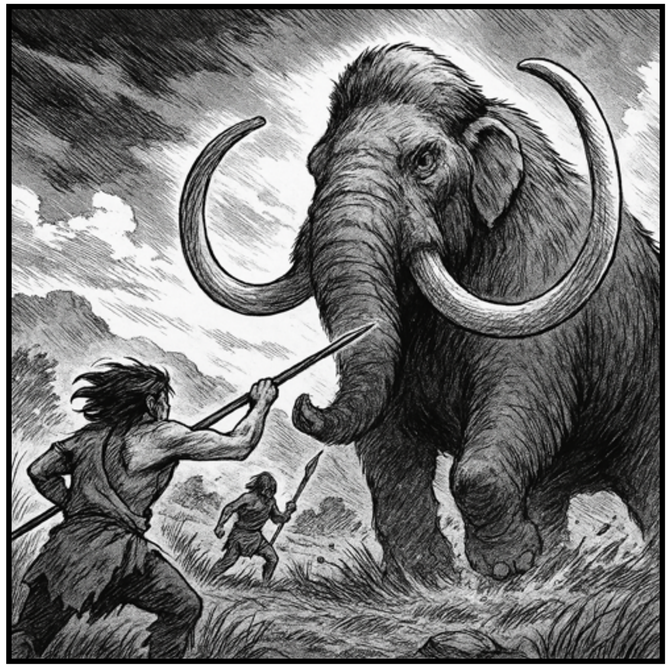
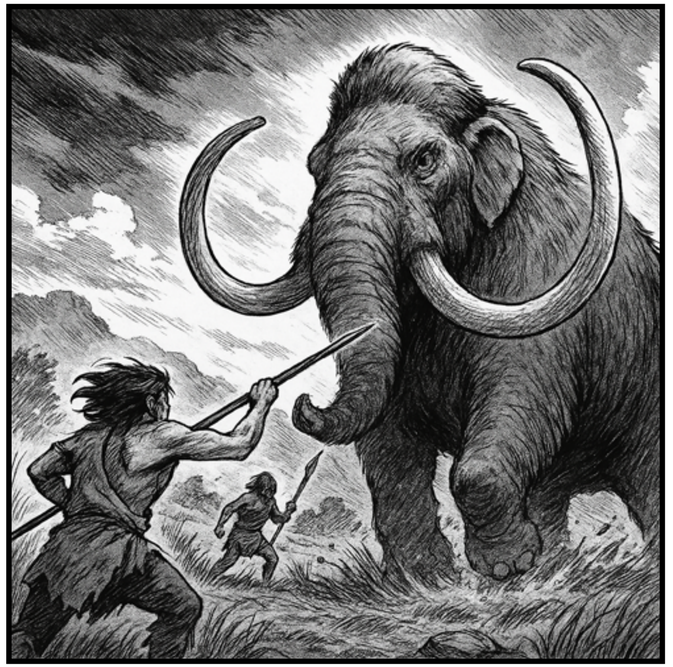
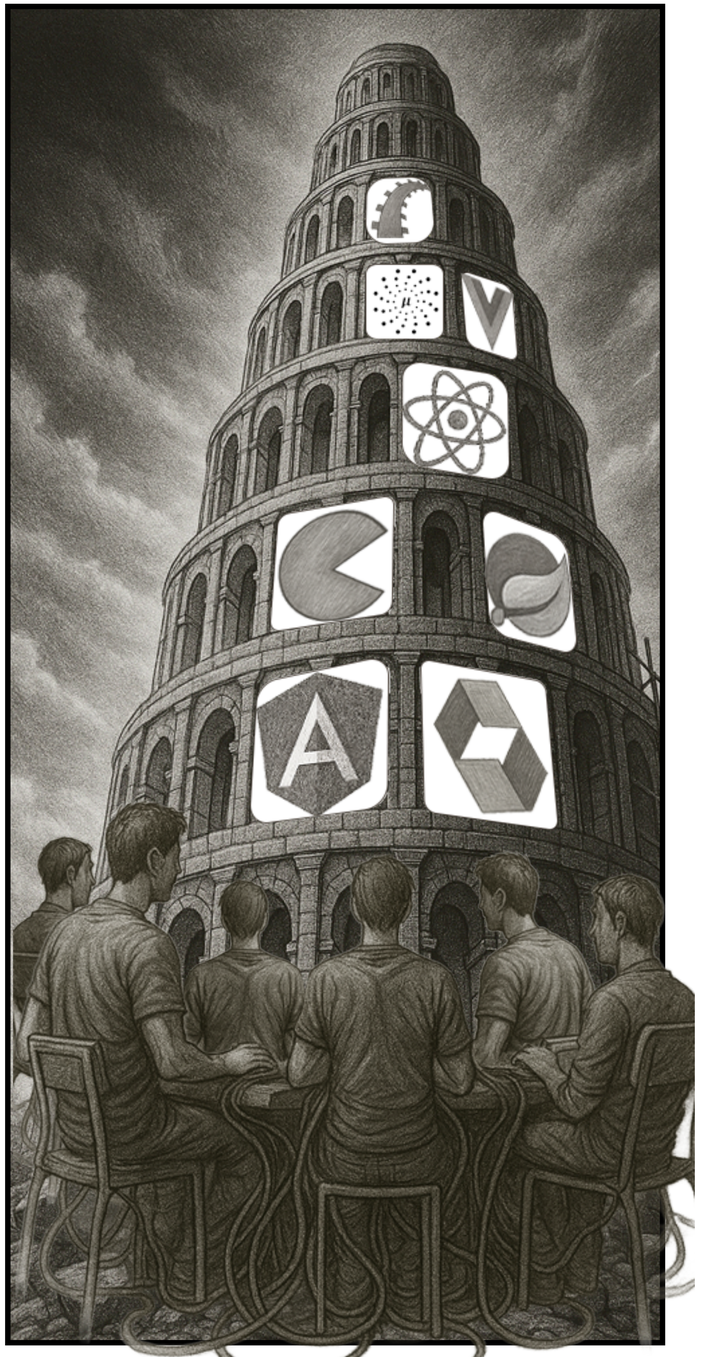
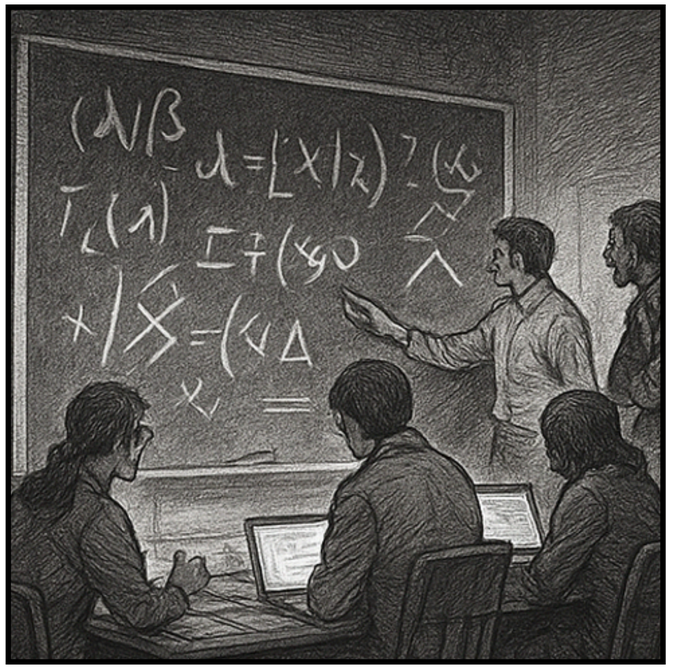
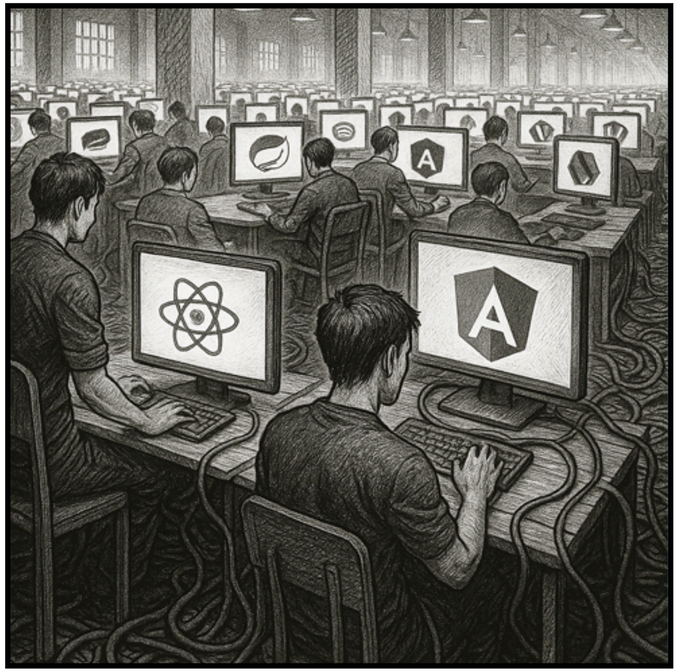


Images generated and refined using AI tools. Framework logos © their respective creators — illustrated here as part of a metaphorical narrative.









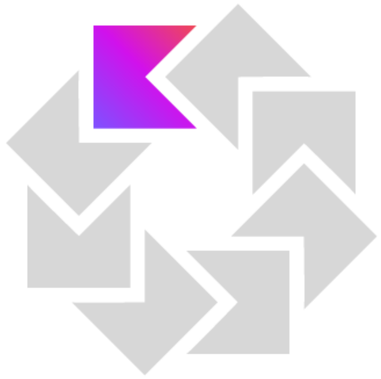
Images generated and refined using AI tools. Framework logos © their respective creators — illustrated here as part of a metaphorical narrative.

Own Your Design

Tiberiu Tofan

About me

- ⚒️ **4+** languages: Java, Kotlin, Scala, Typescript
- 💜 **7+** years since I’m having fun with Kotlin
- 👥 Co-organizer of BUK — Bucharest Kotlin User Group
- 🚀 **20+** years on the JVM — still warming up
- 🧩 **100%** committed to simplify things — eventually
- 🌾 ∞ abstractions harvested, few truly needed
- 🦣 **0** mammoths left to hunt, but plenty of legacy code



BUK

Layered architecture

Presentation

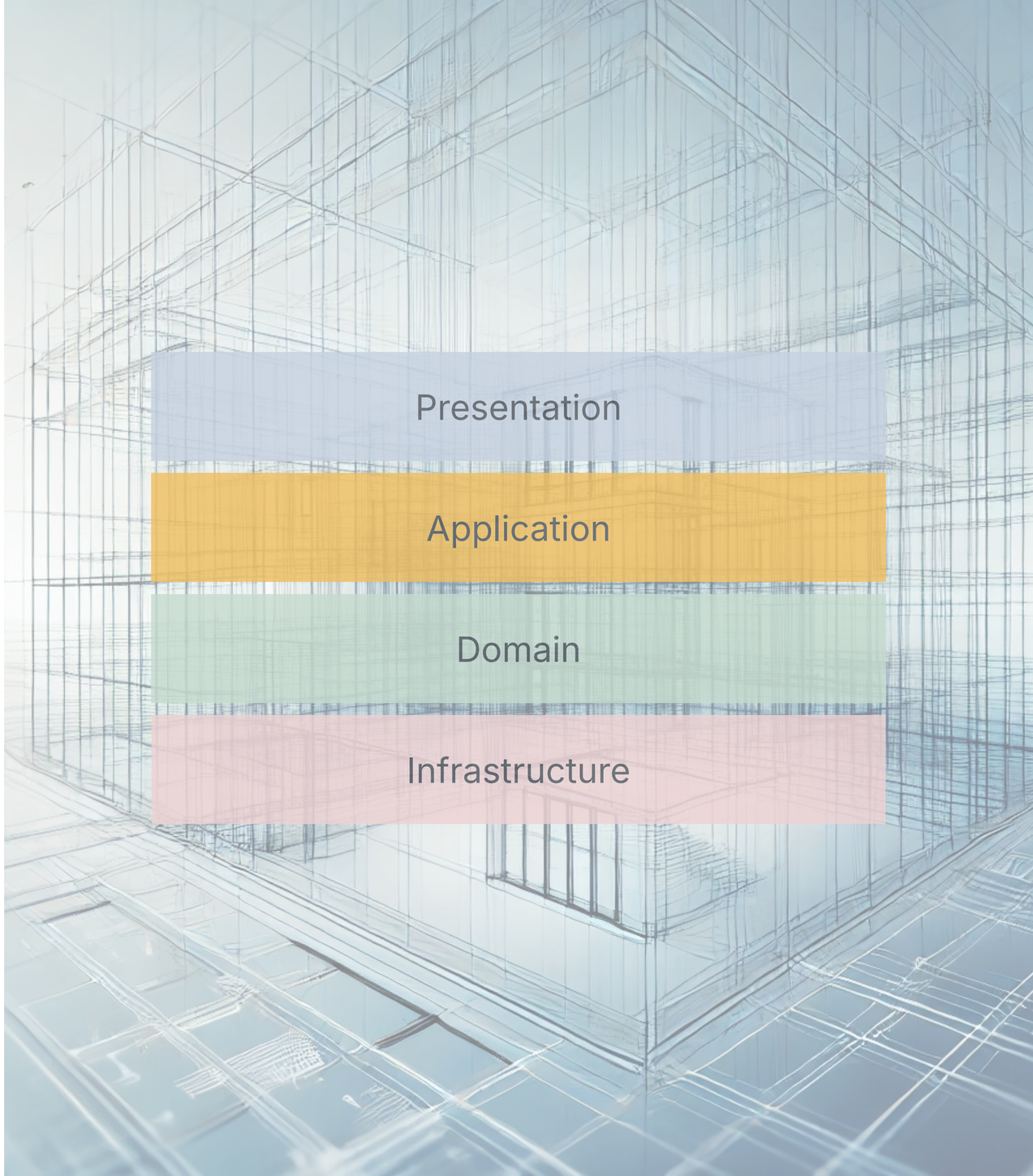
Application

Domain

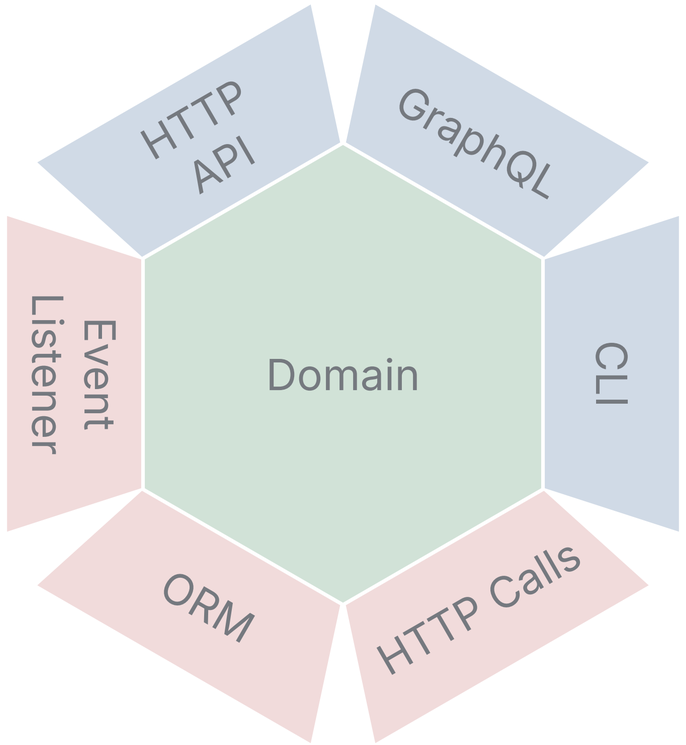
Infrastructure

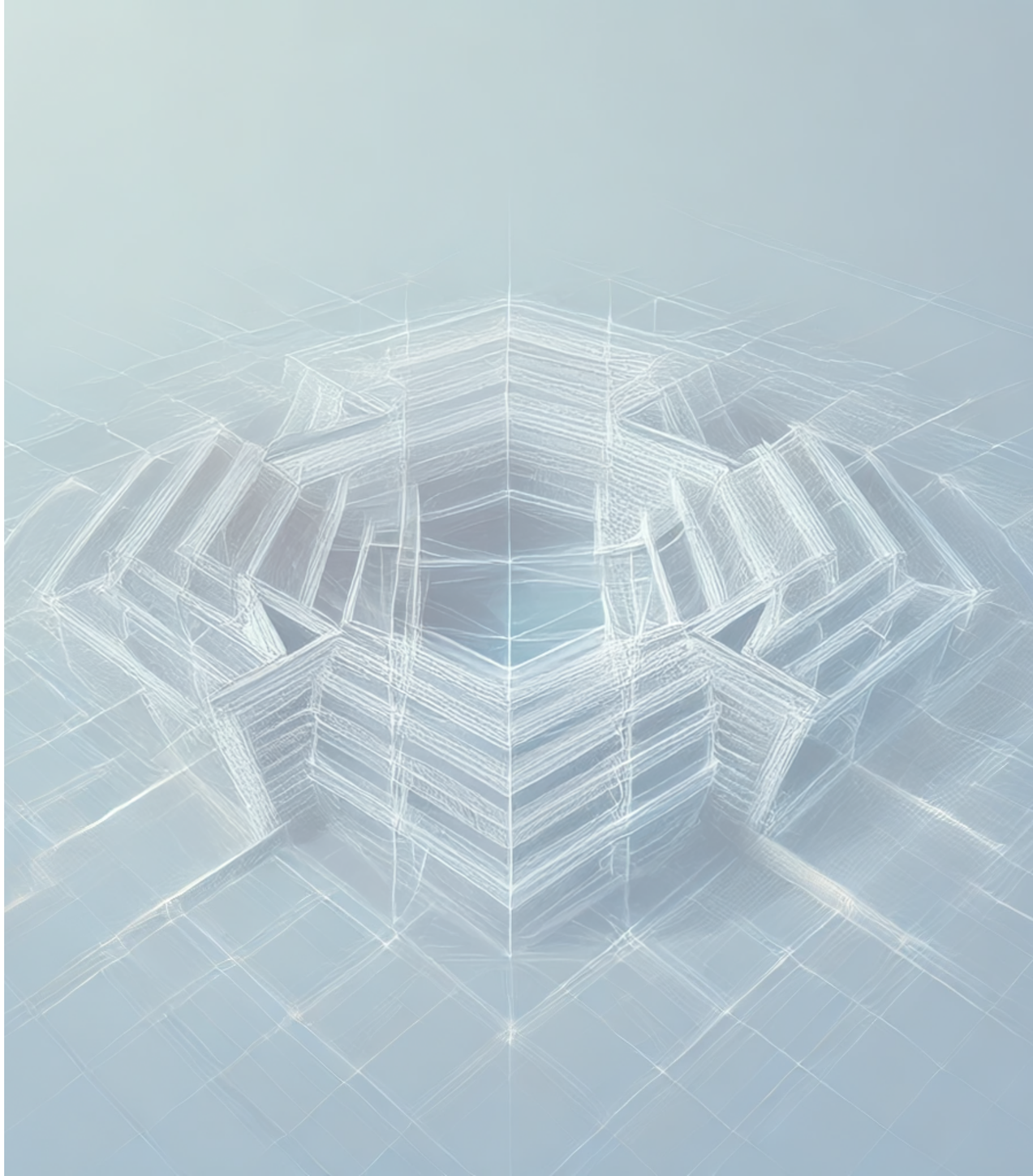
Layered architecture

- **Tight coupling**: Business logic depends on specific technologies such as ORM, framework, and external APIs.
- **Difficult testing**: Hard to isolate business logic from infrastructure; requires extensive mocking.
- **Rigid**: Changing technology (DB, messaging, APIs) requires significant rewrites.



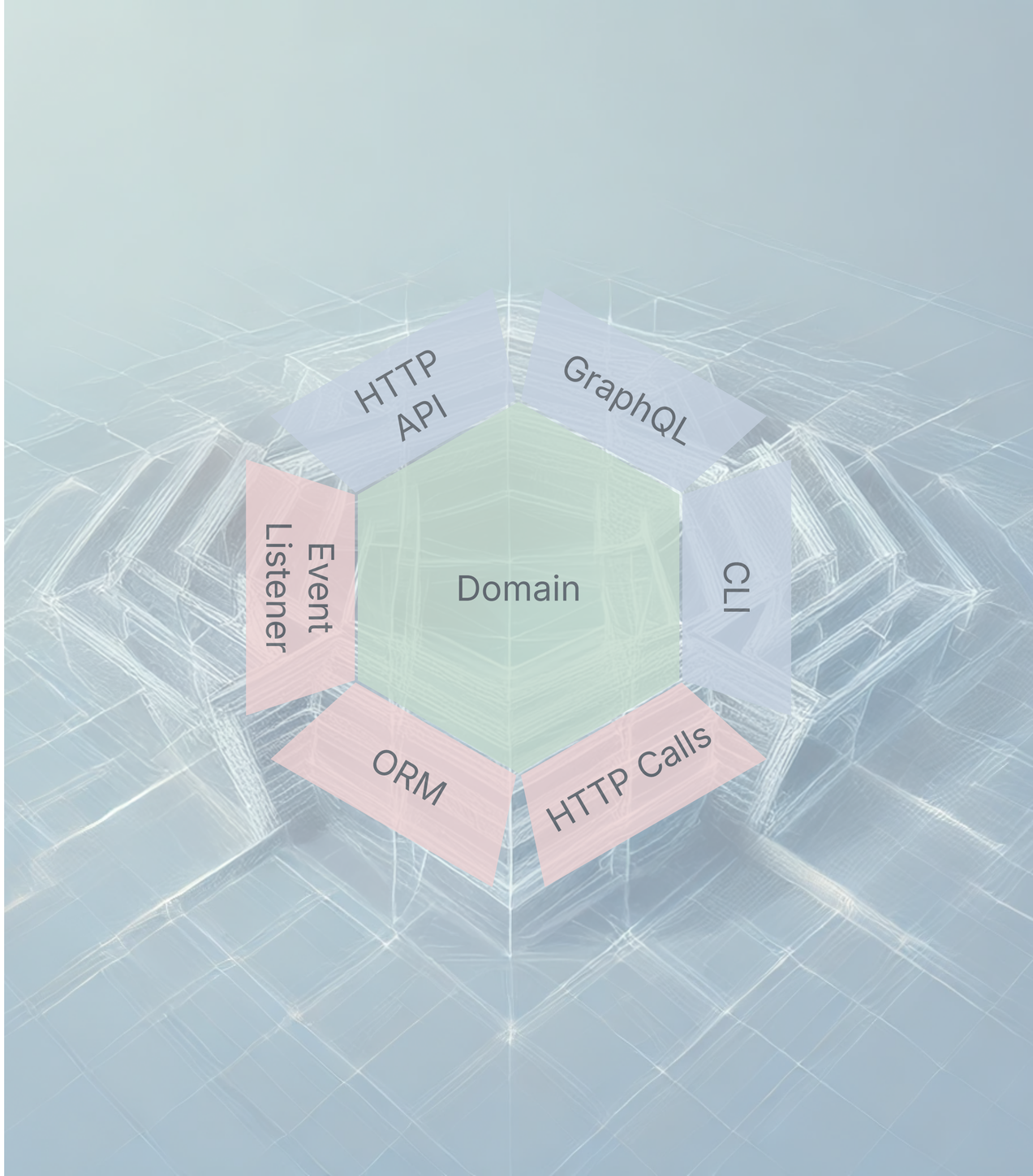
Hexagonal architecture



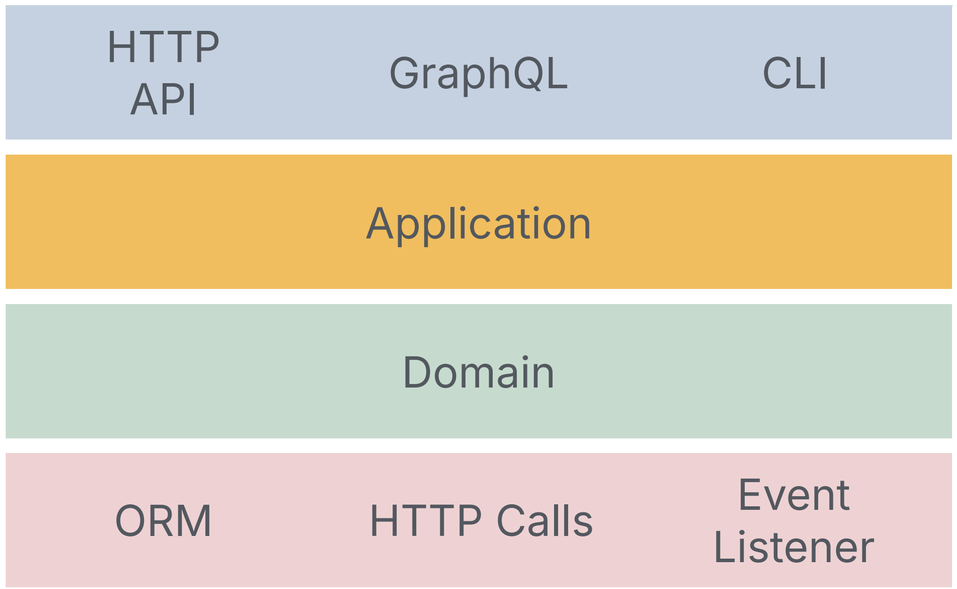


Hexagonal architecture

- **Decoupled**: Business logic is independent on database, APIs or frameworks.
- **Easier testing**: Ports allow for clean unit testing using fake implementations.
- **Flexible**: Swapping any technology requires only changing the adapter.

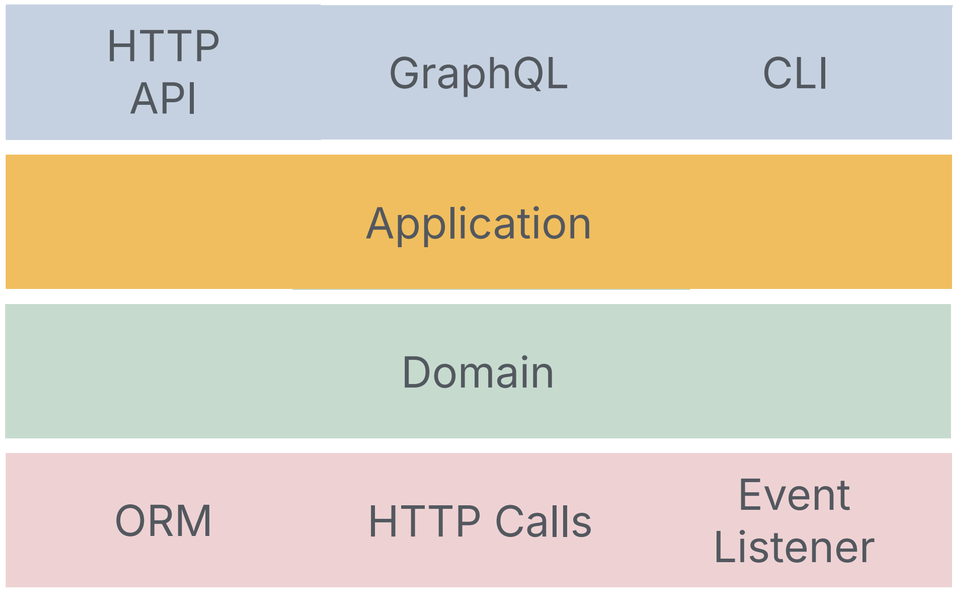


Layered



Hexagonal

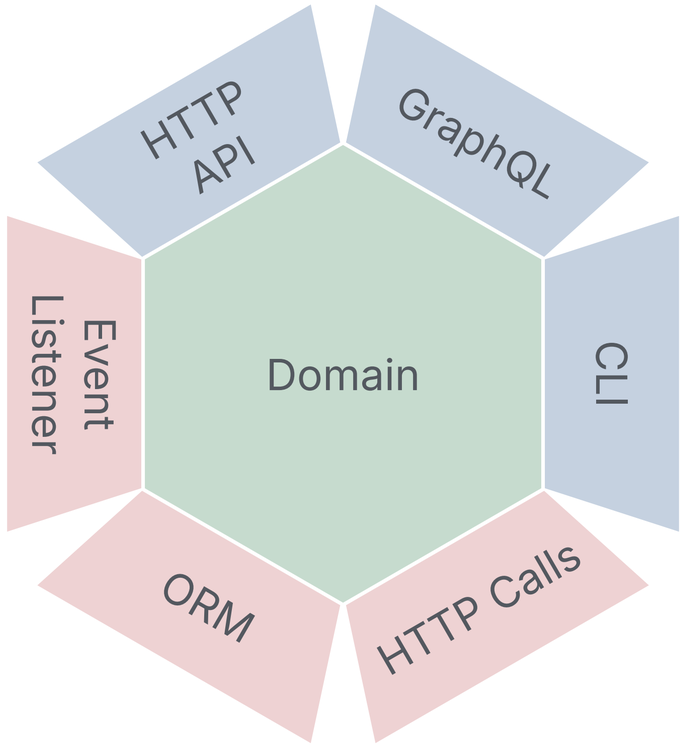
Layered



Hexagonal

Layered

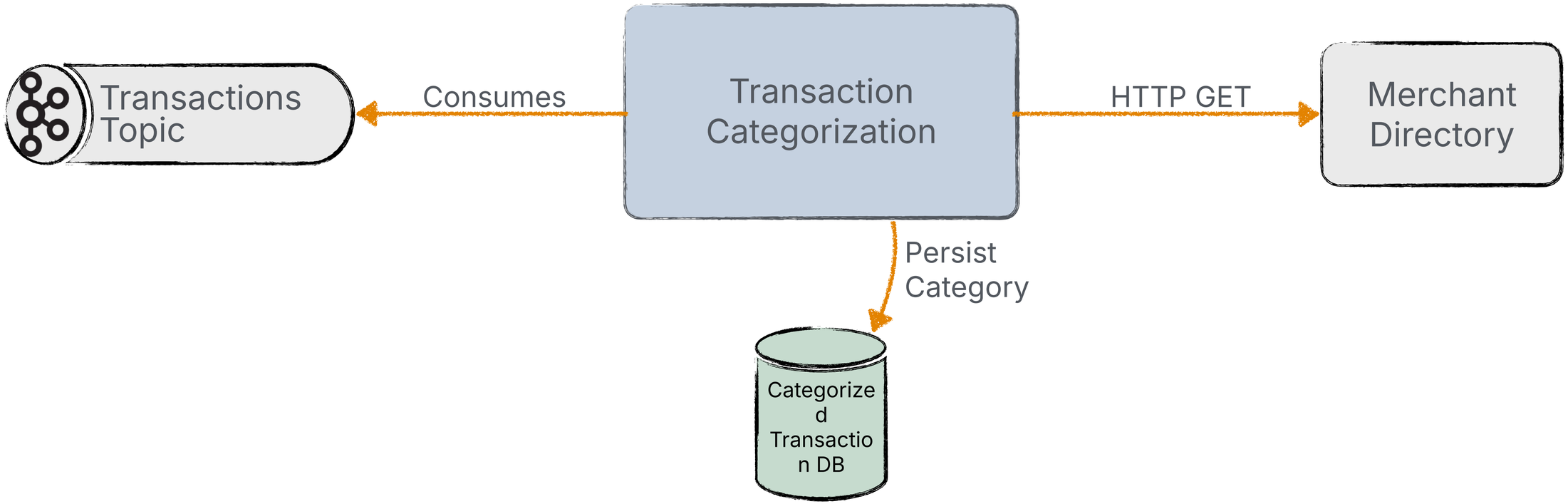
Hexagonal



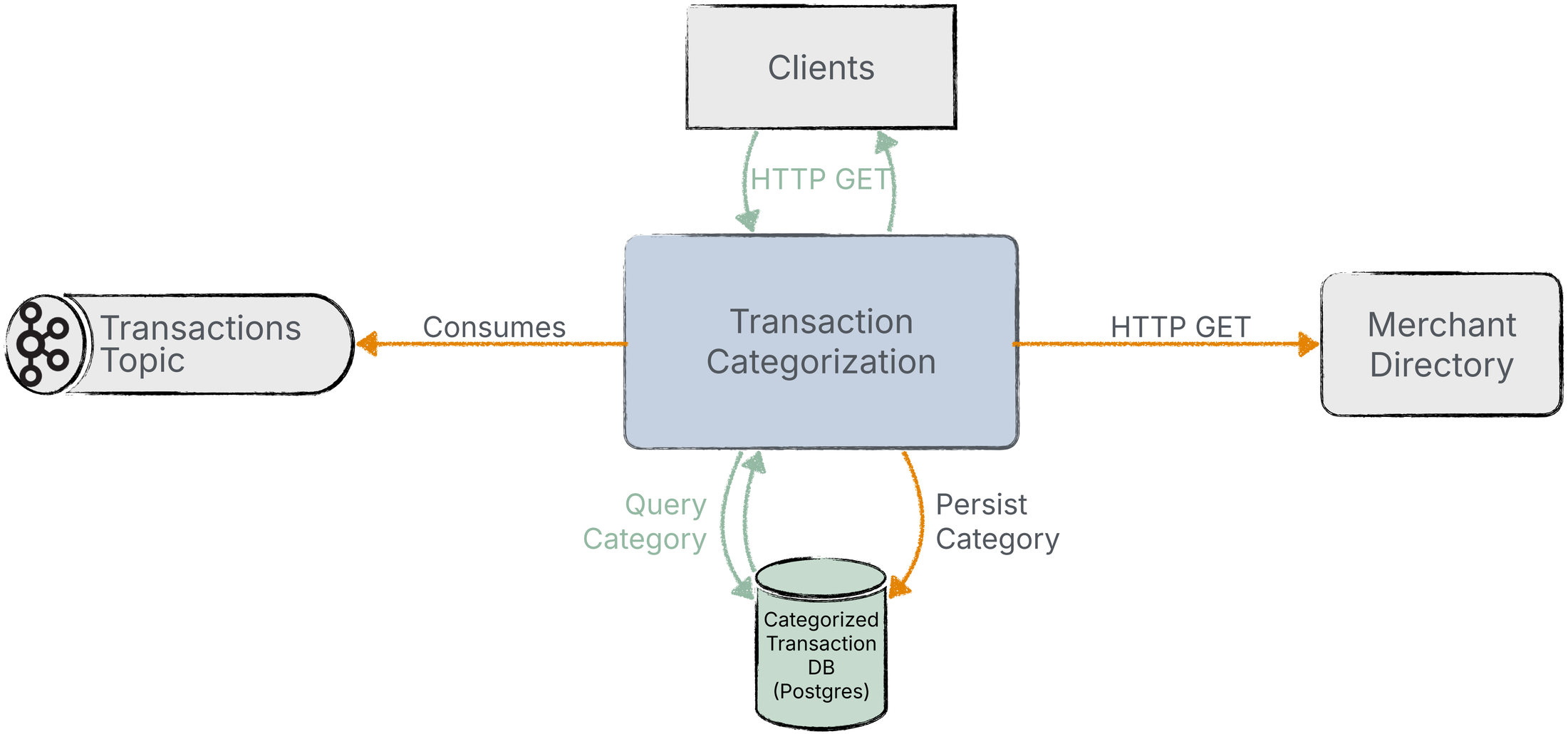
Case study:

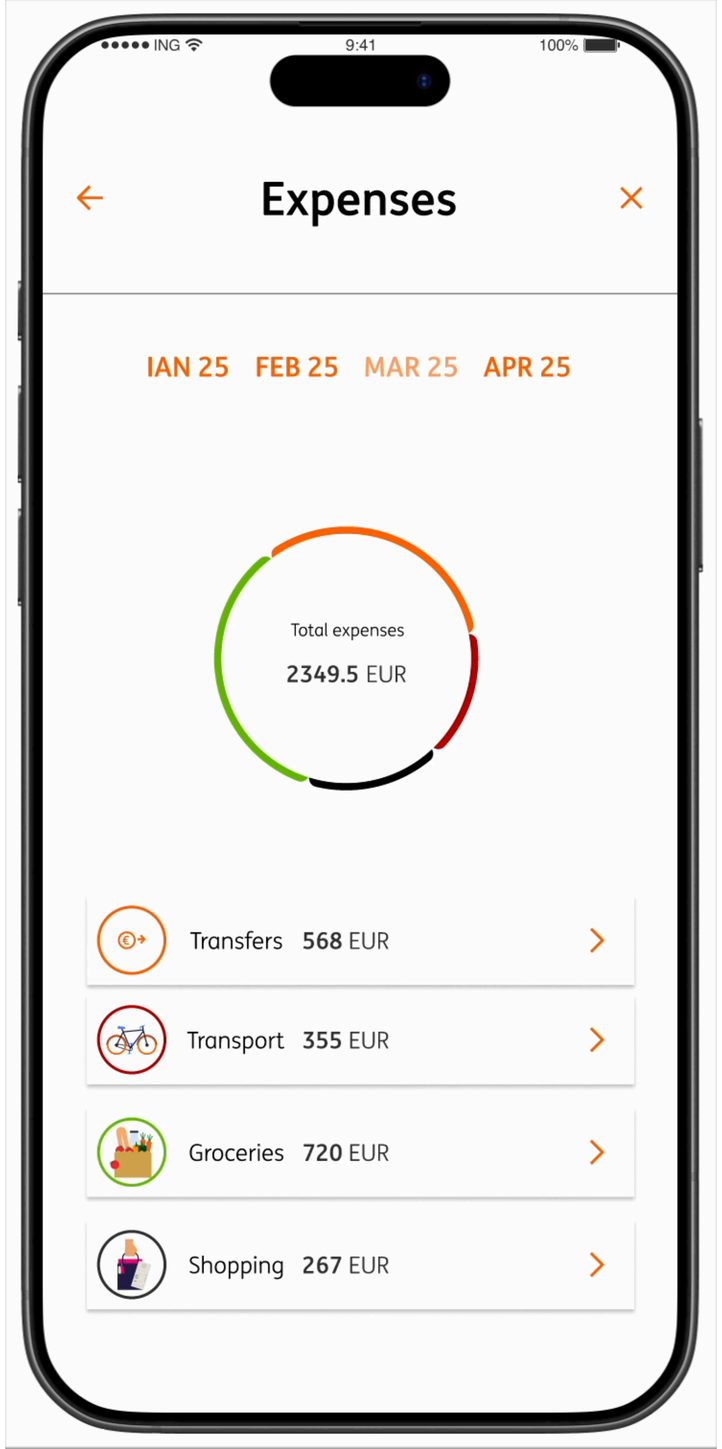
Transaction Categorization Service

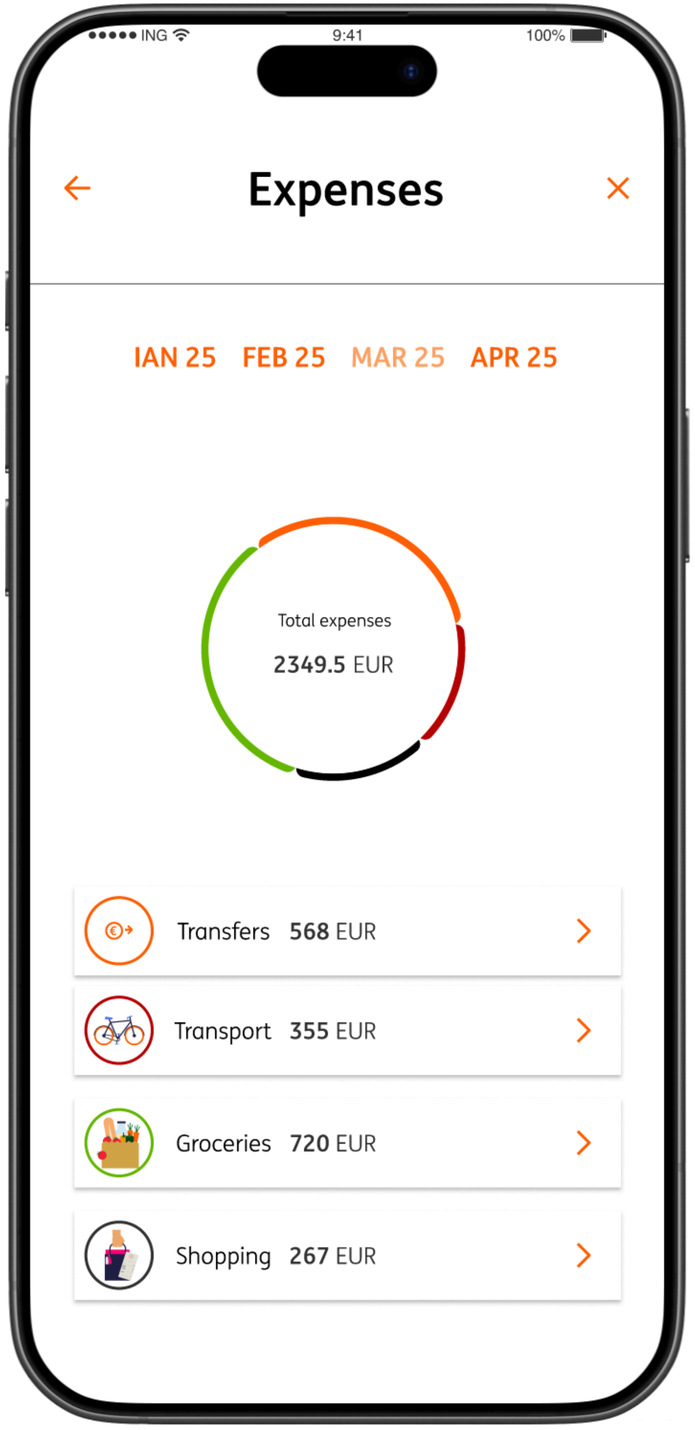
Transaction Categorization Service



Transaction Categorization Service



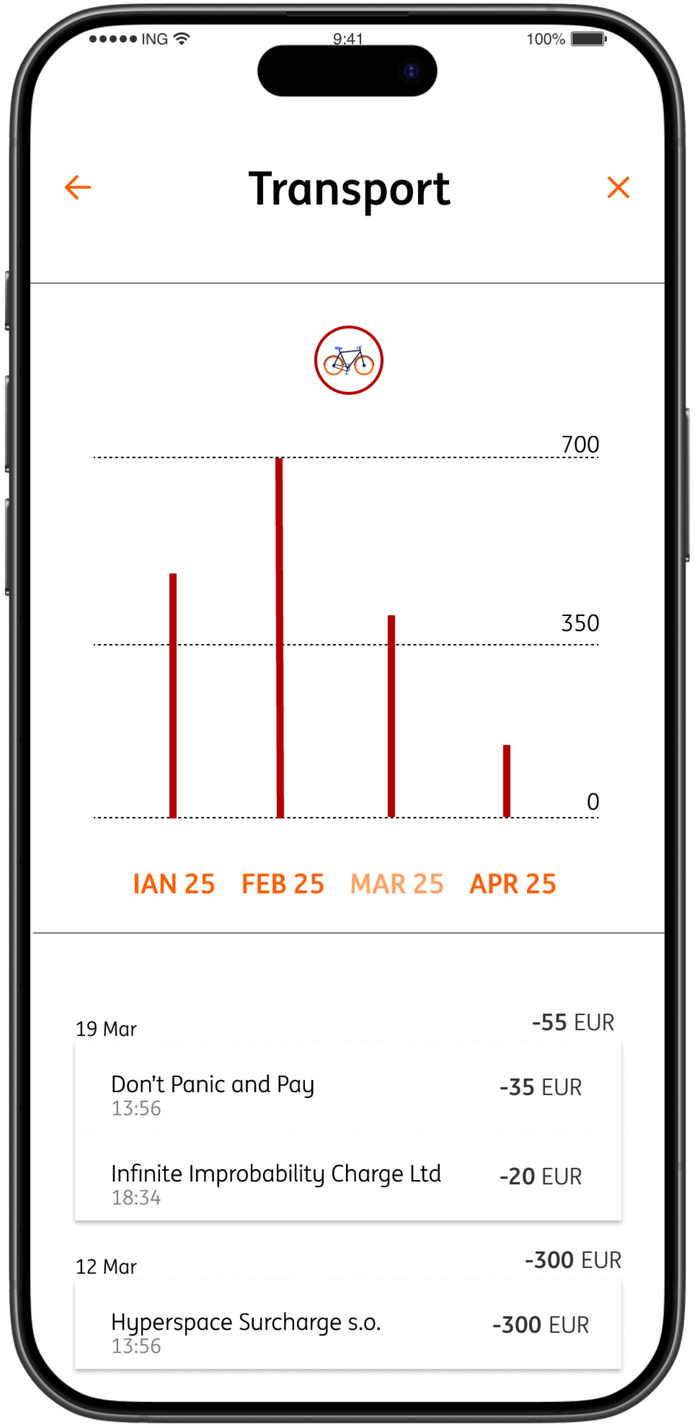

ING
9:41
100%
Expenses
IAN 25
FEB 25
MAR 25
APR 25
Total expenses
2349.5 EUR
Transfers 568 EUR
Transport 355 EUR
Groceries 720 EUR
Shopping 267 EUR



GET /client/{clientId}/categories-budget

Transaction Categorization

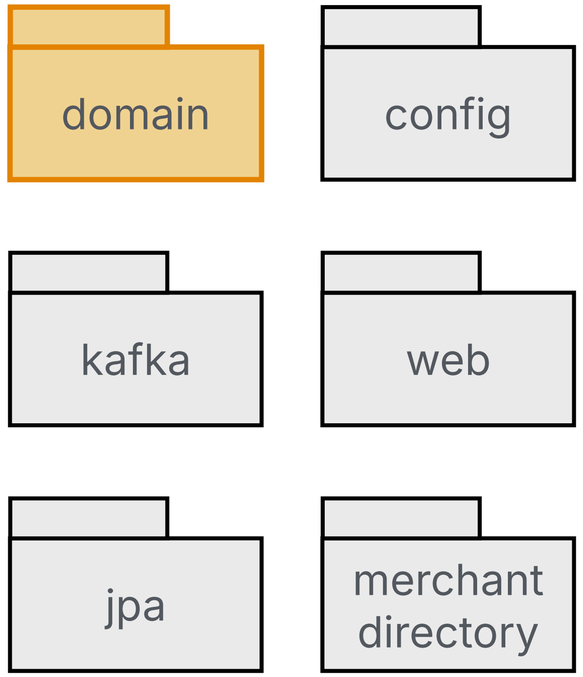
GET /client/{clientId}/transactions?category=transport



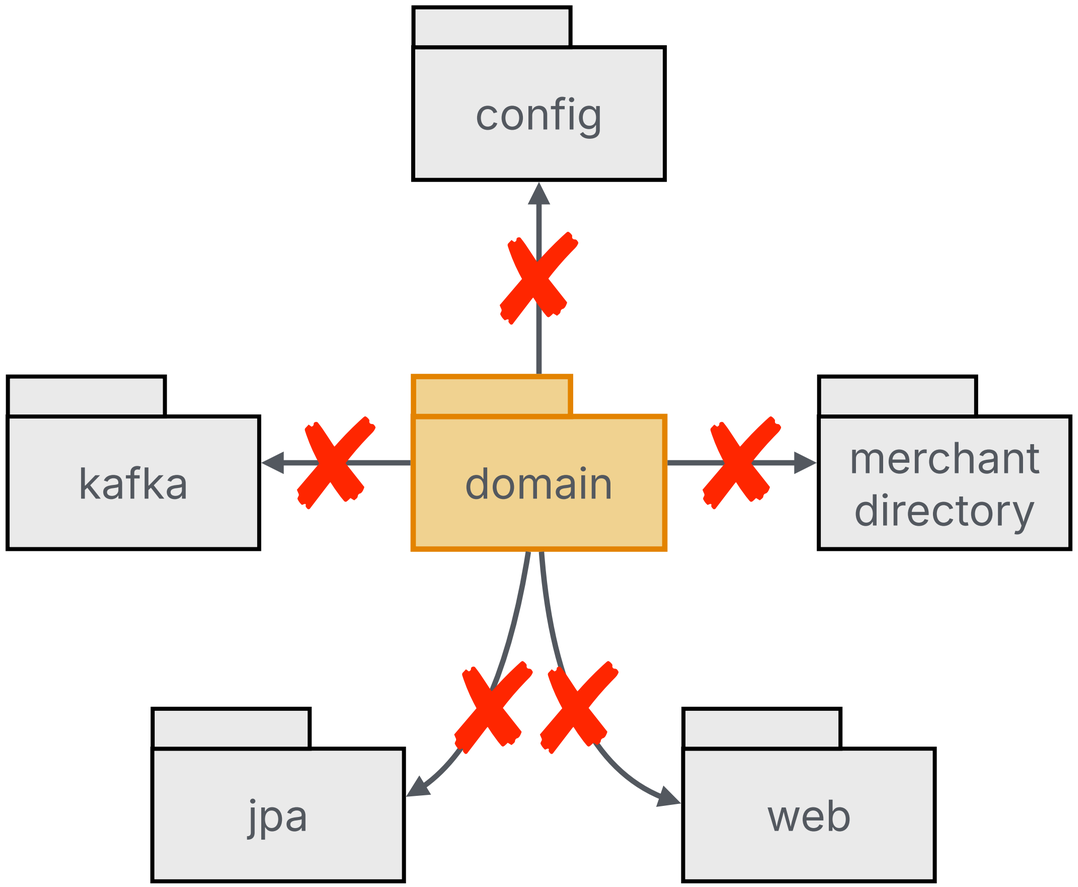
Build a resilient domain

Decouple, isolate, and validate

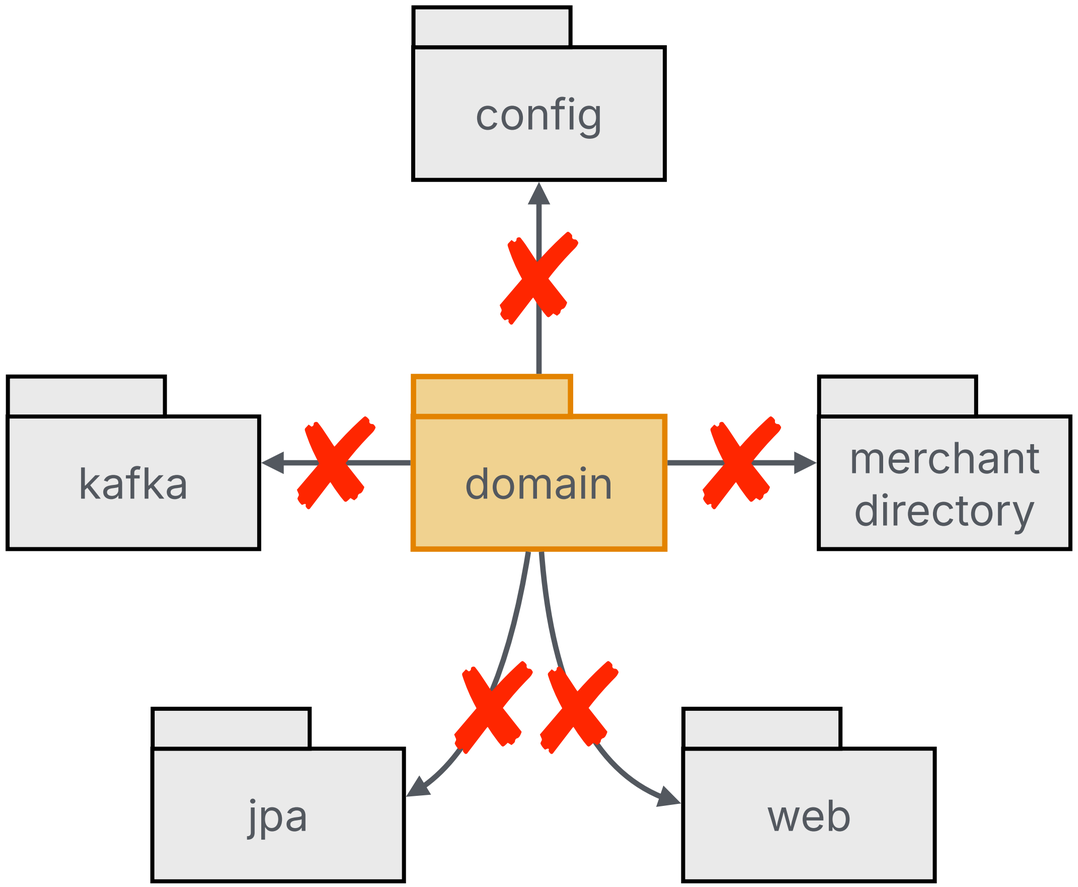
Decoupled infrastructure



Decoupled infrastructure

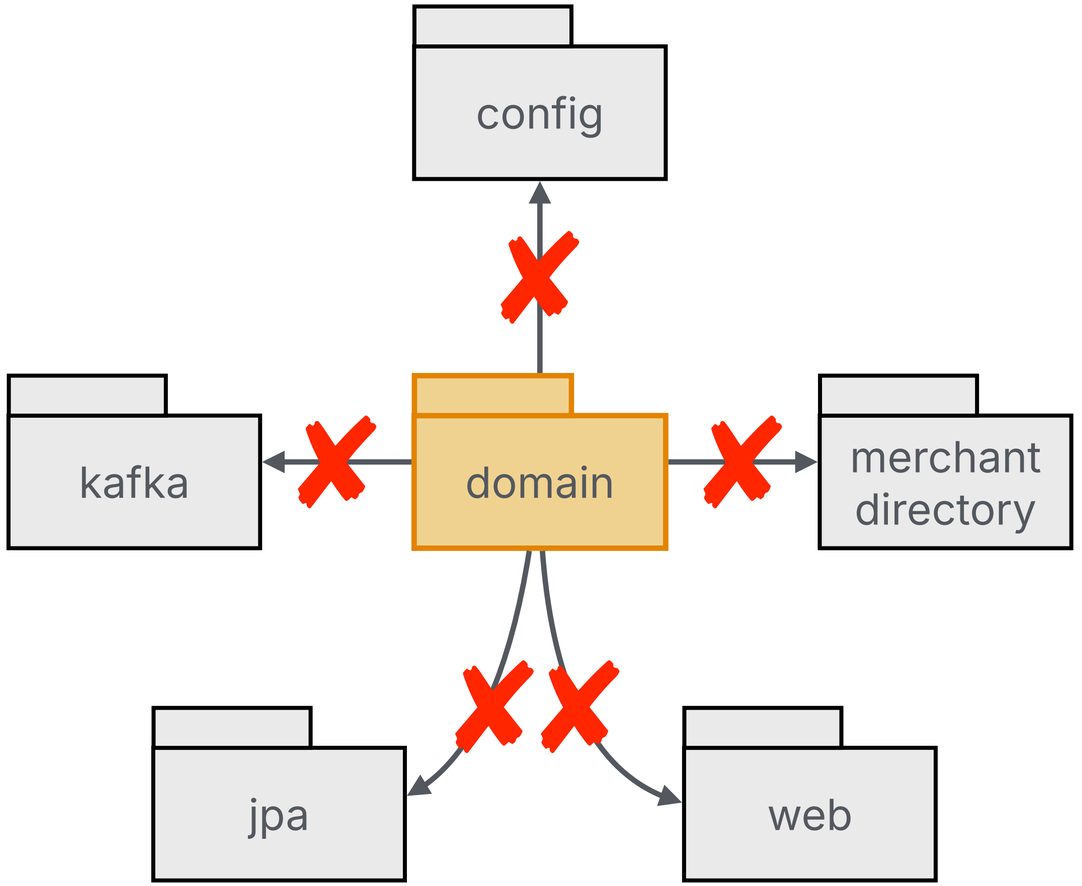


Decoupled infrastructure

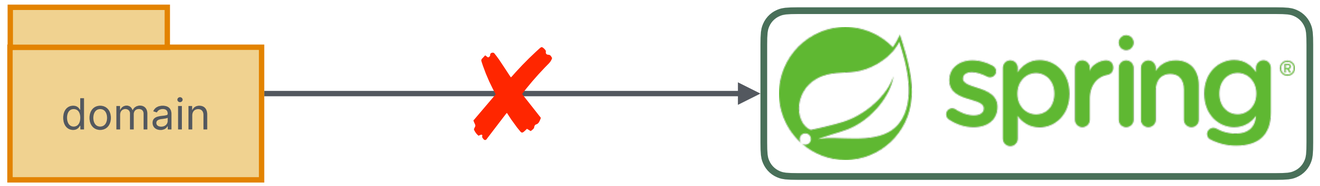


Decoupled infrastructure

```
Konsist
    .scopeFromProduction()
    .assertArchitecture {
        val domain = Layer("Domain", "..domain..")
        val configuration = Layer("Config", "..infra.config..")
        val kafka = Layer("Kafka", "..infra.kafka..")
        val web = Layer("Web", "..infra.web..")
        val jpa = Layer("JPA", "..infra.jpa..")
        val merchantDirectory = Layer("MerchDir", "..infra.md..")

        domain.doesNotDependOn(
            configuration,
            kafka,
            web,
            jpa,
            merchantDirectory
        )
    }
```

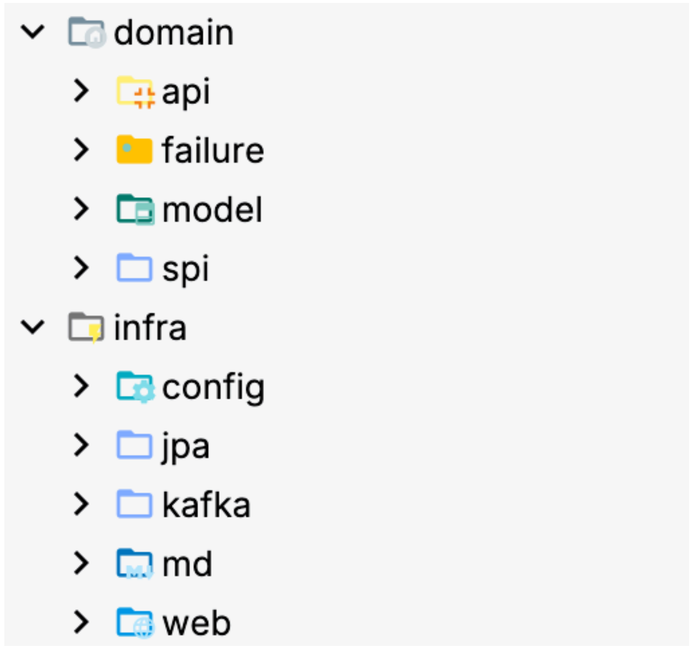


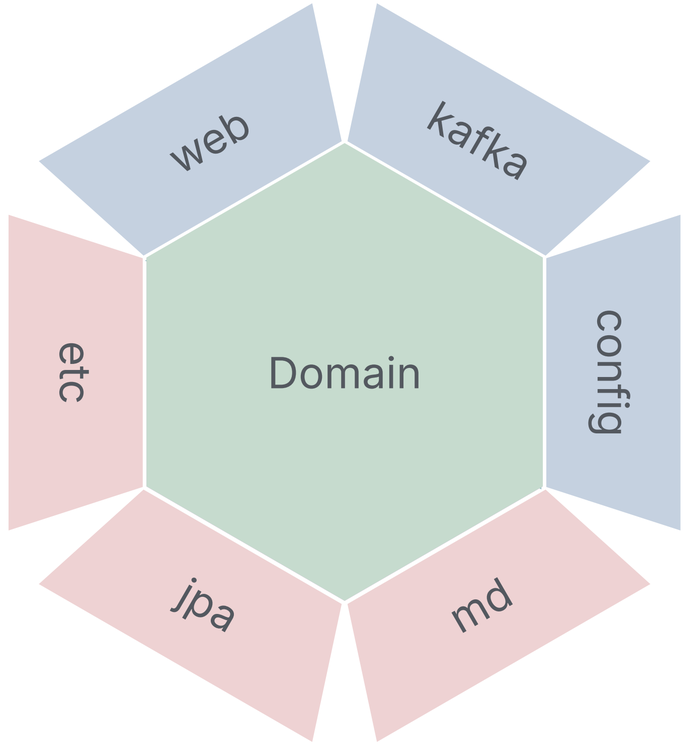
Decoupled Frameworks



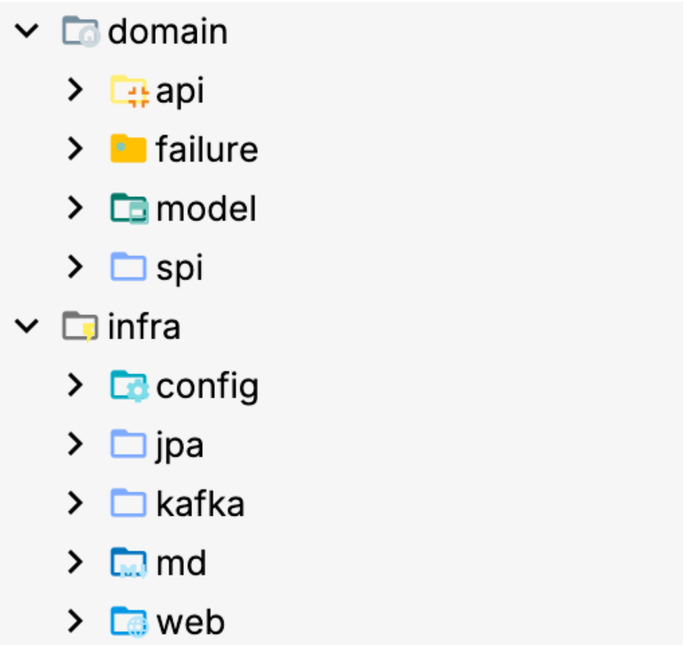

```
Konsist
    .scopeFromProduction()
    .classes()
    .withPackage("fvf4k.demo.domain..")
    .assertFalse { classDeclaration →
        classDeclaration
            .containingFile
            .imports
            .any { import →
                import.hasNameStartingWith("org.springframework")
            }
    }
```

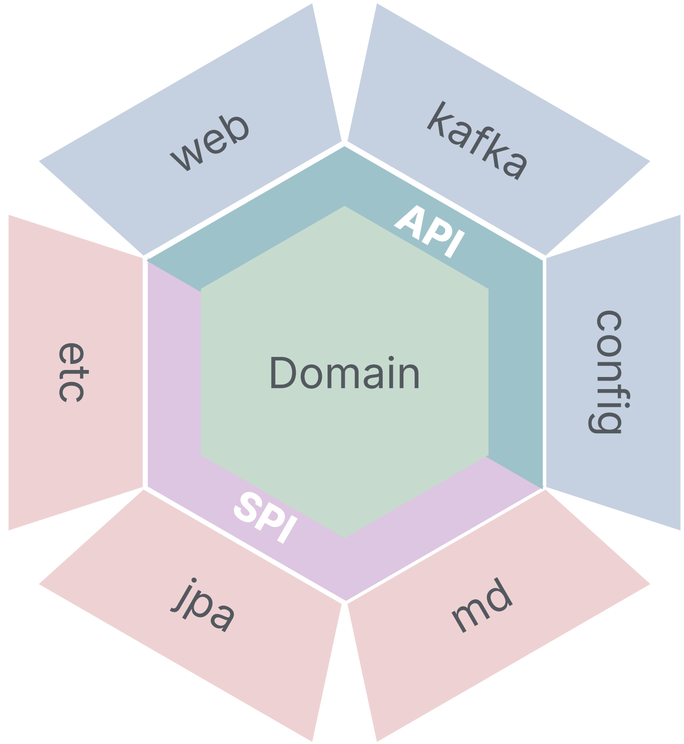
Hex architectural rules





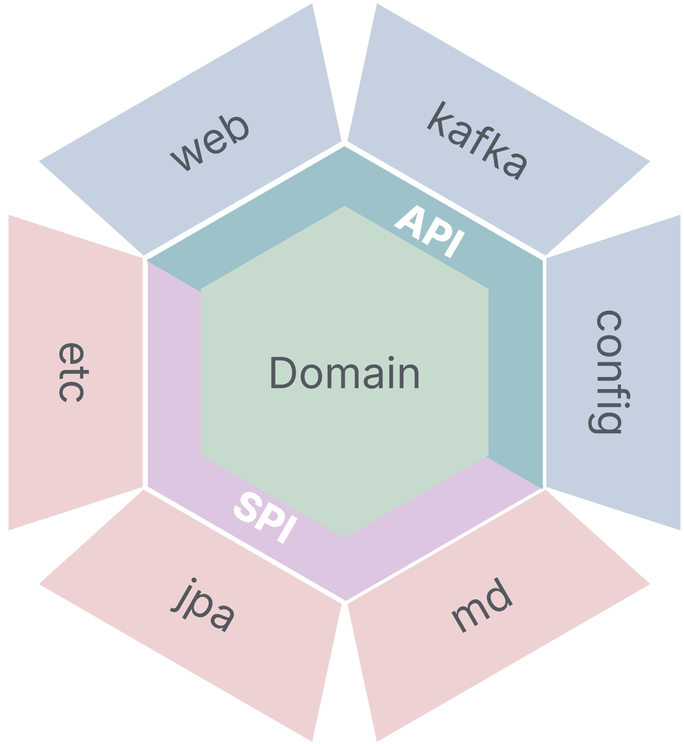
Hex architectural rules





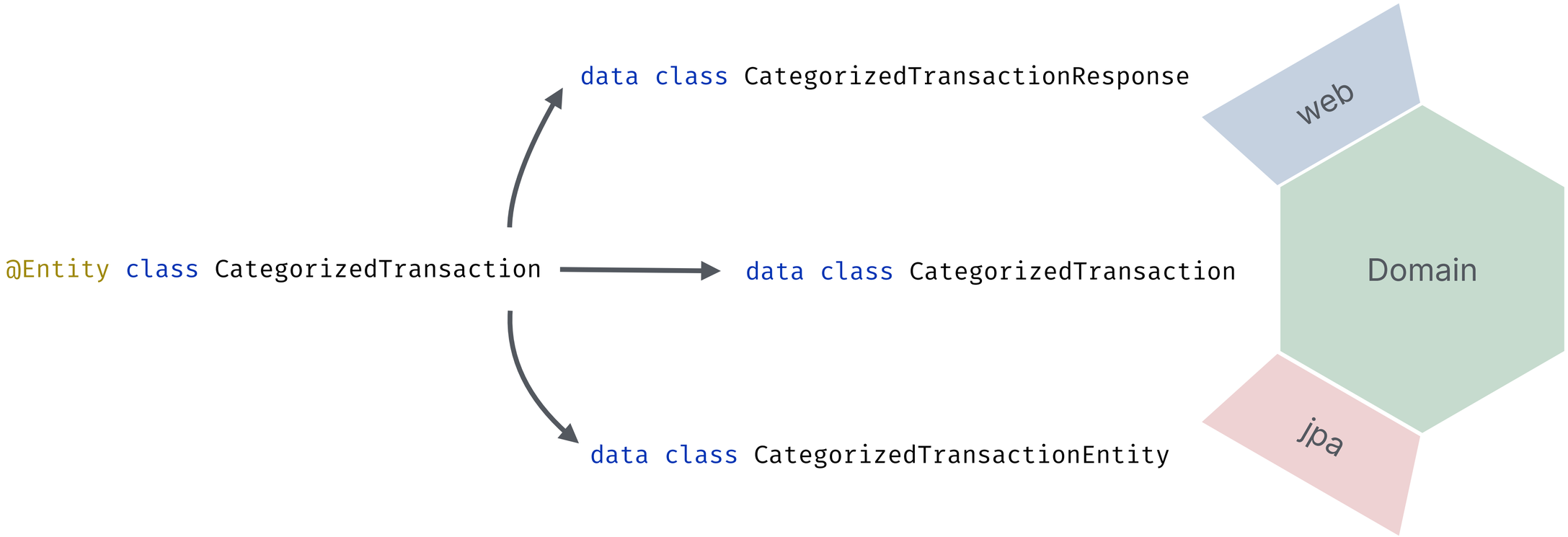
Hex architectural rules

```
Konsist
    .scopeFromProduction()
    .assertArchitecture {
        val domainFailures = Layer("Domain failures", "..domain.failure..")
        val domainModel = Layer("Domain model", "..domain.model..")
        val domainApi = Layer("Domain API", "..domain.api..")
        val domainSpi = Layer("Domain SPI", "..domain.spi..")

        val configuration = Layer("Configuration", "..infra.config..")
        configuration.dependsOn(domainApi)

        val kafka = Layer("Kafka", "..infra.kafka..")
        kafka.dependsOn(domainApi)

        val web = Layer("Web", "..infra.web..")
        web.dependsOn(domainApi)

        val jpa = Layer("JPA", "..infra.jpa..")
        jpa.dependsOn(domainSpi)

        val merchantDirectory = Layer("Merchant Directory", "..infra.md..")
        merchantDirectory.dependsOn(domainSpi)
    }
```



Domain model

Isolated, immutable & validated



Domain model

Focused on business meaning

```
data class CategorizedTransaction(
    val id: CategorizedTransactionId,
    val transaction: Transaction,
    val expenseCategory: ExpenseCategory
)
```

Response model

API-friendly shape

```
@Serializable data class CategorizedTransactionResponse(
    @Contextual val transactionId: TransactionId,
    @Contextual val expenseCategory: ExpenseCategory
)
```

ORM Entity

Database-focus shape

```
@Table(name = "categorized_transaction")
@Entity
data class CategorizedTransactionEntity(
    @Id @GeneratedValue(strategy = GenerationType.UUID) val id: UUID,
    @Column(name = "transaction_id", nullable = false) val transactionId: UUID?,
    @Column(name = "client_id", nullable = false) val clientId: UUID?,
    @Column(name = "account_id", nullable = false) val accountId: UUID?,
    @Column(name = "amount", nullable = false) val amount: BigDecimal?,
    @Column(name = "currencyCode", nullable = false) val currencyCode: String?,
    @Column(name = "mcc", nullable = false) val mcc: String?,
    @Column(name = "expense_category", nullable = false) val expenseCategory: String?
)
```

Primitive Obsession

Primitive Obsession: using primitive data types to represent domain concepts, resulting in unclear intent and scattered validation logic.

— Martin Fowler

What Primitive Obsession looks like


```
data class Transaction(
    val id: String,
    val clientId: String,
    val accountId: String,
    val amount: BigDecimal,
    val currencyCode: String,
    val mcc: String
)
```

Beyond strings and numbers

Enhancing domain models with domain-specific types

```
data class Transaction(
    val id: String,
    val clientId: String,
    val accountId: String,
    val amount: BigDecimal,
    val currencyCode: String,
    val mcc: String
)
```

```
Transaction(
    “629847fd-…”,
    “2e228dde-…”,
    “aa3a1fbe-…”,
    BigDecimal("250.75"),
    "EUR",
    "5411"
)
```

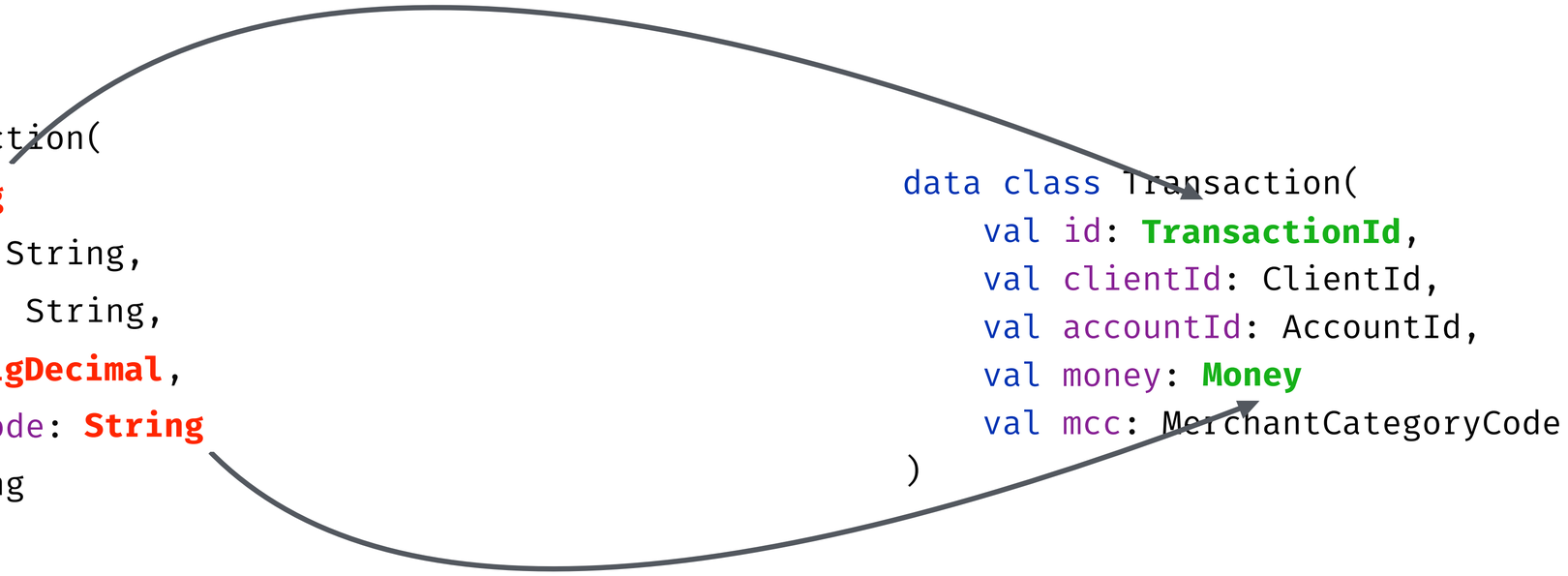
Beyond strings and numbers

Domain-specific types

```
@JvmInline value class TransactionId(val value: UUID)
```

```
data class Transaction(
    val id: String
    val clientId: String,
    val accountId: String,
    val amount: BigDecimal,
    val currencyCode: String
    val mcc: String
)
```

```
data class Transaction(
    val id: TransactionId,
    val clientId: ClientId,
    val accountId: AccountId,
    val money: Money
    val mcc: MerchantCategoryCode
)
```



```
data class Money(
  val value: BigDecimal,
  val currency: Currency
)
```

Beyond strings and numbers

Encapsulated validation

```
@JvmInline value class MerchantCategoryCode(val value: String)
```

Beyond strings and numbers

Encapsulated validation

```
@JvmInline value class MerchantCategoryCode(val value: String) {
    companion object {
        private val MCC_PATTERN = Regex("^\\d{4}$")
    }

    init {
        require(MCC_PATTERN.matches(value)) {
           "Invalid merchant category code: '$value'"
        }
    }
}
```

Beyond strings and numbers

Focused unit tests

```
"should be invalid when non-numeric code is provided" {
    val e = shouldThrow<IllegalArgumentException> {
        MerchantCategoryCode("12A4")
    }

    e.message shouldBe "Invalid merchant category code: '12A4'"
}
```

```
@JvmInline value class MerchantCategoryCode(val value: String) {
    companion object {
        private val MCC_PATTERN = Regex("^\\d{4}$")
    }

    init {
        require(MCC_PATTERN.matches(value)) {
           "Invalid merchant category code: '$value'"
        }
    }
}
```

Throw Exceptions Out of Your Domain

Throw exceptions out of your domain

```
@JvmInline value class MerchantCategoryCode(val value: String) {
    companion object {
        private val MCC_PATTERN = Regex("^\\d{4}$")
    }

    init {
        require(MCC_PATTERN.matches(value)) {
           "Invalid merchant category code: '$value'"
        }
    }
}
```

First-class errors

The domain tells you what can go wrong

```
error object NullMerchantCategoryCode

error class InvalidMerchantCategoryPattern(val value: String)

typealias MccValidationFailed = NullMerchantCategoryCode | InvalidMerchantCategoryPattern
```

First-class errors

The domain tells you what can go wrong

```
error object NullMerchantCategoryCode

error class InvalidMerchantCategoryPattern(val value: String)

typealias MccValidationFailed = NullMerchantCategoryCode | InvalidMerchantCategoryPattern


@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        private val MCC_PATTERN = Regex("^\\d{4}$")

        operator fun invoke(value: String?): MerchantCategoryCode | MccValidationFailed = when {
            value == null -> NullMerchantCategoryCode
            !MCC_PATTERN.matches(value) -> InvalidMerchantCategoryPattern(value)
            else -> MerchantCategoryCode(value)
        }
    }
}
```

Rich errors

```
"should be invalid when non-numeric code is provided" {
    val mcc: MerchantCategoryCode | MccValidationFailed = MerchantCa                    4")

    when(mcc) {
        is MerchantCategoryCode → fail("should not get here")
        is NullMerchantCategoryCode → fail("should not get
        is InvalidMerchantCategoryPattern → mcc shouldBe Inv                          2A4")
    }
}

@JvmInline value class MerchantCategoryCode private constructor(
    companion object {
        private val MCC_PATTERN = Regex("^\\d{4}$")

        operator fun invoke(value: String?): MerchantCategoryCode | MccValidationFailed = when {
            value == null → NullMerchantCategoryCode
            !MCC_PATTERN.matches(value) → InvalidMerchantCategoryPattern(value)
            else → MerchantCategoryCode(value)
        }
    }
}
```

Modeling failures as domain citizens

```
sealed interface Failure {
    val message: String
}

sealed interface ValidationFailed : Failure

data object NullMerchantCategoryCode : ValidationFailed {
    override val message = "Merchant category code cannot be null"
}

data class InvalidMerchantCategoryPattern(val mcc: String) : ValidationFailed {
    override val message = "Merchant category code must be exactly 4 digits. Actual value: '$mcc'"
}
```

Handling failure with Either

Rich errors

```
MerchantCategoryCode | MccValidationFailed
```

```
val result = when {
  value == null → NullMerchantCategoryCode
  !pattern.matches(value) → InvalidM..C..Pattern(value)
  else → MerchantCategoryCode(value)
}
```

```
when(result) {
  is MerchantCategoryCode → …
  is NullMerchantCategoryCode → …
  is InvalidMerchantCategoryPattern → …
}
```

Either (ARROW)

```
Either<MccValidationFailed, MerchantCategoryCode>
```

```
val result = when {
  value == null → Left(NullMerchantCategoryCode)
  !patern.matches(value) → Left(InvalidM..C..Pattern(value))
  else → Right(MerchantCategoryCode(value))
}
```

```
result.fold(
when(result) {
    is Right → …
    is Left → when(result.value) {
        is NullMerchantCategoryCode → ..
        is InvalidMerchantCategoryPattern → ..
    }
}
)
```

Handling failure with Either

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        operator fun invoke(value: String?): Either<ValidationFailed, MerchantCategoryCode> = when {
            value == null → Left(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → Left(InvalidMerchantCategoryPattern(value))
            else → Right(MerchantCategoryCode(value))
        }
    }
}
```

Handling failure with Raise DSL

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        operator fun invoke(value: String?): Either<ValidationFailed, MerchantCategoryCode> = when {
            value == null → Left(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → Left(InvalidMerchantCategoryPattern(value))
            else → Right(MerchantCategoryCode(value))
        }
    }
}
```

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        context(_: Raise<ValidationFailed>)
        operator fun invoke(value: String?): MerchantCategoryCode = when {
            value == null → raise(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → raise(InvalidMerchantCategoryPattern(value))
            else → MerchantCategoryCode(value)
        }
    }
}
```

Handling failure with Raise DSL

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        operator fun invoke(value: String?): Either<ValidationFailed, MerchantCategoryCode> = when {
            value == null → Left(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → Left(InvalidMerchantCategoryPattern(value))
            else → Right(MerchantCategoryCode(value))
        }
    }
}
```

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        context(_: Raise<ValidationFailed>)
        operator fun invoke(value: String?): MerchantCategoryCode = when {
            value == null → raise(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → raise(InvalidMerchantCategoryPattern(value))
            else → MerchantCategoryCode(value)
        }
    }
}
```

Handling failure with Raise DSL

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        operator fun invoke(value: String?): Either<ValidationFailed, MerchantCategoryCode> = when {
            value == null → Left(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → Left(InvalidMerchantCategoryPattern(value))
            else → Right(MerchantCategoryCode(value))
        }
    }
}
```

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        context(_: Raise<ValidationFailed>)
        operator fun invoke(value: String?): MerchantCategoryCode = when {
            value == null → raise(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → raise(InvalidMerchantCategoryPattern(value))
            else → MerchantCategoryCode(value)
        }
    }
}
```

Handling failure with Raise DSL

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        operator fun invoke(value: String?): Either<ValidationFailed, MerchantCategoryCode> = when {
            value == null → Left(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → Left(InvalidMerchantCategoryPattern(value))
            else → Right(MerchantCategoryCode(value))
        }
    }
}
```

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        context(_: Raise<ValidationFailed>)
        operator fun invoke(value: String?): MerchantCategoryCode = when {
            value == null → raise(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → raise(InvalidMerchantCategoryPattern(value))
            else → MerchantCategoryCode(value)
        }
    }
}
```

Handling failure with Raise DSL

```
"should create a valid MerchantCategoryCode for a valid 4-digit code" {
    val mcc: Either<ValidationFailed, MerchantCategoryCode> = either {
        MerchantCategoryCode("1234")
    }

    mcc shouldBeRight { "1234" }
}
```

```
@JvmInline value class MerchantCategoryCode private constructor(val value: String) {
    companion object {
        context(_: Raise<ValidationFailed>)
        operator fun invoke(value: String?): MerchantCategoryCode = when {
            value == null → raise(NullMerchantCategoryCode)
            !MCC_PATTERN.matches(value) → raise(InvalidMerchantCategoryPattern(value))
            else → MerchantCategoryCode(value)
        }
    }
}
```

Error accumulation

```
context(_: Raise<CategorizedTransactionCorrupted>)
fun CategorizedTransactionEntity.toDomain(): CategorizedTransaction =
    withError(::CategorizedTransactionCorrupted) {
        accumulate {
            val validTransactionId by accumulating { TransactionId(transactionId) }
            val validClientId by accumulating { ClientId(clientId) }
            val validAccountId by accumulating { AccountId(accountId) }
            val validMoney by accumulating { Money(amount, currencyCode) }
            val validMcc by accumulating { MerchantCategoryCode(mcc) }
            val validExpenseCategory by accumulating { ExpenseCategory(expenseCategory) }

            CategorizedTransaction(
                id = CategorizedTransactionId(id),
                transaction = Transaction(
                    id = validTransactionId,
                    clientId = validClientId,
                    accountId = validAccountId,
                    money = validMoney,
                    mcc = validMcc
                ),
                expenseCategory = validExpenseCategory
            )
        }
    }
```

Error accumulation

```
context(_: Raise<CategorizedTransactionCorrupted>)
fun CategorizedTransactionEntity.toDomain(): CategorizedTransaction =
    withError(::CategorizedTransactionCorrupted) {
        accumulate {
            val validTransactionId by accumulating { TransactionId(transactionId) }
            val validClientId by accumulating { ClientId(clientId) }
            val validAccountId by accumulating { AccountId(accountId) }
            val validMoney by accumulating { Money(amount, currencyCode) }
            val validMcc by accumulating { MerchantCategoryCode(mcc) }
            val validExpenseCategory by accumulating { ExpenseCategory(expenseCategory) }

            CategorizedTransaction(
                id = CategorizedTransactionId(id),
                transaction = Transaction(
                    id = validTransactionId,
                    clientId = validClientId,
                    accountId = validAccountId,
                    money = validMoney,
                    mcc = validMcc
                ),
                expenseCategory = validExpenseCategory
            )
        }
    }
```

NonEmptyList<ValidationFailure>

Error accumulation

```
context(_: Raise<CategorizedTransactionCorrupted>)
fun CategorizedTransactionEntity.toDomain(): CategorizedTransaction =
    withError(::CategorizedTransactionCorrupted) {
        accumulate {
            val validTransactionId by accumulating { TransactionId(transactionId) }
            val validClientId by accumulating { ClientId(clientId) }
            val validAccountId by accumulating { AccountId(accountId) }
            val validMoney by accumulating { Money(amount, currencyCode) }
            val validMcc by accumulating { MerchantCategoryCode(mcc) }
            val validExpenseCategory by accumulating { ExpenseCategory(expenseCategory) }

            CategorizedTransaction(
                id = CategorizedTransactionId(id),
                transaction = Transaction(
                    id = validTransactionId,
                    clientId = validClientId,
                    accountId = validAccountId,
                    money = validMoney,
                    mcc = validMcc
                ),
                expenseCategory = validExpenseCategory
            )
        }
    }
```

Raise<**ValidationFailure**>

1

NonEmptyList<**ValidationFailure**>

Error accumulation

```
context(_: Raise<CategorizedTransactionCorrupted>)
fun CategorizedTransactionEntity.toDomain(): CategorizedTransaction =
    withError(::CategorizedTransactionCorrupted) {
        accumulate {
            val validTransactionId by accumulating { TransactionId(transactionId) }
            val validClientId by accumulating { ClientId(clientId) }
            val validAccountId by accumulating { AccountId(accountId) }
            val validMoney by accumulating { Money(amount, currencyCode) }
            val validMcc by accumulating { MerchantCategoryCode(mcc) }
            val validExpenseCategory by accumulating { ExpenseCategory(expenseCategory) }

            CategorizedTransaction(
                id = CategorizedTransactionId(id),
                transaction = Transaction(
                    id = validTransactionId,
                    clientId = validClientId,
                    accountId = validAccountId,
                    money = validMoney,
                    mcc = validMcc
                ),
                expenseCategory = validExpenseCategory
            )
        }
    }
```

Raise<**ValidationFailure**>

Raise<**ValidationFailure**>

Raise<**ValidationFailure**>

Raise<**ValidationFailure**>

Raise<**ValidationFailure**>

NonEmptyList<**ValidationFailure**>

1 2 3 4 5 6

Error accumulation

```
context(_: Raise<CategorizedTransactionCorrupted>)
fun CategorizedTransactionEntity.toDomain(): CategorizedTransaction =
    withError(::CategorizedTransactionCorrupted) {
        accumulate {
            val validTransactionId by accumulating { TransactionId(transactionId) }
            val validClientId by accumulating { ClientId(clientId) }
            val validAccountId by accumulating { AccountId(accountId) }
            val validMoney by accumulating { Money(amount, currencyCode) }
            val validMcc by accumulating { MerchantCategoryCode(mcc) }
            val validExpenseCategory by accumulating { ExpenseCategory(expenseCategory) }

            CategorizedTransaction(
                id = CategorizedTransactionId(id),
                transaction = Transaction(
                    id = validTransactionId,
                    clientId = validClientId,
                    accountId = validAccountId,
                    money = validMoney,
                    mcc = validMcc
                ),
                expenseCategory = validExpenseCategory
            )
        }
    }
```

NonEmptyList<ValidationFailure>

1 2 3 4 5 6

Error accumulation

```
context(_: Raise<CategorizedTransactionCorrupted>)
fun CategorizedTransactionEntity.toDomain(): CategorizedTransaction =
    withError(::CategorizedTransactionCorrupted) {

data class CategorizedTransactionCorrupted(
    val innerErrors: ValidationFailures,
    override val message: String = "…"
)

                                                    { TransactionId(transactionId) }
                                                    lientId(clientId) }
                                                    AccountId(accountId) }
                                                    (amount, currencyCode) }
        val validMcc by accumulating { MerchantCategoryCodeCode(mcc) }
        val validExpenseCategory by accumulating { ExpenseCategory(expenseCategory) }

        CategorizedTransaction(
            id = CategorizedTransactionId(id),
            transaction = Transaction(
                id = validTransactionId,
                clientId = validClientId,
                accountId = validAccountId,
                money = validMoney,
                mcc = validMcc
            ),
            expenseCategory = validExpenseCategory
        )
    }
}
```

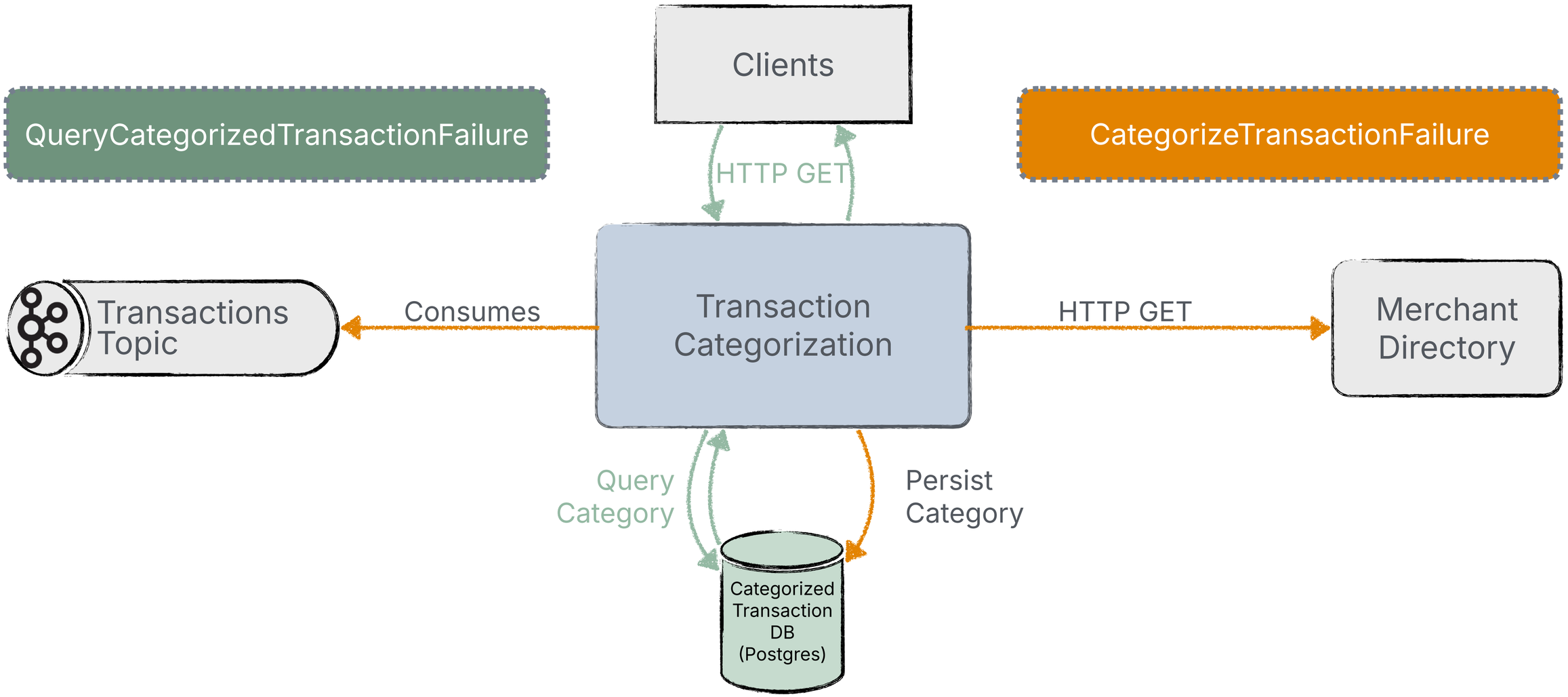
NonEmptyList<ValidationFailure>

1 2 3 4 5 6

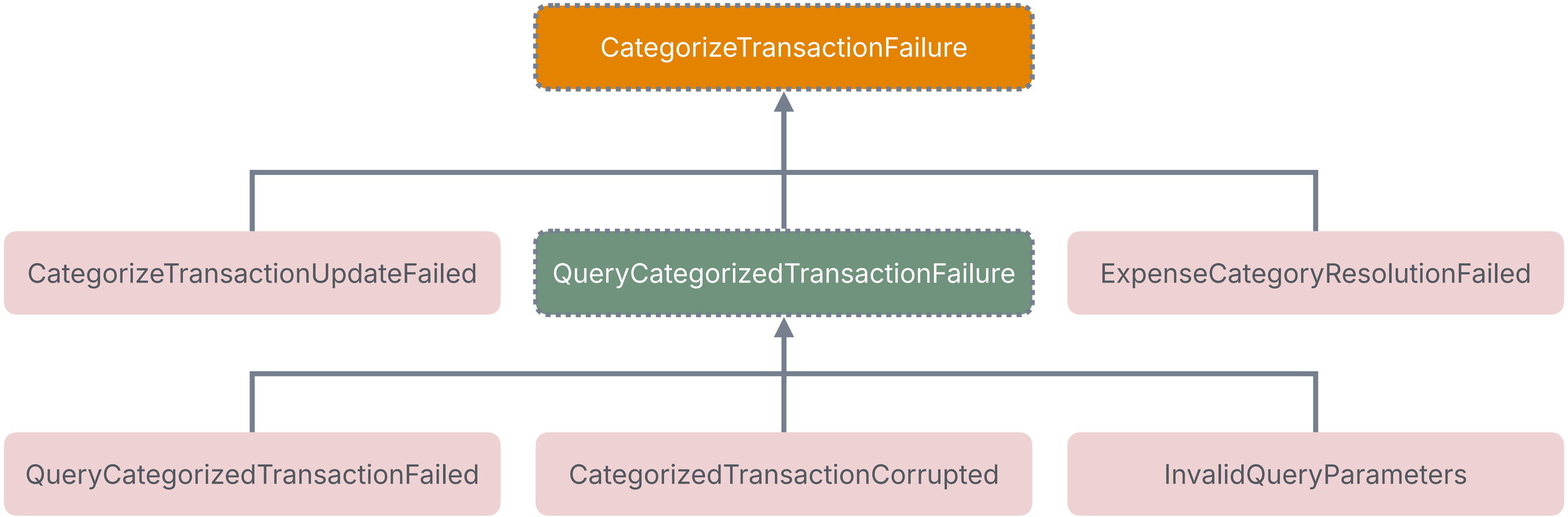
Error accumulation

```
"should accumulate data corruption failures when converting to domain model" {
    val entity = CategorizedTransactionEntity(…)

    val result = either {
        entity.toDomain()
    }

    result shouldBeLeft CategorizedTransactionCorrupted(
        Nel.of(
            NullOrEmpty(property = "clientId", value = ""),
            NullOrEmpty(property = "accountId", value = null),
            InvalidMerchantCategoryPattern("123"),
            NullOrEmpty(property = "expenseCategory", value = "")
        )
    )
}
```

Composing failures



Composing failures



Interface Segregation

Interface Segregation: Treat interfaces as functions. Make interfaces small, focused and single-purpose — ideally, a single abstract method.

Just a simple repository

Monolithic interface

```
@Repository
interface CategorizedTransactionJpaRepository : JpaRepository<CategorizedTransactionEntity, UUID> {

    fun findByTransactionId(transactionId: UUID): CategorizedTransactionEntity?

    fun findByClientIdAndExpenseCategory(
        clientId: ClientId,
        expenseCategory: ExpenseCategory
    ): List<CategorizedTransactionEntity>

    @Query(
        """
            SELECT new fvf4k.demo.infra.jpa.CategoryBudgetResult(t.expenseCategory, SUM(t.amount), t.currencyCode)
            FROM CategorizedTransactionEntity t
            WHERE t.clientId = :clientId
            GROUP BY t.expenseCategory
            """
    )
    fun findCategoryBudgetsByClientId(@Param("clientId") clientId: String): List<CategoryBudgetResult>
}
```

Just a si

Dozens of

```
class TransactionC                                                    ctionJpaRepository) {

    fun categorize
        repository.
    }
}
```

```
findCategoryBudgetsByClientIdList<CategoryBudgetResult>
findByTransactionId(trans CategorizedTransactionEntity?
findByClientIdAndExpenseCategoryList<CategorizedTransa…
findBy(example: Example<S!>, queryFunction: Fur R & Any
save(entity: S & Any)                                  S & Any
leftEither<CategorizedTransactionJpaRepository, Nothin…
bind() for A? in ar CategorizedTransactionJpaRepository
count()                                                Long
count(example: Example<S!>)                            Long
delete(entity: CategorizedTransactionEntity)           Unit
deleteAll()                                            Unit
deleteAll(entities: kotlin.collections.(Mutable)It     Unit
deleteAllById(ids: kotlin.collections.(Mutable)Ite     Unit
deleteAllByIdInBatch(ids: kotlin.collections.(Muta     Unit
deleteAllInBatch()                                     Unit
deleteAllInBatch(entities: kotlin.collections.(Mut     Unit
deleteById(id: UUID)                                   Unit
equals(other: Any?)                                    Boolean
exists(example: Example<S!>)                           Boolean
existsById(id: UUID)                                   Boolean
findAllkotlin.collections.(Mutable)List<CategorizedTra…
findAllkotlin.collections.(Mutable)List<CategorizedTra…
findAll(pageable: P Page<CategorizedTransactionEntity!>
findAll(example: E kotlin.collections.(Mutable)List<S!>
findAll(example: E kotlin.collections.(Mutable)List<S!>
findAll(example: Example<S!>, pageable: Pageab Page<S!>
findAllByIdkotlin.collections.(Mutable)List<Categorize…
findById(id: UL Optional<CategorizedTransactionEntity!>
findOne(example: Example<S!>)                  Optional<S!>
flush()                                                Unit
```

Press ^. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

Just a simple repository

Mocking complexity

```
val transactionCategorizer = TransactionCategorizerService(mockRepository, mockMerchantDirectoryService)

every { mockMerchantDirectoryService.getMerchantCategoryCode(message.mcc) }
    .answers { MerchantInfo(message.mcc, "Transportation") }
every { mockRepository.findByTransactionId(UUID.fromString(message.transactionId)) }
    .answers { null }
every { mockRepository.save(any()) }
    .answers {
        val ct = it.invocation.args[0] as CategorizedTransactionEntity
        ct.copy(id = categorizedTransactionId)
    }
```

Splitting for clarity

```
@Repository                                    <S extends CategorizedTransactionEntity> @NonNull S save(@NonNull S entity)
interface CategorizedTransactionJpaRepository : JpaRepository<CategorizedTransactionEntity, UUID> {
    fun findByTransactionId(transactionId: UUID): CategorizedTransactionEntity?
    fun findByClientIdAndExpenseCategory(
        clientId: ClientId,
        expenseCategory: ExpenseCategory
    ): List<CategorizedTransactionEntity>
    fun findCategoryBudgetsByClientId(@Param("clientId") clientId: String): List<CategoryBudgetResult>
}
```

Infrastructure

Domain

```
fun interface SaveCategorizedTransaction {
    context(_: Raise<SaveCategorizedTransactionFailure>)
    operator fun invoke(transaction: CategorizedTransaction): CategorizedTransaction
}
```

Splitting for clarity

```
@Repository
interface CategorizedTransactionJpaRepository : JpaRepository<CategorizedTransactionEntity, UUID> {
    fun findByTransactionId(transactionId: UUID): CategorizedTransactionEntity?
    fun findByClientIdAndExpenseCategory(
        clientId: ClientId,
        expenseCategory: ExpenseCategory
    ): List<CategorizedTransactionEntity>
    fun findCategoryBudgetsByClientId(@Param("clientId") clientId: String): List<CategoryBudgetResult>
}
```

Infrastructure

Domain

```
fun interface FindByTransactionId {
    context(_: Raise<QueryCategorizedTransactionFailure>)
    operator fun invoke(transactionId: TransactionId): CategorizedTransaction?
}
```

Splitting for clarity

```
@Repository
interface CategorizedTransactionJpaRepository : JpaRepository<CategorizedTransactionEntity, UUID> {
    fun findByTransactionId(transactionId: UUID): CategorizedTransactionEntity?
    fun findByClientIdAndExpenseCategory(
        clientId: ClientId,
        expenseCategory: ExpenseCategory
    ): List<CategorizedTransactionEntity>
    fun findCategoryBudgetsByClientId(@Param("clientId") clientId: String): List<CategoryBudgetResult>
}
```

Infrastructure

Domain

```
fun interface FindByTransactionId {
    context(_: Raise<QueryCategorizedTransactionFailure>)
    operator fun invoke(transactionId: TransactionId): CategorizedTransaction?
}
```

Splitting for clarity

```
@Repository
interface CategorizedTransactionJpaRepository : JpaRepository<CategorizedTransact
    fun findByTransactionId(transactionId: UUID): CategorizedTransactionEntity?
    fun findByClientIdAndExpenseCategory(
        clientId: ClientId,
        expenseCategory: ExpenseCategory
    ): List<CategorizedTransactionEntity>
    fun findCategoryBudgetsByClientId(@Param("clientId") clientId: String): List<CategoryBudgetResult>
}
```

throws Exceptions

Infrastructure

Domain

```
fun interface FindByTransactionId {
    context(_: Raise<QueryCategorizedTransactionFailure>)
    operator fun invoke(transactionId: TransactionId): CategorizedTransaction?
}
```

One adapter, multiple ports

```
class CategorizedTransactionReadRepositoryAdapter(
    val jpaRepository: CategorizedTransactionJpaRepository
) : FindByTransactionId,
    FindByClientIdAndExpenseCategory,
    FindBudgetsByCategory
```

One adapter, multiple ports

```
class CategorizedTransactionReadRepositoryAdapter(
    val jpaRepository: CategorizedTransactionJpaRepository
) : FindByTransactionId,
    FindByClientIdAndExpenseCategory,
    FindBudgetsByCategory {

    context(_: Raise<QueryCategorizedTransactionFailure>)
    override fun invoke(transactionId: TransactionId): CategorizedTransaction? =
        catch({
            jpaRepository.findByTransactionId(transactionId.value)?.toDomain()
        }) { exception ->
            raise(
                QueryCategorizedTransactionFailed(
                    "could not execute query by transaction id. " +
                            "id='$transactionId' cause: ${exception.message}"
                )
            )
        }
}
```

Pick only what you need

Services depend on ports, not frameworks

```
class TransactionCategorizerService(
    private val findByTransactionId: FindByTransactionId,
    private val saveTransaction: SaveCategorizedTransaction,
    private val resolveExpenseCategory: ResolveExpenseCategory
) : CategorizeTransaction
```

Domain

Infrastructure

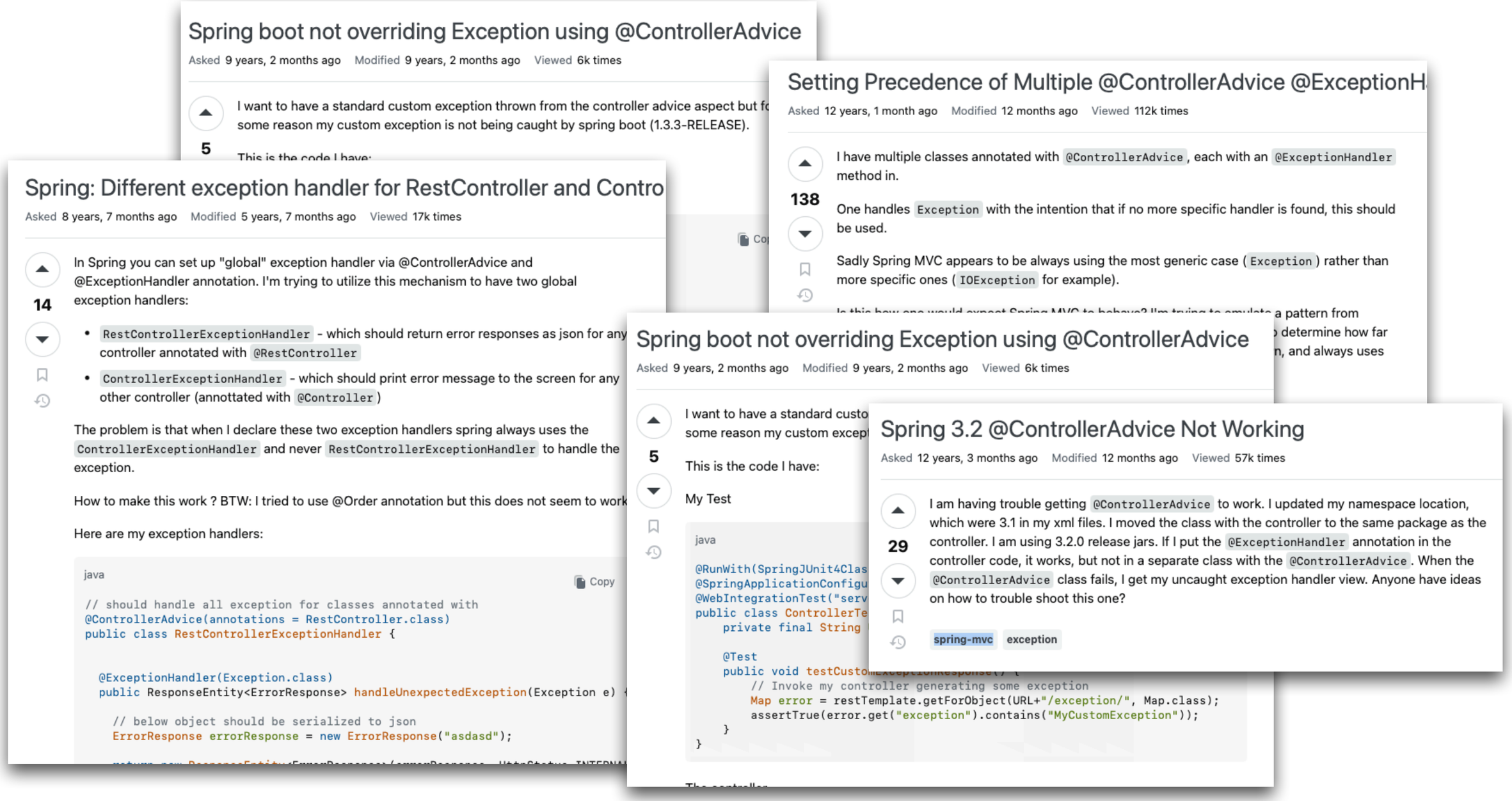
```
@Configuration
internal class TransactionCategorizationConfiguration {

    @Bean
    fun transactionCategorizerService(
        findByTransactionId: FindByTransactionId,
        saveTransaction: SaveCategorizedTransaction,
        resolveExpenseCategory: ResolveExpenseCategory,
    ): CategorizeTransaction =
        TransactionCategorizerService(findByTransactionId, saveTransaction, resolveExpenseCategory)
}
```

Lambdas instead of mocks

```
val transactionCategorizer = TransactionCategorizerService(mockRepository, mockMerchantDirectoryService)

every { mockMerchantDirectoryService.getMerchantCategoryCode(message.mcc) }
    .answers { MerchantInfo(message.mcc, "Transportation") }
every { mockRepository.findByTransactionId(UUID.fromString(message.transactionId)) }
    .answers { null }
every { mockRepository.save(any()) }
    .answers {
        val ct = it.invocation.args[0] as CategorizedTransactionEntity
        ct.copy(id = categorizedTransactionId)
    }
```

Lambdas instead of mocks

```
val transactionCategorizer = TransactionCategorizerService(
    findByTransactionId = {categorizedTransaction.copy(id = testId)},
    saveTransaction = { it },
    resolveExpenseCategory = { ExpenseCategory("Transportation") }
)
```



Spring boot not overriding Exception using @ControllerAdvice

Asked 9 years, 2 months ago Modified 9 years, 2 months ago Viewed 6k times

5

I want to have a standard custom exception thrown from the controller advice aspect but fo
some reason my custom exception is not being caught by spring boot (1.3.3-RELEASE).

This is the code I have:

Setting Precedence of Multiple @ControllerAdvice @ExceptionH

Asked 12 years, 1 month ago Modified 12 months ago Viewed 112k times

138

I have multiple classes annotated with `@ControllerAdvice`, each with an `@ExceptionHandler` method in.

One handles `Exception` with the intention that if no more specific handler is found, this should be used.

Sadly Spring MVC appears to be always using the most generic case (`Exception`) rather than more specific ones (`IOException` for example).

Is this how one would expect Spring MVC to behave? I'm trying to emulate a pattern from
determine how far
n, and always uses

Spring: Different exception handler for RestController and Contro

Asked 8 years, 7 months ago Modified 5 years, 7 months ago Viewed 17k times

14

In Spring you can set up "global" exception handler via @ControllerAdvice and @ExceptionHandler annotation. I'm trying to utilize this mechanism to have two global exception handlers:

- `RestControllerExceptionHandler` - which should return error responses as json for any controller annotated with `@RestController`
- `ControllerExceptionHandler` - which should print error message to the screen for any other controller (annottated with `@Controller`)

The problem is that when I declare these two exception handlers spring always uses the `ControllerExceptionHandler` and never `RestControllerExceptionHandler` to handle the exception.

How to make this work ? BTW: I tried to use @Order annotation but this does not seem to work

Here are my exception handlers:

java Copy

```
// should handle all exception for classes annotated with
@ControllerAdvice(annotations = RestController.class)
public class RestControllerExceptionHandler {


  @ExceptionHandler(Exception.class)
  public ResponseEntity<ErrorResponse> handleUnexpectedException(Exception e) {

    // below object should be serialized to json
    ErrorResponse errorResponse = new ErrorResponse("asdasd");
```

Spring boot not overriding Exception using @ControllerAdvice

Asked 9 years, 2 months ago Modified 9 years, 2 months ago Viewed 6k times

5

I want to have a standard custo
some reason my custom except

This is the code I have:

My Test

java

```
@RunWith(SpringJUnit4Clas
@SpringApplicationConfigu
@WebIntegrationTest("serv
public class ControllerTe
    private final String

    @Test
    public void testCusto
        // Invoke my controller generating some exception
        Map error = restTemplate.getForObject(URL+"/exception/", Map.class);
        assertTrue(error.get("exception").contains("MyCustomException"));
    }
}
```

The controller

Spring 3.2 @ControllerAdvice Not Working

Asked 12 years, 3 months ago Modified 12 months ago Viewed 57k times

29

I am having trouble getting `@ControllerAdvice` to work. I updated my namespace location, which were 3.1 in my xml files. I moved the class with the controller to the same package as the controller. I am using 3.2.0 release jars. If I put the `@ExceptionHandler` annotation in the controller code, it works, but not in a separate class with the `@ControllerAdvice`. When the `@ControllerAdvice` class fails, I get my uncaught exception handler view. Anyone have ideas on how to trouble shoot this one?

spring-mvc exception

Failures at the edge

```
@RestController
class CategorizedTransactionController(val queryService: TransactionQueryService)

    @GetMapping("/client/{clientId}/categories-budget")
    fun getClientExpensesGroupedByCategory(
        @PathVariable clientId: String?
    ): ResponseEntity<*> = either {
        val validClientId = withError( ::InvalidQueryParameter) {
            ClientId(clientId)
        }

        queryService.queryBudgetByCategory(validClientId)
    }
```

Raise<ValidationFailed>

Raise<QueryCategorizedTransactionFailure>

Either<QueryCategorizedTransactionFailure, List<BudgetCategory>>

Failures at the edge

```
@GetMapping("/client/{clientId}/categories-budget")
fun getClientExpensesGroupedByCategory(
    @PathVariable clientId: String?
): ResponseEntity<*> = either {
    val validClientId = withError(::InvalidQueryParameter) {
        ClientId(clientId)
    }

    queryService.queryBudgetByCategory(validClientId)
}.fold(
        ifLeft = { error ->
            when (error) {
                is InvalidQueryParameters -> ResponseEntity.badRequest()
                    .body(FailureResponse(error.message, error.innerErrors.map { it.message }))
                is QueryCategorizedTransactionFailed -> ResponseEntity.status(HttpStatus.SERVICE_UNAVAILABLE)
                    .body(FailureResponse(error.message))
                is CategorizedTransactionCorrupted -> ResponseEntity.internalServerError()
                    .body(FailureResponse(error.message, error.innerErrors.map { it.message }))
            }
        },
        ifRight = { ct -> ResponseEntity.ok(ct) }
    )
}
```

Another example: OpenFeign

```
fun interface ResolveExpenseCategory {
    context(_: Raise<ExpenseCategoryResolutionFailed>)
    operator fun invoke(mcc: MerchantCategoryCode): ExpenseCategory
}
```

Domain

Infrastructure

```
@FeignClient(name = "merchant-directory", url = "\${merchant.directory.url}")
interface MerchantDirectoryService {

    @GetMapping("/merchant-directory/{merchantCategoryCode}")
    fun getMerchantCategoryCode(@PathVariable("merchantCategoryCode") merchantCategoryCode: String): MerchantInfo
}

class ExpenseCategoryResolverAdapter(val merchantDirectoryService: MerchantDirectoryService) : ResolveExpenseCategory {
    context(_: Raise<ExpenseCategoryResolutionFailed>)
    override fun invoke(mcc: MerchantCategoryCode): ExpenseCategory {
        val merchantInfo = catch(
            block = { merchantDirectoryService.getMerchantCategoryCode(mcc.value) },
            catch = { exception →
                raise(ExpenseCategoryResolutionFailed("could not retrieve info for mcc='${mcc.value}': ${exception.message}"))
            }
        )
        return ExpenseCategory.Companion(merchantInfo.category)
    }
}
```

Key Takeaways

Keep Your Domain SAFE

- **S**egregate Interfaces
  - Small, functional interfaces simplify testing and maintainability.
- **A**void Ambiguous Types
  - Use domain-specific types for clarity, validation, and type safety.
- **F**ail Explicitly
  - Represent errors explicitly with domain-friendly Result/Either types—no hidden exceptions.
- **E**ncapsulate the Domain
  - Decouple from infrastructure/frameworks and ensure your domain invariants are always protected

Thank You!

Slides, Code & Contact

Let’s continue the conversation.

tibtof/fun-vs-framework

in/tibtof