Clean Architecture

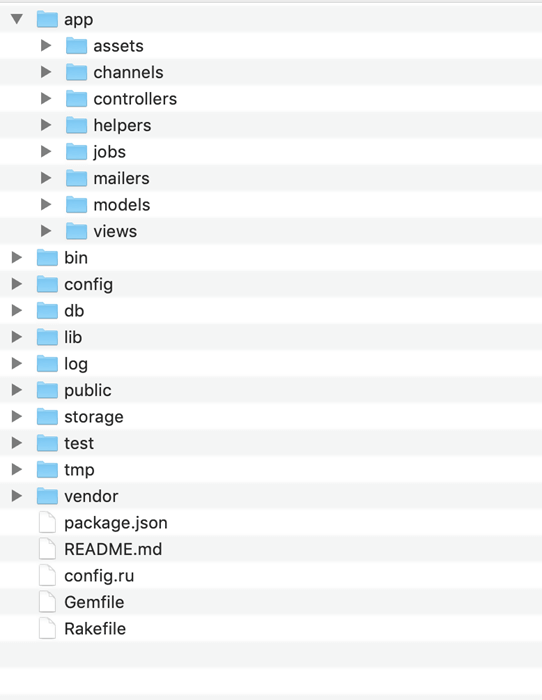
Swift Application

Take a look at this application. The top-level structure has stuff like Controllers, CoreData, Models, Networking, etc… Digging down into it we see objects are being grouped by their place in the technology stack: Menus, Pages, Collections, CoreData, Models, Networking.

But what does this application do?? Where is the business logic? What does a business flow look like?

Ruby on Rails App

Here’s a Rails app. Same thing. Folders for assets, channels, controllers, models, views…



What do these things do? Isn’t the business domain the most important part of your application?

Shouldn’t the business domain be obvious in your code structure?

What if we put the business at the core of our applications? What if you could look at the code and just know where the business use-cases are, and where all the business rules are?

Let me propose an alternate structure:

Let’s create an object for every business use-case.

“A *use case* is a list of actions or event steps typically defining the interactions between a role (known in the [Unified Modeling Language](https://en.wikipedia.org/wiki/Unified_Modeling_Language) (UML) as an [*actor*](https://en.wikipedia.org/wiki/Actor_(UML))) and a system to achieve a goal.” -Wikipedia

In an application that handles invoice and billing, you’d have usecases for Generate Invoice, View Invoice, Make Payment, View Payment History, etc…

Take an example of Make Payment:

The business rules for Make Payment would need to interact with the customer account, a payment system, an invoice with various line items with individual prices, quantities, etc. Each of those objects will have its own business rules that deal with the behavior of that object. For example, a specific customer may have earned or negotiated a discount on certain types of merchandise; the Customer object will implement those rules.

We need those objects to be modeled. Let’s call them Entities.

Those usecases and entities are the heart of our application. They’re the reason the application exists. Wouldn’t it make sense if the rest of the application was in service to those classes? And we want to keep those classes “pure” – no information about the system’s I/O or choice of frameworks should leak into this domain model.

In fact, maybe we should not rely on any frameworks at all for our business domain. This gives the code the most flexibility and portability; it removes any dependencies on stuff we don’t control.

Let’s isolate the business domain behind interfaces. We’ll expose an interface per use-case. On the inside of this interface, we work with usecases and entities; on the outside we work with simple data and abstract methods. Let’s expose each usecase via an interface and create DTOs or VOs or Java 17 records to pass data in and out of that interface.

Let’s take an example of Make Payment:

Input: Invoice Number and Payment Information

Output: Payment Confirmation

Simple scenario: Paid in Full (Credit Card)

1. Customer sends invoice number and payment information
2. Look up invoice
3. Verify payment is equal to invoice amount due
4. Request credit card company to transfer payment amount to our company’s account
5. If successful, mark invoice as paid
6. Return “Success, Paid in Full” to customer

Create Invoice Number, Payment Information, Payment Confirmation and use these as parameters to the Make Payment interface.

Now we have something that we can start to test. The interface helps isolate the code and gives us an invocation point. Since this is the heart of our application, we want full coverage on the business rules.

Looking again at the Make Payment use-case, we see that we are going to need some way to search for and update the invoice and we need a way to connect to an external payment system.

For reading and updating the invoice, that’s probably going to be a database of some kind. But will it be a relational database? Document DB? Graph DB? Object-oriented? A Key-Value store?

“The job of an architect is to defer decisions as long as possible.” Good architecture enables writing most of the code for the application before making a given decision.

“A good architect maximizes the number of decisions not made.”

Let’s not worry about it yet. Let’s create an interface with all the methods we’d want a database to implement:

* Find Invoice by Invoice Number
* Update Invoice
* Find all unpaid invoices for a given customer
* Etc…

We’ll pass needed information to that interface and get back Entity objects in return.

For electronic payment of invoices, we need to call an external system. We use credit card in our scenario, but what about PayPal? Bank transfer? Google Wallet? Apple Pay? We don’t want to hardcode the payment method inside the business logic, because who knows what we will be using in the future? We want some way to keep it generic inside the business logic.

Let’s implement an interface to the payment system. To keep it simple and to minimize dependencies, let’s define it using simple data, just like our use-case interface.

With those interfaces in place, I can completely test the Make Payment use-case even before I implement the code on the other side of those interfaces.

What we’ve done so far is to create our business domain, and we’ve isolated it by defining interfaces to everything outside of the domain. We’ve created what is in effect a plug-in architecture.

The business logic doesn’t care which database you select – that’s just a detail. In fact, we could swap out the database and replace it with a microservice or a Service Oriented Architecture component, and nothing would need to change. The database is a detail. (Contrast to DB-centric architectures.)

Likewise, the payment system might be to a CC transaction system, or PayPal, or the Federal Reserve Bank transaction system – we don’t care.

The use-case interface doesn’t know if it will be triggered by a web page, another microservice, a messaging bus, etc. (The web is a detail.)

The business logic is isolated from those decisions and is unaffected by whatever choices are made if the implementations adhere to the interfaces defined by the business domain. In Hexagonal Architecture terms, those interfaces are ports; the code that implements the port is called an adapter.

So far, we haven’t said a thing about Spring or Spring Boot. That’s intentional. As mentioned previously we don’t want the business domain code to be dependent on any frameworks.

But the business logic is dependent on some external systems – in the Make Payment use-case, it depends on the Invoice entity port and the payment system port.

We have to somehow inject those adapters into the Make Payment use-case, without making it depend on any framework. Let’s use constructor injection, which will make our unit tests work:

Example…

With the code in our use-case complete and tested, let’s turn attention to the component that calls the use-case. This is sometimes called a primary port or a driving port in hexagonal architecture.

We’ll implement this as a Spring Boot REST API …

We can completely test this in isolation …

We need a way to get the Make Payment use-case implementation…