Getting Functional Programming into Domain Driven Design - for real

Michael Sperber & Benedikt Stemmildt

Created: 2022-11-16 Wed 08:56

Getting Functional Programming into Domain Driven Design - for real

Mike Sperber

Active Group GmbH

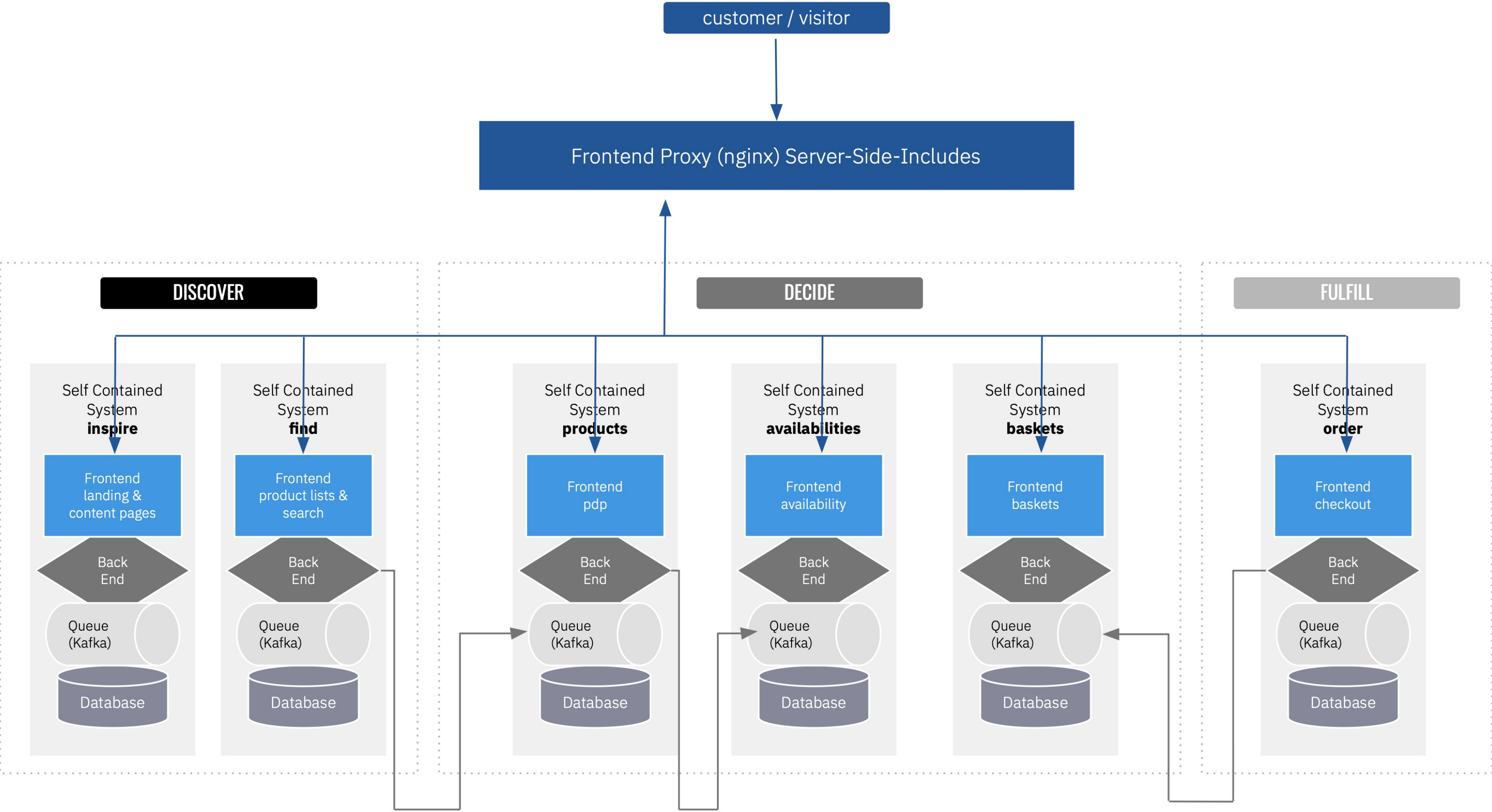
`@sperbsen`

Benedikt Stemmildt

TalentFormation

`@slashBene`

Self-Contained Systems @ Blume2000



Domain-Driven Design

- validation in outer layers
- business logic only in domain layer
- encode ubiquitous language in types

Functional Programming

- pervasive immutability / value objects
- pure functions
- everything is data
- higher-level abstractions
- combinator models

Functional Programming Languages

- generic function types
- functional/immutable data structures
- unrestricted abstraction

Functional Programming in Kotlin

- **generic function types**!
- ~~functional/immutable data structures~~
- ~~unrestricted abstraction~~
- ~~pervasive immutability / value objects~~
- pure functions
- everything is data
- higher-level abstractions
- (combinator models)

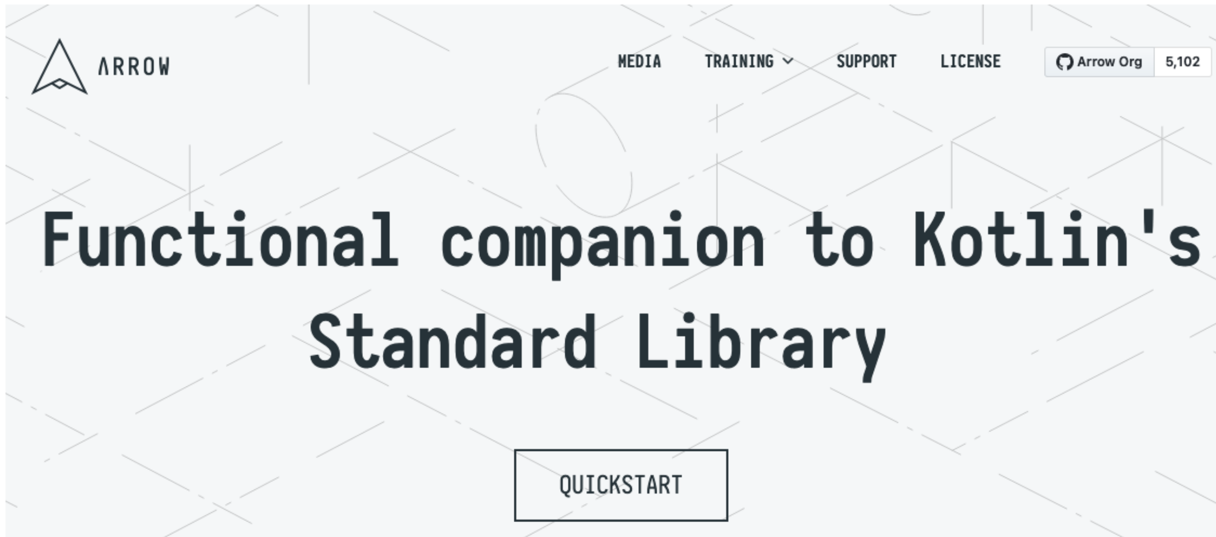
Domain-Driven Design

- **validation in outer layers**
- business logic only in domain layer
- encode ubiquitous language in types

Optional Values

```
sealed class Option<out A>
public object None : Option<Nothing>()
public data class Some<out T>(val value: T) : Option<T>()

fun safeDivide(x: Int, y: Int): Option<Int> =
  if (y == 0)
     None
  else
     Some(x/y)
```



Why Validation

Evans:

"A few days later, mysterious problems surfaced in the bill-payment application module [...] The mystery records had no value in the "percent deductible" field, although the validation of the data entry application required it and even put in a default value."

Validation vs. Null and Exceptions

```
public void check() {
    if (date == null) throw new IllegalArgumentException("date
    LocalDate parsedDate;
    try {
      parsedDate = LocalDate.parse(date);
    }
    catch (DateTimeParseException e) {
      throw new IllegalArgumentException("Invalid format for da
    }
    if (parsedDate.isBefore(LocalDate.now())) throw new IllegalAr
    if (numberOfSeats == null) throw new IllegalArgumentExcepti
    if (numberOfSeats < 1) throw new IllegalArgumentException("
}
```

Martin Fowler: Replacing Throwing Exceptions with Notification in Validations

Modern OO Validation

```
data class Position(
    @field:Min(1, message = "ANZAHL_ZERO") var anzahl: Int,
    @field:Valid var preis: Preis,
    @field:Valid var produkt: Produkt
)

... validator.validate(position) ...
```

Problems

- same type for valid and invalid position
- control flow unclear
- effectively weird DSL
- hard to replace
- dependencies

Unvalidated Data at the Door

Image source: James Marshall - Red Riding Hood

"Make illegal states unrepresentable."

Yaron Minsky

FP Validation

```
fun of(anzahl: Int, preis: Preis, product: Produkt)
    : Validated<List<ValidationErrorDescription>, Position>

sealed class Validated<out E, out A>
data class Invalid<out E>(val value: E) : Validated<E, Nothi
data class Valid<out A>(val value: A) : Validated<Nothing, A:

fun <A> A.valid(): Validated<Nothing, A> = Valid(this)
fun <E> E.invalid(): Validated<E, Nothing> = Invalid(this)
```

"Don't validate, parse"

```
fun of(anzahl: Int, preis: Preis, product: Produkt)
    : Validated<List<ValidationErrorDescription>, Position>
    if (anzahl > 1)
      Valid(Position(anzahl, preis, product))
    else
      Invalid(listOf(MinViolation(preis, 1)))
```

Validation Summary

- objects always validated
- validity expressed in type system
- standard Kotlin
- convenient through functional abstractions (map)
- composability
- no annotation DSL

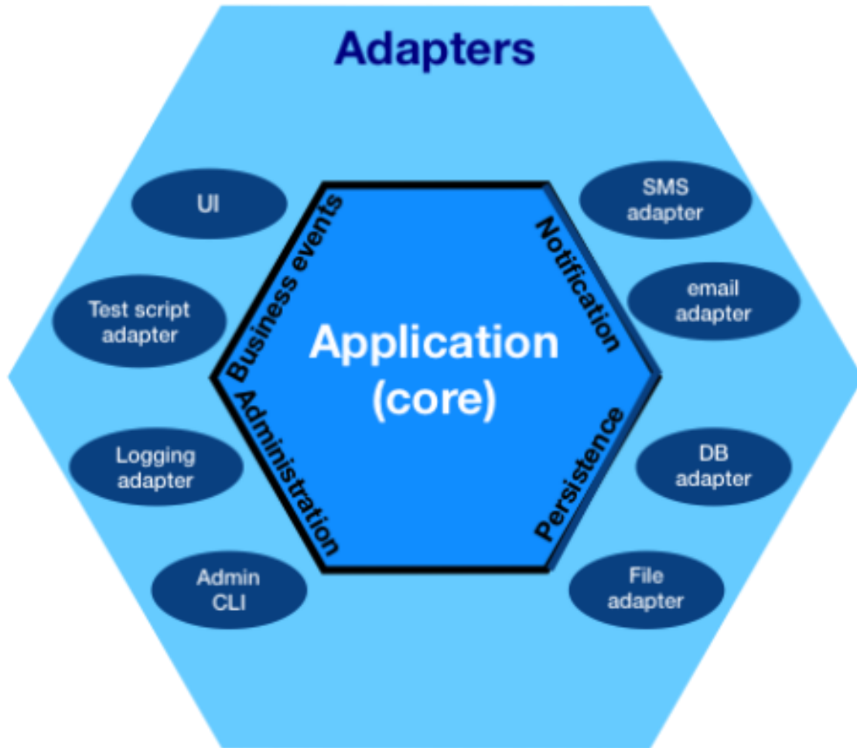
Domain-Driven Design

- validation in outer layers
- **business logic only in domain layer**
- **encode ubiquitous language in types**

Orchestrating Domain Functions into Workflows

```
val image1 = getProductImage(draft1)
val image2 = getProductImage(draft2)
val price1 = getProductPrice(draft1)
val price2 = getProductPrice(draft2)
publishProduct(Product.Published(draft1, image1, price1))
publishProduct(Product.Published(draft2, image2, price2))
...
```

Hexagonal Architecture



Source: Wikimedia Commons, CC-SA 4.0

Issues

- need domain language for workflows
- dependency injection
- cross-cutting concerns
- logging
- profiling
- concurrency
- ...

Profiling

```
fun updateWarenkorb(
  httpWarenkorbVeraenderung: HttpWarenkorbVeraenderung,
  sessionIdPayload: String,
): Warenkorb {
  val start = System.currentTimeMillis()

  val produkt = produktRepository.holeProduktViaProduktNummer(Pr
  ...
  val getProduktEnd = System.currentTimeMillis()
  logger.info { "get Produkt from DB took ${getProduktEnd - start

  if (produkt == null) { ... }
  ...
  val warenkorbVeraenderungTransformationEnd = System.currentT
  logger.info { "transform into warenkorb-veränderung took ${ware
  ...
  val warenkorbVeraenderungValidationEnd = System.currentTimeM
  logger.info { "warenkorb-veraenderung validation took ${warenko
  ...
  val getWarenkorbEnd = System.currentTimeMillis()
  logger.info { "get Warenkorb from DB took ${getWarenkorbEnd - w
```

```
        ...
}
```

DDD Workflows

- workflow description: value objects
- use types to express ubiquitous language
- need dependency injection

Representing Workflows as Value Objects

```
sealed interface PublishProductProcess<A> {
  data class GetProductImage<A>
     (val productDraft: Product.Draft,
      val nextStep: (ProductImage) -> PublishProductProcess<A
        : PublishProductProcess<A>

  data class GetProductPrice<A>
     (val productDraft: Product.Draft,
      val nextStep: (ProductPrice) -> PublishProductProcess<A
      : PublishProductProcess<A>

  data class PublishProduct<A>
     (val product: Product.Published,
      val nextStep: (Unit) -> PublishProductProcess<A>)
      : PublishProductProcess<A>

  data class Yield<A>(val result: A) : PublishProductProcess
}
```

Monadic Notation

```
fun <A> publishProductProcess
  (block: suspend PublishProductProcess.Step.() -> A)
     : PublishProductProcess<A> = ...

publishProductProcess {
  val image1 = getProductImage(draft1)
  val image2 = getProductImage(draft2)
  val price1 = getProductPrice(draft1)
  val price2 = getProductPrice(draft2)
  publishProduct(Product.Published(draft1, image1, price1))
  publishProduct(Product.Published(draft2, image2, price2))
  ...
}
```

Kotlin suspend = Monad

```
suspend fun computation(arg: T) {
    val res1 = computation1(arg, 1)
    val res2 = computation2(arg, 2)
    return res1 + res2
}

=>

fun computation(arg: T, cont: Continuation<...>) {
    computation1(arg, 1) { res ->
    computation2(arg, 2) { res ->
    cont.resume(res1 + res2)
    }}
}
```

Dependency Injection

```
override tailrec suspend fun <A>
  run(publishProductProcess: PublishProductProcess<A>)
    : Either<PublishProductError, A> =
  when (publishProductProcess) {
    is GetProductImage ->
      run(publishProductProcess.nextStep(...))
    is GetProductPrice ->
      run(publishProductProcess.nextStep(...))
    is PublishProduct ->
      if (saveProduct(publishProductProcess.product).isInvalid)
        PublishProductError.PublishFailed.left()
      else
        run(publishProductProcess.nextStep(Unit))
    is Yield -> publishProductProcess.result.right()
  }
```

Monadic Workflows in Kotlin

- abstract over control flow
- abstract over cross-cutting concerns
- separate domain logic from adapter
- "functional dependency injection": interpreter
- works with DI frameworks

DDD + Functional Programming = ❤️

- validation
- domain workflows
- dependency injection
- immutability
- combinators

Great about Kotlin

- data classes => low-overhead immutability
- function types
- monadic syntax via suspend

Missing from Kotlin

- standard immutable collections
- unrestricted abstraction over types
- composable overloading as in Scala, Haskell

Conclusion

- functional programming principles complement DDD
- Kotlin + functional programming = ❤️
- immutability, purity, types, monads
- monads FTW
- functional languages + functional programming = ❤️❤️❤️

Outlook

- didactics of monads
- blog-posts / articles / talks
- more convenient monads for Kotlin
- combinator libraries
- make Spring Boot object-oriented

https://gitlab.com/BeneStem/verticalization-example-service-one