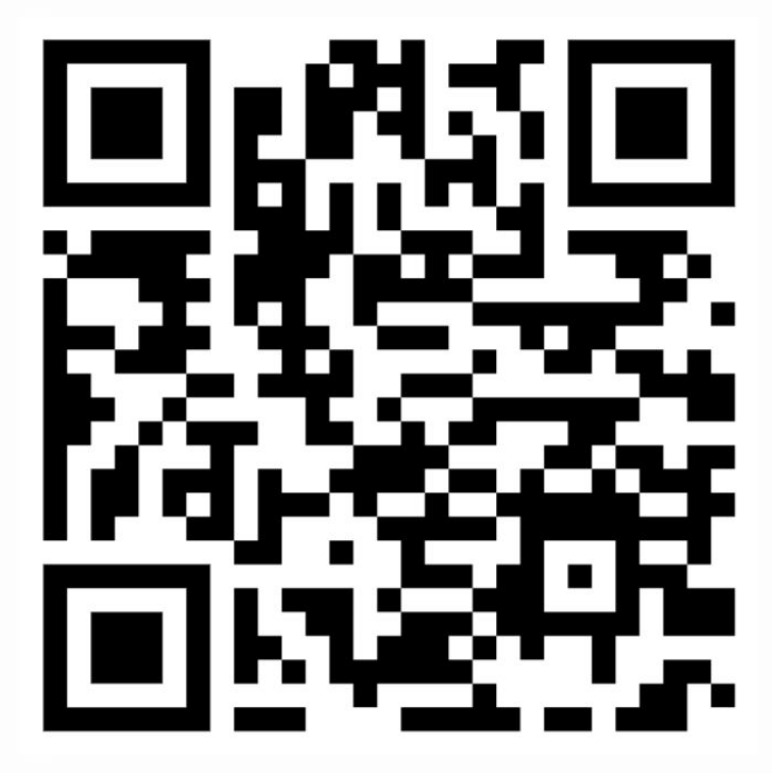
Functional Error Handling

Rafa Gómez

Repositorio



Errores

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

Pueden ser categorizadas en dos grupos:

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

Errores de Sistema

Errores que ocurren en cualquier tipo de programa independientemente del contexto sobre el que operen. Suelen ir relacionados con funcionamiento interno de las herramientas que utilizamos (OS, Framework, BD…) y nos informan de errores irrecuperables en nuestro sistema.

Ejemplos:

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

Errores de Negocio

Condiciones que incumplen reglas de nuestro negocio y podrían generar inconsistencias en nuestro sistema de no ser controladas.

- **Validación de Datos**
  - Value Objects
- **Lógica de negocio**
  - Casos de Uso
  - Invariantes de Agregados

Gestión de errores

La gestión de errores es el proceso de gestionar y responder a errores o situaciones inesperadas que ocurren durante la ejecución de un programa. Garantiza que el programa pueda **producir un feedback** apropiado en lugar de fallar o producir resultados incorrectos

Un manejo de errores efectivo hace que el software sea más **robusto**, **predecible** y **fácil de depurar y mantener**.

- **Gestión de errores Imperativa**
- **Gestión de errores Funcional**

Gestión de errores imperativa

- Propagación implícita de errores (excepciones)
- Cambios de estados (locales o globales)

Excepciones

El mecanismo de control de errores más extendido en el desarrollo del software, se utilizan para cortar la ejecución de nuestros programas.

Entre sus características tenemos

- Generan un **stack trace** para proveernos información sobre el error ocurrido (que tiene un impacto en el rendimiento)
- Se **propagan automáticamente** por todo el stack hasta llegar a inicio de nuestro flujo si no son capturadas.
- Nos permiten escribir un código "libre de errores" y centrarnos en el caso ideal (**happy path**)

Excepciones

```
class ImperativeUserVerifier(
    private val repository: SolutionUserRepository,
    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

Lanzamos excepciones por:

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

Y aceptamos problemas como:

- ❌ Leve pérdida de rendimiento
- ❌ Aumento carga cognitiva
- ❌ Peor experiencia de desarrollo

Gestión de errores funcional

- Gestión explícita de resultados
- Modelado explícito de errores
- Composición de funciones

Errors as Values

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

De entre las diferentes opciones para implementar este patrón, nos centraremos en una combinación de:

- **Sealed Classes/Traits**
- **Either**

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 Hierarchies

```
sealed interface CreateUserError {
    object InvalidEmail : CreateUserError
    object InvalidName : CreateUserError
    object InvalidSurname : CreateUserError
    object InvalidMobilePhone : CreateUserError
    object UserAlreadyExists : CreateUserError
}
```

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

Como crear un Either

```
fun guardUserDoesNotExists(email: Email): Either<CreateUserError, Unit> =
    if (repository.existByEmail(email)) UserAlreadyExistsError().left()
    else Unit.right()
```

Either con excepciones

```
fun invoke(id: UserId) {
    val user = finder.invoke(id)
    val verifiedUser = user.verify()

    repository.save(user)
    publisher.publish(verifiedUser.pullEvents())
}
```

Either con excepciones

```
sealed class VerifyUserError {
    object UserNotFound : VerifyUserError()
    object NotAllDocumentVerified : VerifyUserError()
    object UserAlreadyVerified : VerifyUserError()
    object UserStatusCannotBeVerified : VerifyUserError()
    object CardStatusNotConfirmed : VerifyUserError()
}
```

Either con excepciones

```
private fun User.safeVerify(): Either<VerifyUserError, Unit> =
    Either.catch { user.verify }
        .mapLeft { error →
            when(error) {
                is NotAllDocumentVerifiedException → NotAllDocumentVerified
                is UserStatusCannotBeVerifiedException → UserStatusCannotBeVerified
                is UserAlreadyVerifiedException → UserAlreadyVerified
                else → throw error
            }
        }
```

Either con excepciones

```
operator fun invoke(userId: UserId): Either<FunctionalVerifyUserError, Unit> = either {
    val user = repository.find(ById(userId)) ?: raise(UserNotFound)

    val verifiedUser = user.safeVerify().bind()

    repository.save(verifiedUser)
    publisher.publish(verifiedUser.pullEvents())
}

private fun User.safeVerify(): Either<VerifyUserError, Unit> =
    Either.catch { user.verify }
        .mapLeft { error ->
            when(error) {
                is NotAllDocumentVerifiedException -> NotAllDocumentVerified
                is UserStatusCannotBeVerifiedException -> UserStatusCannotBeVerified
                is UserAlreadyVerifiedException -> UserAlreadyVerified
                else -> throw error
            }
        }

sealed class VerifyUserError {
    object UserNotFound : VerifyUserError()
    object NotAllDocumentVerified : VerifyUserError()
    object UserAlreadyVerified : VerifyUserError()
    object UserStatusCannotBeVerified : VerifyUserError()
    object CardStatusNotConfirmed : VerifyUserError()
}
```

Control de flujo

```
operator fun invoke(userId: UserId): Either<FunctionalVerifyUserError, Unit> = either {
    val user = repository.find(ById(userId)) ?: raise(UserNotFound)

    val verifiedUser = verify(user).bind()

    repository.save(verifiedUser)
    publisher.publish(verifiedUser.pullEvents())
}
```

Control del flujo

```
fun invoke(userId: UserId): Either<VerifyUserError, Unit> =
    findUser(userId)
        .flatMap { user → user.safeVerify() }
        .flatMap { user → repository.save(user) }
        .flatMap { user → publisher.publish(user.pullEvents()) }
```

Errors as Values

Lo bueno:

- 🟢 Composición de funciones
- 🟢 Menor carga cognitiva
- 🟢 Expresividad
- 🟢 Errores en tiempo de compilación

Lo malo:

- 🔴 Curva de aprendizaje
- 🔴 Cambio de modelo mental
- 🔴 Herramientas dependen de excepciones

Ejemplos

🔴 Agregados

```
data class SolutionUser(
    val id: UserId,
    val email: Email,
    val phoneNumber: PhoneNumber,
    val createdOn: ZonedDateTime,
    val name: Name,
    val surname: Surname,
    val documents: Set<Document>,
    val status: Status,
    val cardStatus: CardStatus
): Aggregate() {

    fun verify(): SolutionUser {
        guardAllDocumentsAreVerified()
        guardStatusCanBeVerified()
        guardCardStatusToBeVerified()

        return copy(status = VERIFIED)
            .also { it.pushEvent(UserVerifiedEvent(id.toString())) }
    }

    private fun guardAllDocumentsAreVerified() =
        if(documents.all { document -> document.status == DocumentStatus.VERIFIED }) Unit
        else throw NotAllDocumentVerifiedException()

    private fun guardStatusCanBeVerified() {
        when(status) {
            INCOMPLETE -> throw UserStatusCannotBeVerifiedException()
            VERIFIED -> throw UserAlreadyVerifiedException()
            PENDING_VERIFICATION -> Unit
        }
    }

    private fun guardCardStatusToBeVerified() {
        when(cardStatus) {
            PENDING -> throw CardStatusNotConfirmedException()
            CONFIRMED -> Unit
        }
    }
}
```

Agregados

```
data class SolutionUser(
    val id: UserId,
    val email: Email,
    val phoneNumber: PhoneNumber,
    val createdOn: ZonedDateTime,
    val name: Name,
    val surname: Surname,
    val documents: Set<Document>,
    val status: Status,
    val cardStatus: CardStatus
): Aggregate() {

    fun safeVerify(): Either<VerifyUserDomainError, SolutionUser> = either {
        guardAllDocumentsAreVerifiedEither().bind()
        guardStatusCanBeVerifiedEither().bind()
        guardCardStatusToBeVerifiedEither().bind()

        return copy(status = VERIFIED)
            .also { it.pushEvent(UserVerifiedEvent(id.toString())) }
            .right()
    }

    private fun guardAllDocumentsAreVerifiedEither() =
        if(documents.all { document -> document.status == DocumentStatus.VERIFIED }) Unit.right()
        else NotAllDocumentVerified.left()

    private fun guardStatusCanBeVerifiedEither() =
        when(status) {
            INCOMPLETE -> UserStatusCannotBeVerified.left()
            VERIFIED -> UserAlreadyVerified.left()
            PENDING_VERIFICATION -> Unit.right()
        }

    private fun guardCardStatusToBeVerifiedEither() =
        when(cardStatus) {
            PENDING -> CardStatusNotConfirmed.left()
            CONFIRMED -> Unit.right()
        }
}

sealed class VerifyUserDomainError {
    object NotAllDocumentVerified : VerifyUserDomainError()
    object UserAlreadyVerified : VerifyUserDomainError()
    object UserStatusCannotBeVerified : VerifyUserDomainError()
    object CardStatusNotConfirmed : VerifyUserDomainError()
}
```

🔴 Value Objects

```
data class Email(val value: String) {
    private val regex: Regex =
        "[a-zA-Z0-9+._%'\\-+]{1,256}@[a-zA-Z0-9][a-zA-Z0-9\\-]{0,64}(\\.[a-zA-Z0-9][a-zA-Z0-9\\-]{0,25})+".toRegex()

    init {
        if (!regex.matches(value)) throw InvalidEmailException(value)
    }

    override fun toString(): String = value
    class InvalidEmailException(email: String) : RuntimeException("Email $email invalid")
}
```

🟢 Value Objects

```kotlin
@JvmInline
value class Email private constructor(val value: String) {

    override fun toString(): String = value

    companion object {
        private val regex: Regex =
            "[a-zA-Z0-9+._%'\\-+]{1,256}@[a-zA-Z0-9][a-zA-Z0-9\\-]{0,64}(\\.[a-zA-Z0-9][a-zA-Z0-9\\-]{0,25})+".toRegex()

        fun <Error> createOrElse(value: String, onError: () -> Error): Either<Error, Email> =
            if(regex.matches(value)) Email(value).right()
            else onError().left()
    }
}
```

🔴 Validación de Datos

```
class ImperativeCreateUserCommandHandler(
    repository: SolutionUserRepository,
    publisher: DomainEventPublisher
) {

    private val creator = ImperativeUserCreator(repository, publisher)

    fun handle(command: ImperativeCreateUserCommand) {
        with(command) {
            val name = Name.create(name)
            val surname = Surname.create(surname)
            val email = Email.create(email)
            val phoneNumber = PhoneNumber.create(phoneNumber, phonePrefix)

            creator.invoke(
                id = UserId(id),
                email = email,
                phoneNumber = phoneNumber,
                createdOn = createdOn,
                name = name,
                surname = surname
            )
        }
    }
}
```

🟢 Validación de Datos

```
class FunctionalCreateUserCommandHandler(
    repository: SolutionUserRepository,
    publisher: DomainEventPublisher
) {

    private val creator = FunctionalUserCreator(repository, publisher)

    fun handle(command: FunctionalCreateUserCommand): Either<CreateUserError, Unit> = either {
        with(command) {
            val name = Name.createOrElse(name) { InvalidName }.bind()
            val surname = Surname.createOrElse(surname) { InvalidSurname }.bind()
            val email = Email.createOrElse(email) { InvalidEmail }.bind()
            val phoneNumber = PhoneNumber.createOrElse(phoneNumber, phonePrefix) { InvalidMobilePhone }.bind()

            creator.invokeBlock(
                id = UserId(id),
                email = email,
                phoneNumber = phoneNumber,
                createdOn = createdOn,
                name = name,
                surname = surname
            ).bind()
        }
    }
}
```

🔴 Casos de uso

```
class ImperativeUserCreator(
    private val repository: SolutionUserRepository,
    private val publisher: DomainEventPublisher
) {

    fun invoke(
        id: UserId,
        email: Email,
        phoneNumber: PhoneNumber,
        createdOn: ZonedDateTime,
        name: Name,
        surname: Surname
    ) {
        guardUserExists(email)

        val user = SolutionUser.create(id, email, phoneNumber, createdOn, name, surname)

        repository.save(user)
        publisher.publish(user.pullEvents())
    }

    private fun guardUserExists(email: Email) {
        if (repository.existBy(email)) throw UserAlreadyExistsException()
    }
}
```

🟢 Casos de uso

```kotlin
class FunctionalUserCreator(
    private val repository: SolutionUserRepository,
    private val publisher: DomainEventPublisher
) {

    fun invokeBlock(
        id: UserId,
        email: Email,
        phoneNumber: PhoneNumber,
        createdOn: ZonedDateTime,
        name: Name,
        surname: Surname
    ): Either<CreateUserError, Unit> = either {
        guardUserExists(email).bind()

        val user = SolutionUser.create(id, email, phoneNumber, createdOn, name, surname)

        repository.save(user)
        publisher.publish(user.pullEvents())
    }

    private fun guardUserExists(email: Email)=
        if (repository.existBy(email)) UserAlreadyExists.left()
        else Unit.right()
}

sealed interface CreateUserError {
    object InvalidEmail : CreateUserError
    object InvalidName : CreateUserError
    object InvalidSurname : CreateUserError
    object InvalidMobilePhone : CreateUserError
    object UserAlreadyExists : CreateUserError
}
```

🟢 Error feedback

```kotlin
@RestController
class FunctionalCreateUserController(
    private val handler: FunctionalCreateUserCommandHandler,
    private val idGenerator: IdGenerator,
    private val clock: Clock
) {

    @PostMapping("/functional/users")
    fun create(@RequestBody body: CreateUserRequestBody): Response<*> =
        with(body) {
            handler.handle(
                FunctionalCreateUserCommand(
                    id = idGenerator.generate(),
                    email = email,
                    phonePrefix = phonePrefix,
                    phoneNumber = phoneNumber,
                    name = name,
                    surname = surname,
                    createdOn = clock.now()
                )
            ).toServerResponse(
                onError = { error -> error.toServerError() },
                onValidResponse = { Response.status(CREATED).withoutBody() }
            )
        }

    private fun CreateUserError.toServerError(): Response<*> =
        when(this) {
            is InvalidEmail -> Response.badRequest().body(INVALID_EMAIL)
            is InvalidMobilePhone -> Response.badRequest().body(INVALID_PHONE_NUMBER)
            is InvalidName -> Response.badRequest().body(INVALID_NAME)
            is InvalidSurname -> Response.badRequest().body(INVALID_SURNAME)
            is UserAlreadyExists -> Response.status(CONFLICT).body(USER_ALREADY_EXISTS)
        }
}
```

Recomendaciones

Recomendaciones

- 🟢 Mantener los errores "atómicos"

Recomendaciones

- Mantener los errores "atómicos"
- Utilizar '*when*' **exhaustivos** al gestionar errores

🔴 When exhaustivo

```
fun CreateUserError.toServerError(): Response<*> =
    when(this) {
        is UserAlreadyExists → Response.status(CONFLICT).body(USER_ALREADY_EXISTS)
        else → Response.badRequest().body(INVALID_PARAMETER)
    }
```

🟢 When exhaustivo

```
fun CreateUserError.toServerError(): Response<*> =
    when(this) {
        is InvalidEmail → Response.badRequest().body(INVALID_EMAIL)
        is InvalidMobilePhone → Response.badRequest().body(INVALID_PHONE_NUMBER)
        is InvalidName → Response.badRequest().body(INVALID_NAME)
        is InvalidSurname → Response.badRequest().body(INVALID_SURNAME)
        is UserAlreadyExists → Response.status(CONFLICT).body(USER_ALREADY_EXISTS)
    }
```

Recomendaciones

- 🟢 Mantener los errores "atómicos"
- 🟢 Utilizar '*when*' **exhaustivos** al gestionar errores

- 🟡 Modelar errores desconocidos (**Unknown**)

🟡 Modelar Unknown

```
sealed interface CreateUserError {
    object InvalidEmail : CreateUserError
    object InvalidName : CreateUserError
    object InvalidSurname : CreateUserError
    object InvalidMobilePhone : CreateUserError
    object UserAlreadyExists : CreateUserError
    // Avoid mapping this if you're on server
    class Unknown(val reason: Throwable) : CreateUserError
}
```

Recomendaciones

- 🟢 Mantener los errores "atómicos"
- 🟢 Utilizar '*when*' **exhaustivos** al gestionar errores

- 🟡 Modelar errores desconocidos (**Unknown**)
- 🟡 Either en clientes (**3rd parties**, **backend**)

🟡 Either en clientes

```
interface AuthenticationRepository {
    ...
    fun confirmSignUp(userId: UserId, confirmationCode: ConfirmationCode): ConfirmSignUpResult
    ...
}

sealed class ConfirmSignUpResult {
    object CodeMismatch : ConfirmSignUpResult()
    object ExpiredCode : ConfirmSignUpResult()
    object ExceededAttempts : ConfirmSignUpResult()
    object UserNotAuthorized : ConfirmSignUpResult()
    object InvalidParameter : ConfirmSignUpResult()
    object Success : ConfirmSignUpResult()
}
```

🟡 Either en clientes

```
interface AuthenticationRepository {
    ...
    fun confirmSignUp(userId: UserId, confirmationCode: ConfirmationCode): Either<ConfirmSignUpResult, Unit>
    ...
}

sealed class ConfirmSignUpError {
    object CodeMismatch : ConfirmSignUpResult()
    object ExpiredCode : ConfirmSignUpResult()
    object ExceededAttempts : ConfirmSignUpResult()
    object UserNotAuthorized : ConfirmSignUpResult()
    object InvalidParameter : ConfirmSignUpResult()
}
```

Recomendaciones

- 🟢 Mantener los errores "atómicos"
- 🟢 Utilizar '*when*' **exhaustivos** al gestionar errores

- 🟡 Modelar errores desconocidos (**Unknown**)
- 🟡 Either en clientes (**3rd parties**, **backend**)
- 🟡 Cuidado con el uso de bind()

🟡 Bind

```
fun invoke(userId: UserId): Either<VerifyUserError, Unit> = either.eager {
    val user = findUser(userId)

    val verifiedUser = user.safeVerify().bind() // ⟵ shortcircuits on left (error)

    repository.save(verifiedUser)
    publisher.publish(verifiedUser.pullEvents())
}

private fun findUser(id: UserId): Either<VerifyUserError, User> =
    repository.find(ById(userId))?.right() ?: UserNotFound.left().bind() // ⟵ Throws error
```

🟡 Bind

```
fun invoke(userId: UserId): Either<VerifyUserError, Unit> = either.eager {
    val user = findUser(userId).bind()

    val verifiedUser = user.safeVerify().bind()

    repository.save(verifiedUser)
    publisher.publish(verifiedUser.pullEvents())
}

private fun findUser(userId: UserId): Either<VerifyUserError, User> =
    repository.find(ById(userId))?.right() ?: UserNotFound.left()
```

Recomendaciones

🟢 Mantener los errores "atómicos"
🟢 Utilizar '*when*' **exhaustivos** al gestionar errores

🟡 Modelar errores desconocidos (**Unknown**)
🟡 Either en clientes (**3rd parties**, **backend**)
🟡 Cuidado con el uso de bind()

🔴 Either en código que no pueda fallar

🔴 Either en código que no falla

```
fun create(
    id: UserId,
    email: Email,
    phoneNumber: PhoneNumber,
    createdOn: ZonedDateTime,
    name: Name,
    surname: Surname,
    documents: Set<Document> = emptySet(),
    status: Status = INCOMPLETE,
    cardStatus: CardStatus = PENDING
): Either<Throwable, SolutionUser> = catch {
    SolutionUser(id, email, phoneNumber, createdOn, name, surname, documents, status, cardStatus)
        .also {
        it.pushEvent(it.toUserCreatedEvent(id, email, phoneNumber, name, surname, createdOn, status, cardStatus))
    }
}
```

🟢 Either en código que no falla

```
companion object {
    fun create(
        id: UserId,
        email: Email,
        phoneNumber: PhoneNumber,
        createdOn: ZonedDateTime,
        name: Name,
        surname: Surname,
        documents: Set<Document> = emptySet(),
        status: Status = INCOMPLETE,
        cardStatus: CardStatus = PENDING
    ) = SolutionUser(id, email, phoneNumber, createdOn, name, surname, documents, status, cardStatus)
        .also {
            it.pushEvent(it.toUserCreatedEvent(id, email, phoneNumber, name, surname, createdOn, status, cardStatus))
        }
```

Recomendaciones

🟢 Mantener los errores "atómicos"
🟢 Utilizar '*when*' **exhaustivos** al gestionar errores

🟡 Modelar errores desconocidos (**Unknown**)
🟡 Either en clientes (**3rd parties**, **backend**)
🟡 Cuidado con el uso de bind()

🔴 Either en código que no pueda fallar
🔴 Either en repositorios de bases de datos

🔴 Either en repos

```
interface SolutionUserRepository {
    fun find(criteria: FindUserCriteria): Either<Throwable, SolutionUser>
    fun exists(criteria: ExistsUserCriteria): Either<Throwable, Boolean>
}

sealed interface ExistsUserCriteria {
    class ById(val id: UserId) : ExistsUserCriteria
    class ByEmail(val email: Email) : ExistsUserCriteria
}

sealed interface FindUserCriteria {
    class ById(val id: UserId) : FindUserCriteria
}
```

🟢 Either en repos

```
interface SolutionUserRepository {
    fun save(user: SolutionUser)
    fun find(criteria: FindUserCriteria): SolutionUser?
    fun exists(criteria: ExistsUserCriteria): Boolean
}

sealed interface ExistsUserCriteria {
    class ById(val id: UserId) : ExistsUserCriteria
    class ByEmail(val email: Email) : ExistsUserCriteria
}

sealed interface FindUserCriteria {
    class ById(val id: UserId) : FindUserCriteria
}
```

Recomendaciones

- 🟢 Mantener los errores "atómicos"
- 🟢 Utilizar '*when*' **exhaustivos** al gestionar errores

- 🟡 Modelar errores desconocidos (**Unknown**)
- 🟡 Either en clientes (**3rd parties**, **backend**)
- 🟡 Cuidado con el uso de bind()

- 🔴 Either en código que no pueda fallar
- 🔴 Either en repositorios de bases de datos
- 🔴 Reutilizar modelos de errores

Recursos

Artículos y Conferencias

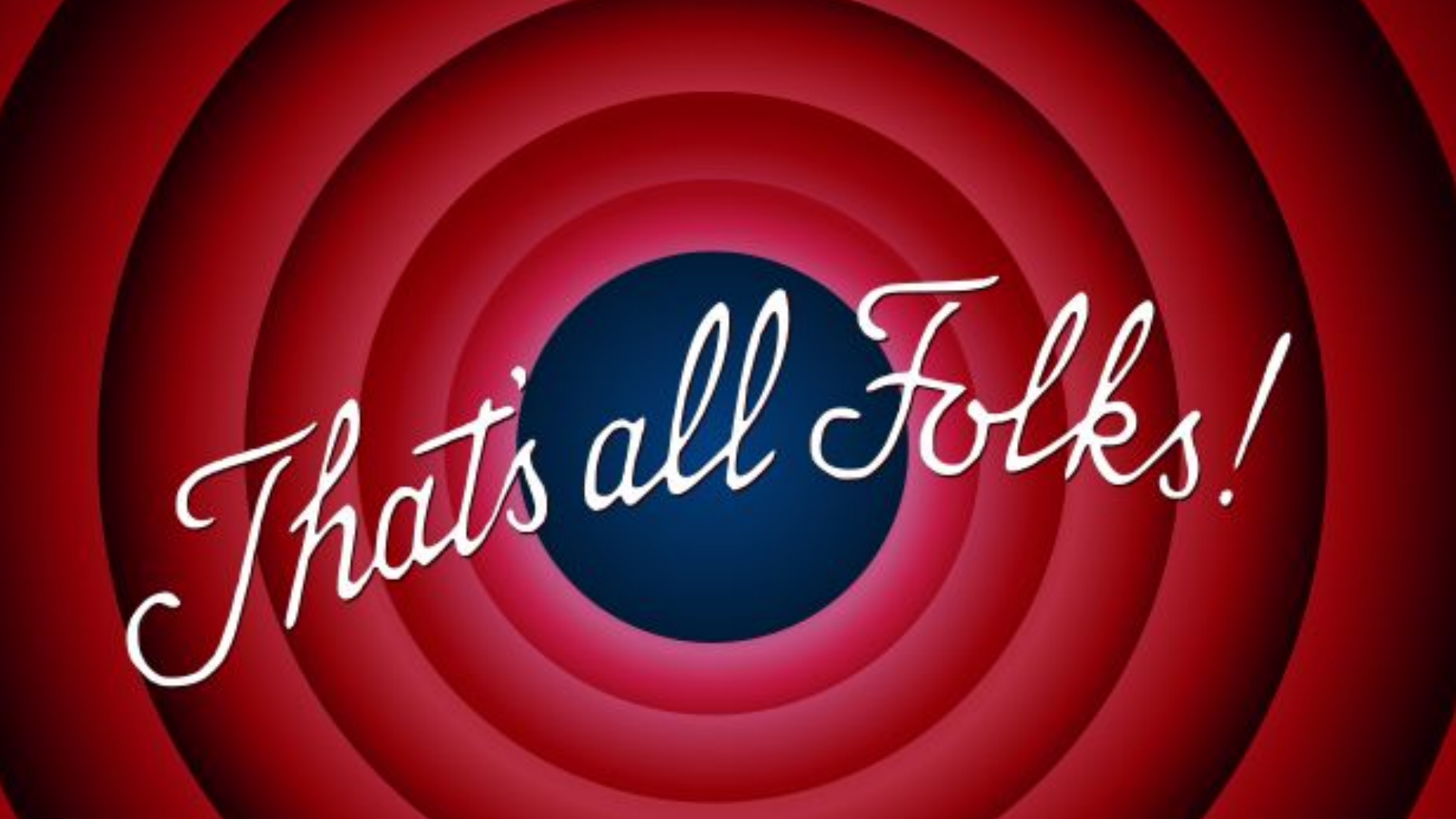
1. Exceptional Performance - Aleksey Shipilev
2. Functional Domain Modeling - Simon Vergauwen
3. Sealed Classes to model State - Patrick Cousins
4. Giravolta backend Arrow ADR

Librerías funcionales (JVM)

1. Arrow - Kotlin
2. Vavr - Java

Repositorio

1. Demo


That's all Folks!