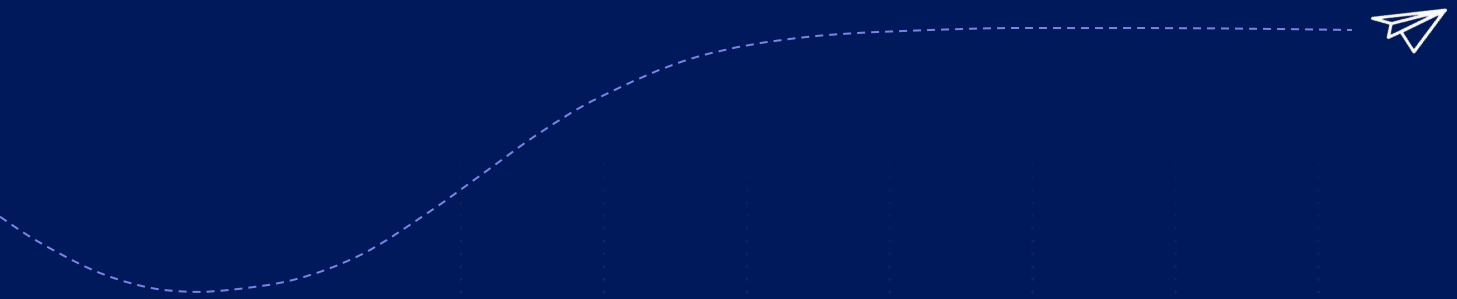
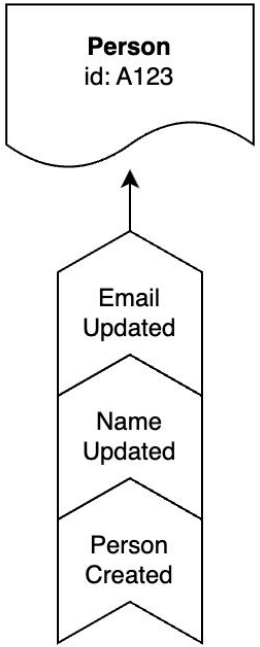


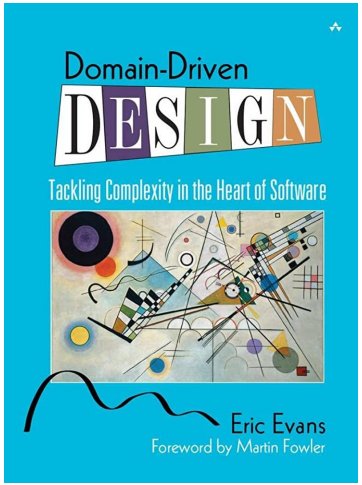
Anatomy of a Spring Boot App with Clean Architecture

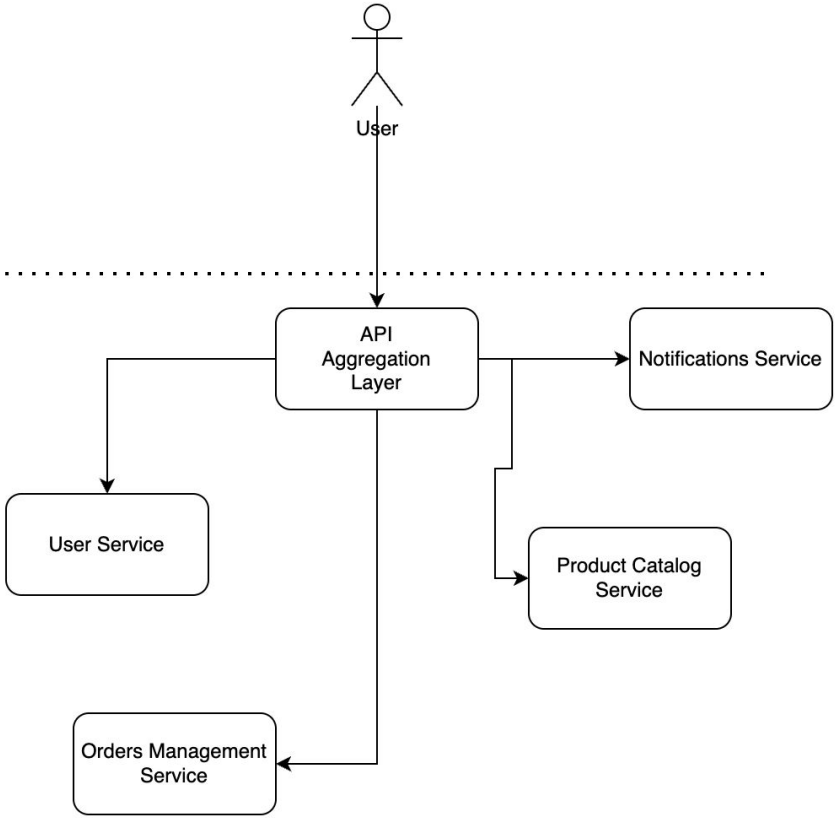
Steve Pember

Senior Staff @ Stavvy, Inc

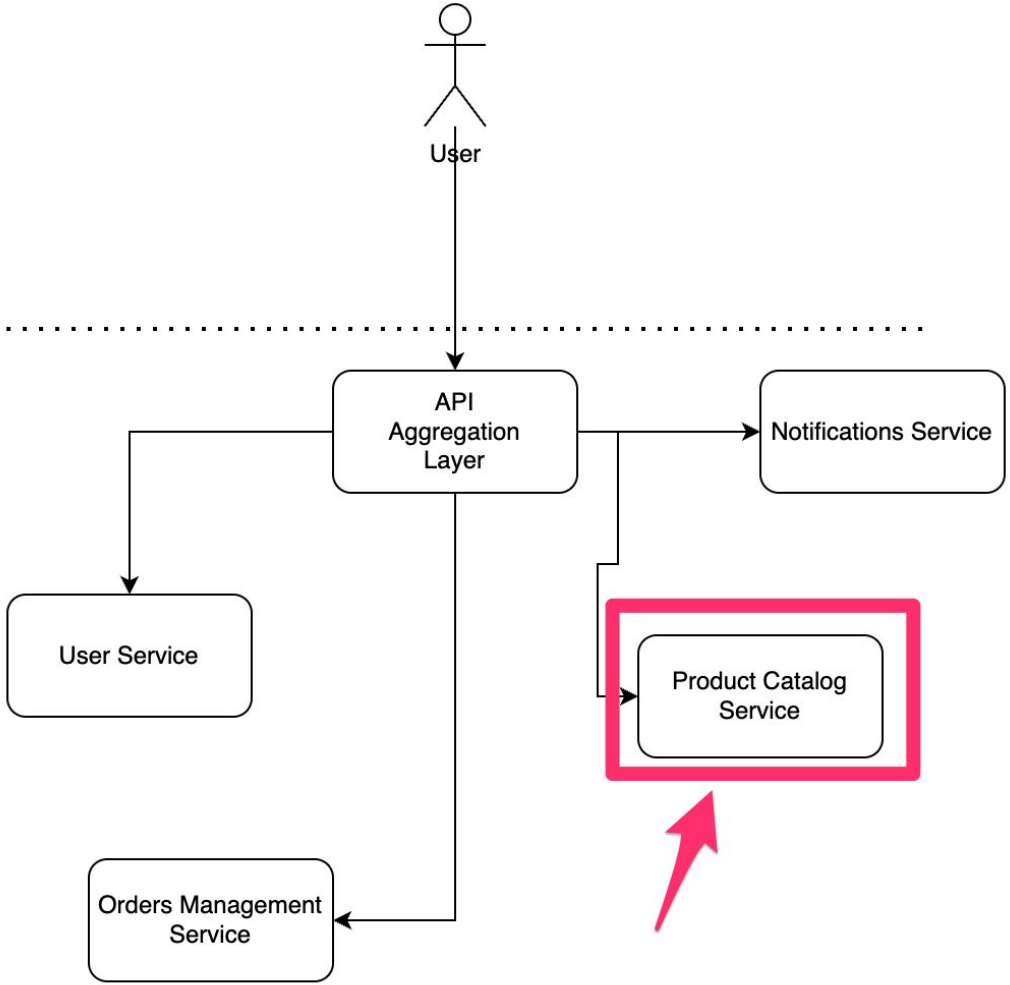
Spring I/O 2023


Person
id: A123
Email Updated
Name Updated
Person Created

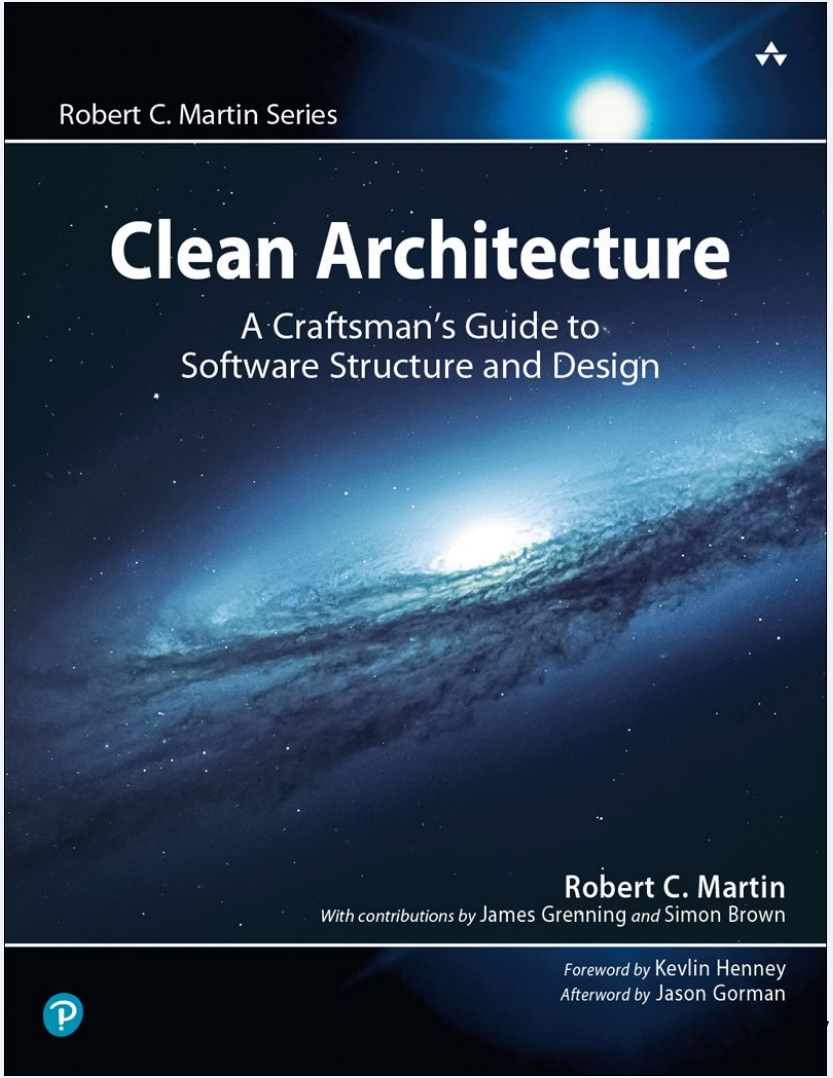

Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

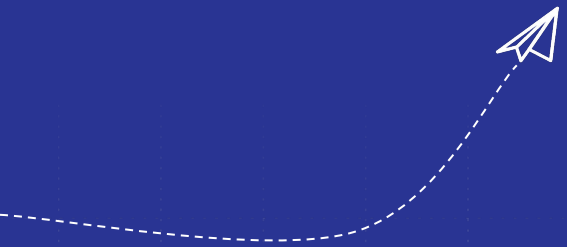

User
API Aggregation Layer
Notifications Service
User Service
Product Catalog Service
Orders Management Service

Stavvy


User
API
Aggregation
Layer
Notifications Service
User Service
Product Catalog
Service
Orders Management
Service

Some Light Pandemic Reading

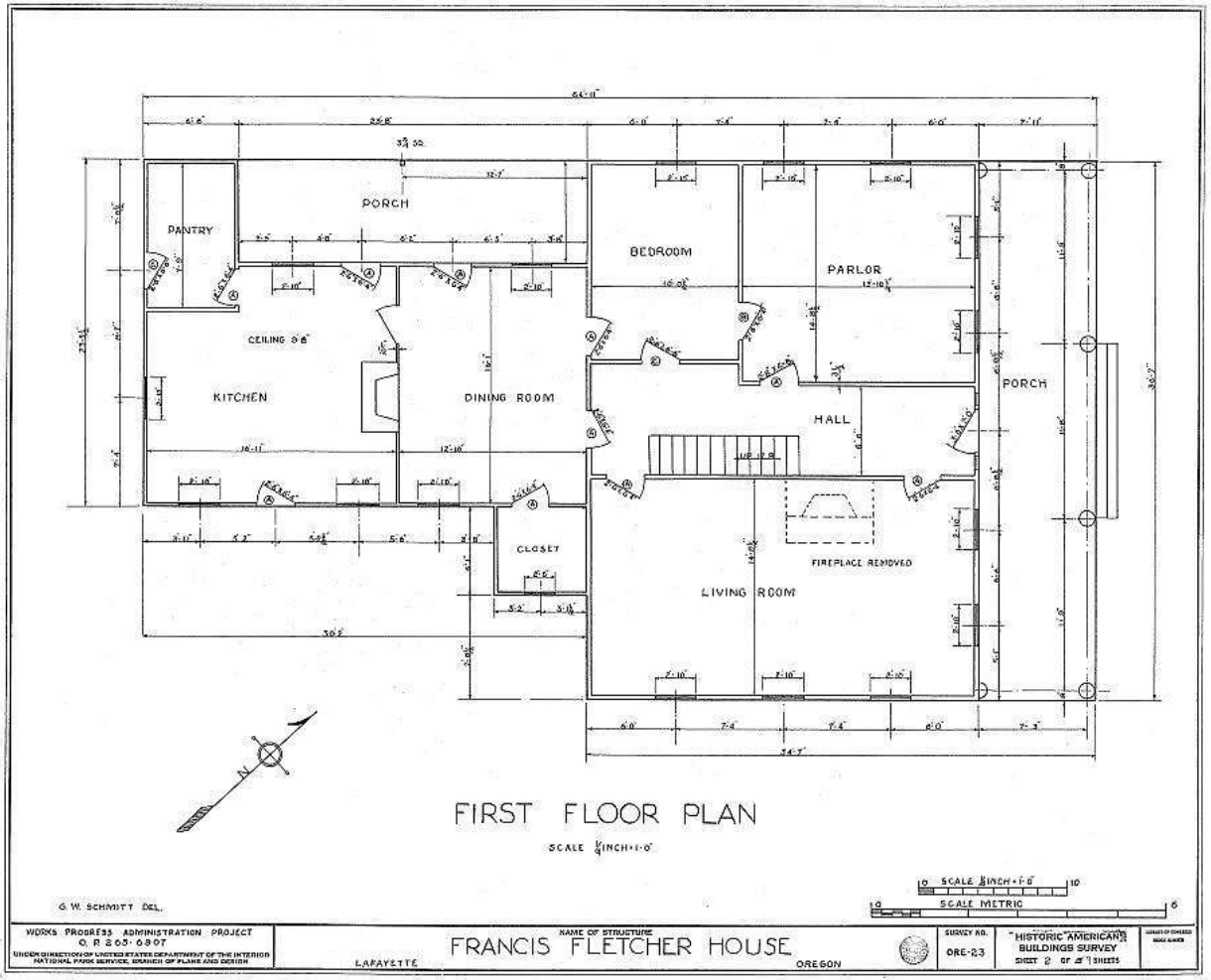


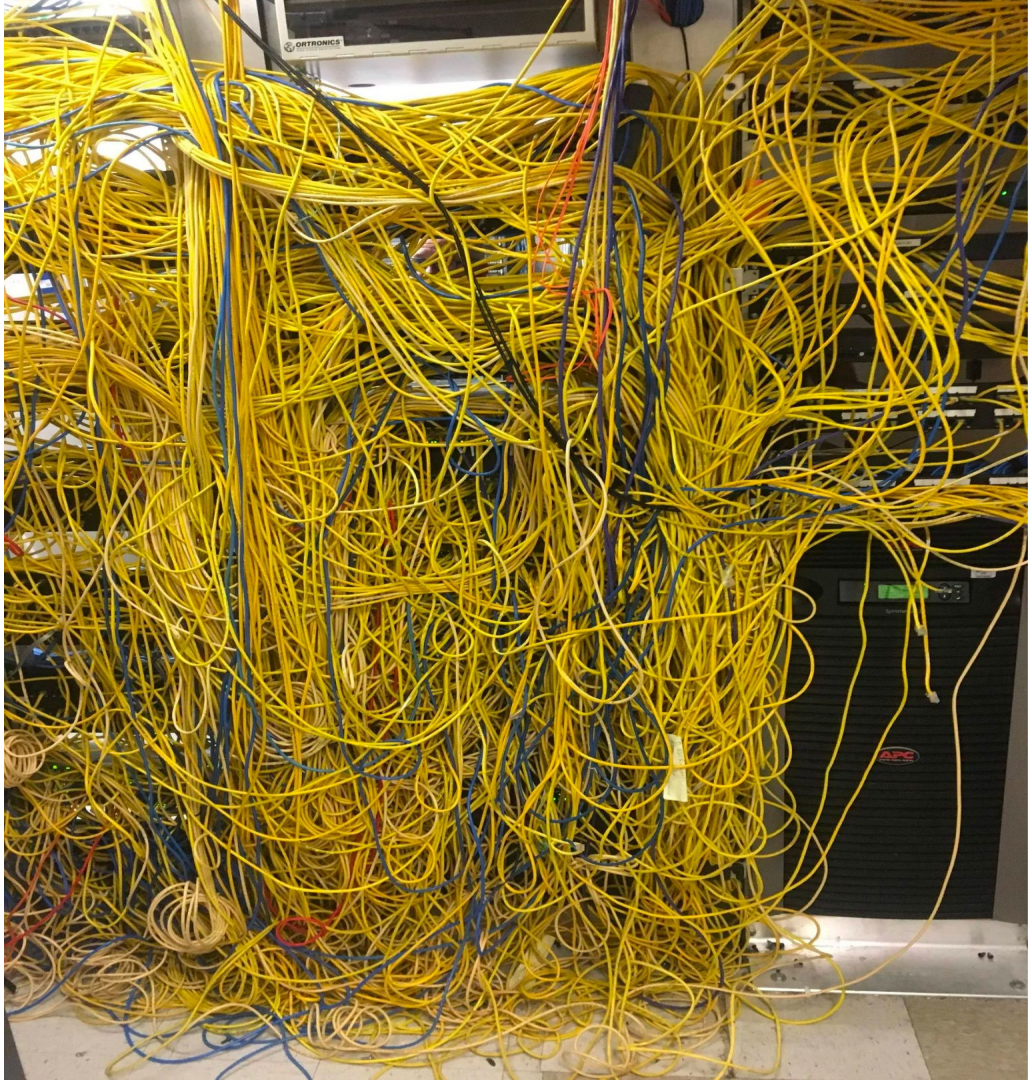


Today

- What is Software Architecture?
- Clean Architecture, an Overview
- Anatomy of a Spring App with Clean Architecture
- Demo App

What Even Is Software Architecture?




ORTRONICS
APC


MALLALA
NATIONALS
NATIONALS
NATIONALS
NATIONALS
Blimey
5
44
6
69


19
11
FedEx
3
TRACKER
14
9
41
SHW
HAAS
88
Nationwide
10
Smithfield
40
AdventHealth
6
66


Barilla
BUCATINI n.9
SPAGHETTONI n.7
COTTURA 12 MINUTI
MARCA N°1 IN ITALIA
SPAGHETTI
INTEGRALE

Some Benefits of a Well Architected Code Base

- Separation of Concerns / Modularity
- Flexibility
- Testability
- Maintainability
- Development Velocity

“Slow is Fast”

Software Architecture is the set of Structures to Guide Future development…

… and the ***discipline*** to adhere to those structures

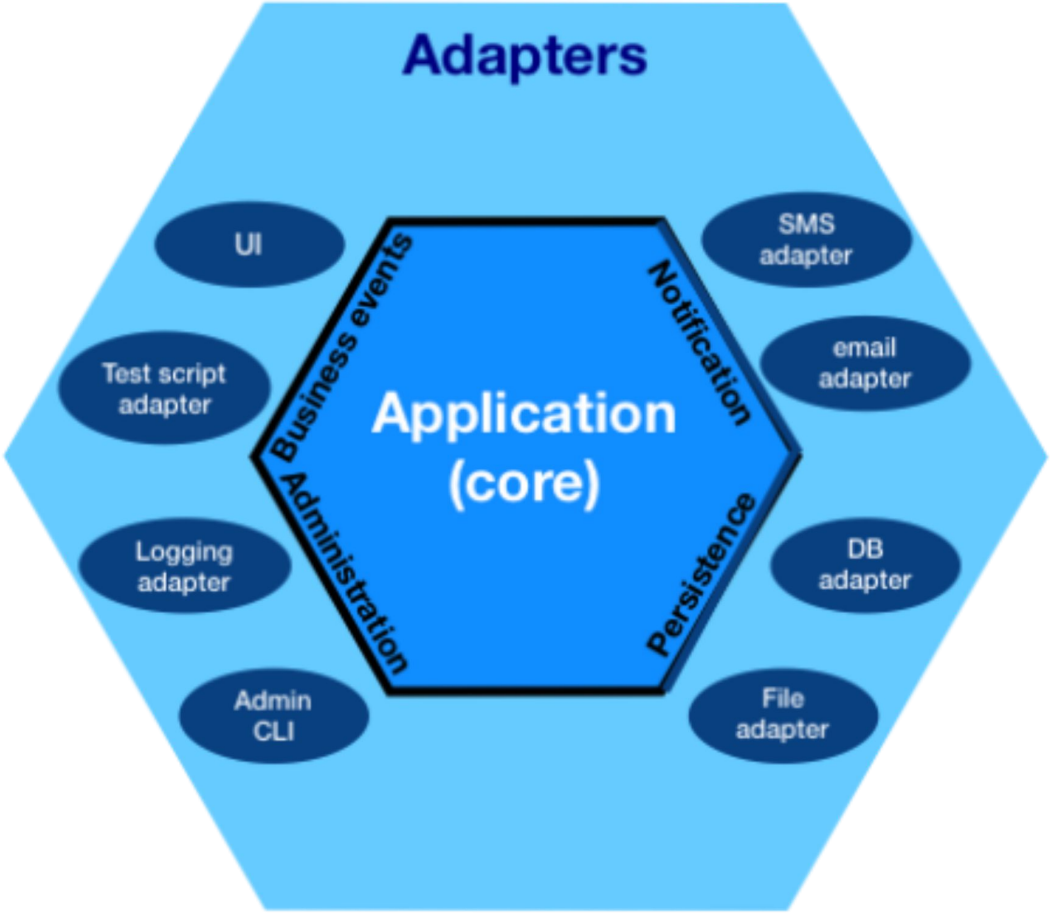
Existing Architectural Principles

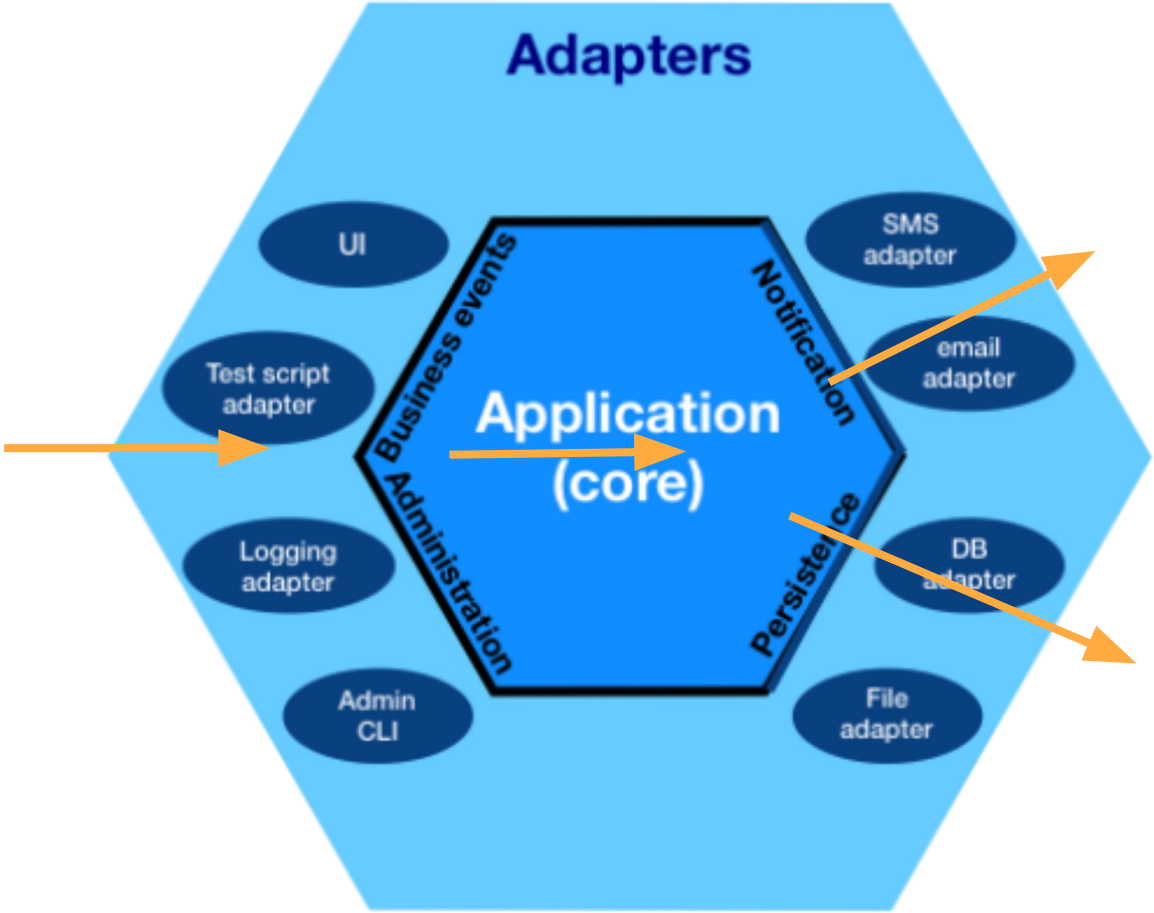
There are many Architecture Patterns,
for Specific situations.

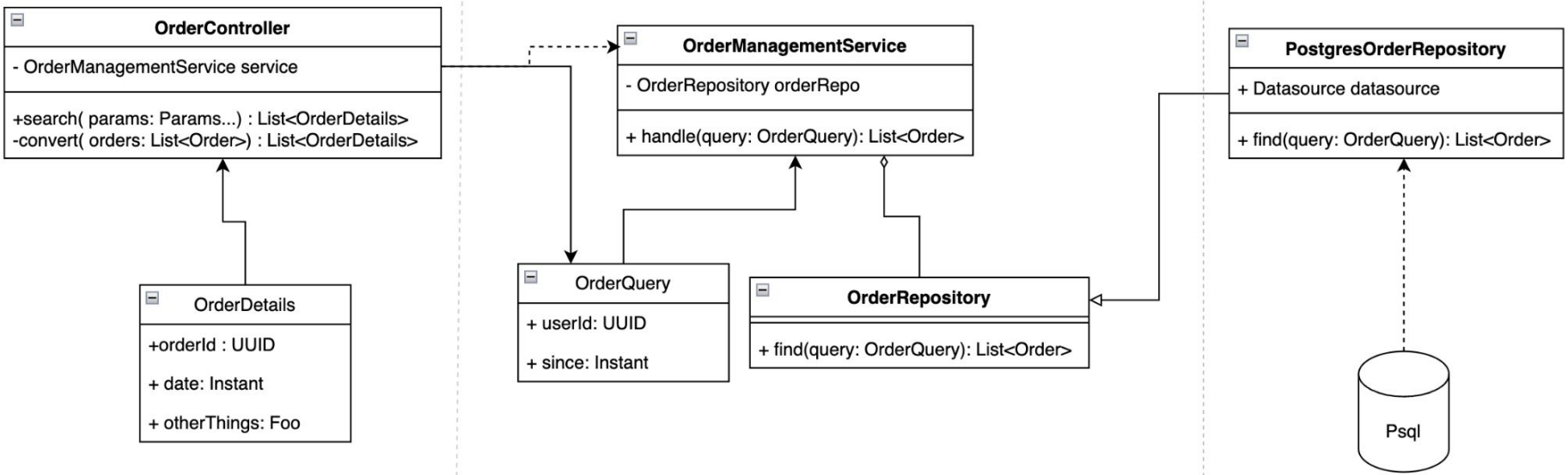
Focus on ‘Broad’ ones

Broadly Useful Patterns

- Ports and Adapters
- Hexagonal Architecture
- Onion
- SOLID + Clean Architecture


Adapters
UI
Test script adapter
Logging adapter
Admin CLI
SMS adapter
email adapter
DB adapter
File adapter
Business events
Notification
Administration
Persistence
Application (core)


Adapters
UI
Test script adapter
Logging adapter
Admin CLI
SMS adapter
email adapter
DB adapter
File adapter
Business events
Administration
Notification
Persistence
Application (core)


OrderController
- OrderManagementService service
+search(params: Params...) : List<OrderDetails>
-convert(orders: List<Order>) : List<OrderDetails>
OrderManagementService
- OrderRepository orderRepo
+ handle(query: OrderQuery): List<Order>
PostgresOrderRepository
+ Datasource datasource
+ find(query: OrderQuery): List<Order>
OrderDetails
+orderId : UUID
+ date: Instant
+ otherThings: Foo
OrderQuery
+ userId: UUID
+ since: Instant
OrderRepository
+ find(query: OrderQuery): List<Order>
Psql

Common features

- Layering + Separate Boundaries
- Agnostic, Encapsulated Core Domain
- Inversion of Control

Clean Architecture

Clean Highlights

1. S.O.L.I.D.
2. Component Principles
3. Boundaries & Dependencies
4. Nearly Everything is Just A “Detail”

1) Your Code Should Follow SOLID Principles

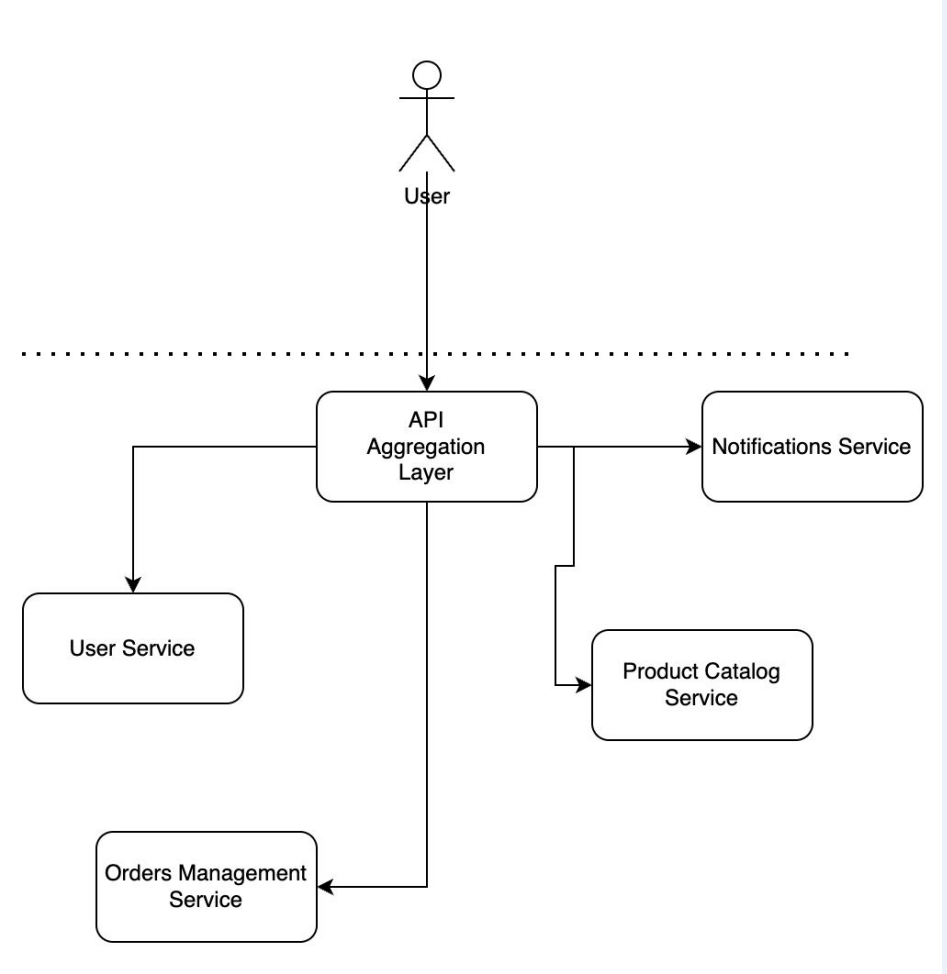
- **(S)ingle Responsibility**: a class should only have one responsibility, or one reason to change; used by a single user Role
- **(O)pen-closed**: “open for extension, closed for modification” (aka use interface implementation, Delegation, or Strategy patterns)
- **(L)iskov’s substitution**: aka “behavioral subtyping”; implementations/sub-classes should adhere to client expectations.
- **(I)nterface segregation:** code should not depend on methods it does not use (create more, smaller classes and interfaces)
- **(D)ependency inversion**: “depend upon abstractions, not concretions” (dependency injection using interfaces)

2) Component Principles

- A “Component” is the ideal boundary
- Partition a system into isolated, *independently developable* structures
- Components are handed out to teams
- Components should be able to be individually compiled, built, and tested
- Components should be Loosely Coupled

2) Component Principles

Microservices are the ultimate Component



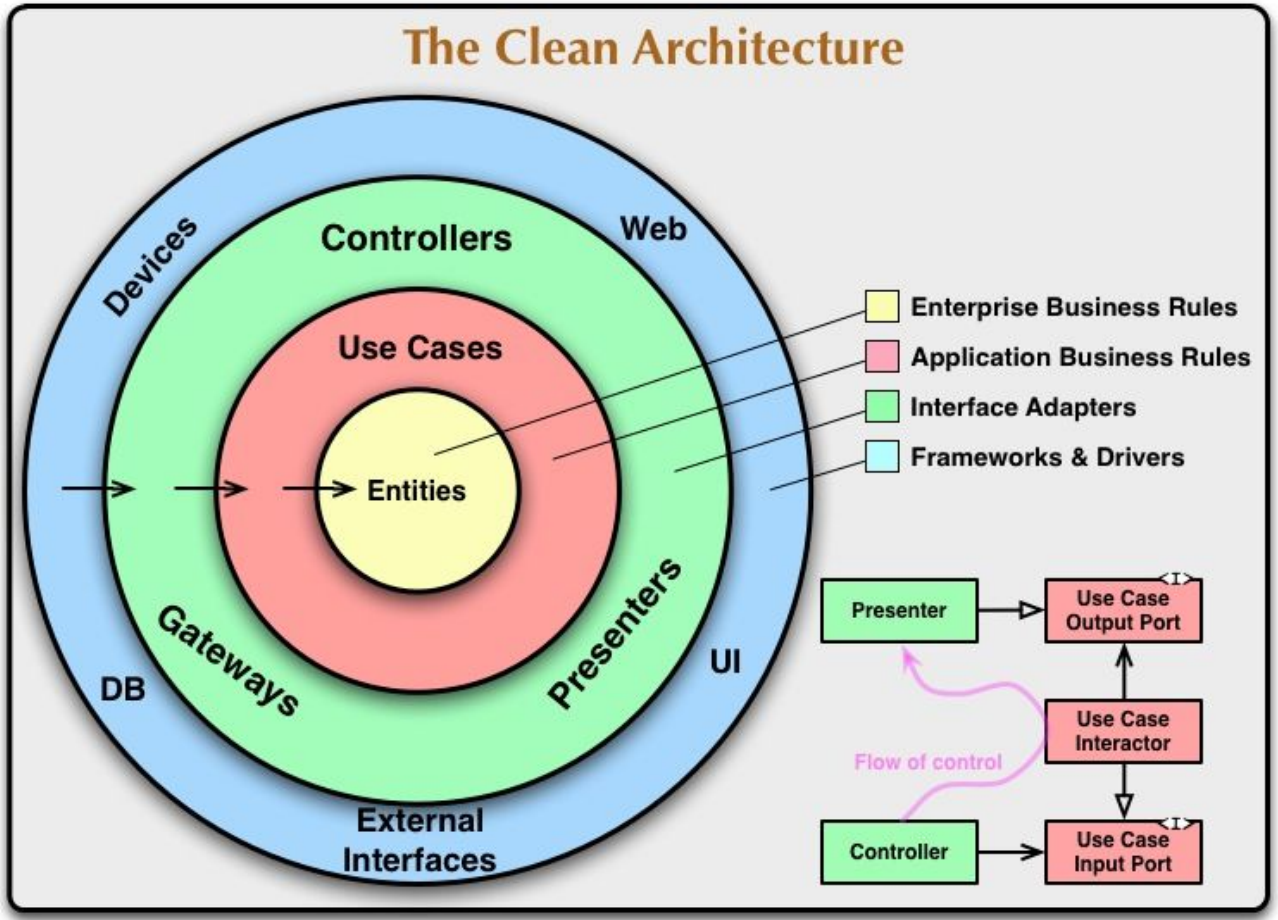
Stavvy

Difficult to Design Components “Top Down”

2) Component Principles

Avoid Dependency Cycles: relationships should form a D.A.G!

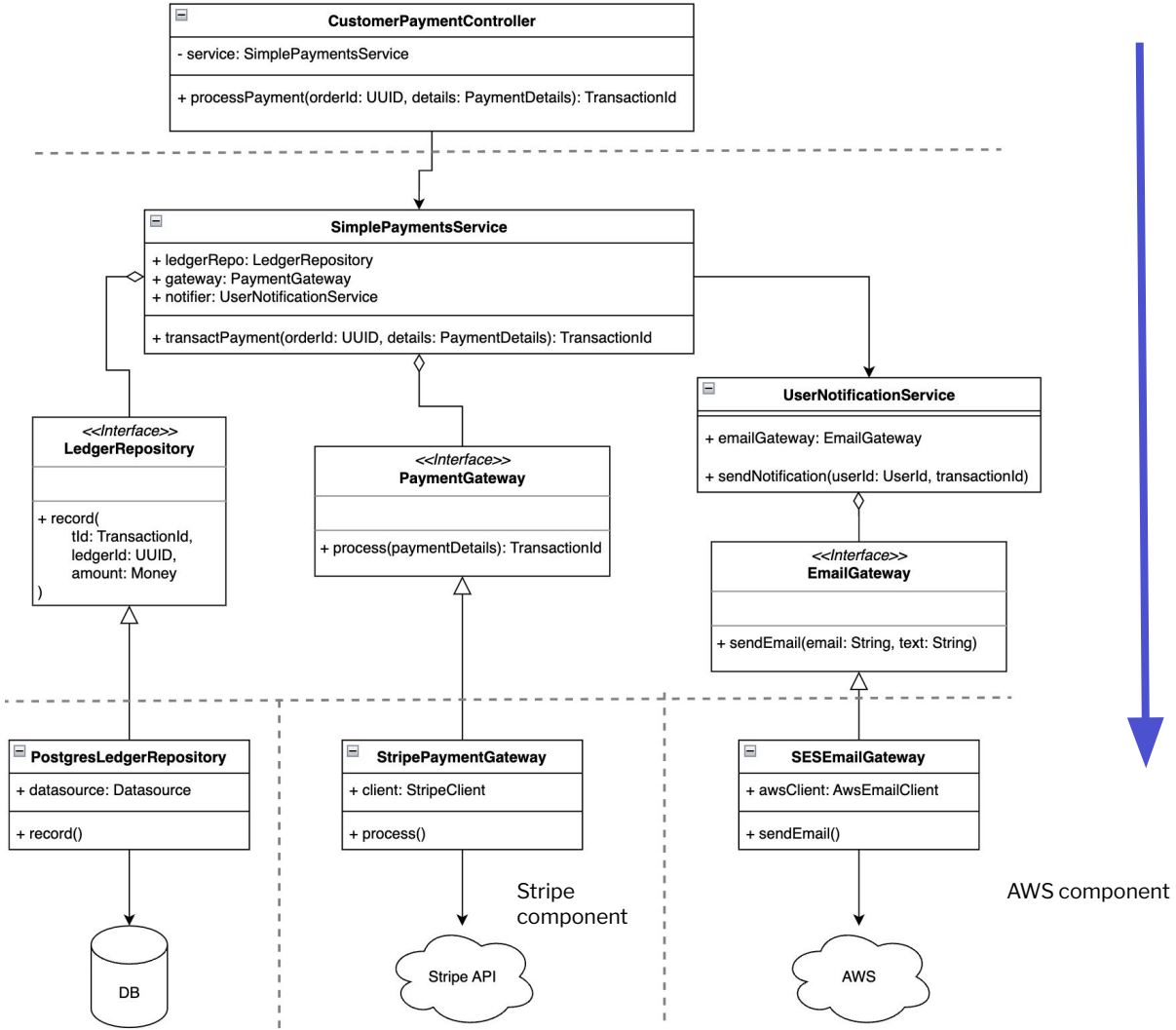
3) Boundaries and Dependency Flow



HTTP Component

Entities &
Services
Component

Postgres
Component



Main
Data Flow

Stripe
component

AWS component

Stavvy

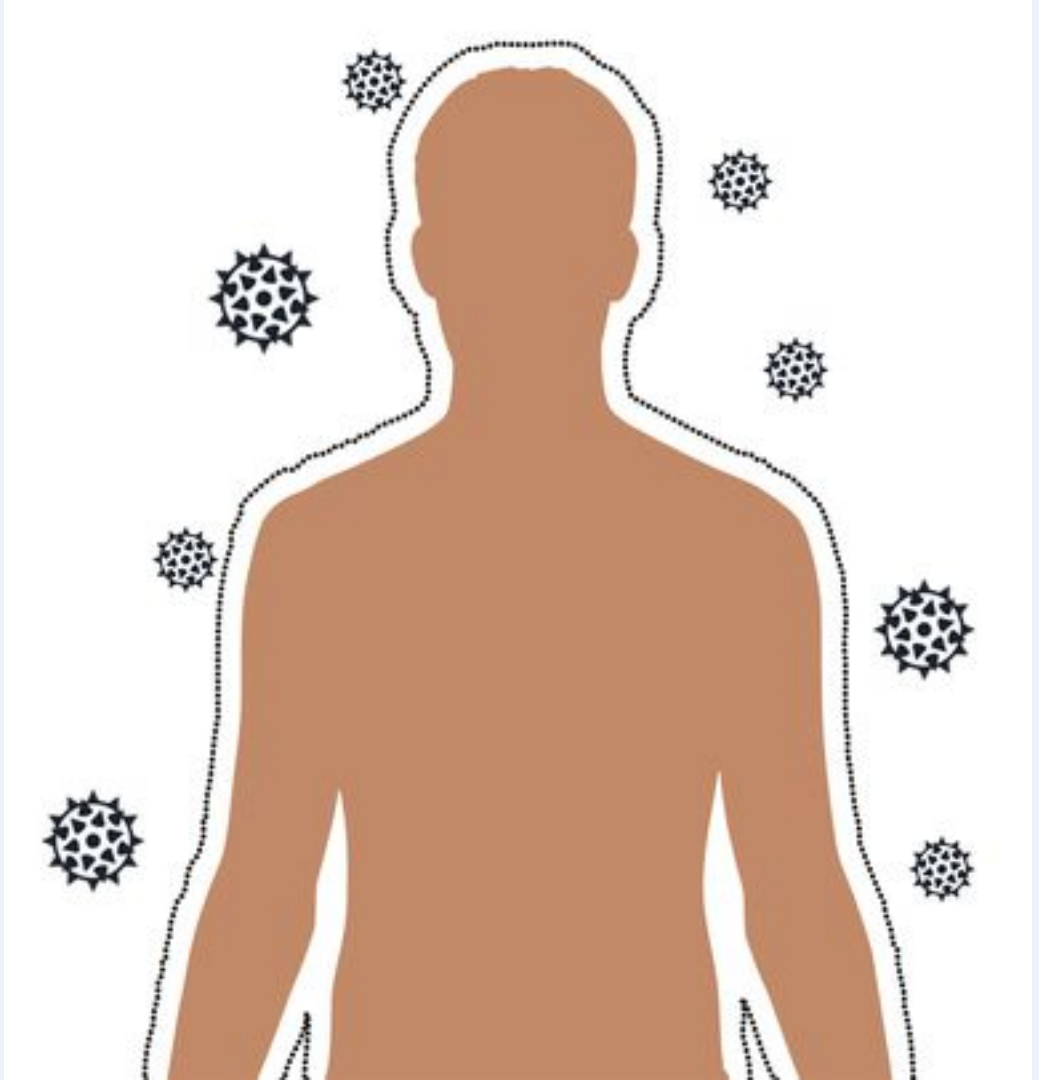
3) Boundaries and Dependency Flow

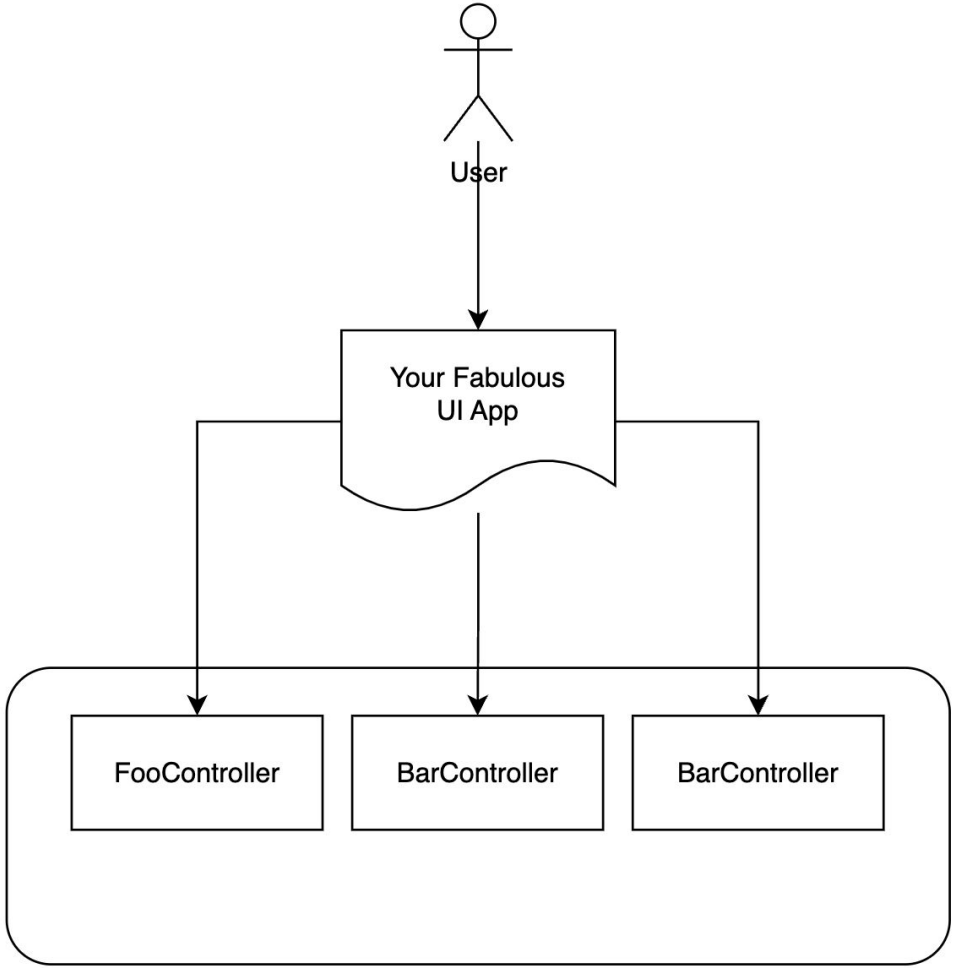
Simple Structures (method calls) going IN, Interfaces going OUT

```
package io.spring.shoestore.core.products

import java.math.BigDecimal

Steve Pember
data class ShoeLookupQuery(val byName: String?, val byPrice: BigDecimal? ) {
    Steve Pember
    fun isEmpty() = byName == null && byPrice == null
}
```

Outer layers may only see what is in an Inner layer!

Inner Layers Should Be Immune To Outer




User
Your Fabulous
UI App
FooController
BarController
BarController

4) Nearly Everything is an Implementation Detail

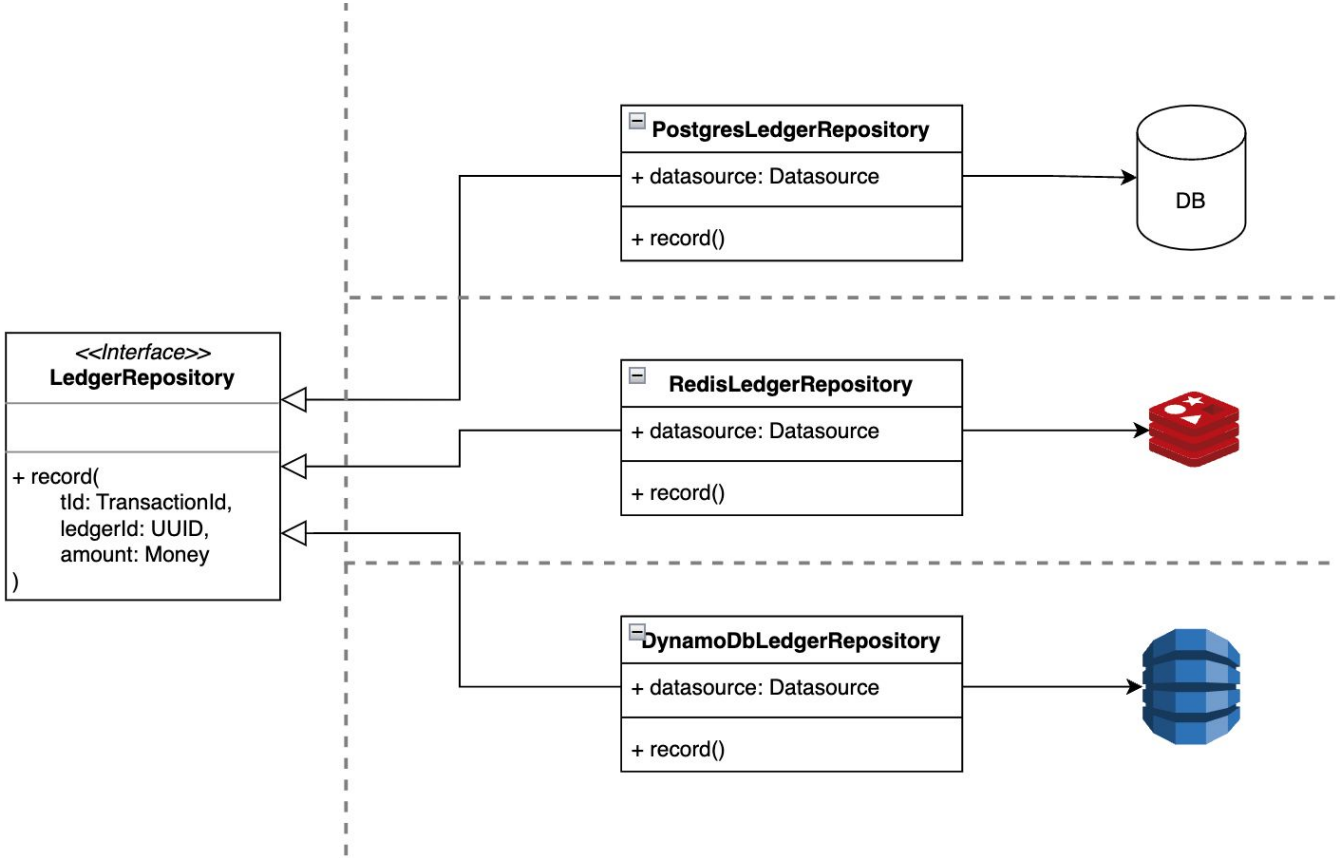
- Maintain Flexibility by separating code into “Policy” and “Details”
- Everything **not** in your “Core” is a Detail.

Defer Making Decisions about Details For As Long As You Can

4.1) Your Database Is A Detail

- Are we using Postgres?
- Mysql?
- Dynamodb?
- Redis?
- … it doesn’t matter. Not Really.

4.1) Your Database Is A Detail



Stavvy

4.1) Your Database Is A Detail

```
@Bean
fun getLedgerRepository(jedis: Jedis): LedgerRepository {
    return RedisLedgerRepository(jedis)
}
```

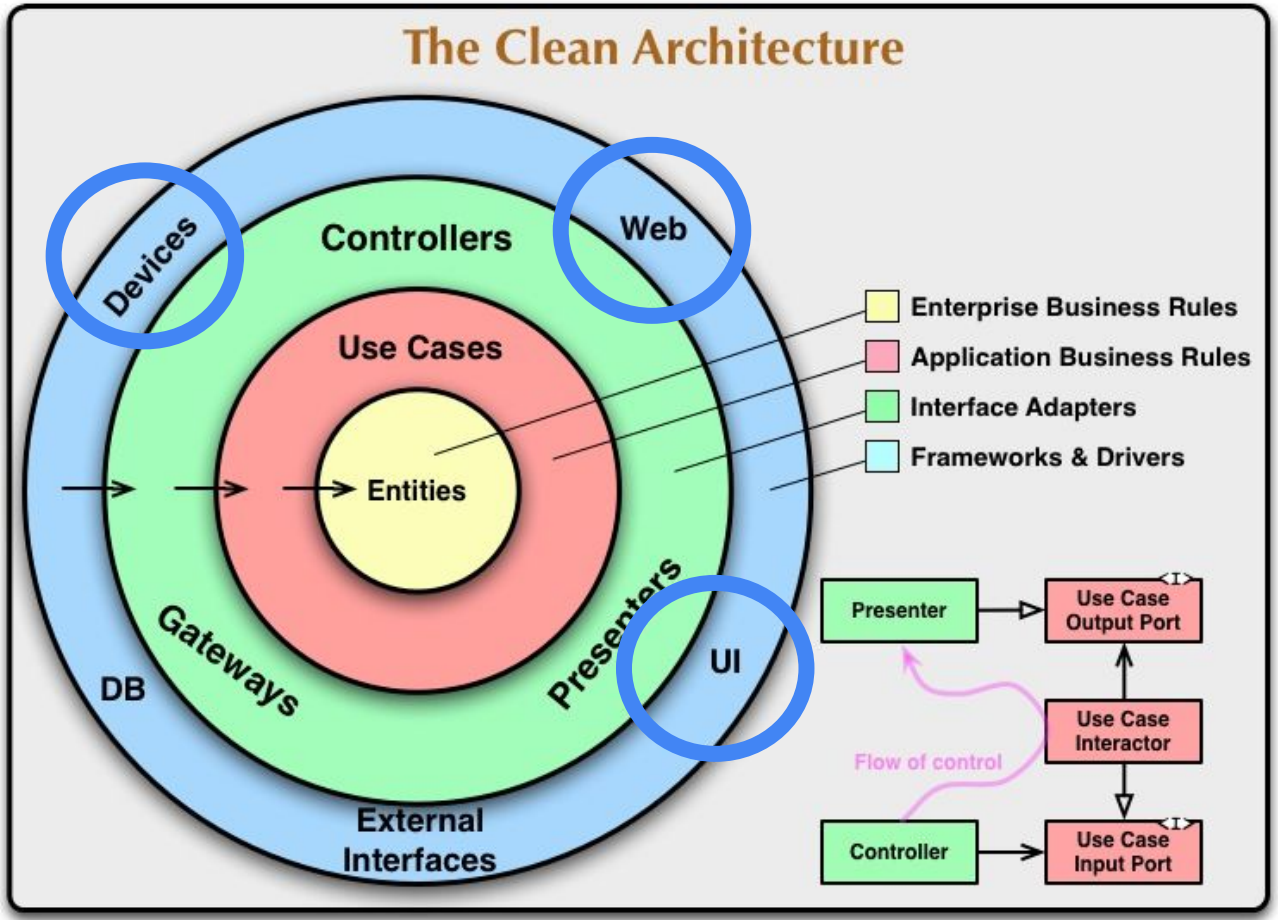
A Quick Tangent:
Don’t Think About Your Schema First

Makes me a tiny bit crazy

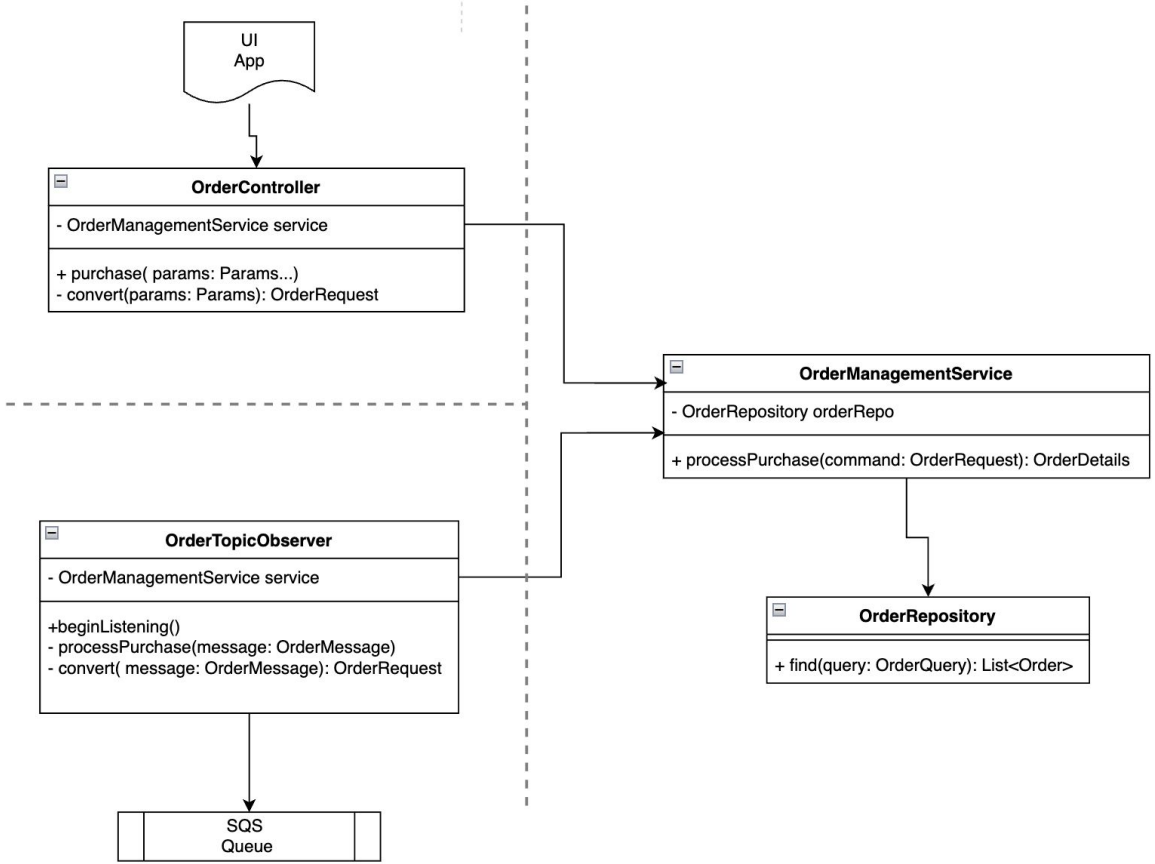
4.2) The Environment Is A Detail

- Your code must not know or care *where* it is being run
  - Local laptop
  - QA environment
  - Inside a Docker Container on Kubernetes
  - Production on an ec2 instance
  - Bare Metal DataCenter
  - Integration Test Suite
- … it doesn't matter
- Avoid "environment specific" configuration
- Inject Environment Variables or fetch Config Values dynamically

4.3) Input Is A Detail



4.3) Input Is A Detail



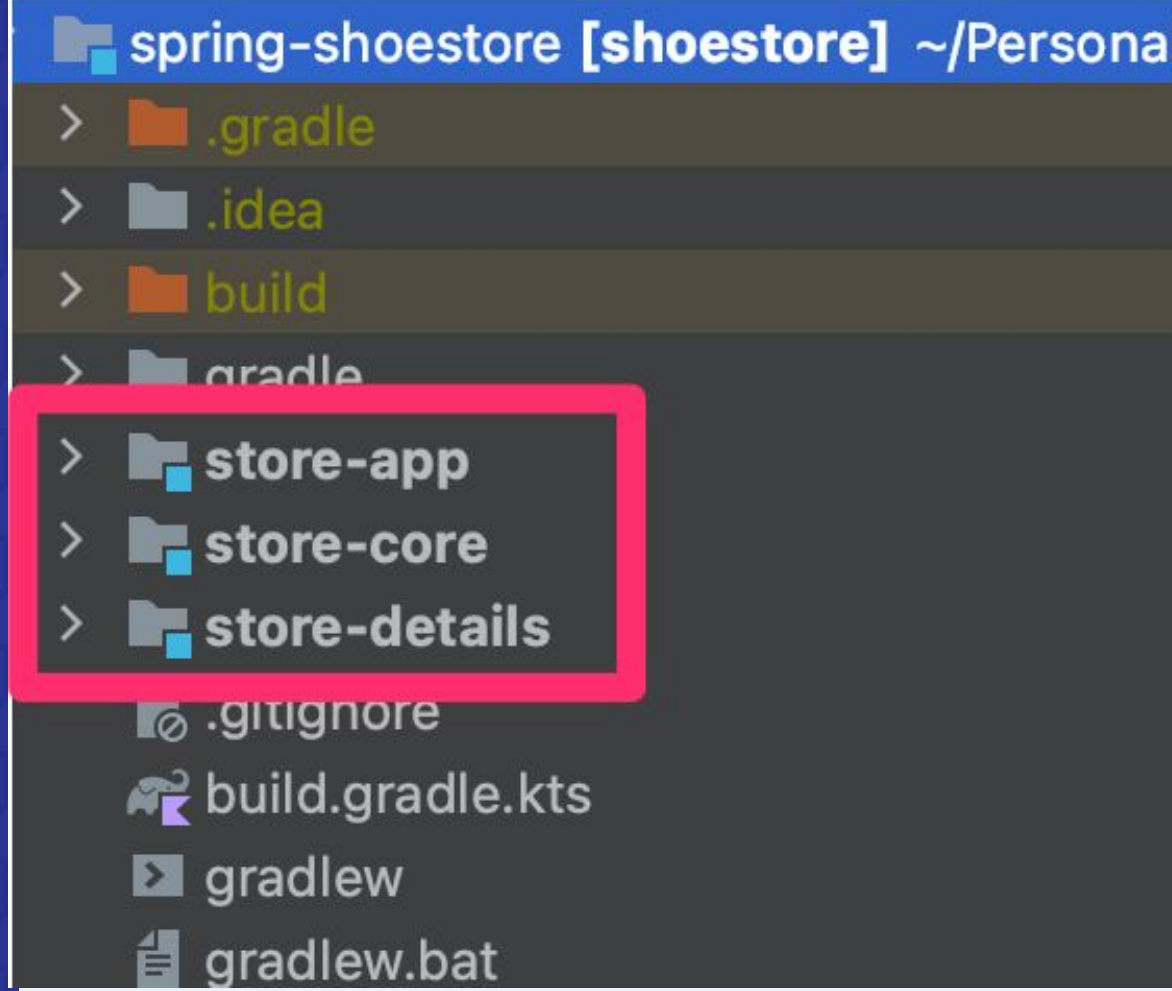
… 4.4) Even The Framework Is A Detail

Spring And the JVM

Clean Arch Examples

- Modules & Subprojects
- Controllers / HTTP Input
- Message Input
- Profiles + Env vars
- Testing
- “Output” Repositories

Modules / Subprojects as Components



⅓ - The Core

- Contains your core business logic
- Entities
- “Use Cases” / Service Classes
- Lightweight DTOs
- Interfaces
- Logging and Metrics? (Use Slf4j & Micrometer)

⅔ - The Details

- The Implementations of Interfaces
- Detail-Specific Class Structures
- External Clients
- SQL
- Good place for database Migrations

3/3 - The App

- AKA “Where Spring Lives”
- Application Configuration
- Application Lifecycle
- “Input” (e.g. Controllers)
- Impossible to Clean; the “Dirtiest” of components

Hide Your Internals

- Not Everything needs to be “public”

```
internal class ProductVariantMapper: RowMapper<ProductVariant> {
```

```
// package private!
class ShoeRowMapper implements RowMapper<Shoe> {
```

- module-info.java

```
module stavvy.game.supplier {
    exports com.stavvy.reference.gamesupplier.api;
    requires kotlin.stdlib;
    requires messaging.core;
    requires messaging.aws;
    requires org.slf4j;
```

These 3 are just a start. Plan on growing.

Input: HTTP / Controllers

- Simple: Use Spring Web / MVC
- Controllers should be as minimal as possible:
  - Convert User Input
  - Handle AuthZ / AuthN (use Filters)
  - Call into Core Services via function calls or lightweight objects

```
@GetMapping("/shoes")
fun listShoes(@RequestParam name: String?): ShoeResults {
    val query = ShoeLookupQuery(name, byPrice: null)
    return ShoeResults(shoeService.search(query).map { convert(it) })
}
```

Input: Messages / Queues

- Use Observer Pattern
- Connect / Disconnect In Spring Lifecycle Listeners
- Spring Integration & MessageTemplate

Service Classes

- Closest to “Use Cases”
- Keep them small and focused: Remember: Single Use
- Avoid “OrderService”, prefer “CustomerOrderQueryService”
  - It’s more specific and evocative!
- Almost all of them will live in **Core**

Created as @Beans manually

```
@Bean
fun getInventoryManagementService(jdbcTemplate: JdbcTemplate, jedis: Jedis): InventoryManagementService {
    return InventoryManagementService(
        PostgresProductVariantRepository(jdbcTemplate),
        RedisInventoryWarehousingRepository(jedis)
    )
}
```

Or

```
@Bean
// Kotlin-ified!
fun getInventoryManagementService(jdbcTemplate: JdbcTemplate, jedis: Jedis) = InventoryManagementService(
        PostgresProductVariantRepository(jdbcTemplate),
        RedisInventoryWarehousingRepository(jedis)
    )
```

Profiles and Env Vars

- Configuration should be Environment Agnostic
- Avoid Profiles for Environment Specific Config
  - Profiles are for profile-based things.
- Inject Env Vars
- Take a look at Spring Cloud Config and Spring Cloud Vault!

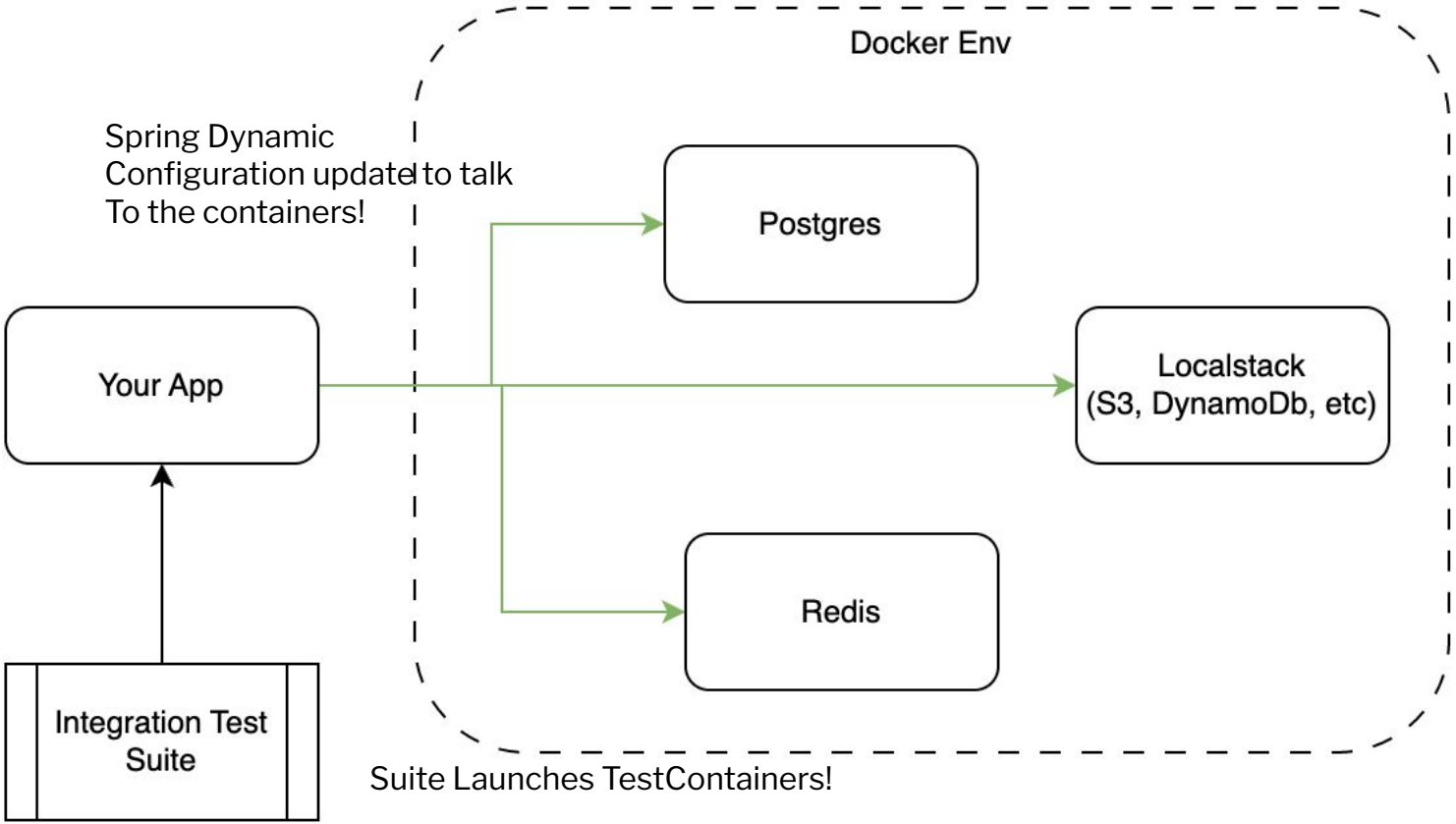
Beware Env Logic

```
if (environment!!.activeProfiles.contains("staging")) {
    // adjust some billing logic if we're in staging
    setPriceToZero()
    allowAnyUser()
}
```

Testing

- Components -> Fast Unit Tests
- Integration Tests go in the “App” Component
- Testcontainers (and Localstack) FTW!

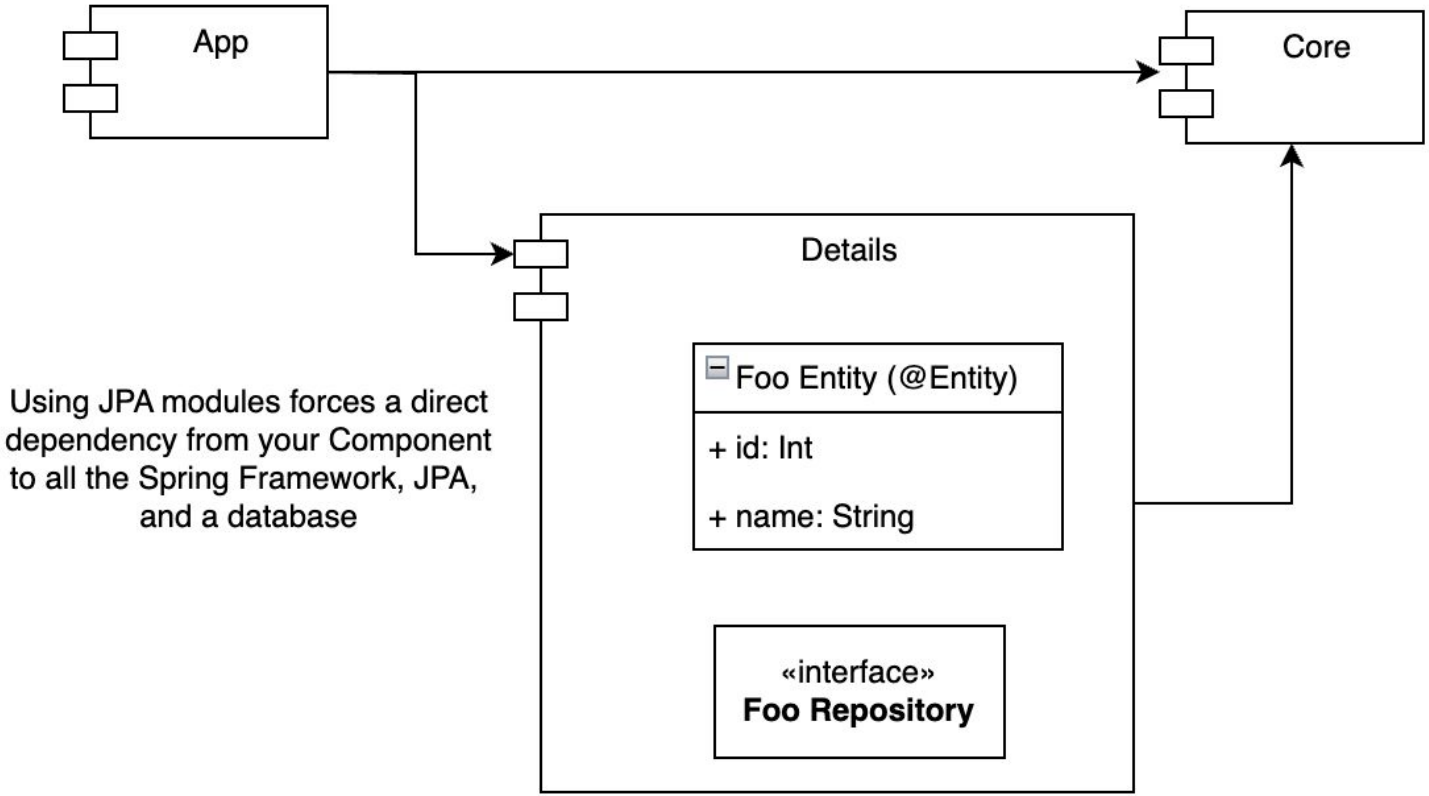
Testing


Docker Env
Spring Dynamic
Configuration update to talk
To the containers!
Postgres
Your App
Localstack
(S3, DynamoDb, etc)
Redis
Integration Test
Suite
Suite Launches TestContainers!

Datastores

- Remember: databases and stores are tools
- Everything goes behind a Repository Interface
- Implementation goes in the Details Module
- Databases, Caches, Microservice calls, Third Parties

Put JPA components behind a boundary



Time to Pray to the Demo Gods

Sure, following ‘Clean’ adds more Files & more mapping code…

… But it is an excellent way to achieve that “Guardrail Balance”

Thank You!

Any Questions?

Also happy to chat afterwards!

Stavvy

Links

- Twitter: @svpember
- Example Code Repo: https://github.com/spember/spring-shoestore
-

Images

- Blueprint:
- Messy IT closet
- Boxes of Spaghetti:
- Cars not crashing
- Car Crash
- Cobra Kai
- Dog Jump
- Tightrope Walker
- Immune System
- Blanket
- Crying Baby