
- Drastically reduce the boilerplate
- Java, Android
- Javascript
- Native

2. Kotlin

- Data Class
  - toString
  - equals
  - hashCode
  - copy
  - properties - get/set

```
data class UserCreatedEvent(
        val id: Long,
        var name: String,
        val state: State = State.NEW
)
```

```
val event = UserCreatedEvent(name = "User", id = 1L)
```

Lambda

Java

```
IntStream.rangeClosed(1, 100)
         .filter((i) → i % 2 == 0)
         .map((even) → even * 2)
         .boxed()
         .collect(Collectors.toList());
```

Kotlin

```
(1..100).filter { it % 2 == 0 }
        .map { it * 2 }
```

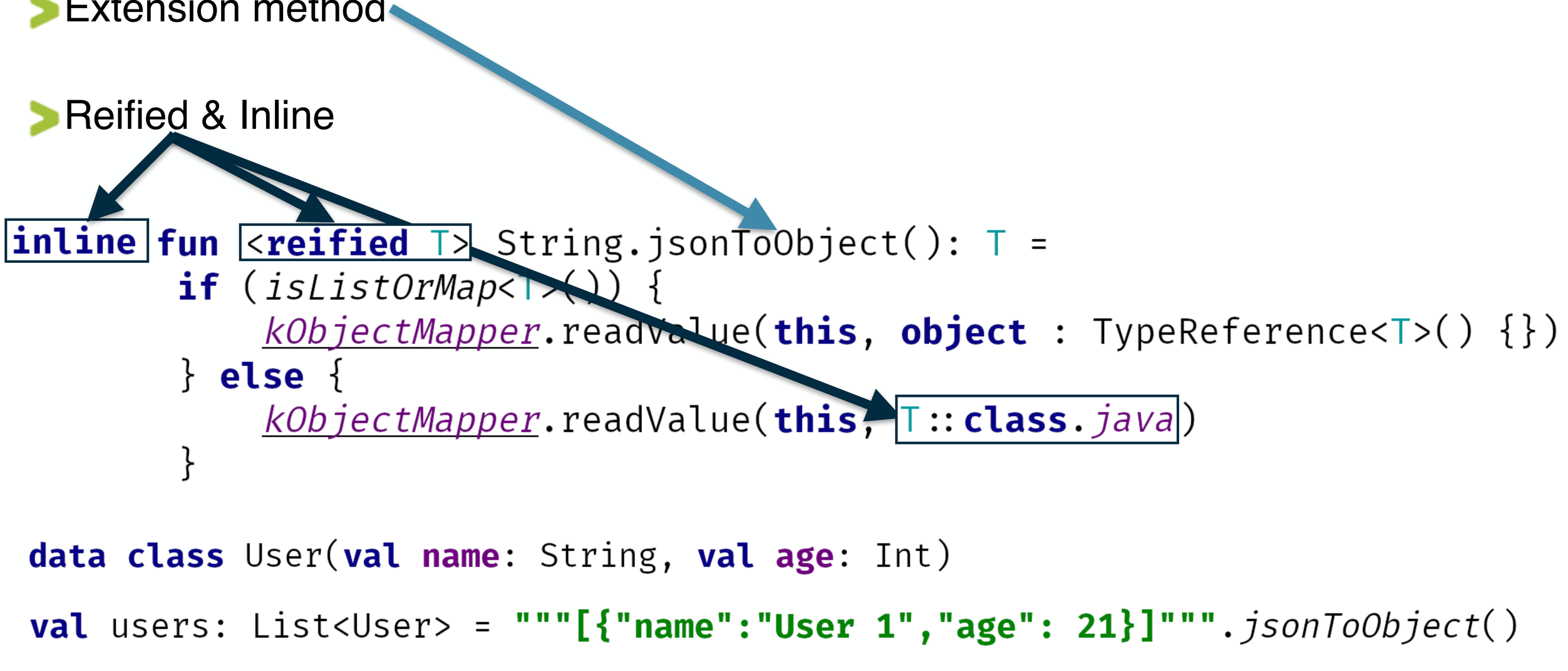
Higher Order Functions

```
fun higherOrder(age: Int = 0, op: (age: Int) → Int): Int =
        op(age)
```

```
higherOrder( age: 10, { age → age * 2})

higherOrder( age: 10, { it * 2 })

higherOrder( age: 10) { it * 2 }
```

2. Kotlin

- Extension method
- Reified & Inline



```
inline fun <reified T> String.jsonToObject(): T =
        if (isListOrMap<T>()) {
            kObjectMapper.readValue(this, object : TypeReference<T>() {})
        } else {
            kObjectMapper.readValue(this, T::class.java)
        }

data class User(val name: String, val age: Int)

val users: List<User> = """[{"name":"User 1","age": 21}]""".jsonToObject()
```

2. Kotlin

- Smart cast
- Pattern matching
- String template
- Reified
- Operator overloading
- Unit and Nothing
- High order function
- Infix
- Internal
- Properties
- Extension methods/properties
- Lazy initialization, Delegation
- Object (singleton)
- Deprecated
- Default value
- Tacit programming: let, also, apply, run, with
- Tailrec
- …

3. Event Sourcing & CQRS

- Event Sourcing (ES)
  - 100% reliable logging (aka: version control)
  - all operations on the system are stream event-oriented: nothing happens before the event being committed
  - it is possible to rebuild all the views since the beginning of time
  - you can build new views using the same principle
  - **events are immutable**

3. Event Sourcing & CQRS

- CQRS
  - "It's a pattern" (Greg Young)
  - uses a different model to update (Command) and another to read (Query)
  - Command *(accept/reject)*
    - Change state
    - Events
  - Query (*views*)
    - Are free of side effects

3. Event Sourcing & CQRS

- Motivation
  - ES, CQRS
    - Clients: Bank, Telecom
    - Audit Log (query the past)
    - Insights
    - Multiple views (Multi-Tenant)

  - E.g.: Catalog manager
    - Reliable logs
    - Offers are immutable
    - High requests rate (millions)
    - Decoupled views

3. Event Sourcing & CQRS

accept/reject

Real Wave Microservices

Command Api

Query Api

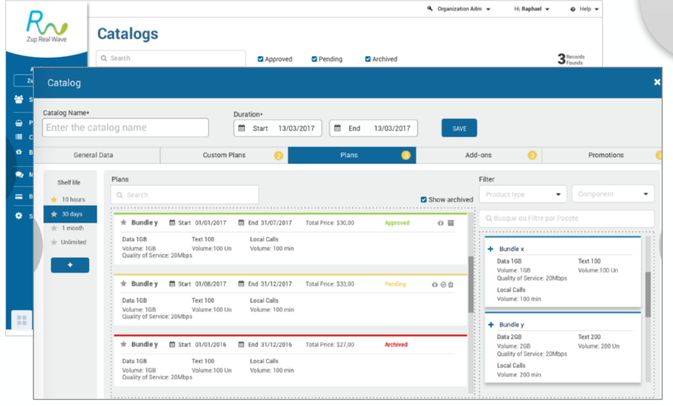
Catalog Manager

Catalog Search

Catalog Processor

query response

Offers



Join Use Pay

3. Event Sourcing & CQRS

Command API

Commands (accept/reject)

Command Handlers

Domain

Event Store Repositories

Catalog Manager

Event Store Database

Events

Events

Query API

Query (response)

Query Layer

Catalog Search Views

PostgreSQL

Elastic Search

JDBC Repositories

Elastic Repositories

Event Handler

Catalog Processor

Low write latency

High read latency

3. Event Sourcing & CQRS

Command API

Commands (accept/reject)

Command Handlers

Domain

Event Store Repositories

Event Store Database

Events

Events

Event Handler

Query API

Query (response)

Query Layer

Views

PostgreSQL

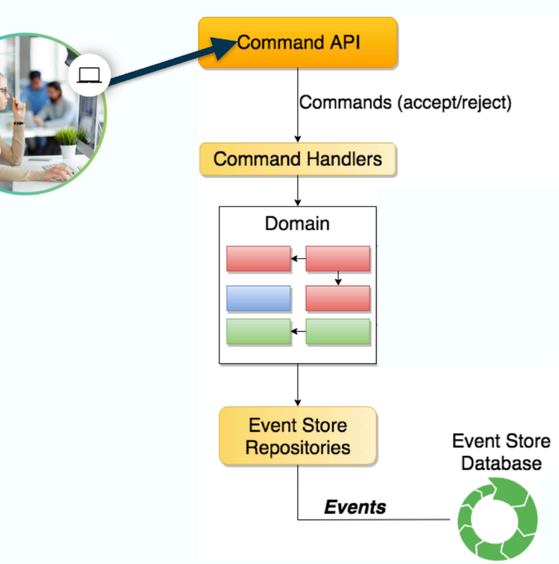
Elastic Search

JDBC Repositories

Elastic Repositories

> Code

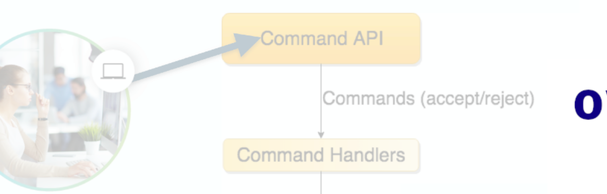
3. Event Sourcing & CQRS



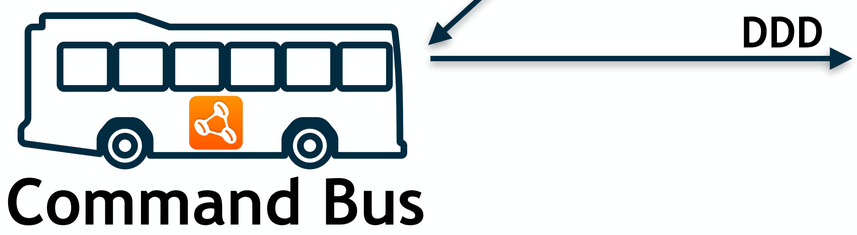

```
@RestController
@RequestMapping( ...value: "/catalogs")
class CatalogController ( ... )
```

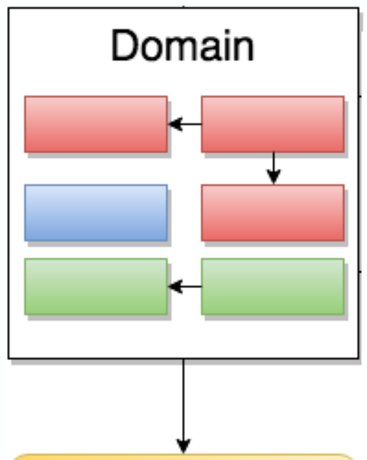
```
@PostMapping
@ResponseStatus(CREATED)
fun createCatalog(@Valid @RequestBody
                  request: CreateCatalogRequest) =
        dispatch(CreateCatalogCommand(CatalogId(),
                                      request.name,
                                      request.availability,
                                      request.channelIds,
                                      request.profileIds))
```

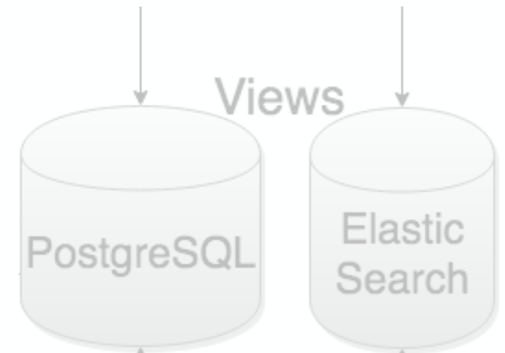
3. Event Sourcing & CQRS



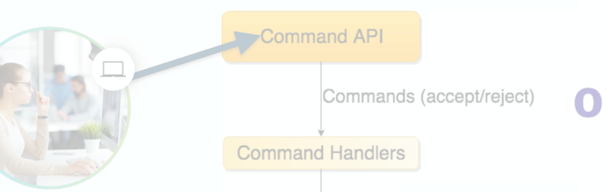

```
override fun create(command: CreateCatalogCommand): CatalogId =
        validate(command)
                .let { commandGateway.sendAndWait(it) }
```





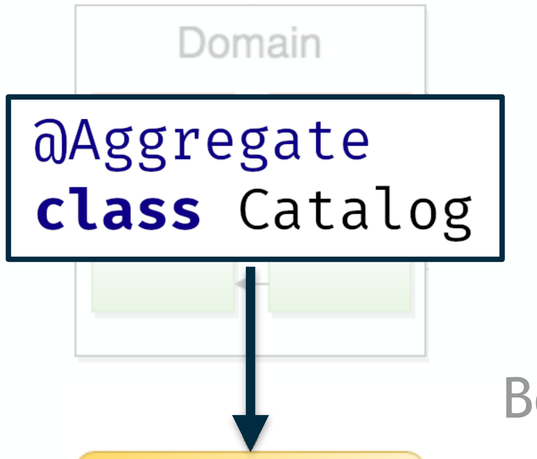


3. Event Sourcing & CQRS




```
override fun create(command: CreateCatalogCommand): CatalogId =
        validate(command)
                .let { commandGateway.sendAndWait(it) }
```

Command Bus

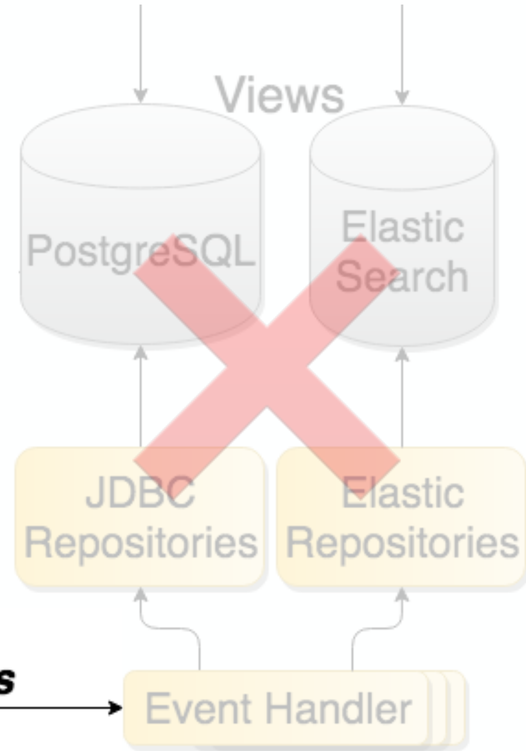


Bounded Context

Event Store Database

Events

Events



3. Event Sourcing & CQRS

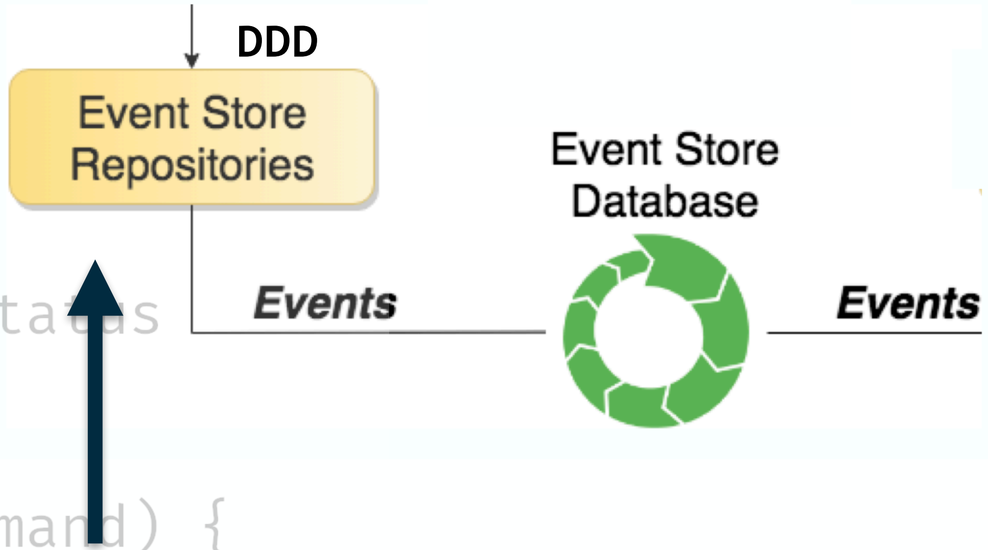

```
@Aggregate
class Catalog {

    @AggregateIdentifier
    private lateinit var id: CatalogId
    private lateinit var status: CatalogStatus

    @CommandHandler
    constructor(command: CreateCatalogCommand) {
        apply(CatalogCreatedEvent(
                command.id,
                command.name,
                command.availability,
                command.channelIds,
                command.profileIds))
    }
```

First Command

Nothing was replayed yet

3. Event Sourcing & CQRS

```
@Aggregate
class Catalog {

    @AggregateIdentifier
    private lateinit var id: CatalogId
    private lateinit var status: CatalogStatus

    @CommandHandler
    constructor(command: CreateCatalogCommand) {
        apply(CatalogCreatedEvent(command.id, /* ... */
    }

    @EventSourcingHandler
    fun on(event: CatalogCreatedEvent) {
        this.id = event.id
        this.status = event.status
    }
```



3. Event Sourcing & CQRS

```
@Aggregate
class Catalog {

    @AggregateIdentifier
    private lateinit var id: CatalogId
    private lateinit var status: Catalog

    @CommandHandler
    constructor(command: CreateCatalogC
        apply(CatalogCreatedEvent(commar
    }

    @EventSourcingHandler
    fun on(event: CatalogCreatedEvent) {
        this.id = event.id
        this.status = event.status
    }
```

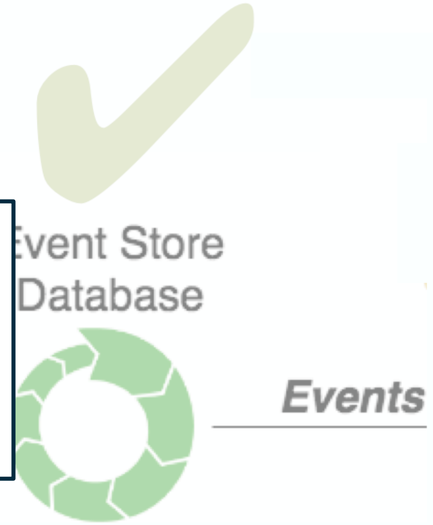
Event ID	METADATA	TYPE	PAYLOAD
1	17:00:00	CatalogCreatedEvent	{ "id": "1xx", "status": "PENDING", "name": "Mother's Day", "availability": { /*...*/ } }

3. Event Sourcing & CQRS

```
@Aggregate
class Catalog {

    @AggregateIdentifier
    private lateinit var id:
    private lateinit var st

    @CommandHandler
    constructor(command: CreateCatalogCommand) {
        apply(CatalogCreatedEvent(command.id, /* ... */
    }

    @EventSourcingHandler
    fun on(event: CatalogCreatedEvent) {
        this.id = event.id
        this.status = event.status
    }
```

Happens in the past

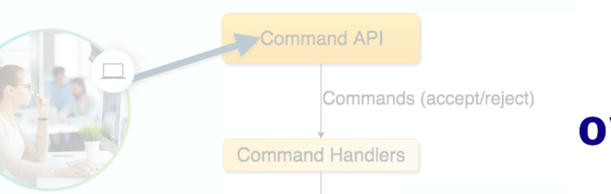
Just accept, no logic here



Event ID	METADATA	TYPE	PAYLOAD
1	17:00:00	CatalogCreatedEvent	{ "id": "1xx", "status": "PENDING", "name": "Mother's Day", "availability": { /*...*/ } }

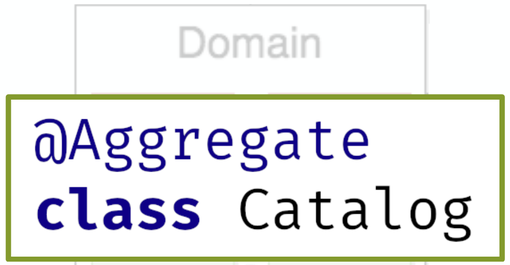
CatalogStatus.PENDING

3. Event Sourcing & CQRS




```
override fun approve(command: ApproveCatalogCommand): CatalogId =
    validate(command)
            .let { commandGateway.sendAndWait(it) }
```

Command Bus



3. Event Sourcing & CQRS

Command Bus

```
override fun approve(command: ApproveCatalogCommand): CatalogId =
    validate(command)
        .let { commandGateway.sendAndWait(it) }
```

```
data class ApproveCatalogCommand(
        @field:[TargetAggregateIdentifier NotNull Valid]
        val id: CatalogId) {
```

⚠️ **Replay for all past events will start for this ID before this Command executes**

3. Event Sourcing & CQRS

```
@Aggregate
class Catalog

    @EventSourcingHandler
    fun on(event: CatalogCreatedEvent) {
        this.id = event.id
        this.status = event.status
    }

    @CommandHandler
    fun approve(command: ApproveCatalog

    @EventSourcingHandler
    fun on(event: CatalogApprovedEvent) {
        this.status = event.status
    }
```

1 - Replay

CatalogStatus.PENDING

Event ID	METADATA	TYPE	PAYLOAD
1	17:00:00	CatalogCreatedEvent	{ "id": "1xx", "status": "PENDING", "name": "Mother's Day", "availability": { /*...*/ } }

3. Event Sourcing & CQRS

Event ID	METADATA	TYPE	PAYLOAD
1	17:00:00	CatalogCreatedEvent	{ "id": "1xx", "status": "PENDING", "name": "Mother's Day", "availability": { /*...*/ } }

```
@Aggregate
class Catalog

    @EventSourcingHandler
    fun on(event: CatalogCreatedEvent) {
        this.id = event.id
        this.status = event.status
    }

    @CommandHandler
    fun approve(command: ApproveCatalogCommand)

    @EventSourcingHandler
    fun on(event: CatalogApp
        this.status = event.
    }
```

1 - Replay

2 - Accepts/Rejects

```
if (this.status ≠ CatalogStatus.PENDING) {
    throw BusinessException
            .of(CmErrorCode.ILLEGAL_COMMAND_EXCEPTION)
}

apply(CatalogApprovedEvent(command.id))
```

3. Event Sourcing & CQRS

Event ID	METADATA	TYPE	PAYLOAD
1	17:00:00	CatalogCreatedEvent	{ "id": "1xx", "status": "PENDING", "name": "Mother's Day", "availability": { /*...*/ } }
2	17:05:00	CatalogApprovedEvent	{ "id": "1xx", "status": "APPROVED" }

```
@Aggregate
class Catalog

    @EventSourcingHandler
    fun on(event: CatalogCreatedEvent)
        this.id = event.id
        this.status = event.status
    }

    @CommandHandler
    fun approve(command: ApproveCatalogCommand) { ... }

    @EventSourcingHandler
    fun on(event: CatalogApprovedEvent) {
        this.status = event.status
    }
```

2 - Accepts/Rejects

3 - Accepted

3. Event Sourcing & CQRS

Command API

Commands (accept/reject)

Command Handlers

Domain

Event Store Repositories

Events

Event Store Database

Events

Event Handler

Query API

Query (response)

Query Layer

Views

PostgreSQL

Elastic Search

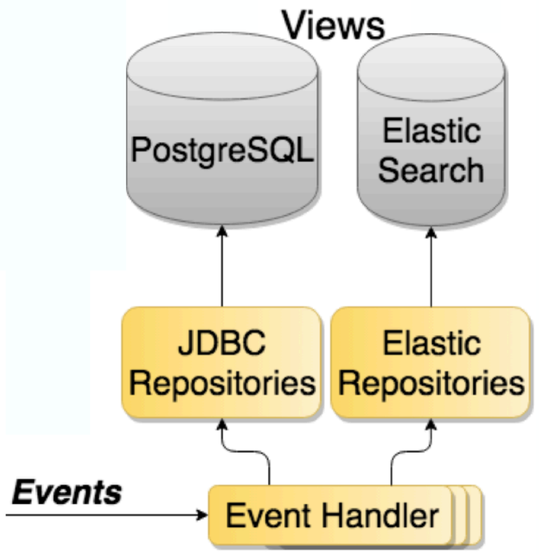
JDBC Repositories

Elastic Repositories

Catalog Processor

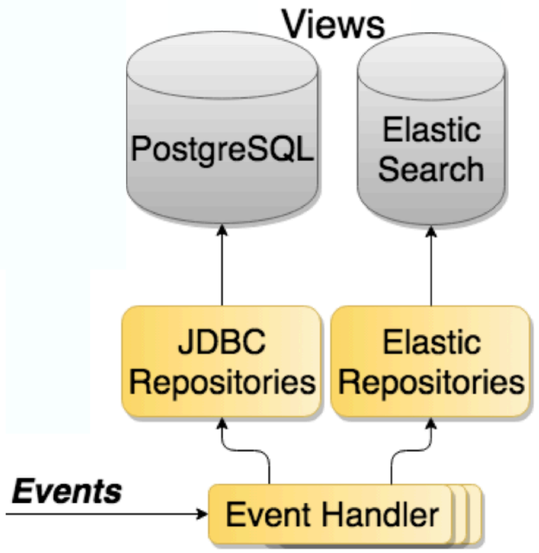
🤔 *Another team maybe?*

3. Event Sourcing & CQRS



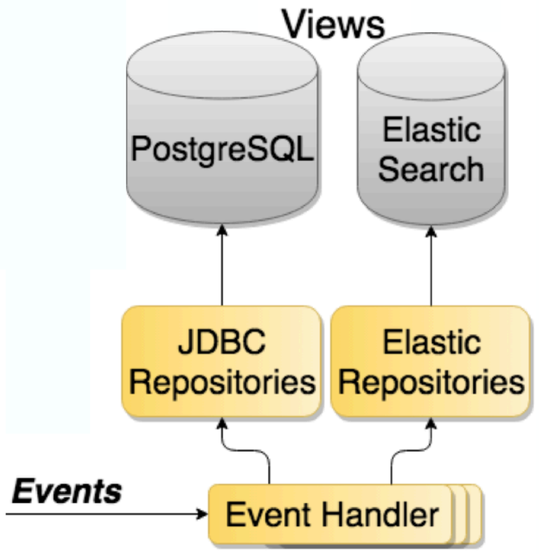

```
@Component( value: "jpaCatalogListener")
open class CatalogEventListener ( ... ) {

    @EventHandler
    fun on(event: CatalogCreatedEvent): Catalog {
        LOG.info("CatalogCreatedEvent received {}", event)
        return Catalog.create(
                repositoryFactory, validator,
                event.id,
                event.name,
                event.availability,
                event.channelIds,
                event.profileIds,
                event.user)
            .also { LOG.info("Catalog $it JPA created!") }
    }
```

3. Event Sourcing & CQRS




```
@Component( value: "elasticCatalogListener")
open class CatalogEventListener ( ... ) {

    @EventHandler
    fun on(event: CatalogCreatedEvent): Catalog { ... }

    @EventHandler
    fun on(event: CatalogUpdatedEvent) { ... }
```

3. Event Sourcing & CQRS




```kotlin
@Component( value: "kafkaCatalogListener")
open class CatalogEventListener ( ... ) {

    @EventHandler
    fun on(event: CatalogCreatedEvent): Catalog { ... }

    @EventHandler
    fun on(event: CatalogUpdatedEvent) { ... }

    @EventHandler
    fun on(event: CatalogDeletedEvent) { ... }
```

3. Event Sourcing & CQRS

Command API

Commands (accept/reject)

Command Handlers

Domain

Event Store Repositories

Event Store Database

Events

Events

Event Handler

JDBC Repositories

Elastic Repositories

Query API

Query (response)

Query Layer

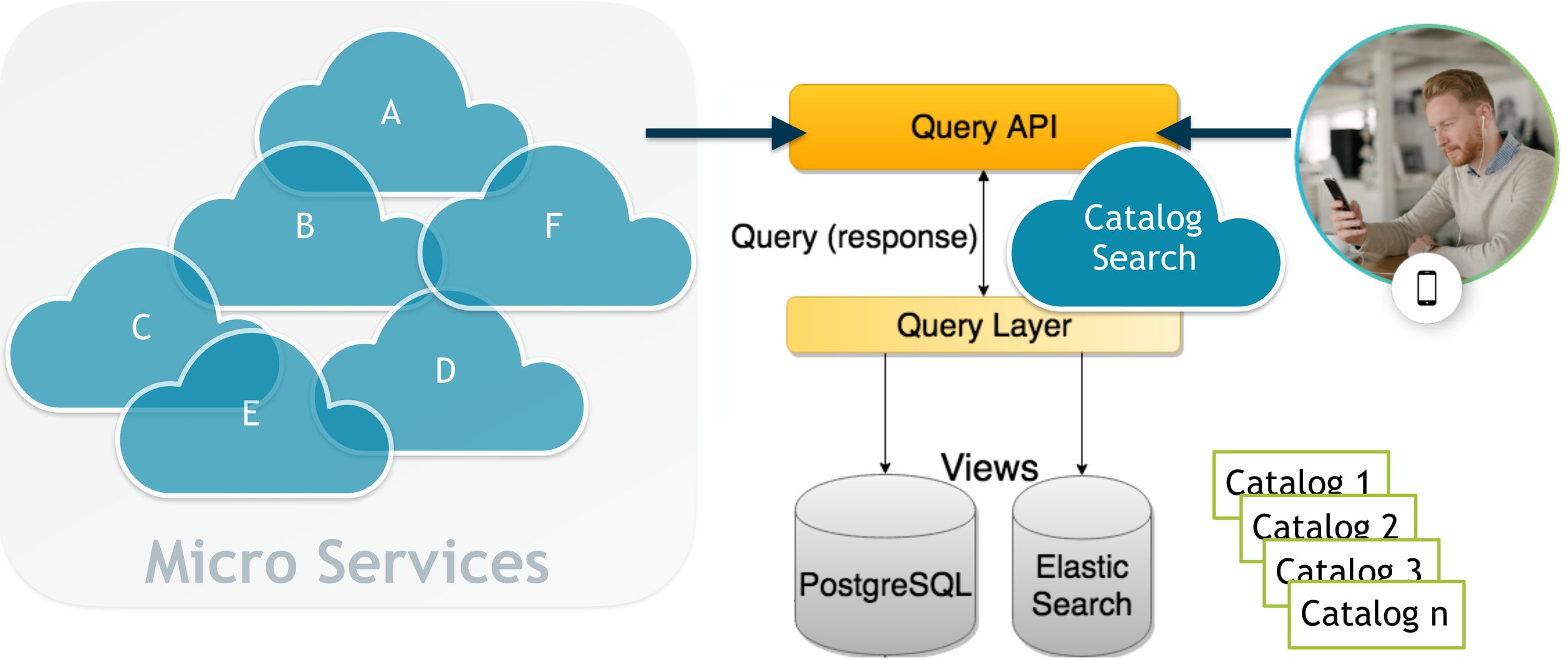
Views

PostgreSQL

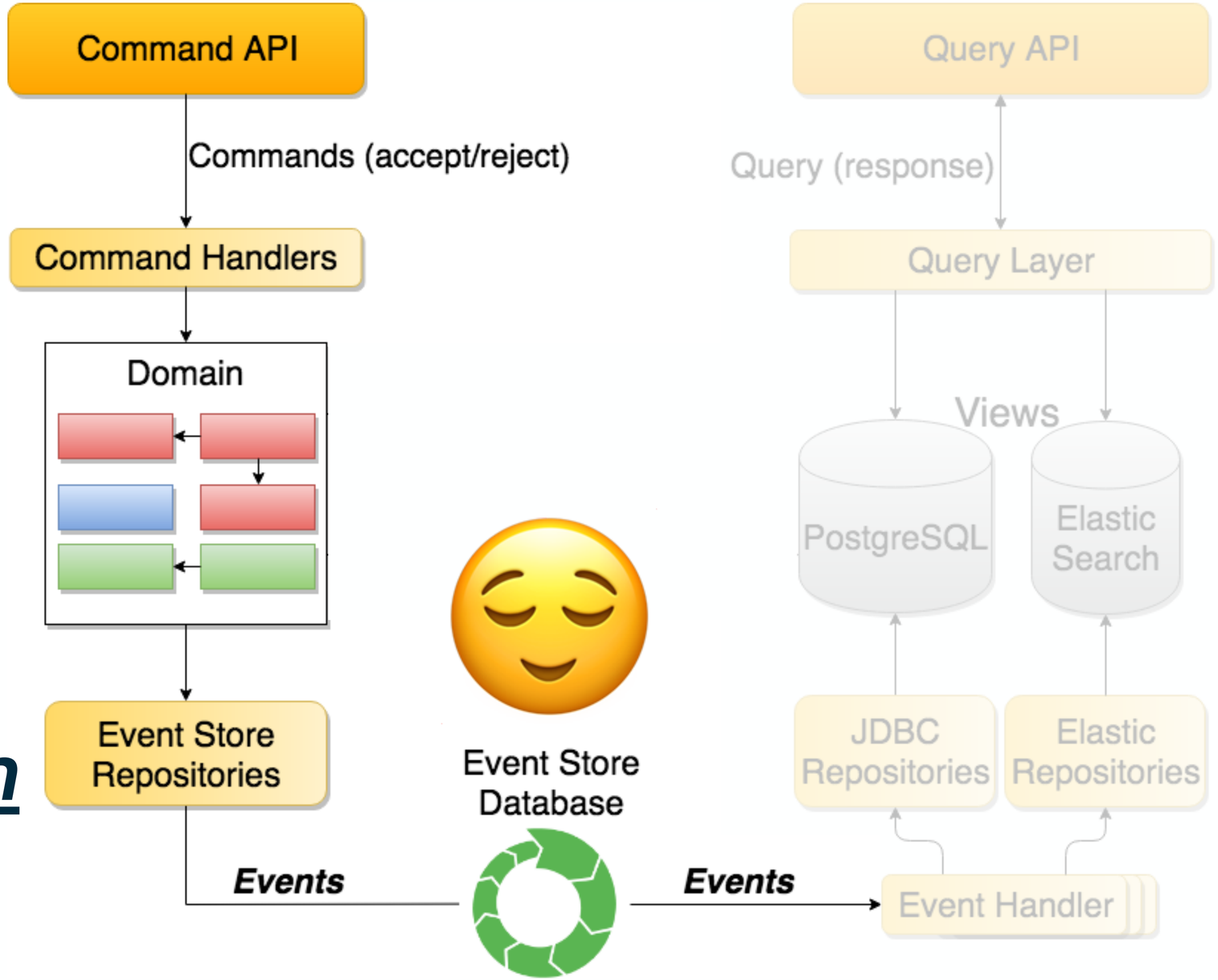
Elastic Search

Catalog Search

3. Event Sourcing & CQRS

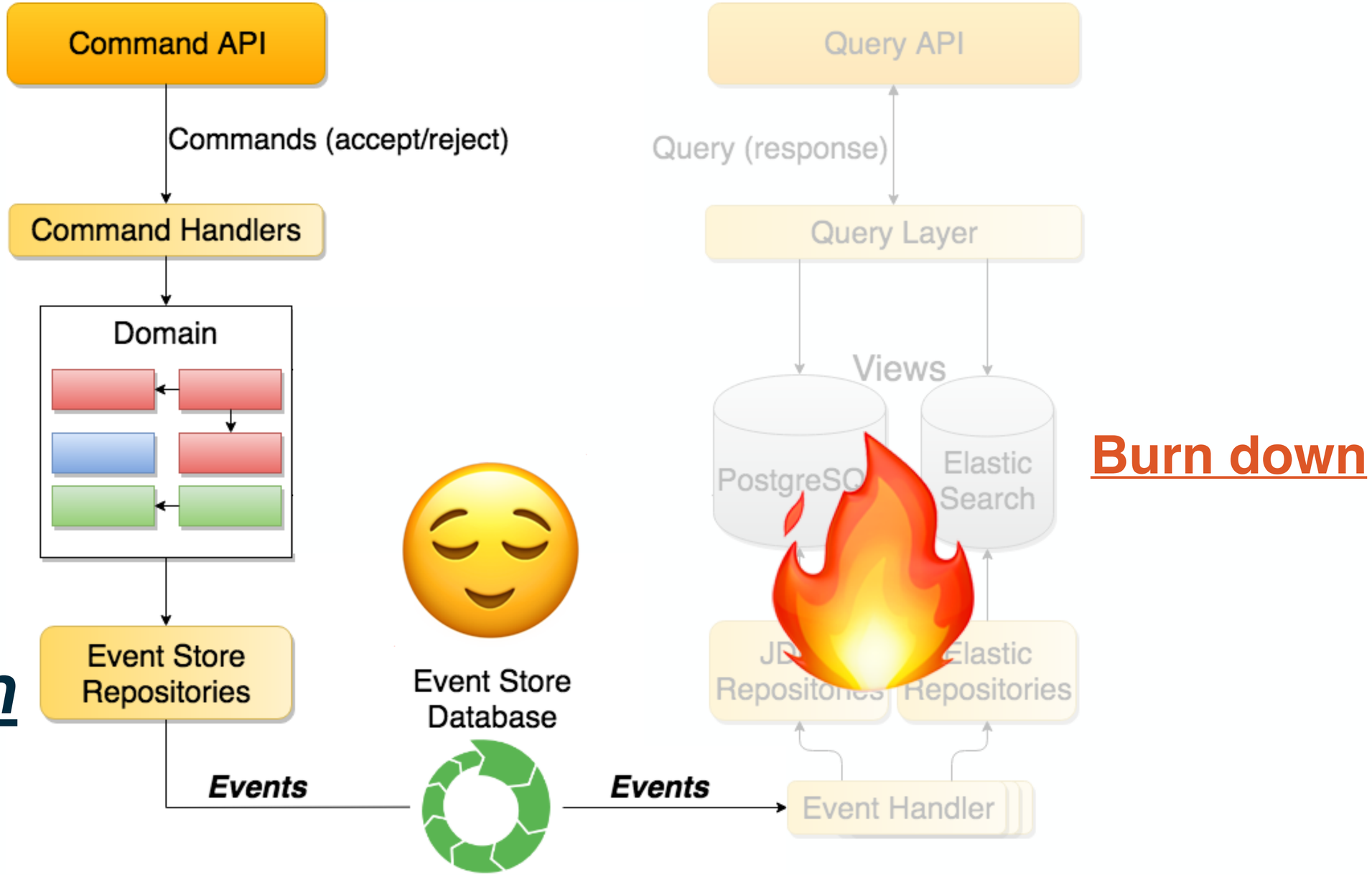


3. Event Sourcing & CQRS



Conclusion

3. Event Sourcing & CQRS

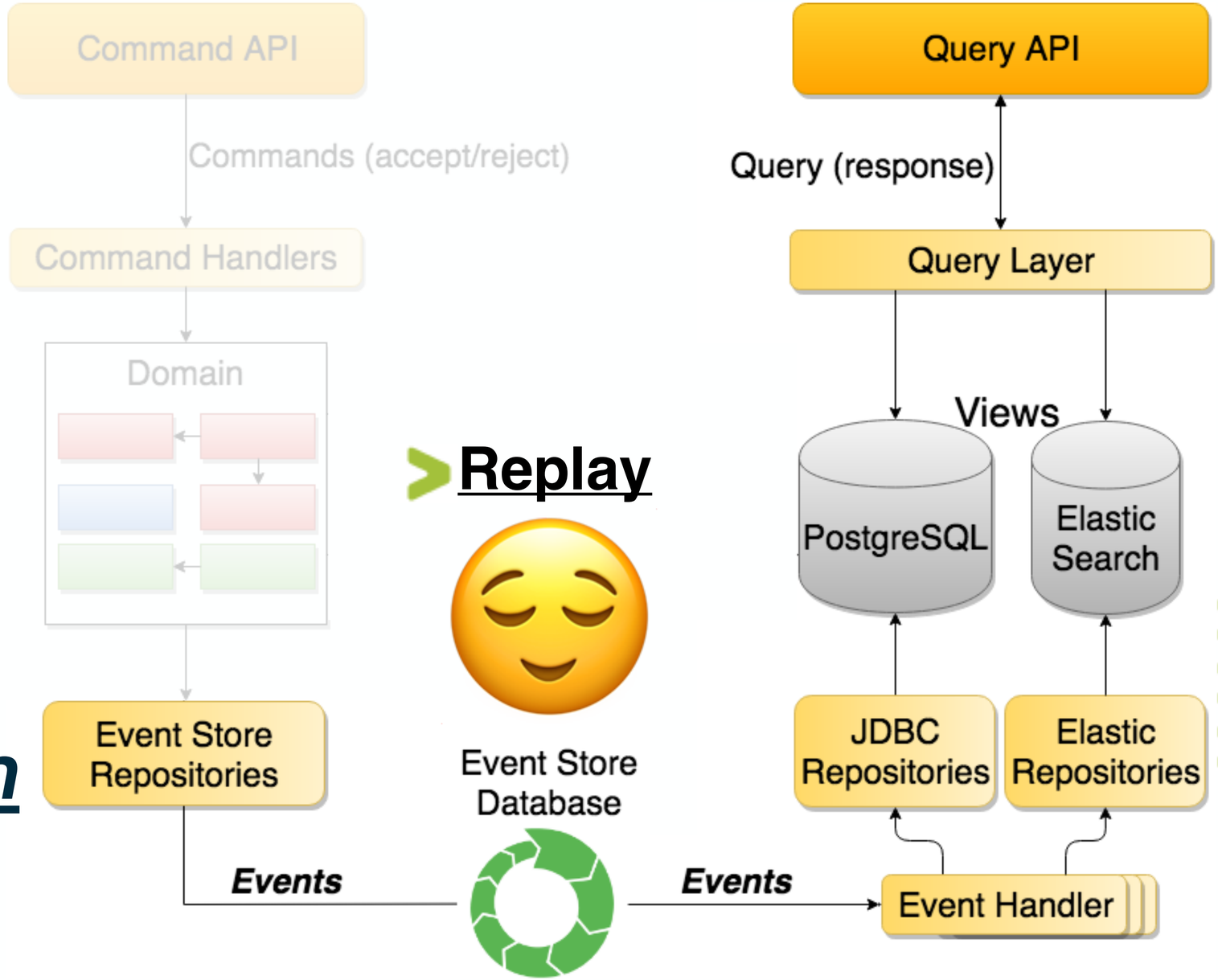


Burn down

Conclusion

3. Event Sourcing & CQRS

- ***TODO***
  - *snapshots*
  - *sagas*



- **Replay**

- **recovery**
- **fix inconsistency**
- **build a new view**

Conclusion

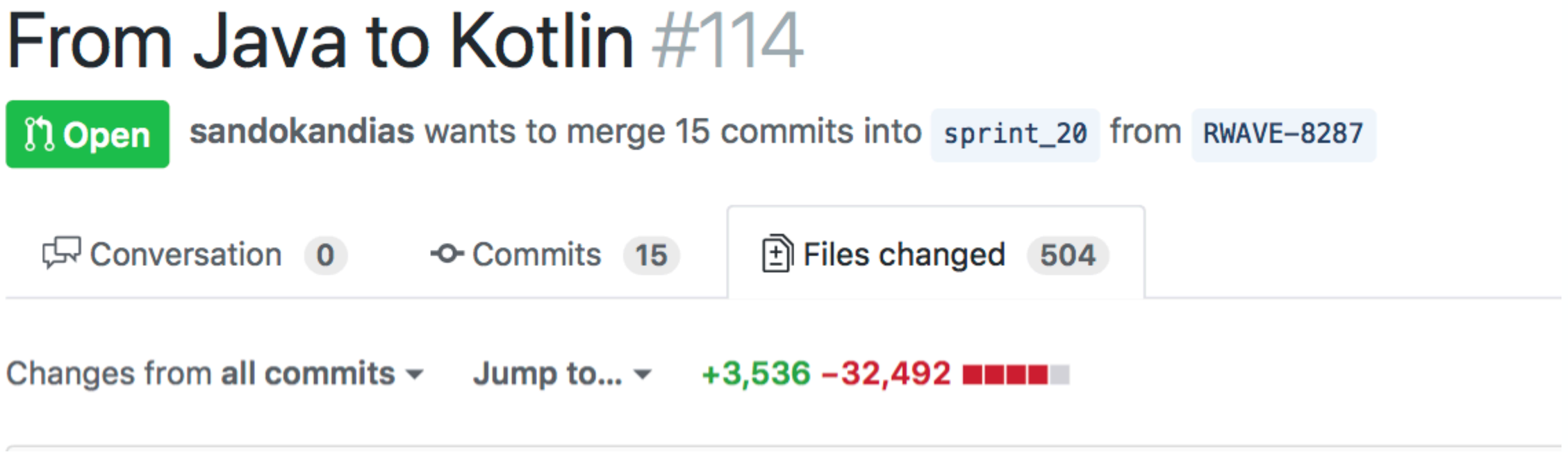
4. Kotlin, DDD, ES & CQRS

- Motivation

- Kotlin
  - Events (immutable) (Axon)
    - *Catalog v1*
  - Data class (boilerplate)
  - Event Stream
    - lambda
  - Modern
  - Easy to learn
  - …

4. Kotlin, DDD, ES & CQRS

- Why should you try using Kotlin?



Sealed Class

Pattern Matching

Smart Casts

> Events (Event Store Database)

```
sealed class OrderEvent : Event() {

    fun apply(aggregateId: AggregateId, handler: OrderEventHandler) =
            when (this) {
                is OrderCreated → handler.on(aggregateId, this)
                is ProductAdded → handler.on(aggregateId, this)
                is ProductDeleted → handler.on(aggregateId, this)
            }

}

data class OrderCreated(val orderId: String, val status: OrderStatus) : OrderEvent()
data class ProductAdded(val productId: String) : OrderEvent()
data class ProductDeleted(val productId: String) : OrderEvent()
```

Final

4. Kotlin, DDD, ES & CQRS

- Events (Event Store Database)

```
sealed class OrderEvent : Event() {

    fun apply(aggregateId: AggregateId, handler: OrderEventHandler) =
            when (this) {
                is OrderCreated → handler.on(aggregateId, orderCreated: this)
                is ProductAdded → handler.on(aggregateId, productAdded: this)
                is ProductDeleted → handler.on(aggregateId, productDeleted: this)
            }

}

data class OrderCreated(val orderId: String, val status: OrderStatus) : OrderEvent()
data class ProductAdded(val productId: String) : OrderEvent()
data class ProductDeleted(val productId: String) : OrderEvent()
data class AddressAdded(val address: String) : OrderEvent()
```

'when' expression must be exhaustive, add necessary 'is AddressAdded' branch or 'else' branch instead

4. Kotlin, DDD, ES & CQRS

Events (Event Store Database)

```
sealed class OrderEvent : Event() {

    fun apply(aggregateId: AggregateId, handler: OrderEventHandler) =
            when (this) {
                is OrderCreated → handler.on(aggregateId, orderCreated: this)
                is ProductAdded → handler.on(aggregateId, productAdded: this)
                is ProductDeleted → handler.on(aggregateId, productDeleted: this)
                is AddressAdded → handler.on(aggregateId, this)
            }
}

data class OrderCreated(val orderId: String, val status: OrderStatus) : OrderEvent()
data class ProductAdded(val productId: String) : OrderEvent()
data class ProductDeleted(val productId: String) : OrderEvent()
data class AddressAdded(val address: String) : OrderEvent()
```

None of the following functions can be called with the arguments supplied.

- on(AggregateId, **OrderCreated**) *defined in* br.com.skip.the.dishes.domain.order.elements.OrderEventHandler
- on(AggregateId, **ProductAdded**) *defined in* br.com.skip.the.dishes.domain.order.elements.OrderEventHandler
- on(AggregateId, **ProductDeleted**) *defined in* br.com.skip.the.dishes.domain.order.elements.OrderEventHandler

- **Value Objects - *data class***
  - Typesafe
    - Primitive Obsession
    - Invalid Objects
    - Immutability

```
data class Price(
        @field:[NotNull Min(value = 0)] val amount: PriceAmount,
        @field:[NotBlank] val currency: Currency,
        @field:[NotNull Min(value = 0)] val scale: PriceScale = 0) {

    init {
        validate( instance: this)
    }

}
```

- **Anemic Domain Model - *kotlin extensions***
  - Only Getters / Setters
  - Services Layers are OVER ABUSED!!!

```
if (customer.orders.any { it: Order
                it.status == OrderStatus.PENDING }) {
    // do something
}
```

```
if (customer.hasPendingOrders()) {
    // do something
}
```

Tests

```
@Test
fun `Try to add a product to an order already canceled`() {
    assertException(
            expectedException = AttemptChangeOrderStatusException::class.java,
            expectedMessage = "It is not allowed change order status.",
            assertThat = { e →
                e.status == OrderStatus.FINISHED
            },
            block = {
                order.addProduct( productId: "Product ID 2")
            }
    )
}
```

Coroutines

- asynchronous programming
- we put the complications into libraries
- experimental != unsecurity

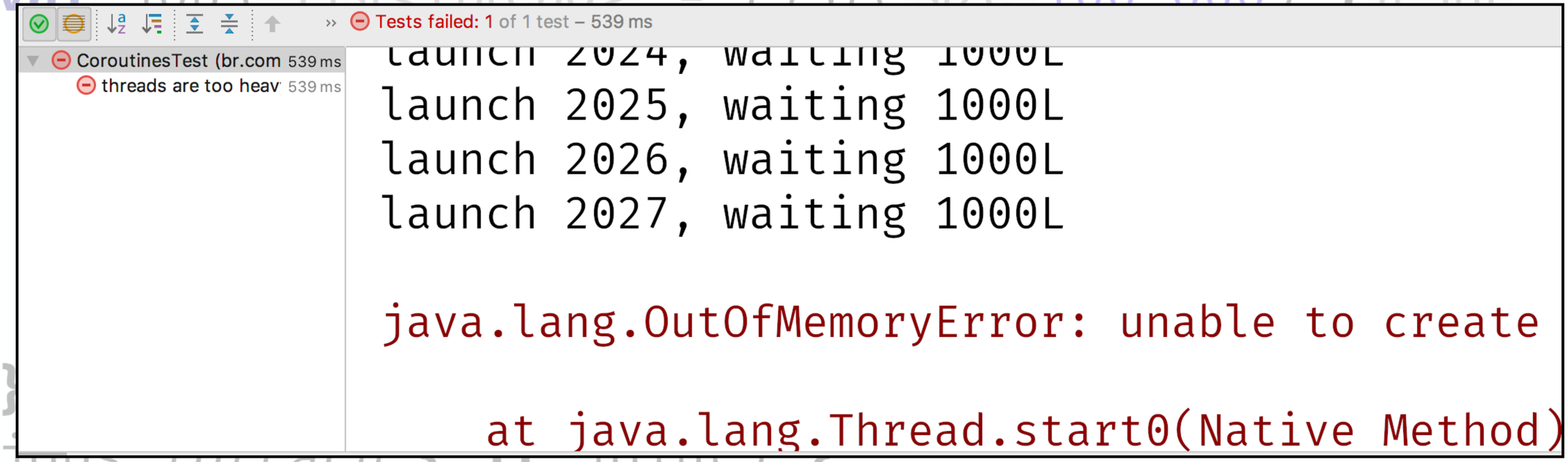
Coroutines

What's the result?

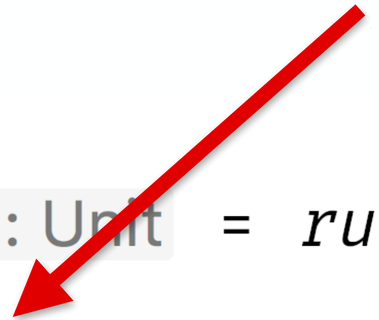
```
val jobs: List<Thread> = List( size: 100_000) { it: Int
    thread {
        println("launch $it, waiting 1000L")
        Thread.sleep( millis: 1000L)
        print("$it done ")
    }
}
jobs.forEach { it.join() }
```

Coroutines

What's the result?

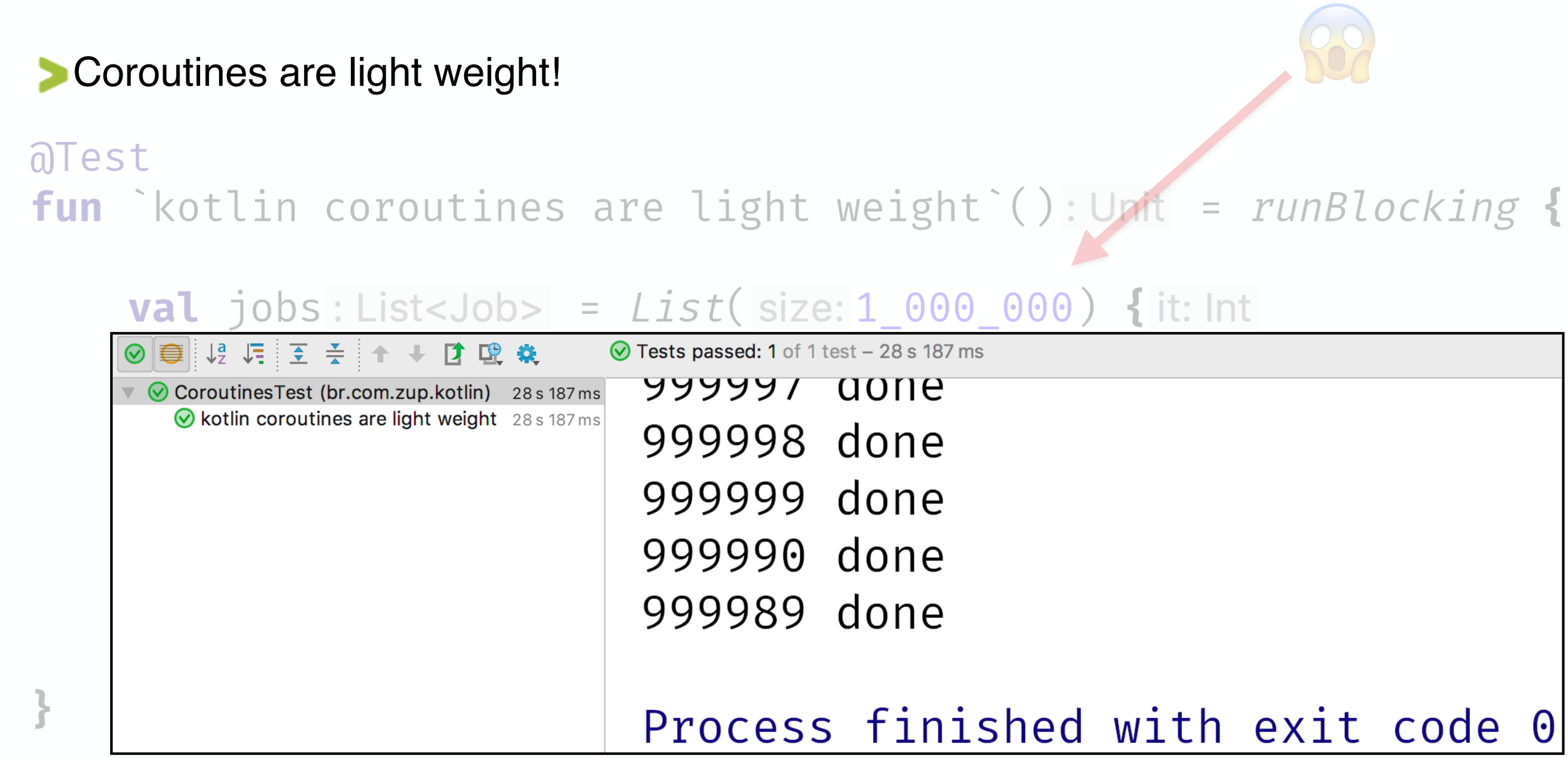


- Coroutines are light weight!



```
@Test
fun `kotlin coroutines are light weight`(): Unit = runBlocking {

    val jobs: List<Job> = List( size: 1_000_000) { it: Int
        launch { this: CoroutineScope
            println("launch $it, waiting 200L")
            delay( time: 200L)
            println("$it done ")
        }
    }
    jobs.forEach { it.join() }
}
```

- Coroutines are light weight!



- Coroutines - async/await

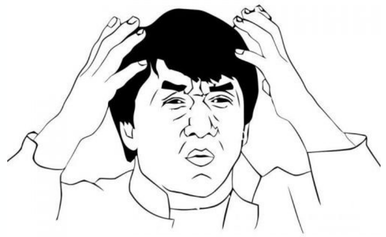
```
val findOne1: Deferred<Catalog> =
    async { service.findCatalogId(CatalogId( value: 1)) }

val findOne2: Deferred<Catalog> =
    async { service.findCatalogId(CatalogId( value: 2)) }

runBlocking { this: CoroutineScope
  val catalog1: Catalog = findOne1.await()
  val catalog2: Catalog = findOne2.await()

  print("($catalog1) ($catalog2)")
}
```

5. Kotlin 102

- Kotlin is the path to Functional Programming. Functional Programming leads to Pure Functions, Pure Functions leads to Monad and *Asynchronous Programming*. Asynchronous Programming leads to Reactive Programming.

Master Yoda - Return of the Jedi

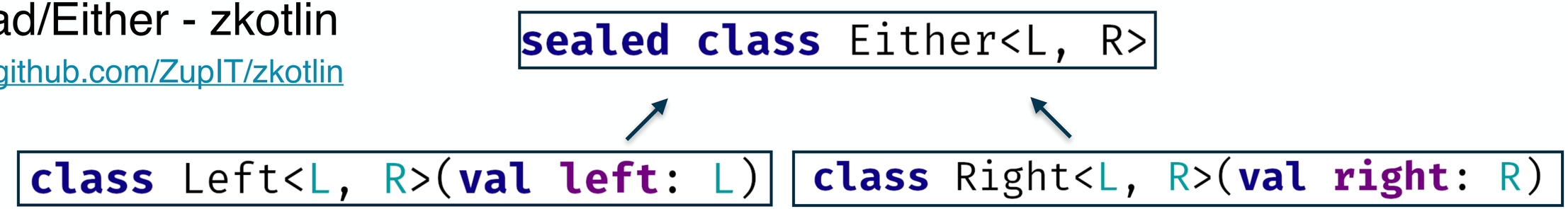
- Functional Programming
  - Purity (no-side effect)
  - Referential Transparency
  - Purity vs I/O
    - Exceptions propagation
    - Thread limits



```
fun get(page: PageNumber, callback: Callback<List<Game>>) {
    try {
        val games: List<Game> = gameRestV1.getGames(page)
        callback.onSuccess(games)
    } catch (e: Exception) {
        logger.error(e)
        callback.onError(e.message)
    }
}
```

Monad/Either - zkotlin

https://github.com/ZupIT/zkotlin




```
fun get(page: PageNumber): List<Game> {

    val result : Either<Failure, List<Game>> = gameRest.getGames(page)

    return result.fold({ it: Failure
                emptyList()
            },
            { it: List<Game>
                it
            })
}
```

```
result.isLeft()
result.isRight()
result.get()
result.failure()
result.success { ... }
result.failure { ... }
result.getOrElse { ... }
result.andThen { ... }
```

Monad - Either -> Either -> Either

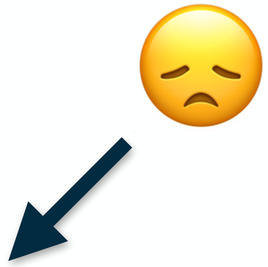
```
fun updateWithCallBackHell(gameId: GameId) {
    gameRest.getGame(gameId).fold(
            { logger.error(it) },
            { game : Success<Game> →
                gameRest.getPlayers(game.result).fold(
                        { logger.error(it) },
                        { it: Success<GamePlayer>
                            gameRest.updateResults(it.result).fold(
                                    { logger.error(it) },
                                    { result : Success<GameResult> →
                                        result.also { it: Success<GameResult>
                                            logger.info( msg: "finally!")
                                        }
                                    })
                        })
            })
}
```

FOLD HELL

- Either foldCompose

```
gameRest.getGame(gameId)
    .foldCompose({ ... }, { gameRest.getPlayers(it.result) })
    .foldCompose({ ... }, { gameRest.updateResults(it.result) })
    .fold({ /* error */ }, { /* success */ })
```

- Either foldCompose
  - ping-pong effect
  - E.g. Payments Retry



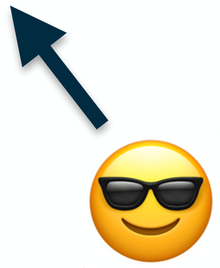
```
gameRest.getGame(gameId)
  .foldCompose({ backupRest.tryRecovery(gameId, it) }, { ... })
  .foldCompose({ ... }, { gameRest.updateResults(it.result) })
  .fold({ ... }, {  /* success */ })
```

- Either foldCompose
  - ping-pong effect

```
gameRest.getGame(gameId)
  .foldCompose({ backupRest.tryRecovery(gameId, it) }, { ... })
  .foldCompose({ ... }, { gameRest.updateResults(it.result) })
  .fold({ ... }, {  /* success */ })
```

- Either foldCompose
  - ping-pong effect

```
gameRest.getGame(gameId)
  .foldCompose({ backupRest.tryRecovery(gameId, it) }, { ... })
  .foldCompose({ ... }, { gameRest.updateResults(it.result) })
  .fold({ ... }, {  /* success */ })
```



Either, Coroutines, Extensions, zkotlin all together

- *Wraps I/O problems with Either<L,R>*

```
sealed class ServiceError(val error: String?,
                          val exception: Exception? = null) {
    class NotFound(error: String?): ServiceError(error)
    class UnknownError(error: String, exception: Exception):
            ServiceError(error, exception)
}
```

> *Wraps I/O problems with Either<L,R>*

```
fun findBundleById(bundleId: BundleId):
                    Either<ServiceError, BundleRepresentation> =
    try {
        findOne(bundleId)
                ?.let( ::eitherRight)

                ?: eitherLeftNotFound(bundleId)

    } catch (e: Exception) {
        eitherLeftUnexpectedError(bundleId, e)
    }
```

Zup

> sync {}

```
private fun findAtPcm(tenant: Tenant, bundleId: BundleId):
        EitherBundleRepresentation = tenant { this: Tenant
    bundleSearch.findBundleById(bundleId)
}
```

> async {}

```
private fun asyncFindAtPcm(tenant: Tenant,
                           bundleId: BundleId):
        DeferredEitherBundleRepresentation = async {
    findAtPcm(tenant, bundleId)
}
```

runBlocking

```
private fun saveBundleIds(tenantContext: TenantContext,
                          bundleIds: List<BundleId>): List<Bundle>
    = runBlocking { this: CoroutineScope
      bundleIds
              .map(asyncFindPcmByTenant(tenantContext))
              .await()
              .foldError(::handlerFindPcmError)
              .foldSuccess(::mapToModel)

              .map(asyncSaveByTenant(tenantContext))
              .await()
              .foldError(::handlerSaveError)
              .foldSuccess()
}
```

zkotlin

```
.await()
.foldError( ::handlerFindPcmError)
.foldSuccess( ::mapToModel)
```

Couroutines/Ext. Methods

```
suspend fun <T> List<Deferred<T>>.await

inline fun <L, R> List<Either<L, R>>.foldError

inline fun <L, R, T> List<Either<L, R>>.foldSuccess
```

5. Kotlin 102

- Conclusion

- Kotlin & Functional Programming
  - Purity, Referential Transparency, Immutability

- Higher-order functions
  - *Sugar syntax (it, ƛ as last parameter)*
    - *Less boilerplate*
  - *a nicer syntax to compose Either pattern*

- Coroutines is easy to use
  - Ready for Spring Reactor

- **Catalog Reindex Process before Coroutines: +/- 30 m, after < 4 m**

6. Kotlin, Java & Legacy

- Everything is final on Kotlin

Spock

```
<dependency>
    <groupId>de.jodamob.kotlin</groupId>
    <artifactId>kotlin-runner-spock</artifactId>
    <version>${kotlin.runner.spock.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib-nodep</artifactId>
    <version>${cglib.nodep.version}</version>
    <scope>test</scope>
</dependency>
```

Mockito

```
<dependency>
    <groupId>com.nhaarman</groupId>
    <artifactId>mockito-kotlin</artifactId>
    <version>${mockito.kotlin.version}</version>
    <scope>test</scope>
</dependency>
```

SpringBoot - allopen

final

```
@Configuration
class MyConfig {
}
```

Classes annotated with '@Configuration' could be implicitly subclassed and must not be final more... (⌘F1)

```
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-allopen</artifactId>
    <version>${kotlin.version}</version>
</dependency>
```

- JPA

```
@Entity
data class Customer(@field:Id val id: CustomerId,
                    val name: CustomerName,
                    @field:Enumerated(EnumType.STRING)
                    val gender: Gender)
```

```
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-noarg</artifactId>
    <version>${kotlin.version}</version>
</dependency>
```

6. Kotlin, Java & Legacy

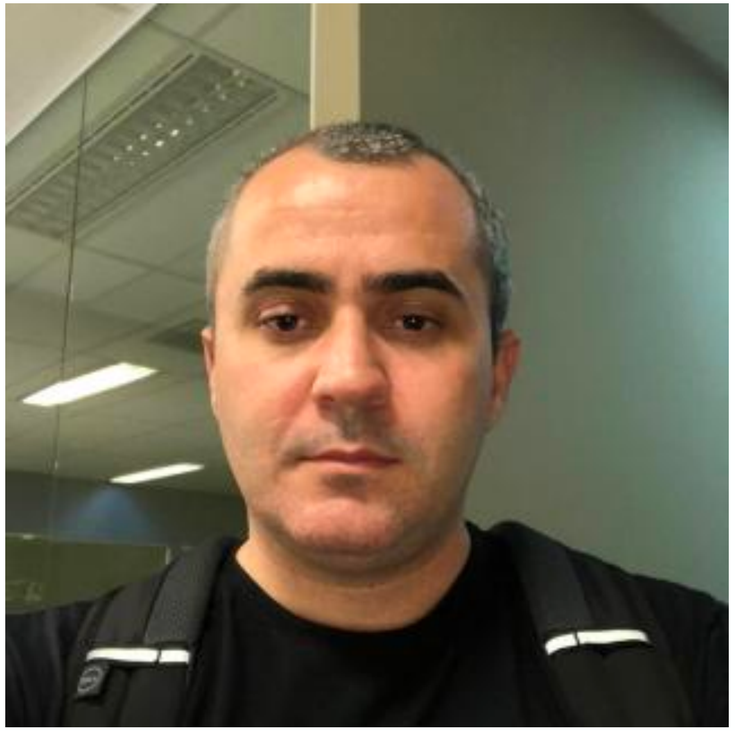
- Spring Data QueryDSL - Plugins (modules)
- Default Interface Java 8 (Kotlin 1.2.40)
- Lambdas return - Unit

QCon SÃO PAULO

Zup

Conclusion

Mike Shigueru Matsumoto
Email: mike.matsumoto@zup.com.br



Augusto Branquinho
Email: augusto.branquinho@zup.com.br

Zup

We’re a fast growing company (2011), but just starting.

Endeavor Global

Gartner Cool Vendor

Kaszek Ventures

Kevin Efrusy
Accel Ventures

Martin Scobari
General Atlantic

Romero Rodrigues
Buscapé Company

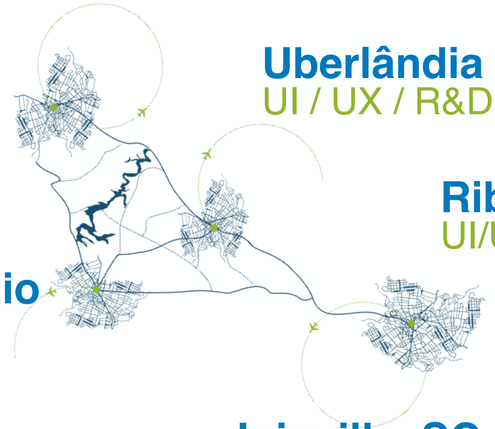
Silvio Genesini
Silvigen Consultoria

540+ Zuppers.

PCI Compliant

BH MG
UI / UX / R&D / DEV

Uberlândia MG
UI / UX / R&D / DEV / SUPPORT



Ribeirão Preto SP
UI/UX/DESIGN

São José do Rio Preto SP
DEV

Joinville SC
R&D

Lisboa, Portugal
SALES / CONSULTING / R&D

São Paulo SP
SALES / CONSULTING / PMO / REQUIREMENTS
UI/UX