# System Development

## System Design

### System Architecture

Pick the best architecture that would fit our needs, was a pretty long process, due to the fact that the requirements from the company were not quite clear from the begging and they changed several times, during the development process.

We decided to represent them through Umlet diagrams, as they would be seen in the Solution.

Figure 1, represents our very first drawn system architecture, and you can see that we were planning on a server-client type of system, due to the fact that Means wanted a system that did not require for them to constantly install/uninstall software on their employees computers.

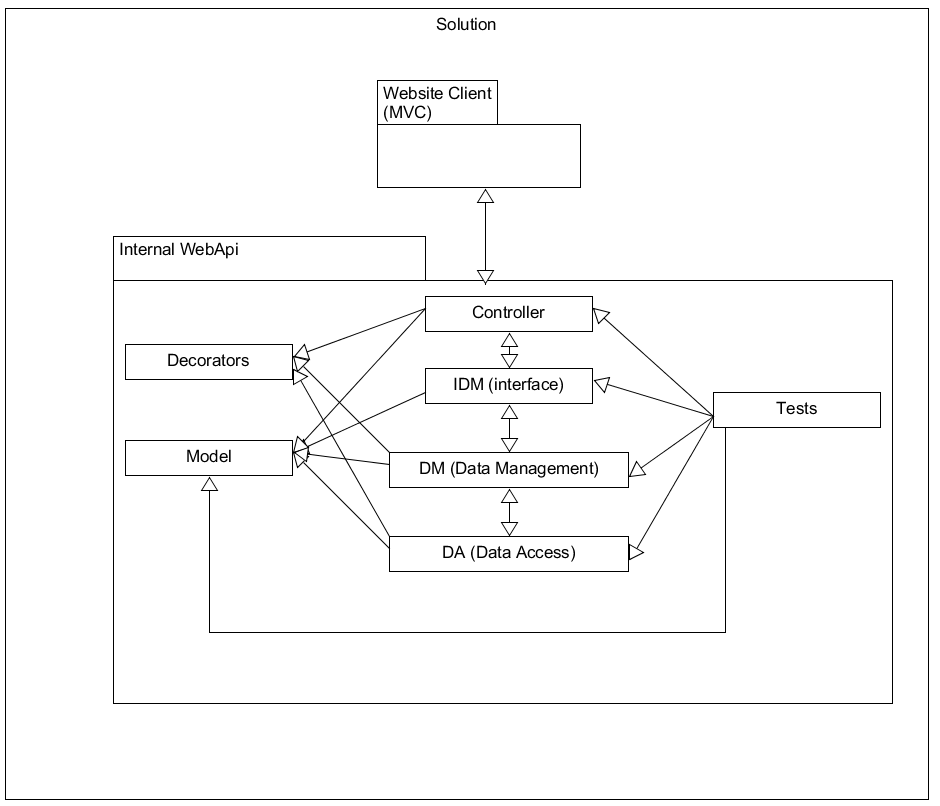


Figure 1

We decided that, in the begging we would have only a web client, built following the MVC framework.

As for the Server, we wanted to make a WebApi, composed of several parts:

* Controllers(aka. Endpoints): which would deal with httpRequests/Responses, serializing and deserializing and other communication parts;
* DataManagement (and the affiliated interface): meant to deal with most of the logic of for this solution;
* DataAccess: meant to deal with the communication to the database;
* Model: meant to hold all of the objects that the system will work with;
* Decorators: meant to decorate actions/classes. One good example being the Logging decorator, which logs all of the actions that a user does, helping in the debugging process, once the solution is deployed;
* Tests: meant to hold all of the test classes we will create;

Figure 2 represents version 0.2 of our architecture. The difference from the previous being the new Project called “External Api”, which got added to our architecture, as we thought of several possible threats and problems that come with the creation of a software solution that uses the internet as part of its base logic.

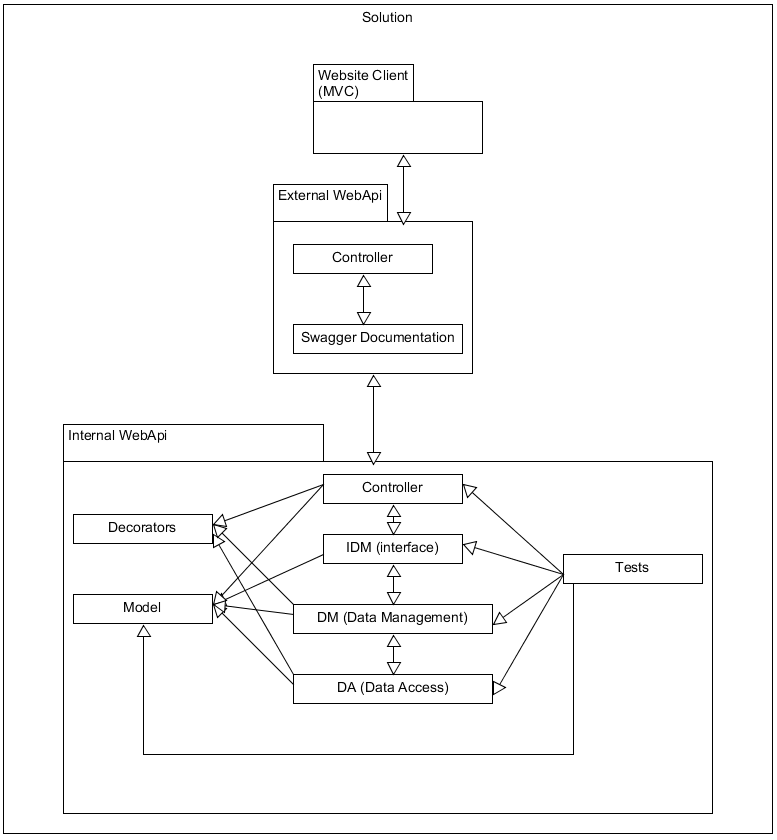


Figure 2

External Api, was our response to the problem of future development, for other programming teams. This new project, would help with that, by adding all the required Documentation, that any WebApi should have. For the documentation, we decided to use Swagger, as it uses the scaffolded XML files, and makes a website that fancily displays our webapi’s endpoints, what their parameters and their possible response messages.

Another function that was attributed to the External Api, was security, by acting as a Proxy, between the clients and the actual server. Providing us just another layer of anonymity, from possible directed attacks.

It also allowed Means to switch internet providers or hosting services providers, without having to notify the clients of the changes, as they would always be connected to the proxy, which redirects their calls to the server, instead of directly being connected to the internal webapi.

We soon realised that we could reduce the code duplication, improve building times and overall performance, while also increasing code’s cohesion, by simply moving around some of the already existing parts, into dedicated projects. This gave birth to our 3rd and “final” architecture, (represented in Figure 3) or so we thought.

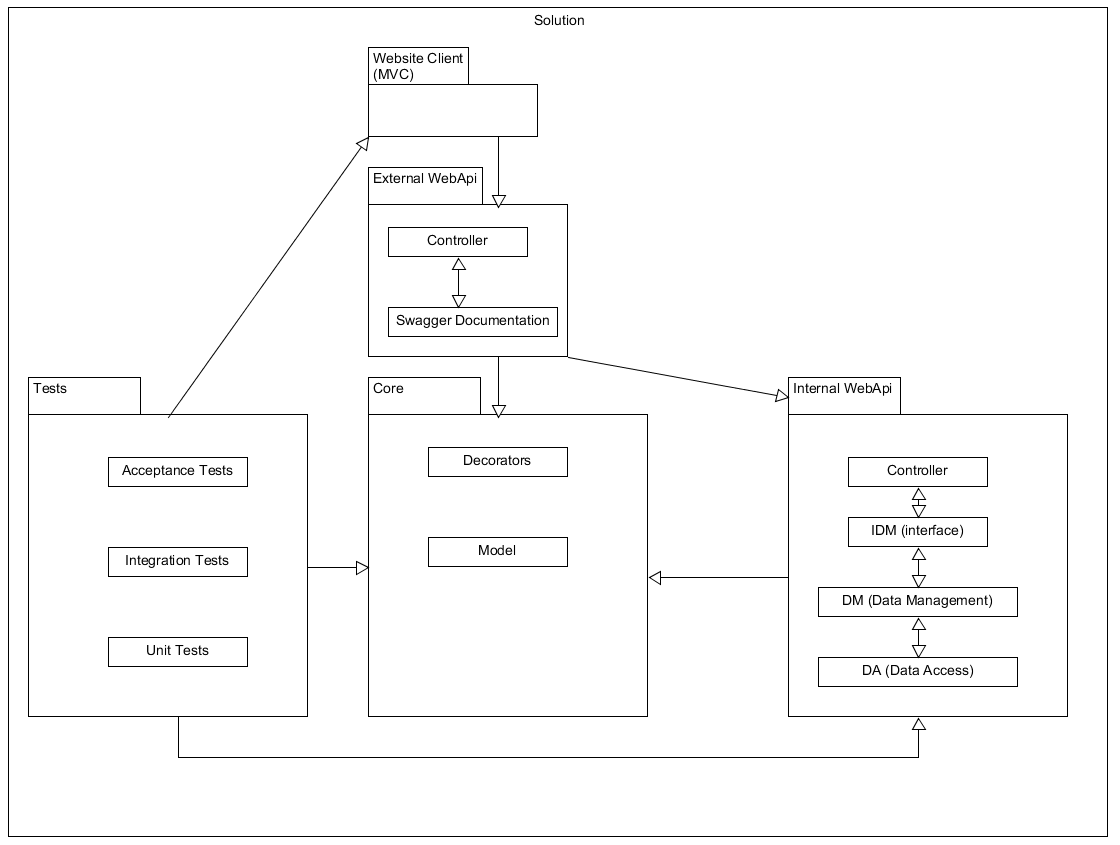


Figure 3

We decided to move Models and Decorators into a special Project called “Core” as it would hold information shared by all other projects. This would also allow us to simply change a Core class, without having to rebuild the other projects, as long as it did not affect their functionality.

Tests also got a dedicated project, because we don’t want to rebuild all the tests, every time we make a small change in the rest of our solution. Not only that, but tests became a bit fluffier, with the addition of Unit Tests (which would test individual methods), Integration Tests (which would test Internal Api’s complex functionality), and Acceptance Tests (which would simulate a user’s interaction with the API, through the MVC client, essentially testing main and most common use cases).

After a good while, and well into the development process, we got new requirements and information from Means, in which they said, among others, that our solution, will not be hosted on the internet, but rather on a local computer/server and will be accessible only from LAN.

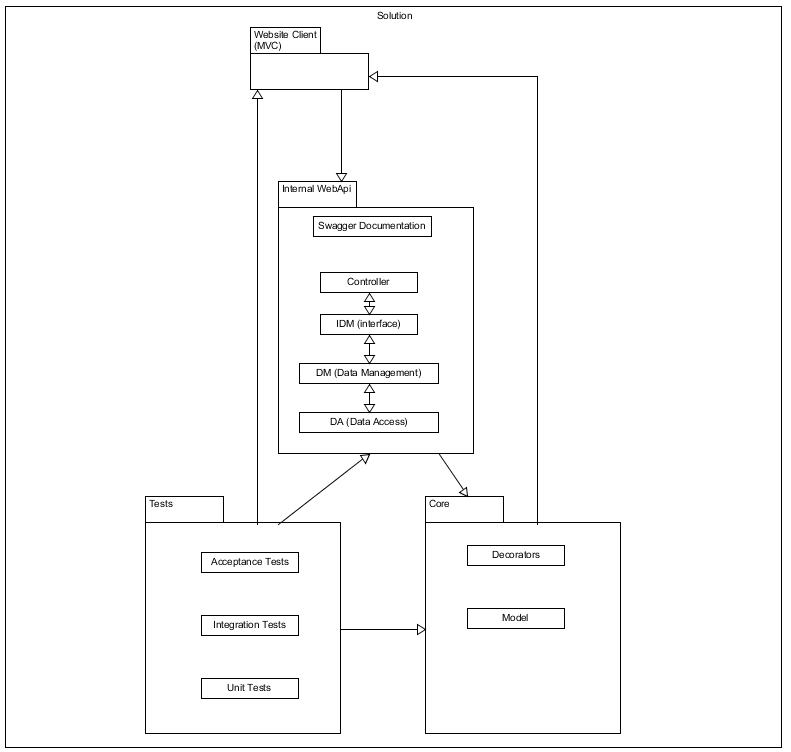


Figure 4

This new information determined us to remove the External API, as it would no longer bring the benefits of having a proxy in the client-server relationship, while bringing all of the downsides of having one. Downsides such as: slightly slower responses, and harder to develop new/edit endpoints.

But we decided to keep the Swagger documentation for possible future dev teams that might work on this solution. And we decided to move it to the Internal Api project.