Abstractions in Clean Architecture

"The error is your best teacher, embrace him"

Víctor A. Higuita C. <vickodev@gmail.com> <@vickodev> GitHub <@harvic3>

¡Hi everyone!

¡Curiosities!🤓

*Replacing the fossil fuel energy we use requires +1000 times global GDP.
*Plastic pollution is so alarming that every week we ingest the equivalent of one TC.
*Global warming is a positive feedback problem.

¿Qué vamos a explorar?

- What is software architecture?
- What is the patterns?
- What is abstraction? 🤯
- What is inversion of control IoC?
- What is dependency inversion DIv?
- What is dependency injection DIj?
- What is clean architecture (CA)? 🥁
- The magic formula for making CA and not failing in the attempt🤔
- Let's review the **Application** diagram of our solution
- Let's review **Infrastructure** and **Adapters** layer diagrams 🤓
- Challenges to our solution 🤖

What is the dynamic?

Short definition, !my head exploded! 🤯

In the flow of the presentation, each main slide will have an **image allusive to the topic** or **concept**, adjacent to it the short definition as an adaptation of the concept, and here in this area will be the interlocution (Explanation). (💩)

Mentioning the emoji a reflection, Hammer 🔨 or Pliers ✂️?

What is the Software Architecture?

We can define it as the **design decisions** (structure) that are made to meet the **quality attributes** of a software product.

In the beginning it was all obscurity (terminals if anything)👀, and in the 1960s concepts like **modularity**, **inheritance**, **encapsulation** began to be touched upon, but it was in the 1990s where the term **architecture** was used in contrast to **design**, evoking notions of **abstraction** and **standards**.

What is the Patterns?

Architecture Patterns

They are **archetypes** (structures) that tell us how the parts of a software-based system **relate/interact** at the macro level.

Design Patterns

They are proven base **templates** for solving common problems. They are categorized into three groups (**creative**, **structural**, **behavioral**). Originally 23 were published but now there are many more.

What is Abstraction?

It can be defined as the separation of **the what** and **the how**, that is, separating what something **does** from how it **does it**.

The what is **the action** (Interface, Abstract class), this action receives or not input parameters and tells us what the result is. **The how** is **what it does** (behavior) with its inputs and its internal objects to return the result, that is the **concrete Class or implementation**.

In code it looks something like this

The what does

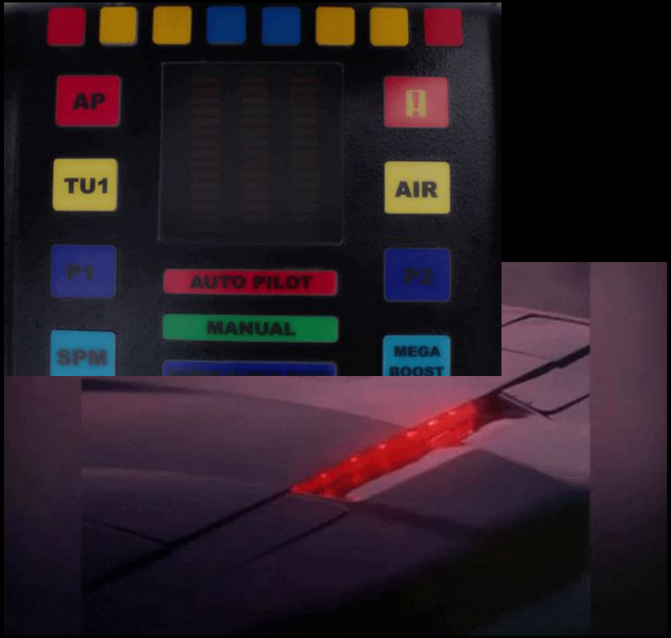
```
// Interface way
interface IChatProvider {
  send(recipient: string, text: string): Promise<void>;
  read(): Promise<Message[]>;
}

// Abstract class way
abstract class ChatProviderContract {
  abstract send(recipient: string, text: string): Promise<void>;
  abstract read(): Promise<Message[]>;
}
```

How it does it

```
class ChatProvider implements IChatProvider {
// class ChatProvider extends ChatProviderContract
  #address: string;
  #messages: Message[] = [];

  constructor(address: string) {
    this.#address = address;
  }

  async send(recipient: string, text: string): Promise<void> {
    // implement the how it does it
  }
  async read(): Promise<Message[]> {
    // implement the how it does it
  }
}
```

Víctor A. Higuita C. <@vickodev>

What is inversion of control (IoC)?



It's **to delegate control** of certain aspects of an application to a third party.

This third party is usually a **framework** or **service**, and can control from the lifecycle of objects (DIj) to that of the application itself, it can also control events and invoke application actions to those events (UI), it can also control data persistence and providers of different types of services.

What is dependency inversion (DIv)?

It's basically **Abstraction**, and its main function is to **decouple**.

This is one of the SOLI***D*** (DIvP) principles and landing its compendiums to a non-technical language we can say that:

- **Important classes** should not depend on **less important classes**, **both** should depend on abstractions.
- **The what** should not depend on **the how**, **the how** should depend on **the what**.

What is dependency injection (DIj)?

It is to separate the **Construction/Creation** (instances) from the execution context where that object interacts.

It is **to delegate the creation** of objects to a third party, in this case to a dependency injection container (DIjC) with the intention of decoupling the core of our application from the outside world, that world of "trivial" things.

Now the dilemma🤔

Is’t possible to use Dependency Injection without using Dependency Inversion?

What is Clean Architecture?🥁

It can be defined as a framework or set of best practices where the objective is to facilitate the **construction**, **testability** and **maintenance** of software products.

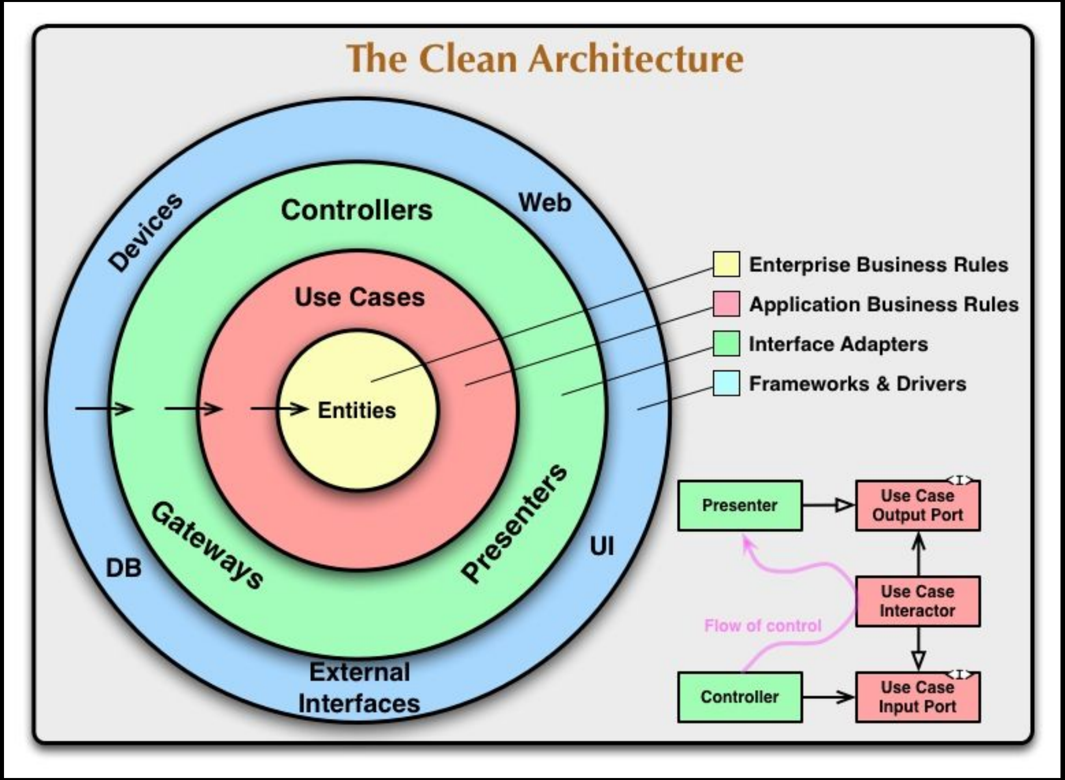
The term was coined by **Robert Martin** and in this compendium he took part of other architectural concepts (**Tiers and Layers**, **Onion**, **Hexagonal**) where the goal is **abstraction**, structuring the code through layers and isolating the inner world (application and business logic) from the "trivial" things like the outside world.

Let's review the characteristics of CA

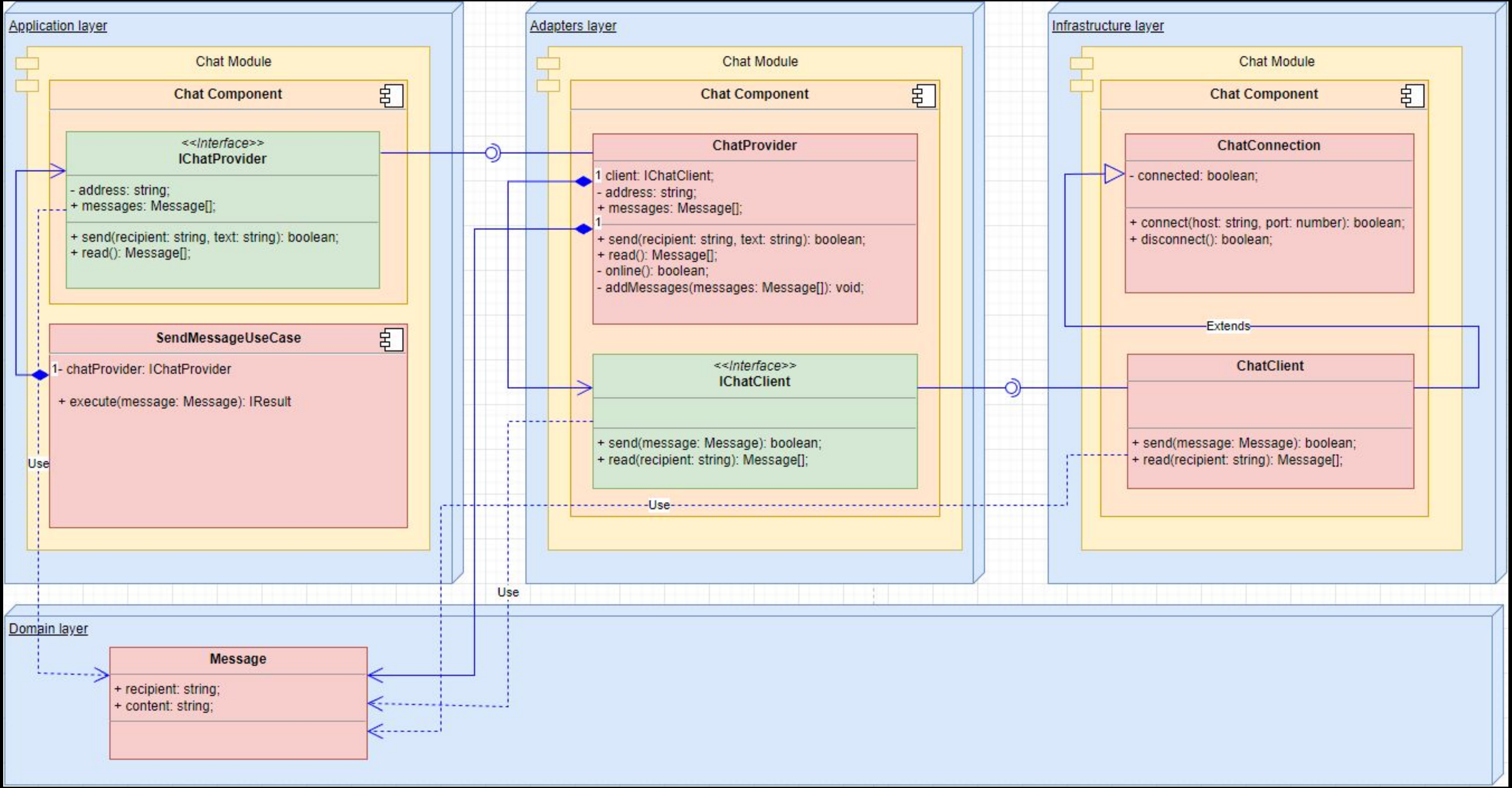
- Coupling is only allowed between a layer and its inner neighboring layer. *
- An inner layer must not depend on (be coupled to) an outer layer.

Entities = Business ***Domain***
Use Cases = ***Application*** Domain and it orchestrates the business domain
Layer green = ***Adapters*** layer
Layer blue = ***Infrastructure*** layer

* Part of this is negotiated with the technical team.



Magic formula to do CA and not fail 🤓



In code it looks something like this

```
// Application layer
interface IChatProvider {
  send(recipient: string, text: string): Promise<void>;
  read(): Promise<Message[]>;
}
```

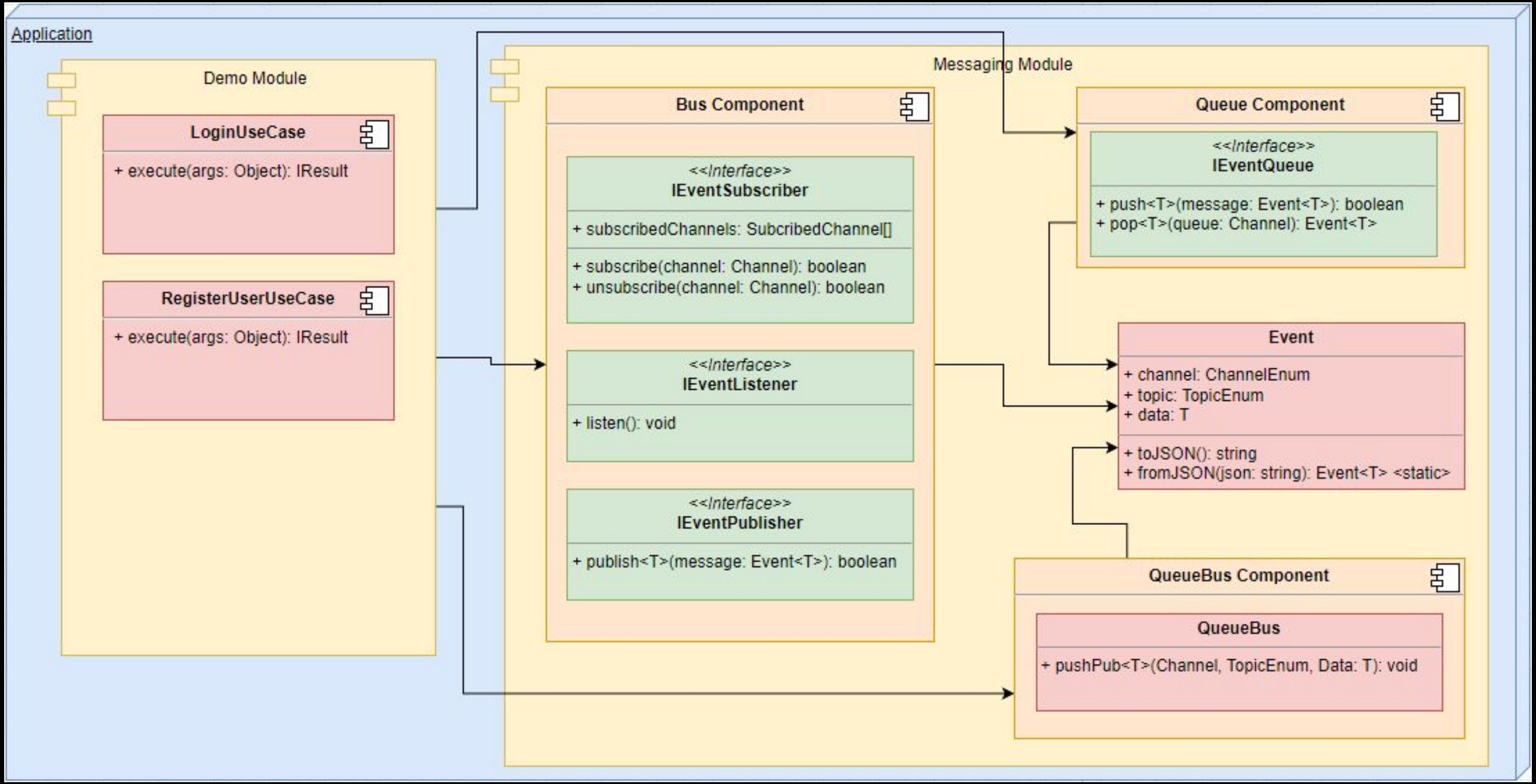
```
// Adapter layer
interface IChatClient {
  send(message: Message): Promise<boolean>;
  read(recipient: string): Promise<Message[]>;
}

class ChatProvider implements IChatProvider {
  #address: string;
  #chatClient: IChatClient;
  #messages: Message[] = [];

  async send(recipient: string, text: string): Promise<boolean>
{
    // implementation
  }
  async read(): Promise<Message[]> {
    // implementation
  }
}
```

```
// Infrastructure layer
class ChatConnection {
  #connected: boolean = false;
  #chatIO: ChatIO;

  connect(host: string, port: number): boolean {
    // implementation
  }
  disconnect(): boolean {
    // implementation
  }
}

class ChatClient implements IChatClient {
  #connection: ChatConnection;

  async send(message: Message): Promise<boolean> {
    // implementation
  }
  async read(recipient: string): Promise<Message[]> {
    // implementation
  }
}
```

Now, let’s see it in an implementation 🤓

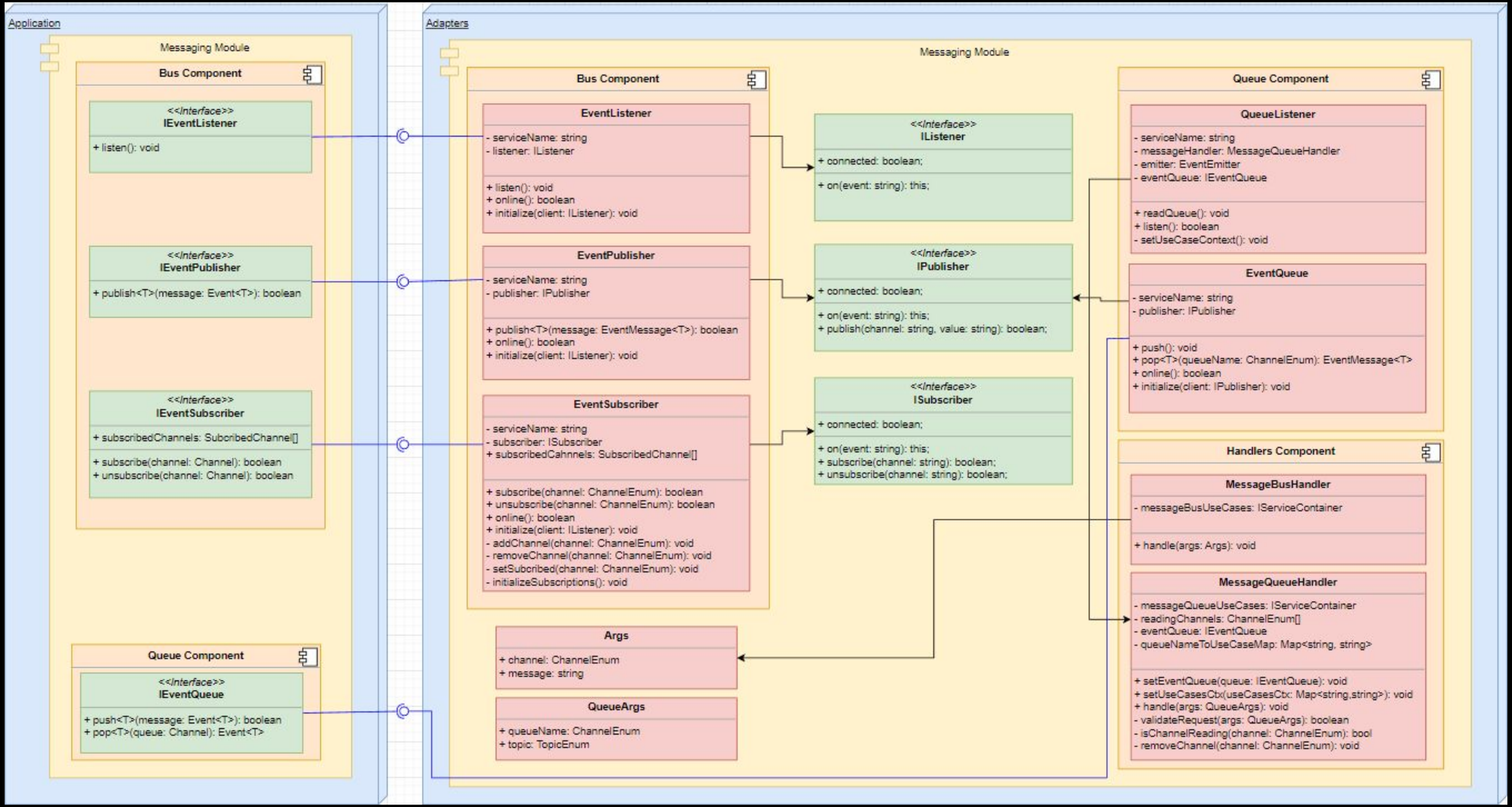
We will focus on the **Application**, **Infrastructure** and **Adapters** layers of a Clean Architecture based solution. 🤩

Let's review our Application layer


Application
Demo Module
LoginUseCase
+ execute(args: Object): IResult
RegisterUserUseCase
+ execute(args: Object): IResult
Messaging Module
Bus Component
<<Interface>>
IEventSubscriber
+ subscribedChannels: SubcribedChannel[]
+ subscribe(channel: Channel): boolean
+ unsubscribe(channel: Channel): boolean
<<Interface>>
IEventListener
+ listen(): void
<<Interface>>
IEventPublisher
+ publish<T>(message: Event<T>): boolean
Queue Component
<<Interface>>
IEventQueue
+ push<T>(message: Event<T>): boolean
+ pop<T>(queue: Channel): Event<T>
Event
+ channel: ChannelEnum
+ topic: TopicEnum
+ data: T
+ toJSON(): string
+ fromJSON(json: string): Event<T> <static>
QueueBus Component
QueueBus
+ pushPub<T>(Channel, TopicEnum, Data: T): void

Víctor A. Higuita C. <@vickodev>

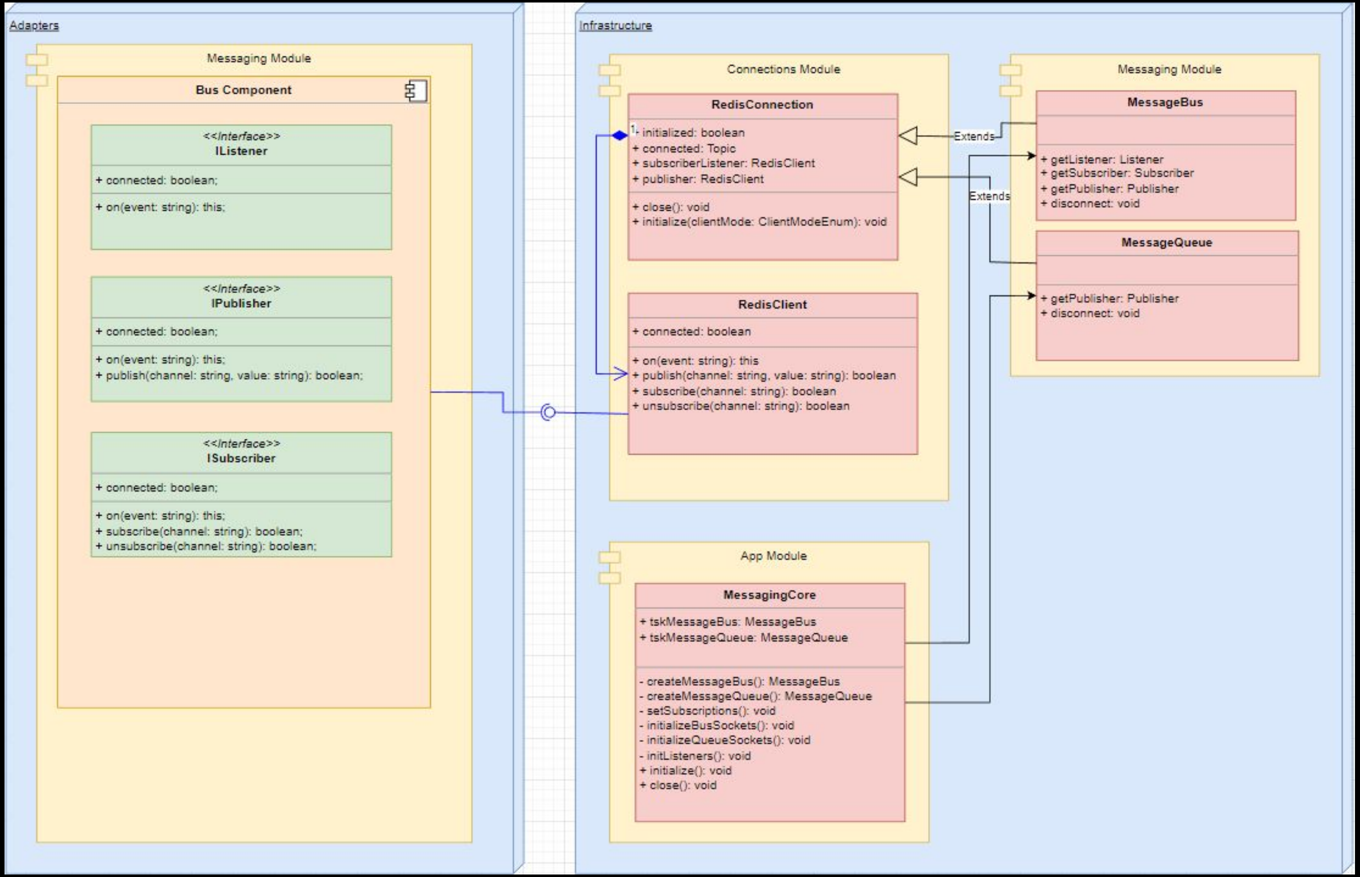
Let’s review our Adapters layer



Application
Messaging Module
Bus Component
<<Interface>>
IEventListener
+ listen(): void
<<Interface>>
IEventPublisher
+ publish<T>(message: Event<T>): boolean
<<Interface>>
IEventSubscriber
+ subscribedChannels: SubcribedChannel[]
+ subscribe(channel: Channel): boolean
+ unsubscribe(channel: Channel): boolean
Queue Component
<<Interface>>
IEventQueue
+ push<T>(message: Event<T>): boolean
+ pop<T>(queue: Channel): Event<T>
Adapters
Messaging Module
Bus Component
EventListener
- serviceName: string
- listener: IListener
+ listen(): void
+ online(): boolean
+ initialize(client: IListener): void
EventPublisher
- serviceName: string
- publisher: IPublisher
+ publish<T>(message: EventMessage<T>): boolean
+ online(): boolean
+ initialize(client: IListener): void
EventSubscriber
- serviceName: string
- subscriber: ISubscriber
+ subscribedCahnnels: SubscribedChannel[]
+ subscribe(channel: ChannelEnum): boolean
+ unsubscribe(channel: ChannelEnum): boolean
+ online(): boolean
+ initialize(client: IListener): void
- addChannel(channel: ChannelEnum): void
- removeChannel(channel: ChannelEnum): void
- setSubcribed(channel: ChannelEnum): void
- initializeSubscriptions(): void
<<Interface>>
IListener
+ connected: boolean;
+ on(event: string): this;
<<Interface>>
IPublisher
+ connected: boolean;
+ on(event: string): this;
+ publish(channel: string, value: string): boolean;
<<Interface>>
ISubscriber
+ connected: boolean;
+ on(event: string): this;
+ subscribe(channel: string): boolean;
+ unsubscribe(channel: string): boolean;
Args
+ channel: ChannelEnum
+ message: string
QueueArgs
+ queueName: ChannelEnum
+ topic: TopicEnum
Queue Component
QueueListener
- serviceName: string
- messageHandler: MessageQueueHandler
- emitter: EventEmitter
- eventQueue: IEventQueue
+ readQueue(): void
+ listen(): boolean
- setUseCaseContext(): void
EventQueue
- serviceName: string
- publisher: IPublisher
+ push(): void
+ pop<T>(queueName: ChannelEnum): EventMessage<T>
+ online(): boolean
+ initialize(client: IPublisher): void
Handlers Component
MessageBusHandler
- messageBusUseCases: IServiceContainer
+ handle(args: Args): void
MessageQueueHandler
- messageQueueUseCases: IServiceContainer
- readingChannels: ChannelEnum[]
- eventQueue: IEventQueue
- queueNameToUseCaseMap: Map<string, string>
+ setEventQueue(queue: IEventQueue): void
+ setUseCasesCtx(useCasesCtx: Map<string,string>): void
+ handle(args: QueueArgs): void
- validateRequest(args: QueueArgs): boolean
- isChannelReading(channel: ChannelEnum): bool
- removeChannel(channel: ChannelEnum): void

Víctor A. Higuita C. <@vickodev>

Let’s review our Infrastructure layer


Adapters
Messaging Module
Bus Component
<<Interface>>
IListener
+ connected: boolean;
+ on(event: string): this;
<<Interface>>
IPublisher
+ connected: boolean;
+ on(event: string): this;
+ publish(channel: string, value: string): boolean;
<<Interface>>
ISubscriber
+ connected: boolean;
+ on(event: string): this;
+ subscribe(channel: string): boolean;
+ unsubscribe(channel: string): boolean;
Infrastructure
Connections Module
RedisConnection
1
+ initialized: boolean
+ connected: Topic
+ subscriberListener: RedisClient
+ publisher: RedisClient
+ close(): void
+ initialize(clientMode: ClientModeEnum): void
RedisClient
+ connected: boolean
+ on(event: string): this
+ publish(channel: string, value: string): boolean
+ subscribe(channel: string): boolean
+ unsubscribe(channel: string): boolean
Extends
Extends
Messaging Module
MessageBus
+ getListener: Listener
+ getSubscriber: Subscriber
+ getPublisher: Publisher
+ disconnect: void
MessageQueue
+ getPublisher: Publisher
+ disconnect: void
App Module
MessagingCore
+ tskMessageBus: MessageBus
+ tskMessageQueue: MessageQueue
- createMessageBus(): MessageBus
- createMessageQueue(): MessageQueue
- setSubscriptions(): void
- initializeBusSockets(): void
- initializeQueueSockets(): void
- initListeners(): void
+ initialize(): void
+ close(): void

Preparing for the Challenges

Repository

¡We learn by doing, so let's get to work!

Clone the repository and follow the instructions in the README file to run it. Keep in mind that the branch that has the **redis** implementation is feature/queue-bus-events

Once the repository is running, test the requests given in the README examples, both the user registration and authentication requests, test with the default user and the new ones you want to create (Warning, in-memory database*).

The Challenges

Play with the template, explore its components and try to understand how they work, then mess with it, change and move things around, and carefully analyze the effects you find. Level1

Create a single Messaging Handler Messaging.handler.ts in a messaging directory in the path 'src\adapters\messaging\handlers' and remove the other handler directories (bus and queue directories) and handle the use cases from that single handler WITHOUT USING the Queue's EventEmitter, which you must remove from the solution. Level2

Abstractly set up a cache provider using the current Redis infrastructure that coexists with the Message Bus and Message Queue, and use that cache in the authentication use case to cache the session. Check through the terminal that the session is cached. You can use the maskedUserId as key.The contracts should stay in the path 'src\application\shared\cache' and everything should still work as is, plus the new cache provider functionality. Level3

SCAN ME

Those who achieve the **Level2** or **Level3** challenges will be given ***GitHub Swag***, and we will open a discussion for questions or concerns about the challenges.

Decouple the **Adapters** layer from the **Infrastructure** layer in terms of dependencies coupled from **Redis**. Level3+

¡Success and may you learn a lot!

Víctor A. Higuita C. <@vickodev>

Acknowledgments, resources and credits

Article

Architecture

Profile SDP

IoC

Clean Architecture

Customized animations https://ezgif.com and GIFs from https://giphy.com/

¡That's all, thank you very much!

@vickodev

Víctor A. Higuita C. <vickodev@gmail.com> <@vickodev> GitHub <@harvic3>