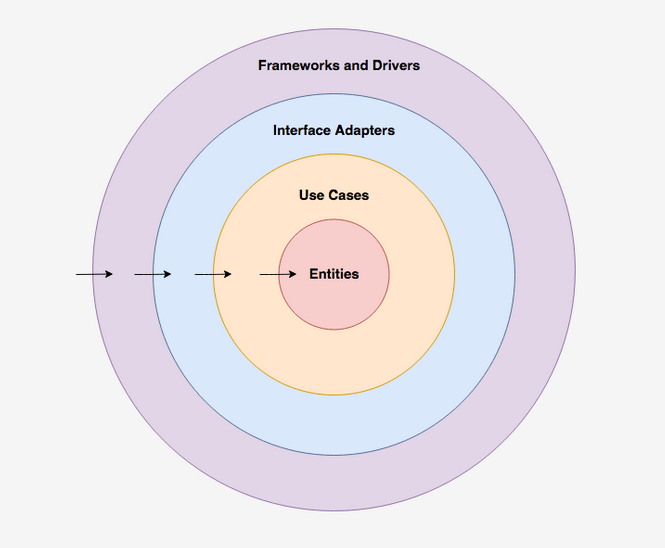
# Project Architecture

First, let’s have a look at the chosen architecture:



1: <https://medium.com/slalom-build/clean-architecture-with-java-11-f78bba431041>

As in the case of the SPA, the chosen architecture is Clean Architecture, which provides a good modularity for the project and it could suit this project to separate all the parts pointed out in the image above. These parts are the following:

* **Framework:** frameworks and tools to make the app run.
* **Interface:** convert data to the use case layer.
* **Use Cases:** Implement and encapsulate business rules.
* **Entities:** Refers to the business objects. These should not be affected by external changes and should be the most stable code. These could be models or POJOs for example.

To bear in mind, as in the SPA project: inner layers should not know nothing about external ones, use case and entities should use the minimal external libraries.

At the use case layer, the architecture should be very explicit, each class should have a name that indicates which specific business rule is being implemented, such as ‘CreateUser’ or ‘FindUser’.

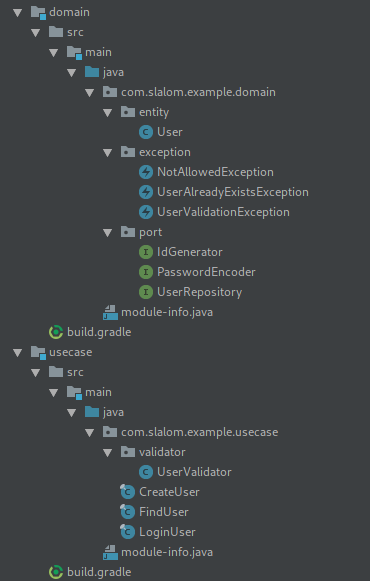
Any interaction in layers with external parts should be handled through interfaces to represent that external interactions.

## Project structure

The project structure will be based in a couple projects got as reference. These projects are: [clean-architercture-example](https://github.com/carlphilipp/clean-architecture-example), by carlphilipp and [spring-clean-architecture](https://github.com/coi-gov-pl/spring-clean-architecture) by cardil.

Basically, we’ll have a folder, or even a sub-project, for each layer, to keep clases structured and independent, enhancing modularity and usability of code.

A good starting point for the API structure would be the following, taken from the Carl-Philipp project and tutorial:



2https://medium.com/slalom-build/clean-architecture-with-java-11-f78bba431041

Doing so, it’s clear that we have a good differentiation between layers, as “domain” and “usecase” can be seen above, and we can improve modularity. In the same way, each one of the other layers should be implemented.

# REST API

API should be coded in a way that the consumer can understand it with out any problems, so, the consumer, the SPA in this case, and its requirements, should go first.

This communication should be established through a contract, or agreement, between consumer and provider. A good tool to check this established contract is **Swagger,** that should be used in this kind of project.

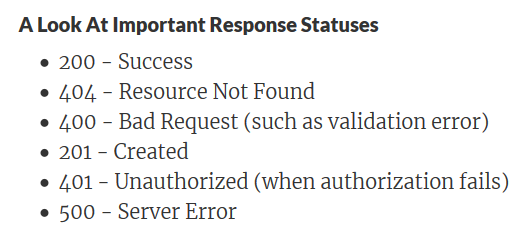
This Contract should be created before the service is implemented. In its first approach for this application, it’s established how the url structure should be created and what kind of response is expected. Later, during implementation, the actual contract should be documented using tools like Swagger.

With this contract first approach, thinking about the implementation details can be avoided so the API becomes much more usable.

To bear in mind when defining uri standards:

YARAS stands for Yet Another RESTful API Standard. YARAS provides standards, guidelines and conventions to be followed while developing RESTful web services

* Use HTTP verbs as they were though, GET to retrieve, POST to create, DELETE to delete and so on.
* Establish representation for request and response bodies. It could be JSON or XML for example.
* Use appropriate response status, bad request, not found, server exception…



3https://www.springboottutorial.com/rest-api-best-practices-with-java-and-spring

* Use plurals for entities
* Have great Documentation, in this project Swagger will be used.
* Support versioning, using different urls (“/**v1**/users”) or using parameters (“/users?version=2”) or with headers.
* Pagination and search should be thought to make it easier to request.

## Defining URL structure

Next, and taking the above introduction, the basic url strutucture will be defined, so it can be checked when developing the API in order to mantian a common structured trough all the endpoints of the API and to make it more readable and usable.

The basic structure will be presented using as example a basic CRUD on the principal entity of the application, the cashFlows.

The main path will be basepath:PORT/api/v1/cashFlows, this path indicates versioning and the entity that we would like to operate with.

The request and response body will be focused on JSON as default, maybe with a expansion to XML in the future to get more usability.

Pagination and search should be requested through parameters, in which it will be indicated the starting element and range, and search parameters, in case any of these is not specified, default values should be included.

For example: “/api/v1/cashFlows?start=10, range=20, amount>20”. To make the filter possible, the characters >, <, = and != will be used to get greater than, less than, equal and not equal respectively.

The main requests wil be:

* Get cashFlows: a GET, with url “basepath:PORT/api/v1/cashFlows” where paginating, search and sorting parameters could be included, or deafault values would be used. The response, if not an error code, will have the list of cashFlows retrieved, and urls for first page, previous page, next page, last page and total number of elements.
* Get cashflow by id: GET, with url “basepath:PORT/api/v1/cashFlows/{cashFlowID}”, with no pagination, search or sorting options. The response, if not an error code, will have a single cashflow.
* Create cashflow: POST, with url “basepath:PORT/api/v1/cashFlows/”, in which body will have a cashflow object. The response, if not an error code, will have the new cashflow created.
* Update cashflow: PUT, with url “basepath:PORT/api/v1/cashFlows/”, in which body will have a cashflow object. The response, if not an error code, will have the updated cashflow.
* Remove cashflow: DELETE, with url “basepath:PORT/api/v1/cashFlows/{cashFlowID}”. The response will have a code with the result of the operation.

# Data Structure

Regarding data structure, just to begin with the application, the cashflows entity will be the first to be created, as it’s intended to be to core of the application, and later, as the application grows, more entities should be added, like users, for example.

All the entities and its relations should be added into the draw.io document, in order to keep a good vison of the project data structure and know how it works in a single sight.

# References

<https://medium.com/slalom-build/clean-architecture-with-java-11-f78bba431041>

<https://www.springboottutorial.com/rest-api-best-practices-with-java-and-spring>

<https://github.com/coi-gov-pl/spring-clean-architecture>

<https://spring.io/guides/tutorials/rest/>