Aplicación de Separación de Responsabilidades (SoC) en el desarrollo de servicio web y aplicación móvil

Badillo L. Antonio[#1], Gamboa T. Brayan[*2], Ojeda I. Gabriel[#3]
Desarrollo de Software, Universidad Católica Andrés Bello, Montalbán, Caracas, Venezuela
[1]aabadillo.18@est.ucab.edu.ve, [2]bjgamboa.19@est.ucab.edu.ve, [3]giojeda.18@est.ucab.edu.ve

***Keywords*— Desarrollo de Software, Diseño Orientado a Objetos, Typescript, Dart, Flutter, NestJS**

RESUMEN

Este short paper detalla el proceso de diseño e implementación de una serie de requerimientos relacionados al proyecto de telemedicina propuesto por la materia de Desarrollo de Software.

El objetivo de este documento es el de exponer el diseño, implementación y deficiencias de dos sistemas - una aplicación móvil en Flutter, y un servicio web en NestJS. El diseño de ambos sistemas busca seguir los principios de diseño SOLID, y el principio de Separación de Responsabilidades (Soc).

Introducción

El siguiente trabajo se basa en el diseño, iteración e implementación de dos diagrama de clases que sirven como solución a un problema, el cual es una aplicación móvil que nos muestre una lista de doctores, que a su vez puedan filtrarse según la especialidad.

Para poder desarrollar una solución a los diferentes requerimientos del problema, se utilizan diferentes metodologías y principios de diseño de software.

A lo largo del documento, se hará referencia a las arquitecturas incluidas en los anexos.

Desarrollo

A. El diseño

El diseño de ambos sistemas está guiado fuertemente por el Principio de Separación de Responsabilidades[1], en el cual ambos diagramas, tanto el de servicio como el de aplicación móvil se diseñaron en base a la separación Horizontal, en la que se dividen en tres capas cada una.

A.1. Aplicación Móvil

La aplicación móvil se divide en tres capas donde la primera es la Presentation Layer, la cual contiene todo acerca de cómo se ve la aplicación en el teléfono, y nos muestra la información que solicitamos, que en este caso sería la lista de doctores y un filtro donde podemos escribir la especialidad de los doctores que deseemos buscar, al estar trabajando en Flutter todo lo que nos muestra en esta capa serian Widgets.

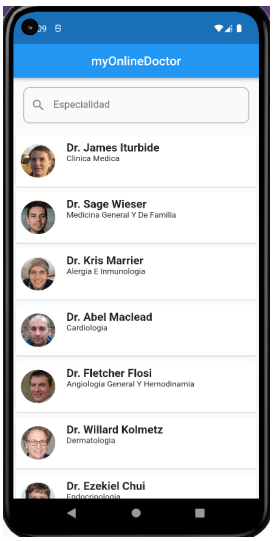


Figura 1: Vista de listado de doctores.

La segunda capa sería la de Business Layer en la cual se abarca el modelo, la lógica de negocio y el flujo de trabajo.

En esta capa utilizamos el patrón Bloc[3] el cual es un patrón que desacopla nuestro código y nos permite trabajar con eventos y estados donde la Presentation Layer nos da eventos y le devolvemos estados.

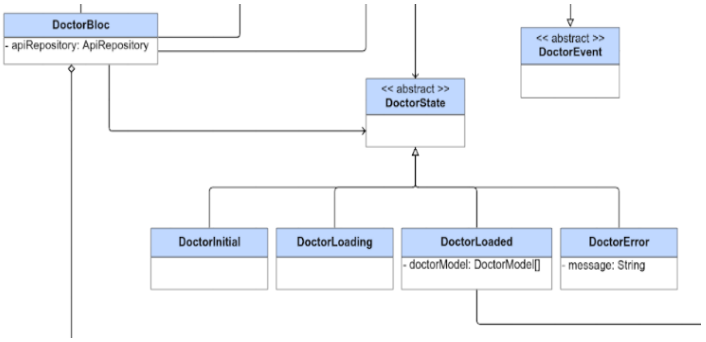


Figura 2: Estados del DoctorBloc.

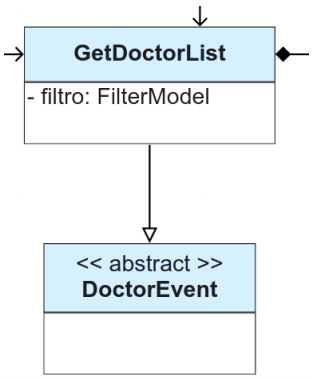


Figura 3: Eventos del DoctorBloc

En nuestra implementación el DoctorBloc recibe eventos y devuelve estados, esto cumple con OCP ya que simplemente podemos crear más estados o eventos según convenga en nuestra aplicación, a su vez el DoctorBloc es el que se comunica con la Presentation Layer.

En este caso el evento seria GetDoctorList, es decir obtener las lista de los doctores, donde dependiendo de lo que devuelve la capa de Resource Access Layer nuestro DoctorBloc devolverá diferentes estados, donde el estado DoctorLoaded nos devuelve la lista de doctores a presentar en la capa de Presentation Layer, mientras que los otros estados son cuando esta cargando la lista u ocurrio algun error.

En esta capa también se encuentra el DoctorModel el cual es el que guarda todos los atributos que pertenecen a un Doctor.

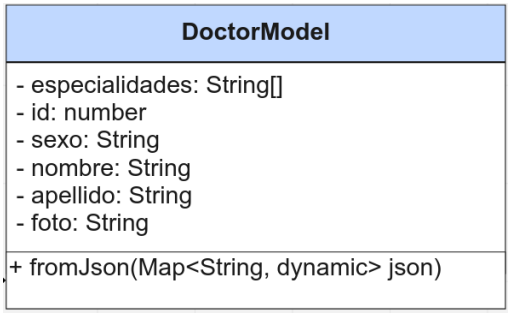


Figura 5: DoctorModel.

La tercera y última capa es la Resource Access Layer que es la capa que nos da la información de la API.

En esta capa se utilizó el patrón asíncrono async/await[3] que se utiliza para la comunicación con la API, donde en ApiRepository se utiliza el FilterModel<T> el cual es una clase genérica que actúa como un filtro cuando se busca la especialidad en la API.

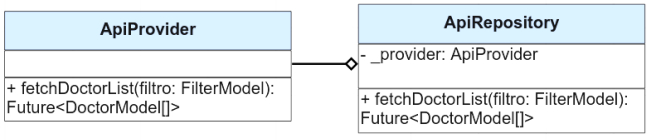


Figura 6: Resource Access Layer.

A.2. Servicio Web

Al igual que la aplicación móvil, el servicio creado en NestJS se divide en tres capas, cada una englobando comportamientos u objetivos comunes que existen entre los objetos que las componen.

La primera capa sería la de Service Interface Layer; en esta capa podemos observar que se encuentran controladores, los cuales serán los encargados de definir las rutas a las cuales se les hará un request del tipo Http en el cual nos llegará una promesa proveniente en este caso de la aplicación móvil a la cual hay que darle respuesta. A su vez como los controladores su principal responsabilidad es atajar el request y generar una nueva promesa (que mediante se pasen de capas se irán anidando) la cual obtendrá respuesta de un handler que se ubicará en la segunda capa y transformará esa respuesta a formato JSON y así resuelve el request HTTP pendiente.

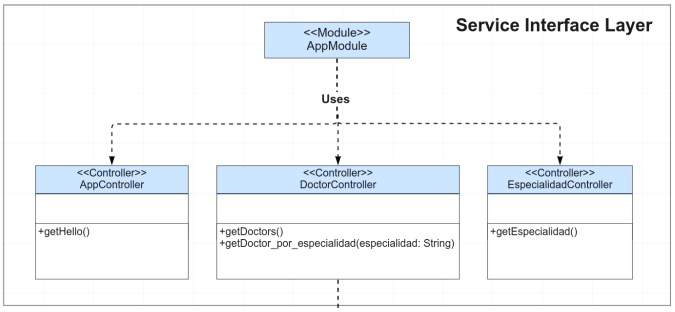


Figura 7: Service Interface Layer Nestjs

La segunda capa corresponde a la Business Layer, donde la responsabilidad de la capa será lo que corresponda a toda la lógica de manipulación de datos y su transformación para dar respuesta a la promesa iniciada en la primera capa.

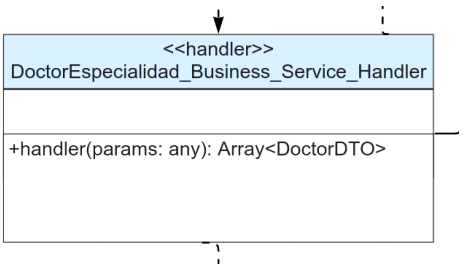


Figura 8: Ejemplo handler doctor filtrado por especialidad

Para lograr esto la promesa es atajada por el handler que corresponda al tipo de petición, donde él mismo como primer paso iniciará una nueva promesa que será respondida por la tercera capa. Luego de obtener la respuesta de la tercera capa, el handler necesitará procesar la data antes de responder la promesa pendiente a la primera capa, para lo que delega el procesamiento a un ORM el cual mapeara la data y la transformará en un DTO (Data Transfer Object) que servirá solo de lectura para luego enviar un arreglo del DTO correspondiente como respuesta a la promesa pendiente de la primera capa.

En la tercera capa, la de Resource Access Layer, se engloba la lógica de obtención de datos (en este caso, directo de una

base de datos PostgreSql) donde está representado la conexión con la base de datos desde una clase abstracta que hereda la conexión a diferentes clases que a su vez representan las diferentes consultas predeterminadas que se pueden hacer para responder las promesas hacia la segunda capa.

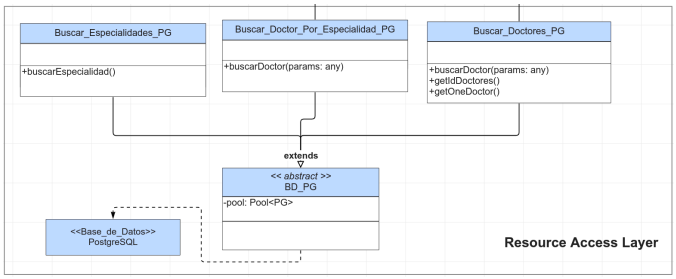


Figura 9: Resource Access Layer.

Los handlers de la segunda capa delegan a la clase correspondiente que alberguen la consulta que se desea hacer a la base de datos y así obtener la data en crudo, es decir sin procesar y responder la solicitud pendiente en el handler. En caso que se quiera agregar una base de datos diferente, se debe crear una nueva clase abstracta para heredar el nuevo tipo de conexión. También es recomendable crear una interfaz que contenga un método *conexión()* el cual se sobreescriba en las clases abstractas para inicializar la conexión a la base de datos que se desee. En el anexo 4 se puede apreciar el diagrama ER que representa la base de datos que se utilizó en PostgreSQL.

B. La implementación

A continuación se describen cuáles aspectos se lograron implementar en Flutter y en Nestjs, así como los que no.

B.1. Flutter

- Se implementaron los widgets necesarios para poder hacer la lista de doctores que nos devuelve la API.
- Se implementaron los widgets necesarios para poder hacer una barra de búsqueda que sirva para escribir la especialidad y al hacer enter poder realizar la petición a la API para que devuelva la lista de doctores filtradas por especialidad.

B.2. NestJS

- Se implementó los ORM y DTO necesarios para procesar la data en crudo obtenida de la base de datos.
- Se implementaron las rutas necesarias para dar respuestas a las promesas generadas desde la aplicación móvil.
- Se implementó una arquitectura con contenedor Docker, la cual fue automatizada con el uso de Github Actions para lograr un despliegue automático a Heroku[2]

Conclución

Al realizar este trabajo utilizando el principio de Separación de Responsabilidades, utilizamos la separación Horizontal, pero nos dimos cuenta que en un futuro podríamos combinarla junto con la separación Vertical, donde cada participante del equipo podría desarrollar una vista con su servicio, a su vez podemos observar las ventajas que nos trajo trabajar con Bloc en la aplicación móvil, ya que es seria nos permite trabajar con un código desacoplado con el que sería más fácil de extender las clases en un futuro.

Tambíen nos dimos cuenta que como definimos el Resource Access Layer de Nestjs, podría ser más efectivo que se creara una interfaz que implementa el método de conexión a la base de datos y así al momento de definir las clases concretas que realicen las consultas, obtendrán por herencia el método de conexión generalizado pero dependiendo de quien lo hereden tendrán la conexión a la base que necesitan haciendo así que se pueda cambiar con mayor facilidad de base de datos sin mayot trabajo a crear las clases necesarias para crear la conexión sin modificar o tocar las clases que ya existen.

Referencias

[1] Derek Greer. “The Art of Separation of Concerns.” *http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/.* Aspiring Craftsman. 2008.

[2] Hirani, Shariq. “Deploy NestJS with Docker, Heroku, and GitHub Actions *”.* Bundle, LLC. *https://www.bundleapps.io/blog/nestjs-docker-heroku-github-actions-guide.* 2021.

[3] Nazar Purwandaru, Arif. “Getting Started with Flutter Bloc Pattern.” Mitrais. *https://www.mitrais.com/news-updates/getting-started-with-flutter-bloc-pattern/*. 2021.

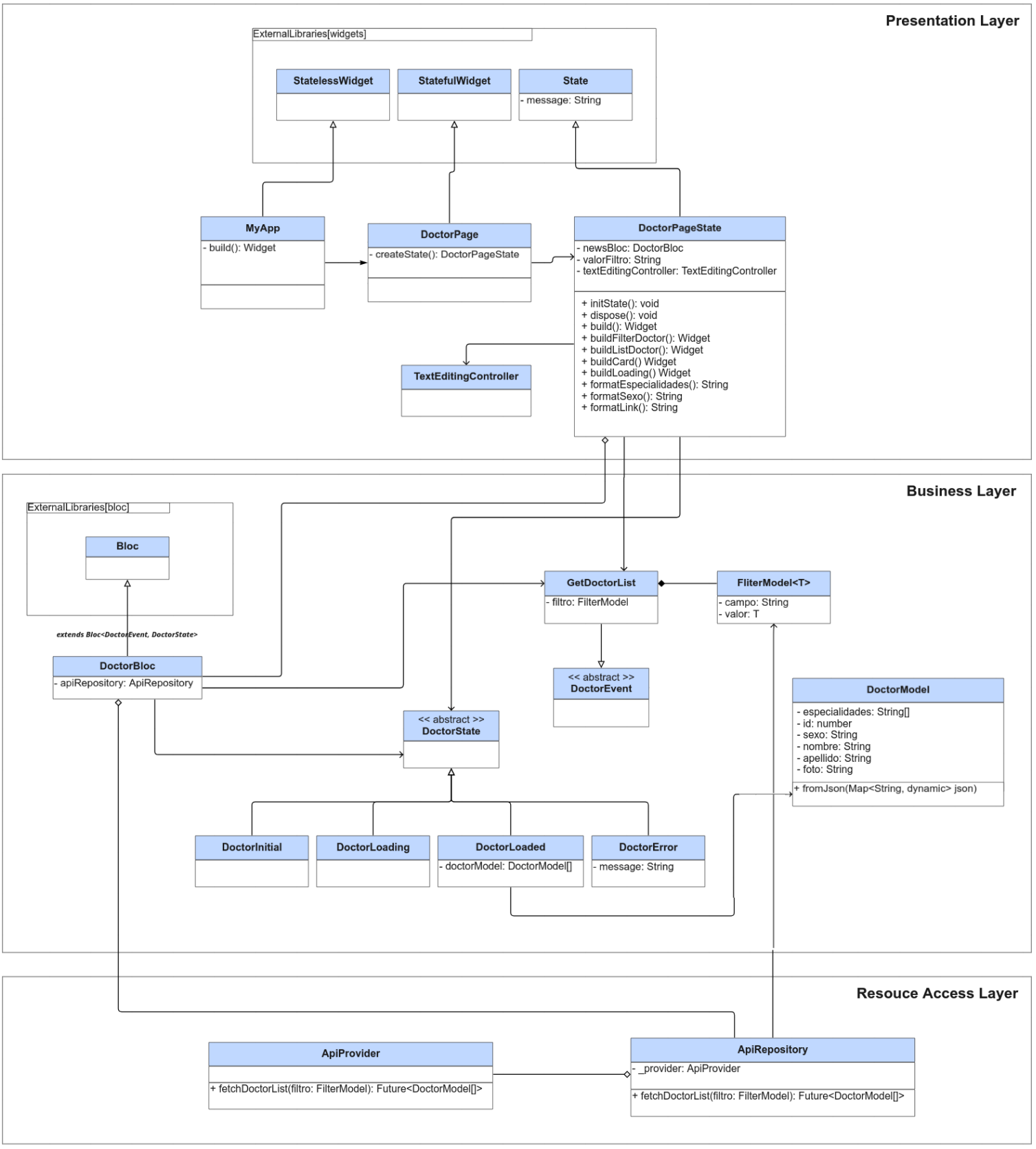
[4] Dart. “Asynchronous programming: futures, async, await.”. Dart. *https://dart.dev/codelabs/async-await.*

[5] Seemann, Mark, and Steven v. Deursen. “Dependency Injection Principles, Practices, and Patterns”. Manning. 2019.

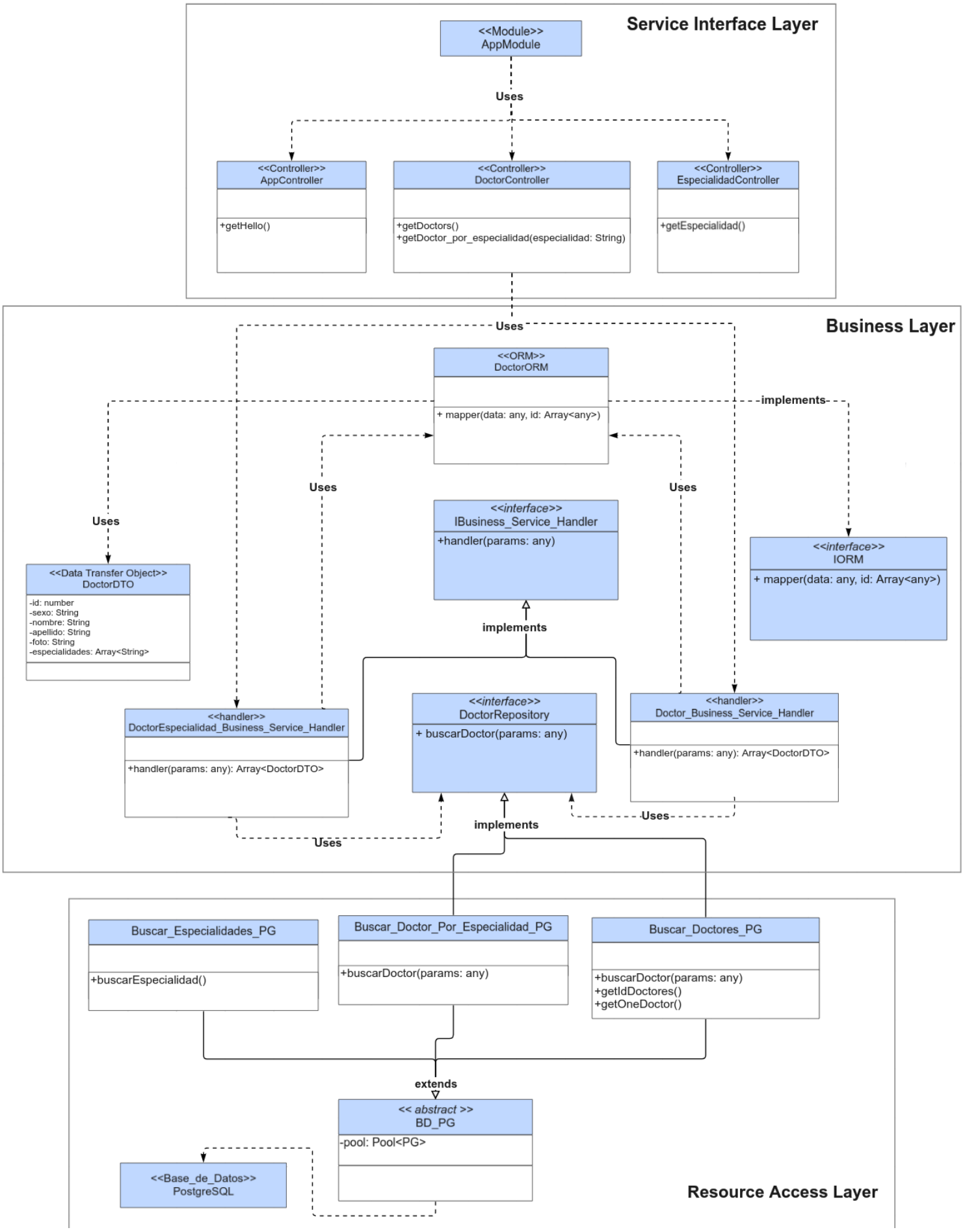
[6] Oscar Blancarte. “Data Transfer Object DTO - Patrón de diseño” *https://www.oscarblancarteblog.com/2018/11/30/data-transfer-object-dto-patron-diseno/. 2018*

Anexos

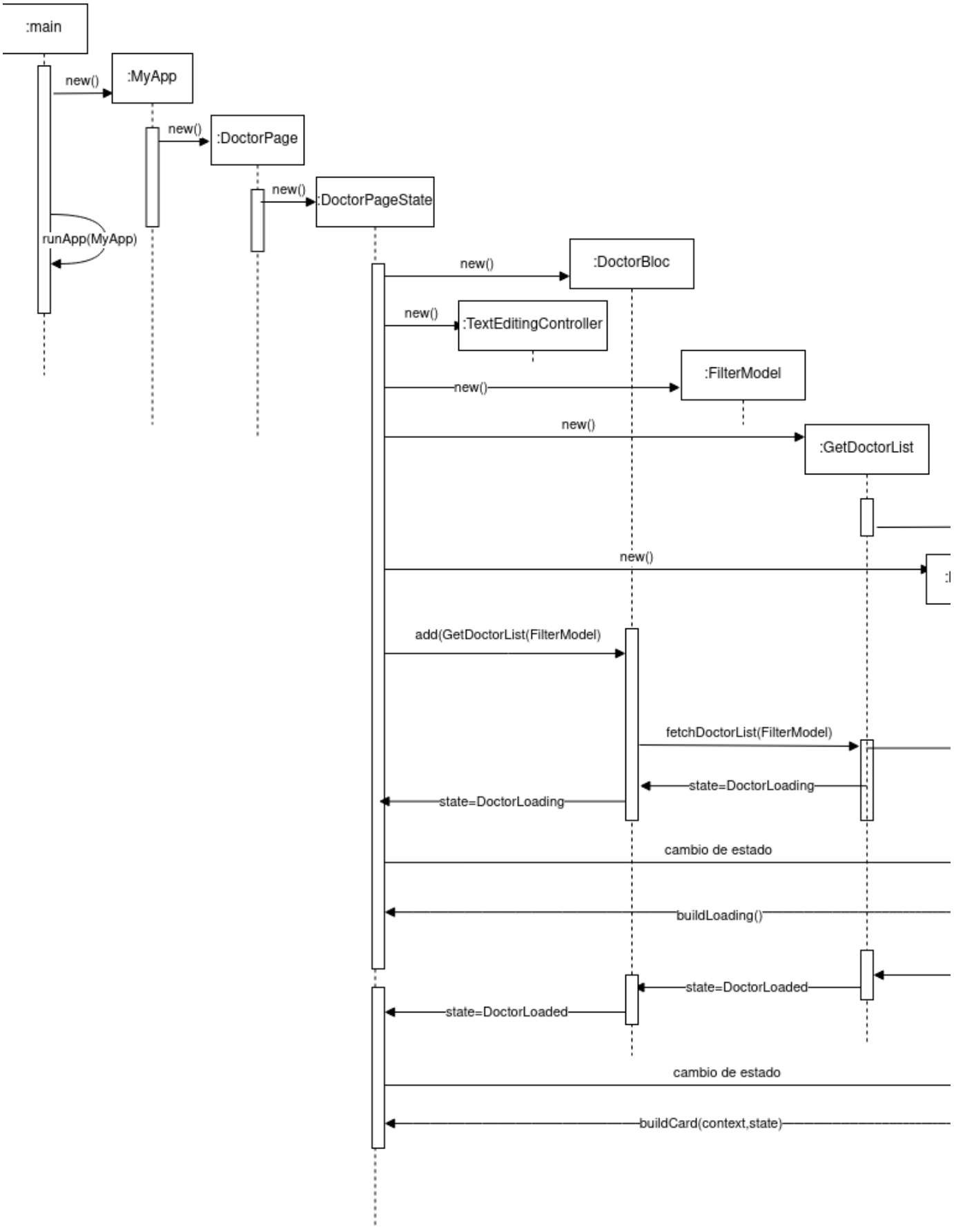
1. *Diagrama de clases UML*
    a. *Aplicación Movil Flutter*

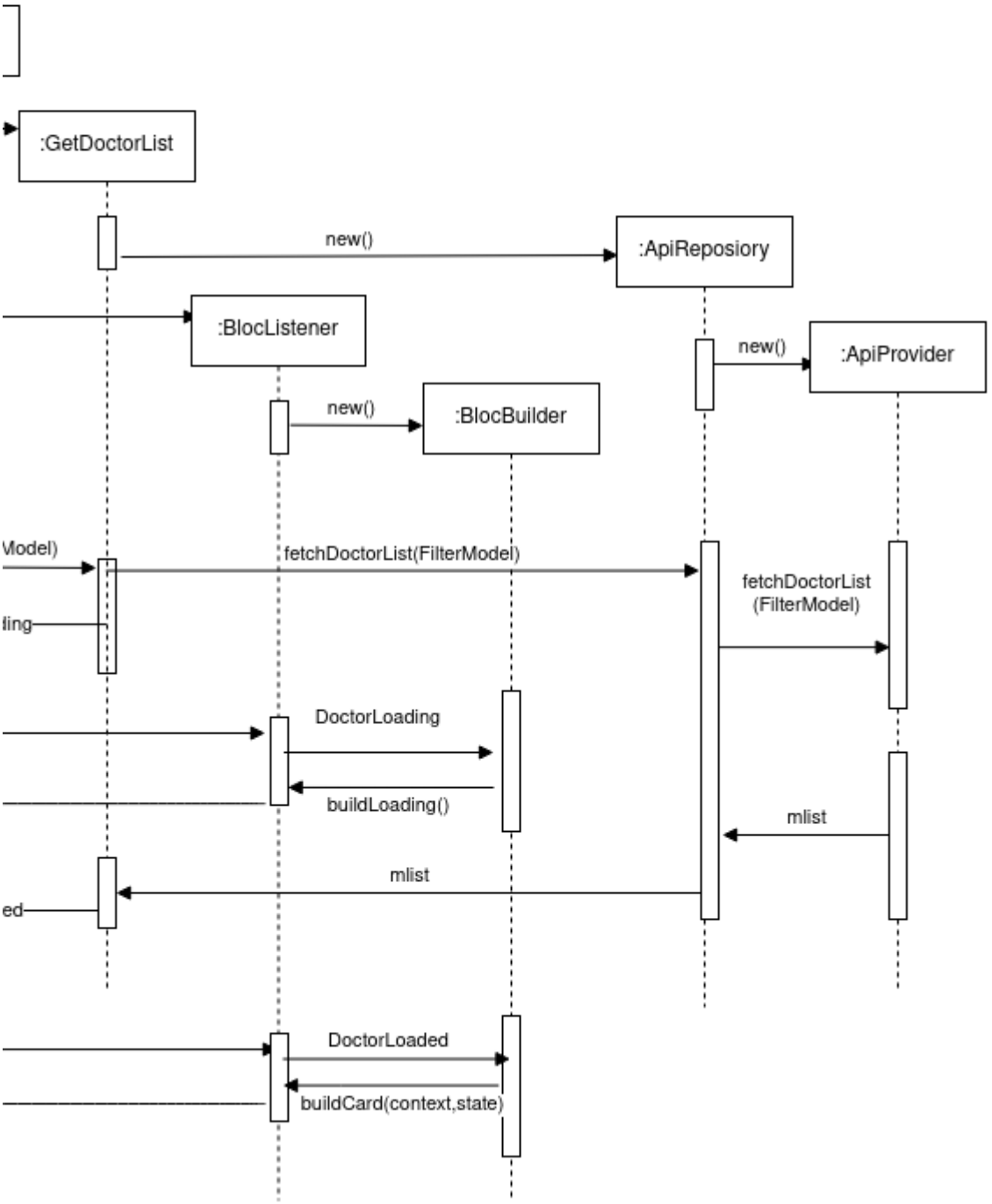


b. Servicio Web NestJS

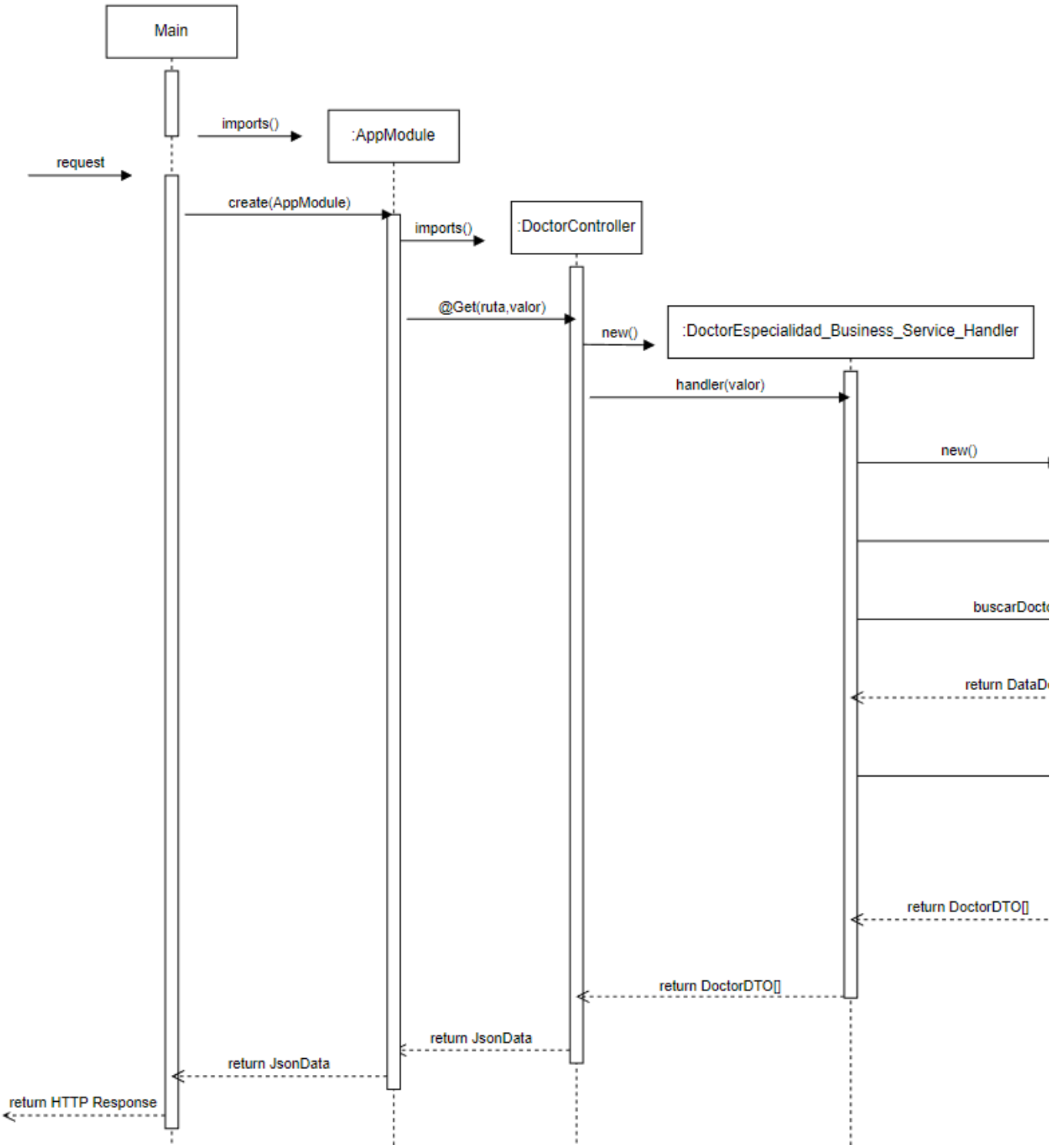


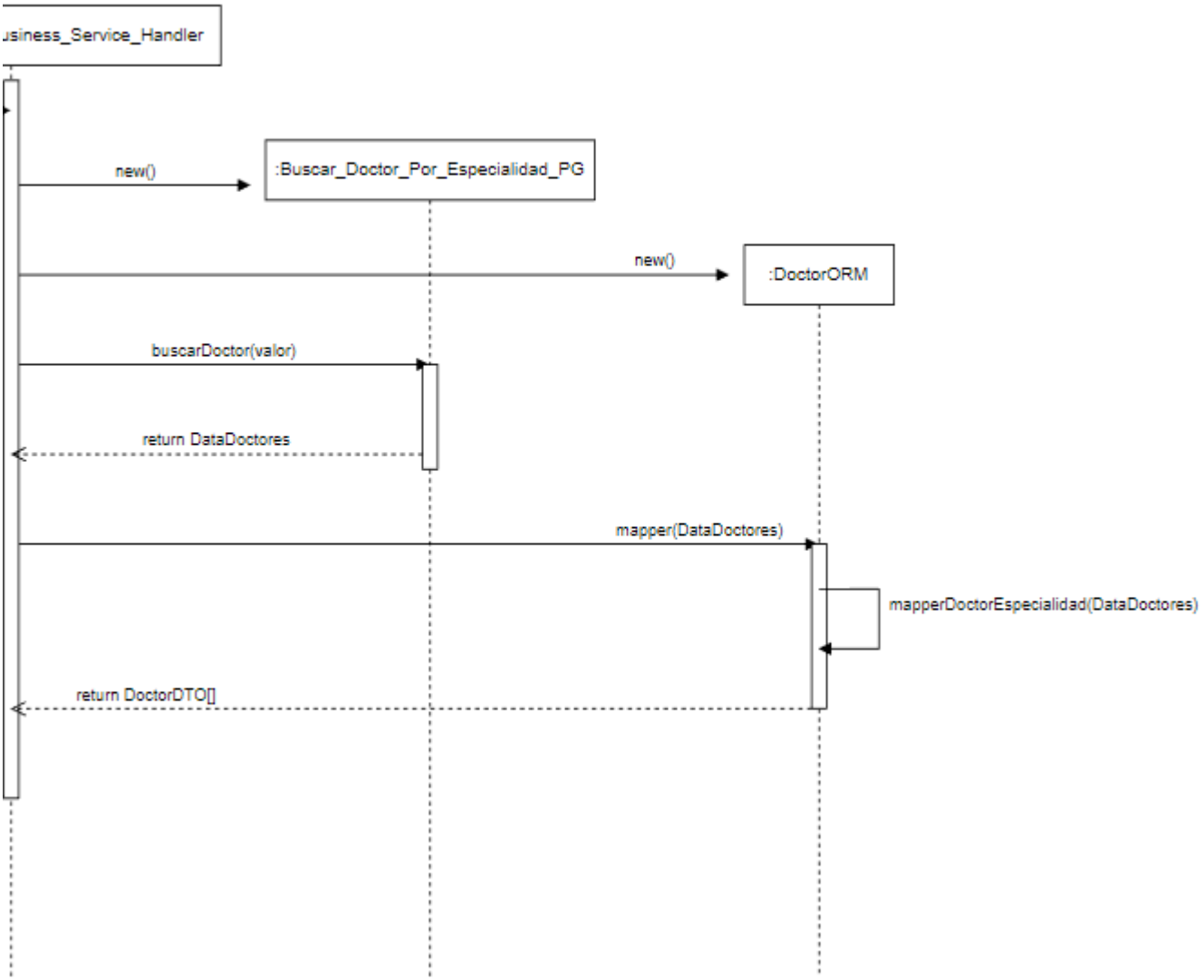
2. *Diagrama de Secuencia:*
    a. *Aplicación Móvil*



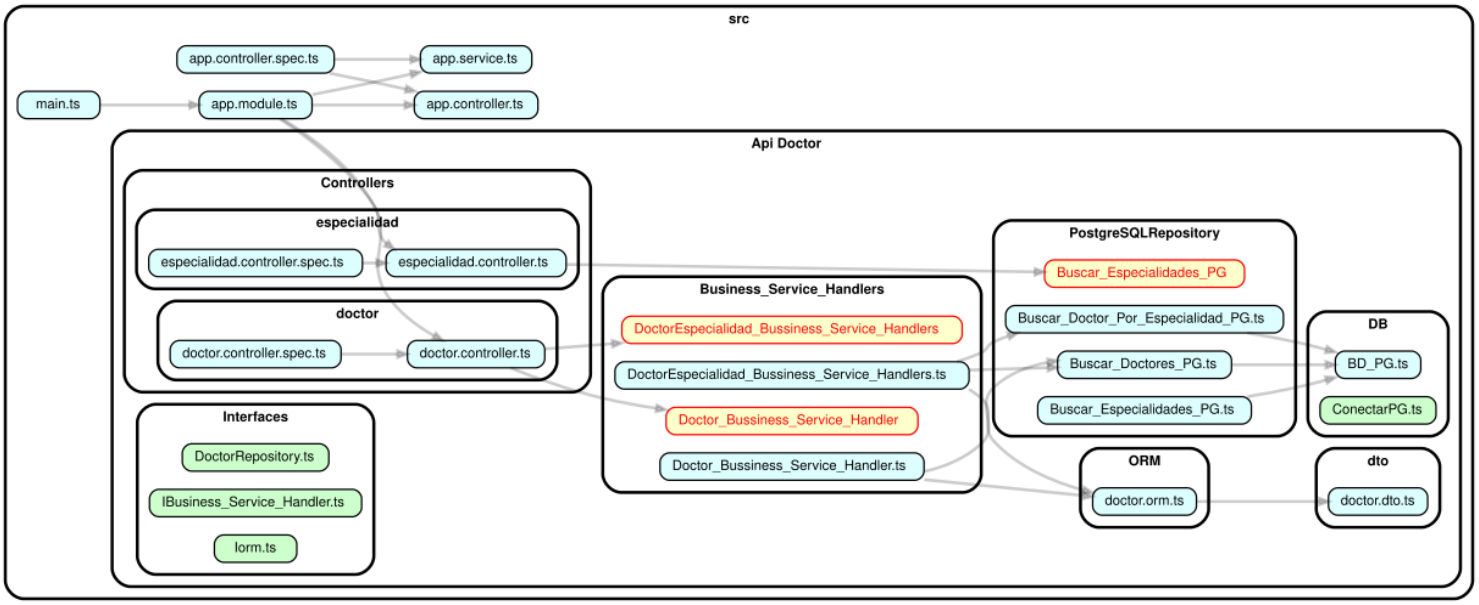

:GetDoctorList
new()
:ApiReposiory
:BlocListener
new()
:ApiProvider
new()
:BlocBuilder
Model)
fetchDoctorList(FilterModel)
fetchDoctorList
(FilterModel)
ling
DoctorLoading
buildLoading()
mlist
mlist
ed
DoctorLoaded
buildCard(context,state)

b. Servicio Web




usiness_Service_Handler
new()
:Buscar_Doctor_Por_Especialidad_PG
new()
:DoctorORM
buscarDoctor(valor)
return DataDoctores
mapper(DataDoctores)
mapperDoctorEspecialidad(DataDoctores)
return DoctorDTO[]

3. *Grafo de dependencias del servicio web*



4. *Modelo Entidad Relación*