

# Arquitectura de Software:

## Obligatorio

Alumnos: Paula Espinosa 210635  
Tomás De Angelis 209270  
Facundo López 219955

LINK AL REPO:

<https://github.com/ORTArqSoft/210635-209270-219955>

# Índice

<b>Índice</b>	<b>1</b>
<b>Introducción</b>	<b>4</b>
Propósito del sistema	4
<b>Requerimientos significativos de arquitectura</b>	<b>5</b>
Resumen requerimientos de atributos de calidad	5
Restricciones	6
<b>Documentación de la arquitectura</b>	<b>7</b>
Vista de Asignación	7
Vista de Despliegue	7
Representación Primaria	7
Justificaciones de Diseño	10
Vista de Módulos	14
Vista de Descomposición	14
1) Processor Service	14
Representación Primaria	14
Catálogo de Elementos	14
Justificaciones de Diseño	15
2) Admin Service	20
Representación primaria	20
Catálogo de Elementos	20
Justificaciones de Diseño	21
3) Logger	22
Representación primaria	22
Catálogo de elementos	22
Justificaciones de Diseño	22
3) Auth-Service	25
Representación primaria	25
Catálogo de elementos	25
Justificaciones de Diseño	25
4) Notification Service	27
Representación primaria	27
Catálogo de Elementos	27
Justificaciones de Diseño	27
5) Pending Service	28
Representación primaria	28
Catálogo de elementos	28
Justificación de diseño	28
Vista de Usos (tomando en cuenta dependencias entre módulos)	29

1) Processor Service	29
Justificación de diseño	29
2) Admin Service	30
3) Otros Servicios genéricos	30
Vista de componentes y conectores	31
Processor Service	32
Representación primaria	32
Catálogo de elementos	32
Comportamiento	36
Justificaciones de diseño	37
Admin service	42
Representación primaria	42
Catálogo de elementos	42
Comportamiento	45
Justificaciones de diseño	46
Auth Service	49
Representación primaria	49
Catálogo de elementos	49
Comportamiento	51
Justificaciones de diseño	51
Pending Service	53
Representación primaria	53
Catálogo de elementos	53
Comportamiento	54
Justificaciones de diseño	54
Reservation Emitter	57
Representación primaria	57
Catálogo de elementos	57
Justificaciones de diseño	58
Logger	60
Representación primaria	60
Catálogo de elementos	60
Justificaciones de diseño	61
VacQueryTools	62
Representación Primaria	62
Catálogo de elementos	62
Comportamiento	63
Justificaciones de diseño	63
Diferencia con las consultas por el plan de Vacunación	66
<b>Otros aspectos de la Documentación</b>	<b>66</b>
Control de Versiones	66
Calidad de Código	67

<b>Manual de Instalación</b>	<b>68</b>
Aclaración sobre Ejecución de Aplicaciones	68
Link para conectar a Mongo desde Compass	68
Manual de instalación para correr en nuestro cluster en Cloud	68
Manual de instalación para correr en base de datos local	69
<b>Anexo</b>	<b>72</b>
Proceso de desarrollo	72
Organización Interna	72
Histórico de Diagramas de Arquitectura	74
Detalles finales que no llegamos a cubrir	81

# Introducción

El propósito de este documento es describir la arquitectura del sistema Vac Planner. Se identificarán los **atributos de calidad** que nos servirán como guía de la **arquitectura** y se especificará la arquitectura implementada en base al modelo Views & Beyond.

## Propósito del sistema

Se trata de una plataforma para agendar y planificar vacunaciones en distintos países. Entre sus objetivos están cubrir las necesidades del ente regulador de la campaña de vacunación, facilitar el proceso de reserva y consultas sobre el estado del plan, y brindar información en todo momento sobre la ejecución de dicho plan.

En primera instancia el sistema funcionará con los datos y las características del Uruguay pero es fundamental que la solución sea adaptable a las características de otros países.

# Requerimientos significativos de arquitectura

## Resumen requerimientos de atributos de calidad

#Req	Atributo de Calidad	Descripción
1	Performance / Availability	<b>Manejo de carga</b> Es importante que se asegure que no se pierdan datos de reservas aun en momentos de saturación, logrando la mejor latencia posible.
2	Performance	Las consultas de VacQueryTool deben tener el menor jitter posible en momentos de carga. En promedio las consultas complejas deben tener una latencia menor a 2 segundos.
3	Seguridad / Disponibilidad	<b>Gestión de Errores y Fallas:</b> El sistema debe proveer suficiente información que permita conocer el detalle de las tareas que se realizan. En particular, en el caso de ocurrir una falla, ataque o cualquier tipo de error, es imprescindible que el sistema provea toda la información necesaria que permita a los disparadores hacer un diagnóstico rápido y preciso sobre las causas.
4	Seguridad	<b>Autenticación de administradores y vacunadores:</b> Algunas funcionalidades u operaciones pueden ser accedidas únicamente por usuarios autenticados.
5	Modificabilidad	La solución para la gestión de fallas y errores, debe contar con la posibilidad de poder cambiar las herramientas o librerías concretas que se utilicen para producir esta información con el menor costo posible.
6	Interoperabilidad	La solución para la gestión de fallas y errores, debe poder ser reutilizada en otras aplicaciones (entendemos de NodeJS), con el menor impacto posible en el código de las mismas.
7	Modificabilidad	Se deben poder modificar mediante API las validaciones realizadas sobre las solicitudes de reserva y los endpoints utilizados para invocar a las APIs externas.
8	Modificabilidad	La incorporación de nuevas librerías o herramientas utilizadas para el registro de errores, así como el cambio de la herramienta utilizada, debe poder realizarse sin necesidad de reiniciar el sistema.
9	Seguridad / Modificabilidad	Las contraseñas deben estar encriptadas.
10	Modificabilidad	Cambiar el algoritmo de asignación de vacunatorios debe tener el menor impacto posible.

11	Performance	La respuesta a la petición de reserva debe darse en una ventana de tiempo entre 30 segundos y 5 minutos. Este tiempo de respuesta es un aspecto que se desea optimizar lo más posible.
12	Modificabilidad	El VacQueryTool permite realizar consultas complejas mediante API REST sobre la base del plan de vacunación. A modo de ejemplo se dan algunas en la letra.
13	Modificabilidad	En un futuro se desea enviar mediante otros proveedores de mensajería con el menor costo de cambio posible.

## Restricciones

- La implementación del backend, debe ser realizada utilizando NodeJS, utilizando las tecnologías vistas en el curso. A su vez, está permitido el uso de otros paquetes que puedan ayudar en el desarrollo.
- Todo el código de fuente, documentación, archivos de configuración o cualquier otro artefacto requerido al desarrollo de los prototipos, debe gestionarse en un repositorio Git, asignado en la organización de GitHub del curso.

# Documentación de la arquitectura

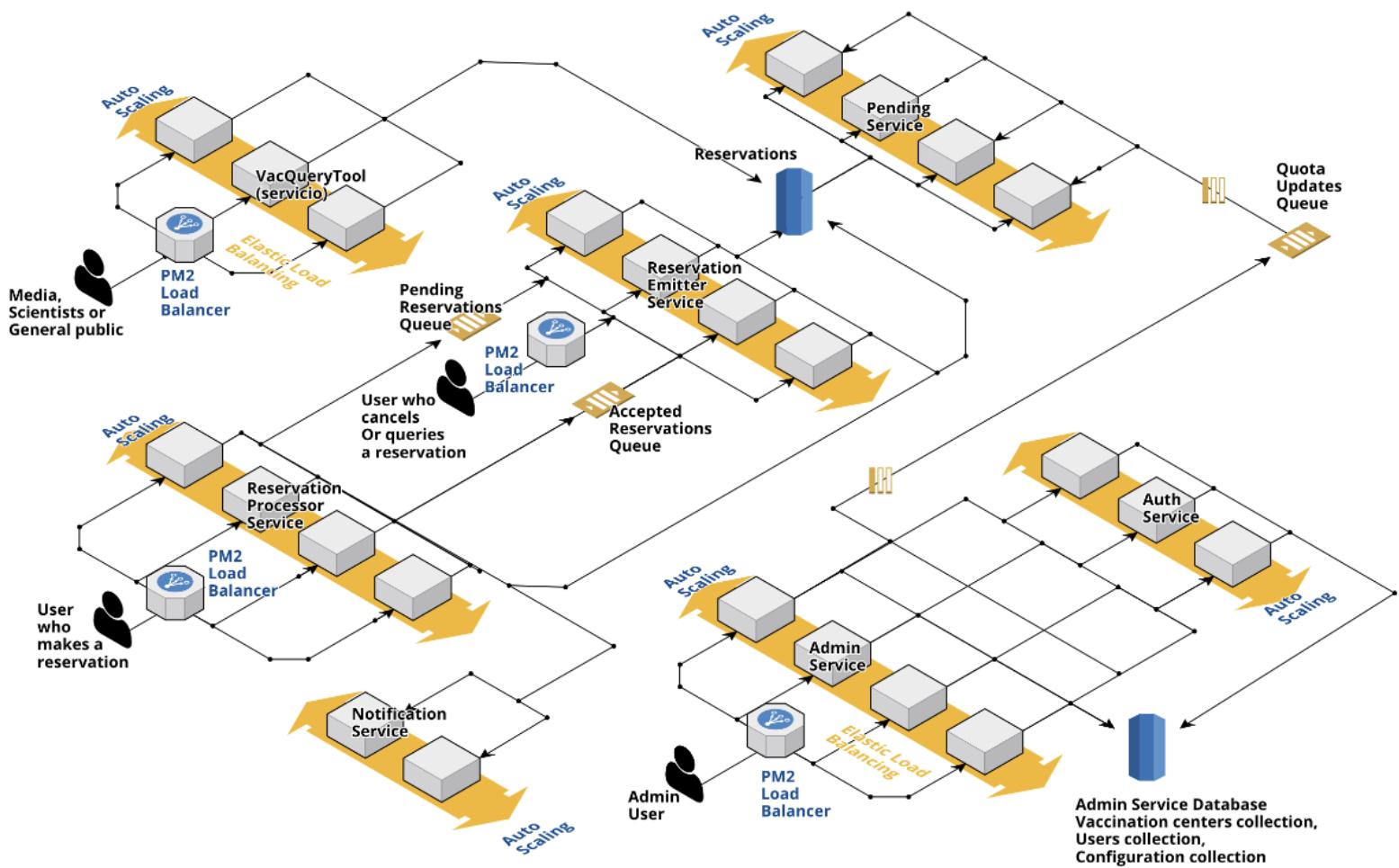
## Vista de Asignación

## Vista de Despliegue

### Representación Primaria

Pese a que sabemos que este tipo de diagramas de despliegue no es de los que dimos en el curso, nos parece una manera clara y fácil de entender la arquitectura que pensamos para nuestra solución. En el siguiente diagrama se ven todos los **servicios, bases de datos, colas, potenciales usuarios, posibles instancias de replicación, y comunicación entre servicios (sumado a las relaciones entre todos estos)**. De esta manera se da una primera impresión de nuestra arquitectura que profundizaremos más adelante.

El diagrama fue realizado en *CloudCraft* y pasó por varias iteraciones, hasta que llegamos a la siguiente versión. El histórico de los diagramas y el proceso que desarrollamos para terminar con esta arquitectura puede encontrarse en la sección “Proceso de desarrollo, Histórico de Diagramas de Arquitectura” en el Anexo. Es un proceso muy interesante, que creemos vale la pena repasar.



Catálogo de Elementos	
Elemento	Responsabilidades
Reservation Processor Service	<p>Es el servicio encargado de recibir y realizar el procesamiento de las reservas para vacunación. Cuando le llega una solicitud de reserva se encarga de realizar las validaciones y transformaciones que corresponden. Busca un vacunatorio assignable según los criterios determinados.</p> <p>Si encuentra un vacunatorio para la reserva, se resta un cupo al vacunatorio y se envía a la cola de reservas aceptadas.</p> <p>Si no encuentra vacunatorio, manda la reserva a la cola de reservas pendientes a esperar por nuevos cupos.</p> <p>El Reservation Emitter Service consumirá ambas colas para continuar.</p> <p>Este servicio también expone un endpoint que permite saber cuales son las validaciones y transformaciones existentes (que consumen los usuarios con privilegios necesarios mediante el AdminService).</p>
Admin Service	<p>Es el servicio al cual los administradores acceden para realizar tareas que requieren autenticación, tales como:</p> <ul style="list-style-type: none"> <li>- crear vacunatorios</li> <li>- crear usuarios (administradores y vacunadores)</li> <li>- agregar cupos a un vacunatorio</li> <li>- configurar y obtener las validaciones y los endpoints de terceros.</li> </ul> <p>Los usuarios vacunadores también acceden a este servicio para marcar a una persona como vacunada.</p> <p>Al agregar nuevos cupos, el servicio los encola en la cola de <i>Quota Updated Queue</i> para ser consumida por el Pending Service. Este intentará asignar las reservas pendientes con estos nuevos cupos.</p>
Reservation Emitter Service	<p>Es el servicio encargado de consumir reservas (ya validadas) de las colas de reservas aceptadas y pendientes. Guarda dichas reservas en la base de datos.</p> <p>A su vez, es el encargado de permitir a los usuarios cancelar una reserva o consultar por el estado de la misma.</p>
Auth Service	<p>Es el servicio interno encargado de autenticar usuarios. Sus endpoints no están expuestos a ningún usuario, sino que se usa entre servicios.</p> <p>Es decir, es accesible únicamente desde la red interna (esto será configurable una vez deployado el sistema).</p> <p>Se encarga de generar tokens, hacer rekey de los tokens y validar que los tokens sean válidos.</p>

Pending Service	Es el servicio encargado de procesar los cupos nuevos sobre vacunatorios existentes para intentar agendar las reservas en lista de espera, y asignarles un vacunatorio.
VacQueryTools	Es el servicio al cual los medios de prensa, el público o los administradores del sistema, acceden para realizar consultas acerca del proceso de vacunación.
Notification Service	Es el servicio encargado de notificar a los usuarios. Notificará a los usuarios según se haya configurado en el <i>AdminService</i> (sms, whatsapp, email...).
Admin Service Database	Es la base de datos encargada de almacenar los datos relacionados con los ajustes del sistema. Almacena: <ul style="list-style-type: none"> <li>- El histórico de cupos que fueron asignados a los vacunatorios</li> <li>- Los usuarios que necesitarán autenticación (administradores, vacunadores)</li> <li>- Los centros de vacunación</li> <li>- Las validaciones y transformaciones correspondientes del país</li> <li>- Los endpoints de terceros a utilizar (notificaciones, validaciones contra el proveedor de identificación...)</li> </ul>
Reservations Database	Es la base de datos encargada de almacenar las reservas de los usuarios. Almacena tanto las reservas pendientes como las realizadas.
Quota Updates Queue	Es la cola en la que se encolan los nuevos cupos disponibles para los vacunatorios agregados desde el <i>Admin Service</i> . Esta cola es consumida por el servicio <i>Pending</i> el cual intentará asignar las reservas pendientes a estos nuevos cupos.
Accepted Reservations Queue	Es la cola en la cual se encolan las reservas aceptadas (aquellas que tienen un vacunatorio y cupo asignado). Las mismas son persistidas por el <i>Reservation Emitter</i> (servicio que consume la cola) en la <i>Reservation Database</i> .
Pending Reservations Queue	Es la cola en la cual se encolan las reservas pendientes (aquellas para las que no se encontró vacunatorio). Las mismas deberán ser persistidas como pendientes en la base <i>Reservation Database</i> por el <i>Reservation Emitter</i> (el servicio que consume la cola).
Whitelisted tokens redis db	Es la base de datos en memoria que usa el <i>Auth Service</i> para almacenar los tokens válidos.

En nuestro proyecto existen dos servicios más que son el *ReservationData* y el *IdProviderMock*. El primero emula las solicitudes de reservas de los usuarios, se puede decidir cuánta carga se desea enviar. El *IdProviderMock* emula ser el servicio de identificación civil con la información personal de todos los usuarios. El emulador de sms se encuentra dentro del Notificacion Service.

Dentro del repositorio, creamos un archivo README.txt donde describimos cada uno de los *npm* que fuimos utilizando en cada servicio a lo largo del desarrollo de la solución y justificando por qué los usamos. Esto es de mucha utilidad, ya que estas librerías suelen sufrir actualizaciones constantes que debemos mantener. Además, es una buena práctica para ser conscientes de que se usa, para qué, y también saber qué cosas se dejaron de usar. Pequeños detalles que ayudan a hacer la diferencia.

### Justificaciones de Diseño

Al estructurar el sistema, tuvimos en cuenta tanto los requerimientos no funcionales pedidos, como aquellos agregados por nosotros para realizar una mejor solución.

Como se puede observar en el anterior esquema, se procuró separar la implementación del *Vac Planner* en más de un servicio de forma que se favorece el bajo acoplamiento, la alta cohesión y la separación de responsabilidades.

Intentamos favorecer al máximo cada atributo de calidad identificado, evaluando las implicancias que estos traían consigo. Uno de los ejemplos más claros de trade off presente es cómo el servicio de procesamiento de reservas solo se encarga de validarlas y determinar si las puede agendar, delegando la persistencia de las reservas al *Emitter Service*, ya sea como pendientes o concretadas. Esta decisión fue tomada con el objetivo de favorecer la performance ya que estamos sacando del proceso principal un acceso a la base de datos que se hará muy seguido por cada petición de reserva válida. Lo que perdemos con esta decisión -su desventaja- es que no podemos estar 100% seguros de que la reserva se guardó correctamente en la base de datos porque no estamos esperando una respuesta correcta de la inserción para continuar el procesamiento. El servicio *Processor* notifica al usuario indicando que su reserva fue agendada luego de encolarla en la cola de reservas. Pero si el servicio *Emitter* falla (y no agrega la reserva a la base), tendremos el problema de que el usuario se habrá quedado con el mensaje de reserva agendada pero esto no habrá pasado. De todas maneras, preferimos favorecer la performance para cumplir con nuestro requerimiento de atributo de calidad 11, el cual desea optimizar lo más posible el procesamiento de la reserva. Igualmente, intentamos que el trade off tenga el menor impacto posible. El *Emitter Service* podría implementar una **táctica de disponibilidad “Recover from faults”** como un **mechanismo de retry**, para que no se pierda la reserva al primer error al intentar persistir.

Tradeoffs como este, y muchos otros, estarán documentados y justificados más adelante.

Identificamos también diversas responsabilidades dentro del sistema, y para disminuir al máximo el costo de cambio intentamos modularizar al máximo nuestra solución. Un claro ejemplo de esto es, por ejemplo, haber realizado un servicio

notificador, o un servicio encargado de la autenticación de usuarios (Auth Service), cuando esto podría estar integrado y formar parte de los otros servicios ya existentes. La decisión detrás de esto es que, como lo diseñamos, un cambio en el servicio de notificación, impacta únicamente a este servicio. Habiendo modularizado la solución, **favorecemos la modificabilidad y también la disponibilidad**. Generamos la ventaja que ante una eventual necesidad de solucionar un problema, o realizar una modificación o adición a uno de los servicios que requiera apagar momentáneamente, el resto de los servicios seguirán funcionando.

Por ejemplo, en caso de tener que apagar el *Processor Service* por unos minutos, el *Emitter Service* podrá seguir persistiendo las reservas que estaban en la cola. En caso de agregarse nuevos cupos, el *Pending Service* podrá seguir agendando las reservas que estaban pendientes. Se podrán seguir realizando consultas desde el *VacQueryTools*. Y desde el *Admin Service* se podrán seguir agregando vacunatorios, cupos, usuarios...

Otra de las ventajas de esta separación en aplicaciones Node, es la posibilidad de destinar distintos niveles de recursos a cada una de las aplicaciones por separado. **Escalar el sistema según necesidad.** Por un lado, esto permite hacer énfasis en la performance, pudiendo realizar grupos de escalada en base a cada aplicación por separado, soportando mayores cargas, donde sea necesario. Por otro lado, si bien no se especifica, en un entorno de producción, la administración de los costos es importante. Y esta disposición, permite aumentar la performance, cuidando de no aumentar los recursos donde no sea necesario. Ya que se podrían aumentar o reducir recursos únicamente para un sistema, basándonos en la carga del CPU en un momento dado, sin implicar al resto. Por ejemplo, si viéramos que 18 de cada 20 queries del plan de vacunacion se hacen pasadas las 19 horas de cada dia, podríamos programar un algoritmo para aumentar y disminuir el cómputo del Servicio *VacQueryTools* en ese horario y favorecer la economía de nuestra empresa así como también satisfacer a los usuarios que desean informarse.

En este caso, la única aplicación que corremos con PM2 con miras de satisfacer la perfomance del procesamiento de las reservas es el *Processor Service*. El motivo específico de esto lo explicaremos más adelante y expandimos.

De todas formas, actualmente el procesamiento del Vac Planner se ve limitado por los recursos de cada máquina en la que corra. En el caso hipotético de estar trabajando en AWS, Azure, podríamos asignar las aplicaciones a grupos de escala automática e incorporar un balanceador de carga en las aplicaciones que deseemos.

Las bases de datos utilizadas fueron (las dos que presentamos anteriormente) MongoDB. Elegimos MongoDB ya que queríamos aprovechar la flexibilidad que brindan las base de datos documentales. Aprovechamos esto, por ejemplo, en la

base de datos Admin (en la colección *quota*), en la que se guardan los cupos de cada vacunatorio. Cuando el criterio de vacunación del vacunatorio es por edad, se necesitan los campos *start\_age* y *finish\_age*. Pero cuando se vacuna por prioridad, se necesita un único campo *priority*. Además, permite escalabilidad y tienen una fácil adaptación de nuevas evoluciones o implementaciones. Decidimos utilizar dos bases de datos, la base Admin relacionada al *Admin Service* y la base de datos de las reservas. Esto es para poder separar responsabilidades, no tener un único punto de falla y hacer que la base de datos Admin en ningún momento se vea saturada por las peticiones de reserva.

Dado que elegimos una base NoSQL, se utilizó el paquete mongoose para poder determinar, en algunos casos, Schemas al momento de comunicarse con la base. En esta oportunidad los índices utilizados fueron agregados directamente desde MongoDB Compass pero mongoose también permite la definición y creación de estos índices, hablaremos de ellos más adelante.

Pese a que el equipo comenzó trabajando con instancias locales de MongoDB, para poder trabajar en conjunto, decidimos crear un cluster en cloud y utilizar ese entre todos. El servicio utiliza ahora el cluster que tenemos arriba corriendo. Esto agrega latencia y más todavía teniendo en cuenta que usamos la versión gratuita del cluster de mongo.

Es decir, no hay que olvidarse de que todas las pruebas de performance que se realizaron en nuestro sistema, performarian mejor si VacPlanner se implementará realmente ya que se utilizarían mejores recursos de cluster de mongo e internet (no usariamos wifi para la conexión y tendríamos un servidor con buena velocidad de conexión dedicado solo a esto). Además, obviamente si hubiéramos implementado el servicio con una base de datos local (en localhost), la latencia de cada consulta sería muchísimo menor.

A lo largo de todos los servicios, usamos la **táctica de disponibilidad “manejo de excepciones”**, implementando catches en donde era necesario, para prevenir errores inesperados. Estos errores los registramos con ayuda del logger, aclarando qué fue lo que sucedió y dónde sucedió.

En las siguientes vistas, procedemos a detallar tanto la arquitectura como el funcionamiento de cada una de las aplicaciones Node desarrolladas, mencionando en más detalle la comunicación entre cada uno de ellos donde sea necesario, y hablando más sobre implementación.

## Conclusión

Lo que debemos tomar como conclusión principal, es que la arquitectura diseñada es escalable, vertical y horizontalmente. ¿Cómo? Agregando más copias de cómputo (horizontal) en computadoras con más potencia (vertical). Además, con pequeños ajustes como un WAF (web application firewall) que bloquee requests

repetidas, y blinde el sistema, reduciremos las probabilidades de que este colapse por un ataque de DDOS (denial of services). Ya que como mostraremos más adelante, la única tensión que aparece en el sistema se da cuando se satura el procesador de reservas.

Es decir, como contemplamos modificabilidad, extensibilidad y seguridad, al escalar el servicio mediante múltiples copias de cómputos, estaremos garantizando la performance y la disponibilidad. Generando un sistema que cumple con los requerimientos y atributos de calidad esperados.

## Vista de Módulos

En esta sección, veremos las distintas vistas de los módulos, enfatizando en las vistas de descomposición, y las vistas de usos.

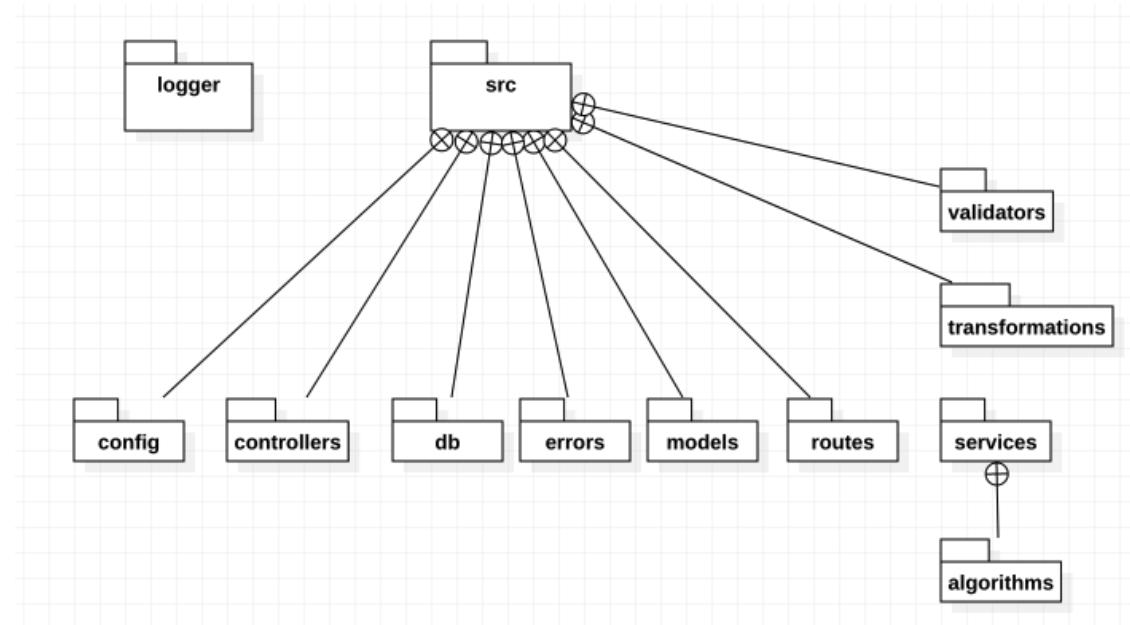
### Vista de Descomposición

Pasemos a ver las vistas de descomposición para cada una de nuestras aplicaciones.

#### 1) Processor Service

##### Representación Primaria

Es la aplicación Node que se encarga de procesar las reservas.



##### Catálogo de Elementos

Catálogo de Elementos	
Elemento	Responsabilidades
logger	Es el módulo que utilizamos para implementar el logueo. Este tiene dos implementaciones, Winston y log4js. Se utiliza en todas las aplicaciones Node como un módulo externo.
src	Contiene todo el resto de los módulos.
config	Es el módulo que contiene los archivos de configuración. En estos se almacenan configuraciones, como las conexiones a las bases de datos, los endpoints terceros necesarios, puerto a utilizar, etc. Además se encarga de inicializar una estructura con la información del

	archivo de configuración correspondiente al entorno en el que se corre.
routes	Es el módulo que contiene las rutas que expone la aplicación Node. Delega en los controllers (siguiente módulo).
controllers	Es el módulo que se encarga de: <ul style="list-style-type: none"> <li>- mandar la request al servicio que corresponda</li> <li>- catchear las excepciones cuando ocurren</li> <li>- armar la respuesta de la request (seteando el estado y el mensaje o body).</li> </ul>
services	Es el módulo que se encarga de ejecutar la lógica, realizar operaciones necesarias, y del manejo de datos.
services/algorithms	Es el módulo que contiene los algoritmos intercambiables (implementaciones específicas de cada país). En este caso, de los criterios para realizar las reservas (seleccionar vacunatorios -y de qué manera hacerlo-).
error	Es el módulo que contiene errores creados por nosotros con estados acordes y mensajes descriptivos.
db	Es el módulo que contiene la inicialización de la/las bases de datos MongoDB a utilizar.
validators	Es el módulo que contiene las diferentes validaciones por las que pasa la reserva. Contiene la clase <i>abstract-validator</i> y sus implementaciones. En caso de crear una nueva implementación de un filtro, deberá extender de <i>abstract-validator</i> y ser colocado en esta carpeta.
transformations	Es el módulo que contiene las diferentes transformaciones por las que pasa la reserva. Contiene la clase <i>abstract-transformation</i> y sus implementaciones. En caso de crear una nueva implementación de una transformación, deberá extender de <i>abstract-transformation</i> y ser colocado en esta carpeta.

## Justificaciones de Diseño

### Justificación genérica de módulos para todos los servicios:

Al construir la aplicación se tomó la decisión de hacer una separación de los módulos en función de la funcionalidad que cada uno debía desempeñar. Esto favorece la modificabilidad debido a que simplificará la incorporación de nuevas funcionalidades al sistema, y además minimizará el impacto que nuevas incorporaciones puedan tener sobre la solución ya existente. A su vez, esta separación favorece a la alta cohesión y al bajo acoplamiento ya que no se encuentran relacionados módulos que no deben estarlo por no requerirse entre sí. No pusimos la lógica de negocio en los controladores. La correcta organización de

la estructura evitara la duplicación de código, mejorará la estabilidad y, potencialmente, ayudará a escalar los servicios.

## Config

Creamos un archivo de configuración para cada entorno en el que se ejecutara la aplicación: develop, staging y live (`env_dev.json`, `env_stg.json`, y `env_live.json`). En estos se permiten configurar:

- La app (nombre, puerto, address, implementación de algoritmos -en este caso, uruguay-)
- Conexión con Redis
- Conexión con MongoDB (para ambas bases de datos)
- Endpoints a usar (auth service, admin service, id provider, notification service)

De esta manera, **favorecemos la modificabilidad** ya que estamos usando la **táctica diferir enlaces**.

Relativo a los diferentes entornos de ejecución, un dato anecdótico. Es de público conocimiento que el sistema de agenda de vacunas de uruguay, unas horas antes de habilitar la agenda para menores de 18 años, habilitó el sistema para una prueba. De esta manera, algunos menores de 18 años encontraron el servicio corriendo y pudieron anotarse.

*“...El pasado 7 de mayo el sistema de agenda sufrió un error y permitió que un grupo de jóvenes menores de edad lograran agendarse para la vacuna. El problema tuvo que ver con la realización de una prueba para la “incorporación de nuevas personas a la agenda”, según consignó El Observador...”* - Montevideo Portal. [Link a nota](#). Esto es un error grosero que no debería pasar en sistemas profesionales, y si el sistema de vacunación hubiese tenido un entorno de testing, puede que esto no hubiera sucedido.

## Errores

En cuanto al módulo de errores, definimos un tipo de error común para toda la aplicación, y dentro de ese tipo de error definimos diferentes implementaciones. De esta manera todos los errores del sistema tienen la misma “forma”, un mensaje descriptivo y un estado. A su vez, al implementar una misma estructura en todas las aplicaciones del sistema, logramos uniformidad, un código más sencillo de entender.

## Logger

El logger es el encargado de registrar los logs informativos y de error, y agregarlos a los archivos `combined.log` y `error.log` respectivamente. Decidimos incluir el logger como una librería externa, para interferir lo menos posible con el código de las aplicaciones. En caso de querer utilizarlo, únicamente se debe realizar “`npm i`” dentro de la carpeta del logger, y ya se estará contando con dos implementaciones listas para usarse. Nuevas implementaciones pueden ser agregadas fácilmente haciendo que extiendan de la clase `AbstractLogger` y eligiéndola como

implementación en el archivo de configuración. Al implementar el logger de esta manera estamos **cumpliendo con el Req #5** el cual establece que la solución debe contar con la posibilidad de poder cambiar las herramientas o librerías concretas que se utilicen para producir la información acerca de gestión de fallas y errores con el menor costo posible. Si en un futuro se desea implementar de distinta manera los demás módulos no se verán impactados, solo se modifica el propio Logger.

#### Justificación específica de módulos del Processor Service:

#### Algorithms

El módulo *algorithms*, puede contener diferentes archivos. Cada uno de estos es una implementación distinta del algoritmo de asignación de vacunatorios. Tomemos como ejemplo Uruguay: existe un archivo uruguay.js.

De esta forma, estamos **cumpliendo con la modificabilidad del Req #10** que indicaba que cambiar el algoritmo de asignación de vacunatorios debería tener el menor impacto posible. En nuestro caso, el impacto de cambio estaría solo dentro de este módulo: *algorithms*, alineándonos con las **tácticas de modificabilidad dividir módulo y aumentar coherencia semántica**. El cambio no se expande por el resto de la aplicación.

El algoritmo de asignación de vacunatorios, según entendemos (y según definimos) tiene partes comunes, sin importar en qué país se implemente el sistema. Siguen la siguiente forma:

- 1) Filtros (validaciones) y transformaciones
- 2) Búsqueda de Cupo
- 3) Persistencia de la reserva
- 4) Notificación de la reserva al usuario

Pero cada implementación ejecutará estos pasos de manera singular. Y queremos que *cambiar dicho algoritmo, tenga el menor impacto posible*. Por eso, tomamos la siguiente decisión: priorizamos que al implementar un nuevo cambio, este implique codificar lo menos posible y tenga el menor impacto en responsabilidades que no sirven el mismo propósito. La decisión que tomamos implica cambiar únicamente la query a mongo para buscar los vacunatorios y cuotas (lugares libres) posibles. Esta decisión tiene sentido por 3 razones:

- Mongo es un lenguaje muy potente, por lo que creemos que la gran mayoría de las veces su potencia será suficiente para lo que el país que implementa el sistema desee.
- Es muy sencillo de cambiar, haciendo que implementar el sistema para un nuevo país (ej: Chile), implique casi ninguna modificación: los filtros y sorting de la query.

- Si bien estamos acoplados a mongo, difícilmente cambiemos el motor de base de datos por las ventajas en materia de flexibilidad, potencia de lenguaje y performance.

De todas formas, todas las reservas, sin importar la implementación, deberán tener los siguientes atributos para que el sistema funcione:

- `reservation_code` (un uuid idealmente o cualquier código identificar)
- `document` (documento para identificar al ciudadano)
- `status` (estado de la reserva, un string que lo indique)
- `date` (fecha en la cual se asignó la reserva)
- `quota_code` (solo cuando se le asigna un cupo: el código de dicho cupo)
- `vaccination_center_code` (solo cuando se le asigna un cupo: el código de vacunatorio donde se asignó dicho cupo)

Además, todas las cuotas deben tener los siguientes atributos:

- `quota` (cantidad de cupos disponibles)
- `quota_code` (uuid o cualquier código que identifique a la quota)
- `center_code` (el código que identifique al vaccination center asociado a la quota)

(Cuando en este documento se hable de “quota” o “cuota”, estaremos haciendo referencia a los nuevos cupos relativos a un vacunatorio. Ejemplo: “el administrador del sistema agregó una nueva cuota para el vacunatorio 3, donde se filtra por edad. Asignó 2000 cupos”).

Esto es una parte crucial de nuestro sistema, y que apoya muchísimo a la **modificabilidad y extensibilidad**. Como vemos, siempre que nuestras reservas y quotas tengan dichos campos básicos, el sistema funcionará. Además, pueden tener muchos otros campos específicos de la implementación. Estos campos serán manejados y utilizados en las funciones implementadas en el módulo algorithms. Y también en el resto de las aplicaciones.

Para hacer la aplicación clara y extensible, creamos los módulos de validaciones y transformaciones. Como lo dice su nombre, dentro del módulos de validaciones se encuentran las validaciones existentes y en el de transformaciones las transformaciones existentes. Cada una se implementa en un archivo .js separado. Los utilizaremos como una especie de filtros por los que van a pasar las reservas entrantes validando que sus datos sean correctos y realizando las transformaciones necesarias. En el momento que una validación de error, inmediatamente se descarta la reserva y retornando el error al usuario.

Nuevas implementaciones pueden ser agregadas fácilmente extendiendo de las clases `abstract-validator` y `abstract-transformation`, almacenadas en el módulo que correspondan. Las validaciones tienen un atributo de tipo booleano ‘atEnd’, además

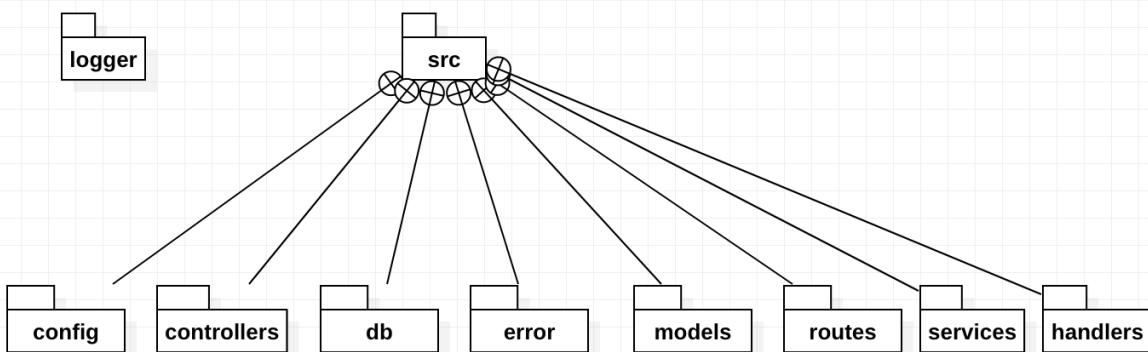
de nombre y descripción. De ser seteado como `true`, dicha validación se ejecutará al final del proceso. Esto atiende la **performance**, ya que evita que todas las reservas entrantes consuman recursos.

Por último, queremos aclarar que tomamos la decisión de usar mongo (y la librería mongoose para Node), pero no utilizamos Schemas, ya que nos limitan en modificabilidad. No queremos atarnos a una implementación de “reservas” (ni de vacunatorio, ni de quota). Por eso, cada implementación debe encargarse de validar que los elementos tengan los atributos necesarios y con las formas adecuadas.

## 2) Admin Service

### Representación primaria

Es la aplicación Node que se encarga de manejar las acciones de los administradores y vacunadores, incluyendo la configuración del sistema.



### Catálogo de Elementos

Catálogo de Elementos	
Elemento	Responsabilidades
logger	Es el módulo que utilizamos para implementar el logueo. Este tiene dos implementaciones, Winston y log4js. Se utiliza en todas las aplicaciones Node como un módulo externo.
models	Es el módulo donde se definen las estructuras de datos que se utilizaran en la aplicación. En este caso, por ejemplo: configuration, given_vaccines, quota, user, vaccination_center, validations.
routes	Es el módulo que contiene las rutas que expone la aplicación Node. A diferencia del <i>Reservation Processor</i> , la mayoría de las rutas en esta aplicación requieren autenticación.
services	Es el módulo que se encarga de ejecutar la lógica, realizar operaciones necesarias, y del manejo de datos. A diferencia del <i>Reservation Processor</i> , y teniendo en cuenta que la mayoría de rutas de esta aplicación requieren autenticación, este módulo también se encarga de validar que el usuario en control tenga los privilegios necesarios para realizar la acción que desea.
handlers	Es el módulo que se encarga de validar que la request contenga un header (llamado 'auth-token') con un token válido.
No creemos pertinente documentar el resto de los módulos, ya que son análogos a los del <i>Reservation Processor</i> . Todas las aplicaciones se desarrollaron utilizando una misma estructura: route ⇒ controller ⇒ service ⇒ data config    app    errors    models (if needed)	

Por lo que la responsabilidad de los módulos es siempre análoga.

### Justificaciones de Diseño

Recordamos que el Admin Service es el servicio encargado de la gestión del sistema, de los usuarios del sistema, de los vacunatorios y de los cupos. Tiene las responsabilidades de:

- Crear vacunatorios
- Mantener cupos para los vacunatorios
- Gestionar la configuración del sistema (validaciones, endpoints a utilizar para notificaciones...)
- Registro y Sesión de usuario (administradores y vacunadores)
- Marcar ciudadanos como vacunados
- Consultas sobre usuarios vacunados y pendientes

Para este servicio también aplican las justificaciones llamadas genéricas en la vista de módulo del Processor y decidimos no repetirlo.

### Handlers

Para la mayoría de los endpoints que esta aplicación expone, se requiere estar autenticado, y autorizado. Para encargarnos de validar que la request contenga un header auth-token, decidimos crear un módulo aparte handler, para separar responsabilidades, ser más cohesivos y extensibles.

### Models

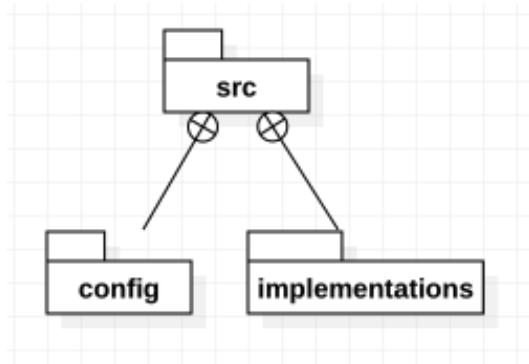
Una característica del Admin Service, es su modificabilidad. No utilizamos Schemas (de mongoose) para persistir la información. El admin service podría (o no) utilizar schemas para definir sus estructuras. De esta forma, cualquier implementación nueva, que cambie la estructura de cualquiera de estos elementos, podrá hacerlo con el menor impacto de cambio (casi nulo). Es decir, modificando el archivo correspondiente en el módulo models.

### Services

Este módulo se encarga de realizar la lógica requerida y el acceso a datos necesario. Pero antes, para los endpoints que requieran autenticación (la mayoría en esta aplicación), autorizará al usuario. Validará que el usuario tenga los privilegios necesarios para ejecutar la acción en cuestión.

### 3) Logger

#### Representación primaria



#### Catálogo de elementos

Catálogo de Elementos	
Elemento	Responsabilidades
src	Contiene todo el resto de los módulos.
implementations	Es el módulo que contiene las implementaciones de <code>AbstractLogger</code> , que pueden ser utilizadas con la capacidad de alternar entre una y otra en tiempo de ejecución.
config	Es el módulo que contiene los archivos de configuración. En estos se almacenan configuraciones, puerto a utilizar, etc. Además se encarga de inicializar una estructura con la información del archivo de configuración correspondiente al entorno en el que se corre.

#### Justificaciones de Diseño

Decidimos implementar el logger como una carpeta a parte, la cual fuera completamente independiente de la aplicación que lo utilizará, y que se tuviera que únicamente ejecutar “npm install” dentro de la carpeta del logger y agregar en el configuración los parámetros de configuración necesarios, para determinar la implementación a utilizar. De esta manera **cumplimos con el requerimiento 6** que obliga a la fácil reutilización de la implementación en otras aplicaciones con el menor impacto posible en el código.

Todas las implementaciones se extienden y deben cumplir con las operaciones de `AbstractLogger`. Esto permite que pueda implementarse **Differ Binding**, haciendo que el usuario del logger tenga que únicamente importar una clase “logger” que se encuentra en la raíz de este proyecto, cuyo único propósito es la invocación de la solución especificada en el archivo de configuración de la aplicación.

Como todas las implementaciones están obligadas a contar con las mismas operaciones, es posible alternar entre las distintas soluciones, e inclusive agregar nuevas implementaciones fácilmente.

Lo que nos permite haber implementado logging a lo largo y ancho del código, es cumplir con el **Req #3 (gestión de errores y fallas)**, y **favorecer la seguridad** del sistema a la vez que la **recuperación de fallas**. De esta forma podemos **auditar** y dar seguimiento a los problemas.

Aprovechamos para aclarar que nos hubiera encantado tener un sistema completo y funcional, pero por falta de tiempo / recursos preferimos desarrollar enfatizando en lo que aportaba valor. las actividades que eran más repetitivas, pero demandantes con respecto a tiempo, documentarlas.

En lo que respecta a los logs, escribirlos a lo largo de todos los servicios llevaría mucho tiempo. Incluso, con más tiempo se hubiera pensado desarrollar algún mecanismo para escribir los logs sin que demandara tanto tiempo.

Los mismos fueron implementados de manera completa en algunos lugares del sistema pero no en todos. Sin dudas que esta es una tarea que si la hubiéramos implementado entera nos hubiera quitado horas que destinamos a otros fines (como ser otros requerimientos, o documentación).

Sin embargo, queremos mostrar cómo deberían hacerse los logs a lo largo de todo el sistema para que estos tengan sentido y cumplan su objetivo. Un claro ejemplo de loggear correctamente es el que se puede encontrar más abajo (presente en el servicio de AdminService), en el cual se detalla la acción que se realizó, quien la realizó (ya que es un endpoint que requiere autenticación), cuando la realizó (dia y hora exacta), y en qué archivo se persiste dicho log. Además se puede ver en el ejemplo un log de error que se mostrará en un archivo aparte de log de errores.

```

const markUserAsVaccinated = async (req) => {
  const centerCode = req.query.vaccination_center_code;
  const document = req.query.document;
  const vaccinationDate = req.query.vaccination_date;

  logger.info(
    "The user with id " +
    req.user_in_control.user_id +
    " requested to mark user with id " +
    req.query.document +
    " as vaccinated."
  );

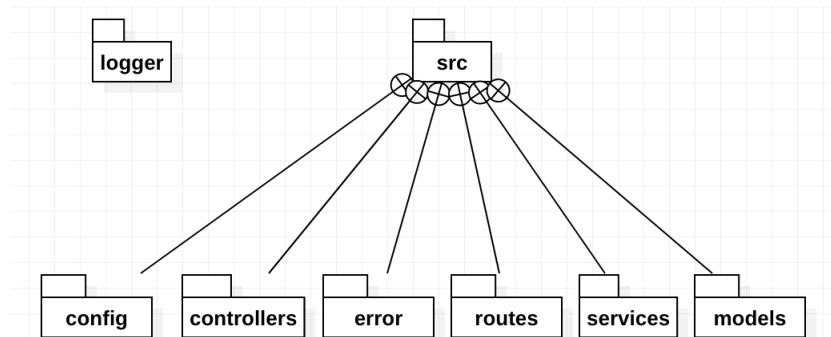
  //1) Validate User in Control Privileges
  if (
    req.user_in_control.auth_client_id != "superadmin" &&
    req.user_in_control.auth_client_id != "vaccinator"
  ) {
    logger.error(
      "The user with id " +
      req.user_in_control.user_id +
      " could not set person as vaccinated as his role is not superadmin or vaccinator. current role:
" +
      req.user_in_control.auth_client_id,
      __filename.split("src")[1]
    );
    throw new vaccinatedUserErrors.ErrorSettingUserAsVaccinated(
      "only 'superadmin' or 'vaccinator' can set user as vaccinated"
    );
  }
}

```

Este proceso puede parecer trivial y sencillo pero requiere de analizar si el endpoint requiere autenticación o no, qué mensaje loguear, y posteriormente ser testeado para garantizar que en el log no se refiera a algo nulo, y que por esto la aplicación caiga. Debido a esto, sumando la falta de tiempo, preferimos documentarlo y no implementarlo exhaustivamente.

### 3) Auth-Service

#### Representación primaria



#### Catálogo de elementos

Catálogo de Elementos	
Elemento	Responsabilidades
models	Módulo que se encarga de definir la estructura de los tokens (tanto para retornar al usuario como para persistir en memoria).
No creemos pertinente documentar el resto de los módulos, ya que son análogos a los del <i>Reservation Processor</i> . Todas las aplicaciones se desarrollaron utilizando una misma estructura: route ⇒ controller ⇒ service ⇒ data config    app    errors    models (if needed) Por lo que la responsabilidad de los módulos es siempre análoga.	

#### Justificaciones de Diseño

Para este servicio también aplican las justificaciones llamadas genéricas en la vista de módulo del Processor y decidimos no repetirlo.

#### Models

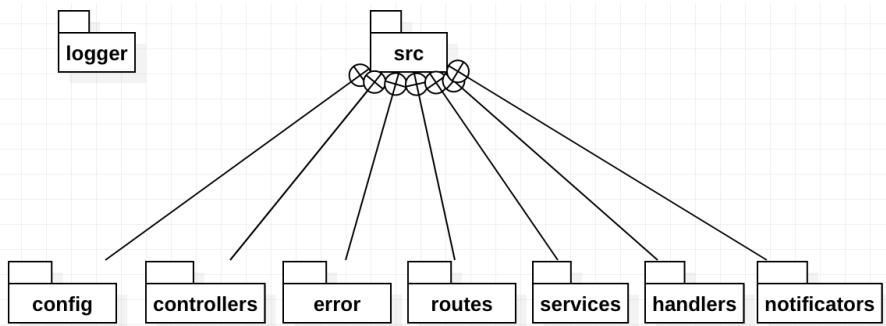
Esta es la única aplicación en la cual utilizamos schemas adrede. Los usamos con el fin de estandarizar los tokens. Los tokens son una parte fundamental de nuestro sistema ya que son el elemento que permite (o no) realizar cierta acción. Para que el sistema funcione de manera correcta, estos deben ser consistentes, y deben tener la información necesaria. Para esto los definimos con la siguiente estructura:

- Full Token Data
  - Se encarga de almacenar toda la información relativa a un token. Este no es el payload del token, sino que contiene otras claims más. Este elemento se le retorna al usuario al hacer un verify.
  - Esto es lo que persistimos en caché asociado al access token.

- Cuenta con: user\_id, auth\_client\_id (rol), expire\_date y podría tener más claims críticas, que no queremos que viajen en el payload.
- Token Data
  - Es la información que lleva el payload del token. En este caso, es lo mismo que tiene el full token data. Pero generalmente tiene menos claims.
  - No puede contener información sensible.
- Token Details
  - Es lo que se le retorna al usuario cuando este se loguea (o cuando genera el token).
  - Contiene el access token y su expiración. Si se desea, se le puede agregar un refresh token con su expiración.

## 4) Notification Service

### Representación primaria



### Catálogo de Elementos

Catálogo de Elementos	
Elemento	Responsabilidades
service/notifiers	Módulo que contiene las distintas implementaciones de notificadores.
handlers	Es el módulo que se encarga de validar que la request contenga un header (llamado 'auth-token') con un token válido.
No creemos pertinente documentar el resto de los módulos, ya que son análogos a los del <i>Reservation Processor</i> . Todas las aplicaciones se desarrollaron utilizando una misma estructura: route ⇒ controller ⇒ service ⇒ data config    app    errors    models (if needed) Por lo que la responsabilidad de los módulos es siempre análoga.	

### Justificaciones de Diseño

Para este servicio también aplican las justificaciones llamadas genéricas en la vista de módulo del Processor y decidimos no repetirlo.

Recordamos que esta aplicación se encarga de las notificaciones a usuarios finales.

Dentro del módulo notifiers se puede encontrar nuestra única implementación de notificador: sms\_emulator (la que se pide en la letra, imprime en consola las notificaciones).

Para seleccionar qué notificador usar, el módulo services se encarga de seleccionar el notificador adecuado para notificar al usuario. Para esto, según la configuración del sistema, importará cierto archivo de la carpeta service/notifiers. Este archivo debe implementar los siguientes métodos:

- notifyPending(notification)

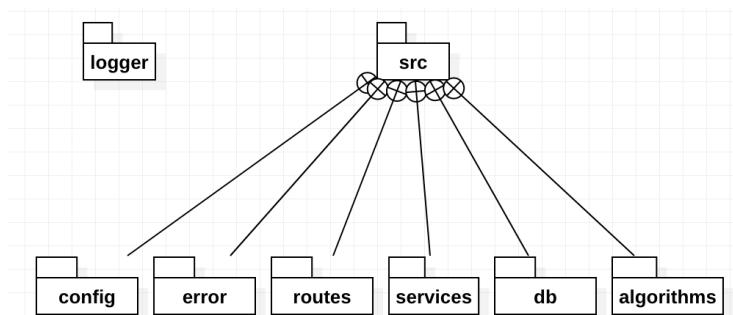
- notifySuccessful(notification)
- notifyCancelled(notification)

Si en el futuro se desea notificar mediante whatsapp, o mediante email, es tan simple como agregar un nuevo archivo dentro de esta carpeta, que se llame “whatsapp.js” o “email.js” respectivamente que implemente las funciones detalladas previamente. Este cambio es simple y afecta únicamente al módulo notifiers, por lo que podemos afirmar que cumplimos con el **requerimiento 13** que planteaba poder enviar mediante otro proveedores de mensajería con el menor impacto de cambio.

Pero esta aplicación es muy extensible, y tan performante como se desee (según las implementaciones que se hagan). Además, el servicio tiene la característica que puede modificarse en tiempo de ejecución, ya que agregar un nuevo archivo dentro del módulo *service/notifiers* no requiere reiniciar el sistema.

## 5) Pending Service

### Representación primaria



### Catálogo de elementos

Los mismos que en vistas anteriores.

### Justificación de diseño

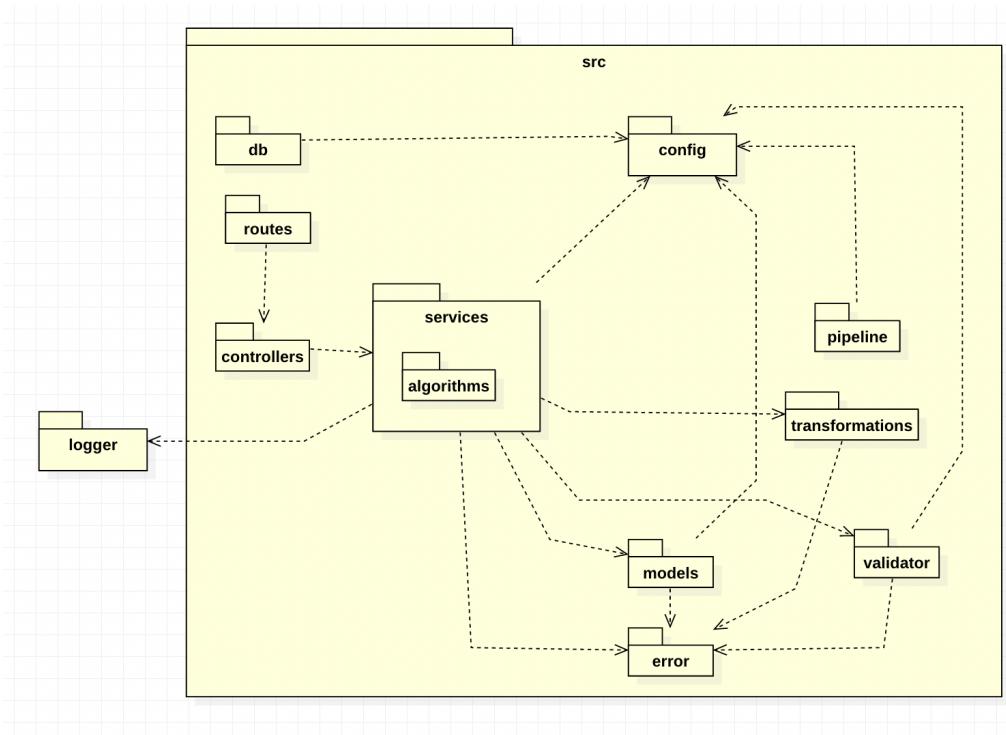
Para este servicio también aplican las justificaciones llamadas genéricas en la vista de módulo del Processor y decidimos no repetirlo.

La diferencia que vemos en este servicio, es que no existe el módulo controller porque este consume de una cola de nuevo cupos agregados, no expone ningún endpoint. El módulo route tampoco tiene que estar necesariamente. El módulo algorithms tiene exactamente el mismo propósito que en el Processor, solo que en este caso dicho algoritmo determina a qué personas se asigna primero cuando van llegando nuevos cupos.

## Vista de Usos (tomando en cuenta dependencias entre módulos)

Presentaremos algunas vistas de uso diferentes, los catálogos de elementos son los mismos que en la vista de módulos, y las justificaciones de diseño son las mismas para todos los diagramas que siguen.

### 1) Processor Service

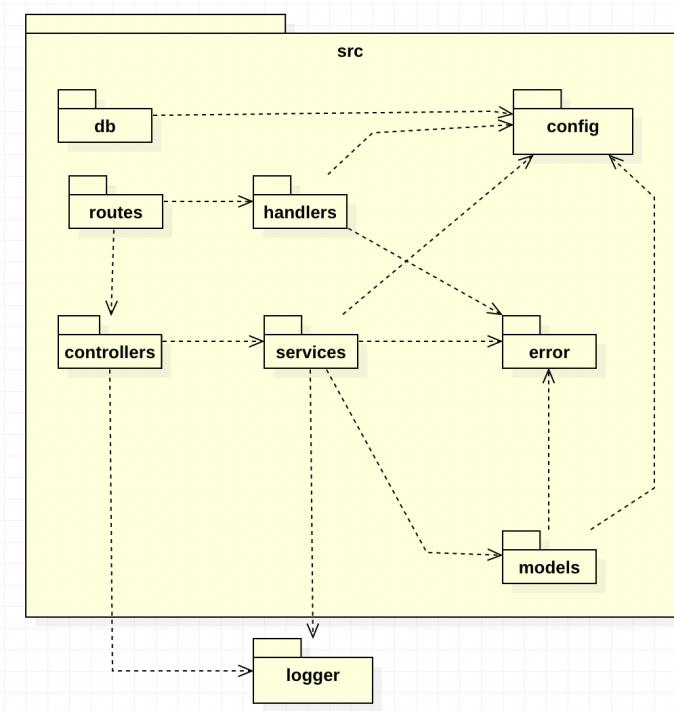


#### Justificación de diseño

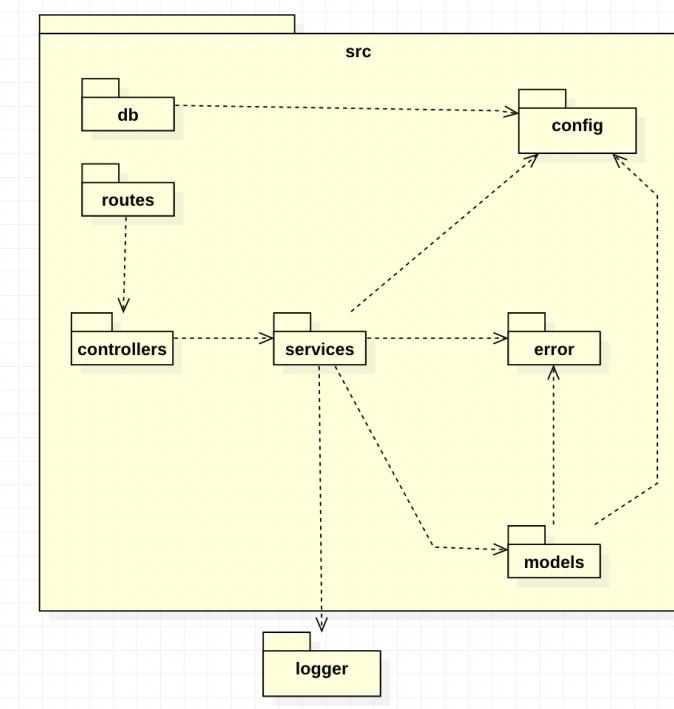
Se redujo el acoplamiento usando la táctica modificabilidad restricción de dependencias, para restringir los módulos con los que cada uno interactúa y así reducir la probabilidad de que un cambio en un módulo se propague a otros. Tal como se puede observar en esta representación primaria de la vista, no se cuenta con una gran cantidad de dependencias de uso entre módulos (exceptuando el caso del Logger, módulo utilizado para la recopilación de las actividades y errores acontecidas en toda la aplicación y el config que contiene las configuraciones del sistema).

Además, intentamos perseguir y tuvimos siempre presente la táctica de modificabilidad de aumentar la coherencia semántica ya que cada responsabilidad que no tiene un mismo propósito, se encuentra en módulos distintos.

## 2) Admin Service



## 3) Otros Servicios genéricos

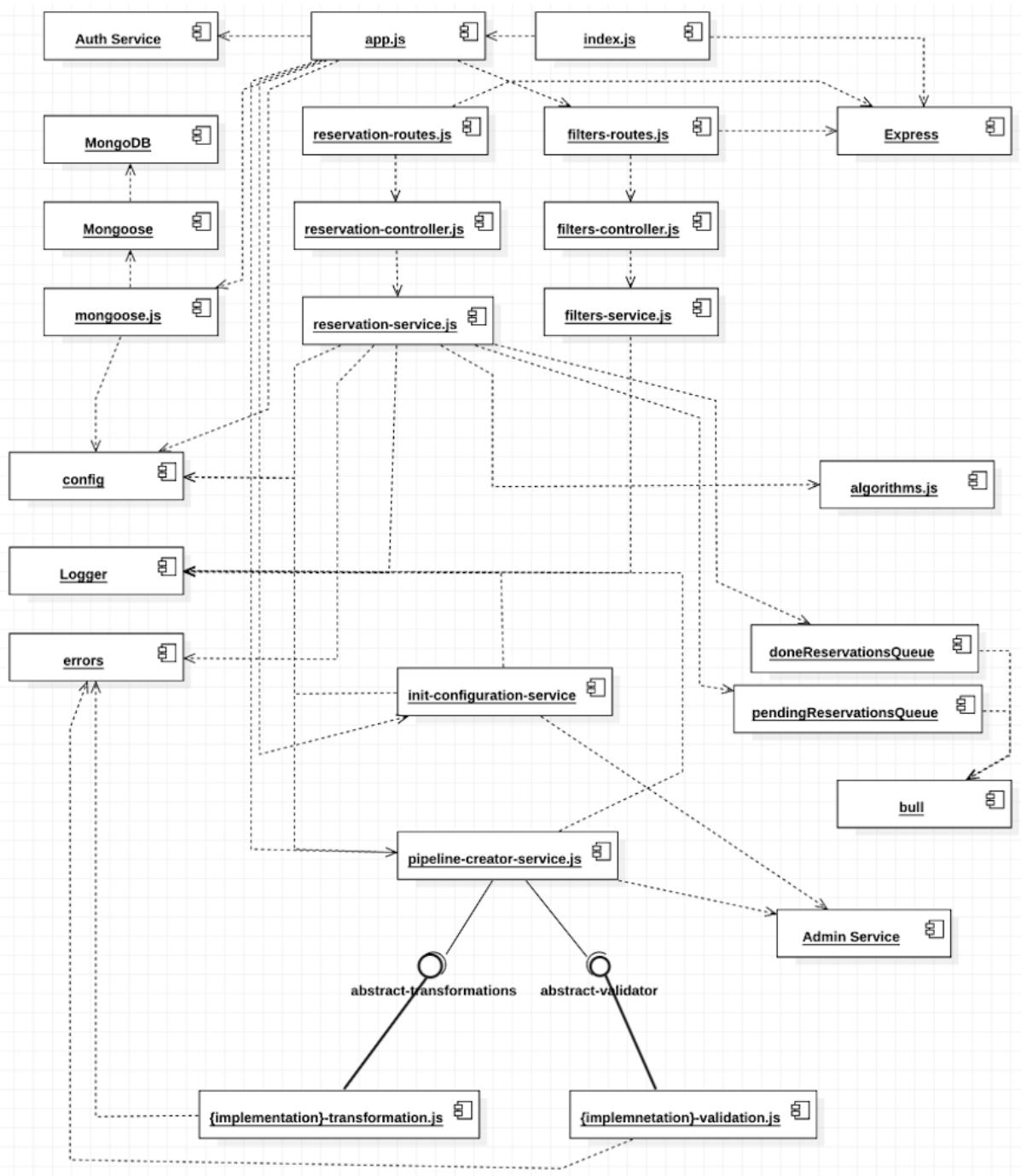


## **Vista de componentes y conectores**

Debemos tener en cuenta que por la naturaleza de Javascript y más particularmente NodeJS, consideraremos cada archivo con extensión **.js** como un componente ya que es una unidad suficiente para obtener una ejecución. Por este motivo, en los diagramas de componentes se encontrarán una gran cantidad de componentes pequeños.

## Processor Service

### Representación primaria



### Catálogo de elementos

Catálogo de elementos	
Elemento	Responsabilidades
logger	Componente, que basado en la configuración del sistema (config files, ya que utilizamos <b>Defer Binding</b> ), selecciona y utiliza una

	<p>implementación de Logger.</p> <p>Tiene como responsabilidad registrar eventos y errores ocurridos en el sistema según la implementación seleccionada</p>
<b>index.js</b>	Es el componente que usamos para inicializar la ejecución de la aplicación. Lo hacemos mediante una llamada al componente app.js que se encargará de inicializar las tecnologías necesarias y de que la aplicación comience a escuchar en el puerto especificado.
<b>app.js</b>	Es el componente que se encarga de inicializar todas las tecnologías y conexiones que la aplicación necesite (bases de datos o colas de mensajes) a la vez que hacer que la aplicación escuche en el puerto especificado. Además, en este se definen parsers para los requests (a json), se setean al “req” las validaciones, transformaciones y configuraciones a usar. Por último, y no menos importante, se encarga de distribuir las requests a los routers (las rutas) correspondientes.
<b>mongoose.js</b>	Es el componente que se encarga de definir la inicialización y manejar la conexión a la base de datos de mongo. Para esto, utiliza la configuración de los archivos de configuración.
<b>errors.js</b>	<p>Es el componente que define el error genérico del Processor. A su vez, tiene las implementaciones de errores concretos relacionados a las diferentes responsabilidades de la aplicación. La estructura de este error genérico involucra:</p> <ul style="list-style-type: none"> <li>- Un nombre</li> <li>- Un mensaje</li> <li>- Un código de error (status)</li> </ul> <p>Esto es extensible y fácilmente modificable. Estos errores son utilizados por los logs, y para retornar a los usuarios en la response de la request.</p>
<b>config.js</b>	Componente que se encarga de obtener la configuración del sistema en base al entorno de ejecución. Hace esto consumiendo el archivo de configuración correspondiente: <code>env_&lt;entorno&gt;.js</code>
<b>init-configuration-service.js</b>	Se encarga de, en la inicialización de la aplicación, conseguir mediante una request al Admin Service la información relacionada con la configuración del sistema (endpoints a utilizar para notificar y para validar identidad).
<b>pipeline-creator-service.js</b>	<p>Si bien su nombre no es el mejor (ya que no se trata de una pipeline, sino de una cadena de filtros de la que se puede salir en cualquier momento), este componente es ejecutado al inicializar la aplicación (en el componente app).</p> <p>Es responsable de generar la cadena de validaciones y</p>

	<p>transformaciones a ejecutar para las reservas que llegan al sistema. Una vez generada, la retorna (para ser utilizada).</p> <p>Esto se hace una única vez: en la inicialización. Y se obtienen mediante una request al Admin Service, pidiendo por los filtros y validaciones almacenados en la configuración del sistema.</p>
<b>filters-routes.js</b>	Componente que se encarga de definir las rutas que expone el servicio, en este caso dicha ruta brinda las validaciones y transformaciones implementadas en el sistema.
<b>filters-controller.js</b>	<p>Componente encargado de redirigir las requests relacionadas a los filtros al servicio filters-service. Además, se encarga del cacheo de errores.</p> <p>También es responsable de la elección y el tipo de mensaje a retornar en la response al usuario (tanto para errores como para casos de éxito).</p>
<b>filters-service.js</b>	Es el responsable de la lógica de las requests que llegan al sistema relativas a los filtros. Concretamente, contiene la lógica necesaria para retornar una lista con todas las validaciones y transformaciones existentes en el sistema.
<b>reservation-routes.js</b>	Componente que se encarga de definir las rutas donde se atenderá lo relacionado a la recepción de registros de reservas.
<b>reservation-controller.js</b>	<p>Componente encargado de redirigir las requests relacionadas a las reservas al servicio reservation-service. Además, se encarga del cacheo de errores.</p> <p>También es responsable de la elección y el tipo de mensaje a retornar en la response al usuario (tanto para errores como para casos de éxito).</p>
<b>reservation-service</b>	<p>Es el responsable de la lógica de las requests relativas a nuevas reservas. Su responsabilidad es validar la reserva, intentar agendarla, y en base a los resultados anteriores, delegar al Emitter para persistir la reserva, y delegar al Notification Service para notificar al usuario (y por último retornar la reserva generada).</p> <ul style="list-style-type: none"> <li>- Debe asegurarse que la reserva es válida, por eso utiliza los filtros generados por el componente <i>pipeline-creator-service.js</i> en la inicialización de la aplicación.</li> <li>- Luego, buscar cupos para dicha reserva.</li> <li>- Según el resultado del paso anterior, encola la reserva en la cola de pendientes o en la de aceptadas.</li> <li>- Según el resultado del paso anterior, delega al notification service la responsabilidad de notificar al usuario (en base a la configuración del sistema).</li> <li>- Por último, retorna la reserva generada (con los datos de esta acorde a los pasos anteriores)</li> </ul>

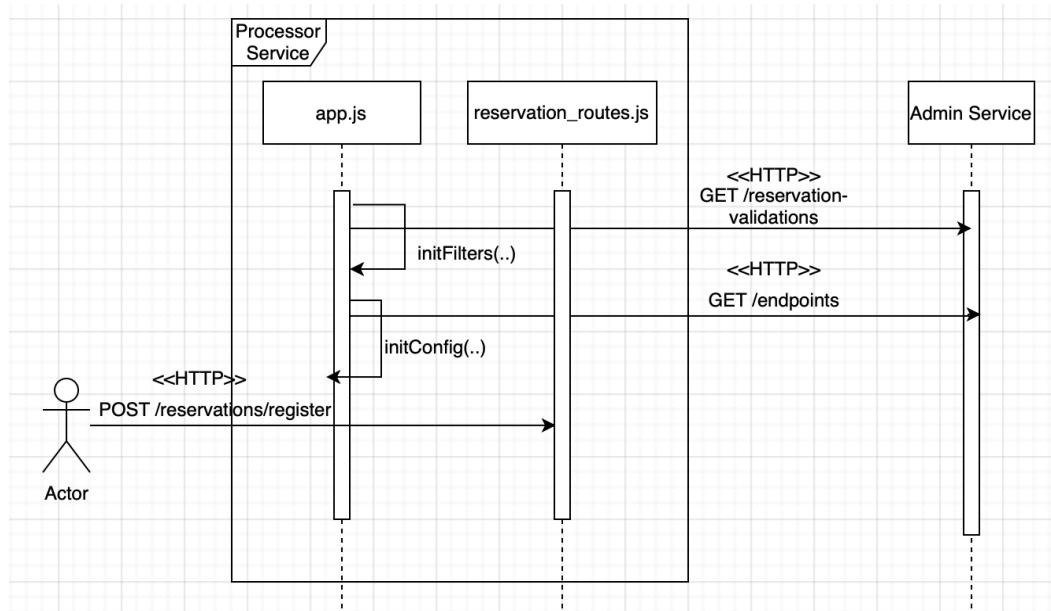
	Para hacer esto, se apoya del componente definido en <code>algorithms/&lt;implementación&gt;</code>
<code>algorithms/&lt;implementation&gt;.js</code>	<p>Su responsabilidad es definir las singularidades del algoritmo de asignación de vacunatorio. Es decir, brindar extensibilidad y modificabilidad al sistema. Para esto, debe estar referenciado en los archivos de configuración que implementación se utilizará (el nombre del componente, ej: uruguay)</p> <p>Concretamente, debe:</p> <ul style="list-style-type: none"> <li>- definir los filtros a utilizar en las queries para obtener los cupos de la base de datos (mongo)</li> <li>- los filtros para ordenar dichas consultas (sorting)</li> <li>- un mecanismo para transformar la reserva obtenida en la request en una reserva manejable por nuestro sistema.</li> </ul>
<code>abstract-transformation.js</code>	Es el componente que estandariza la implementación de transformaciones. Contiene el método <code>Name()</code> que retorna el nombre de la clase, <code>Description()</code> que retorna una descripción de que hace y como funciona la transformación, y <code>Transform()</code> que retorna una función que transformará el dato provisto.
<code>&lt;implementation&gt;.transformation.js</code>	Son los componentes que implementan <code>abstract-transformation</code> . Cada uno tiene una responsabilidad particular.
<code>abstract-validator.js</code>	Es el componente que estandariza la implementación de validaciones. Contiene el método <code>Name()</code> que retorna el nombre de la clase, <code>Description()</code> que retorna una descripción de que hace y como funciona la transformación, y <code>Validate()</code> que retorna una función que valida si el dato es correcto.
<code>&lt;implementation&gt;.validator.js</code>	Son los componentes que implementan <code>abstract-validator</code> . Cada uno tiene una responsabilidad particular.
<code>doneReservationQueue</code>	Componente externo del cual el Emitter Service consume para guardar las reservas hechas.
<code>pendingReservationQueue</code>	Componente externo del cual el Emitter Service consume para guardar las reservas pendientes.
<b>Auth Service</b>	Es un componente externo al Processor Service. Se utiliza para obtener el token para poder usar el Notification Service.
<b>MongoDB</b>	Base de datos no relacional, accedida a través de mongoose. Es la base de datos Admin que almacena los vacunatorios, los cupos de estos vacunatorio, las validaciones y configuraciones.

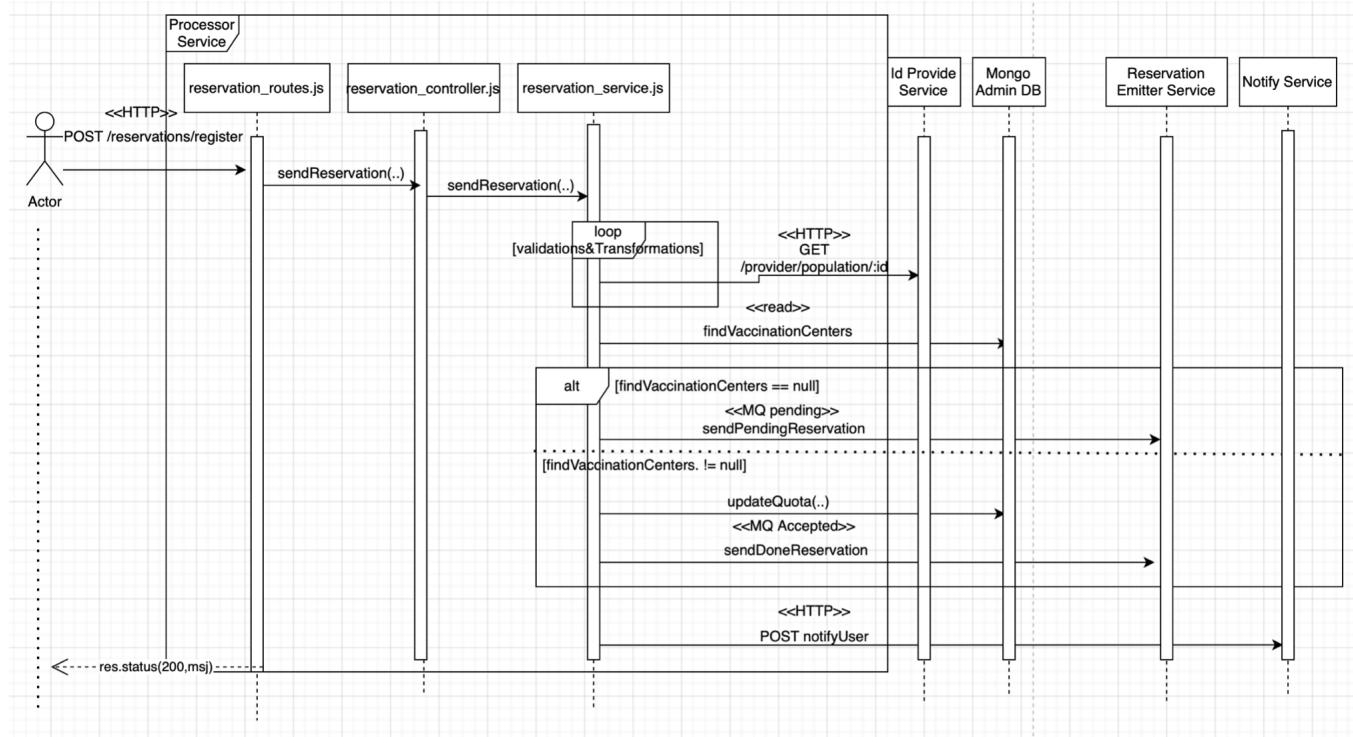
<b>Admin Service</b>	Es un componente externo al Processor Service. Se utiliza para obtener validaciones y configuraciones de endpoints.
----------------------	---

### Comportamiento

Aclaramos que en el diagrama se muestra lo que consideramos interesante del comportamiento, no es algo detallado. Decidimos mostrar una única flecha de respuesta para hacer el diagrama más ameno.

Mostramos el comportamiento de la funcionalidad asignación de reservas, en 2 diagramas. El primero muestra lo que sucede al principio cuando se corre la aplicación. Luego de eso, lo que sucede con las peticiones de reserva entrantes que decidimos mostrarlo en otro diagrama para hacerlo más entendible. El segundo sería la continuación del primero.





## Justificaciones de diseño

Los diagramas de arriba ilustran el comportamiento del *Processor Service* al momento de procesar las solicitudes de reserva.

El atributo de calidad principal que se persigue en el *Processor Service* es la **performance y disponibilidad**. Tratamos de hacerlo lo mas performante posible. Sin embargo, esta aplicación debe realizar muchas tareas para lograr hacer el procesamiento correcto de las reservas (y la mayor parte del comportamiento debe ser sincrónico, por lo que tenemos algunas limitaciones). Empezamos pensando cuáles serían los pasos indispensables y las consultas a datos que fueran indispensables.

- Primero, como se puede ver en el diagrama, sabíamos que teníamos que validar las reservas.
- Segundo que había consultar a la base qué vacunatorios cumplían los criterios, en caso de existir alguno y poder agendar, volver a acceder a la base para restarle 1 cupo a dicha quota.
- Tercero, persistir la reserva (agendada o como pendiente) en una base de datos.
- Ultimo, notificar al usuario en la response a su request y mediante el emulador de sms.

A la primera, ya teníamos 3 accesos a Mongo. Esto sin contar que luego nos daríamos cuenta que dentro de las validaciones estaba el servicio de identificación civil que es otra pegada a un servicio que accede a una base de datos, y otro acceso a la base también para consultar si la persona no tenía reservas generadas. Inmediatamente nos dimos cuenta que con un poco de carga podría saturarse.

Para cumplir con el Req #1 (que establece la importancia de que no se pierdan datos en momentos de carga), en un principio habíamos pensado usar la táctica limit event response poniendo una cola delante del *Processor Service*. Sin embargo, luego nos dimos cuenta que esto no servirá porque el usuario necesita una respuesta y una vez que se manda a la cola, se desentiende de la reserva. De esta manera nos convencimos que la petición de reserva debería ser, en su mayoría, sincronica. Pero el nuevo problema sería cómo hacer para no saturar el endpoint.

La siguiente decisión que tomamos fue usar la **táctica múltiples copias de computo**. De esta manera no solo podremos manejar mejor la carga, sino también **favorecer la performance del Req #11**, el cual establece que la respuesta a la petición de reserva debe darse en una ventana de tiempo entre 30 segundos y 5 minutos. Este tiempo de respuesta es un aspecto que se desea optimizar.

Implementamos esta táctica con el modo cluster de PM2, el cual corre varios procesos (un proceso por cada núcleo que la computadora tenga). También balancea la carga automáticamente, ya que tiene un load balancer incorporado. Si se cae una de las instancias, existen las otras para que sigan aceptando peticiones. De deployar el sistema en AWS, por ejemplo, podríamos escalar prácticamente infinitamente nuestra aplicación (tanto vertical como horizontalmente).

También tomamos decisiones acorde a la **táctica reduce overhead**, que en conjunto contribuyen a tener la aplicación lo más performante posible. Cuando llega una solicitud de reserva al Processor, lo primero que hace es pasar la reserva por un conjunto de “filtros” (repetimos, NO es pipes & filters) que la validan y transforman. Si la reserva no pasa alguna de las validaciones, es inmediatamente descartada. Al tener las validaciones como primer paso, reducimos la carga que tiene que procesar el resto del Processor ya que muchas reservas son descartadas desde el principio. Además, una de las validaciones es garantizar que la persona exista en el servicio de identificación civil, lo que implica realizar una pegada a dicho servicio. Si se tratara de un proyecto real, este servicio de id civil implementado por terceros sería performante y sabría manejar carga, por lo tanto, asumimos que en nuestro caso también sucede esto. Sin embargo esto no es real ya que el servicio de identificación civil fue creado por nosotros, teniendo un impacto real en nuestra performance ya que implica una pegada a otro servicio el cual accede a una base de datos.

Si corremos el *Processor Service* sin utilizar el servicio de identificación civil, se reciben y procesan 10.000 reservas sin ningún problema (muy por debajo de la ventana de tiempo). Cuando lo agregamos, se degrada la performance y comienza a dar socket hung up muchísimo antes. Esto significa que el procesamiento no da a basta. Para contrarrestar esto, decidimos **poner las validaciones más lentas o que consuman más recursos al final**, en este caso la validación del servicio de

identificación civil como última validación para que se llame la menor cantidad de veces posible. Esto lo implementamos agregando un atributo booleano *atEnd* en aquellas validaciones que deseen hacerse al final, además de nombre y descripción. Aquellas que lo tuvieran seteado como true, se ejecutan luego que todos los otros filtros hubieran terminado.

También, como muestra el primer diagrama de comportamiento, cuando se inicializa la aplicación, se hace por única vez la inicialización de los filtros (se hace la pegada al *Admin Service* para traer las validaciones y transformaciones a aplicar de la base) y de las configuraciones de los endpoints (se hace la pegada al *Admin Service* para traer de la base los endpoints configurados del servicio de identificación civil y el de notificaciones). No tendría sentido e *introduciría mucho overhead* tener que pegarle a estos endpoints por cada reserva para saber por qué validaciones tiene que pasar. Ya que estas validaciones cambian muy poco, y el trade off que tomamos es performance por modificabilidad: para modificar los filtros hay que reiniciar el *Processor*, pero cada request al *Processor* performara muchísimo mejor.

Como se mencionó al comienzo de la documentación otra decisión tomada fue que el *Processor* valida las reservas y determina si las puede agendar, las encola, delegando la persistencia de ellas al *Emitter Service*, ya sea como pendientes o concretadas. Esta decisión fue tomada con el objetivo de **favorecer la performance** ya que estamos sacando del proceso principal un acceso a la base de datos que se hará muy seguido por cada petición de reserva válida. El trade off que tiene esto es que no podemos estar 100% seguros de que la reserva se guardó correctamente en la base de datos porque no estamos esperando una respuesta. La **táctica de disponibilidad retry** puede aplicarse para contrarrestar esto.

Consideramos los pequeños detalles. Por ejemplo, **hacer los algoritmos lo más eficientes posible**. Al buscar vacunatorios, preguntar primero a mongo sin considerar el criterio de la hora. Si no trae vacunatorios es porque no hay ninguno disponible, y podemos mandar la reserva a pendientes. Mientras que si primero preguntamos incluyendo el criterio del horario, en caso de no traer vacunatorios no sabríamos si no trajo porque no hay ninguno que coincida con los criterios, o porque no hay en el horario. Y sabiendo que tenemos que hacer el mejor esfuerzo de agendar en el horario que desea y si no se puede, en cualquier otro horario del día, eso implicaría la necesidad de un segundo acceso a la base que podría ser evitado en caso que al final no hubiera vacunatorios para la reserva. Otro pequeño detalle que pensamos fue usar un *FindOne* en vez de un *FindAll* al buscar los vacunatorios para que la respuesta sea lo más rápida posible (aunque esto pueda implicar más accesos a la base de datos, los tiempos de respuesta serán mucho mejores).

El motivo por el cual quisimos delegar la cancelación y consultas de las reservas al *Emitter Service* fue para no ocupar ningún nodo de pm2 en otra cosa que no sea atender las peticiones de reservas y además siendo coherentes con las

responsabilidades esto tiene sentido. Además, estas dos requests no tienen ninguna restricción de performance, por lo que pueden atenderse de otra manera.

Otra **táctica** que se podría usar para mejorar la performance es **incrementar los recursos** aumentando la capacidad de procesamiento. Claramente, nuestra arquitectura performaría mejor con más recursos, ya que nuestra limitante actual es la capacidad de nuestra máquina y no el diseño de la arquitectura.

Aprovechamos esta sección para hablar sobre los problemas de concurrencia del sistema al ser bombardeado con reservas. Al probar bombardear el sistema con reservas encontramos que a veces la cantidad de reservas que se guardan es diferente (es decir, la cantidad de reservas enviadas desde la aplicación *Reservation Processor* al *Reservation Emitter*). Esto se debe a que existe una validación que chequea que la persona no tenga una reserva ingresada en el sistema. Puede pasar que el sistema procese una reserva ‘A’, luego procese una reserva “B” que sea igual a “A”, y permita continuar el procesamiento de “B” ya que se preguntó si estaba repetida previo a haber insertado ‘A’ en la base de datos. También puede ocurrir el caso contrario en el cual en el momento que el sistema procesa la reserva ‘B’, ‘A’ ya fue insertada y entonces ‘B’ es rechazada (caso correcto).

Esta diferencia implica simples milisegundos, y por lo tanto hay veces que la reserva B es aceptada, y hay veces que B es rechazada. Gajes de usar asincronismo.

La solución que encontramos a este problema fue poner el chequeo de que la reserva ya existe lo más cerca posible del algoritmo que inserta la reserva en la base de datos. Somos conscientes que se podrían llegar a dar inconsistencias, pero fue un tradeoff que decidimos aceptar. **Preferimos favorecer la performance por sobre el 100% de consistencia**. Si hubiéramos querido favorecer la consistencia, basta con aplicar un lock para evaluar si la reserva ya existe y luego insertarlas de a una (lo que genera un cuello de botella).

Continuando con el análisis del *Processor Service*, el primer diagrama nos muestra aspectos de **seguridad**. Por ejemplo, la petición de los filtros de las reservas al *Admin Service*, requiere autenticación. Para que el *Processor Service* tenga el token necesario para realizar dicha petición, hicimos lo siguiente:

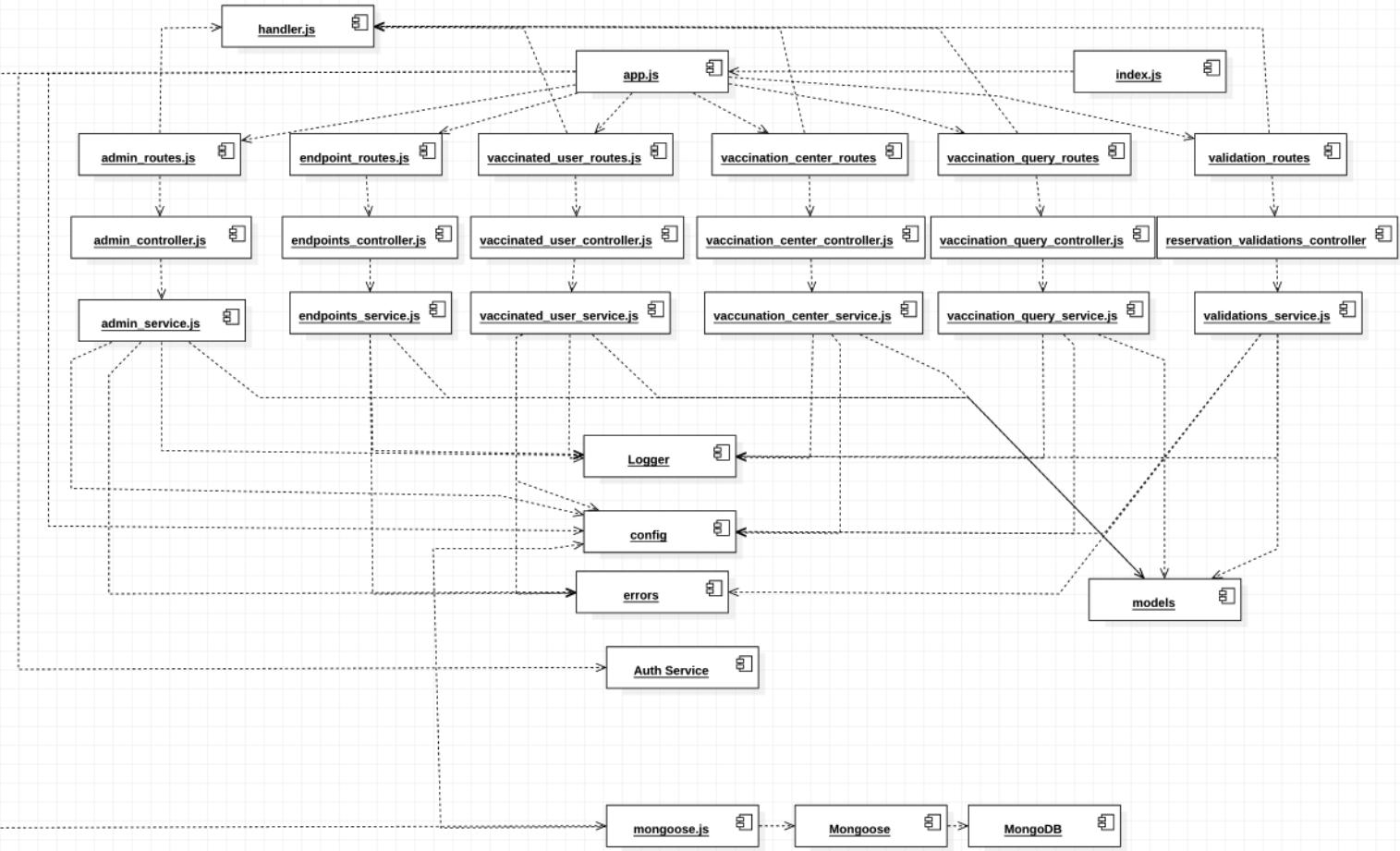
- El *Auth Service* expone un endpoint `.../generate/service`. Se trata de un endpoint interno que genera un token para las aplicaciones (siendo usado en la intercomunicación de las mismas), poniendo como claim “auth\_client\_id” (o rol) el permiso “service”. Es un endpoint interno porque no se pueden generar tokens por ningún otro mecanismo que no sea log in.
- La pegada al *Auth Service* para obtener el token es otra de las tantas tareas que realiza solo una vez al inicializar el servicio *Processor*.

Para poder acceder a las validaciones, transformaciones, el token, las bases de datos, las colas... utilizamos el middleware de Express. Las funciones de

middleware son funciones que tienen acceso al [objeto de solicitud](#) (req) y al [objeto de respuesta](#) (res). Por lo tanto, a cada request entrante le seteamos `req.validations`, `req.transformations`, `req.token`. Y así logramos acceder a estos elementos a lo largo de todo el proceso.

## Admin service

### Representación primaria



### Catálogo de elementos

Catálogo de elementos	
Elemento	Responsabilidades
<code>handler.js</code>	<p>Módulo que tiene como responsabilidad definir diferentes handlers a ser utilizados en las rutas expuestas en <i>routes</i>.</p> <p>Concretamente, expone un <code>authenticationHandler()</code>, responsable de autenticar al usuario que realiza una request contra el auth-service (una de nuestras aplicaciones que estará dentro de nuestra red privada).</p> <p>Por materia de seguridad, decidimos que el único que conozca nuestra clave pública sea también el auth-service.</p> <p>Este componente es muy extensible y modificable. Si se desea agregar</p>

	un nuevo handler que capture la IP del cliente, que obtenga el ID del cliente, el idioma que habla, o cualquier otra información extra, es tan sencillo como implementar una nueva función análoga a <i>authenticationHandler()</i> , pero con otra responsabilidad.
<b>configuration-model.js</b>	Define la estructura a utilizar para persistir una nueva configuración en la base de datos. Este componente podría modificarse si se implementa el sistema para un nuevo país con otros requerimientos (modificar sus validaciones o atributos).
<b>given-vaccines-model.js</b>	Define la estructura a utilizar para persistir una given-vaccine en la base de datos. Al igual que sus validaciones. Este componente podría modificarse si se implementa el sistema para un nuevo país con otros requerimientos (modificar sus validaciones o atributos).
<b>query-model.js</b>	Define la estructura a utilizar para realizar las queries asociadas al Admin Service. Este componente podría modificarse si se implementa el sistema para un nuevo país con otros requerimientos (modificar sus validaciones o atributos).
<b>quota-model.js</b>	Define los campos que una quota debe tener, y los valida. Este componente podría modificarse si se implementa el sistema para un nuevo país con otros requerimientos (modificar sus validaciones o atributos).
<b>user-mode.jsl</b>	Define la estructura a utilizar para persistir un user (admin, vaccinator) en la base de datos. Al igual que sus validaciones. Este componente podría modificarse si se implementa el sistema para un nuevo país con otros requerimientos (modificar sus validaciones o atributos).
<b>vaccination-center-model.js</b>	Define la estructura a utilizar para persistir un centro de vacunación en la base de datos. Al igual que sus validaciones. Este componente podría modificarse si se implementa el sistema para un nuevo país con otros requerimientos (modificar sus validaciones o atributos).
<b>validations-model.js</b>	Define la estructura a utilizar para persistir una validación o transformación en la base de datos. Al igual que sus validaciones.
<b>&lt;specific&gt;-routes.js</b>	Las distintas rutas expuestas por el servicio. En este caso, se exponen rutas para: <ul style="list-style-type: none"> <li>- administradores: registro y sesión</li> <li>- configuración: <ul style="list-style-type: none"> <li>- de endpoints</li> <li>- de validaciones y transformaciones</li> </ul> </li> <li>- usuarios vacunados</li> <li>- centros de vacunación: registro y actualización de quota</li> </ul>

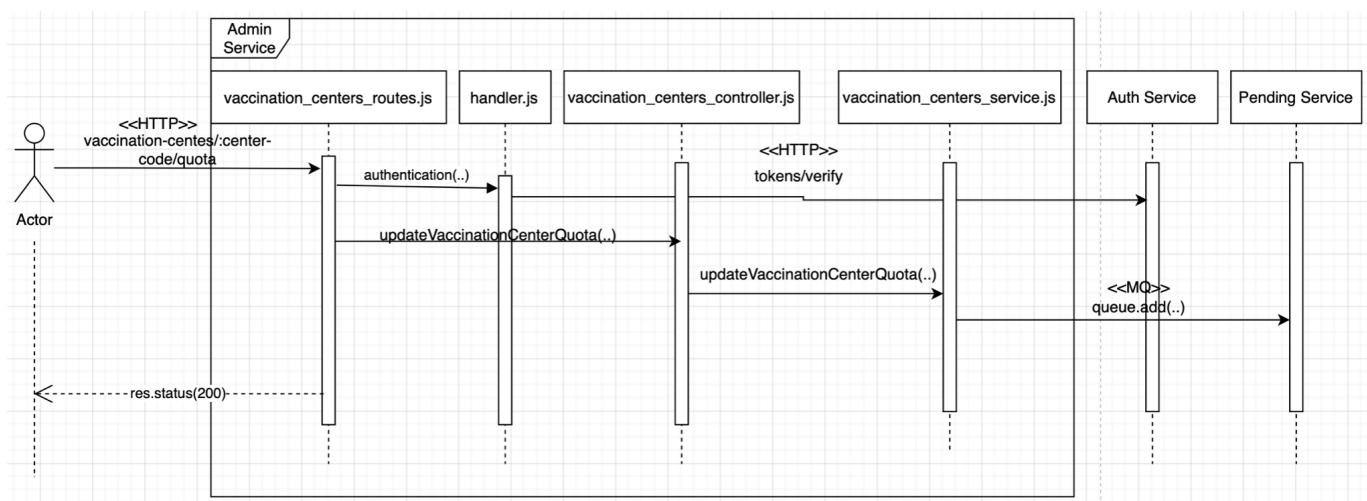
	<ul style="list-style-type: none"> <li>- queries</li> </ul>
<b>admin-service.js</b>	<p>Responsable de manejar la lógica relativa a las requests sobre usuarios y sesión.</p> <p>Relativo a usuarios: debe autorizar al usuario (que tenga permisos para registrar un nuevo usuario), y crear el usuario en la base de datos, según la información recibida.</p> <p>Relativo a sesiones: debe validar las credenciales del usuario, y de ser correctas generar un token mediante el auth-service. Luego, retornarlo. Por otra parte, relativo al log out, y entendiendo que el usuario está autenticado, debe revocar el token mediante el auth-service.</p>
<b>endpoints-service.js</b>	<p>Responsable de agregar, eliminar y retornar los endpoints de configuración de servicios de terceros.</p> <p>Todas estas acciones deben estar autenticadas por un handler, para luego en este componente ser autorizada (según el rol del usuario: <i>superadmin o admin</i>).</p>
<b>vaccinated-user-service.js</b>	<p>Responsable de validar la información recibida, para posteriormente marcar a un ciudadano como vacunado. Este componente cuenta con que se autenticó al usuario, para poder ahora autorizarlo según su rol (superadmin o vacunador).</p> <p>Para editar al usuario, debe acceder a la base de datos.</p>
<b>vaccination-center-service.js</b>	<p>Responsable de crear nuevos vacunatorios, y de gestionar los cupos (quota) de estos. Cuenta con la previa autenticación del usuario, para autorizarlo según su rol (superadmin, admin).</p> <p>Para crear un nuevo vacunatorio, valida la información recibida (apoyándose en <i>vaccination-center-model</i>), y valida que el vacunatorio no esté duplicado.</p> <p>Para crear una nueva quota, valida la información recibida (apoyándose en <i>quota-model</i>), y valida que el vacunatorio exista.</p> <p>Por último, encola la nueva quota en la cola de redis “new_quota” que comunica al <i>Reservation Emitter</i> sobre los nuevos cupos. Esto tiene como objetivo, notificar a todas las reservas pendientes de los nuevos cupos, e intentar agendarlas. Además, al hacerse instantáneo, prioriza a las pendientes sobre las nuevas reservas. A continuación (fuera de la tabla), profundizaremos.</p>
<b>validations-service.js</b>	<p>Responsable de agregar, eliminar y retornar las validaciones y transformaciones seleccionadas en el sistema.</p> <p>Todas estas acciones deben estar autenticadas por un handler, para luego en este componente ser autorizada (según el rol del usuario: <i>superadmin o admin</i>).</p>
Como se observa, no se especificaron todos los componentes de la aplicación. Esto es para no poner información redundante, ya que mucha de esta es análoga con la del resto de aplicaciones (por la estructura que elegimos para implementarlas).	

index, app, db ⇒ Igual responsabilidad y funcionalidad  
errors ⇒ Idem (definir un error genérico y sus implementaciones -puede haber más de 1 file-)  
routes ⇒ Igual responsabilidad y funcionalidad (definir requests, delegar a controllers)  
controllers ⇒ Igual responsabilidad y funcionalidad (catch, delegar a service, response)  
service ⇒ Si bien gran parte es análoga, aquí se encuentran las principales diferencias

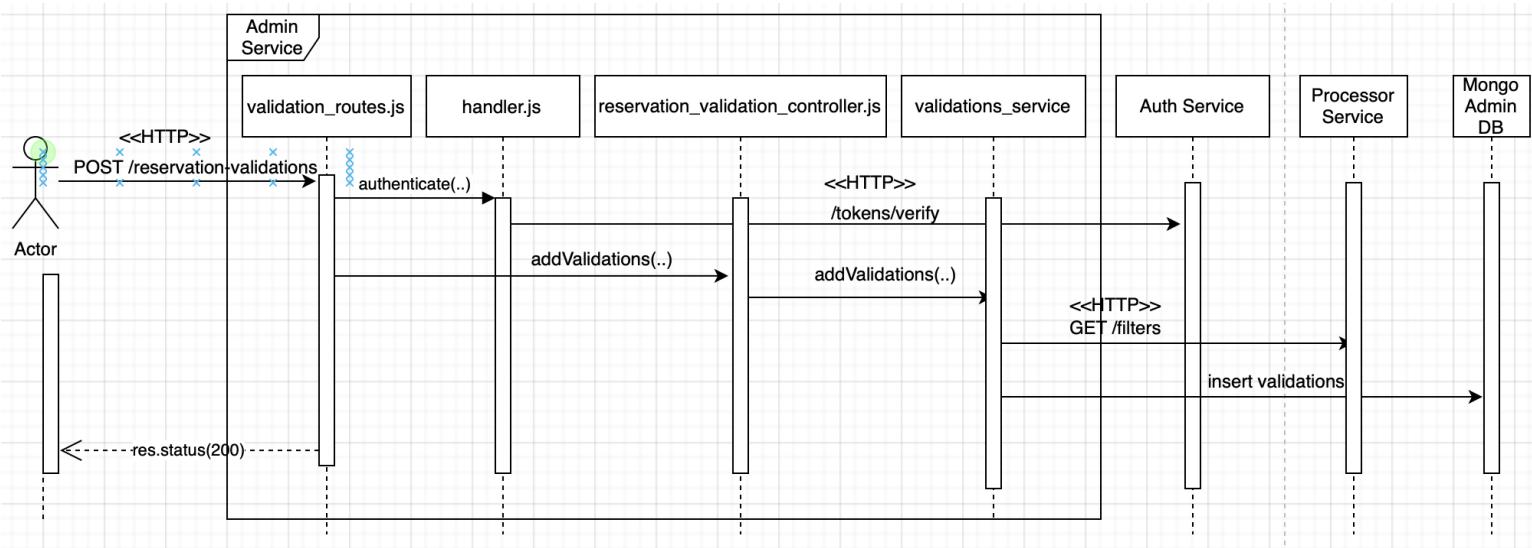
## Comportamiento

Aclaramos que en el diagrama se muestra lo que consideramos interesante del comportamiento, no es algo detallado. Decidimos mostrar una única flecha de respuesta para no hacer el diagrama inentendible.

En el siguiente diagrama se muestra el comportamiento de la funcionalidad de agregar cupos.



En el siguiente diagrama se muestra el comportamiento de la configuración de las validaciones o transformaciones.



### Justificaciones de diseño

Como se puede ver en el diagrama, se implementan varias funcionalidades. Todas estas operaciones requieren autenticación. Para **cumplir con el Req #4 de seguridad**, que establece que algunas funcionalidades u operaciones pueden ser accedidas por usuarios autenticados. Decidimos utilizar la tecnología Json Web Token (se explica mejor en la sección de componentes y conectores del auth-service).

El Admin Service es capaz de resistir ataques gracias a que utilizamos las **tácticas de seguridad autenticar y autorizar autores**, y detectarlos gracias la **táctica verificar la integridad de los mensajes**. Por el hecho de utilizar JWT estamos cumpliendo principalmente 3 de los 4 requisitos de seguridad, nos permite autenticar (en base a su payload), comprobar la integridad y cumplir con el no repudio.

Para el administrador o el vacunador poder realizar las operaciones que requieren autenticación, deben loguearse en el Admin Service para así obtener el token. Cuando el usuario intenta loguearse (se puede ver el diagrama de secuencia en la parte de comportamiento del Auth Service), primero validamos que el usuario exista por email, luego validamos las credenciales, seteamos las claims que queremos que estén presentes en el token en el payload y le delegamos al Auth Service la responsabilidad de generar dicho token. Las claims que decidimos incluir fueron *id del usuario*, el谢piració n del token, y el permiso que tiene dicho usuario (auth\_client\_id o rol). De esta forma cuando un usuario desea realizar una operació n, validamos que el rol / permiso sea uno de los aceptados y listo. Distinguimos los permisos en: *admin, vaccinator, y superadmin*.

- El superadmin tiene los permisos de hacer todo lo que requiera autenticació n, crear vacunatorios, agregar cupos, poner como vacunada a una persona, cambiar las configuraciones de validaciones y endpoints, hasta crear nuevos usuarios con permisos, tanto admins como vaccinators.

- Consideramos al superadmin como los desarrollados del sistema
- El admin puede hacer todo lo anterior menos crear usuarios superadmins y marcar a una persona como vacunada.
  - Consideramos como admin los encargados del plan de vacunación de cada país (ejemplo: técnicos del ministerio de salud pública).
- El vaccinator solo puede agregar que una persona fue vacunada.
  - Consideramos como vaccinators los profesionales de la salud que usen la aplicación para marcar usuarios como vacunados.

Existe un último rol que es el de Service que tienen permisos sobre el admin service y sobre el notification service. Es el rol que se le da a los tokens que serán utilizados por nuestras aplicaciones para comunicarse entre ellas.

Para ofrecer más seguridad al usuario, hasheamos la contraseña del mismo utilizando bcrypt al registrarse, cumpliendo así con el **Req #9 de seguridad** que menciona la importancia de la encriptación de la contraseña así como también la **táctica de seguridad cifrado de datos**.

Otro rol importante que tiene el *Admin Service*, como se puede ver en el diagrama de secuencia de más arriba, es la configuración de validaciones, transformaciones y endpoints de servicios de terceros para cumplir con el **Req #7 de modificabilidad**.

El *Processor* es la aplicación que va utilizar todos o algunos de estos filtros. Para esto se deben configurar mediante API aquellos que se deseen utilizar. Como se ve en el diagrama, el *Admin Service* expone un endpoint que permite que el administrador, envíe los nombres de los filtros a utilizar y se almacenan en la base de datos *Admin*. El *Admin Service* es el encargado de validar que los filtros recibidos existan y que no sean un simple string sin significado alguno. Para validar que exista, se apoya de un endpoint que expone el *Processor Service* que devuelve los nombres de todas las validaciones que hay en el módulo *validator* y las transformaciones que hay en el de *transformations*.

Entonces estamos **cumpliendo con el Req #7** porque si se desean cambiar las configuraciones actuales, lo único que hay que hacer es pegarle al endpoint del *Admin Service* pasándole las configuraciones nuevas y reiniciar el *Processor Service*. Tener que reiniciar la aplicación no es algo ideal pero como estas configuraciones cambian muy poco, nos pareció adecuado implementarlo de esta manera. Otro de los trade-offs que tomamos.

Queremos aclarar que tomamos como válidos aquellos formatos de fechas que acepta el *new Date()* y no las tomas como inválidas. En cuanto a la transformación de la fecha que pedía la letra, no logramos transformarla a dd/mm/yyyy justamente porque esta no era una fecha válida para *new Date()*. De todas maneras, dicha transformación permite que cada país decida con qué formato de fecha desea

trabajar internamente en el sistema, mientras que sea una fecha aceptada por `new Date()`.

Además, no faltó implementar el filtro que valida el dígito verificador simplemente por falta de tiempo. Se implementa siguiendo la misma idea que los otros.

Esta aplicación tiene una característica: no es tan performante como el Processor. Pero esto no es un problema. Este servicio es crítico, y necesita garantizar **seguridad y consistencia** (de la información) antes que performance. Además, quienes utilicen este servicio serán empleados. No es un problema tan grave que un empleado tenga que esperar. Pero si es un problema trancar a un ciudadano. Esto no quiere decir que la aplicación no funcione como se espera, o que demore mucho. Sino que en casos límites, donde las bases de datos estén saturadas, y el admin service esté saturado, esta aplicación demorara más tiempo en procesar las requests.

Por último, haciendo referencia al primer diagrama de secuencia, hablaremos sobre las reservas pendientes, ya que es el *Admin Service* es quien notifica al *Pending Service* sobre nuevos cupos. En la sección de componentes y conectores, hablaremos más a fondo sobre esto. Pero queremos aclarar que priorizamos las pendientes sobre las nuevas reservas, ya que notificamos al *Pending Service*, para que instantáneamente intente agregar las reservas pendientes a los nuevos cupos. Esto no es una garantía de que se priorizará al 100% las pendientes, pero al ser tan baja la probabilidad, decidimos no contemplarlo. Si se quisiera ser mas estricto en esto, se podrían optar por dos caminos:

- Cuando se ingresen nuevos cupos (actividad que se da poco seguido), se bloquea o apaga el *Processor* por unos minutos, mientras que se procesan las pendientes. De esta manera se evita que lleguen nuevas peticiones, priorizando las pendientes enteramente.
- Pero si se desea mantener el *Processor* corriendo todo el tiempo, se puede generar el siguiente mecanismo:
  - Al recibir una nueva reserva (por ejemplo, en forma de una nueva validación) validar que no exista ninguna reserva en pendientes esperando un cupo con los mismos criterios que la nueva reserva.
  - De coincidir los criterios de la nueva reserva con alguna de las de pendientes, entonces agregamos la nueva reserva a pendientes. Y si no, le intentamos asignar un cupo.
  - Este mecanismo podría implementarse utilizando un caché entre el *Pending Service* y el *Processor*, para acelerar el tiempo de la consulta, generando que esta validación dure lo menos posible.

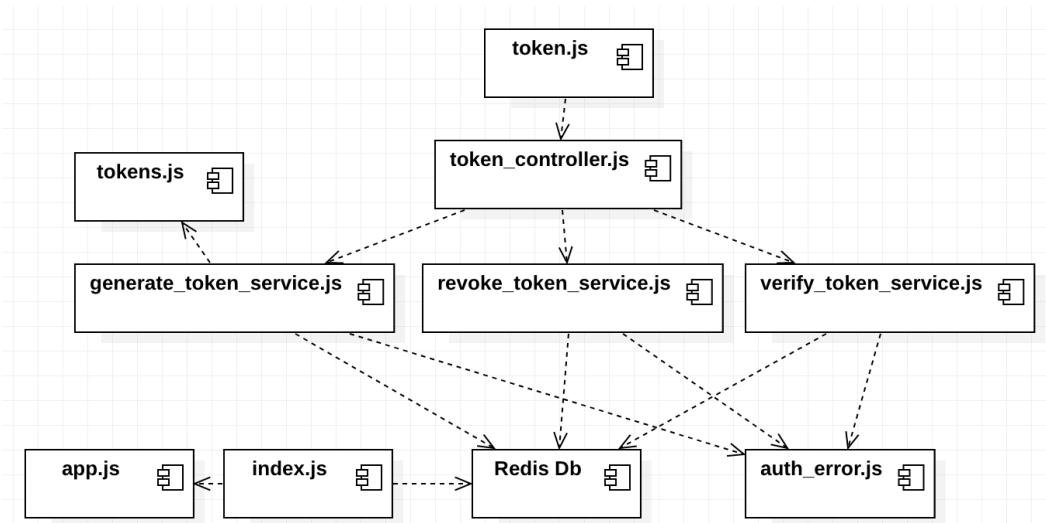
Debemos considerar que ambas alternativas tienen sus desventajas. La primera implica apagar la aplicación de *Processor*. La segunda implica sacrificar la performance. Pero como el *Processor* debe priorizar la performance, y no se exige

estrictamente respetar las pendientes, optamos por no implementar alguna de estas dos estrategias recién mencionadas.

En conclusión, a lo largo del sistema priorizamos el atributo de calidad **seguridad** (con el auth-service, autenticación de endpoints, validación de autorización en servicios, manejo de bcrypt para los contraseñas utilizando clave pública / clave privada, el uso extensivo de logs, y el uso potencial de una red privada que nos permita exponer únicamente ciertos endpoints...). De todas formas, en los archivos de configuración se exponen las credenciales para acceso a bases de datos, mongodb, redis, al igual que otros datos que son sensibles y que deberíamos ocultar. No creemos relevante guardar claves en un vault, o esconderlas del código (por ejemplo usando variables de entorno), ya que no creemos valga la pena para esta instancia. Además, el código será utilizado únicamente por administradores de confianza. Lo mismo sucede con la clave privada y pública del *Auth Service*. El archivo existe en el mismo servicio. Repetimos: la solución a esto sería esconder todo dato sensible en un vault (por ejemplo, HashiCorp Vault).

## Auth Service

### Representación primaria



### Catálogo de elementos

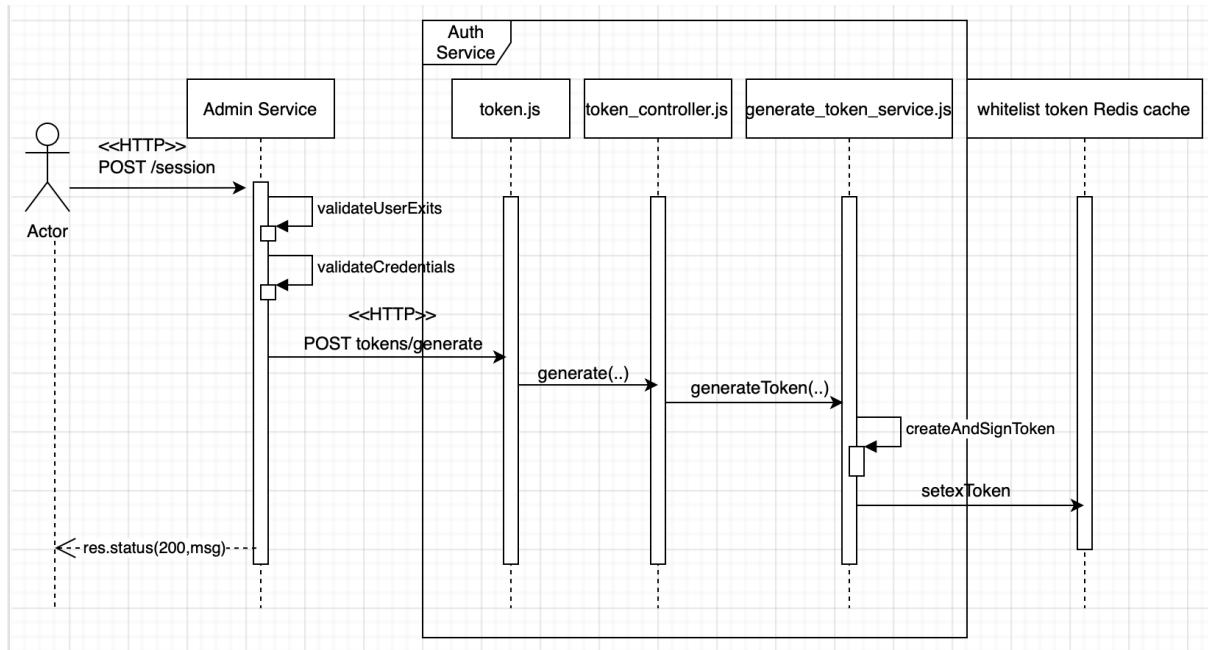
Catálogo de elementos	
Elemento	Responsabilidades
<b>config/arq_private.key</b>	No es un componente, pero creemos relevante mencionar que esta aplicación debe contar con un par de claves privada / pública, necesarias para la firma de los tokens (JWT).
<b>token-model.js</b>	<p>Responsable de definir las estructuras relacionadas con un token y su información, y para cada una de estas, implementar un método <code>toJSON()</code> que las convierta en un JSON.</p> <p>Debe definir 3 estructuras y sus claims:</p> <ul style="list-style-type: none"> <li>- <u>Token Details</u> <ul style="list-style-type: none"> <li>- contiene access token y expiración (para retornar a usuario al log in)</li> </ul> </li> <li>- <u>Token Data</u> <ul style="list-style-type: none"> <li>- contiene user_id, auth_client_id (rol) y expire_date (es el payload del token, no puede tener información sensible)</li> </ul> </li> <li>- <u>Full Token Data</u> <ul style="list-style-type: none"> <li>- contiene user_id, auth_client_id (rol) y expire_date (es la información que se retorna al hacer un verify, y que se asocia en la white-list al access token. Puede contener claims con información sensible)</li> </ul> </li> </ul>
<b>token-routes.js</b>	<p>Responsable de definir las rutas de la aplicación:</p> <ul style="list-style-type: none"> <li>- generate token</li> <li>- generate token for service</li> </ul>

	<ul style="list-style-type: none"> <li>- verify token</li> <li>- revoke token</li> </ul> <p>Recordemos que el auth service estará dentro de nuestra red privada, por lo que los endpoints que exponga no podrán ser accedidos por usuarios, sino que únicamente por el resto de nuestras aplicaciones también pertenecientes a esta red.</p>
<b>generate-token-service.js</b>	<p>Es el componente responsable de (utilizando la librería de JWT):</p> <ul style="list-style-type: none"> <li>- Generar un token dada la información recibida (previamente validada)</li> <li>- Firmarlo con la clave privada dentro del módulo config asignando como payload la TokenData generada</li> <li>- White-listearlo (agregarlo a la base de datos en memoria de Redis) junto con la FullTokenData generada</li> <li>- Retornar al usuario el TokenDetails generado con el access token y su expire_date</li> </ul> <p>Para esto se utilizará el componente explicado previamente (token models).</p>
<b>revoke-token-service.js</b>	Es el responsable de revocar un token. Es decir, eliminarlo de la white-list para que no sea más un token válido.
<b>verify-token-service.js</b>	Es el responsable de validar un token recibido. Esto implica: <ul style="list-style-type: none"> <li>- Validar la integridad del token usando la librería de JWT</li> <li>- Validar que el token este white-listado (esto implica que el token no haya expirado, ya que de haber caducado, se eliminará automáticamente de la base de datos en memoria).</li> <li>- De ser válido el token, retornar al usuario la FullTokenData asociada con el access token en la base de datos en memoria (recordemos que es una clave-valor de strings, por lo que nos apoyamos de stringify y parse para convertir entre JSON y string).</li> </ul>
<b>Redis Db</b>	Es responsable de mantener una white-list (para favorecer la seguridad) de todos los tokens válidos. Esto se logra con esta base de datos en memoria (Redis) que favorece la performance.
<p>Como se observa, no se especificaron todos los componentes de la aplicación. Esto es para no poner información redundante, ya que mucha de esta es análoga con la del resto de aplicaciones (por la estructura que elegimos para implementarlas).</p> <p><u>index</u>, <u>app</u>, <u>db</u> ⇒ Igual responsabilidad y funcionalidad  <u>errors</u> ⇒ Idem (definir un error genérico y sus implementaciones -puede haber más de 1 file-)  <u>routes</u> ⇒ Igual responsabilidad y funcionalidad (definir requests, delegar a controllers)  <u>controllers</u> ⇒ Igual responsabilidad y funcionalidad (catch, delegar a service, response)  <u>service</u> ⇒ Si bien gran parte es análoga, aquí se encuentran las principales diferencias</p>	

## Comportamiento

Aclaramos que en el diagrama se muestra lo que consideramos interesante del comportamiento, no es algo detallado. Decidimos mostrar una única flecha de respuesta para no hacer el diagrama inentendible.

El siguiente diagrama muestra el rol del *Auth Service* a la hora de hacer log in.



## Justificaciones de diseño

Esta aplicación surgió con el objetivo de brindar un manejo de sesiones y de autenticación unificado, basándose en tokens (JWT). Lo hicimos de manera **performante y segura**. Implementamos este servicio según el **patrón Federated Identity**. Otras aplicaciones y servicios, utilizan a esta aplicación con el fin de validar que un usuario sea quien dice ser (**táctica de seguridad autenticar usuarios**).

¿Cómo funciona? Esta aplicación expone 4 endpoints:

- GenerateToken (genera un token para un usuario según las claims recibidas)
- GenerateTokenForService (genera un token para una aplicación o servicio según las claims recibidas)
- VerifyToken (verificar la correctitud de un token)
- RevokeToken (marca cierto token como invalido)

Como se puede ver en el primer diagrama de secuencia, el servicio genera tokens (utilizando una librería de JWT) según información recibida en la request y retorna access tokens que expirarán automáticamente en el tiempo seteado en la configuración del sistema (en este caso en los archivos de configuración). El motivo por el que expiran es meramente **seguridad**. Utilizamos JWT ya que, en combinación con un sistema de cifrado utilizando **claves pública/privada** nos

garantizan seguridad, además de que performa muy bien. Es suficiente con dicho token para manejar sesión, es simple y es descentralizado. Además, nos permite expirar los tokens de manera sencilla (usando un TTL).

Al generar un token, el auth-service agrega dicho token a un caché en Redis, asociado a la información que recibió en la request. Este caché sirve como whitelist: al validar un token, si no pertenece a la whitelist, es invalido. De esta forma agregamos otra barrera de seguridad. Cuando el token expira, o cuando se recibe una orden de revoke sobre el token, se eliminará automáticamente del caché, garantizando que este token no es más válido.

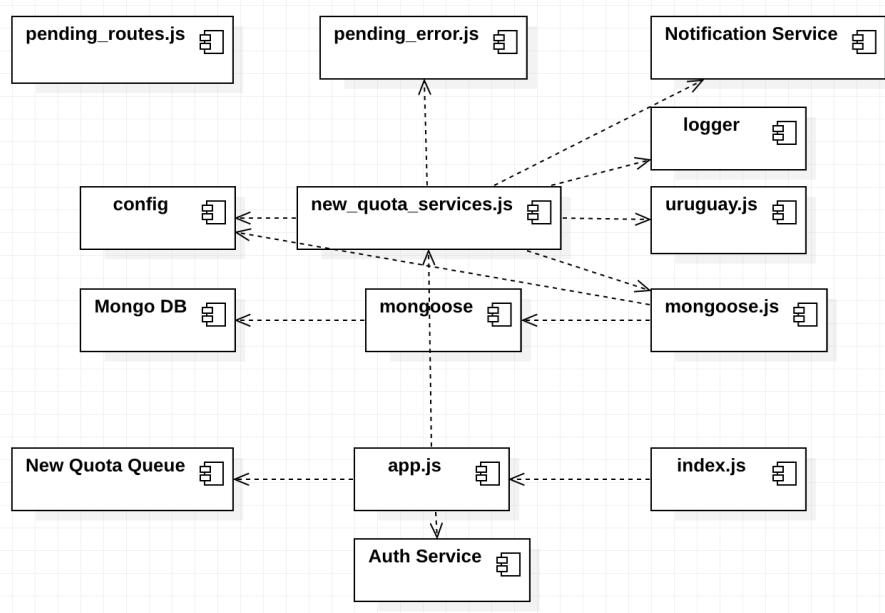
Y... ¿Cómo utilizan el resto de las aplicaciones el auth-service? Esto se puede ver en los diagramas de secuencia en comportamiento del Admin Service. Como dijimos, esta aplicación estará dentro de nuestra red privada: ninguna persona tendrá acceso al auth service directamente. El Admin Service, será la única aplicación que podrá generar tokens para usuarios, ya que es el único punto de nuestro sistema en el que manejamos sesiones. Entonces si alguien necesita un token, lo solicitará mediante *Log In* al Admin Service como se ve en el diagrama de secuencia de arriba, quien se lo proveerá.

Por otra parte, si una aplicación necesita un token para realizar pegas que necesiten autenticación, deberá conocer la dirección de nuestra red privada que conecte con el auth-service (y esto no es un problema, ya que se especifica en los archivos de configuración).

Este servicio performa muy bien por su simpleza, por el manejo de asincronismo y por implementar la whitelist en caché de Redis, el cual es de rapidísimo acceso. Además, y gracias a la estructura de la aplicación (como en el resto de las aplicaciones) es sencilla y extensible: el comportamiento se puede extender sin problemas y con el menor costo.

## Pending Service

### Representación primaria



### Catálogo de elementos

Catálogo de elementos	
Elemento	Responsabilidades
<b>pending-routes.js</b>	Lo mencionamos como curiosidad, y buena práctica. Esta aplicación no expone endpoints útiles, más que un health check. Esto viene a mano si se desea saber si la aplicación está funcionando de manera correcta. Podría no estar.
<b>app.js</b>	<p>Este componente, además de tener las responsabilidades de inicialización que el resto de apps tienen, se encarga de inicializar el listener de la cola de mensajes (de Redis, utilizando la librería Bull) que notifica de la creación de nuevos cupos (quota). Claramente, esta es una tarea que se realiza una única vez.</p> <p>Configuraremos el listener para que permita procesar como máximo 100 mensajes a la vez. Esto delega al servicio, para procesar la new_quota.</p>
<b>new-quota-service.js</b>	<p>Es el componente responsable de procesar una nueva quota (nuevos cupos) y asignar todas las pendientes posibles a estas nuevas quotas, teniendo algunas consideraciones:</p> <ul style="list-style-type: none"> <li>- Las características de la implementación del sistema (para eso nos apoyamos en archivos de configuración y en el componente mencionado a continuación)</li> <li>- Asignar las reservas según los criterios requeridos</li> <li>- Asignar las reservas utilizando alguna prioridad (en el caso de uruguay, edad y grupo prioritario)</li> </ul>

	<p>Para hacer esto, se traen primero todas las reservas que encajen en los criterios de la cuota. Una por una se intentan agendar, hasta que:</p> <ul style="list-style-type: none"> <li>a) no queden mas cupos, se termina la ejecución</li> <li>b) no queden más reservas, por que se agendaron todas</li> </ul>
<b>algorithm/&lt;impl&gt;.js</b>	<p>La implementación de las particularidades relacionadas con la implementación del sistema, y la asignación de nuevos cupos a reservas pendientes.</p> <p>Debe implementar 3 funciones:</p> <ul style="list-style-type: none"> <li>- <code>generateUpdateFields(reservation)</code> <ul style="list-style-type: none"> <li>- Genera el filtro del update que modifica la reserva seteando la información de la nueva quota y modificando el estado.</li> </ul> </li> <li>- <code>generateReservationForDatabase(reservation)</code> modifica la reserva recibida, seteando los campos para que esta quede con la forma adecuada.</li> <li>- <code>getFilterGivenQuota(quota)</code> que retorna el filtro de mongo necesario para seleccionar las reservas que cumplan con la quota</li> <li>- <code>getFilterSorting()</code> que retorna el filtro necesario para retornar la query de las reservas ordenadas según la prioridad deseada (en el caso de uruguay: mayor edad y mayor prioridad)</li> </ul>
<b>New Quota Queue</b>	<p>El objetivo de esta cola es mantener informado al Pending Service de cuando se agregan nuevos cupos y cuáles fueron estos, para que el pueda asignar las reservas pendientes.</p>
<b>Notification Service y Auth Service</b>	<p>Se necesita el Notification Service ya que una vez agendado alguien de pendientes, se manda un sms al usuario. El Auth Service es necesario porque para poner notificar, requiere estar autenticado.</p>
<p>Como se observa, no se especificaron todos los componentes de la aplicación. Esto es para no poner información redundante, ya que mucha de esta es análoga con la del resto de aplicaciones (por la estructura que elegimos para implementarlas).</p> <p> <u>index</u>, <u>app</u>, <u>db</u> ⇒ Igual responsabilidad y funcionalidad  <u>errors</u> ⇒ Idem (definir un error genérico y sus implementaciones -puede haber más de 1 file-)  <u>routes</u> ⇒ Igual responsabilidad y funcionalidad (definir requests, delegar a controllers)  <u>controllers</u> ⇒ Igual responsabilidad y funcionalidad (catch, delegar a service, response)  <u>service</u> ⇒ Si bien gran parte es análoga, aquí se encuentran las principales diferencias     </p>	

## Comportamiento

Funcionalidad de asignación de las personas desde pendientes

## Justificaciones de diseño

La decisión de crear un servicio de pendientes nace con el fin de no saturar al Processor para el cual es muy importante la performance, y por que el estos tienen responsabilidades muy distintas. Entonces creamos el Pending Service. Esto hace que el *Processor* sea más performante, cohesivo y desacoplado. Pero también hace que el *Pending* pueda performar mejor (y escalarse independientemente). El resultado, son aplicaciones más cohesivas, menos acopladas, independientes, más modificables y fácilmente extensibles.

En adición, no queríamos interferir en el procesamiento de reservas del Processor. Mientras se asignan las pendientes el Processor sigue funcionando, aceptando nuevas solicitudes de reserva porque son los procesos totalmente separados. Mezclar estas responsabilidades impactaría en su performance.

El *Pending Service* no necesita explícitamente ser performante, por lo que la asignación de cupos de las pendientes no es una tarea que demore pocos segundos (en casos borde donde se cuente con muchísimas pendientes y muchísimos cupos, esta tarea podría llegar a demorar minutos). Esto no es un problema, ya que nadie está esperando por las tareas de este servicio. De todas maneras, se implementó lo más performante y consistente posible, buscando además priorizar las pendientes sobre las nuevas reservas (como ya mencionamos en la justificación de componentes y conectores del Admin Service, cuando hablamos sobre la cola de mensajes que notifica al Pending sobre nuevos cupos).

Si se desea ver de qué manera se priorizan las pendientes sobre las nuevas, o si se desea ver cómo podría modificarse esta implementación, referir a la sección de *Justificación de Diseño de Componentes y Conectores* para el *Admin Service* (último párrafo).

Por último, queremos decir que este servicio es fácilmente escalable horizontal y verticalmente. Logrando esto mediante múltiples copias de cómputo, y una computadora más potente respectivamente.

¿Cómo priorizar a las personas mayores o prioritarias primero? Simple. Si repasamos el código, veremos que traemos de mongo toda reserva pendiente que cumpla los criterios de la quota. Tenemos dos opciones: traer los registros desordenados (sin priorizarlos) o ordenados según el criterio deseado (priorizado según se defina en la implementación dentro de *algorithms/<implementation>.js*). Es la segunda opción por la cual nos volcaremos. Concretamente para Uruguay, traeremos las reservas ordenadas por edad y siguiente por prioridad. Si se desea cambiar esto, es tan sencillo como modificar la función *getFilterSorting()* del archivo específico de implementación. Actualmente nuestro Pending Service asigna por los criterios de estado, zona, edad o zona, estado y prioridad. Nos faltó concretamente

implementar la parte de que sea el más viejo y el más prioritario el que se saque primero, pero como mencionamos recién, esto no sería difícil y tendría solo impacto en esta clase del algoritmo. Lo que sí tuvimos en consideración fue que si la fecha en la que se quería vacunar la persona ya pasó, la asigne en cualquier otro vacunatorio que cumpla con el resto de los criterios. De no considerar esto, esa reserva pendiente nunca se asignaría.

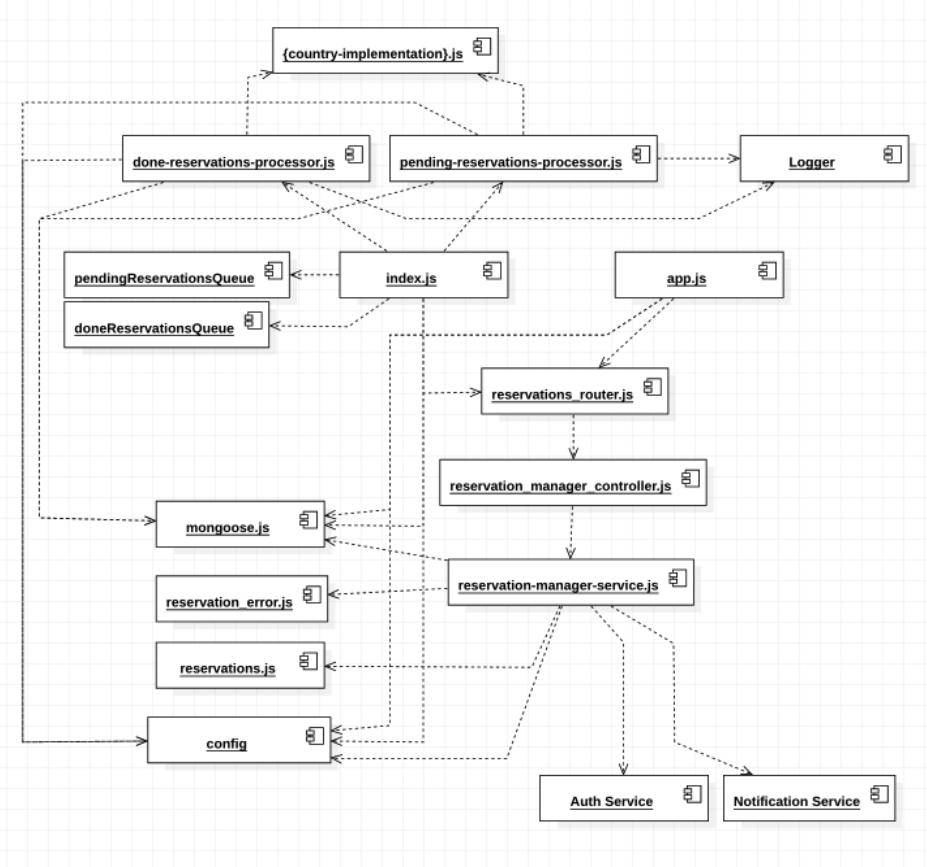
La aplicación favorece la **modificabilidad**. Si se desea modificar la manera en la que se seleccionan las reservas según la quota (es decir, si se desea modificar el algoritmo de asignación de pendientes), es tan sencillo como implementar en *algorithms/<implementation>.js* la función `getFilterGivenQuota(quota)` que retorna el filtro de mongo necesario para esto. Al ser mongo un lenguaje tan potente, creemos que siempre será suficiente con el filtro de la query para que se cumplan con los requerimientos que plantean los clientes.

Además de esta función, en el archivo dentro de *algorithms* se deben implementar 2 funciones más:

- `generateUpdateFields(reservation)`
  - Genera el filtro del update que modifica la reserva seteando la información de la nueva quota y modificando el estado.
- `generateReservationForDatabase(reservation)` modifica la reserva recibida, seteando los campos para que esta quede con la forma adecuada.

## Reservation Emitter

### Representación primaria



### Catálogo de elementos

Catálogo de elementos	
Elemento	Responsabilidades
reservations-routes.js	Responsable de definir las rutas para cancelar una reserva y consultar el estado de la reserva.
app.js	<p>Este componente, además de tener las responsabilidades de inicialización que el resto de apps tienen, se encarga de inicializar el listener de la cola de mensajes (de Redis, utilizando la librería Bull) que notifica de la recepción de nuevas reservas (exitosas o pendientes) por parte del Processor, con el fin de persistirlas. Claramente, esta es una tarea que se realiza una única vez.</p> <p>Configuramos el listener para que permita procesar como máximo 100 mensajes a la vez. Esto delega al servicio, para procesar la <i>reservation</i> recibida.</p>
reservation.js	Responsable de definir la estructura y validaciones de una reserva, antes de persistirla en la base de datos.

<code>algorithms/&lt;impl&gt;.js</code>	Define qué transformaciones hacer a la reserva antes de persistir en la base. Dicho archivo debe implementar dos funciones: - pendingTransformations(reservation) - doneTransformations(reservation) Ambos transforman la reserva, transformando su <code>reservation.date</code> a formato Date.
<code>done-reservations-service.js</code>	Responsable de procesar una reserva <i>exitosa</i> ya validada (enviada y validada por el Processor) para posteriormente insertarla en la base de datos de mongo de reservas.
<code>pending-reservations-service.js</code>	Responsable de procesar una reserva <i>pendiente</i> ya validada (enviada y validada por el Processor) para posteriormente insertarla en la base de datos de mongo de reservas.
<code>reservations-manager-service.js</code>	Responsable de conocer la lógica necesaria para cancelar una reserva, modificando la base de datos de reserva y notificando al usuario de dicha acción.  Además, es responsable de conocer la lógica necesaria para retornar la información relativa a una reserva, para cuando se solicite conocer el estado de una reserva.
<b>Auth Service y Notification Service</b>	Se necesita el Notification Service ya que una vez cancelada la reserva, se manda un sms al usuario. El Auth Service es necesario porque para poner notificar, requiere estar autenticado.
<code>pendingReservationQueue</code> y <code>doneReservationQueue</code>	Las reservas que se consumen de estas colas son las que se persisten en la base.
Como se observa, no se especificaron todos los componentes de la aplicación. Esto es para no poner información redundante, ya que mucha de esta es análoga con la del resto de aplicaciones (por la estructura que elegimos para implementarlas).	
<u>index, app, db</u> ⇒ Igual responsabilidad y funcionalidad <u>errors</u> ⇒ Idem (definir un error genérico y sus implementaciones -puede haber más de 1 file-) <u>routes</u> ⇒ Igual responsabilidad y funcionalidad (definir requests, delegar a controllers) <u>controllers</u> ⇒ Igual responsabilidad y funcionalidad (catch, delegar a service, response) <u>service</u> ⇒ Si bien gran parte es análoga, aquí se encuentran las principales diferencias	

## Justificaciones de diseño

Esta aplicación, surge para liberar al *Processor* de la responsabilidad de persistencia y mantenimiento de reservas, por 2 motivos:

- **Performance:** sumarle estas acciones a dicha aplicación afectará negativamente la performance, ya que se vería sometido a otro acceso a la base.

- **Responsabilidades:** el *Processor* ya cuenta con suficientes responsabilidades como para asignarle esta, que si bien tiene que ver con las tareas del *Processor*, son responsabilidades diferentes, que conviene estén separadas.

A diferencia del *Notification Service*, que debería performar medianamente bien, para notificar al usuario de lo ocurrido dentro de la ventana de tiempo permitida, este servicio no tiene un requerimiento estricto de performance asociado. Esto se debe a que el *Emitter* puede persistir las reservas en base de datos sin apuro, ya que no hay ninguna acción asociada a esto. Como ya se mencionó antes, para estar preparado para las fallas y repararlas, el *Emitter Service* podría implementar una **táctica de disponibilidad “Recover from faults”** como un **mecanismo de retry**, para que no se pierda la reserva al primer error al intentar persistir.

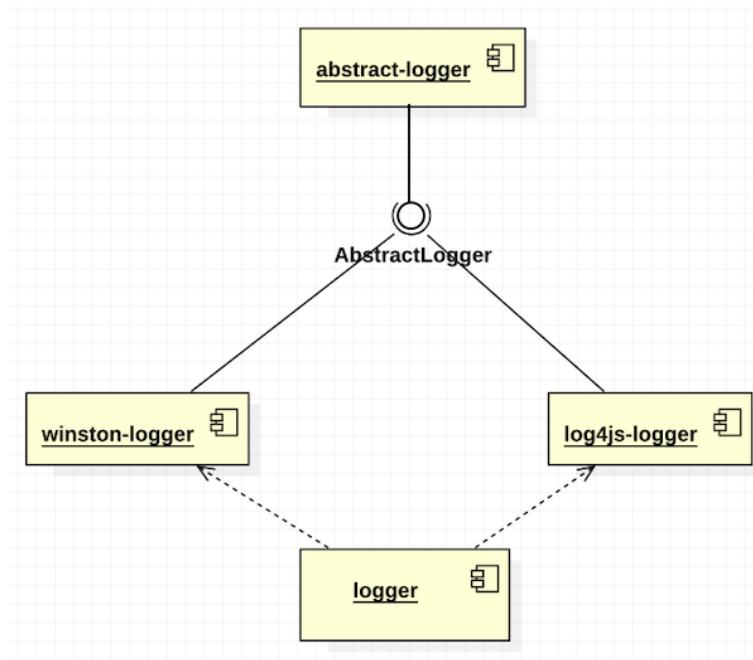
De todas formas, el endpoint de cancelar reserva y de traer reserva, conviene performen bien para no tener al usuario esperando del otro lado (aunque esto no es crítico).

Este servicio depende del funcionamiento correcto del *Notification Service*, ya que necesita notificar cuando una reserva ha sido cancelada exitosamente. Por eso, en los archivos de configuración se configura el endpoint relativo al *Notification Service* mediante el cual se notifica al usuario.

Una vez más, dada la prolijidad de la estructura de la aplicación, y la prolijidad del código, esta aplicación es fácilmente modificable y extensible.

## Logger

### Representación primaria



### Catálogo de elementos

Catálogo de elementos	
Elemento	Responsabilidades
logger	Es el componente que a través del archivo de configuración .JSON (utilizado como parte del Differ Binding), utiliza la implementación y configuración especificadas por los servicios que lo utilicen, y se encarga de registrar eventos y errores, según las distintas llamadas que reciba de los otros componentes que dependen de él. Este es el que los servicios interesados en utilizar el logger, deben importar en su archivo.
log4js-logger	Es el componente que contiene una implementación para logging, utilizando el módulo log4js.
winston-logger	Es el componente que contiene una implementación para logging, utilizando el módulo winston.
abstract logger	Es el componente que declara las operaciones que una implementación debe contener para poder ser utilizada. Define las operaciones comunes a los sistemas de logging y es extendido por las implementaciones dentro de las cuales se podrá optar al utilizar el logger.

## Justificaciones de diseño

Para facilitar el cumplimiento del Requerimiento No Funcional 3 (Seguridad / Disponibilidad) en todos los servicios desarrollados que requirieron loguear información, decidimos desarrollar un módulo independiente que fuera simple y permitiera realizar todas las tareas de las que se encarga un sistema de logging común.

El requerimiento de calidad Interoperabilidad, exige que la solución para la gestión de fallas y errores, cuente con la posibilidad de ser reutilizada en otras aplicaciones, con el menor impacto posible en el código de las mismas. Es por esta razón, que decidimos realizar la implementación de la solución de forma independiente, como otro módulo, el cual ya estuviera listo para usarse en segundos, simplemente teniendo que importar (copiar y pegar) la carpeta del módulo al nuevo proyecto, y agregando las opciones de configuración deseadas al archivo de configuración de la aplicación.

Esto permite hacer un uso muy rápido, agregando capacidad de registro de eventos, teniendo que únicamente modificar el código del proyecto, en los lugares donde se quiere ejecutar operaciones de logging, y únicamente teniendo que realizar el importe del componente logger al archivo de interés.

Con respecto al requerimiento de calidad de Modificabilidad, optamos por realizar una solución que contara con una AbstractLogger, que obligara a las implementaciones a contar con estas operaciones, para después poder intercambiarlas a través de Differ Binding, sin necesidad de tocar el código donde ya se habían configurado las operaciones a realizar con anterioridad.

Un usuario que integra esta solución de logging a su proyecto, cuando con 2 implementaciones muy completas ya listas para usarse desde el comienzo, pero en caso de querer agregar más, únicamente tiene que agregar un archivo a la carpeta “implementations”, en el mismo extender abstract-logger, y desarrollar las siguientes operaciones:

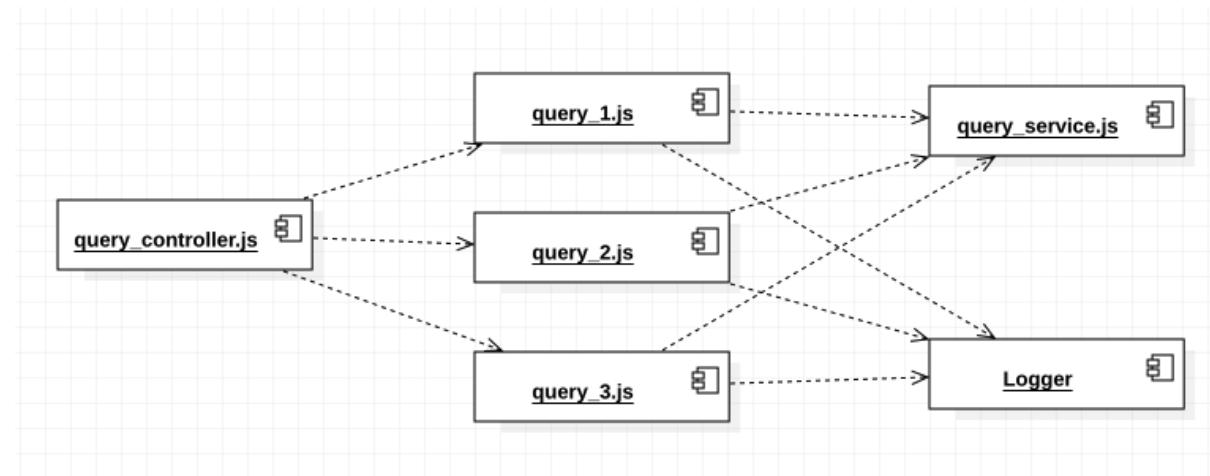
- error (message, label)
- warn (message, label)
- info (message, label)
- verbose (message, label)
- debug (message, label)
- silly (message, label)

Luego de haber realizado esto, basta con modificar un archivo de configuración para poder usarla.

Como explicamos en la vista de módulos, si bien la librería quedó muy prolífica y extensible, no implementamos logging exhaustivamente a lo largo del código ya que no creímos que aporte para esta instancia. De todas formas dejamos comentarios en los lugares que consideramos un log pertinente.

## VacQueryTools

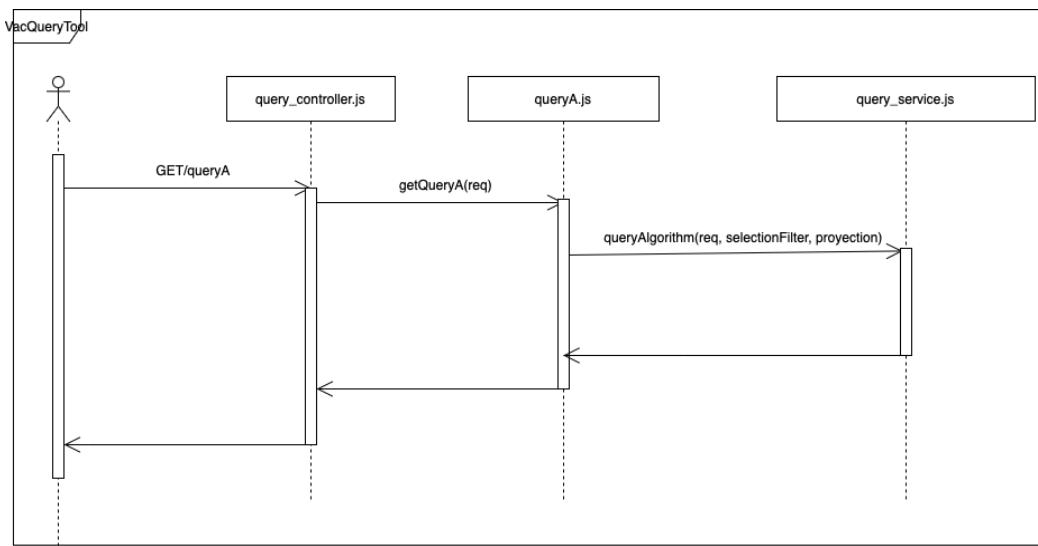
### Representación Primaria



### Catálogo de elementos

Catálogo de elementos	
Elemento	Responsabilidades
query_controller.js	Es el componente que se encarga de redirigir los requests recibidos al servicio correspondiente. Además, atrapa los errores de los servicios para poder trabajar más libremente en estos.
query_1.js	Es el componente que se encarga de generar los filtros para la primer query del VacQueryTools y luego llamar a query_service.js para obtener la respuesta.
query_2.js	Es el componente que se encarga de generar los filtros para la segunda query del VacQueryTools y luego llamar a query_service.js para obtener la respuesta.
query_3.js	Es el componente que se encarga de generar los filtros para la tercera query del VacQueryTools y luego llamar a query_service.js para obtener la respuesta.
query_service.js	Es el componente que provee el algoritmo de búsqueda y agrupamiento de queries. Es utilizado por cada query particular, y como ya se mencionó, provee un algoritmo que recibe el filtro y la proyección y se encarga de ir a buscar a la base de datos las reservas que cumplan con ese filtro y traer únicamente los campos indicados en la proyección. Este archivo ya le devuelve la respuesta armada a cada query particular.

## Comportamiento



Nota: en el ejemplo se representa una query cualquiera llamada “A”. En el obligatorio las queries fueron llamadas `query1`, `query2` y `query3` porque consideramos que los nombres autodescriptivos eran muy largos y confundían más de lo que aportaban.

## Justificaciones de diseño

En el anterior diagrama exemplificamos una de las posibles queries a efectuar en el VacQueryTools hacia la base de datos de reservations. Si el día de mañana VacQueryTools necesitará utilizar otra base de datos, la incorporación de una nueva base de datos sería tan fácil como lo que se hizo en otros servicios que utilizan múltiples db como ser el servicio AdminService.

La independización de este servicio fomenta el no acoplamiento de responsabilidades en un único componente. Mapeando a la vida real, este sería el servicio que se consulta cuando los ciudadanos entran a coronavirus UY y quieren ver las estadísticas diarias. Tiene sentido que sea un módulo aparte ya que con un poco de analytics podríamos analizar las horas en las que más se consulta este servicio (probablemente de noche) y aumentar los recursos o replicar el servicio.

Con el objetivo de aumentar la **Performance**, se incorporó el uso de índices compuestos por las queries de búsqueda en las colecciones correspondientes. Dichos índices son, por lo tanto, compuestos por:

- `was_vaccinated` y `vaccination_date` (Query 1).
- `was_vaccinated`, `age`, y `vaccination_date` (Query 2).
- `status` (Query 3).

Otro de los elementos que se tuvo en cuenta fue que en el armado de las consultas a la base, se tuvo especial precaución en el orden en que se indican las sentencias

de las mismas, lo cual también tuvo un impacto sumamente positivo a nivel de la latencia de las consultas. Independientemente de todas estas optimizaciones consideradas, la latencia obtenida va a estar supeditada parcialmente a los recursos del equipo en el que se encuentre siendo ejecutado.

Name and Definition	Type	Size	Usage	Properties	Drop
Query 1 was_vaccinated, vaccination_date	REGULAR	20.5 KB	0 since Wed Jun 23 2021	COMPOUND	
Query 2 vaccination_date, age, was_vaccinated	REGULAR	20.5 KB	0 since Wed Jun 23 2021	COMPOUND	
Query 3 status	REGULAR	20.5 KB	0 since Wed Jun 23 2021		
_id	REGULAR	36.9 KB	57 since Tue Jun 22 2021	UNIQUE	

Para poder cumplir con el requerimiento de **Modificabilidad**, se intentó hacer el algoritmo de consulta lo más práctico y extensible posible. Para esto, dentro del paquete de servicios hicimos un algoritmo genérico y optimizado que puede ser utilizado por cada query. De esta manera, si el dia de mañana aparece una nueva query, realizarla es tan fácil como hacer un nuevo archivo .js que provea el **filtro** y la **proyección**, llamando al algoritmo genérico con dichos parámetros. De esta manera estamos cumpliendo con la **modificabilidad nuestro requerimiento 12** que plantea que a modo de ejemplo se dan algunas consultas en la letra, pero podrían realizarse otras.

Por ejemplo, la Query 1 es tan fácil de realizar como se detalla a continuación:

\*El logging del servicio fue dejado fuera del ejemplo para entender mejor.



```
const queries = require("./query_service");

const Query1 = async (req) => {
    fromDate = req.query.from_date;
    toDate = req.query.to_date;

    const selectionFilters = {
        vaccination_date : {
            $gte: new Date(fromDate),
            $lte: new Date(toDate),
        },
        was_vaccinated: true,
    };

    const projection = "state vaccination_date _id";
    return await queries.queryAlgorithm(req, selectionFilters, projection);
};

module.exports = {
    getQuery1,
};
```

Vemos que lo que se debe hacer es llamar a la función *queryAlgorithm* con los filtros de selección y la proyección que queremos. A la hora de agrupar, el *queryAlgorithm* solo trae de la base de datos la proyección (campos por los cuales se agrupara). Esto aumenta claramente la performance del sistema ya que estamos trayendo solo los datos que utilizaremos de la base.

El algoritmo de *queryAlgorithm*, por su parte, está optimizado para trabajar con objetos en vez de arrays y funcionar de manera más óptima que se nos ocurrió. Es un algoritmo que llevo trabajo y pienso y del cual estamos conformes con el resultado.

Además, la base de datos de reserva es accedida por el Emitter Service cuando consume de la cola, por lo que consideramos que el acceso a la base de reservas está más controlado, lo que amortigua un poco la saturación de la base.

Consideramos que los índices y el algoritmo diseñado y lo recién mencionado alcanzan para cumplir parcialmente el **requerimiento 2** que menciona que las consultas de VacQueryTool deben tener el menor jitter posible en momentos de carga. En promedio las consultas complejas deben tener una latencia menor a 2 segundos.

Para mejorar aún más la performance se podrían haber incluido otras estrategias, que en algún momento consideramos, como puede ser realizar una replica set de solo lectura en mongo y que el VacQueryTools solamente interactuara con esa base de datos. Otra alternativa era hacer modelos de bases de datos que hicieran más eficiente la consulta, por ejemplo ya tener agrupados por los campos que piden y como todas las queries eran de contar, agregar +1 donde correspondiera. Esta idea

fue descartada porque al tener que ser extensible el VacQueryTool, tener un modelo para cada request no lo haría extensible.

### **Diferencia con las consultas por el plan de Vacunación**

La principal diferencia entre el VacQueryTool y las consultas del plan de vacunación es que el VacQuery puede acceder cualquiera mientras que a las consultas por el plan de vacunación se debe estar autenticado. Además, en el VacQueryTools se debe tener muy en cuenta la performance, mientras que en las consultas por el plan de vacunación no.

Las consultas por el plan de vacunación tienen un algoritmo muy parecido al del VacQueryTools, pero como no la performance no es lo más importante también los ordenamos por estado a la hora de dar la respuesta al administrador que quiere saber la cantidad de vacunas pendientes de asignar por estado y zona.

# Otros aspectos de la Documentación

En esta sección mencionaremos otros aspectos que no precisamente eran el foco del obligatorio pero que también consideramos coherente mencionar ya que aportan al resultado final.

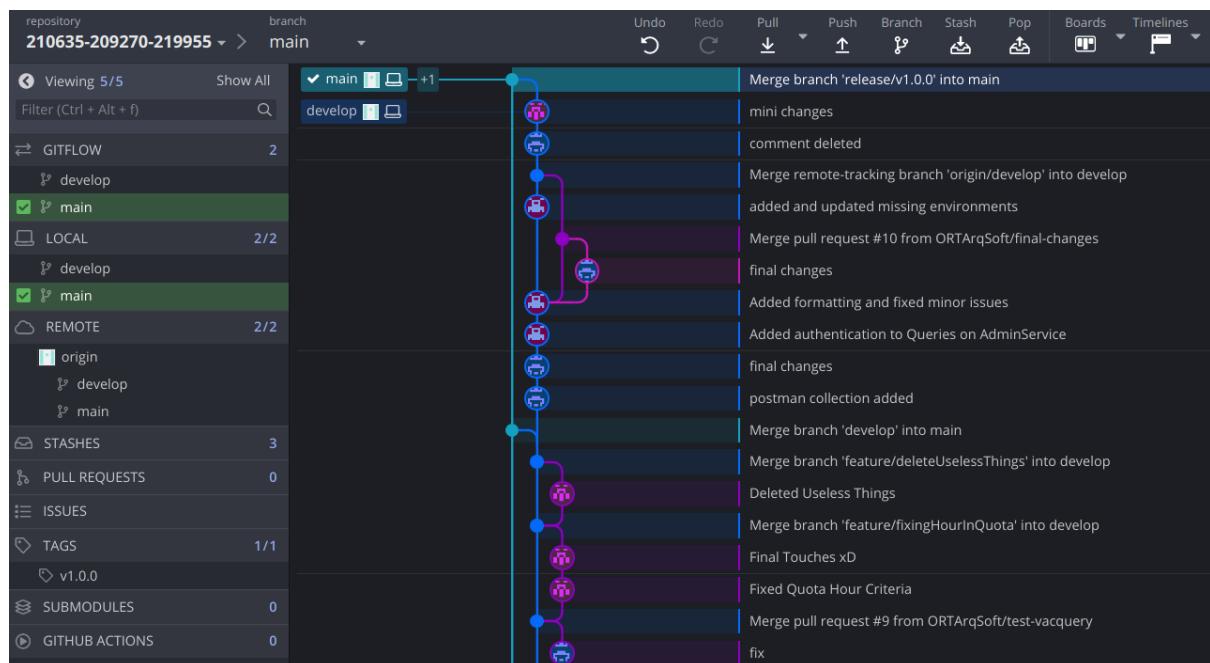
## Control de Versiones

Para mantener organizados los distintos avances y cambios que surgieron a lo largo del proyecto, trabajamos con control de versiones.

Para la documentación, se utilizó Google Docs, mientras que para la codificación, se utilizó un repositorio en Github al que tienen acceso los tres miembros del equipo y los docentes. El link al repositorio fue facilitado en la carátula del obligatorio.

Se trabajó en la totalidad utilizando Git y la metodología Git Flow, teniendo en cuenta tanto la estructura de features como la de releases. En la rama master se podrán ver las releases, y en la rama develop se irá registrando el desarrollo del sistema. Como manejador de Git (y únicamente de manera anecdótica) utilizamos Git Kraken, el cual recomendamos.

Intentamos mantener estándares y prolijidad a todo nivel de trabajo, desde el código hasta el repositorio. En el mismo, por ejemplo, nos aseguramos que no hubieran ramas abiertas que no se estén utilizando, o que las releases cumplan con el estándar de la industria.



## Calidad de Código

Con respecto a la calidad de código, intentamos cumplir con las convenciones de Node JS.

En primer lugar, no involucramos lógica de negocio en los controladores, respetando las responsabilidades de cada módulo. De la mano de esto, se encuentra que solo se configura y retorna la respuesta de la request de express (“res” en nuestro código) dentro de los controllers, respetando una vez más las responsabilidades.

Una posible mejora, es implementar un paquete de *repositories*. No lo implementamos porque los accesos a base de datos son pocos y en contados lugares, por lo que preferimos ahorrarnos la complicación de implementar esa estructura. Aunque creemos que es una mejora necesaria para el sistema.

Respecto a estándares de NodeJS, podemos decir que todo nuestro código está formateado de acuerdo con estos estándares. Intentamos no dejar *imports* innecesarios. Respetamos la estructura y flujo de ruta ⇒ controlador ⇒ servicio. Seguimos los estándares de manejo de error de Node JS (cacheamos todas las promesas, usamos el error de java y heredamos de este...). Sustituimos los callbacks por promesas, lo que genera un código mucho más prolífico, sencillo de leer y entender, y modificable. Nombramos las funciones siempre que pudimos, lo que generó que la detección de errores y el tracing fueran más sencillos (aunque sabemos que deberíamos haber hecho un mejor trabajo en esto). Usamos arrow functions. Hicimos correcto uso de logging. Utilizamos PM2 para repartirnos en distintos cores del CPU. Manejamos distintos entornos de ejecución. Tomamos medidas relativas a seguridad (menos sobre secretos que están en el código, que no consideramos necesario ocultarlo para esta instancia): whitelist de tokens, validación de jsons, uso de JWT, uso de bcrypt para el hash de contraseñas, uso de claves pública/privadas. En resumen, creemos cumplir con todas las buenas prácticas de Node JS competentes de la siguiente [referencia](#).

Paralelo a esto, hicimos el mayor énfasis en Clean Code posible, lo que facilitó la codificación, reduce y simplifica el re-trabajo, y ayudó a generar una solución más completa y que aporte más valor.

# Manual de Instalación

Pese a que el equipo comenzó trabajando con instancias locales de MongoDB, para poder trabajar en conjunto, decidimos crear un cluster en cloud y utilizar ese entre todos. Los archivos de configuración actuales que se encuentran en cada servicio proporcionan acceso a este cluster.

Por lo tanto, para documentar el manual de instalación proveeremos dos posibilidades: o utilizar nuestra instancia de mongo que tenemos corriendo (opción más fácil y recomendada a pesar que agrega latencia por no estar local), o crear una nueva base de datos local (en localhost) y correr el VacPlanner en una base local (opción más performante, no recomendada).

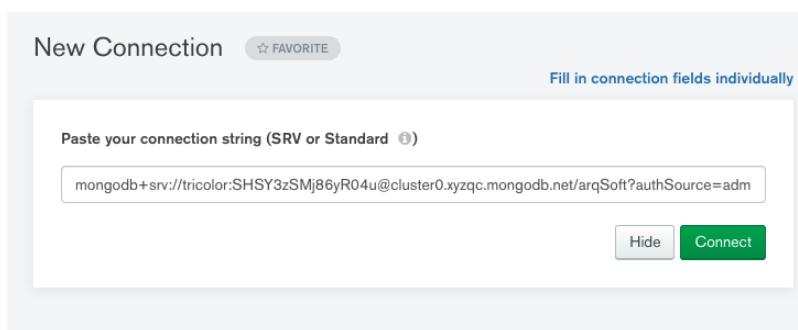
## Aclaración sobre Ejecución de Aplicaciones

Cuando se corra cualquiera de las aplicaciones, se deberá pasar como primer variable / argumento el entorno en el cual se está corriendo, siendo estos “live”, “dev” y “stg”. Esto es para indicar de qué archivo de configuración queremos consumir. Si no se provee ningún argumento, se utilizará por defecto “dev”.

Ejemplo: `node index.js dev`

## Link para conectar a Mongo desde Compass

<mongodb+srv://tricolor:SHSY3zSMj86yR04u@cluster0.xyzqc.mongodb.net/arqSoft?authSource=admin&replicaSet=atlas-mvx07p-shard-0&w=majority&readPreference=primary&appname=MongoDB%20Compass&retryWrites=true&ssl=true>



## Manual de instalación para correr en nuestro cluster en Cloud

Para correr el servicio con el cluster que estamos utilizando nosotros es bastante sencillo. Se deberán seguir los siguientes pasos:

- 1) Levantar una instancia de redis. Esto es descargar redis y correrlo con el comando `redis-server`.
- 2) Ejecutar el comando `npm install` en todas las carpetas que contienen un archivo **package.json**, incluidas las carpetas que contienen el **logger** como módulo.
- 3) Navegar a auth\_service/config y renombrar el archivo **arq\_private.txt** a **arq\_private.key**. Esta será nuestra clave privada que en realidad no deberíamos de exponer. Lo guardamos como un txt porque el git ignore nos saca todos los archivos con extensión key.
- 4) Pararse en src y hacer `node index.js` en una terminal separada para cada una de las siguientes carpetas **y en orden**.
  - a) auth\_service
  - b) AdminService
  - c) IdProviderMock
  - d) NotificationService
  - e) ReservationEmitter
  - f) ReservationProcessor

En caso de querer usarse:

- g) VacQueryTools
- h) Pending Service
- 5) Pararse en src y hacer `node index.js` en una terminal separada dentro de la carpeta **ReservationData**. En caso de que desee regular flujo de datos, modificar los parámetros en el método `getReservations` ubicado en el controlador de **ReservationData**. Su ubicación es:  
`ReservationData/src/controllers/reservation_controller/getReservations`

**Nota:** Quizás lo más acorde hubiese sido sacar estas variables que indican en el flujo del archivo de configuración para evitar la manipulación de código. Es un cambio que no llevaría más de 5 minutos.

## Manual de instalación para correr en base de datos local

**NOTA:** No recomendamos utilizar este manual de instalación ya que requeriría cambiar la configuración de todos los archivos de configuración del entorno de desarrollo **development**.

Instrucciones de uso para correr con mocks.

- 1) Levantar una instancia de **MongoDB**
- 2) Levantar una instancia de **redis-server**
- 3) Navegar a auth\_service/config y renombrar el archivo **arq\_private.txt** a **arq\_private.key**. Esta será nuestra clave privada que en realidad no deberíamos de exponer. Hacemos cambiar la extensión ya que el archivo arq\_private.txt no es persistido por git.
- 4) Crear 4 bases de datos de mongo:
  - a) adminService: dentro de esta base de datos crear las siguientes colecciones:
    - i) configurations
    - ii) quota
    - iii) users
    - iv) vaccinationCenters
    - v) validations
  - b) reservations: dentro de esta base de datos crear la siguiente colección:
    - i) reservations (son las reservas de VacPlanner)
  - c) populations: dentro de esta base de datos crear la siguiente colección:
    - i) populations
  - d) arqSoft: dentro de esta base de datos crear la siguiente colección:
    - i) reservations (son las reservas previstas)
    - ii) reservations2
    - iii) reservations3
- 5) Ir uno por uno a todos los servicios, pararse en los archivos de configuración (Servicio/src/config) del entorno de desarrollo **env\_dev** y cambiar allí la configuración de mongodb para que matchee la base de datos local levantada por el usuario.
- 6) Navegar al Proyecto ReservationData, y luego entrar a la carpeta data.

7) Ejecutar los siguientes comandos desde la carpeta data para agregar estos datos a la base de datos.

- a) mongoimport -d arqSoft -c population --type csv  
--file population.csv --headerline
- b) mongoimport -d arqSoft -c reservations --type csv  
--file reservations.csv --headerline
- c) mongoimport -d arqSoft -c reservation2 --type csv  
--file reservation2.csv --headerline
- d) mongoimport -d arqSoft -c reservation3 --type csv  
--file reservation3.csv --headerline

8) Ejecutar el comando `npm install` en todas las carpetas que contienen un archivo **package.json**, incluidas las carpetas que contienen el **logger** como módulo.

9) Pararse en src y hacer `node index.js` en una terminal separada para cada una de las siguientes carpetas **y en orden**.

- a)
  - i) auth\_service
  - ii) AdminService
  - iii) IdProviderMock
  - iv) NotificationService
  - v) ReservationEmitter
  - vi) ReservationProcessor

En caso de querer usarse:

- vii) VacQueryTools
- viii) Pending Service

10) Pararse en src y hacer `node index.js` en una terminal separada dentro de la carpeta **ReservationData**. En caso de que desee regular flujo de datos, modificar los parámetros en el método `getReservations` ubicado en el controlador de **ReservationData**. Su ubicación es:

`ReservationData/src/controllers/reservation_controller/getReservations`

**Nota:** Quizás lo más acorde hubiese sido sacar estas variables que indican en el flujo del archivo de configuración para evitar la manipulación de código. Es un cambio que no llevaría más de 5 minutos.



# Anexo

## Proceso de desarrollo

Siendo este el obligatorio que nos llevó más tiempo desde que comenzamos la carrera, al que mas seriedad le dedicamos, y el que más investigación y pienso contempló, consideramos sumamente necesario plasmar en el documento no solo la solución, sino que también todo el proceso que contribuyó a dicha solución, desde la evolución de nuestra arquitectura, hasta la manera de organizarnos.

## Organización Interna

Previo a comenzado el obligatorio, ya teníamos una noción del tiempo que nos llevaría este proyecto. Fue por eso que consideramos crucial aplicar los conocimientos aprendidos en las ingenierías de Software pasadas y organizarnos de manera excelente.

Nos planteamos metas, y utilizamos una plataforma relativamente nueva y muy poderosa llamada Notion para gestionar nuestro avance, ir construyendo la documentación, y poder planificar de manera correcta.

## Tableros

Vamos a plantear los requerimientos para ordenarlos a medida que los vayamos pensando/entendiendo/modelando.

List View		Properties	Filter	Sort	Search	...	New
<a href="#">Sistema en general</a>		Implementando	(FPT)	May 8, 2021 1:40 AM			
<a href="#">RESERVAS</a>			(FPT)	May 8, 2021 1:40 AM			
<a href="#">Req 1 - Crear reserva</a>		Implementando	(FPT)	May 8, 2021 1:11 AM			
<a href="#">Req 2 - Consultar dia agendado</a>		Realizado a medias	(FPT)	May 8, 2021 1:11 AM			
<a href="#">Req 3 - Cancelar reserva</a>		Pronto pero falta revision	(FPT)	May 8, 2021 1:11 AM			
<a href="#">AUTORIDAD SANITARIA:</a>			(FPT)	May 8, 2021 1:33 AM			
<a href="#">Req 4 - Mantenimiento Vacunatorios</a>		Pronto pero falta revision	(FPT)	May 8, 2021 1:11 AM			
<a href="#">Req 5 - Cupos Vacunatorios</a>		Pronto pero falta revision	(PFT)	May 8, 2021 1:11 AM			
<a href="#">Req 6 - Registro de Vacunacion</a>		Pronto pero falta revision	(FPT)	May 8, 2021 1:12 AM			
<a href="#">Req 7 - Consulta Vacunacion</a>		Pronto pero falta revision	(PTF)	May 8, 2021 1:12 AM			
<a href="#">HERRAMIENTA DE CONSULTAS</a>			(FPT)	May 8, 2021 1:12 AM			
<a href="#">Req 8 - Solicitud VacQueryTools</a>		Implementado	(FPT)	May 8, 2021 1:12 AM			
<a href="#">+ OTROS REQUERIMIENTOS</a>			(FPT)	May 8, 2021 1:12 AM			
<a href="#">Req 9 - Validacion sobre reserva</a>		Sin Informacion	(FPT)	May 8, 2021 1:12 AM			
<a href="#">Req 10 - APIs externas</a>		Sin Informacion	(FPT)	May 8, 2021 1:12 AM			
<a href="#">Req 11 - errores y fallas</a>		Realizado a medias	(FPT)	May 8, 2021 1:11 AM			
<a href="#">Req 12 - Proteccion datos</a>		Realizado a medias	(FPT)	May 8, 2021 2:44 AM			
<a href="#">Req 13 - Independencia Lec Esc</a>		Sin Informacion	(FPT)	May 8, 2021 2:46 AM			
<a href="#">Req 14 - Manejo de carga</a>		Pronto pero falta revision	(FPT)	May 8, 2021 2:47 AM			
<a href="#">Req 15 - Cambio algoritmo Vacunatorios</a>		Pensado	(FPT)	May 8, 2021 2:49 AM			
<a href="#">FIN OBLIGATORIO</a>			(FPT)	May 8, 2021 2:50 AM			

Utilizamos un tablero, donde marcamos los requerimientos del sistema para irlos analizando y procesando individualmente. Asociado a cada requerimiento pudimos marcar el estado en el cual estaba. Además, le asignamos a cada uno uno o más recursos: Paula, Tomás o Facundo.

## Req 1 - Crear reserva

The screenshot shows a Notion card for a requirement titled "Req 1 - Crear reserva". The card has the following properties:

- Created: May 08, 2021 1:11 AM
- Last Edited Time: Jun 19, 2021 1:25 PM
- Status: Implementando
- Participants: Facundo Lopez, Tomas De Angelis
- Date: Empty

Below the card, there is a "Add a property" button and a "Add a comment..." button. To the right of the card is a dropdown menu titled "Implementando" which lists various states:

- Pensado
- Pronto para implementar
- En proceso
- Sin Informacion
- Implementando
- Implementado
- Pronto pero falta revision
- Realizado a medias

## Solución

Atributos de calidad: PERFORMANCE

PM2 → Múltiples copias de computo.

Performance: Se necesita que el usuario en poco tiempo (menos de 5 min) tenga respuesta a su request, aun cuando el sistema esté saturado.

Sobre performance, vamos a usar un load balancer (pm2, aws) teniendo múltiples copias de computos con el procesor (o receiver si queremos). Vamos a hacer uso de asincronismo.

Por último, manejamos también diferentes vistas de las tareas y un kanban board para poder gestionar tareas pendientes. Además, Notion nos aportó otras vistas (de calendario por ejemplo) donde al marcar fechas de finalización para los requerimientos, podíamos ver cuantos días quedaban.

## Conclusión

En conclusión, creemos que la organización que tuvimos desde el día 0 aportó muchísimo y es algo que quizás hace un año no habríamos hecho. Con este tipo de proyectos complejos comprendemos la necesidad de las ingenierías de software y quedamos realmente muy conformes con la capacidad que tuvimos de ponerlas en práctica.

Adicionalmente, consideramos que la herramienta que elegimos para gestionar nuestro avance (Notion) es muy interesante y la recomendamos mucho.

## Histórico de Diagramas de Arquitectura

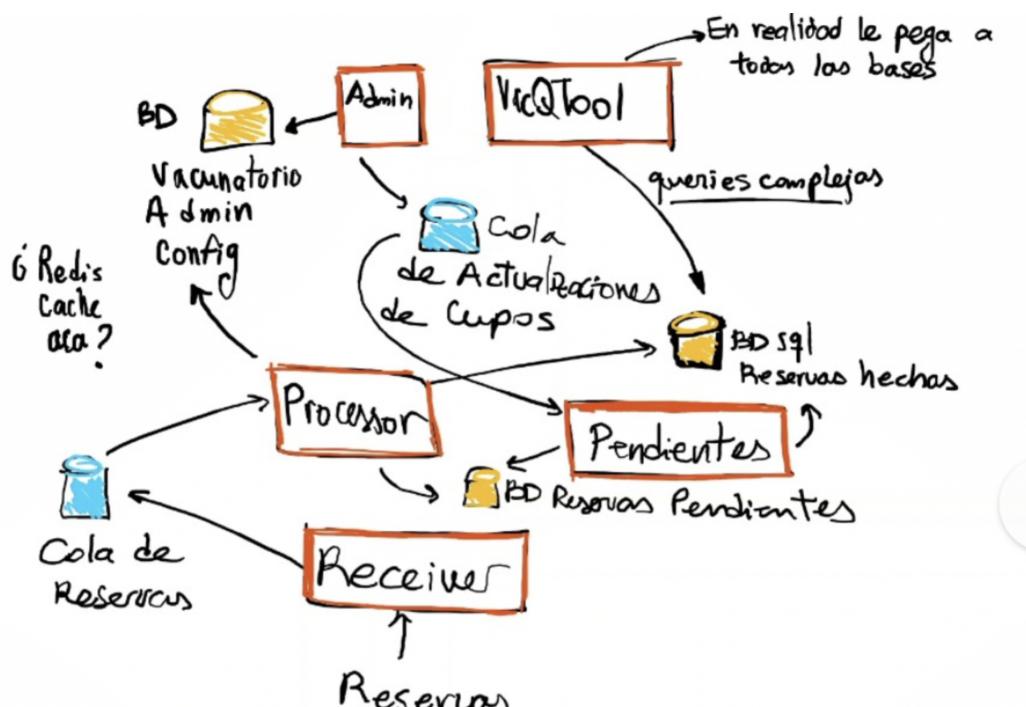
A la hora de enfrentarnos a este obligatorio, consideramos necesario empezar a trabajar lo antes posible para no tener problemas de tiempo y resolver cosas a las apuradas. Esto trajo grandes ventajas pero a su vez otras desventajas. La principal desventaja fue que a dicha altura no manejábamos con fluidez los diagramas dados en el curso, y por lo tanto diseñar una primera versión de la arquitectura era algo muy difícil.

La solución que encontramos fue realizar un primer diseño “a mano”, y posteriormente, gracias a una recomendación, utilizamos la herramienta CloudCraft. Esta herramienta nos pareció muy útil y fácil de usar, y nos gustó tanto que la continuamos usando en conjunto con los diagramas hasta la entrega final.

Pasaremos ahora a analizar el histórico de versiones del sistema:

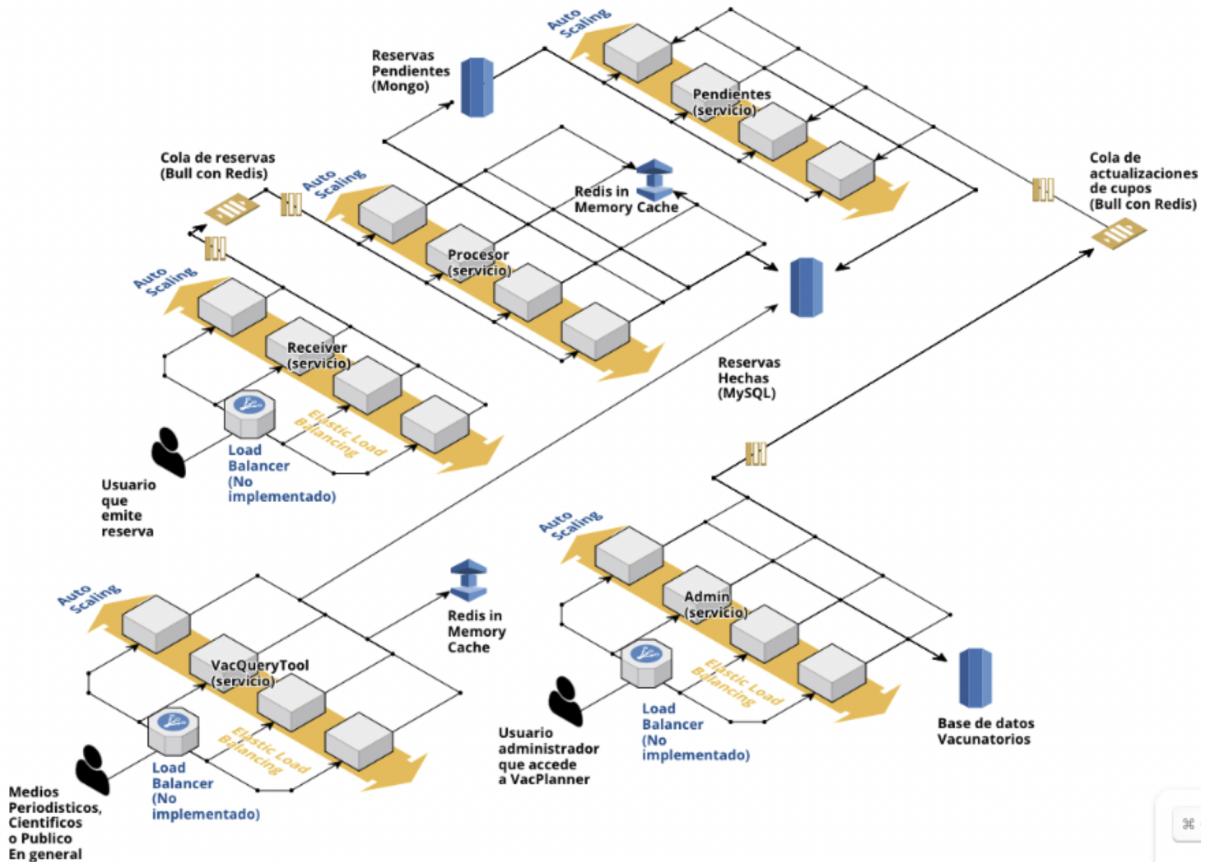
### 0) Diagrama a mano

Comenzamos plasmando los diferentes sistemas definidos por los requerimientos básicos y la interacción entre sistemas, con colas en el medio, bases de datos, etc.



→ Usar índices en las bd.

## 1) Primera versión en CloudCraft



Como vemos, la primera versión que plasmamos en CloudCraft tiene un parecido bastante importante con la versión final. Obviamente realizamos cambios en el medio pero ya desde la primera versión se notó que teníamos el flujo bastante resuelto.

El adminService tuvo cambios bastante chicos, y la cola de actualizaciones de cupos se mantuvo hasta el final, al igual que el servicio de pendientes.

Sin embargo, el sistema recibió cambios significativos: En principio consideramos tener dos bases de datos para reservas y separar por reservas pendientes y reservas hechas, o pensamos separar el servicio que reciba los requests y el que las procese, cosa que no prosperó (Explicaremos más adelante).

Luego de terminada esta primera versión de la arquitectura decidimos evaluar nuestro producto hasta el momento.

Llegamos a las siguientes conclusiones:

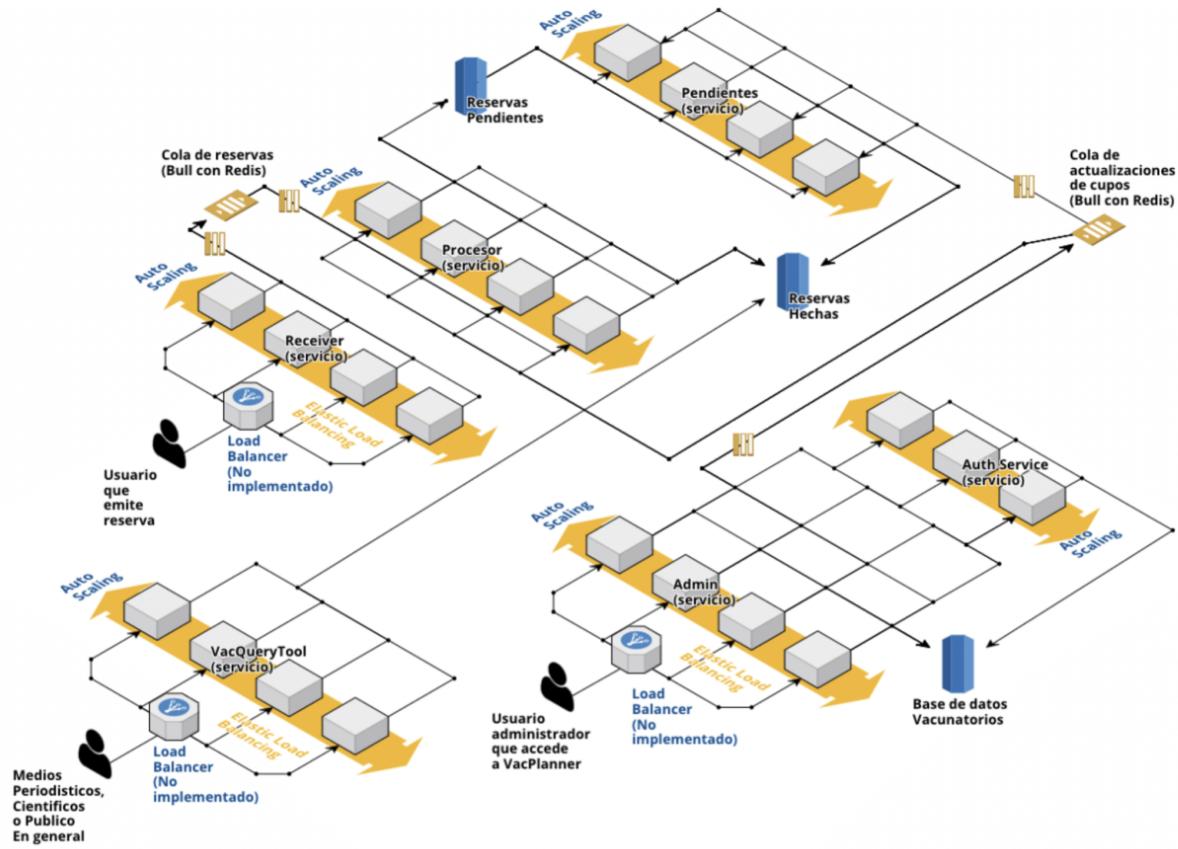
- El In Memory Cache de mongo no es necesario por ahora porque estamos asumiendo una decisión muy importante con una premisa muy débil (no

sabemos aún si las queries demoran mucho y requieren este cache). Esta decisión afectará en demasiado, ya que tranca el sistema a la hora de actualizar el caché y adicionalmente estamos tomando como hecho que al usuario no le interesa saber con total precisión el número de vacunados (Argumento que no fue aclarado). Si en un futuro vemos que el producto final está teniendo problemas a la hora de realizar dichas queries podemos ahí sí analizar la importancia de un In Memory Cache.

- Tenemos que analizar qué motor de base de datos usar más en profundidad teniendo en cuenta que operaciones puntualmente se harían en cada una. Para esto debemos analizar a detalle qué operaciones realiza VacQueryTools.
- Está la posibilidad de utilizar publisher subscriber para los nuevos cupos de vacunación, actuando el servicio de proceso y pendientes como subscriptores y el de admin como publicador.

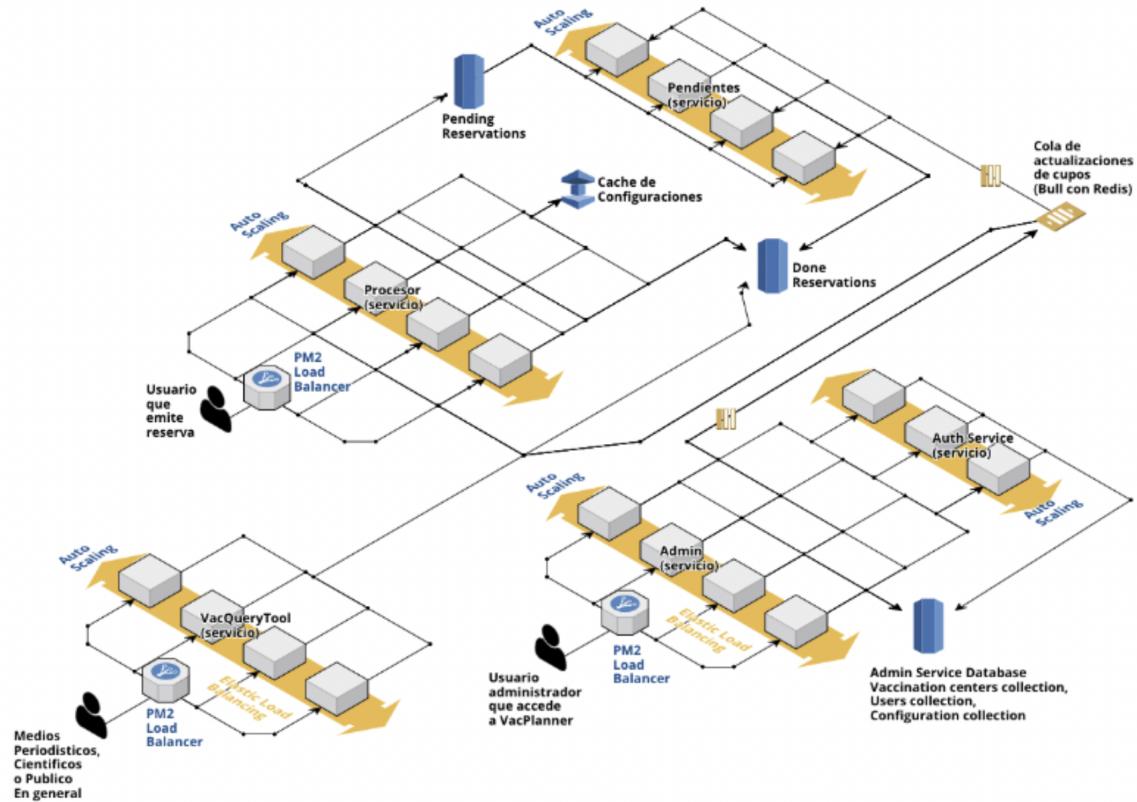
Estas conclusiones, así como también nuevas ideas, motivaron a la siguiente versión de arquitectura.

## 2) Segunda Versión en CloudCraft



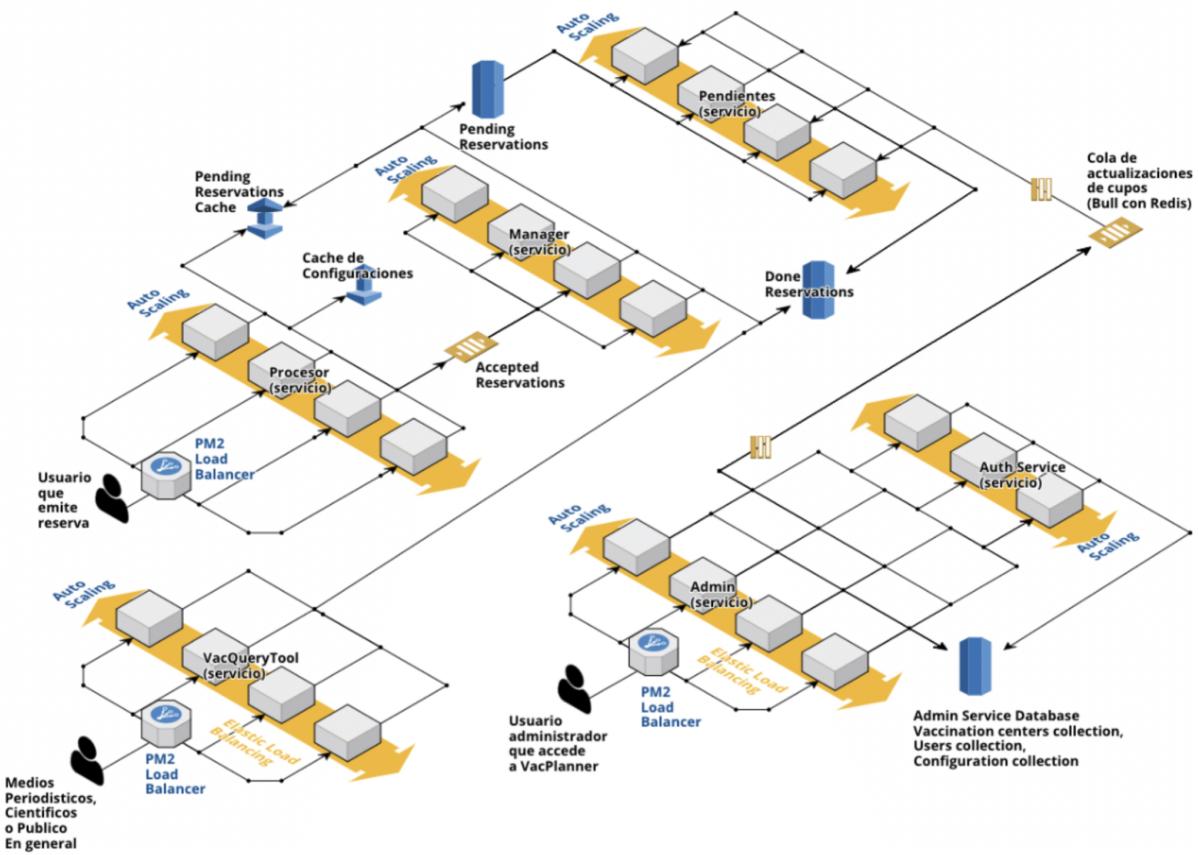
El principal problema que detectamos con esta versión fue entre el servicio Receiver y el Servicio Processor. Inicialmente pensamos en que el Receiver solamente se encargaba de aceptar los requests, y las delegaba a una cola de reservas, las cuales el procesador consumía, y validaba para luego responderle al Receiver con la velocidad que este pudiera. El problema con esto es que teníamos el requerimiento de responderle al usuario que realiza una request en un tiempo determinado, y de esta manera no podríamos asegurar cuánto tiempo iba a transcurrir entre que un usuario realiza una request y el processor le responde. Además, el procesador debería responderle al receiver y este hacer de pasamano para llevarle esa respuesta al usuario. Esto agregaba overhead innecesario y decidimos descartarlo, derivando en la siguiente solución.

### 3) Fusionando el Servicio Receiver y el Servicio Procesador



Habiendo eliminado la cola previamente mencionada, nuestra solución era algo de este estilo. Posteriormente, consideramos que el servicio Procesador estaba realizando demasiado trabajo recibiendo la request, procesándola, y luego insertándola en la database. Entonces pensamos en que parte de ese proceso podía ser delegada. La respuesta fue que podíamos delegar la inserción de la reserva a la base de datos luego de que esta haya sido validada y solo requiera de ser insertada. Gracias a esto llegamos a la siguiente solución.

#### 4) Integración de un servicio que solo inserte la reserva en la DB.

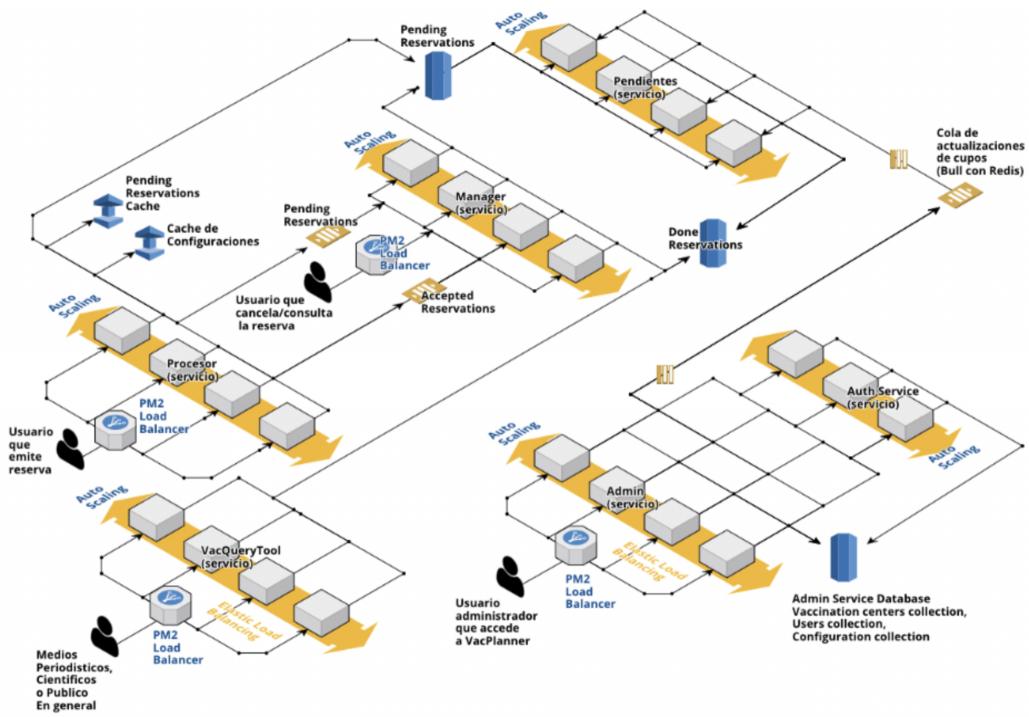


Esta solución está muy cerca de ser la final, con al excepción de unos pocos ajustes:

La próxima versión fue motivada por la necesidad del usuario de acceder al servicio de manager para cancelar/consultar una reserva. El acceso a cancelar/consultar reservas podría haber sido realizado por el Procesador pero consideramos que no era su responsabilidad. Consideramos que el Servicio de Manager era el más adecuado para realizar esto debido a que su responsabilidad es el acceso a las bases de datos de reservas y pendientes.

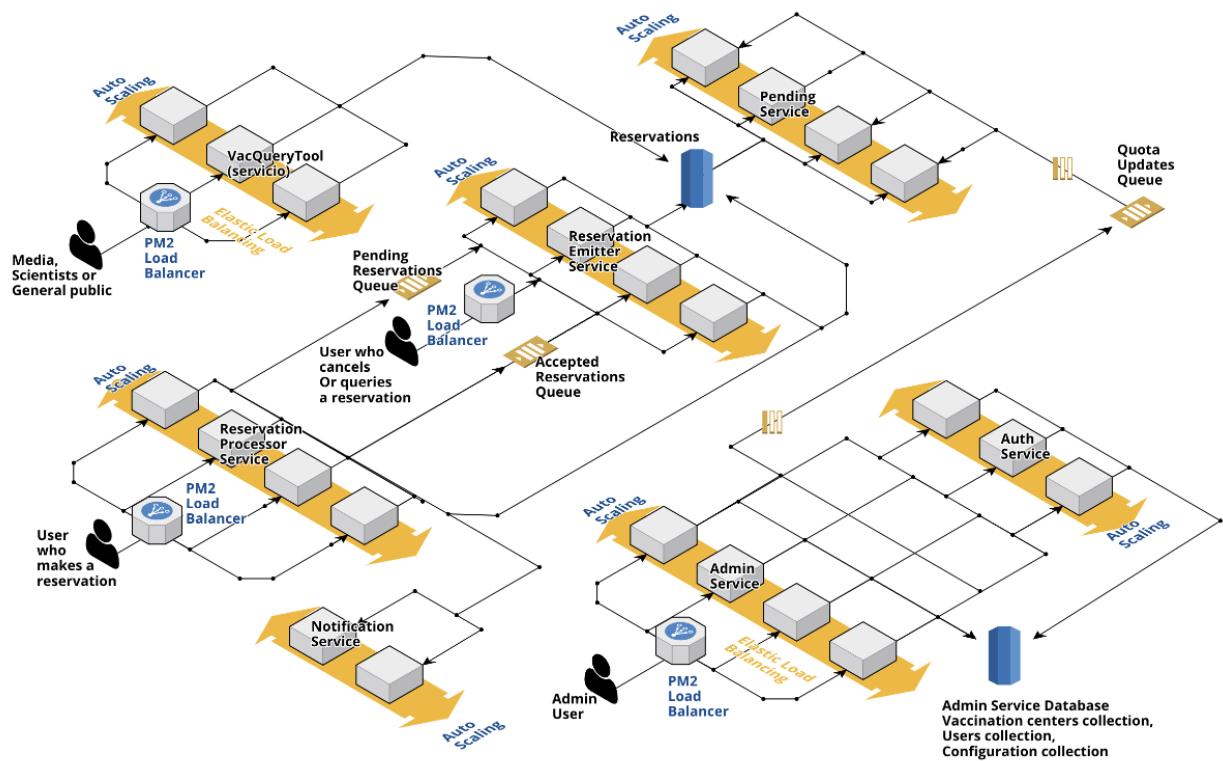
Además, faltaba una segunda cola para las reservas que fueron relegadas y debían de ser encoladas.

## 5) Penultima version



Esta versión solamente necesito un par de variantes para llegar a su etapa final. Las principales fueron que decidimos utilizar una única base de datos para las reservas en vez de separarlas, y que los cachés que originalmente habíamos planteado resultaron no ser tan útiles ni necesarios. Habiendo realizado estos ajustes, llegamos a la versión final.

## 6) Versión final



## Detalles finales que no llegamos a cubrir

En esta sección documentaremos detalles o pequeñas cosas que no llegamos a cubrir. La idea es presentar cada uno de los siguientes puntos, comentar también cómo se harían, y estimar el tiempo que llevaría.

Este apartado se debe a que nos propusimos como objetivo dejar el código pronto el domingo previo a la entrega y no tocar más por miedo a poder romper algo. Posterior a esa fecha, se hizo algún ajuste, donde el más grande fue **formatear la totalidad del código**, pero sin embargo, no se hizo ningún cambio en cuanto a funcionalidad.

Consideramos que obviamente hubiera estado bueno cumplir con estos puntos, pero **ponerse a tocar estas cosas de código los días previos a la entrega era un riesgo demasiado grande para lo que nos garantizaba**. Más todavía programando en un lenguaje no tipado como JS en el cual para probar tus cambios deberías volver a correr los servicios y hacer pegas por postman.

Sin embargo todos son cambios fáciles, que requieren de poco tiempo de programación y pueden ser realizados muy faciles, ademas de no afectar a la arquitectura que era lo primero en lo que debíamos enfocarnos. Estos cambios sería lo primero que hiciéramos si mañana diéramos comienzo a una versión 2 del sistema, y nos llevarían en total menos de un día de programación.

### 1 - Console.logs

Nos quedaron muchos console.log() que utilizamos para “debuggear” (junto con el debugger) que deberíamos haber borrado. Esto ensucia bastante el código y no queda atractivo a la vista. Borrar estos console.log() llevaría entre 10 y 15 minutos. Ninguno debería actuar oficialmente como logger ya que el logger por su parte imprime por consola. La realidad es que esos console.log() no deberían afectar en nada a la funcionalidad o al flujo si son borrados, pero preferimos no contar con ese riesgo ya que tiempo para probar todo nuevamente no es algo que nos sobre.

### 2 - Comentarios

Otro punto bastante tonto, vimos en clean code que el mejor comentario es aquel que no tiene necesidad de hacerse. Nosotros tenemos algún que otro comentario que sería bueno borrar. Este cambio llevaría entre 10 y 15 minutos.

### 3 - Imports no utilizados

Pese a que intentamos no dejar *imports* sin utilizar, puede que haya alguno que todavía este. Borrar los imports no utilizados es un cambio que puede llevar entre 10 y 15 minutos ya que el compilador marca aquellos imports no utilizados.