

TRABAJO PRÁCTICO N° 2
ARQUITECTURA CLEAN

Integrantes:

Cánepa Enzo

Jones Martin

Monzón Job

Tomasella Tobias

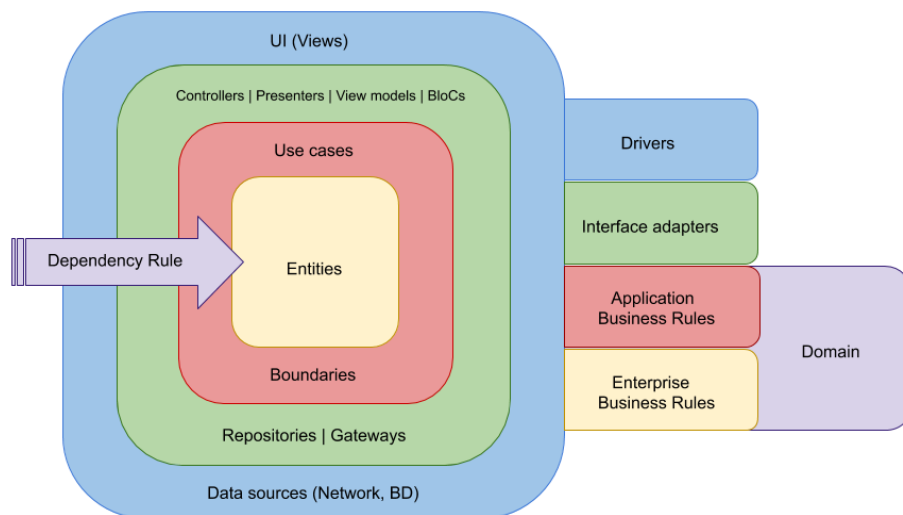
Profesor/a: Jose A. Fernandez

Cátedra: Paradigmas y Lenguajes de Programación III Cuatrimestre 2do

Carrera: Ingeniería en sistemas de información

¿Qué es?

Clean Architecture es un nombre popularizado por Robert Cecil Martin, conocido como “Uncle Bob” que se basa en la premisa de estructurar el código en capas contiguas, es decir, que solo tienen comunicación con las capas que están inmediatamente a sus lados. Basados en esta idea podemos encontrar artículos que hablan sobre Clean Architecture, Onion Architecture, Hexagonal Architecture, todas ellas tienen diferentes enfoques, pero comparten la idea de que cada nivel debe realizar sus propias tareas y se comunica únicamente con sus niveles inmediatamente contiguos.



La regla de dependencia:

La principal característica de Clean Architecture frente a otras arquitecturas es la regla de dependencia.

En Clean Architecture, una aplicación se divide en responsabilidades y cada una de estas responsabilidades se representa en forma de capa.

De esta forma tenemos capas exteriores y capas interiores:

- Las capa más exterior representa los detalles de implementación
- Las capas más interiores representan el dominio incluyendo lógica de aplicación y lógica negocio empresaria

La regla de dependencia nos dice que un círculo interior nunca debe conocer nada sobre un círculo exterior. Sin embargo, los círculos exteriores sí pueden conocer círculos interiores. La lógica de dominio es lo que menos va a cambiar con el tiempo y por lo tanto se debe evitar que tenga dependencias de detalles de implementación que van a cambiar con más frecuencia.

En qué capas se divide Clean Architecture?

Clean architecture se divide en las siguientes capas:

- Entidades
- Casos de uso
- Adaptadores
- Frameworks y drivers

También podemos ver estas capas bajo la siguiente agrupación:

- Dominio -> entidades y casos de uso
- Adaptadores
- Detalles de implementación -> frameworks y drivers

Dominio

El dominio es el corazón de una aplicación y tiene que estar totalmente aislado de cualquier dependencia ajena a la lógica o los datos de negocio.

Entidades

La lógica de negocio empresarial es aquella lógica que existiría aunque no tengamos una aplicación para automatizar los procesos de una compañía.

Por ejemplo, para realizar un pedido es necesario que deba existir un cliente con nombre, apellidos, NIF y una dirección.

La lógica y datos de negocio empresarial se representa utilizando las entidades.

Las entidades contienen los datos de negocio así como las reglas de negocio empresarial.

Las entidades de una aplicación podrían ser compartidas entre diferentes aplicaciones dentro de una compañía.

Por ejemplo en el caso anterior la entidad cliente podría compartirse entre una aplicación móvil, si quisiéramos realizar las validaciones sin necesidad de una llamada de red, y su backend o API porque las reglas para dar de alta un cliente serían las mismas.

Casos de uso

Los casos de uso representan la lógica de aplicación, que existe principalmente debido a la automatización de procesos mediante la aplicación y es inherente a cada aplicación.

Por ejemplo para crear un pedido, el caso de uso entre la aplicación móvil y la API tiene sentido que sean diferentes.

En el caso de uso crear pedido en la aplicación móvil, puede tener unas validaciones mínimas necesarias si queremos evitar llamadas a red previas y la posterior llamada a la API para crear el pedido.

Sin embargo en la API, tiene sentido que sea diferente. A parte de crear el pedido, es posible que el caso de uso se encargue de coordinar el envío de email al usuario, gestión de stock, generación de factura etc.

Adaptadores

Los adaptadores se van a encargar de transformar la información como se entiende y es representada en los detalles de implementación o frameworks, drivers a como la entiende el dominio.

Es habitual utilizar en este punto junto con clean architecture diferentes patrones de presentación como MVC, MVP, MVVM o BloC donde los presenters, controllers, viewmodels o blocs serían los adaptadores encargados de transformar la información de las vistas a información que necesitan los casos de uso.

Características:

1. Es independiente de cualquier framework

La arquitectura limpia debe ser capaz de aplicarse a cualquier sistema sin importar el lenguaje de programación o las librerías que utilice. Las capas deben quedar tan bien separadas que puedan sobrevivir individualmente, sin necesidad de externos.

2. Testeable

Entre más pura sea una función, clase o módulo más fácil será predecir el resultado a obtener. Cuando hablamos de que algo sea puro nos referimos a que no tenga efectos colaterales. Lee este artículo para que entiendas mucho más sobre los efectos colaterales y funciones puras. (Importante leerlo para completar la lección). Cada módulo, tanto de UI, base de datos, conexión a API Rest, etc., se debe poder testear de forma individual.

3. Independiente de la interfaz de usuario (UI)

Uno de los componentes que sufren cambios constantemente es la interfaz de usuario. La UI debe ser capaz de cambiar sin alterar todo el sistema y, si vamos más allá, esta capa debería vivir tan independiente que podría ser desensamblada y sustituida por otra. Por ejemplo, cambiar una UI Móvil por una en modo consola.

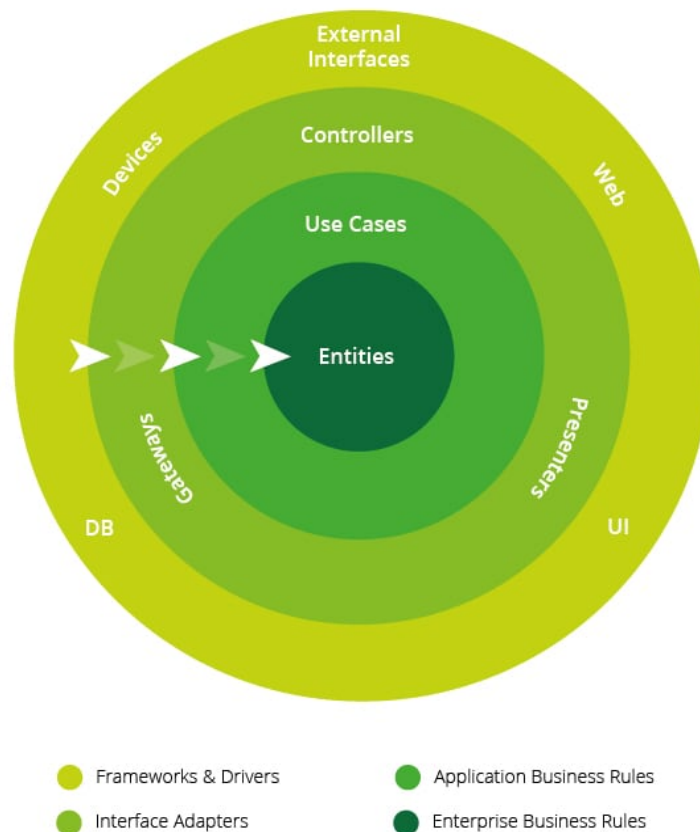
4. Independiente de la base de datos

Así como en el punto anterior, esta capa debe ser tan modular que sea posible agregarle múltiples fuentes de datos, e incluso múltiples fuentes del mismo tipo de datos. Por ejemplo, manejar varias bases de datos como MySQL, PostgreSQL, Redis, etc.

5. Independiente de cualquier elemento externo

Si en algún punto nuestro sistema necesita de una librería, otro sistema o cualquier elemento a conectar, debería ser fácilmente ensamblado y también debería ser modularizado. De hecho, para el sistema esta capa externa debería ser transparente. Estos principios fueron graficados por el Tío Bob en el siguiente diagrama:

Este es el gráfico original que propone Uncle Bob y se puede apreciar cómo los diferentes niveles anteriormente descritos sólo se comunican con los inmediatamente contiguos.



A continuación, tenemos otra interpretación del gráfico tradicional, en la que se han añadido descripción de las tareas que se suelen realizar en cada uno de los niveles.



Cada uno de los niveles aquí representados podría considerarse dentro de la estructura de nuestra aplicación como carpetas independientes, diferentes módulos o librerías que se incluyen como dependencias del proyecto principal o incluso podríamos aplicar este tipo de estructuración sin la necesidad de reflejar de forma explícita en la estructura del proyecto.

¿Cuándo usar Clean Architecture?

Clean architecture tiene sentido cuando estamos desarrollando una aplicación que va a tener una duración de vida media o larga.

Si tienes que crear una aplicación para un evento como una feria y después no se va a mantener más tal vez no tenga sentido.

Si por el contrario va a tener una duración media o larga, donde la aplicación va a ser evolucionada con nuevas features, vamos a actualizar librerías de las que depende a nuevas versiones en un futuro o utilizar nuevas, tiene sentido usar Clean Architecture.

Ventajas

Independencia: cada capa tiene su propio paradigma o modelo arquitectónico como si se tratara de una aplicación en sí misma sin afectar al resto de los niveles.

Estructuración: mejor organización del código, facilitando la búsqueda de funcionalidades y navegación por el mismo.

Desacoplamiento: cada capa es independiente de las demás por lo que podríamos reemplazarla o incluso desarrollar en diferentes tecnologías. Además de reutilizar alguna de ellas en diferentes proyectos.

Facilidad de testeo: podremos realizar test unitarios de cada una de las capas y test de integración de las diferentes capas entre sí, pudiendo reemplazarlas por objetos temporales que simulan su comportamiento de forma sencilla.

Desventajas

Metodología: todo el equipo de desarrollo debe conocer la metodología que se está aplicando y cada desarrollador debe ser responsable de entender y aplicar las reglas establecidas a medida que se está desarrollando.

Complejidad: la velocidad de desarrollo al comienzo del proyecto es menor debido a que hay que establecer esta estructura y todo el equipo debe adaptarse a la nueva forma de trabajar, pero poco a poco y a medida que la aplicación va creciendo el mantenimiento y ampliación será más sencillo.

Conclusiones

Hemos visto porque utilizo clean architecture, una introducción a las principales partes de arquitectura y sus ventajas.

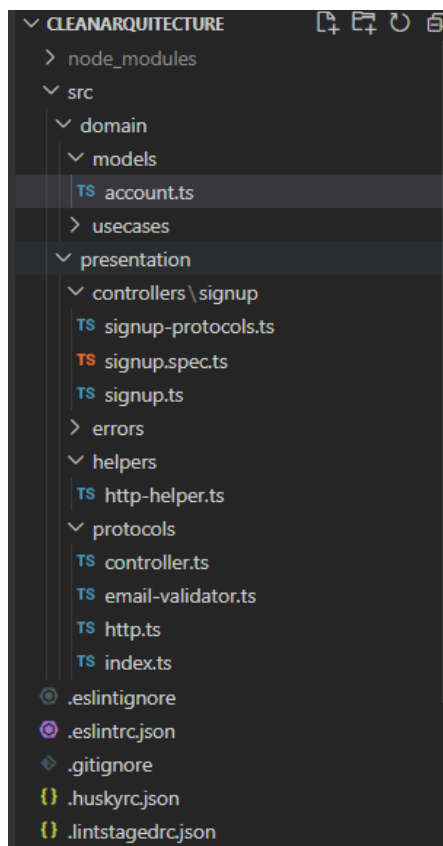
Al final se trata de utilizar las herramientas en tu trabajo con las que te sientas más cómodo y hacer tu trabajo mejor, de una forma más productiva.

Cumplir con estas reglas simples no es difícil y le ahorrará muchos dolores de cabeza en el futuro. Al separar el software en capas y cumplir con **la regla de dependencia**, creará un sistema que es intrínsecamente comprobable, con todos los beneficios que ello implica.

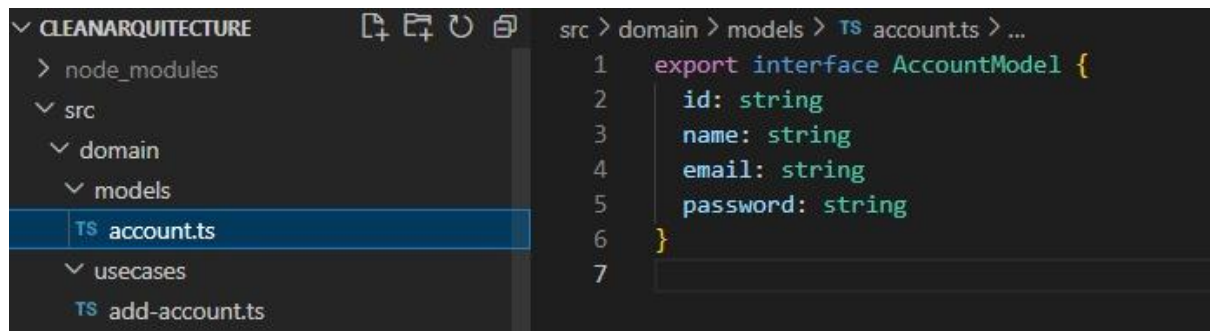
Cuando alguna de las partes externas del sistema se vuelve obsoleta, como la base de datos o el marco web, puede reemplazar esos elementos obsoletos con un mínimo de esfuerzo.

Código de Ejemplo

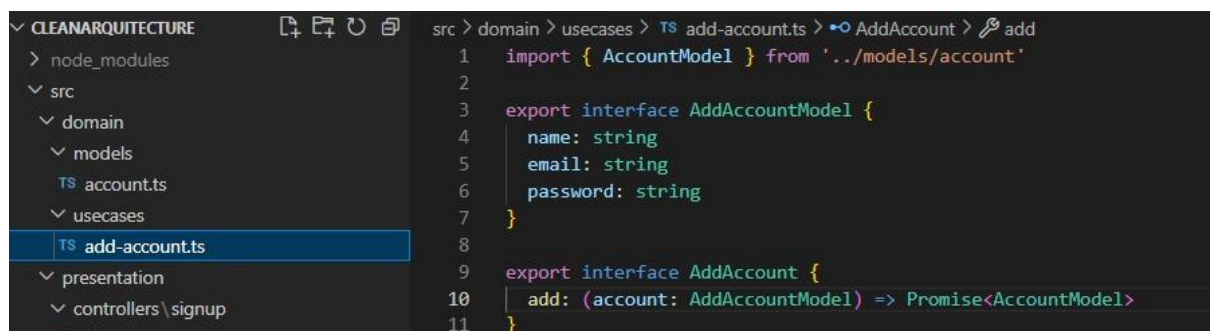
El ejemplo que encontramos es de un verificador de emails en donde se ve claramente la estructura y distribución de capas que comentamos con anterioridad. Se puede observar que se divide en este caso en dominio y presentación.



El dominio contiene a `models` y a los casos de uso en donde se encuentra `account.ts` y `add-accounts.ts`.



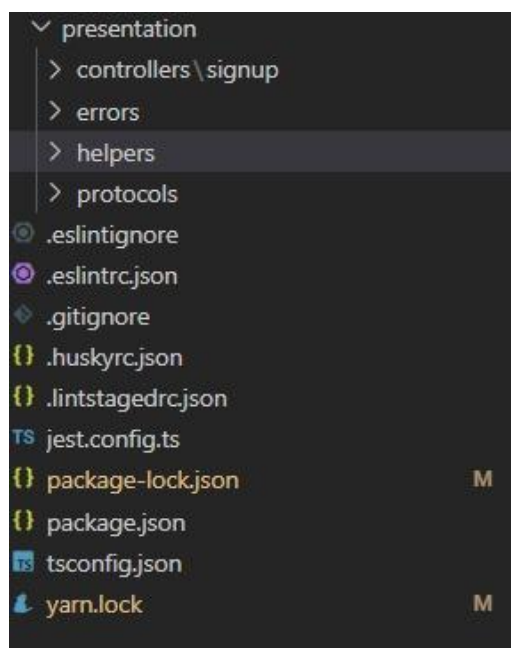
```
src > domain > models > TS account.ts > ...
1  export interface AccountModel {
2      id: string
3      name: string
4      email: string
5      password: string
6  }
7
```



```
src > domain > usecases > TS add-account.ts > AddAccount > add
1  import { AccountModel } from '../models/account'
2
3  export interface AddAccountModel {
4      name: string
5      email: string
6      password: string
7  }
8
9  export interface AddAccount {
10     add: (account: AddAccountModel) => Promise<AccountModel>
11 }
```

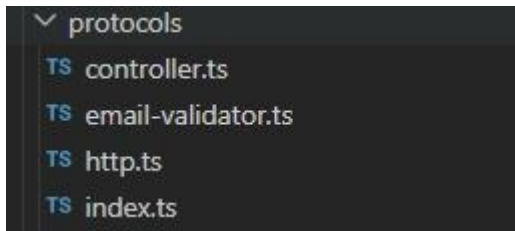
Dentro de presentacion, se encuentran las interfaces como controllers, email-validator, http request y index.ts donde se estaría exportando todo lo que sería controllers y http.

Lo que hace una interfaz es define un comportamiento a las clases que se van a implementar después a esta interfaz



```
presentation
├── controllers\signup
├── errors
├── helpers
├── protocols
├── .eslintignore
├── .eslintrc.json
├── .gitignore
├── .huskyrc.json
├── .lintstagedrc.json
├── jest.config.ts
├── package-lock.json
├── package.json
├── tsconfig.json
└── yarn.lock
```

Protocols:



Se generaliza un comportamiento para las clases y no nos va importar cómo esas clases se implementan. Aquí se implementa controller por lo que vamos a necesitar un método handle que va a recibir un httpRequest y va a devolver una promesa HttpResponse.

Promesa: Una promesa es una ejecución asíncrona, un ejemplo sería cuando hacemos una request a la api, con un fetch, se estaría haciendo una ejecución asíncrona, en Typescript es como que se queda escuchando.

```
src > presentation > protocols > TS controller.ts > ...  
1  import { HttpRequest, HttpResponse } from './http'  
2  
3  export interface Controller {  
4    handle: (HttpRequest: HttpRequest) => Promise<HttpResponse>  
5  }  
6
```

En email se define una interfaz en la cual no nos va a interesar cómo se ejecuta la validación del email pero sabemos que el que implemente esta interfaz tiene que tener un método que se llama isValid que contiene si o si un email que es de clase string y va a devolver un booleano. Esto no tiene lógica, solo es una definición del método, no se hace ninguna verificación, solo se define el comportamiento.

```

src > presentation > protocols > TS email-validator.ts > EmailValidator
1  ✓ export interface EmailValidator {
2    isValid: (email: string) => boolean
3  }
4

```

El index.ts solo hace la exportación a protocols para el que ingrese tenga acceso a las interfaces de http

```

src > presentation > protocols > TS index.ts
1  export * from './controller'
2  export * from './http'
3

```

HttpResponse es un objeto que tiene dos parámetros, statusCode : son los códigos status de http que es un número y el body. Estos son algunos errores que salen como por ejemplo

error 200: Que pasa , que está todo ok

error 300: Que indican que la solicitud tiene más de una solicitud posibles

error 400: Indican que el servidor no puede o no procesará la solicitud porque percibe un error del cliente

error 500: Son errores en el servidor, una respuesta de error genérica de http.

```

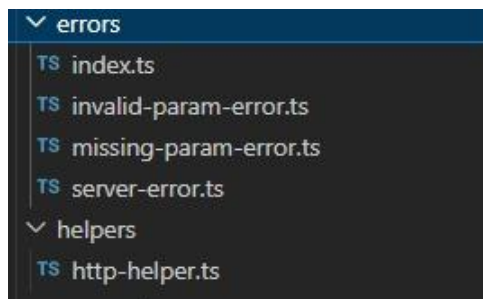
src > presentation > protocols > TS http.ts > •• Htt
1  export interface HttpResponse {
2      statusCode: number
3      body: any
4  }
5
6  export interface HttpRequest {
7      body?: any
8  }
9

```

Errores

Errores son métodos que se separan en errores y helpers para después ser usados dentro de los controladores

Son funciones compartidas que van a ser usadas por los métodos dentro del controlador básicamente una especie de api



Helpers: Es un archivo helpers que son constantes donde se devuelve los tipos de respuestas

```
src > presentation > helpers > TS http-helper.ts > ...
1  ∨ import { ServerError } from '../errors'
2  import { HttpResponse } from '../protocols/http'
3
4  ∨ export const badRequest = (error: Error): HttpResponse => ({
5      statusCode: 400,
6      body: error
7  })
8
9  ∨ export const serverError = (): HttpResponse => ({
10     statusCode: 500,
11     body: new ServerError()
12 })
13
14 ∨ export const ok = (data: any): HttpResponse => ({
15     statusCode: 200,
16     body: data
17 })
18
```

Controllers: (En singup.ts)

En Singup-protocols se agrupan todas las exportaciones en un solo archivo y que no se tengan que escribir los import en singup.ts. Dentro de controler ya se estaría haciendo un poco de la logica de las acciones, pero se delega la logica del negocio a los casos de uso

Por ejemplo en la línea 16 se estaría controlando que se esté recibiendo un nombre, un email, una contraseña y que esta contraseña sea correcta, y por cada uno verifica si es un campo requerido y si no existe devuelve un badRequest y incluso pasa el mensaje del error

Y en el caso que no de ningún error el programa se va a seguir ejecutando hasta que encuentre uno , en el caso de que no encuentre un error, el controlador signup va a verificar de que se registró el mail.

```

src > presentation > controllers > signup > ts signup.ts > SignUpController > handle > account
1  import { InvalidParamError, MissingParamError } from '../../errors'
2  import { HttpRequest, HttpResponse, Controller, EmailValidator, AddAccount } from './signup-protocols'
3  import { badRequest, serverError, ok } from '../../helpers/http-helper'
4
5  export class SignUpController implements Controller {
6      private readonly emailValidator: EmailValidator
7      private readonly addAccount: AddAccount
8
9      constructor (emailValidator: EmailValidator, addAccount: AddAccount) {
10         this.emailValidator = emailValidator
11         this.addAccount = addAccount
12     }
13
14     async handle (httpRequest: HttpRequest): Promise<HttpResponse> {
15         try {
16             const requiredFields = ['name', 'email', 'password', 'passwordConfirmation']
17             for (const field of requiredFields) {
18                 if (!httpRequest.body[field]) {
19                     return badRequest(new MissingParamError(field))
20                 }
21             }
22             const { name, email, password, passwordConfirmation } = httpRequest.body
23
24             if (password !== passwordConfirmation) {
25                 return badRequest(new InvalidParamError('passwordConfirmation'))
26             }
27
28             const isValid = this.emailValidator.isValid(email)
29             if (!isValid) {
30                 return badRequest(new InvalidParamError('email'))
31             }
32
33             const account = await this.addAccount.add({
34                 name,
35                 email,
36                 password
37             })
38         } catch (error) {
39             return serverError(error.message)
40         }
41         return ok()
42     }
43 }

```

signup.specs

Esta sería por así decir la última capa del programa, “el framework”.

Acá sería donde se probarían los test, y entonces en cada test se prueba de que esté funcionando correctamente el software.

```

src > presentation > controllers > signup > ts signup.specs > makeEmailValidator > EmailValidatorStub > isValid
1  import { SignUpController } from './signup'
2  import { InvalidParamError, MissingParamError, ServerError } from '../../errors'
3  import { EmailValidator, AddAccountModel, AddAccount } from './signup-protocols'
4  import { AccountModel } from '../../domain/models/account'
5
6  const makeEmailValidator = (): EmailValidator => {
7      class EmailValidatorStub implements EmailValidator {
8          isValid (email: string): boolean {
9              return true
10             }
11         }
12         return new EmailValidatorStub()
13     }
14 }

```


Test:

```
SignUp Controller
✓ Should return 400 if no name is provided (7 ms)
✓ Should return 400 if no email is provided (1 ms)
✓ Should return 400 if no password is provided (1 ms)
✓ Should return 400 if no password confirmation is provided (1 ms)
✓ Should return 400 if no password confirmation fails (2 ms)
✓ Should return 400 if an invalid email is provided (4 ms)
✓ Should call EmailValidator with correct email (4 ms)
✓ Should call AddAccount with correct values (3 ms)
✓ Should return 500 if EmailValidator throws (1 ms)
✓ Should return 500 if AddAccount throws (1 ms)
✓ Should return 200 if valid data is provided (1 ms)
```

Bibliografía:

<https://dev.to/japhernandez/clean-architecture-with-nodejs-typescript-and-mongo-3ene>

<https://es.linkedin.com/pulse/clean-architecture-cristofer-padilla>

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

<https://platzi.com/clases/1619-arquitectura-android/21324-que-es-clean-architecture/>

<https://codersopinion.com/blog/clean-architecture-building-software-that-lasts/>