
A basic Hapi.js API following Clean Architecture principles

Getting started (< 2mn)

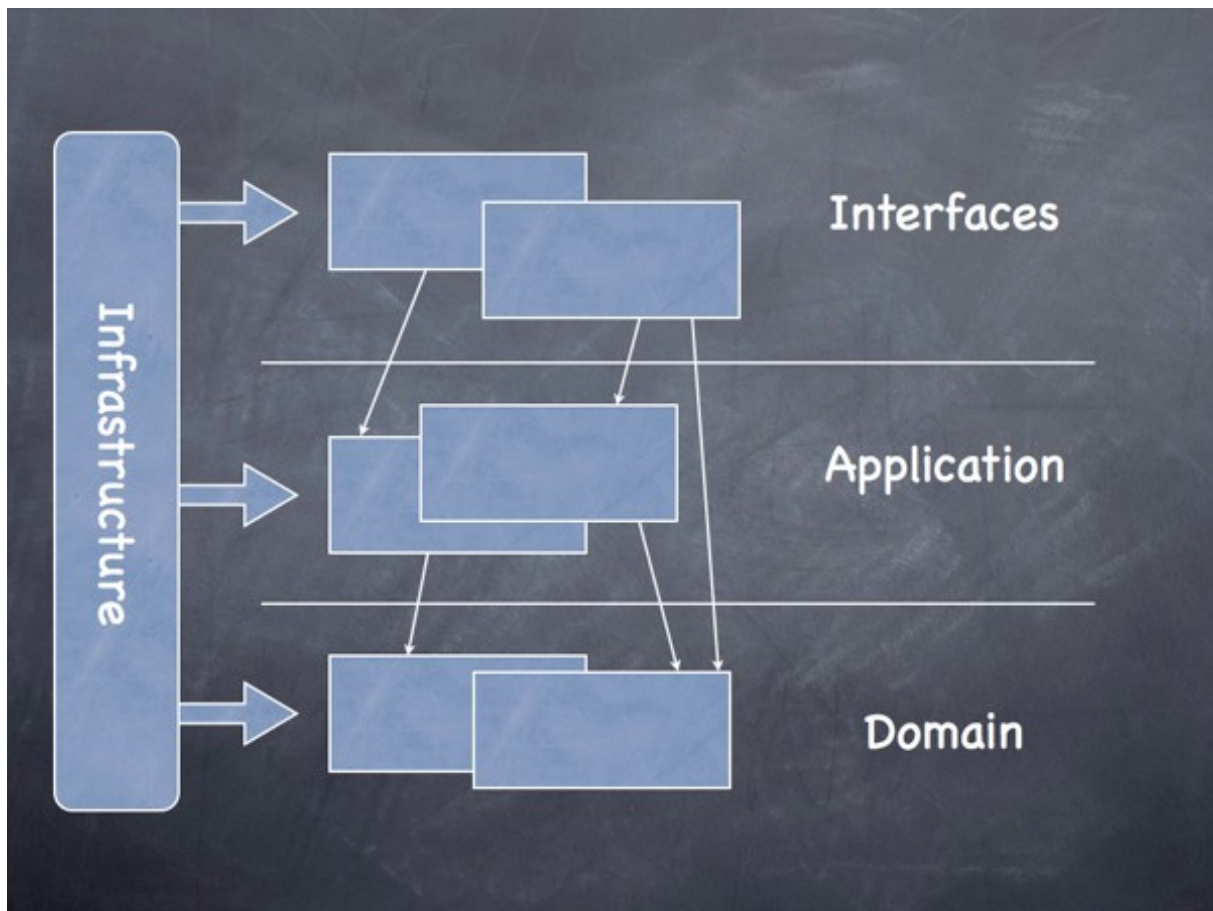
```
1 git clone git@github.com:jbuget/nodejs-clean-architecture-app.git
2 cd nodejs-clean-architecture-app
3 npm install
4 npm test
5 npm start
```

In a browser, open <http://localhost:3000/hello>.

Domain Driven Architectures

Software design is a very hard thing. From years, a trend has appeared to put the business logic, a.k.a. the (Business) Domain, and with it the User, in the heart of the overall system. Based on this concept, different architectural patterns was imagined.

One of the first and main ones was introduced by E. Evans in its Domain Driven Design approach.



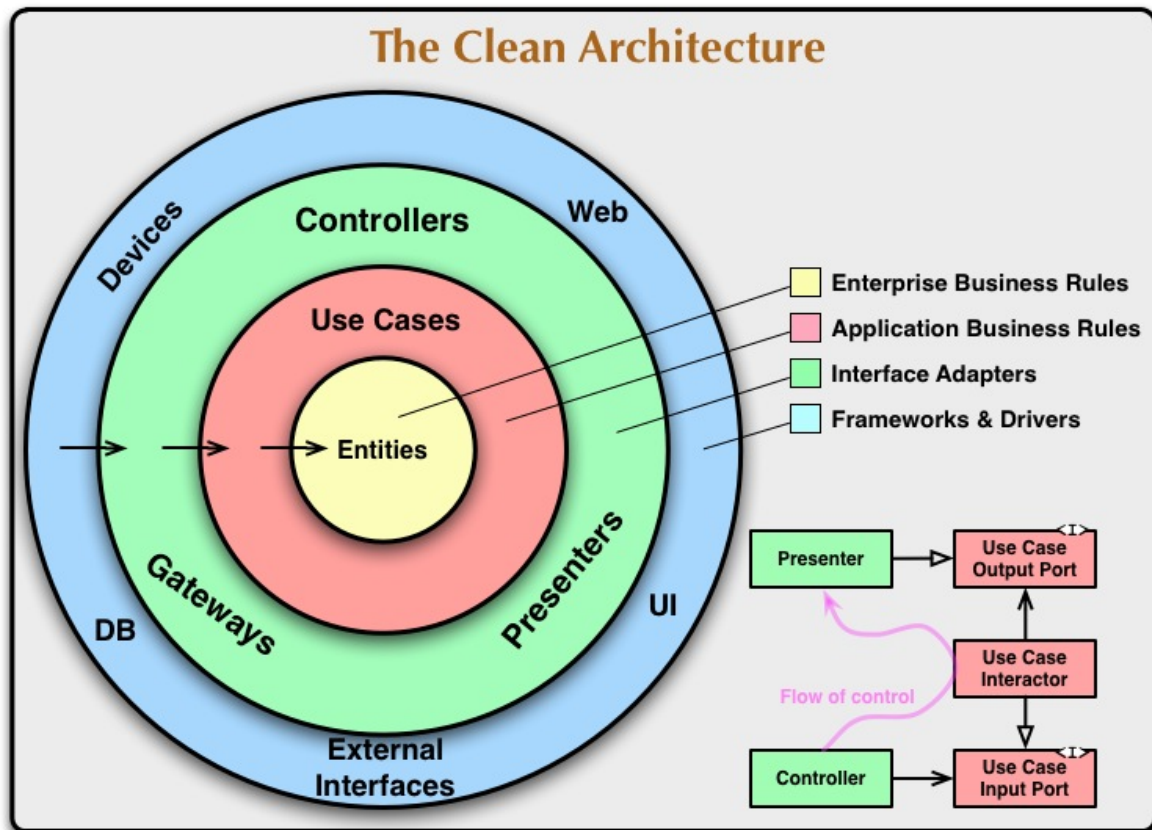
Based on it or in the same time, other applicative architectures appeared like Onion Architecture (by J. Palermo), Hexagonal Architecture (by A. Cockburn) or Clean Architecture (by R. Martin).

This repository is an exploration of this type of architecture, mainly based on DDD and Clean Architecture, on a concrete and modern JavaScript application.

DDD and Clean Architecture

The application follows the Uncle Bob “Clean Architecture” principles and project structure :

Clean Architecture layers

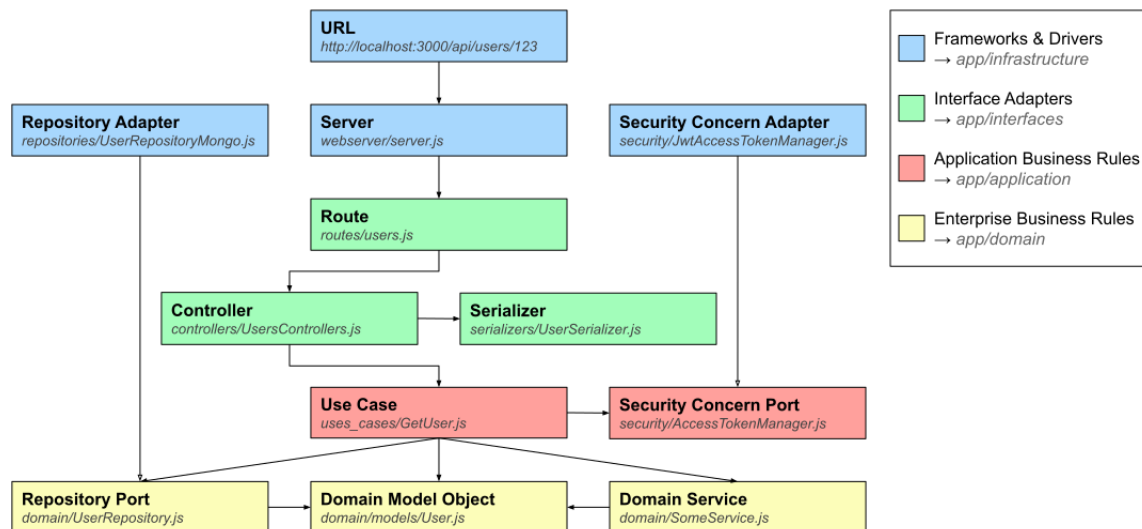


Project anatomy

1	app ^L	
2	lib →	Application sources ^L
3	application →	Application services layer ^L
4	security →	Security tools interfaces (ex: AccessTokenManager.js, to generate and decode OAuth access token) ^L
5	use_cases →	Application business rules ^L
6	domain →	Enterprise core business layer such as domain model objects (Aggregates, Entities, Value Objects) and repository interfaces ^L
7	infrastructure →	Frameworks, drivers and tools such as Database, the Web Framework, mailing/logging/glue code etc. ^L
8	config →	Application configuration files, modules and services ^L

9	service-locator.js →	Module that manage service implementations by environment ^L
10	orm →	Database ORMs middleware (Sequelize for SQLite3 or PostgreSQL, Mongoose for MongoDB) ^L
11	mongoose →	Mongoose client and schemas ^L
12	sequelize →	Sequelize client and models ^L
13	repositories →	Implementation of domain repository interfaces ^L
14	security →	Security tools implementations (ex : JwtAccessTokenManager) ^L
15	webserver →	Hapi.js Web server configuration (server, routes, plugins, etc.) ^L
16	oauth →	Hapi.js authentication strategies and schemas ^L
17	server.js →	Hapi.js server definition ^L
18	interfaces →	Adapters and formatters for use cases and entities to external agency such as Database or the Web ^L
19	controllers →	Hapi.js route handlers ^L
20	routes →	Hapi.js route definitions ^L
21	serializers →	Converter objects that transform outside objects (ex: HTTP request payload) to inside objects (ex: Use Case request object) ^L
22	node_modules (generated) →	NPM dependencies ^L
23	test →	Source folder for unit or functional tests ^L
24	index.js →	Main application entry point

Flow of Control



The Dependency Rule

The overriding rule that makes this architecture work is The Dependency Rule. This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes, variables, or any other named software entity.

src. <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html#the-dependency-rule>

Server, Routes and Plugins

Server, routes and plugins can be considered as “plumbing-code” that exposes the API to the external world, via an instance of Hapi.js server.

The role of the server is to intercept the HTTP request and match the corresponding route.

Routes are configuration objects whose responsibilities are to check the request format and params, and then to call the good controller (with the received request). They are registered as Plugins.

Plugins are configuration object that package an assembly of features (ex: authentication & security concerns, routes, pre-handlers, etc.) and are registered at the server startup.

Controllers (a.k.a Route Handlers)

Controllers are the entry points to the application context.

They have 3 main responsibilities :

1. Extract the parameters (query or body) from the request
2. Call the good Use Case (application layer)
3. Return an HTTP response (with status code and serialized data)

Use Cases

A use case is a business logic unit.

It is a class that must have an `execute` method which will be called by controllers.

It may have a constructor to define its dependencies (concrete implementations - a.k.a. *adapters* - of the *port* objects) or its execution context.

Be careful! A use case must have only one precise business responsibility!

A use case can call objects in the same layer (such as data repositories) or in the domain layer.