Informe de Arquitectura Funcional

Título del Proyecto: Arquitectura Funcional del Proyecto YARG Flow

Autor(es):

Alejandro Vera, Andy Bryan

Bermúdez Santos, Gino Maximiliano

Reyes Ricardo, Alisson Yamel

Rivera Torres, Yandris Miguel

Fecha: 29 de octubre del 2025

Tabla de contenido

1. Introducción

Descripción:
YARG Flow es una aplicación de chat en tiempo real destinada a comunicación rápida y fiable entre usuarios (1:1 y canales), con un núcleo de procesamiento funcional para manejo de mensajes e historial y una cáscara imperativa que gestiona I/O (WebSocket, persistencia, notificaciones). El objetivo principal es permitir el intercambio de mensajes en tiempo real, gestión de usuarios y contactos, persistencia y consulta del historial, soporte de canales y presencia, con una separación clara entre lógica pura y efectos secundarios.

2. Filosofía Arquitectónica: Cáscara Imperativa, Núcleo Funcional

Descripción:
La arquitectura de este proyecto se basa en el patrón "Cáscara Imperativa, Núcleo Funcional". Este enfoque separa el código en dos áreas distintas:

- **El Núcleo Funcional:** Contiene toda la lógica de negocio pura. Está compuesto por funciones puras y estructuras de datos inmutables. No realiza operaciones de entrada/salida (I/O), no modifica estado global y no tiene efectos secundarios. Esto hace que el núcleo sea fácil de probar, razonar y componer. En YARG Flow el núcleo funcional contiene las estructuras de datos inmutables y las funciones puras que transforman estados (por ejemplo: enviar/recibir mensaje, actualizar historial, aplicar filtros).
- **La Cáscara Imperativa:** Es la capa externa que interactúa con el "mundo real". Se encarga de manejar los efectos secundarios como leer la entrada del usuario, escribir en la consola, acceder al sistema de archivos o realizar peticiones de red. Su principal responsabilidad es orquestar las llamadas al núcleo funcional y gestionar los resultados. Dentro de nuestro proyecto coordina las operaciones I/O (WebSocket, DB, email, logging, integración con sistemas externos) y transforma datos del mundo real hacia/desde el núcleo.

Esta separación mejora testabilidad, razonamiento y permite desplegar o simular la cáscara en entornos distintos sin cambiar la lógica de negocio. (Requisitos y tareas organizadas según Scrum).

3. El Núcleo Funcional (The Pure Core)

Descripción:
El núcleo funcional se apoya en modelos inmutables, los cuales representan las entidades centrales del sistema.
En Angular, estos modelos se implementarán como interfaces TypeScript junto con estructuras de estado inmutables (por ejemplo, mediante readonly o librerías de estado como NgRx, Signals o BehaviorSubjects inmutables).

3.1. Estructuras de Datos Principales (Modelos)

A continuación, se definen los modelos principales en pseudocódigo TypeScript:

```
// Representa a un usuario registrado en el sistema
export interface User {
  readonly id: string;              // UUID
  readonly username: string;
  readonly displayName: string;
  readonly email: string;
  readonly avatarUrl?: string;
  readonly createdAt: Date;
  readonly preferences: ReadonlyMap<string, any>;
}

// Representa un mensaje dentro de una conversación o canal
export interface Message {
  readonly id: string;
  readonly senderId: string;
  readonly receiverId?: string;     // null para canales
  readonly channelId?: string;      // null para chat 1 a 1
  readonly content: string;
  readonly timestamp: Date;
  readonly edited: boolean;
  readonly deleted: boolean;
  readonly metadata?: ReadonlyMap<string, any>;
}

// Representa un canal o grupo de conversación
export interface Channel {
  readonly id: string;
  readonly name: string;
  readonly type: 'PUBLIC' | 'PRIVATE';
  readonly members: ReadonlyArray<string>;   // Lista inmutable de
usuarios
  readonly ownerId: string;
  readonly createdAt: Date;
}

// Representa el estado de presencia de un usuario
export interface Presence {
  readonly userId: string;
  readonly status: 'ONLINE' | 'OFFLINE' | 'AWAY';
  readonly lastSeen: Date;
}

// Estado global de la aplicación (frontend)
export interface AppState {
  readonly users: ReadonlyMap<string, User>;
  readonly channels: ReadonlyMap<string, Channel>;
  readonly messagesByConversation: ReadonlyMap<string,
ReadonlyArray<Message>>;
  readonly presences: ReadonlyMap<string, Presence>;
}
```

3.2. Funciones Puras Clave

Las funciones del núcleo se encargan de transformar el estado de manera determinista. Todas ellas siguen la forma general:

function nombreDeLaFuncion(estadoActual, datosEntrada): nuevoEstado | resultado

Estas funciones:

- No realizan I/O (no leen base de datos, ni envían peticiones HTTP).
- No modifican el estado directamente (devuelven una nueva copia del estado).
- Son completamente deterministas.

A continuación se describen las principales funciones del núcleo funcional:

Funciones para la Gestión de Mensajes

function applySendMessage(state: AppState, message: Message): AppState

Agrega un nuevo mensaje al historial correspondiente (chat privado o canal) y devuelve un nuevo AppState.

function getConversationHistory(

state: AppState,

conversationId: string,

options?: { limit?: number; offset?: number; }

): ReadonlyArray<Message>

Devuelve una lista inmutable de mensajes de una conversación, permitiendo filtrado o paginación.

Funciones para la Gestión de Usuarios

function addUser(state: AppState, user: User): AppState

Agrega un nuevo usuario al estado global.

function updateUserPreferences(

state: AppState,

userId: string,

preferences: ReadonlyMap<string, any>

): AppState

Devuelve un nuevo estado con las preferencias del usuario actualizadas.

Funciones para la Gestión de Canales

function createChannel(

state: AppState,

name: string,

type: 'PUBLIC' | 'PRIVATE',

ownerId: string

): [AppState, Channel]

Crea un nuevo canal, lo asocia al creador y devuelve el nuevo estado junto con el canal generado.

Funciones para la Gestión de Presencia

function applyPresenceUpdate(state: AppState, presence: Presence): AppState

Actualiza el estado de conexión de un usuario en el sistema (por ejemplo, al entrar o salir de la aplicación).

Funciones para Notificaciones (cálculo funcional)

function computeNotificationRecipients(

state: AppState,

message: Message

): ReadonlyArray<string>

Devuelve la lista de usuarios que deben ser notificados ante un nuevo mensaje, sin enviar nada (la cáscara se encarga del envío real).

Elixir

El estado de nuestra aplicación se modela con las siguientes estructuras inmutables (definidas en Elixir usando Ecto.Schema, que por naturaleza promueve la inmutabilidad):

```
# lib/inmutable_chat/accounts/user.ex
defmodule InmutableChat.Accounts.User do
  use Ecto.Schema

  schema "users" do
    field :username, :string
    field :email, :string
    field :password_hash, :string
    timestamps()
  end
end

# lib/inmutable_chat/chat/message.ex
defmodule InmutableChat.Chat.Message do
  use Ecto.Schema

  schema "messages" do
    field :content, :string
    belongs_to :sender, InmutableChat.Accounts.User
    belongs_to :channel, InmutableChat.Chat.Channel
    belongs_to :receiver, InmutableChat.Accounts.User # Para chat 1-a-1
    timestamps()
  end
end

# lib/inmutable_chat/chat/channel.ex
defmodule InmutableChat.Chat.Channel do
  use Ecto.Schema

  schema "channels" do
```

```
    field :name, :string
    belongs_to :owner, InmutableChat.Accounts.User
    many_to_many :members, InmutableChat.Accounts.User, join_through:
"channel_members"
    timestamps()
  end
end
```

Funciones claves puras

El núcleo está compuesto por un módulo de lógica pura (ChatLogic) que opera sobre estas estructuras. Estas funciones no interactúan con la base de datos ni con la red; solo transforman datos.

```
# lib/inmutable_chat/chat/chat_logic.ex

# Firma: process_new_message(payload :: map(), sender :: User.t()) -> Message.t()
def process_new_message(payload, sender) do
  # Descripción: Toma el mapa de entrada (payload) y el usuario que envía.
  # Devuelve una *nueva* estructura de Mensaje, lista para ser insertada,
  # aplicando lógica de negocio (ej. validación, timestamps).
  %Message{
    content: payload["content"],
    sender_id: sender.id,
    receiver_id: payload["receiver_id"], # (Será nil si es un canal)
    channel_id: payload["channel_id"]  # (Será nil si es 1-a-1)
  }
end

# Firma: validate_username(username :: string()) -> :ok | {:error, reason :: string()}
def validate_username(username) do
  # Descripción: Valida que un nombre de usuario cumpla con las reglas de negocio.
  # Devuelve un átomo :ok o una tupla de error. No tiene efectos secundarios.
  if String.length(username) > 3 do
    :ok
  else
    {:error, "El nombre de usuario es muy corto"}
  end
end
```

4. La Cáscara Imperativa (The Impure Shell)

Descripción:
La cáscara imperativa de *YARG Flow* está compuesta por los componentes y servicios que interactúan directamente con el mundo exterior: red, almacenamiento, servidor y usuario. Su función es recibir datos del entorno, convertirlos en estructuras manejables por el núcleo funcional, y luego ejecutar efectos secundarios (guardar, enviar, mostrar, o notificar) según los resultados devueltos por el núcleo.

En Angular, la cáscara está representada principalmente por:

- **Servicios (Services)**, encargados de comunicarse con APIs REST y WebSockets.
- **Componentes (Components)**, que capturan la interacción del usuario y muestran los resultados.
- **Stores / Signals**, que orquestan la sincronización del estado inmutable proveniente del núcleo con la interfaz visual.

Esta capa no contiene lógica de negocio: su responsabilidad es coordinar los flujos de entrada/salida y mantener sincronizados los datos entre el entorno externo y el núcleo funcional.

4.1. Componentes de Entrada/Salida (I/O)

La cáscara imperativa se compone de los siguientes módulos agrupados según el tipo de interacción:

Módulo de Red (HTTP y WebSocket)

ChatService
Responsable de la comunicación en tiempo real mediante WebSocket.
Efectos: envío y recepción de mensajes a través de red.

- *connectSocket(userId: string): void*
  Abre la conexión WebSocket con el servidor y suscribe al usuario a su canal de comunicación.
  (Efecto: apertura de conexión de red).
- *sendMessage(message: Message): void*
  Envía el mensaje por el socket al backend.
  (Efecto: escritura en red).
- *onMessageReceived(callback: (msg: Message) => void): void*
  Escucha los mensajes entrantes desde el servidor y los pasa al núcleo funcional para actualización de estado.
  (Efecto: lectura de red).

ApiService
Responsable de las peticiones HTTP al backend (autenticación, carga inicial, historial,

usuarios, canales).
Efectos: comunicación REST con el servidor.

- *getInitialState(): Observable<AppState>*
  Solicita al backend los datos iniciales del usuario y sus canales.
  (Efecto: lectura desde API externa).
- *saveMessage(message: Message): Observable<void>*
  Envía al servidor un mensaje recién generado para su persistencia.
  (Efecto: escritura en base de datos remota).
- *updateUserPreferences(userId: string, prefs: Map<string, any>): Observable<void>*
  Actualiza las preferencias del usuario en la base de datos remota.
  (Efecto: escritura en red).

Módulo de Presentación (Interfaz Gráfica)

ChatComponent
Responsable de mostrar las conversaciones y permitir al usuario enviar mensajes.
Efectos: interacción con el usuario.

- *ngOnInit()*
  Inicializa la conexión del usuario, carga el historial desde el núcleo y se suscribe a eventos de nuevos mensajes.
  (Efecto: inicialización de UI).
- *onSend()*
  Captura el mensaje del campo de texto, llama al núcleo (applySendMessage) para generar el nuevo estado, y luego ordena al ChatService enviarlo al servidor.
  (Efecto: lectura de input y escritura en red).

ChannelListComponent
Responsable de mostrar los canales disponibles y permitir unirse o crearlos.
Efectos: interacción visual.

- *onJoinChannel(channelId: string)*
  Llama a la función pura joinChannel() del núcleo, luego comunica el cambio al backend vía ApiService.
  (Efecto: escritura en red).
- *onCreateChannel()*
  Solicita al núcleo crear un nuevo canal (createChannel) y luego persiste la información por HTTP.
  (Efecto: escritura en red).

Módulo de Persistencia Local

StorageService
Encargado de almacenar información temporal (por ejemplo, sesión del usuario o mensajes pendientes).
Efectos: acceso al almacenamiento del navegador.

- *saveSession(user: User): void*
  Guarda los datos de sesión del usuario en localStorage.
  (Efecto: escritura en sistema local).
- *loadSession(): User | null*
  Recupera los datos de sesión guardados.
  (Efecto: lectura de sistema local).
- *cacheMessages(conversationId: string, messages: Message[]): void*
  Almacena mensajes localmente cuando no hay conexión.
  (Efecto: escritura local).

Módulo de Notificaciones

NotificationService
Responsable de mostrar alertas, notificaciones push y mensajes de sistema.
Efectos: interacción con el sistema del navegador.

- *showMessageNotification(message: Message): void*
  Muestra una notificación visual (toast o push) al usuario.
  (Efecto: escritura en interfaz del sistema).
- *showError(error: string): void*
  Presenta un mensaje de error en la interfaz.
  (Efecto: escritura en pantalla).

Módulo de Registro y Monitoreo

LoggerService
Encargado de registrar eventos y errores del sistema.
Efectos: escritura en consola o sistema de logs.

- *log(event: string, data?: any): void*
  Imprime eventos en consola con contexto.
  (Efecto: escritura en stdout).
- *error(message: string, trace?: any): void*
  Registra errores críticos y los envía a un servicio de monitoreo.
  (Efecto: escritura y envío en red).

4.2. El Bucle Principal (Entry Point)

En Angular, el bucle principal no se ejecuta como un ciclo tradicional (while/for), sino como **un flujo reactivo** que coordina servicios, componentes y stores.
El archivo app.component.ts actúa como el punto de entrada del frontend, inicializando los servicios y el estado de la aplicación.

A continuación se describe la secuencia principal de ejecución:

1. **Inicio de la aplicación:**
   AppComponent se carga e invoca ApiService.getInitialState() para obtener los datos iniciales del usuario (canales, mensajes, contactos).
   (Efecto: lectura desde red).
2. **Inicialización del estado:**
   El resultado de la llamada se pasa al **núcleo funcional** para construir el AppState inicial.
   (Operación pura).
3. **Conexión en tiempo real:**
   ChatService.connectSocket(user.id) abre un canal WebSocket para recibir y enviar mensajes en tiempo real.
   (Efecto: apertura de conexión).
4. **Recepción de eventos:**
   Cada vez que se recibe un mensaje por socket, ChatService.onMessageReceived() lo entrega al núcleo mediante applySendMessage(), generando un nuevo estado inmutable.
   (Transformación pura).
5. **Actualización de la interfaz:**
   El nuevo estado se propaga a los componentes (por ejemplo, ChatComponent y ChannelListComponent), los cuales renderizan los cambios en la UI.
   (Efecto: escritura en pantalla).
6. **Envío de mensajes:**
   Cuando el usuario envía un mensaje, ChatComponent llama a applySendMessage() (núcleo) para actualizar el estado local y luego usa ChatService.sendMessage() para transmitirlo al backend.
   (Efectos: escritura en red y actualización visual).
7. **Persistencia y sincronización:**
   Si el usuario está offline, los mensajes se almacenan temporalmente con StorageService.cacheMessages() y se reenvían al recuperar conexión.
   (Efecto: lectura/escritura local).
8. **Finalización:**
   Al cerrar sesión, se limpian las suscripciones, se cierra el socket y se borra la sesión almacenada.
   (Efecto: cierre de conexión y limpieza local).

5. Interacción entre la Cáscara y el Núcleo

Descripción:
A continuación se ilustra el flujo completo de datos para una acción típica del usuario: **“Enviar un mensaje”** en la aplicación de chat *YARG Flow*.
Este flujo demuestra claramente la separación entre la **cáscara imperativa (Angular + servicios)** y el **núcleo funcional (lógica pura)**, así como cómo se comunican ambos componentes durante el ciclo de interacción.

Flujo paso a paso: “Enviar mensaje”

1. **Cáscara (Interfaz de usuario):**
   El usuario escribe un mensaje en el campo de texto y pulsa el botón **"Enviar"** dentro del componente `ChatComponent`.
2. **Cáscara (Controlador de eventos):**
   El evento `onSend()` del componente obtiene el texto del mensaje y el usuario actual desde el servicio de sesión.
   Se mantiene el **estado actual** del chat (`currentState`) en memoria (por ejemplo, el `AppState` en el store de Angular).
3. **Cáscara → Núcleo:**
   `ChatComponent` invoca la función pura del núcleo:

   ```
   newState = applySendMessage(currentState, message)
   ```

   La cáscara le pasa al núcleo:

   - El estado actual (`currentState`)
   - El nuevo objeto `message` con su ID, remitente, contenido y marca de tiempo.
4. **Núcleo (Lógica pura):**
   La función `applySendMessage()` agrega el mensaje a la estructura `messagesByConversation` y devuelve un **nuevo estado inmutable (`newState`)** con la conversación actualizada.
   El estado original **no se modifica**.
5. **Núcleo → Cáscara:**
   El `newState` es devuelto a la cáscara.
   Ningún efecto externo ha ocurrido hasta este punto: solo se han transformado datos.
6. **Cáscara (Actualización del estado):**
   El store o servicio de estado de Angular (`MessageStore` o `Signal Store`) reemplaza el estado actual por `newState`.
   La interfaz (`ChatComponent`) se actualiza automáticamente al detectar el cambio de estado.
7. **Cáscara (Efectos secundarios):**
   Se ejecutan acciones impuras con base en el nuevo estado:
   - `ChatService.sendMessage(message)` → Envía el mensaje por WebSocket al backend (efecto de red).
   - `ApiService.saveMessage(message)` → Persiste el mensaje en la base de datos remota (efecto de almacenamiento).
   - `NotificationService.showMessageNotification(message)` → Muestra un aviso visual al usuario (efecto de UI).
8. **Cáscara (Sincronización de retorno):**
   El servidor (a través de WebSocket) notifica a los demás clientes conectados.
   Cuando llega un nuevo evento `message:received`, `ChatService.onMessageReceived()` lo reenvía al núcleo mediante `applySendMessage()` y el ciclo se repite.

6. Justificación del Diseño y Reflexiones

Descripción:
Durante el desarrollo de *YARG Flow*, la adopción del patrón **Cáscara Imperativa / Núcleo Funcional** permitió una separación clara entre la **lógica de negocio pura** y los **efectos secundarios** (como comunicación en red, persistencia o renderizado de interfaz).
Esta decisión fue tomada para responder a los **requisitos no funcionales** del proyecto definidos en el documento de Scrum: mantenibilidad, facilidad de prueba y escalabilidad.

Reflexiones y decisiones clave

- **Testabilidad:**
  El núcleo funcional se diseñó sin dependencias externas. Esto permitió escribir pruebas unitarias para funciones como applySendMessage, createChannel o applyPresenceUpdate **sin necesidad de mocks ni servicios HTTP**.
  Se comprobó que, dado un estado de entrada y un mensaje, el núcleo devolvía siempre el mismo estado de salida, lo cual facilitó validar la lógica de negocio.
- **Claridad y mantenibilidad:**
  Evitar los efectos secundarios dentro del núcleo hizo que el código fuese más **predecible y fácil de razonar**.
  Por ejemplo, los errores de red o latencia no afectaban la consistencia del historial de mensajes, ya que la mutación de estado siempre ocurría primero en el núcleo y luego se persistía desde la cáscara.
- **Dificultades encontradas:**
  Inicialmente resultó tentador incluir operaciones como console.log() o llamadas HTTP dentro de funciones puras, pero eso habría roto la separación funcional.
  Fue necesario refactorizar para que todos los efectos se centralizaran en servicios de la cáscara (por ejemplo, ChatService, ApiService).
- **Beneficios concretos:**
  - Aceleración de las pruebas automatizadas (el 80 % del código funcional se podía testear sin ejecutar Angular).
  - Reutilización del núcleo en futuros clientes (por ejemplo, una versión móvil o de escritorio).
  - Facilidad para sustituir servicios externos (por ejemplo, cambiar Firebase por otra API sin tocar la lógica principal).
- **Alternativas consideradas:**
  Se evaluó implementar una arquitectura completamente orientada a servicios (microservicios independientes para chat, presencia y notificaciones), pero se descartó debido a la complejidad inicial y a que el objetivo del curso era resaltar la **pureza funcional** más que la escalabilidad extrema.
  También se descartó concentrar toda la lógica en componentes Angular, por dificultar las pruebas y el mantenimiento.

7. Conclusión

En resumen, el proyecto *YARG Flow* se desarrolló exitosamente siguiendo el patrón arquitectónico de **"Cáscara Imperativa, Núcleo Funcional"**.
Esta separación permitió cumplir los objetivos definidos en el documento de requisitos bajo metodología Scrum:
un sistema modular, predecible, mantenible y fácilmente escalable.

El **núcleo funcional** encapsula toda la lógica de negocio del chat (mensajes, canales, presencia y usuarios) a través de funciones puras, inmutables y deterministas, garantizando coherencia y facilidad de prueba.
La **cáscara imperativa**, implementada con Angular, coordina la interacción del usuario, la comunicación con el backend, la persistencia de datos y la visualización.

Este enfoque resultó en:

- Un código **más limpio y predecible**,
- **Menor acoplamiento** entre frontend, backend y lógica,
- **Pruebas más simples** y confiables,
- Y una base sólida para escalar la aplicación en futuras iteraciones del producto.

Finalmente, el uso de este patrón arquitectónico fue una **excelente experiencia formativa**: demostró en la práctica cómo la programación funcional y la separación de responsabilidades fortalecen la calidad del software, incluso en un entorno ágil como Scrum.

---

8. Apéndices (Opcional)

- **Apéndice A:** Enlace al repositorio de código: https://github.com/Brydyan/Project---Programac-nFuncional