



Excepciones Excepcionales

Rafa Gómez

Contenidos



- Excepciones
- Railway Programming
- Errors as Values
- Monads

Excepciones



```
class VerifyUserCommandHandler(  
    repository: UserRepository,  
    publisher: DomainEventPublisher  
) {  
    private val verifier = UserVerifier(repository, publisher)  
  
    fun handle(command: VerifyUserCommand) {  
        verifier.invoke(UserId.fromString(command.userId))  
    }  
}
```

```
class UserVerifier(  
    private val repository: UserRepository,  
    private val publisher: DomainEventPublisher  
) {  
    private val finder = UserFinder(repository)  
  
    fun invoke(id: UserId) {  
        val user = finder.invoke(id)  
        val verifiedUser = user.verify()  
  
        repository.save(user)  
        publisher.publish(verifiedUser.pullEvents())  
    }  
}
```

Excepciones



Situaciones inesperadas o erróneas que pueden llevar al programa a fallar.

Tienen un impacto en el rendimiento de nuestras aplicaciones ya que generan un **stack trace** para proveernos información sobre el error ocurrido.

Pueden ser categorizadas en dos grupos:

- **Sistema**
- **Negocio**

Excepciones de Sistema



Excepciones del lenguaje de programación para lidiar con errores que ocurren en cualquier tipo de programa independientemente del contexto sobre el que operen.

Nos informan de un error creado por nuestro código.

Ejemplos:

- **NullPointerException**
- **OutOfMemory**
- **IncorrectExpectedResultSize**

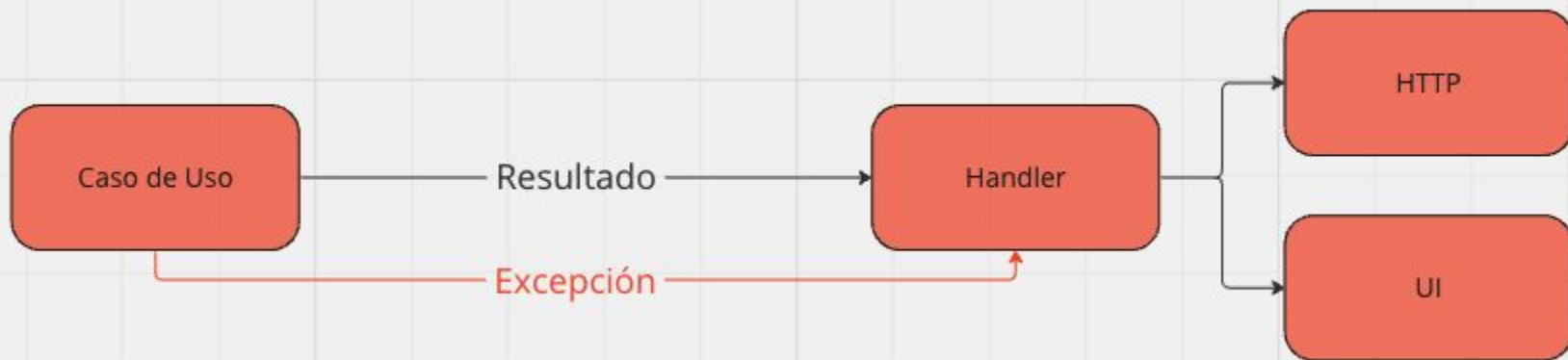
Excepciones de Negocio



Mecanismo de control para "romper" el flujo de nuestros casos de uso , de manera intencionada, cuando encontramos una condición inválida en el contexto en el que nuestro programa opera.

- **Validación de Datos**
 - Value Objects
- **Lógica de negocio**
 - Casos de Uso
 - Dominio

Excepciones de Negocio



Repositorio



Excepciones

Lanzamos excepciones por:

- ▲ Comodidad
- ▲ Inercia / Desconocimiento de Alternativas
- ▲ Centrarnos en el 'happy path'
- ▲ Nuestras herramientas se basan en ellas

Y aceptamos problemas como:

- ▼ Pérdida de rendimiento
- ▼ Aumento carga cognitiva
- ▼ Peor experiencia de desarrollo
- ▼ Acoplamiento a herramientas externas

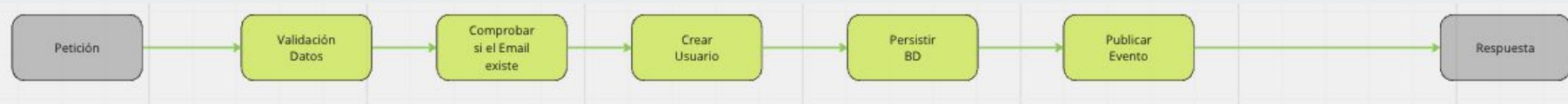
Railway Programming



Patrón de diseño que estructura nuestros casos de uso en dos "**raíles**". Uno responsable de la ejecución **correcta** de la secuencia de operaciones mientras el otro es el encargado de gestionar los **errores**.

Cada operación tiene la posibilidad de continuar por "**raíl**" correcto o cambiar al de errores.

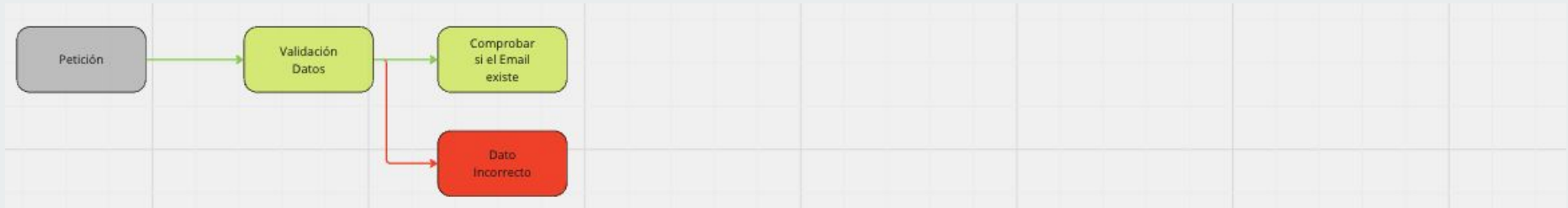
Railway Programming



Railway Programming



Railway Programming



Railway Programming



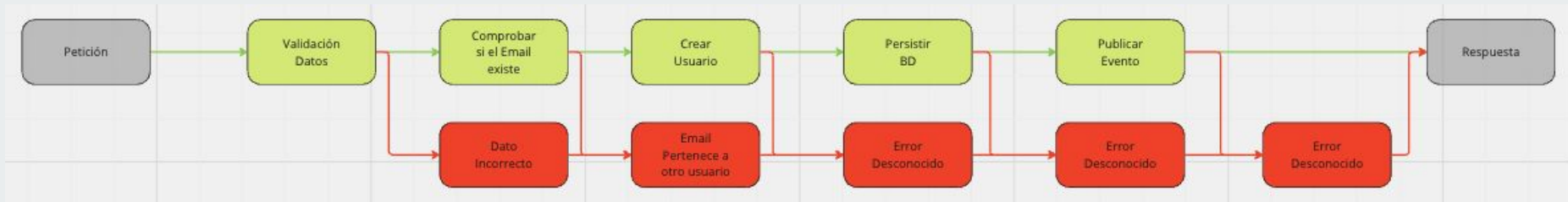
Railway Programming



Railway Programming



Railway Programming



Errors as Values



Patrón de diseño donde los errores son siempre devueltos como un posible resultado tras la ejecución de una función.

Entre los diferentes patrones para devolver un error existen:

- **Sealed Classes/Traits**
- **Monads**

Sealed Hierarchies



Sealed Hierarchies, tipo de datos que restringe la cantidad de posibles implementaciones del mismo. Aplicable tanto a **clases** como **interfaces**, nos permite representar un concepto como una colección de posibles subconceptos.

Suele utilizarse para modelar:

- **Dominios**
- **Estados**
- **Resultados**
- **Opciones**

Sealed Class



```
sealed class Validation<Value> {  
    class Success<Value>(val value: Value): Validation<Value>()  
    class Failure<Value>(): Validation<Value>()  
}
```

Ejercicio



Monads



Contenedores de datos que nos proporcionan una **funcionalidad** gestionada de manera **transparente** y permiten **combinar** funciones.

Monads



Para que una estructura de datos sea considerada una Monad ha de cumplir tres reglas:

1. Tener una función unidad (**unit**, **return**)
2. Tener un combinador/es (**flatMap**, **map**)
3. Elemento identidad (**identity element**)

Monads



Entre las Monads más comunes encontramos:

- **Optional / Maybe** -> Nullabilidad
- **List / Sequence** -> Conjunto de elementos
- **Future / Promise** -> Asincronía
- **Either / Try** -> Gestión de errores
- **I/O** -> Operaciones de input/output
- **Reader** -> Inyección de dependencias
- **State** -> Gestión de un estado modificable

Combinar Funciones



Para combinar dos funciones han de cumplirse una condición principal:

- Segunda función ha de pertenecer al contexto del resultado de la primera

Un combinador (**combinator**) es una función que actúa como **medio** para **combinar** diferentes funciones permitiéndonos evitar la condición anterior.

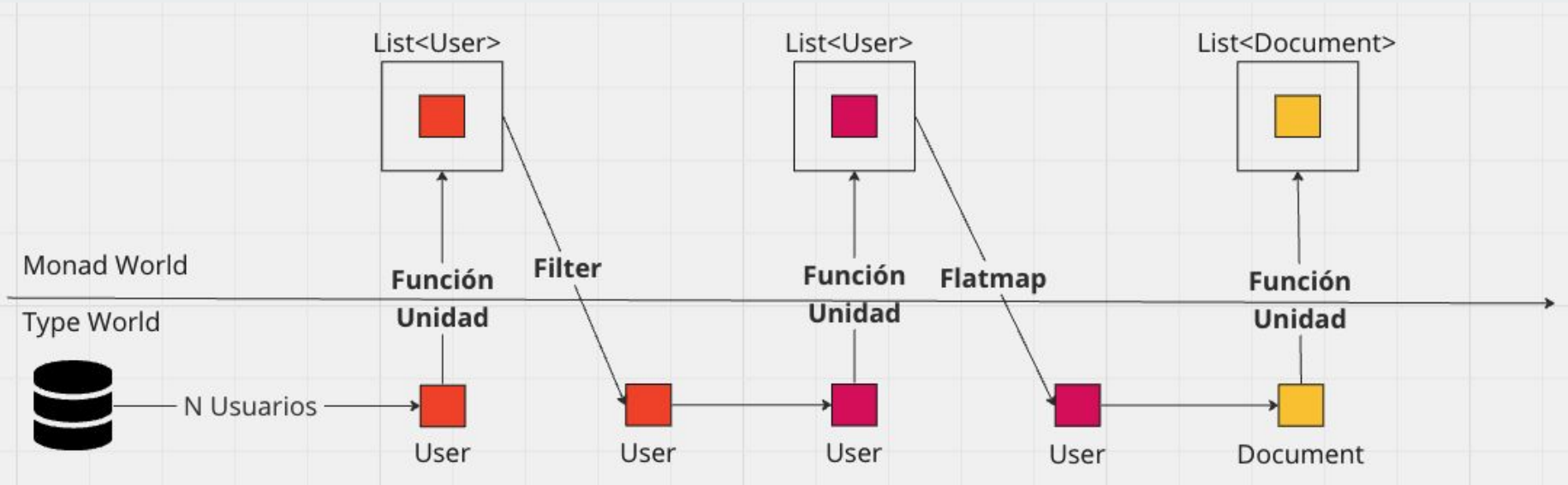
También pueden utilizarse para **modificar** o **reemplazar** el contenido de nuestra monad.

Combinar Funciones



```
fun usersToDocuments(users: List<User>): List<Document> =  
    users  
        .filter { user -> user.documents.isNotEmpty() }  
        .flatMap { user -> user.documents }  
        .sortedBy { document -> document.status }
```

Combinator



Combinator Hell



```
fun handlePost(request: HttpRequest): Result<HttpResponse, Error> =
    request.readJson()
        .flatMap { json ->
            json.toCommand()
                .flatMap { command ->
                    loadResourceFor(request)
                        .flatMap { resource ->
                            performCommand(resource, command)
                                .flatMap { outcome ->
                                    outcome.toHttpResponseFor(request)
                                }
                            }
                        }
                    }
        }
    }
```

Combinator Heaven



```
fun handlePost(request: HttpRequest): Either<Error, HttpResponse> = either {  
    val command = request.readJson().toCommand().bind()  
    val resource = loadResourceFor(request).bind()  
  
    val outcome = performCommand(resource, command).bind()  
    return outcome.toHttpResponseFor(request).bind()  
}
```

Either



Either es una Monad que nos permite representar el resultado de un función como un conjunto excluyente de valores (**Left** o **Right**). Un Either puede ser únicamente uno de sus valores al mismo tiempo.

- **Left** -> Representa y contiene un error
- **Right** -> Representa y contiene un resultado correcto

Ejercicio



Monads

Lo bueno:

- ▲ Funcionalidades gratis
- ▲ Composición de funciones
- ▲ Menor carga cognitiva
- ▲ Expresividad

Lo malo:

- ▼ Curva de aprendizaje
- ▼ Cambio de estructuración de nuestro código/modelo mental
- ▼ No cuentan con soporte nativo en todos los lenguajes/herramientas

Recursos



Artículos y Conferencias

1. [Exceptional Performance](#) - Aleksey Shipilev
2. [Functional Domain Modeling](#) - Simon Vergauwen
3. [Sealed Classes to model State](#) - Patrick Cousins

Librerías funcionales

1. [Vavr](#) - Java
2. [Arrow](#) - Kotlin
3. [Rambda](#) - Javascript/Typescript
4. [Cats/Zio](#) - Scala



Software Crafters Barcelona

X EDITION #SCBCN23

UNIXcorn

Friends for life



PegaSO



PonIA



CODESAI



Grafana





That's all Folks!