Default architecture

- Documentation information
  - Documentation scope
  - Documentation roadmap
  - How a View is documented
- Top View
- Layers View
  - Domain View
  - Application View
  - Infrastructure View
- Project structure
  - Symfony

Documentation information

Documentation scope

The scope of this documentation is to describe the basic software architecture of Firmaprofesional projects.

Documentation roadmap

This documentation is organised in the following sections:

- Documentation information: gives information about the organisation of this documentation.
- Views: describes every relevant part of the architecture in a format called View.

How a View is documented

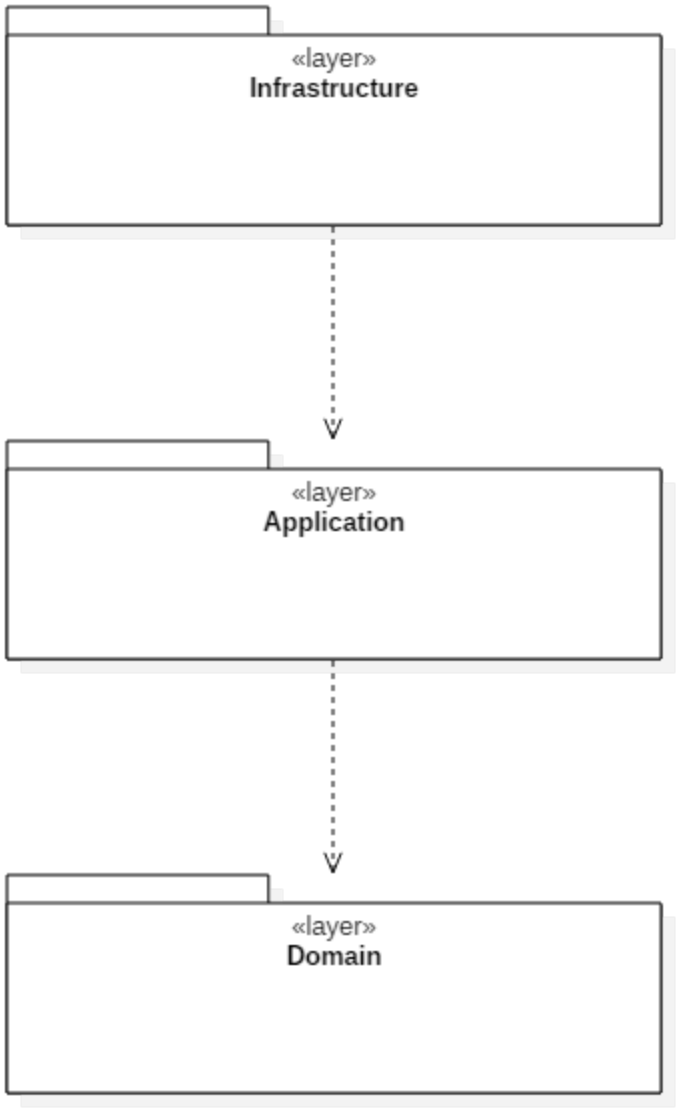
A View represents a group of a system elements and their relationships.

Every View is organised as follows:

1. Primary presentation: main diagram of the view.
2. Catalog of elements: enumeration and description of the elements composing the view.
3. Context diagram: depicts the scope of a view.
4. Rationale: explains the design decisions made in this view.

Top View

Primary presentation



Rationale

The architecture of our projects is based on a layered architecture. A layered architecture divides an application into n-layers.

A layer is characterised by:

- It has a differentiated responsibility and isolates the layers below.
- It has rules about what classes belong to it.
- You can access the lower layers but not the higher ones.

The pattern Ports and Adapters, also known as Hexagonal Architecture, inspires our layers definition. This pattern isolates our code from the surrounding infrastructures: databases, webservices, etc.
This separation creates two layers: Infrastructure and Core.
Also our Core will divided into two layers: Application and Domain. The Domain layer will contain our business logic. And the Application layer will organise the logic of the Domain.

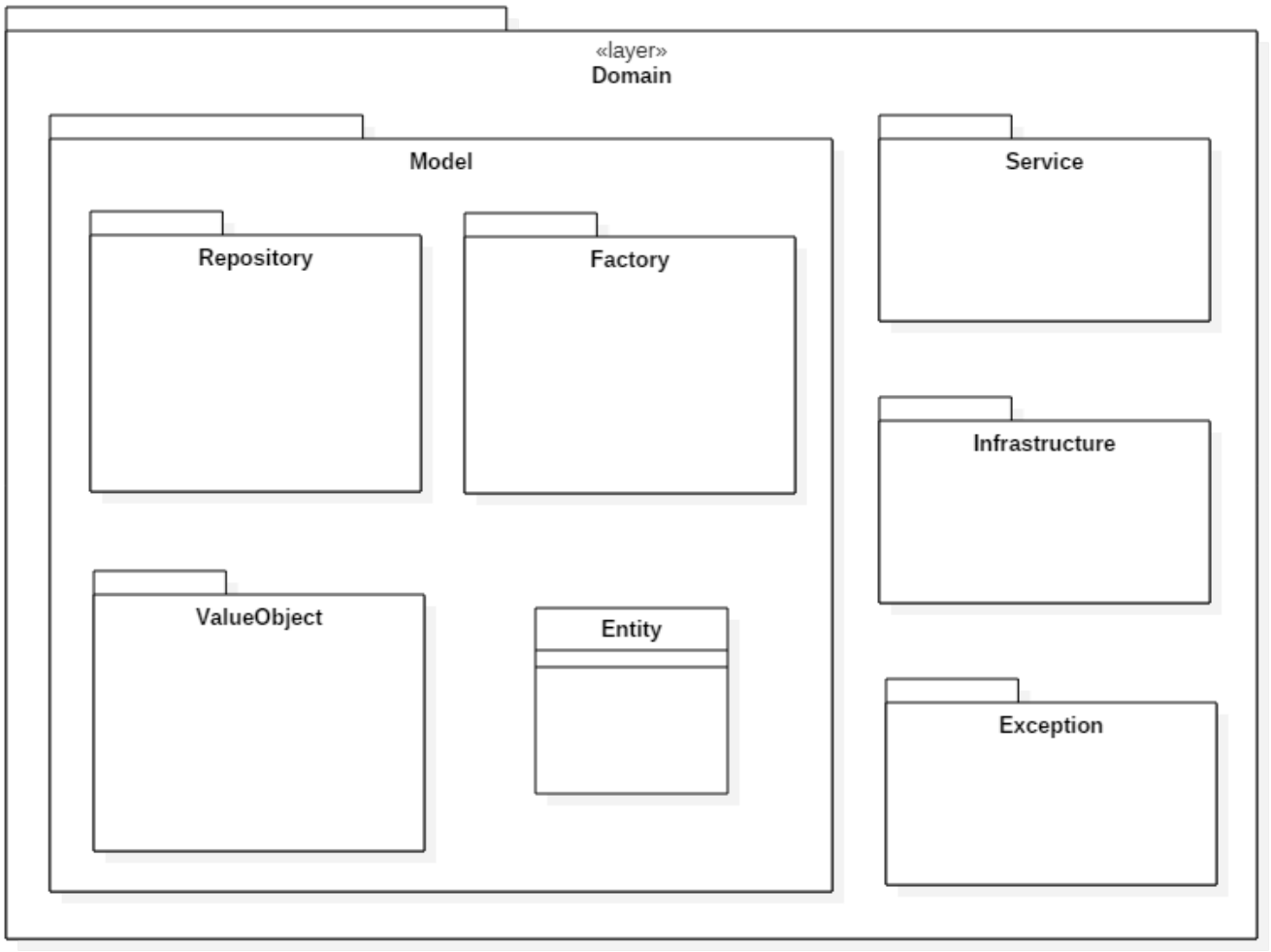
Summarising, the architecture of our projects will be organised in the following layers:

- **Infrastructure**
- **Application**
- **Domain**

Layers View

Domain View

Primary presentation



Element catalog

Model
Package where we define the entities of our application.
An entity represents a concept of our domain. This concept has an individuality and distinguishing it has a benefit for us.
Entities contain the business logic of our application.
Repository
Package where we define the repositories of the domain entities.
A repository represents a collection of entities.
Repositories have methods to add, delete and search.
Repositories return instantiated objects or collections of objects.
Factory
Package where we define the factories of the domain entities.
A factory handles the logic involved in the creation of entities.
ValueObject
Package where we define the Value Objects of the domain.
They are objects which do not have an individuality but are identified by their value. Value Objects measure, quantify or describe.
They are not stored in the database but can be part of an entity.
They are immutable, that is, if any operation is performed on them their value is not modified, but a new instance is returned with the resulting value.
Service
Services contain complex logic that can not defined within entities.
Exception

Exceptions that our application can throw.
Infrastructure
Contains interfaces that the Infrastructure layer will implement.

Rationale
This layer contains the business logic of our application.

Following the design/patterns of DDD this layer will contain:

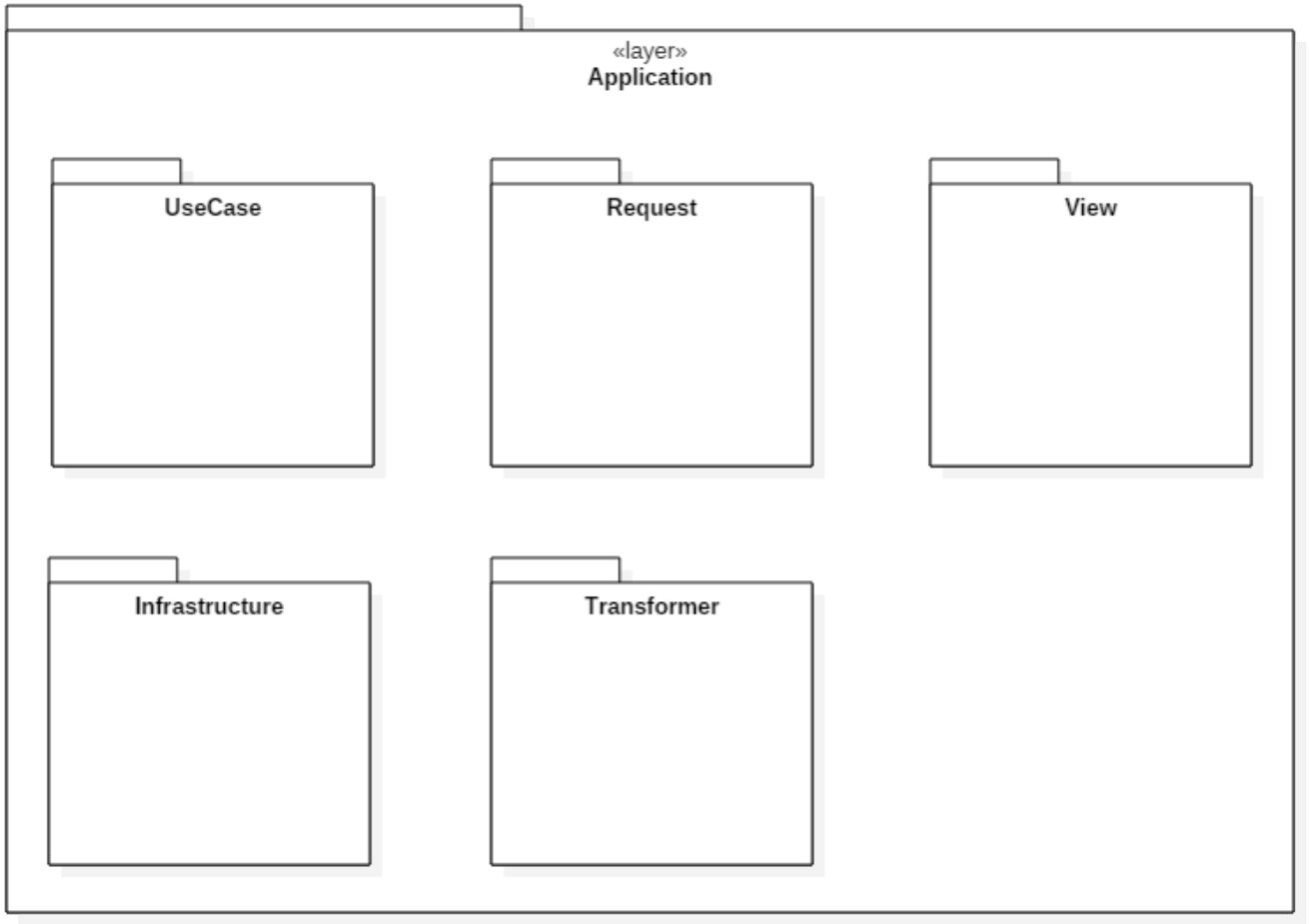
- Entities
- Repositories
- Factories
- Domain services
- Value objects

This layer does not have direct access to the infrastructure. This means that from this layer we can not access the repositories. To solve this we use the dependency injection.
To be able to use a functionality of a superior layer we will define an interface. This interface will have the necessary methods for this layer. Using the necessary technologies, the superior layer will implement the interface.

If our application reaches an erroneous state we will throw an exception. We will create exceptions that will represent the errors that can occur. By using custom exceptions we know which error has occurred and perform a specific task.

Application View

Primary presentation



Element catalog

UseCase

CommandHandlers that deal with the use cases of our application.

View

CommandHandlers that handle the queries of CQS of our application.

Request

Commands that represent the UseCase and the Views.

Infrastructure

Contains interfaces that the Infrastructure layer will implement.

Transformer

Convert a domain entity to a DTO that consumers of the Views will use.

Rationale
The application layer is used so that our business logic does not leak to higher layers. Also to concentrate all the functionalities of our application at one point. It is the point of contact between the outside world and our domain.

This layer uses the command pattern to organise the classes that comprise it. Using the CQS principle we will split them in Queries and Commands. This principle tells us that if a method modifies the state of an object it is a command. And if it returns a value it is a query and it should not change the state of any object. We will call the Commands: UseCases and the Queries: Views

A UseCase does not contain business logic. It is responsible for controlling persistence, security, sending emails, ... It is in charge of obtaining entities from a repository and executing operations on them. A use case must be small, since it only coordinates operations between domain objects. If a use case becomes very large we should create a domain service. A use case can return identifiers of created entities but they must not return entities.

A View does not change the status of the application. It only queries our application and returns some data. A View can not return a Domain Entity. If a View returns an Entity it could be modified outside the Application layer. A View must transform the result of the query to a data transport object(DTO).

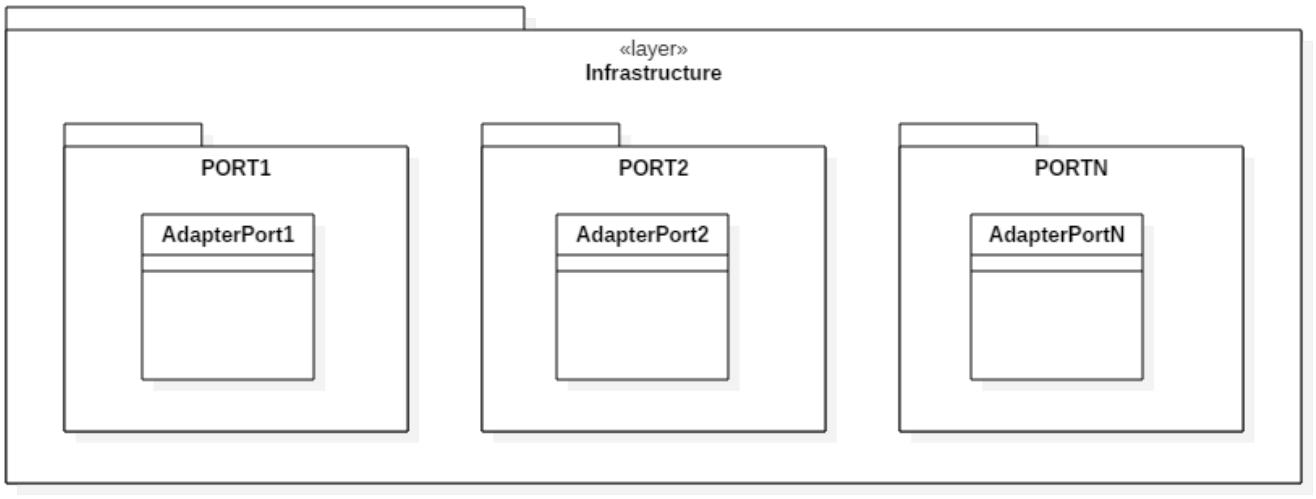
According to the command pattern for each task we will need two classes: a Command and a assigned Handler. A command is a DTO with the necessary data to perform a task. And a Handler is the responsible for executing the corresponding logic with the data. In our design the Commands are the Requests and the Handlers the UseCases or the Views.

With the command pattern we normalise the input parameters. We also avoid duplication of functionalities. When we want to perform a task for which a command already exists, what we will do is adapt our data to the corresponding command.
Views rationale

The aim of a View is to control the access to our data and return them in a format that is useful for the client.
Views will be grouped by a root domain entity.
For each entity we will have as many Views as necessary. The view name will state of what type it is: UsersEnabled, UserOfId, etc.
Views will control that whoever makes the query has permission to do so.
Views will never return a domain entity. If they did, it would be possible to change it from the Infrastructure layer, without going through the Application layer.
To return the data we will use a Transformer. These are responsible for convert the data of the queries in the format that the client needs. This way we do not link the structure of our domain with our UI.
Views will use the ViewsRepository. These will contain queries that will not use the automatic creation of objects of the ORM. They will only return simple types arrays.
If any query involves more than one entity, it will be placed in the repository of the root entity.

Infrastructure View

Primary presentation



Element catalog

Port

It represents a communication point of our application with the outside.

Adapter

Code that adapts the functionality of the port to our application.

Rationale

This layer is based on the Ports and Adapters pattern. The aim of this pattern is to isolate our application from outside communications. For outside communications we mean: databases, external devices, webservices, etc.

Each communication point is a Port and the specific code that allows us to use it is an Adapter.

An Adapter can be a single class, a set of them or even a framework. The same Port can have several Adapters depending to our need.

Project structure

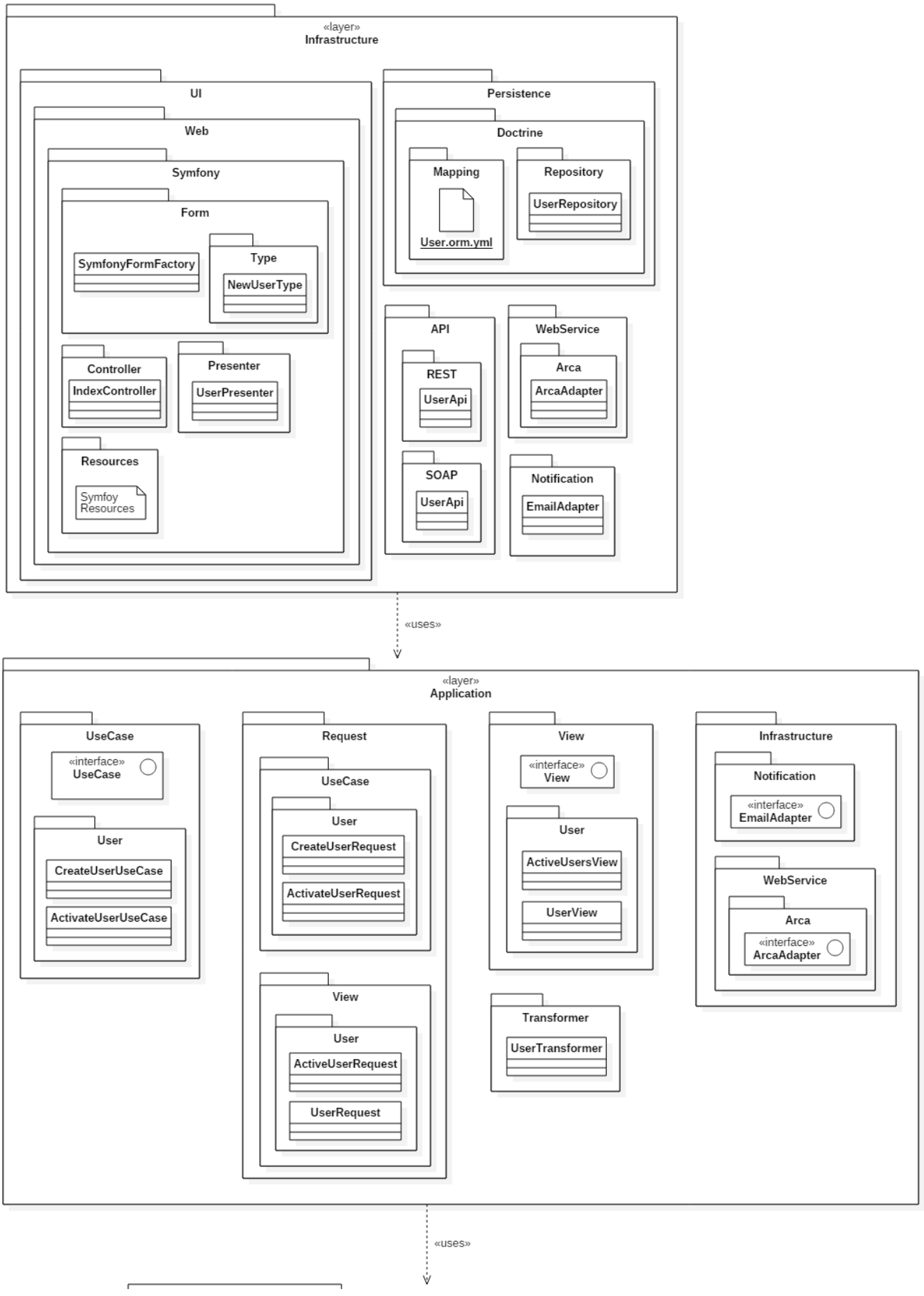
Applying the previous points to a file structure, we would have something like this:

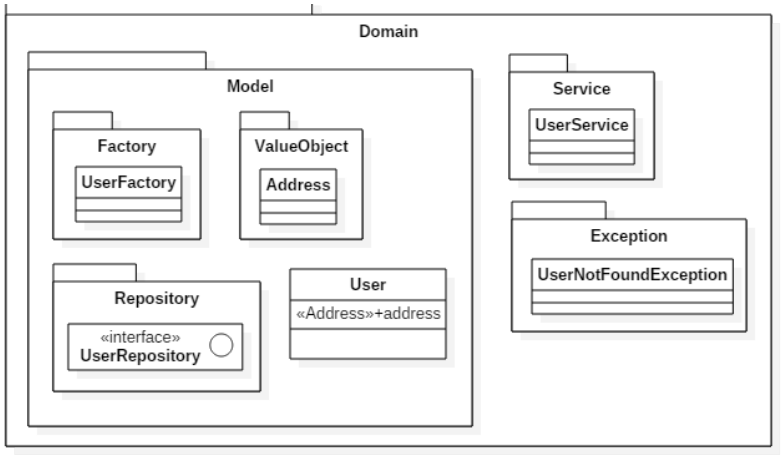
```
src/
    FP/<Project>/
                Domain/
                        Model/
                                Factory/
                                Repository/
                                ValueObject/
                        Service/
                        Exception/
                        Infrastructure/
                                <Port>/
                                        <AdapterInterface>
                                ...
                Application/
                        Usecase/
                        View/
                        Request/
                        Infrastructure/
                                <Port>/
                                        <AdapterInterface>
                                ...
        Infrastructure/
            <Port>/
                <Adapter>/
                <Adapter>/
                ...
            <Port>/
                <Adapter>/
                <Adapter>/
                ...
            ...
```

Symfony

Most of our projects use the Symfony framework. In this section we will see how a Symfony project fits in this design.

Primary presentation


«layer»
Infrastructure
UI
Web
Symfony
Form
SymfonyFormFactory
Type
NewUserType
Controller
IndexController
Presenter
UserPresenter
Resources
Symfoy Resources
Persistence
Doctrine
Mapping
User.orm.yml
Repository
UserRepository
API
REST
UserApi
SOAP
UserApi
WebService
Arca
ArcaAdapter
Notification
EmailAdapter
«uses»
«layer»
Application
UseCase
«interface»
UseCase
User
CreateUserUseCase
ActivateUserUseCase
Request
UseCase
User
CreateUserRequest
ActivateUserRequest
View
User
ActiveUserRequest
UserRequest
View
«interface»
View
User
ActiveUsersView
UserView
Transformer
UserTransformer
Infrastructure
Notification
«interface»
EmailAdapter
WebService
Arca
«interface»
ArcaAdapter
«uses»



Element catalog

Infrastructure/UI/Web/Symfony

Here we will place the Symfony bundle code: Controllers, Forms, configs, etc.
Symfony app/config must be adapted to point to this directory.

Infrastructure/UI/Web/Symfony/Presenter

Presenters are responsible for adapting Views responses to the most convenient format for the UI. The goal of Presenters is to simplify the templates and controllers.

When a controller wants to visualise some data it will call the corresponding View to get them. To adapt the results of the View to the response that the client needs we use a Transformer. We will inject it into the Views to adapt to different clients/UI.
If in a template we need to do some logic depending on the Views data we will use a Presenter.

Controllers and Presenters must not contain business logic, e.g.: Does this user have access to this entity? They should only contain logic to adapt the response of the Views. This kind of logic must be inside the View.

If necessary, a Presenter can call the View to simplify the controller.

Forms

To create a form we will use a factory: SymfonyFormFactory. This factory will contain all the necessary logic to create each of the forms that we need.
Every Form Type will be inside the Type/ folder and will represent only one form. Symfony best practices will be followed for the creation of FormTypes.

Infrastructure/Persistence/Doctrine

Here we will have all the necessary configuration to be able to link our entities with the database.
In Mapping/ we will have the orm mapping of our entities to tables in the database.
In Repository/ we will have the implementation of the repositories.

API and WebService

These two ports are very similar since in the end both are webservices. The difference is that API is a primary port and WebService is a secondary port.
A primary port tells our application what to do. Translates what comes from outside to our application commands.
A secondary port is used by our application. It implements an interface defined by another layer.

Project structure

The structure of a project would be:

```
app/
src/
    FP/<Project>/
                Domain/
                        Model/
                                Factory/
                                        UserFactory.php
                                Repository/
                                        UserRepositoryInterface.php
                                ValueObject/
                                        Address.php
                                User.php
                        Service/
                                UserService.php
                        Exception/
                                UserNotFound.php
                Application/
                        Usecase/
                                User/
                                        CreateUserUseCase.php
                                        ActivateUserUseCase.php
                                UseCaseInterface.php
                        View/
                                User/
                                        ActiveUsersView.php
                                        UserView.php
                                ViewInterface.php
                        Request/
                                UseCase/
                                        User/
                                                CreateUserRequest.php
                                                ActivateUserRequest.php
                                View/
                                        User/
                                                ActiveUserRequest.php
                                                UserRequest.php
                        Transformer/
                                UserTransformer.php
                        Infrastructure/
                                Notification/
                                        EmailAdapterInterface.php
                                WebService/
                                        Arca/
                                                ArcaAdapterInterface.
php
        Infrastructure/
            Notification/
                                EmailAdapter.php
                        Persistence/
                                Doctrine/
```

```
                                        Mapping/
                                                User.orm.yml
                                        Repository/
                                                UserRepository.yml
                        UI/
                                Web/
                                        Symfony/
                                                Controller/
                                                        IndexController.
php
                                                Presenter/
                                                        UserPresenter.
php
                                                Form/
                                                        Type/

NewUserType.php

SymfonyFormFactory.php
                                                Resources/
                                                        config/

services.yml
                                                        translations/

messages.es.yml
                                                        views/
                                                                index.
html.twig
                        API/
                                REST/
                                        Presenter/
                                                UserPresenter.php
                                        UserApi.php
                        WebService/
                                Arca/
                                        ArcaAdapter.php
vendor/
web/
```