

Easycourt

- Rubén Fernández Farelo
 - Ignacio Areal Antolín
 - Eduardo Fernández Fraga
 - Xoel Penas Sotelo
 - Estela Pillo González
 - Darío Santos Lois
 - Alejandra Gacho Mediavilla
-



Índice

1. Requisitos funcionales

2. Requisitos no funcionales

3 . Diagramas de casos de uso

4 . Arquitectura

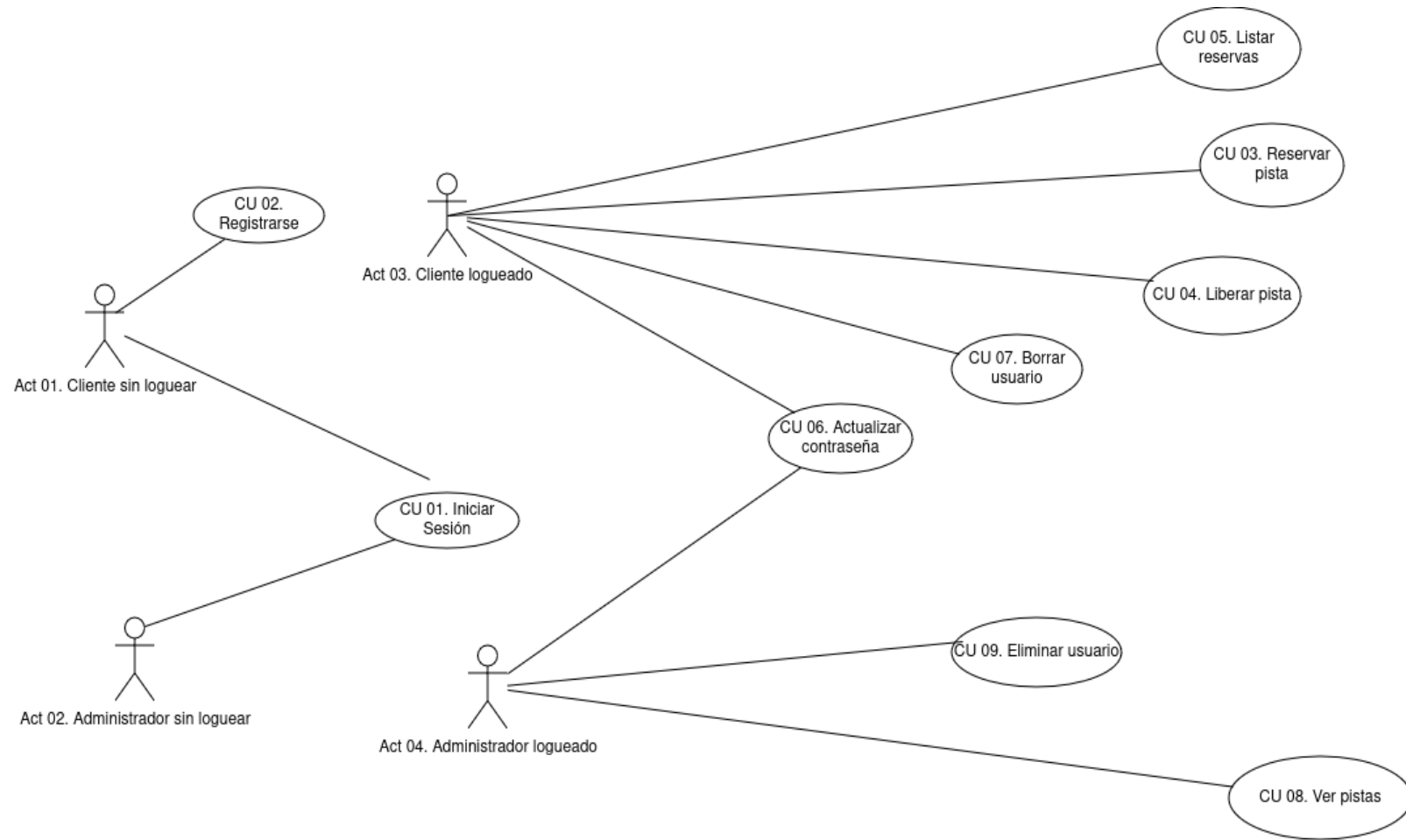
5. Diagramas C4

6. Decisiones de diseño

7. Tests de la aplicación

8 .Demostración

Diagrama de casos de uso



Requisitos Funcionales (Cliente)

Registrarse -> Permite a un nuevo usuario crear una cuenta en la aplicación

Iniciar Sesión -> Permite al usuario autenticarse en el sistema

Reservar Pista -> Permite al usuario reservar una pista

Liberar Pista -> Permite al usuario cancelar una reserva

Listar Reservas -> Permite al usuario visualizar todas las reservas activas que ha realizado

Actualizar Contraseñas -> Permite al usuario cambiar su contraseña para iniciar sesión.

Borrar Usuario -> Permite al usuario eliminar sus credenciales de inicio de sesión

Requisitos Funcionales (Admin)

Iniciar Sesión -> Permite al usuario autenticarse en el sistema

Actualizar Contraseñas -> Permite al usuario cambiar su contraseña para iniciar sesión.

Eliminar Usuario -> Permite al administrador eliminar los credenciales de un usuario de la aplicación

Ver pistas -> Permite al administrador ver el estado de las pistas.



Requisitos no funcionales

- Escalabilidad
- Tolerancia a fallos
- Seguridad
- Usabilidad
- Mantenibilidad



Escalabilidad

- El sistema debe de ser capaz de manejar un aumento en el número de usuarios y en el número de pistas.
- Módulo supervisor.
- Incorporación de diferentes bases de datos: (Usuarios | Pistas).



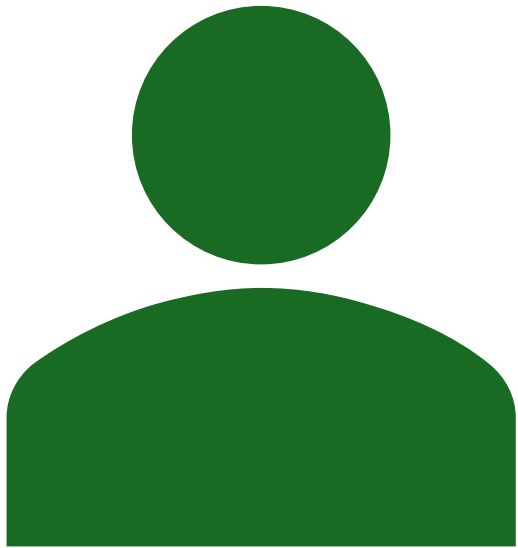
Tolerancia a fallos

- Capacidad de un sistema para continuar operando correctamente, incluso ante fallos o errores en algunos de sus componentes.
- Uso del modulo MiSupervisor.



Seguridad

- Implementación de un encriptado de las contraseñas.
- Algoritmo de encriptado por bloques con padding.

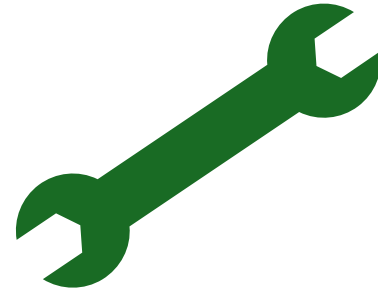


Usabilidad

- Arquitectura modular
- Interfaces claras para el cliente y el administrador
- Interacción sencilla y eficiente con el sistema.
- Fácil comprensión y mantenimiento del proyecto.

Mantenibilidad

- Diseño modular y separación de responsabilidades
- Código limpio y legible
- Pruebas automatizadas
- Documentación completa
- Uso de herramientas de control de versiones
- Gestión de dependencias



Cliente-Servidor

El cliente interactúa con el servidor a través del menú, donde selecciona la acción que quiere realizar. El servidor maneja esas solicitudes, con ayuda del directorio, e interactúa con las bases de datos, respondiendo al cliente con los resultados.



MODELO CLIENTE SERVIDOR

Diagramas

C4:

Contexto

Diagrama de Contexto - Aplicación de Reservas de Pádel



Diagramas

C4:

Contenedor

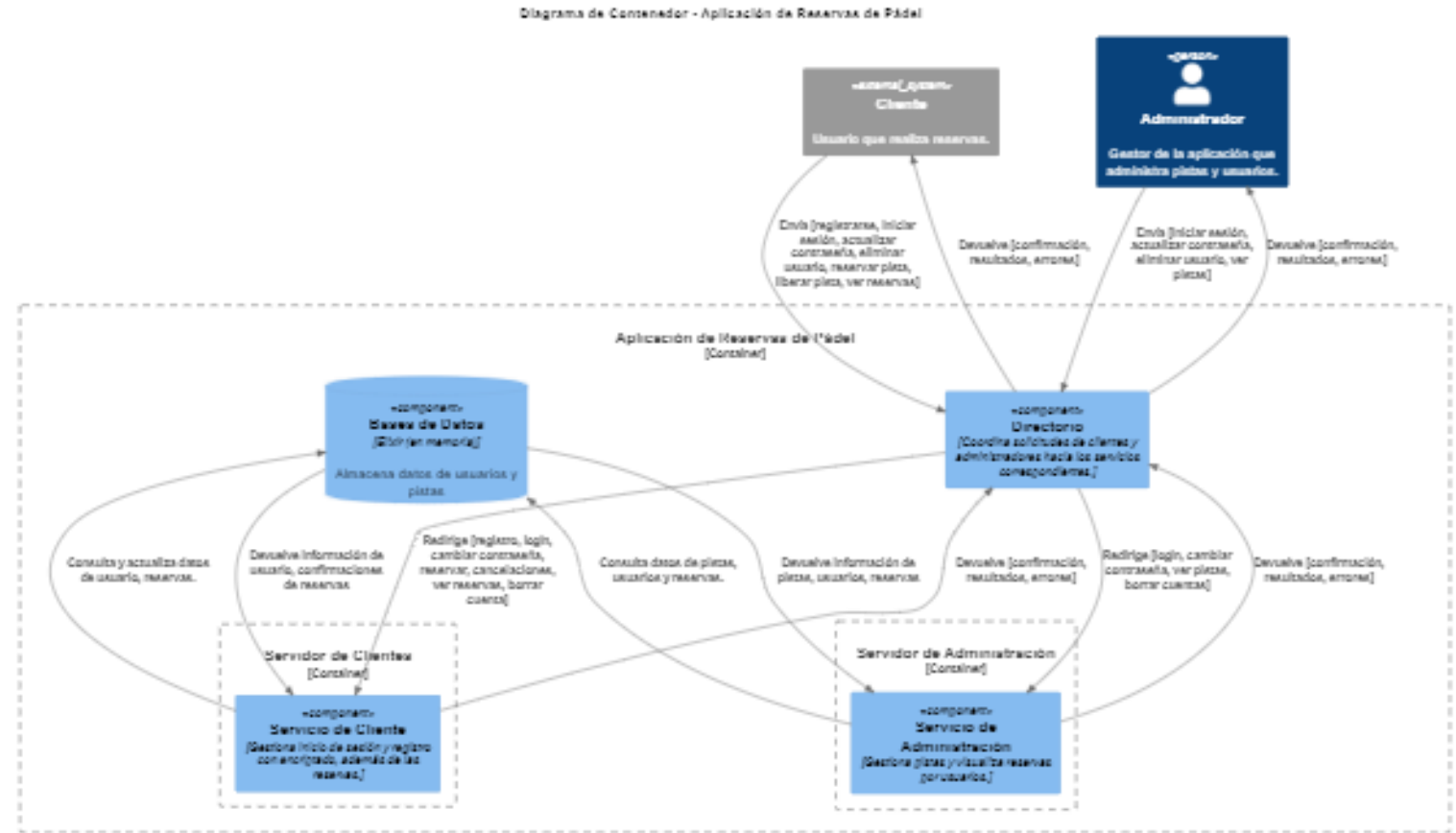


Diagrama C4: Componentes

Diagrama de Componentes - Servidor de Clientes

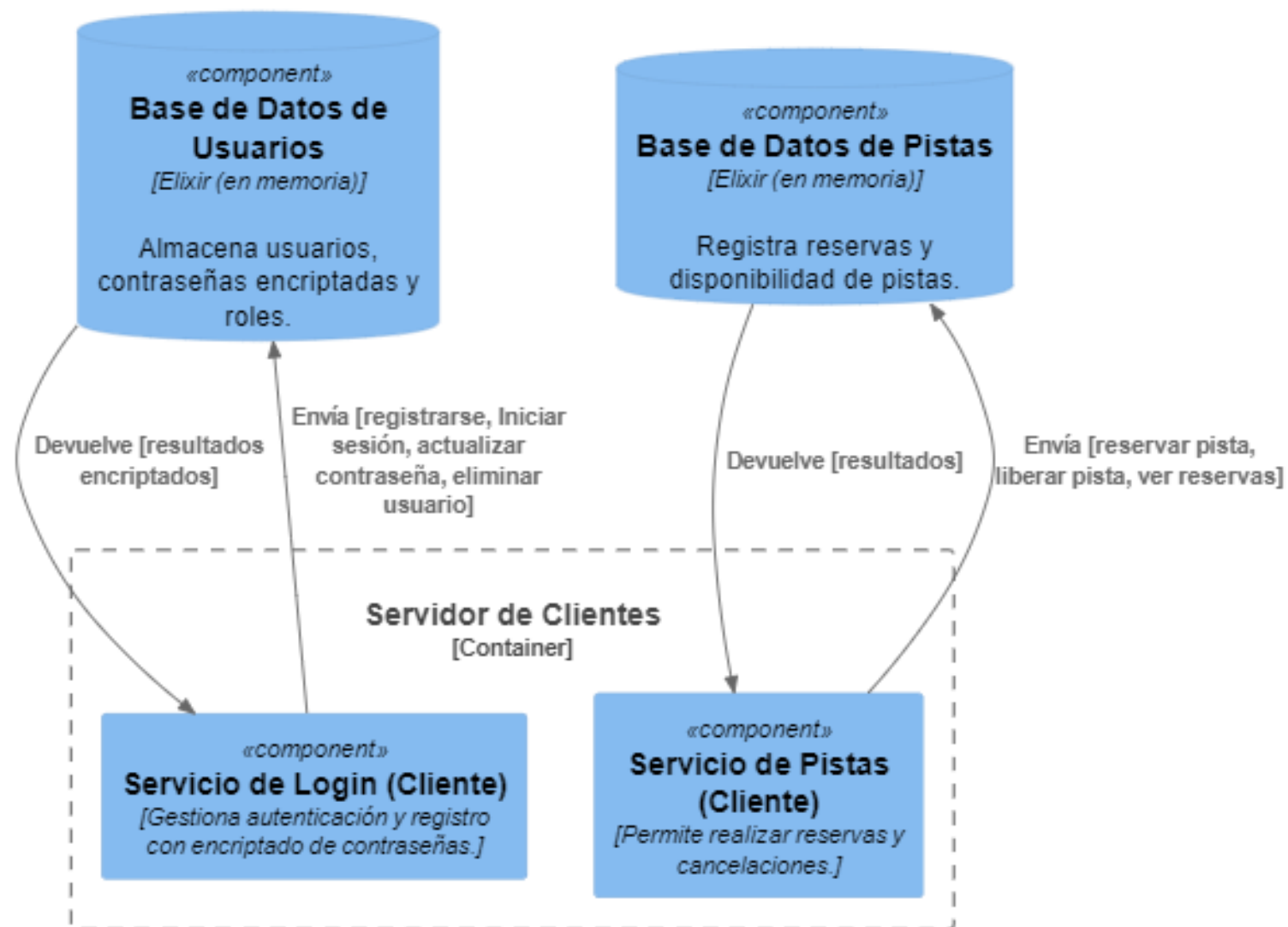
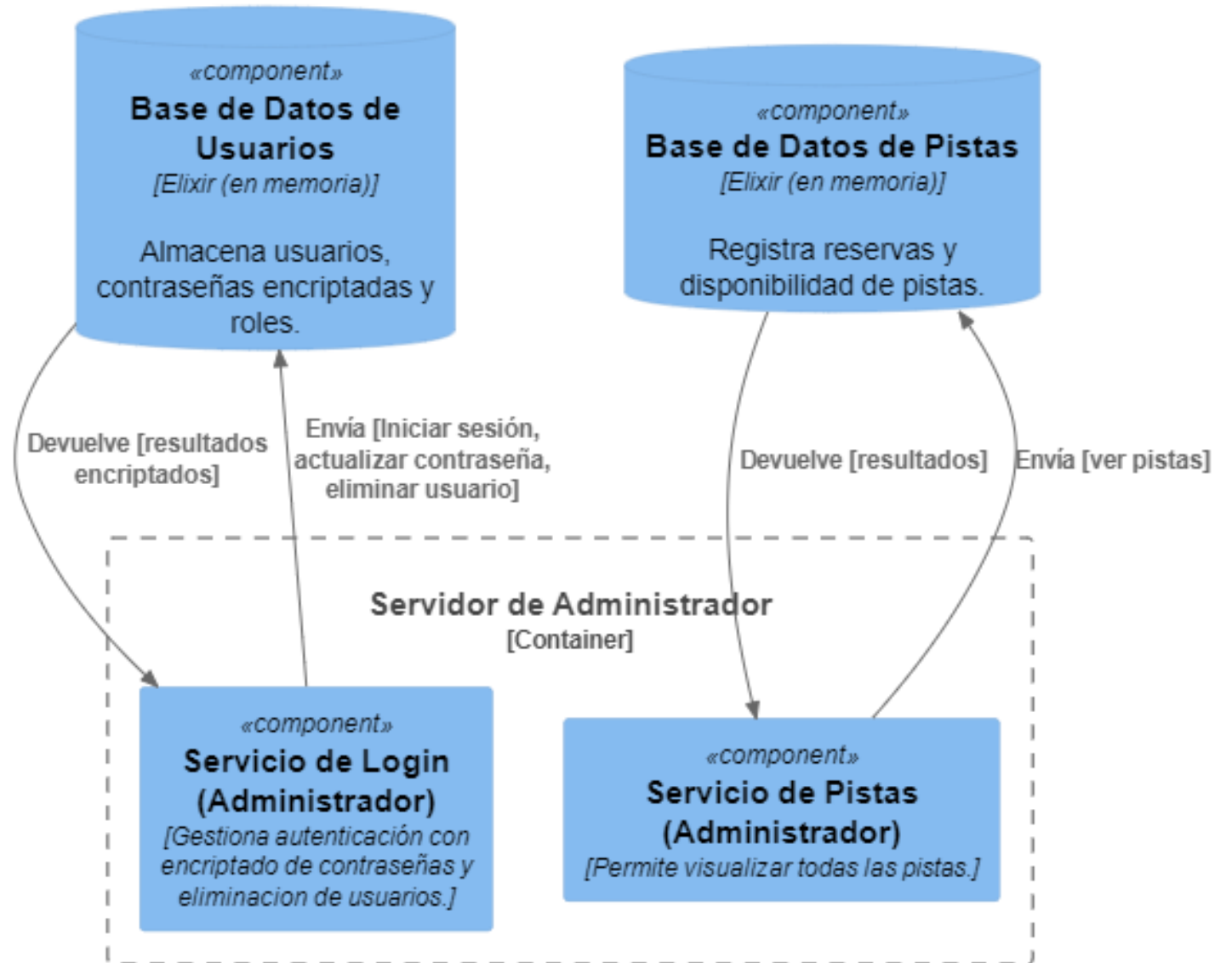


Diagrama C4: Componentes

Diagrama de Componentes - Servidor de Administrador





Decisiones de diseño

ADR-01: Incorporar el encriptado de las contraseñas

Decisión:

- Encriptar contraseñas usando un algoritmo de cifrado con padding.
- Claves y vectores de inicialización (IV) gestionados de manera segura.
- Codificación en Base64 para almacenamiento/transmisión.



Decisiones de diseño

- **ADR-02: Incorporar dos tipos de usuarios**
- **Decisión:**
- Cliente:
 - Gestiona sus propias reservas.
 - Acceso limitado.
- Admin:
 - Gestiona usuarios y visualiza todas las reservas activas.
- Implementado con un sistema de roles en la base de datos.



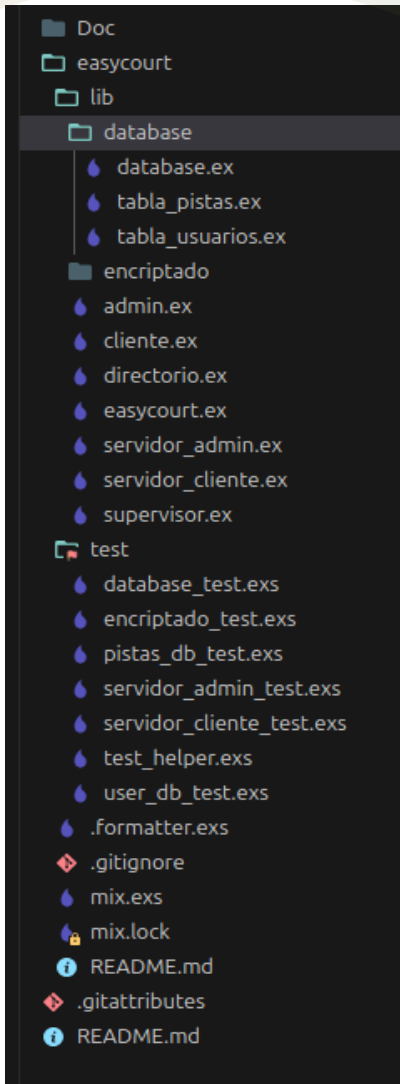
Decisiones de diseño

- **ADR-03: Diferenciar espacios en la base de datos**
- **Decisión:**
- Separar en dos espacios lógicos:
 - **Usuarios:** Credenciales y datos personales.
 - **Pistas:** Información de reservas, horarios y disponibilidad.
- Esquemas separados en el mismo sistema de gestión de bases de datos.

ESTRUCTURA DEL REPOSITORIO

La estructura del repositorio es:

- **Documentación (Doc):** contiene toda la documentación relevante del proyecto.
- **Easycourt:** creada con `mix new`
 - **Código (lib):** organizado en múltiples carpetas y ficheros:
 - **Base de datos (database):** contiene los distintos componentes de la base de datos.
 - **Encriptado:** contiene el fichero de encriptación.
 - **Admin**
 - **Cliente**
 - **Directorio**
 - **Easycourt**
 - **Servidor_admin**
 - **Servidor_cliente**
 - **Supervisor**
- **Test:** contiene las pruebas unitarias y de integración del proyecto.



```
@doc """
Actualiza la contraseña de un usuario.
"""
def updatePassword(pid, user, pass) do
  GenServer.call(pid, {:updatePassword, user, pass})
end
```

ELEMENTOS DESTACADOS

- **GenServer:** utilizado para implementar procesos genéricos del servidor, a través de un módulo que utiliza el GenServer y define los *callbacks* necesarios.
- El uso de GenServer en nuestro proyecto se centra en la gestión de operaciones como la autenticación de usuarios, la administración de reservas de pistas, y la actualización de contraseñas

Supervisor

```
def init(:ok) do
  # Inicializar las bases de datos
  {:ok, _} = DB.TablaUsuarios.start_link()
  {:ok, _} = DB.TablaPistas.start_link()

  # Definir los procesos hijos que serán supervisados
  children = [
    {Directorio, []},
    {ServidorCliente, [1]},
    {ServidorAdmin, [2]},
    {Cliente, []},
    {Admin, []}
  ]

  # Iniciar el supervisor con la estrategia :one_for_one
  Supervisor.init(children, strategy: :one_for_one)
end
```

El módulo **MiSupervisor** gestiona y supervisa los procesos del sistema. Si alguno de los procesos hijos falla, se reinicia automáticamente el proceso que falló. Esto proporciona robustez y tolerancia a fallos.

TESTS REALIZADOS

Primero de todo se inicializan los servidores sobre los que se van hacer pruebas, así como las bases de datos.

Para comprobar el correcto funcionamiento de la aplicación realizamos distintos test automatizados y unitarios, de tal forma que unos no afecten al estado del otro.

Comprobamos el funcionamiento del encriptado, del funcionamiento de las bases de datos y de los servidores. Se probarán así el correcto uso de los requisitos funcionales.

Para ejecutar los test se ejecutaría mix test para ejecutar todos los que se encuentran dentro de la carpeta test

```
test "password" do
  pass_encriptada = Encriptado.encriptar("password")
  assert Encriptado.desencriptar(pass_encriptada) == "password"
end
```

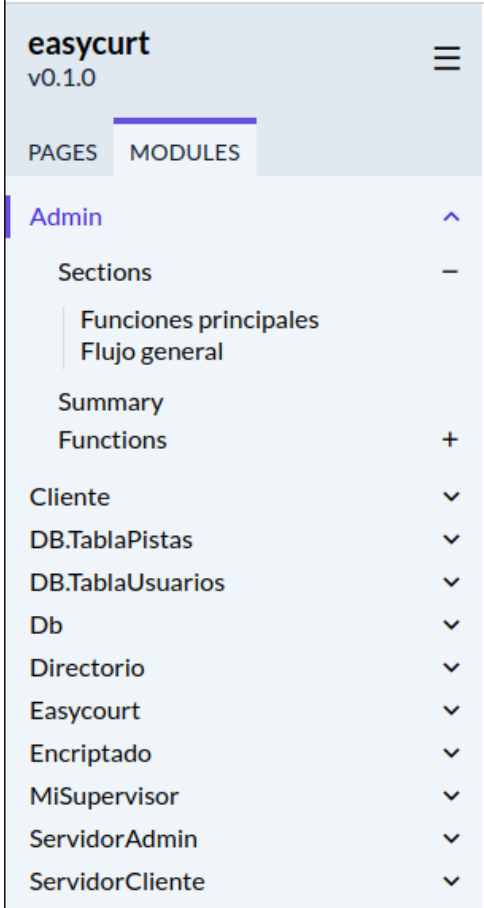
```
test "actualizar contraseña de un usuario", %{admin_pid: admin_pid} do
  assert {:ok, "usuario4"} == ServidorAdmin.signIn(admin_pid, "usuario4", "pass4")
  assert {:updated} == ServidorAdmin.updatePassword(admin_pid, "usuario4", "new_pass4")
  assert {:ok, "usuario4"} == ServidorAdmin.checkPassword(admin_pid, "usuario4", "new_pass4")
end
```

```
test "reservar pista no existente" do
  assert {:ok, "usuario1"} = TablaUsuarios.addUser("usuario1", "pass1", "roll")
  assert {:error, :pista_no_existente} = TablaPistas.reservar_pista("usuario1", 25)
end
```

```
test "registrar un usuario y verificar su existencia", %{admin_pid: admin_pid} do
  assert {:ok, "usuario1"} == ServidorAdmin.signIn(admin_pid, "usuario1", "pass1")
  assert {:user_exists} == DB.TablaUsuarios.existsUser("usuario1")
end
```

DOCUMENTACIÓN

- **ExDoc:** genera documentación a partir de comentarios estructurados en el código, como @moduledoc y @doc
- Para generar la documentación, simplemente utilizar el comando **mix docs** en el terminal. Esto creará un índice organizado de los módulos de tu proyecto, facilitando la navegación y comprensión del mismo.



easycourt v0.1.0		≡
PAGES	MODULES	
Admin		^
Sections		-
Funciones principales		
Flujo general		
Summary		
Functions		+
Cliente		v
DB.TablaPistas		v
DB.TablaUsuarios		v
Db		v
Directorio		v
Easycourt		v
Encriptado		v
MiSupervisor		v
ServidorAdmin		v
ServidorCliente		v



DEMOSTRACIÓN

Antes de hacer `Easycourt.start_cliente()` o `Easycourt.start_admin()` hay que hacer un `Easycourt.start(:normal, [])`

https://drive.google.com/drive/folders/1z9Wj7sNLUHE_UxuOox1IXhvM2QLH3gCj?usp=sharing