Universidad ORT Uruguay

Facultad de Ingeniería

Arquitectura de software
Obligatorio

Santiago Toscanini (232566)
Nahuel Biladóniga (211138)
Sofia Tejerina (209239)

Entregado como requisito de la materia Arquitectura de Software

24 de junio de 2021

Resumen

El objetivo de este obligatorio es poder aplicar y reafirmar los conocimientos aprendidos en el curso de Arquitectura de Software para resolver un problema de la vida real. En este caso se nos solicitó implementar el software del backend para un sistema de vacunación de un país, basándonos en el que se utiliza actualmente en Uruguay.

Se brindaron los requerimientos del cliente, de los que obtuvimos los requisitos de calidad y en base a ellos construimos una arquitectura e implementamos un solución que buscara satisfacer las expectativas del cliente.

En este documento detallaremos todo lo relacionado a la solución, justificaciones de diseño y uso de tecnologías basándonos en el cumplimiento de los requisitos impuestos por el cliente.

El código del sistema se encuentra almacenado en Github dentro de la organización de la materia Arquitectura de Software, se puede acceder a él haciendo click aquí.

Índice general

1. Tecnologías Utilizadas

Respecto a las tecnologías utilizadas, cabe aclarar que todos los servicios corren sobre contenedores de Docker, utilizando Docker-Compose, esto nos permite abstraernos de la maquina en donde se desarrolla y lograr asegurar el correcto funcionamiento en producción sin importar configuraciones/dependencias locales.

Utilizamos multi-stage building para las imágenes de las APIs, esta optimización genera imágenes de Docker mucho más livianas.

Para crear los multiples servicios utilizamos:

- NodeJS
- Express
- Morgan
- Typegoose
- Axios
- JWT
- Nodemon
- TypeScript
    - Prettier
    - ESLint

Para guardar los datos utilizamos

- Mongo DB
- PostgreSQL
- Redis (el servicio se encuentra en el compose, aunque no se llego a implementar el caching).

Para los mocks de las APIs de registro civil y SMS utilizamos

- Phyton

- Tornado
- Pandas

Para las pruebas de carga utilizamos

- Artillery.io

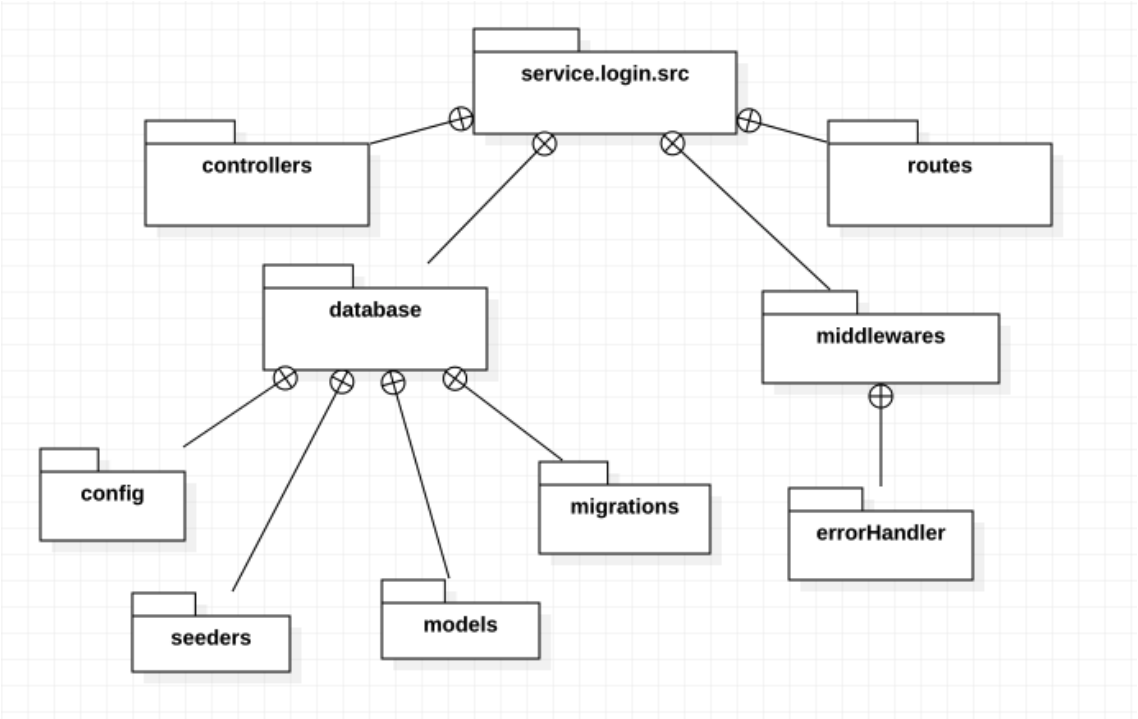
Finalmente para el balanceo de carga entre las múltiples replicas de los servicios de Docker utilizamos

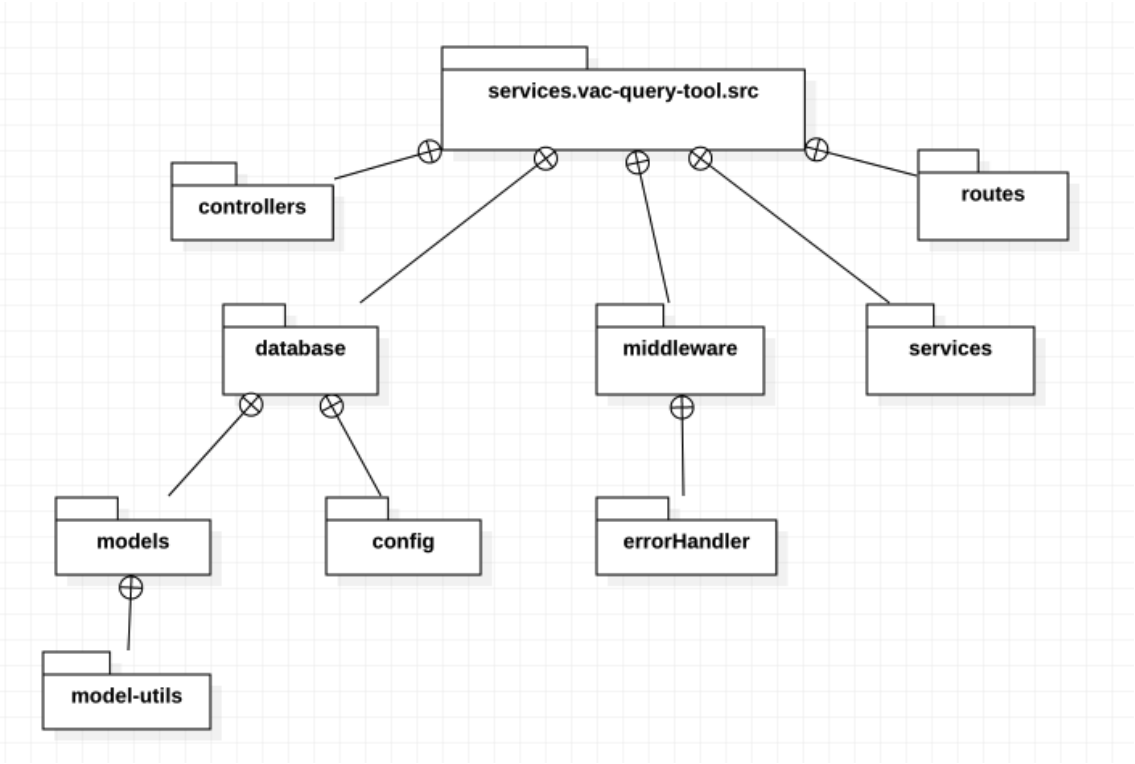
- Nginx

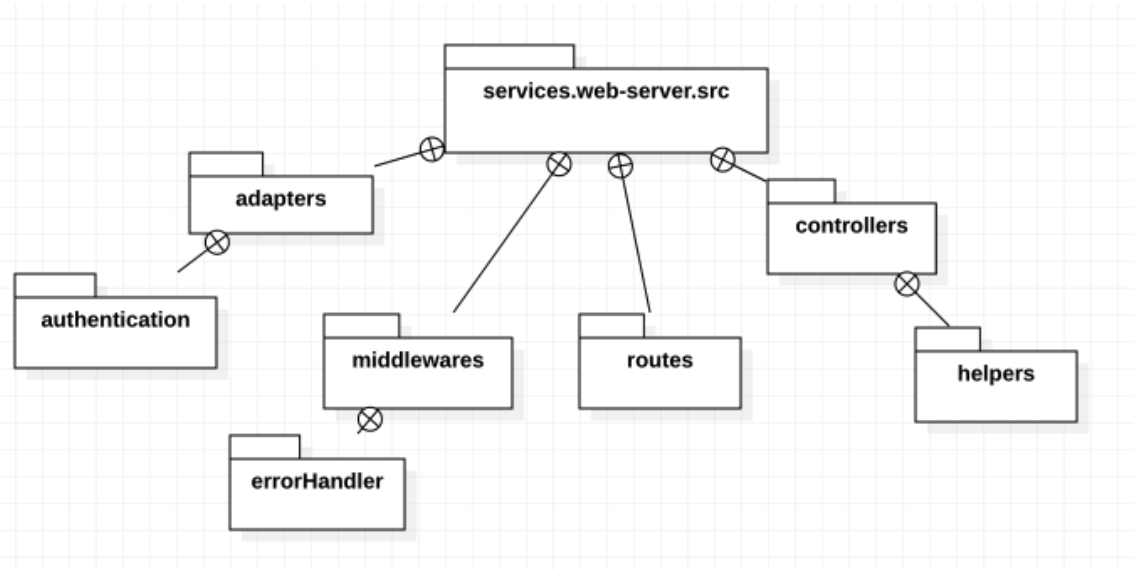
2. Arquitectura

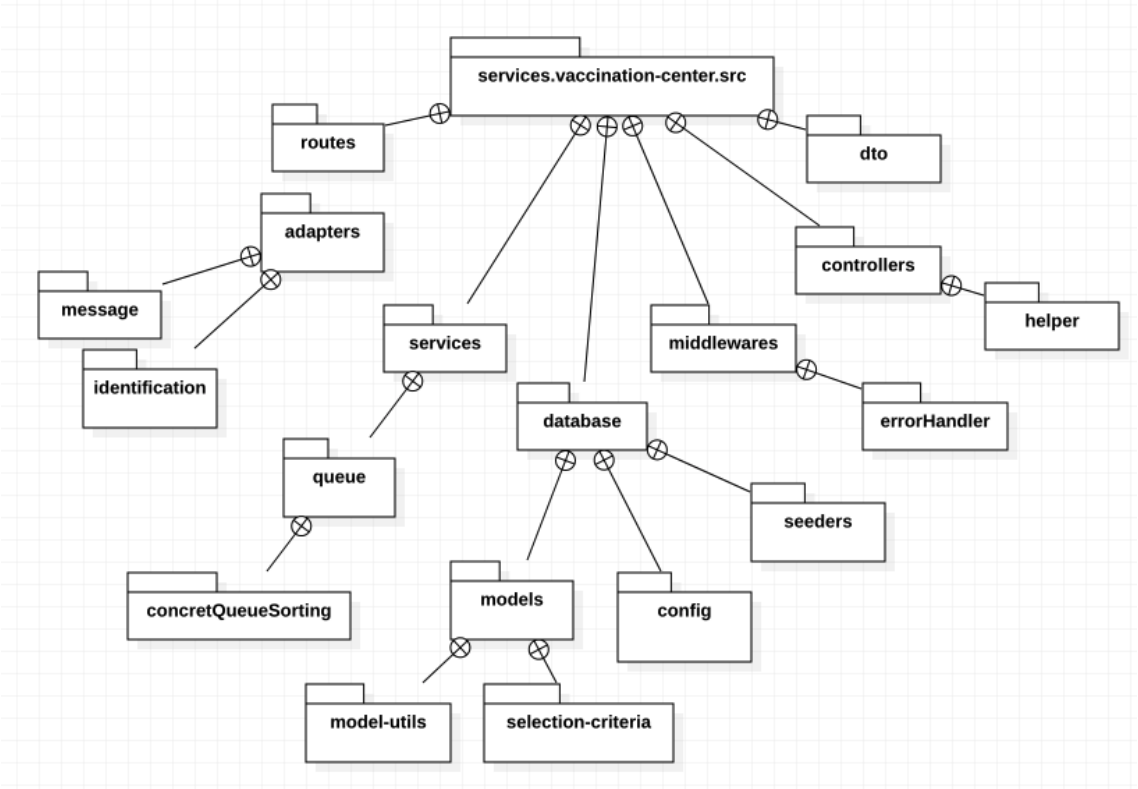
2.1. Módulos

2.1.1. Representación primaria




services.vac-query-tool.src
controllers
routes
database
middleware
services
models
config
errorHandler
model-utils


services.web-server.src
adapters
controllers
authentication
middlewares
routes
helpers
errorHandler



2.1.2. Catálogo de elementos

Cada uno de los diagramas muestra un modulo al que nosotros denominamos servicio (web-server, vac-query-tool, login y vaccination-center), a su vez estos módulos están divididos en sub módulos con algunos de los siguientes elementos:

- **Controladores:** El paquete Controllers tiene los controladores de nuestra aplicación, estos son los responsables de recibir la requests y devolver la respuesta, más no de la lógica, ya que esta se encuentra en los servicios.

- **Middlewares:** son los encargados de interactuar con la request antes de que llegue al controller o con la response antes de que se envíe al usuario (luego de salir del controller). Aquí tenemos middlewares que validan determinados headers y middlewares que toman los errores y devuelven la respuesta correspondiente al error.

- **Migraciones:** guarda las migraciones que realizaron de la base de datos, crean los modelos de las bases de datos.

- **Routes:** es el encargado de recibir la consulta y redirigirla al Controller correspondiente.

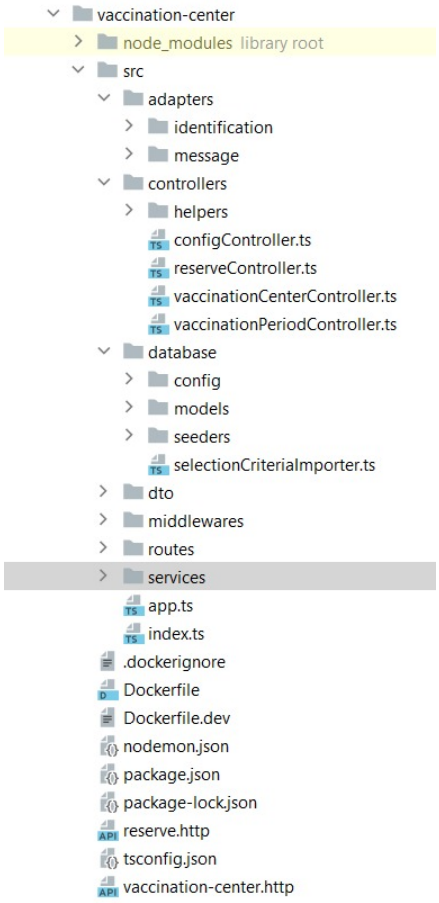
- **ErrorHandler:** contiene al Middleware encargado de recibir los errores que salen de los Controladores y devolver la respuesta adecuada a el error. También contiene el error custom creado para casos puntuales de nuestro sistema.

- **Database y Seeders:** El paquete database contiene información relacionada a la base de datos, ejemplo la configuración para conectarnos, los modelos a utilizar, y los seeders (data que se inserta al levantar la aplicación con el motivo de facilitar el uso en los entornos de pruebas).

- **Models:** Es el paquete encargado de almacenar las clases Modelo, que nos permiten acceder a las entidades almacenadas en la base de datos.

- **Adapters:** El paquete adapters contiene interfaces e implementaciones de lógica con comunicación con servicios externos, esto nos permite fácilmente extender estas mismas y así lograr por ejemplo integrar otro servicio de autenticación, identificación civil, SMS (WhatsApp, Mail), etc.

Un ejemplo de la estructura de archivos:



2.1.3. Decisiones de arquitectura

Los modelos tendrán la posibilidad de interactuar entre si. Estos podrán ser configurables en los siguientes aspectos.

El paquete de config, tendrá todo los relacionado a las configuraciones dinámicas de la app, las que serán posible realizar en tiempo de ejecución.

Los middlewares permitieron configurar al momento de compilación los distintos filtros que se le aplican a una request. Lo cual esta asociado al patrón Pipe And Filters. Aprovechamos el entorno ofrecido por Express para lograr esta funcionalidad

Las migraciones serán para ir evolucionando nuestro modelo de base de datos relacional, este sera aplicado para nuestro servicio de login que utilizo el patrón de Identidad Federada, siendo fácilmente modificable para agregarle nuevos atributos al mismo.

Los modelos utilizados permitirán la posible configuración de la validación de los atributos mediante la configuración del componente validations. Dicho componente podrá solamente ser cambiadas en tiempo de compilación por una limitante de la tecnología.

La modificabilidad consiste en reducir el costo de cambio de nuestro sistema.

Para cumplir con este atributo buscamos implementar nuestro arquitectura siguiendo las tácticas adecuadas.

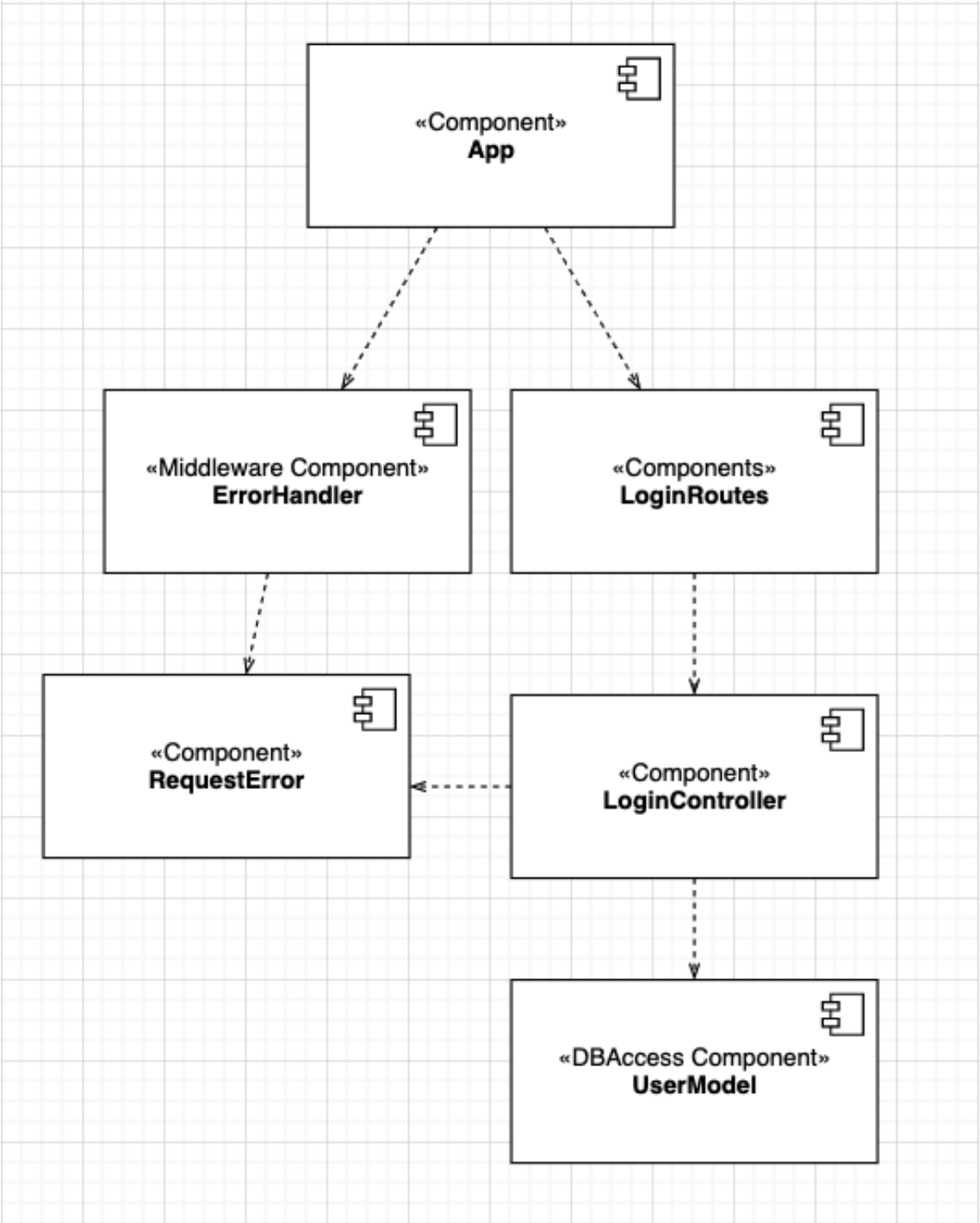
Para reducir el tamaño de los módulos utilizamos "Split Module", dividiendo nuestro sistema en módulos pequeños, cada uno de ellos conteniendo responsabilidades con un mismo propósito incrementado la cohesión, buscando que la menor cantidad de módulos se vean afectados por el mismo cambio ("Refactor").

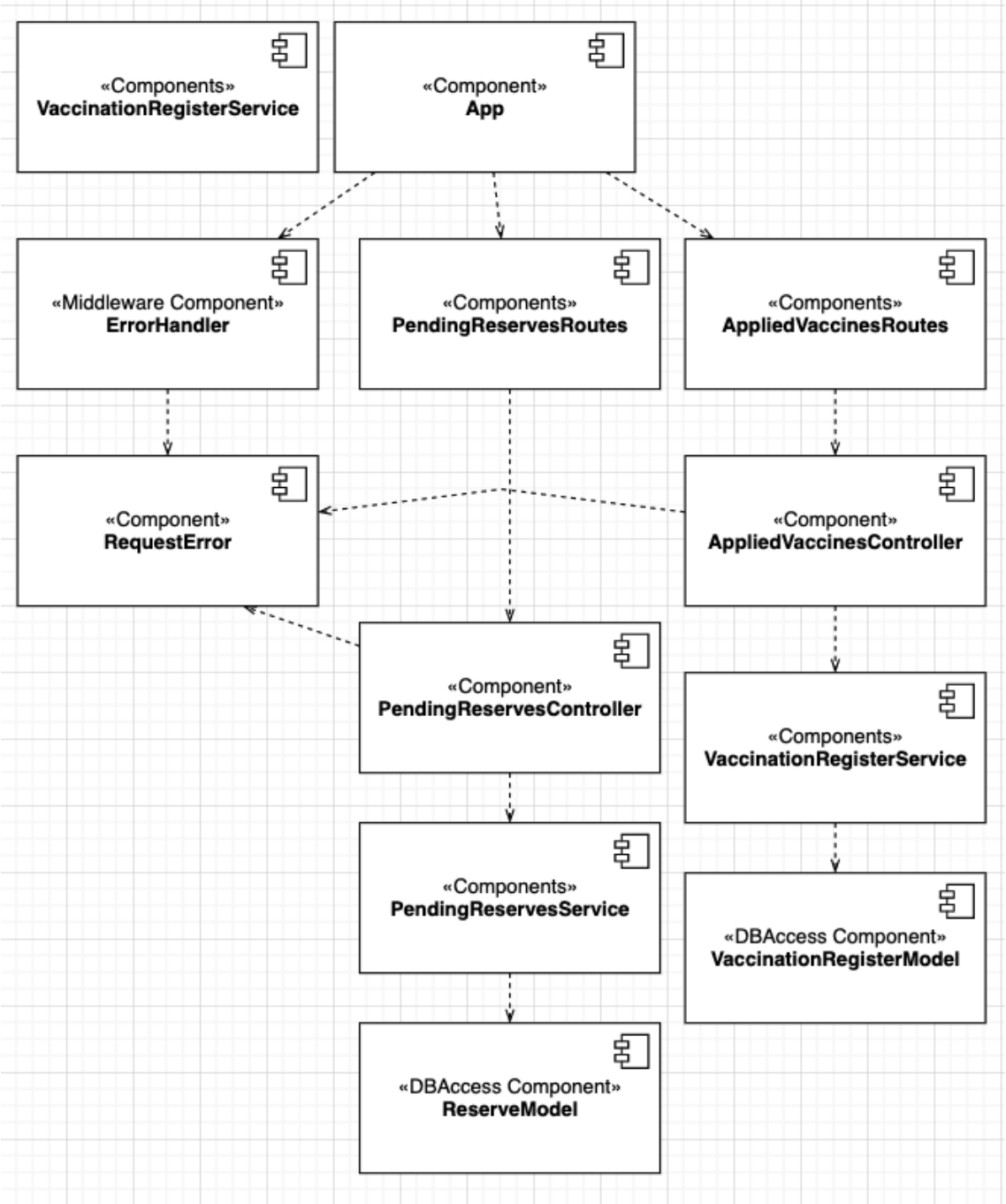
Reducimos el acoplamiento utilizando intermediarios para comunicar las distintas capas de nuestro sistema, también restringimos la visibilidad de los módulos para evitar interacciones innecesarias ("Restrict dependencies").

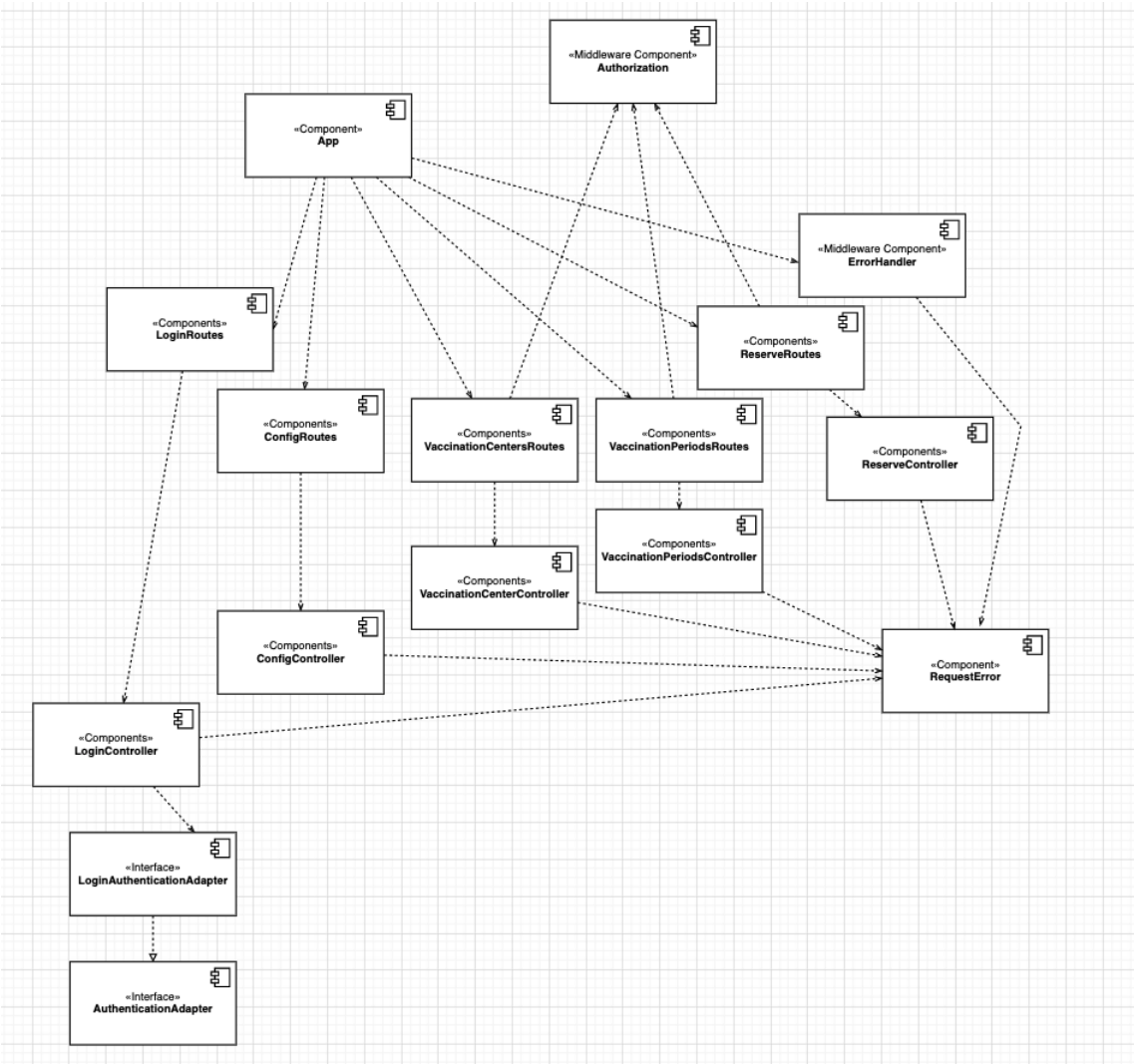
Por último cumplimos con "Defer Binding" utilizando la flexibilidad brindada por JavaScript para reducir los cambios que debe realizar el desarrollador para agregar o cambiar una funcionalidad.

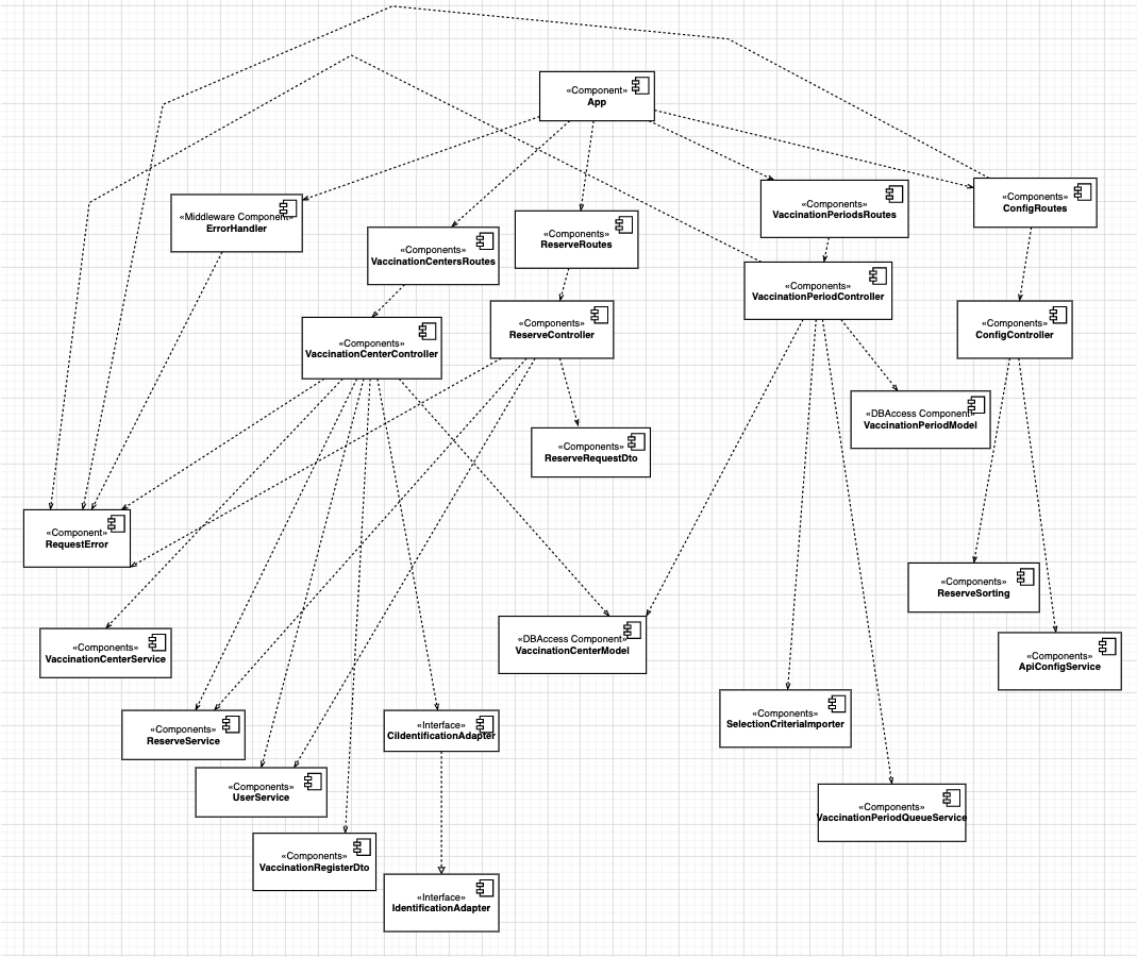
2.2. Componentes y conectores

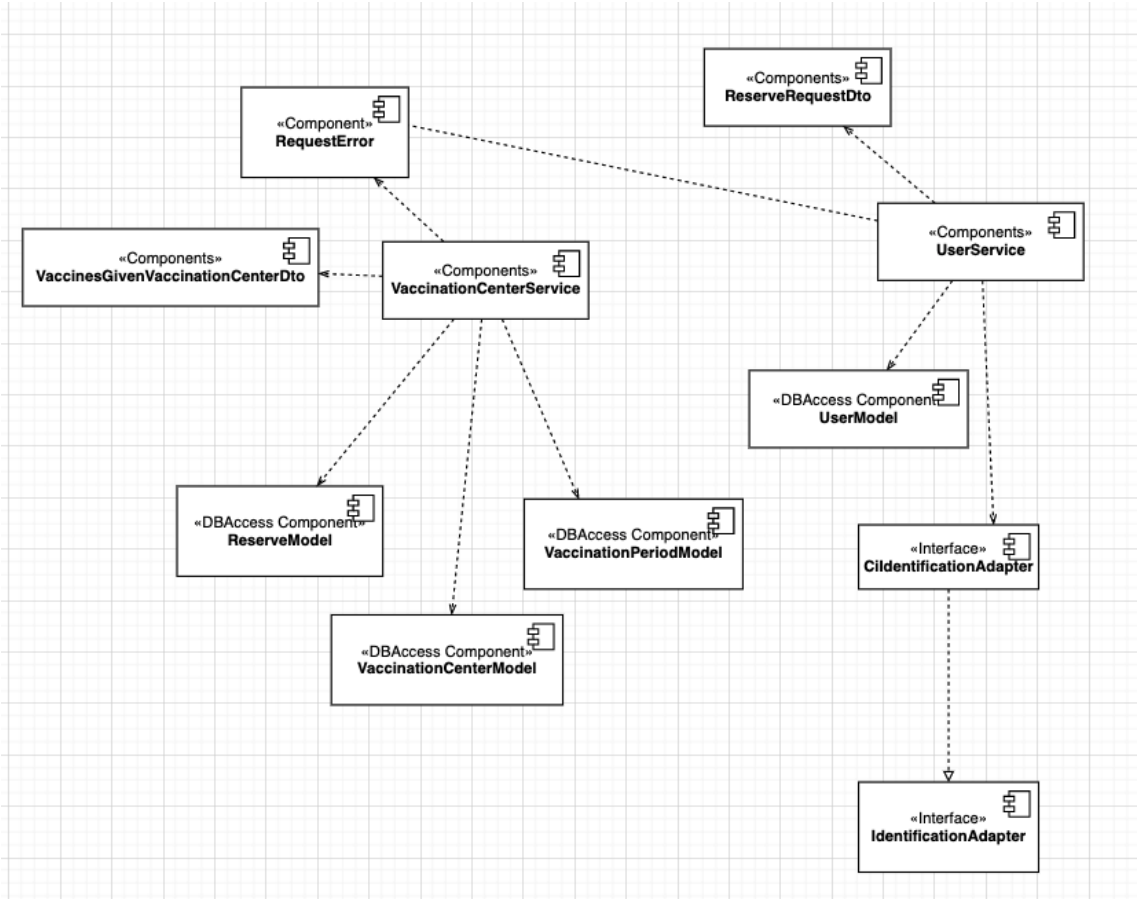
2.2.1. Representación primaria

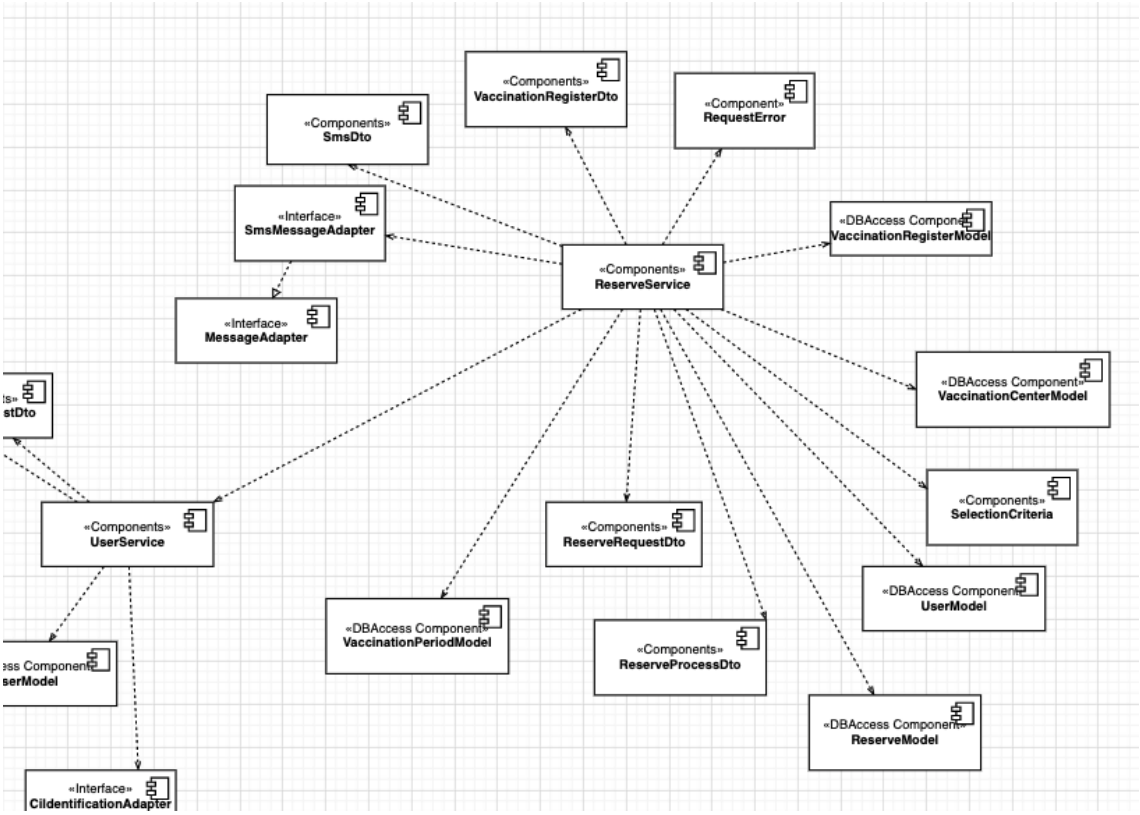


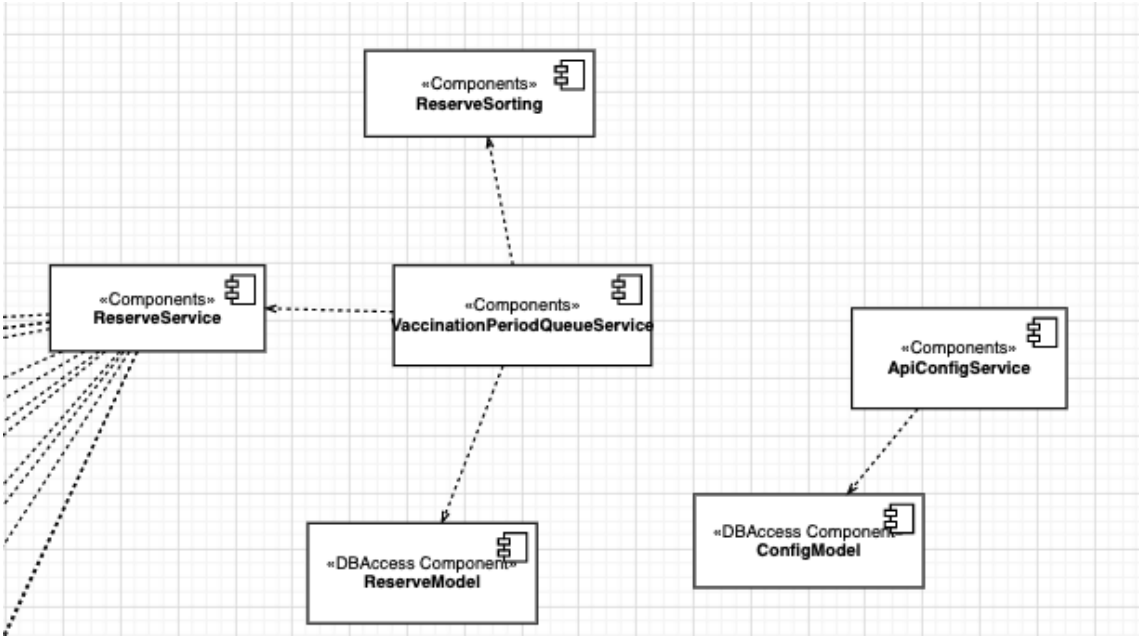

«Components»
VaccinationRegisterService
«Component»
App
«Middleware Component»
ErrorHandler
«Components»
PendingReservesRoutes
«Components»
AppliedVaccinesRoutes
«Component»
RequestError
«Component»
AppliedVaccinesController
«Component»
PendingReservesController
«Components»
VaccinationRegisterService
«Components»
PendingReservesService
«DBAccess Component»
VaccinationRegisterModel
«DBAccess Component»
ReserveModel


«Middleware Component»
Authorization
«Component»
App
«Middleware Component»
ErrorHandler
«Components»
LoginRoutes
«Components»
ReserveRoutes
«Components»
ConfigRoutes
«Components»
VaccinationCentersRoutes
«Components»
VaccinationPeriodsRoutes
«Components»
ReserveController
«Components»
VaccinationPeriodsController
«Components»
VaccinationCenterController
«Components»
ConfigController
«Component»
RequestError
«Components»
LoginController
«Interface»
LoginAuthenticationAdapter
«Interface»
AuthenticationAdapter


«Component»
App
«Middleware Component»
ErrorHandler
«Components»
VaccinationCentersRoutes
«Components»
ReserveRoutes
«Components»
VaccinationPeriodsRoutes
«Components»
ConfigRoutes
«Components»
VaccinationPeriodController
«Components»
VaccinationCenterController
«Components»
ReserveController
«Components»
ConfigController
«DBAccess Component»
VaccinationPeriodModel
«Components»
ReserveRequestDto
«Component»
RequestError
«Components»
ReserveSorting
«Components»
VaccinationCenterService
«DBAccess Component»
VaccinationCenterModel
«Components»
ApiConfigService
«Components»
ReserveService
«Interface»
CiIdentificationAdapter
«Components»
SelectionCriteriaImporter
«Components»
UserService
«Components»
VaccinationPeriodQueueService
«Components»
VaccinationRegisterDto
«Interface»
IdentificationAdapter


«Components»
ReserveRequestDto
«Component»
RequestError
«Components»
UserService
«Components»
VaccinesGivenVaccinationCenterDto
«Components»
VaccinationCenterService
«DBAccess Component»
UserModel
«DBAccess Component»
ReserveModel
«DBAccess Component»
VaccinationPeriodModel
«Interface»
CiIdentificationAdapter
«DBAccess Component»
VaccinationCenterModel
«Interface»
IdentificationAdapter


«Components»
VaccinationRegisterDto
«Components»
SmsDto
«Component»
RequestError
«Interface»
SmsMessageAdapter
«DBAccess Compone
VaccinationRegisterModel
«Components»
ReserveService
«Interface»
MessageAdapter
«DBAccess Component»
VaccinationCenterModel
«Components»
SelectionCriteria
«Components»
UserService
«Components»
ReserveRequestDto
«DBAccess Component»
UserModel
«DBAccess Component»
VaccinationPeriodModel
«Components»
ReserveProcessDto
«DBAccess Component»
ReserveModel
«Interface»
CiIdentificationAdapter



2.2.2. Catálogo de elementos

Esta vista se encuentran los siguientes elementos que la componen

- DTOs: Data Transfer Object, son los objetos utilizados para la comunicación de datos, nos permite una mayor modificabilidad ya que con solo modificar una clase podemos actualizar los datos necesarios para una lógica determinada.

- Routes: Son los responsables de definir puntos de acceso de la aplicación, usado por los controladores.

- RequestError: Clase que representa un error personalizado con la request, con esto podemos devolver un mensaje adaptado para el usuario.
- Controller: Es el encargado de despachar las solicitudes realizadas a la API

- App: Se configuran la API con los distintos módulos, rutas y middlewares a utilizar en cada uno de los servicios.

- Model: Se encarga de la estructura de los datos que serán guardados en la base de datos.
- Adapter: Se encarga de implementar el patrón Adapter para poder utilizar las APIs externas de identificación y sms, y de esa forma hacer más fácil intercambiar este servicio.
- Sorting: Se definirán los algoritmos de vacunación, los cuales incluirán la implementación como una abstracción para ser utilizada.

- Servicio: son los responsables de realizar la lógica de negocio, es utilizado por los controllers y otros servicios, utilizan los Modelos para acceder a la base de datos.

2.2.3. Guía de variabilidad

Las configuraciones a nivel de deployments seran configuradas en el archivo .env, estás son el puerto del load balancer del web server, configuraciones de las bases de datos, cómo usuarios y contraseñas, secrets de Json Web Tokens, etc.

De esta forma al ser un archivo que normalmente se encuentra ignorado (no lo ignoramos en este caso para que pueda ser probado) ganamos seguridad al no exponer nuestras credenciales, cómo también extensibilidad al poder cambiar estas credenciales para adaptarnos a distintos entornos.

2.2.4. Decisiones de arquitectura

Para el desarrollo de esta vista, desarrollamos los siguientes puntos.

La separación entre los distintos componentes fue realizada mediante el uso de DTO (Data Transfer Model) los mismos son utilizados verticalmente por las distintas capas para poder compartir información entre ellos y estos mismos funcionan como backbone. Logrando esta desacoplaron entre los distintos componentes permite que cada uno de ellos se comuniquen mediante este medio en común y que no generen dependencia cíclica entre ellos.

Durante el desarrollo de los distintos componentes y la comunicación entre ellos, la performance es un atributo de calidad clave. Dado que existirán servicios que serán utilizados mas frecuentemente que otros y existe la necesidad de poder separarlos, un claro ejemplo de esto se encuentra con el componente de VacQueryTool. Decidimos implementar CQRS en el cual cada servicio tanto de lectura/escritura.

Para abstraer los componentes para reuso futuro o cambio de implementaciones utilizamos interfaces con modelos de contratos los cuales nos indicaran el comportamiento de dichos componentes pudiendo ser así intercambiados fácilmente. El componente de manejo de SMS, fue implementado con esto en mente, en el cual se definió una interfaz para el manejo del mismo pero fácilmente se puede modificar para otro gestor de SMS un ejemplo WhatsApp. Esto se basa en el patrón de diseño Adapter donde se definirá la conversión de los distintos servicios. Otro ejemplo de uso de este patrón es la API de identificación.

En lugar de realizar una aplicación monolito optamos por una orientada a SOA. En el cual separamos en los siguientes APIS

- Web-Server
  Esta sera la encargada de recibir las request y despacharlas a los demás servicios

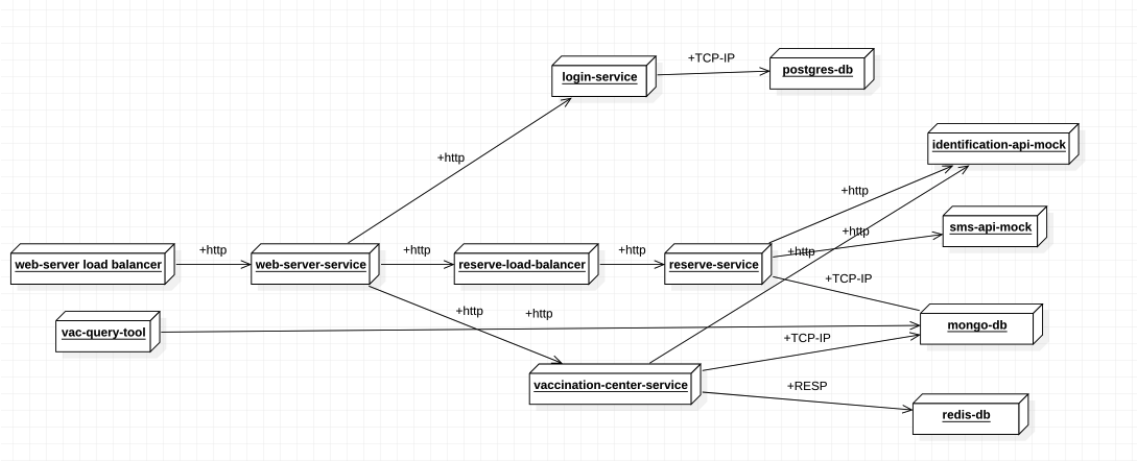
- Reserve
  Esta sera la encargada de interactuar con la población para pode gestionar las reservas. Las funcionalidades que esta incluye son
  - Creación de una reserva
  - Cancelar una reserva
  - Obtener una reserva
- VaccinationCenter
  Sera el centro operativo para el manejo de los vacunatorios, el cual contiene las funcionalidades
  - Crear un vacunatorio
  - Obtener cantidad de vacunas restantes
  - Vacunar un usuario
  - Crear un periodo de vacunación
  - Obtener datos sobre un vacunatorio
  - Configurar las URL de los servicios de ID
  - Configurar el algoritmo de reservas
- Login
  Sera el servicio de autenticación y autorización, el cual se apoya en el patrón de Autenticación Federada.
- VacQueryTool
  Se encargara de realizar consultas sobre los datos de los sobre las vacunaciones realizadas

Todos estos servicios mencionados anteriormente se comunicaran mediante el protocolo HTTP con REST, los cual nos permiten mantener una estandarización en la comunicación entre los servicios.

Optamos por la arquitectura SOA, dado que se nos un problema que se requiere poder mantener una arquitectura mantenible en el tiempo y modificable para los distintos contexto especialmente para los distintos países. Otra ventaja frente al monolito es la posibilidad gestionar los servicios independientemente y poder aumentar horizontalmente logrando así reducir los costos de ampliación de servicios. Una desventaja que se nos presento al utilizar un modelo SOA es el overhead de desarrollo de la misma.

2.3. Asignación

2.3.1. Representación primaria



2.3.2. Catálogo de elementos

- **Web server load balancer service:** Load balancer para manejar la carga del web server.
- **Web server service:** Web server, es el punto común para comunicarnos con nuestro sistema, lo que ganamos con esto es centralizar la autenticación y autorización, de esta forma pudiendo servir de firewall para los Application Servers.
- **vac-query-tool service:** Servicio sobre el cual podemos realizar consultas.
- **reserve-service load-balancer:** Load balancer para el servicio de reservas.
- **reserve-service:** Servicio encargado de las reservas, responsable de asignar un turno, cancelarlo y ver información, también envia los SMS.
- **vaccination-center-service:** Servicio encargado de los centros de vacunación, podemos crear estos últimos, crear periodos y consultar el estado del plan.
- **login-service:** Servicio responsable de la autorización, guarda datos en PostgreSQL.
- **postgres-db:** Basde de datos de Postgres SQL. La cual utiliza una base de datos relacion definiendo tablas.
- **identification-api-mock:** Mock del servicio de autenticación civil, lee datos de un CSV y los expone en una API Rest.
- **sms-api-mock:** Mock del servicio de envió de SMS, recibe un teléfono y un mensaje y loggea por consola.

- **mongo-db:** Base de datos de Mongo, es una base de datos no relacional por documentos, con esto conseguimos datos no estructurados.
- **redis-db:** Base de datos de redis, la misma maneja una base de datos del estilo key-value, con esto intentamos conseguir mayor performance al cachear los resultados.

2.3.3. Decisiones de arquitectura

Performance

Es el tiempo y la capacidad del sistema de cumplir con los requisitos de tiempo.

Para cumplir con la Performance introducimos asincronía haciendo uso de promesas de JavaScript.

Aunque no se llegó a implementar dada la limitante de tiempo (el servicio esta levantado en el docker-compose.yml), nuestra idea era en las consultas para el estado del plan de vacunación estuvieran guardadas en Redis y cuando el usuario consulte conseguir los datos desde ahí, realizando un job cada una hora que actualiza los datos de caché.

Para cubrir la alta demanda en los servicios levantamos varias replicas de los servicios más accedidos, esto sumado a un balanceo de carga utilizando Nginx genera que el tráfico se vea distribuido generando así una mejor performance ya que no se sobrecarga una sola instancia de la aplicación.

Para sacar el mejor provecho de esto separamos los módulos en base a responsabilidades, por lo que se va a escalar el módulo necesario y estimamos que las consultas por módulo serán similares (teniendo similares tiempos de procesamiento); de esa forma evitamos generar "cuellos de botella" en las instancias.

También utilizamos el patrón CQRS, cuando confirmamos una vacunación, al mismo tiempo guardamos datos en un modelo auxiliar, esto genera que las consultas de VacQueryTool sean mucho más rápidas, ya que los datos están completos y no debemos realizar múltiples consultas.

Seguridad

La seguridad es la medida en que un sistema es capaz de proteger los datos y la información de accesos no autorizados.

Para esto hacemos autenticación y autorización de autores mediante un login y uso de JWT (Json Web Token).
La autenticación la realizamos guardando un usuario y una contraseña que se guarda encriptada, esto va a una base de datos PostgreSQL, esta misma al ser relacional nos asegura cumplir con ACID. esto es de gran valor para guardar información tan

sensible, ya que aseguramos consistencia y durabilidad de los datos.

Garantizamos que los autores autorizados tengan acceso a los datos y servicios mediante Middlewares que realizan las validaciones de seguridad correspondientes para poder cumplir con la solicitud. De esta forma también nos aseguramos que el acceso a los recursos sensibles este limitado, complementando esto también limitamos la exposición de los módulos, evitando que se pueda usar de forma externa, sin el intermediario implementado para su uso (el Web Server).

Para evitar que se pueda acceder a la capa de Application Server, en el docker-compose no exponemos los puertos de forma pública, si no que estos están solamente expuestos en la red de Docker-Compose, por lo que a modo de firewall, nuestros servidores únicamente aceptan requests que vengan a traves de su Load Balancer, que a su vez únicamente acepta requests del Web Server, únicamente realizamos validaciones en el Web Server, ya que los otros servicios no pueden ser accedidos, esto nos da la ventaja de poder utilizar otros patrones como por ejemplo gRPC en caso de ser necesario.

Disponibilidad

Propiedad del software de estar al alcance y listo para realizar su tarea cuando se necesite.

La disponibilidad esta muy relacionada con los atributos de calidad Seguridad y Performance, por lo que al implementar tácticas y mecanismos que cumplieran esos atributos complementamos al cumplimiento de este atributo.

Como comentamos en la sección de Preformance utilizamos múltiples réplicas balanceando el tráfico mediante Nginx, evitando comprometer los atributos de calidad.

Una posible evolución de esto podría ser utilizar una tecnología como Kubernetes, con esto generaríamos que el sistema siempre se mantenga en un estado constante de intentar llegar a cumplir la disponibilidad, ya que en caso de que el uso de CPU/Memoria suba cierto umbral (configurable mediante políticas de escalamiento) podemos levantar nuevos contenedores de Docker (utilizando pods) y con esto no generar afectación a nuestros usuarios.

Realizamos el manejo de los errores del sistema mediante el uso de un Middleware, para además de para evitar vulnerabilidades (cumpliendo con la Seguridad) también evitamos que el sistema deje que funcionar afectando la disponibilidad.

Bibliografía

[1] Universidad ORT Uruguay. (2013) Documento 302 - Facultad de Ingeniería. [Online]. Available: https://www.ort.edu.uy/fi/documentos/normas-especificas-para-la-presentacion-de-trabajos-finales-de-carrera.pdf

[2] Len Bass, Paul Clements, Rick Kazman. (2015) Software Architecture in Practice - Third edition. [Online]. Available: http://jz81.github.io/course/sa/Software%20Architecture%20in%20Practice%20(3rd).pdf