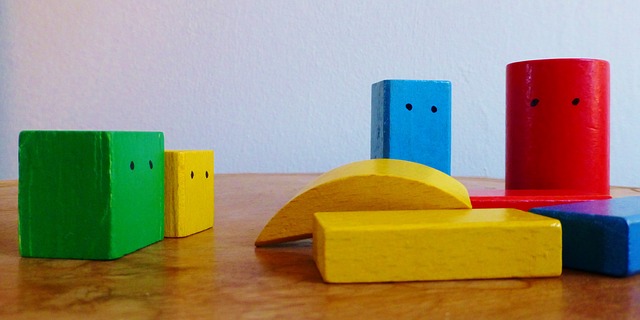
Clean Architecture design in NodeJS

[13 July, 2017](https://solidgeargroup.com/clean-architecture-in-nodejs) by [Siro Ramírez](https://solidgeargroup.com/author/siro-ramirez)

[NodeJS](https://nodejs.org/es/) is one of the core technologies that we use in SolidGEAR to bring our applications to life. But as almost every technology available out there, the most critical part is not about the technology itself but how we use it to accomplish our goals. Therefore, this post tries to **gather all the experience** earned by us working with NodeJS to **define a good and tested architecture**, based mostly in concepts like ‘**Clean Architecture**‘ and ‘**Hexagonal Architecture**‘, so that you’ll be able to follow it and design your backend in a way that it can adapt to most projects and scale in an appropriate way.



This is not the ultimate guide to design foolproof applications. There’s surely a lot of improvements that can be applied, specially regarding to the needs of each project. But you can take advantage of our previous work and several years of testing this architecture to avoid some problems and headaches while being in the middle of a project that has evolved in an unexpected way for you.

Hidden problems behind some NodeJS architectures

Let’s start designing a NodeJS architecture on our own. Most concepts can be extrapolated to other backend technologies, but let’s focus on this one. We’ll choose [Express](https://expressjs.com/es/) as the API framework (based on REST) and [MongoDB](https://www.mongodb.com/es) (using Mongoose) for the database, because **MEAN stack** has become widely mainstream for the last few years. Nevertheless, concepts addressed in this section are transparent to the technology stack chosen.

An example of a basic NodeJS server can be seen in this link, which basically has a resource to allow us to create a new ‘user’:

<https://github.com/siro47/CornerCleanArchitectureNode/tree/v1>

This server has several good ideas behind. It has **separated layers** for handling API requests and db queries. Separated files for modules of REST resources and also for db modules (although just a single module of users management has been added for this example). It looks like it can escalate smoothly as the quantity and complexity of resources increase, but that’s not quite true.



Let’s assume that we have to add a new resource in our server to create a group. A group is designed to contain users. But our business logic determines that a group should create a default user that will be in charge of managing the group. First steps are quite easy to implement, as we can see here:

<https://github.com/siro47/CornerCleanArchitectureNode/tree/v2>

But how are we going to **reuse our previous code** for creating the user?

Importing the method in ‘routes’ layer is annoying, because we need to start passing express linked objects (‘req’, ‘res’,…) from one place to another. Importing the method in ‘db’ layer is inaccurate because we’re missing strategically added functionality in the ‘routes’ layer (setting the completeName of the user).



Moreover, let’s consider that we are going to add some [**unit tests**](https://solidgeargroup.com/mocha-javascript-unit-testing) (because of course, we want unit tests in our code) to our current functionality. Tests for our ‘routes’ domain wouldn’t be a bit express-linked? If we move to a, for example, [GraphQL](http://graphql.org/learn/)API, will we have to rewrite them all?

How would you solve these problems? Splitting the current methods? Creating a new layer? Maybe creating several new layers?

Do not reinvent the wheel. Clean Architecture + Hexagonal Architecture

We are not unveiling a trailblazer solution. We have just evaluated the **best current architecture models** available right now and adapted them to our needs.

So we started analysing one of the most known architecture pattern, the one proposed by [**Uncle Bob**](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html) some years ago, called **Clean Architecture**. It supports the idea of making the **model independent from the framework**, libraries, dbs… (seems logic, doesn’t it?), by creating an intermediate layer called ‘**Interface adapters**‘. He also remarks the necessity of the code to be easily testable, and how this kind of architecture will allow us to create unit tests of the code not linked to any external technological element.

Additionally, we analysed another great and widely used architectural model, the [**Hexagonal Architecture**](http://alistair.cockburn.us/Hexagonal+architecture) proposed by **Alistair Cockburn** in 2005. Different geometric shapes, but very similar concepts. Enclose your app functionality to be used for any outsider through a system of ports and adapters.



So following those rules we **split all the business logic from our API framework**, creating a new ‘domain’ layer:

<https://github.com/siro47/CornerCleanArchitectureNode/tree/v3>

Achieving this way:

1. Our domain code is reusable and independent from our API framework
2. The code is also easily testable and tests can survive through frameworks and libraries changes.

And we can see in the final example version how the core functionality remains unchanged, although we have added a new GraphQL based API that makes use of the same domain methods. On top of that, we have created some unit tests that assure the correct functioning of our business logic autonomously, whatever the API framework is used.

https://github.com/siro47/CornerCleanArchitectureNode/tree/v4

Further improvements

[](https://nodejs.org/es/)

As we have stated previously, this is just a **base architecture** on top of which you should **adapt your project’s needs**. Also, as we don’t want this post to be humongous, we are not going to detail any of the followings improvements, but it’s the duty of the advanced reader to take this issues also in consideration. (And you can ask us for more info in the comments section below if you want)

* Improve **API adapters** definition: Our REST API adapter unties our business logic from resources, requests, responses, and other REST stuff. But a more complete layer should include http codes mapping, i18n messages, and everything we need to exploit the full potential of the API protocol
* **Add adapter layer to db**: In the same way we have split our API technology from our code, we should do the same with the db layer to make our application utterly independent of the db technology we are using. Likewise, we should add an adapter for every external framework we want to isolate from our domain code.
* Define domain entities: Some **Clean Architecture** models talk about defining the **core entities** in our business logic and abstract them into a inner layer. That may depend on your application, but it’s something you should definitely consider if you are going to have some well defined entities along your application logic.

Ready for facing the challenge of having a scalable NodeJS server prepared to adapt every situation?

There’s something else you think that should be mandatory in a good NodeJS architecture?