Hexagonal Architecture in Opportunities Service

- Resources
- Motivation
- Core Concepts
- Strategies
    - Slicing
    - Mapping
    - Enforcing architectural boundaries
- Third-party libraries
- Current State of Opportunities Service
- Target Architecture for the Opportunities Service
- POC

Resources

- *[Book]* Get Your Hands Dirty on Clean Architecture (by Tom Hombergs)
- *[Book]* Building Mircro-services (by Sam Newman). Chapter 3: How to model services
- *[Article]* Hexagonal Architecture: the practical guide for a clean architecture

Motivation

We would like to leverage the most profitable advantages of the hexagonal architecture (also known as “ports and adapters”), which can be described as follows:

- It’s a **domain centric** architecture. As such, it **enables DDD** at the application core.
- It **decouples the domain logic from any foreign concerns** such as infrastructure, frameworks and technologies, so that changes in those areas do not require changes in the domain. The domain becomes **technology agnostic**.
- It works with the concept of **ports and adapters**, where each external component can be plugged to a specific port to access the core of the application, in a sort of “plug & play” style.
- **Every dependency points to the center of the application towards the domain**, hence the domain is **self-contained** and **stand-alone**. This also means testing is easier and tests are more proficient as they can concentrate in the pure domain logic which is not polluted with technical requirements.
- All these features result in a **significant maintainability improvement**. More maintainable code means changes are less expensive and more efficient. It also makes onboarding of new team members easier.

Core Concepts

In a traditional layered architecture we tend to think about ***what’s up*** and ***what’s down*** (or what’s left and what’s right). In hexagonal architecture, however, we think in terms of ***what’s in*** and ***what’s out*** (and what can pass). This is an important shift in the way that we perceive architectural dimensions.

We apply the **Dependency Inversion Principle** to ensure that every dependency points ***inwards***. This principle states that we can invert the direction of a dependency and make it go against the direction of the control flow. This will be based on the way that we define modules and packages to distribute interfaces and implementors. See pictures below.

We are also applying the **Single Responsibility Principle**, which states that a component should have only one reason to change, since by decoupling the domain logic from external factors we are eliminating most of those reasons.

Let’s define the main elements of this architecture:

- **Application**
    - **Domain**
        - ***Models*** (can be anaemic or contain rich domain logic)
        - ***Services*** (execute use case logic around the domain models)
    - **Ports**
        - ***Incoming ports*** (to drive the application, also known as **use cases**)
        - ***Outgoing ports*** (to be driven by the application)
- **Adapters**
    - ***Incoming adapters*** (driving the application)
    - ***Outgoing adapters*** (being driven by the application)

- External components or devices (database, UI, external APIs, etc)
- The domain entities and services should also be free of dependencies on **third-party libraries** and **frameworks** like Spring and such.

The way in which we assemble our application also plays a key role. We make use of **dependency injection** to guarantee that all dependencies are provided at runtime by a neutral component (typically Spring framework) upon application start up so that no component needs to instantiate (i.e *know*) any particular implementation.

Images taken from the book Get Your Hands Dirty on Clean Architecture.

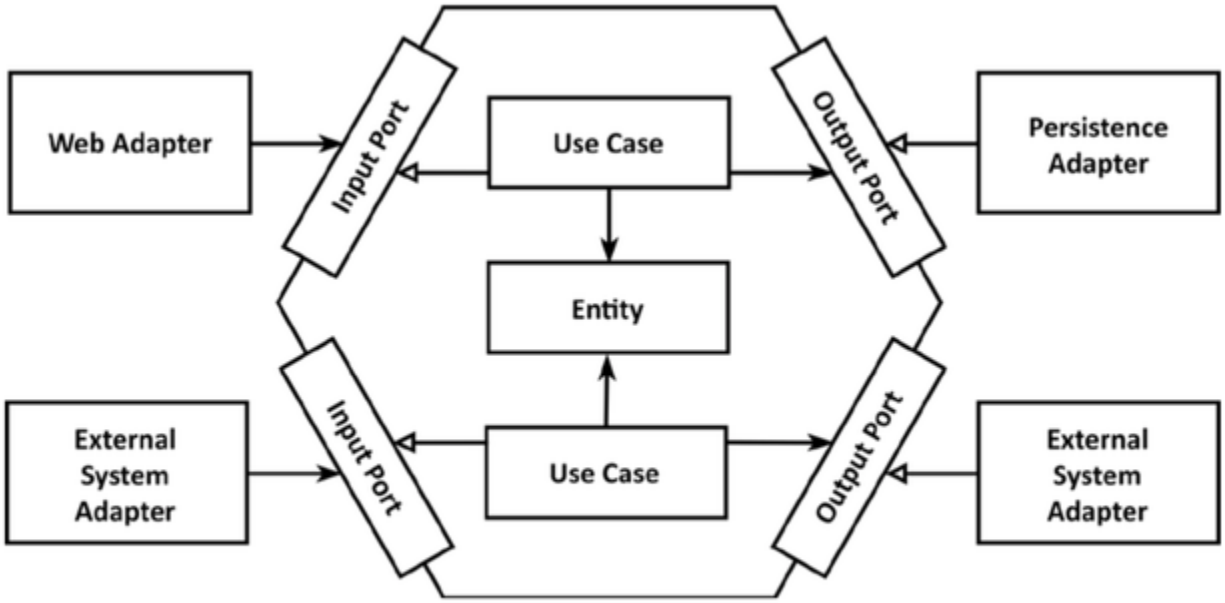


Figure 2.4: A hexagonal architecture is also called a "ports-and-adapters" architecture since the application core provides specific ports for each adapter to interact with

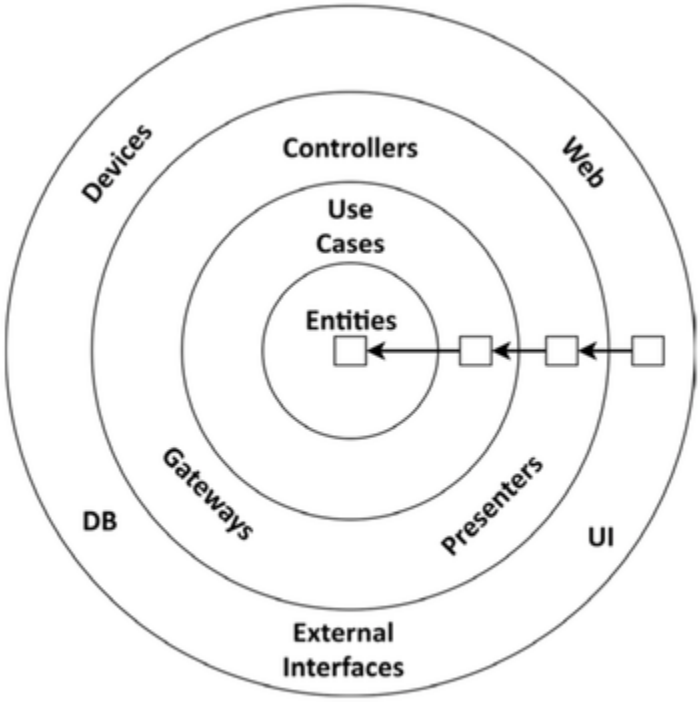


Figure 2.3: In a clean architecture, all dependencies point inward toward the domain logic.
Source: "Clean Architecture" by Robert C. Martin

Strategies

In order to fulfil this kind of architecture there are some key decisions that have to be made and which will depend in the characteristics of project that we are working on, being the most important distinction whether we're refactoring an existing project or we're spinning up a whole new one.

The three most relevant strategies that we need to define are:

- **Slicing**
- **Mapping**
- **Enforcing architectural boundaries**

These strategies **are not independent from each other** as the way that we define them will impact on the way that we define the remaining ones.

Let's get into details:

Slicing

Slicing is **the way that we will split interfaces and implementors**, whether those are ports, adapters or services.

We want to apply the **Single Responsibility Principle** (a component should have only one reason to change) and the **Interface Segregation Principle** (broad interfaces should be split into specific ones so that client only know the methods they need) as this will contribute to **decoupling our application transversally** (across a service, port or adapter). This means **even less reasons to change** in every component of our application, not only the domain logic.

Ideally, this would mean splitting our services, ports and adapters so that each of them holds a single responsibility and is connected to other interfaces and implementations that in turn hold a single responsibility too. For example, a service that implements one single use case (single method) which is connected to ports and adapters supporting that single use case as well. The opposite of this would be a set of broad interfaces and/or implementors supporting many use cases each.

In the particular case of an outgoing adapter such as a persistence adapter, it's suggested that the ideal slicing strategy would not be *per operation* but ***per aggregate*** (cluster of domain models conforming a unit, e.g. opportunity + items).

Mapping

The kind of mapping that will take place between layers (between adapters, ports, and the application core). This will help either **enforce** or **loosen** architectural boundaries.

Some of the possible mapping strategies:

- **No mapping**
  - Domain model is used everywhere
  - Good for *queries* or *CRUD* if exactly the same information in exactly same structure
- **One-way mapping**
  - All models from all layers implement same interface
  - Good for DDD factories
  - Best when models across layers are similar like in *queries* or *CRUD* operations
- **Two way mapping**
  - Adapters have their own models
  - Domain model still resides in the ports
- **Full mapping**
  - Separate *input* and *output* model *per* operation
  - Best from *incoming adapter* to *application layer* to demarcate *state-modifying* use cases
  - Better *not* from *application layer* to *outcoming adapter* due to mapping overhead

Mapping strategies **can and should be mixed**. The following should be considered:

- *Modifying use cases* **vs** *queries*
- *Incoming adapter and application layer* **vs** *application layer and outgoing adapter*
- Different mapping strategies for *different use cases*

We can start with a simple strategy that allows us to quickly evolve the code and later move to a more complex one that helps us to better decouple the layers. **Simple mapping strategies should always be kept an eye on and be rapidly changed** as soon as the layers' concerns start leaking into one another.

Enforcing architectural boundaries

The key to a successful hexagonal architecture that evolves in a healthy way over time are the architectural boundaries. These boundaries will **ensure that no dependency will be pointing the wrong way at any given time** and will make it difficult for a developer to introduce such a dependency. Or at least will force him/her to do it consciously instead of accidentally.

Three ways to achieve this:

- Using the Java ***package-private*** visibility modifier:
  - If we are working in a **single module** this modifier can help us conceal certain classes and interfaces. In a single module every element in the architecture can potentially access the others.
  - The general rule on visibility is this one: ***domain models*** and ***ports*** are **public** (domain models are needed by the services implementing user cases, and ports are interfaces that need to be implemented), ***services*** and ***adapters*** are **package-private** (services and adapters are not accessed by anyone).
  - The problem with package-private modifier is that **sub-packages can not see classes in the parent package** marked as package-private, which makes it difficult to use when there are many classes in the project (most of cases).
- Using **post-compile checks**:
  - We could execute **runtime checks during automated tests** within a continuous integration build.
  - A tool that supports this kind of check for Java is **ArchUnit** (https://github.com/TNG/ArchUnit). With ArchUnit we can check the dependencies between our layers, assuming that each layer has its own package.
  - The downside is that a single refactoring **renaming a package can make the whole test useless**. Post-compile checks always have to be maintained parallel to the codebase.
- Using **build artefacts**:
  - Using a build tool such as Gradle we can define any number of independent modules. **Boundaries between modules can not be crossed** unless there's an explicit dependency in the build script.
  - The finer we cut our modules, the stronger we can control dependencies between them. The finer we cut, however, the more mapping we have to do between those modules, **enforcing one of the mapping strategies**.

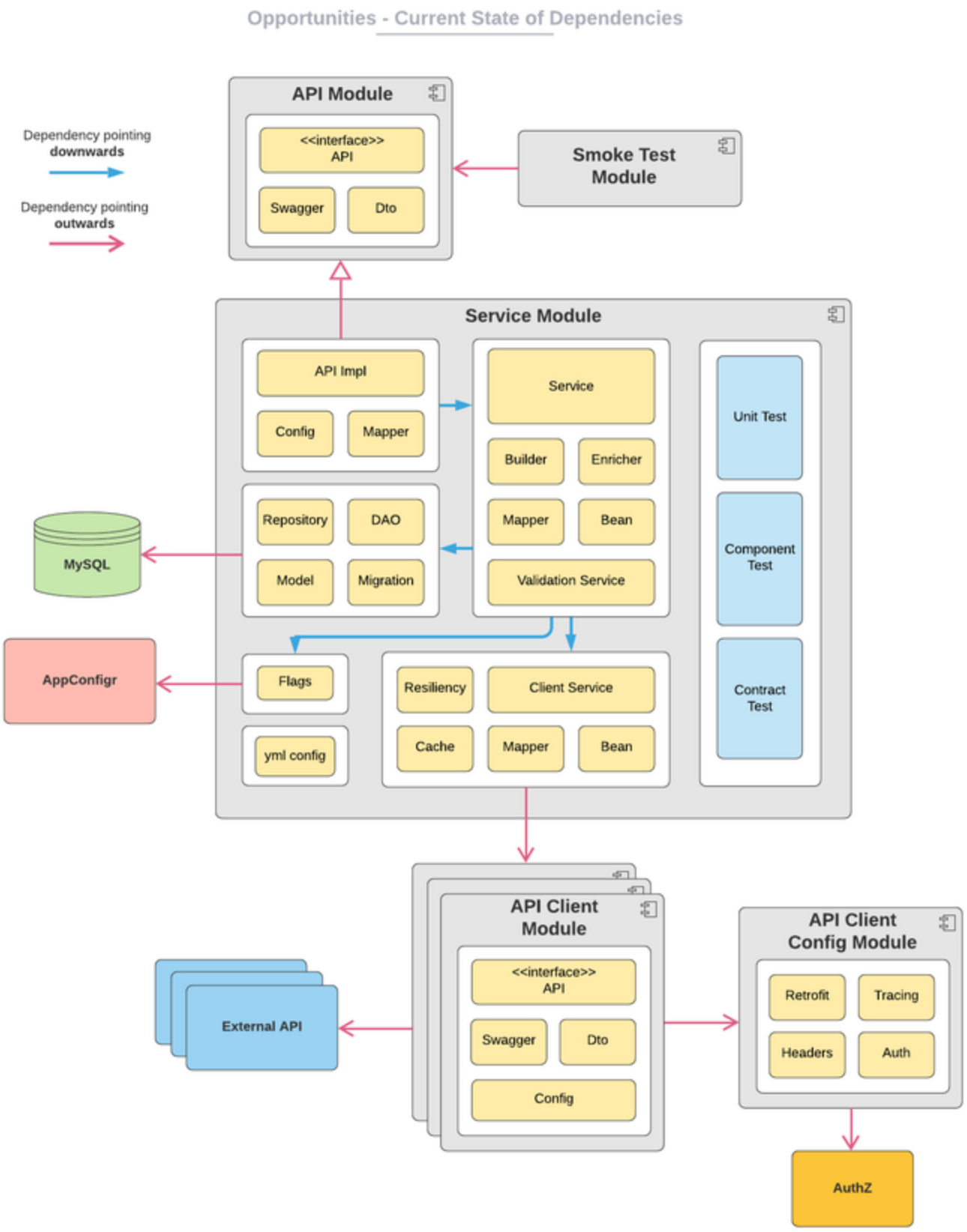
Third-party libraries

We typically use Spring framework to assemble our application and take care of all the dependency injection aspects, either through class pass scanning or configuration classes. The problem is that *stereotype* and *autowired* annotations pollute the domain code with a **dependency to third-party libraries**.

Here is an article on a possible approach to decouple the domain from Spring annotations using custom annotations: Binding the Domain to the Spring Context with ComponentScan. Anyhow, we should consider that using Spring annotations in the domain might be a reasonable trade-off after all.

Current State of Opportunities Service

The following diagram depicts the current state of the application:

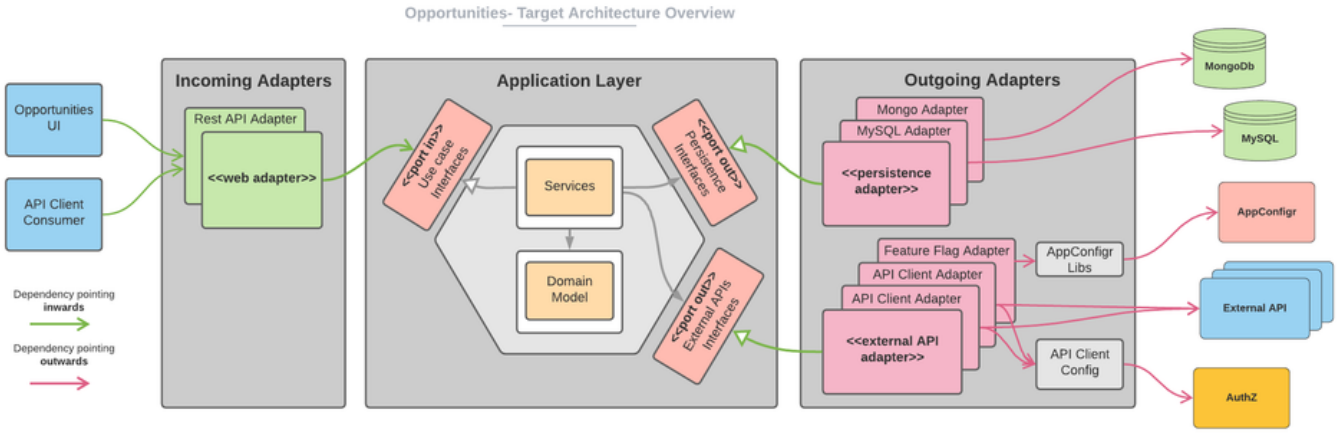


- There seems to be a mix of two different architecture models depending on the scope:
  - At **overall module level** we have some sort of **antagonist to the hexagonal architecture** since the service module depends on the others, hence the dependencies are **pointing outwards**. There is also **immediate dependency to external devices** like the MySQL database and AppConfigr service.

    - At **service module level** it resembles a traditional **three layered architecture**, although there are no formal boundaries between internal components, and dependencies are **pointing downwards** (API Service DAO / API Client / Flag)
- The **main module (service) is "monolithic"** and contains many disparate components (API, DAO, Service, Client Service, Flags) which are **tightly coupled**.
- There are **no boundaries whatsoever between components in the service module**, hence every component can develop a dependency on any other very quickly and easily.
- **Domain entities are anaemic** as they are just data transfer objects.
- **Domain logic** resides mostly in the services, builders, enrichers and mappers of the service component inside the service module.
- **All interfaces and classes are broad**, meaning that they accumulate the sum of use cases/operations for a given component. There is **no slicing** strategy whatsoever.
- There is a **full mapping strategy implemented in the client services modules** (client beans + client ws), and a **two-way mapping in the DAO** (database model + domain bean) **and API** (web ws + domain bean).
- There are **Gradle build modules** of API, Service, API Clients and API Client Config.
- The *OpportunityService* class has dependencies on the following **third-party libraries** (most strong ones):
    - `springframework.beans`
    - `springframework.data`
    - `springframework.stereotype`
    - `springframework.transaction`

Target Architecture for the Opportunities Service

The following diagram expresses the proposal for the target architecture:



- All the components have been arranged in accordance to the hexagonal architecture concepts
- We have all the **expected elements** such as ports, adapters, and an application core comprised of services and domain models.
- All dependencies are pointing **inwards**
- The **domain is decoupled** from the infrastructure
- **A strong enforcement of architectural boundaries through build modules is encouraged**. This refactoring is an important effort in a fast-changing and complex application, therefore we would like to make sure that the new architecture doesn't degrade easily. The optimum build module structure in this case should be conformed by:
    - One module per adapter
    - One module for input ports
    - One module for output ports
    - One module for the application core (domain model + services)
- Such a module structure requires a **full mapping strategy** on all ports/adapters since the ports themselves are located in their own independent modules. We already have such a strategy on the API clients side (client bean + client ws), but we would need to move persistence adapter (DAO) and web adapter (API) from two-way mapping to full mapping. This is a lot of **mapping overhead** but it might be a reasonable price to pay for a **cleaner stronger architecture**. The alternative would be to keep the ports inside the application module but this loosens the boundaries significantly.
- Even though we currently have broad interfaces and services everywhere **it would not be recommend to take on slicing tasks as part of the initial refactoring**. This is already a very large and complex refactoring to also have to deal with slicing strategies. We should probably leave this task for a future iteration.
- For the same reason **it would't be recommended to try to turn the anaemic domain models into rich models** with the initial refactoring. This is also a candidate for a future iteration where we could dig in more profoundly into DDD concepts.

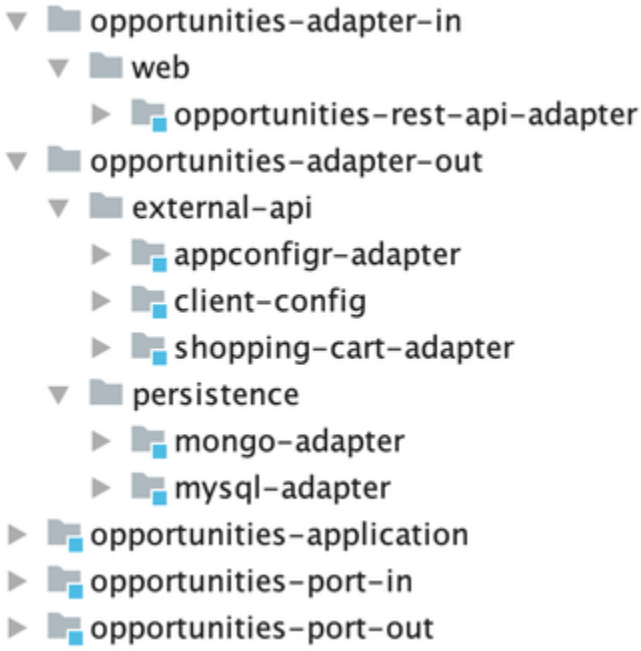
- We might want to **take on decoupling domain from Spring's annotations for another time** as well.

POC

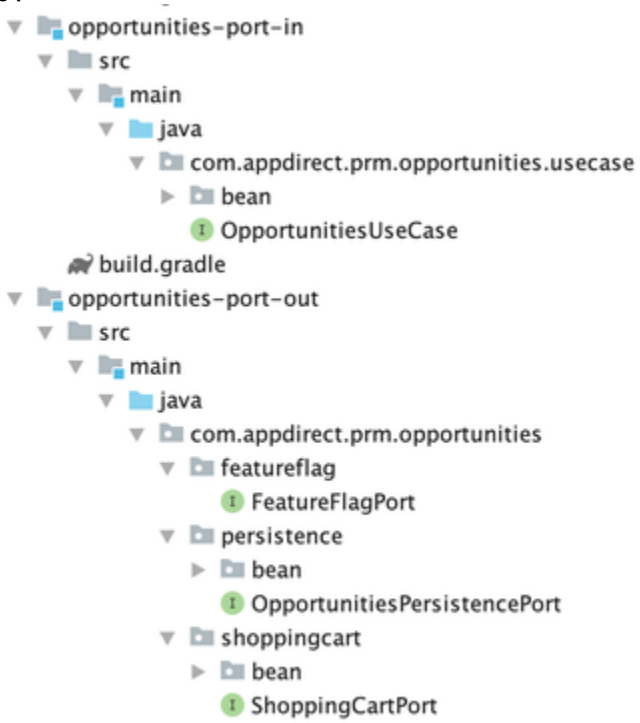
Here's a GitHub PR on the **prm-opportunities** repo with a proof of concept, which has been built around a single use case of *Create Opportunity*:

https://github.com/AppDirect/prm-opportunities/pull/318

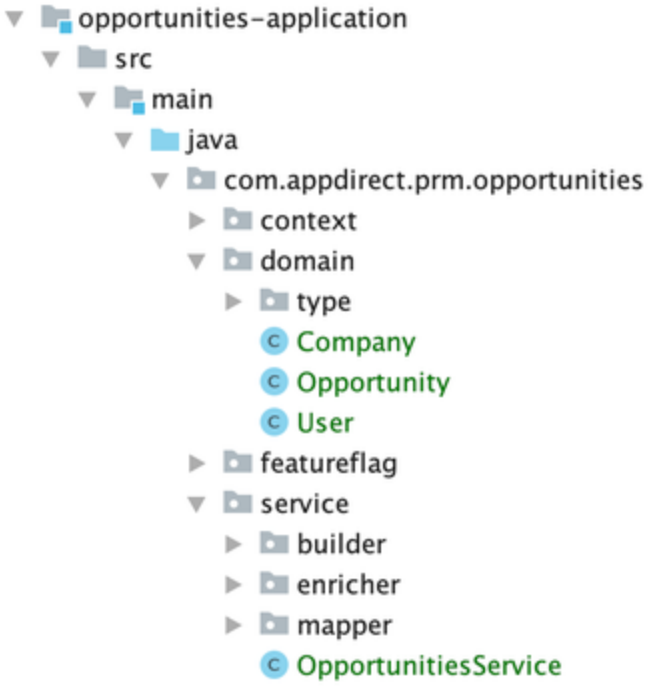
This is how the **module structure** would look like:



These are the **incoming and outgoing port modules**:



This is the **application module** with the **domain models** and **services**:


opportunities-application
src
main
java
com.appdirect.prm.opportunities
context
domain
type
Company
Opportunity
User
featureflag
service
builder
enricher
mapper
OpportunitiesService