Clean Architecture with Java 11

[Go to the profile of Carl-Philipp Harmant](https://medium.com/@cpharmant?source=post_header_lockup)

[Carl-Philipp Harmant](https://medium.com/@cpharmant)Follow

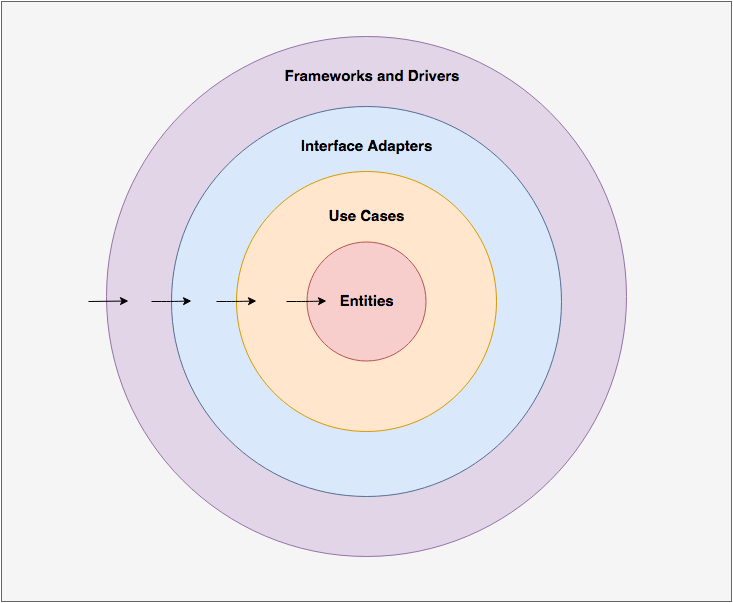
Feb 5

**Introduction**

Software architecture has been one of the most important topics in the last couple years when it comes to software engineering. [Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin) (aka Uncle Bob) deeply developed his vision of a clean architecture in [his book](https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164/ref=sr_1_1?ie=UTF8&qid=1529335011&sr=8-1&keywords=clean+architecture), that I highly recommend. But when it comes to implementation, things get difficult and many questions appear. Where do I start? How do I structure my project? How do I apply Uncle Bob’s principles to the technology I want to use? I will try to answer those questions from the point of view of a Java web developer that would like to use Java 11 and Jigsaw modules from Java 9. This will give you a more concrete vision of Uncle Bob’s clean architecture.

**Clean architecture**

Before diving into the implementation, let’s have a look at the architecture:



* **Entities:** These are the business objects of your application. These should not be affected by any change external to them, and these should be the most stable code within your application. These can be POJOs, objects with methods, or even data structures.
* **Use Cases:** Implement and encapsulate all of the business rules.
* **Interface Adapters:** Convert and present data to the use case and entity layers.
* **Frameworks and Drivers:** Contain any frameworks or tools you need to run your application.

The key concepts here are:

* Any layer can only reference a layer below it and know nothing about what’s going on above.
* The use cases and entities are the heart of your application and should have a minimal set of external library dependencies.

**Implementation**

**Setup of the project**

We are going to use Gradle multi-project and [Java Jigsaw](http://openjdk.java.net/projects/jigsaw/) modules to enforce the dependencies between the different layers.

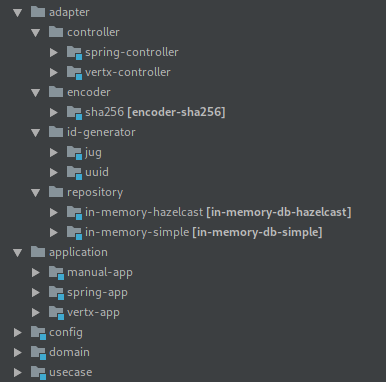
The application we are going to build is very simple and the architecture will probably appear overkill for such a project, but this is the best way to understand how this all works.

The features of the application will be:

* Create a user.
* Find a user.
* List all users.
* Login a user with their password.

To do so, we will start with the inner layers (entities/use cases), then the interface adapters layer, and we will finish with the outer layer. We will also demonstrate the flexibility of the architecture by changing the implementation details and switching between frameworks.

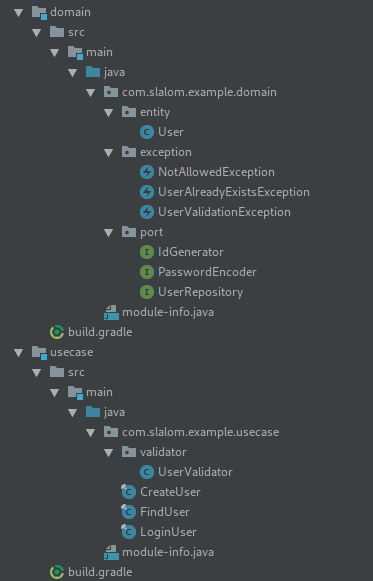
Here is a view of the project:



Let’s dive into the implementation.

**Inner layers**

Our entities and use cases are separated in two sub-projects, ‘domain’ and ‘use-case’:



Domain and Use-case

These two sub-projects represent the heart of our application.

**The architecture must be very explicit.** By having a very quick look at the example above, we know right away what kind of operations exist and where. If you were to create a single UserService instead it would be difficult to tell what kind of operations exist within that service, and you would need to dive into the implementation to understand what the service does. In our clean architecture, we just need to have a quick look at the *usecase* package to understand what kind of operations are supported.

The *entity* package contains all the entities. In our case we are going to have only one, a User:

The *usecase* sub-module contains our business logic. We are going to start with an easy use case, FindUser:

We have two operations, and those two operations need to retrieve users from a repository. This looks pretty standard in a service oriented architecture. UserRepository is an interface that is **NOT implemented within our current sub-project**. This is **considered a detail** in our architecture, and **details are implemented in outer layers**. Its implementation will be provided when the usecase is instantiated (via Dependency Injection for example). This provides some advantages:

* Whatever the implementation is, the business logic remains the same.
* Any change in the implementation does not affect the business logic.
* It is very easy to totally change the implementation because it has no effect on the business logic.

Note that the interface is also known as a **port**,as it makes the bridge between the business logic and the outside world.

Let’s now build a first iteration of our CreateUser use case.

In the same way as the FindUser use-case, we need a repository, a way to generate an ID, and a way to encode a password. These are also details and not business rules, and will be implemented later in outer layers.

We also want to validate that the provided user is valid (contains correct data), and that it does not exist already. This leads to our final iteration of the use-case:

If the user is not valid or exists already, a custom runtime exception is thrown. Those custom exceptions should be handled by other layers.

Our last use-case, LoginUser, is pretty straight forward, and is available in [GitHub](https://github.com/carlphilipp/clean-architecture-example/blob/master/usecase/src/main/java/com/slalom/example/usecase/LoginUser.java).

Finally, to enforce boundaries, both sub-projects use [Jigsaw](http://openjdk.java.net/projects/jigsaw/) modules. Jigsaw modules allow us to expose to the outside world only what needs to be exposed, so no implementation details are leaked. For example, there is no reason to expose the UserValidator class:

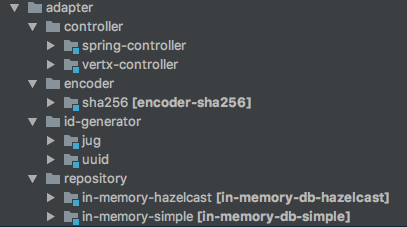
To summarize the role of the inner layers:

* The inner layers contain domain objects and business rules. This should be the most stable and tested part of the application.
* Any interaction with the outside world (like a database, or an external service) is not implemented in the inner layers. We use ports (interfaces) to represent them.
* No framework is used and minimal dependencies.
* Jigsaw modules allow us to hide implementation details.

**Outer layers**

**Adapters**

Now that we have our entities and use cases, we can implement the details. To be able to demonstrate that the architecture is very flexible, we are going to create several implementations and use them in different contexts.



Adapters

Let’s start with the repository.

**Repository:**

UserRepository implementation with a simple HashMap:

Another implementation with Hazelcast can be found on [GitHub](https://github.com/carlphilipp/clean-architecture-example/blob/master/adapter/repository/in-memory-hazelcast/src/main/java/com/slalom/example/db/hazelcast/HazelcastUserRepository.java).

**Other adapters:**

Other adapters are implemented the same way just by implementing the interface declared in the domain. You can find them on [GitGub](https://github.com/carlphilipp/clean-architecture-example/tree/master/adapter):

* [Password encoder](https://github.com/carlphilipp/clean-architecture-example/blob/master/adapter/encoder/sha256/src/main/java/com/slalom/example/encoder/Sha256PasswordEncoder.java)
* [Jug Id Generator](https://github.com/carlphilipp/clean-architecture-example/blob/master/adapter/id-generator/jug/src/main/java/com/slalom/example/jug/JugIdGenerator.java)
* [Uuid Generator](https://github.com/carlphilipp/clean-architecture-example/blob/master/adapter/id-generator/uuid/src/main/java/com/slalom/example/uuid/UuidGenerator.java)

**Putting everything together**

Now that we have our implementation details, we need to assemble them together. To do so, we need to create a config folder that contains the configuration of the app and an application folder that contains code to run the application.

Here is one of the configs:

This config initializes the use cases with relevant adapters. If you wanted to change the implementation you could easily switch from one adapter implementation to another without having to modify the use-case code.

The following is the class that runs the application:

**Web Frameworks**

What if you want to use a web framework like Spring Boot or Vert.x? It’s pretty easy — we just need to:

* Create a new configuration for the web app.
* Create a new application runner.
* Add controllers in the adapter folder. The controller will be responsible for communicating with the inner layers.

Here is what the Spring controller looks like:

You can find the full example of this application in [GitHub](https://github.com/carlphilipp/clean-architecture-example) using both Spring Boot and Vert.x.

**Conclusion**

We tried in this article to show how powerful Uncle Bob’s clean architecture is. Hopefully it’s a little bit clearer for you.

**Pros:**

* **Power:** Your business logic is protected, and nothing from the outside can make it fail. Your code does not depend on any external framework “controlled” by someone else.
* **Flexibility:** Any adapter can be replaced at anytime by any other implementation of your choice. Switching from Spring boot to Vert.x or Dropwizard can be done very quickly.
* **Defer decisions:** What database do I need? What web framework do I need? You can build your business logic without knowing those details.
* **High maintainability:** It’s easy to identify what component fails.
* **Implement faster:** As the architecture separates concerns, you can concentrate on one task at a time and develop faster. This also should reduce the amount of technical debt.
* **Tests:** Unit testing is easier as the dependencies are well-defined, it’s easy to mock or stub.
* **Integration tests:** You can create a specific implementation of any external service you want to hit during your integration tests. For instance, if you do not want to hit a DB hosted in the cloud because you pay per request, just use a in-memory implementation of your adapter.

**Cons:**

* **Learning curve:** At the beginning, the architecture can be overwhelming, especially for junior developers.
* **More** classes, **more** packages, **more** sub-projects. To my knowledge, there is nothing much that can be done about that. As a polygot developer, I would encourage Java developers to explore other languages like Kotlin. Kotlin can, in that case, help a lot in reducing the amount of files created.
* The **complexity** of the project is higher.
* For small projects, this can simply be **over-engineered**.

The project on [GitHub](https://github.com/carlphilipp/clean-architecture-example) provides more details about how to handle the web frameworks. I would encourage you to checkout the code and play with it if you are interested.

Github:

* <https://github.com/carlphilipp/clean-architecture-example>

References:

* <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>
* Uncle Bob’s book: <https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164>