Descripción de Arquitectura

VacPlanner

Joaquín Lamela (233375)
Agustín Hernandorena (233361)
Felipe Najson (232863)

https://github.com/ORTArqSoft/-233375-233361-232863

Junio de 2021

ÍNDICE

1. Introducción

1.1 Propósito

El propósito del presente documento es proveer una especificación completa de la arquitectura de **VacPlanner.**
En el mismo, se mostrarán las principales decisiones tomadas, así como también, el razonamiento seguido para resolver correctamente los requerimientos significativos planteados.

2. Antecedentes

2.1 Propósito del sistema

A raíz de la situación sanitaria del país, facilPlan es una empresa que se dedica a desarrollar software, la cual quiere realizar un aporte a los países ante la posibilidad de una nueva pandemia a futuro. Esto permitirá que la vacunación sea una actividad con mayor frecuencia que la actual. Donde, a modo de resumen los objetivos generales del sistema son:

- Cubrir las necesidades del ente regulador de la campaña de vacunación para cada país.

- Facilitar el proceso de reserva y consultas sobre el estado del plan, reduciendo la ansiedad de las personas interesadas en la campaña de vacunación.

- Brindar información en cualquier momento sobre la situación del plan de vacunación.

- Permitir el análisis de diversas variables históricas asociadas a los planes (como pueden ser cantidad de reservas pendientes, entre otros).

Los usuarios establecidos del sistema son:

- **Ciudadano**: son aquellas personas que desean vacunarse. Para esto requieren agendarse (realizar una reserva). Sobre su reserva quieran realizar consultas sobre el estado así cómo la gestión (cancelación).

- Autoridad Sanitaria: es el organismo que se encuentra conformado por dos tipos de usuarios, **los administradores** encargados de planificar el plan de vacunación teniendo en cuenta las características de las vacunas adquiridas, disponibilidad de vacunatorios (incluyendo la disponibilidad de vacunas que el mismo posee) así como también las prioridades sanitarias de la población para la asignación de cupos. Por otro lado, encontramos a los **usuarios vacunadores**, los cuales son encargados de realizar el registro del acto de vacunación de una persona en un vacunatorio.

- Terceros: este grupo de usuarios está compuesto por **los científicos** los cuales pueden consultar las distintas variables del plan de vacunación para estudiar la efectividad del mismo, como también **medios periodísticos y público en general**, los cuales pueden consultar información sobre el avance del plan de vacunación.

Un dato no menor, es que la población en sí podrá agendarse para vacunarse utilizando una diversa variedad de medios, como pueden ser aplicaciones en dispositivos móviles, bots asociados a WhatsApp u otras aplicaciones de mensajería, por lo cual el sistema debe ser fácilmente adaptable para cualquiera de los clientes que la utilicen. Donde mediante el uso normal del sistema, una vez

realizada la reserva se comunicará los datos de la misma por medio del envío de SMS al usuario correspondiente.

2.2 Requerimientos significativos de Arquitectura

2.2.1 Resumen de Requerimientos Funcionales

ID Requerimiento	Descripción
RF 1 Agendarse para vacunación	El sistema deberá permitir a las personas agendarse para vacunarse brindando: documento de identidad, celular, fecha de reserva, horario, departamento y zona.
RF 2 Consultar el día agendado	El sistema deberá permitir a las personas que se han registrado para la vacunación, consultar los datos de su agenda, brindando una cédula de identidad válida para la cual exista reserva.
RF 3 Cancelar reserva	El sistema deberá permitir a las personas registradas para la vacunación, cancelar su reserva brindando una cédula de identidad válida y el código de reserva.
RF 4 Mantenimiento de vacunatorios	El sistema deberá permitir a un usuario administrador, asignar vacunatorios a distintas zonas de cada departamento.
RF 5 Definición de cupos para los vacunatorios	El sistema deberá permitir a un usuario administrador, definir los cupos de vacunación para cada vacunatorio.
RF 6 Registro de vacunación	El sistema deberá permitir a un usuario vacunador, registrar un acto vacunal.
RF 7 Consultas sobre el estado del plan de vacunación	El sistema deberá permitir a un usuario administrador, realizar consultas acerca del estado del plan de vacunación.
RF 8 Solicitudes de la aplicación VacQueryTool mediante API Rest	El sistema deberá permitir la realización de consultas complejas mediante una API REST sobre la base de datos del plan de vacunación.
RF 9 Configuración de validación sobre campos de reserva	El sistema deberá permitir configurar los tipos de campos de la reserva y las validaciones que la plataforma debe realizar al recibir una solicitud de reserva.
RF 10 Configuración de APIS externas	El sistema deberá permitir configurar los endpoints para invocar la API del proveedor de Identificación Civil y del proveedor de envío de SMS.
RF 11 Gestión de errores y fallas	El sistema deberá ante la ocurrencia de una falla o error, proveer toda la información necesaria que permita a los administradores hacer un rápido diagnóstico de las causas.
RF 12 Protección de datos y acceso al sistema	El sistema deberá contar con algún mecanismo que permita autenticar y autorizar a los usuarios de la autoridad sanitaria minimizando las vulnerabilidades que exponga el sistema.

2.2.2 Resumen de Requerimientos de Atributos de Calidad

ID Requerimiento	ID Requerimiento de Calidad o restricción	Descripción
RF 1 Agendarse para vacunación	*AC 1 Perfomance*	Se expresa claramente que el tiempo de respuesta de este requerimiento es uno de los aspectos que se quiere optimizar, indicando que sería deseable que la respuesta a la

		solicitud esté dentro de una ventana de tiempo de entre 30 segundos y 5 minutos.
	AC 2 Modificabilidad	Se expresa que los requerimientos están basados en los datos y características del Uruguay, siendo fundamental tener en consideración que cada requerimiento se adecue/adapte fácilmente a las características de otros países.
	AC 3 Disponibilidad	Se expresa que se debe lograr la mayor capacidad de procesamiento posible, sin pérdida de datos.
	RS1 API REST	Se indica que el requerimiento debe implementarse mediante una API REST para que diferentes frontends puedan interactuar con *VacPlanner*.
RF 2 Consultar el día agendado	*RS2 API REST*	Se indica que el requerimiento debe implementarse mediante una API REST para que diferentes frontends puedan interactuar con *VacPlanner*.
RF 3 Cancelar reserva	*RS3 API REST*	Se indica que el requerimiento debe implementarse mediante una API REST para que diferentes frontends puedan interactuar con *VacPlanner*.
RF 4 Mantenimiento de vacunatorios	*AC 4 Seguridad*	Se debe tener en cuenta que únicamente usuarios autenticados y autorizados pueden llevar a cabo la definición de vacunatorios para una zona y un estado. Por lo que, la seguridad aquí es un aspecto fundamental, ya que si no existiese este chequeo sucedería que cualquier persona agregaría vacunatorios.
RF 5 Definición de cupos para los vacunatorios	*AC 5 Seguridad*	Se debe tener en cuenta que únicamente usuarios autenticados y autorizados, pueden definir cupos para un vacunatorio, por lo que, la seguridad es un aspecto muy relevante.
	AC 6 Modificabilidad	Sucede que una vez que se actualizan los cupos, aquellos

		usuarios con estado pendiente deben ser asignados al vacunatorio correspondiente. Esto se produce por medio de un algoritmo el cual permite asignar cupos a los vacunatorios. Entonces dicho algoritmo debe ser fácilmente modificable para los diferentes criterios existentes.
RF 6 Registro de vacunación	*AC 7 Seguridad*	Se debe tener en cuenta que únicamente un usuario vacunador puede registrar un acto vacunal, por lo que, la seguridad es un aspecto relevante.
RF 7 Consultas sobre el estado del plan de vacunación	*AC 8 Seguridad*	Se debe tener en cuenta que únicamente un usuario administrador puede realizar consultas acerca del plan de vacunación.
RF 8 Solicitudes de la aplicación VacQueryTool mediante API Rest	*AC 9 Performance*	Se debe asegurar que la latencia para las consultas se encuentre optimizada (en promedio, debajo de los 2 segundos) para minimizar el jitter.
	AC 10 Interoperabilidad	Es necesario definir la interfaz que mejor se adecue al tipo de solicitudes que se realizarán.
	RS 4 API REST	Se indica que las consultas tienen que poder realizarse mediante una API REST.
RF 9 Configuración de validación sobre campos de reserva	*AC 11 Modificabilidad*	Se debe tener en cuenta que el sistema debe permitir configurar los tipos de campos de la reserva y las validaciones que la plataforma debe realizar en recibir una solicitud de reserva, por lo que un aspecto fundamental a tener en cuenta es la modificabilidad, que ayuden a minimizar la codificación de nuevos módulos.
	AC 12 Seguridad	Al ser los parámetros de configuración sobre campos de reserva un aspecto sensible, consideramos que no sería apropiado exponer una API pública, sino que sólo los usuarios administradores

		autorizados y autenticados puedan configurar las validaciones sobre los campos de reserva.
RF 10 Configuración de APIS externas	*AC 13 Modificabilidad*	La plataforma va a utilizar una o más APIS externas y se desea minimizar el costo de codificación al cambiar o incorporar una nueva.
	AC 14 Seguridad	La plataforma debe permitir la configuración de APIS externas, por lo cual se debe tener en cuenta que únicamente un usuario administrador puede realizar la configuración de las APIS externas, ya que si cualquier usuario pudiera hacerlo esto implicaría problemas a futuro para por ejemplo el envío masivo de SMS´s con los datos de las reservas.
RF 11 Gestión de errores y fallas	*AC 15 Modificabilidad*	La plataforma debe contemplar la posibilidad de poder cambiar las herramientas o librerías que almacenan la información generada por los errores y fallas. También debe contemplar la reutilización de esta solución para con otras aplicaciones con el menor impacto posible en el código.
	AC 16 Seguridad	Se indica que la plataforma debe proveer información que permita conocer el detalle de la actividad de los usuarios que requieren autenticación, y de esta forma, se pueden detectar la presencia de intrusos haciendo actividades para las cuales no están autorizados.
	AC 17 Disponibilidad	Se debe contemplar que en el caso que ocurra una falla o cualquier error, el sistema se debe mantener activo, además de proveer toda la información necesaria que permita a los administradores poder saber de manera rápida y precisa la causa de los errores.

RF 12 Protección de datos y acceso al sistema	*AC 18 Seguridad*	Se indica que la plataforma debe contar con algún mecanismo que permita autenticar y autorizar a los usuarios de la autoridad sanitaria, minimizando las vulnerabilidades que exponga el sistema.

Nota: Se presenta como restricción, que el backend de todos los requerimientos presentados en la tabla precedente deben ser desarrollados utilizando la tecnología Node JS.

3. Documentación de la arquitectura

3.1 Vistas de Módulos

En esta sección se describirán las principales representaciones de la vista de los módulos. Estas vistas tienen el propósito de enseñar las principales unidades de implementación del sistema.

3.1.1 Vista de Descomposición

3.1.1.1 Representación primaria

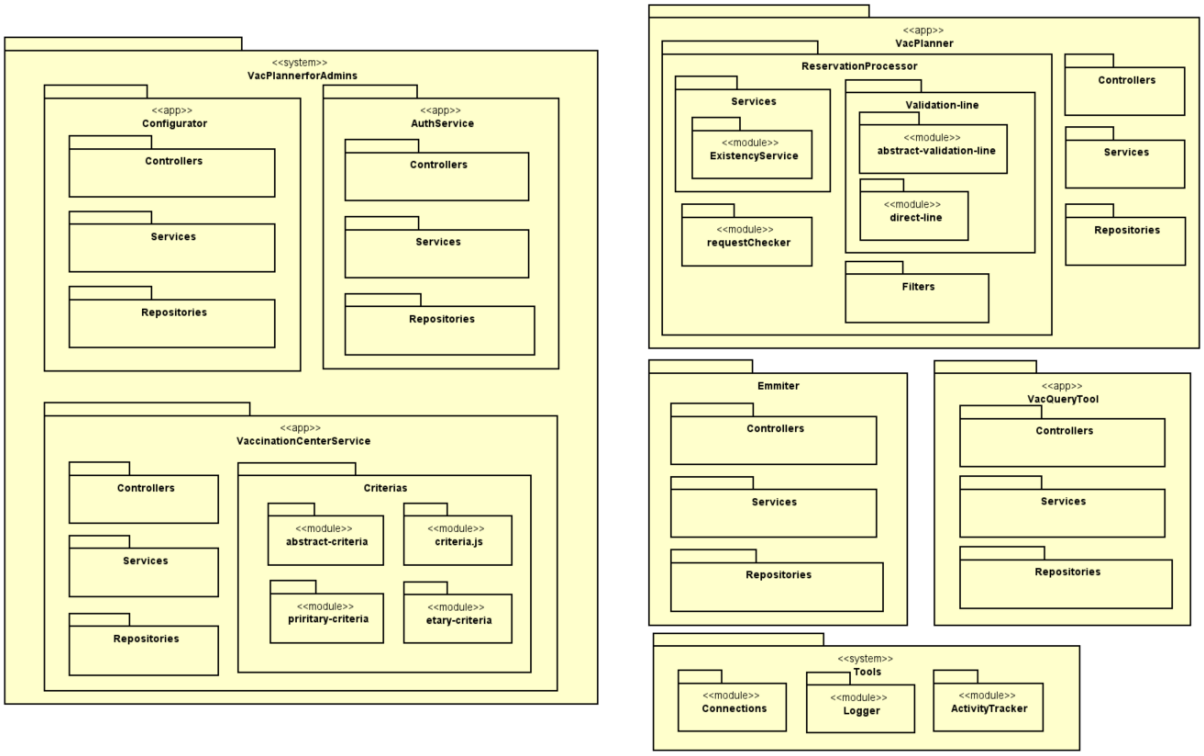

Figura 1. Diagrama de descomposición de la solución.

3.1.1.2 Catálogo de elementos

En los diagramas anteriores se pueden apreciar los distintos módulos en los que se separó *VacPlanner*, estos módulos fueron separados teniendo en cuenta su responsabilidad y cada uno es totalmente independiente de la implementación de los demás, haciendo que las responsabilidades de los módulos sean cohesivas y permitiendo que si uno, por algún motivo cambia, este no afecte a los otros. Donde se puede encontrar:

VacPlannerForAdmins: este módulo está orientado a los administradores y vacunadores.
Dentro encontramos diferentes subsistemas, los cuales son el configurador, el servicio de autenticación y el servicio orientado a los centros de vacunación. Justamente son elementos que únicamente los puede manipular personas que tienen el permiso de la Autoridad Sanitaria correspondiente, por lo cual se encuentran aquí dentro.

- ***Configurator****:* Este es uno de los módulos más importantes ya que la plataforma debería brindar la posibilidad de configurar la mayor cantidad de aspectos del país y el plan de vacunación. Se encarga del procesamiento de las configuraciones de los parámetros a utilizar, para los campos de las reservas y las validaciones correspondientes. Así cómo también, la configuración de las APIS externas minimizando el costo de codificación al querer cambiar o incorporar una nueva.

- ***AuthService****:* Este módulo dentro de VacPlannerForAdmins es el más importante. Se encarga de la autenticación al sistema, y sin su existencia, no se podrían realizar ninguna de las operaciones relacionadas con la configuración de los parámetros y API´s externas, así cómo las operaciones vinculadas al mantenimiento de los vacunatorios, debido a que requieren autenticación.

- ***VaccinationCenterService****:* Este módulo tiene la responsabilidad de exponer una interfaz que permite la asignación de vacunatorios a distintas zonas de departamento, la definición de nuevos cupos de vacunación para cada vacunatorio, así como también, el registro de vacunación de una persona y las consultas sobre el estado del plan de vacunación.
  Es decir, recibe las requests provenientes de uno de los usuarios de la Autoridad Sanitaria y realiza cierto procesamiento para poder dar respuesta a esas solicitudes. Cabe aclarar que la operación de registrar el acto vacunal, es realizada única y exclusivamente por un vacunador (si cuenta con las credenciales correspondientes, una vez que se autentica el token devuelto únicamente tiene permisos para realizar la inscripción del acto vacunal), mientras que el resto de operaciones mencionadas son realizadas única y exclusivamente por los administradores (si cuenta con credenciales correctas, una vez autenticado, el token devuelto tiene los permisos para realizar dichas operaciones).

VacPlanner: este módulo tiene la responsabilidad de exponer una interfaz que permite la creación, consulta y eliminación de la agenda para vacunarse, es decir, recibe las requests provenientes del cliente y realiza cierto procesamiento para poder dar respuesta a esas solicitudes.
Al ser un subsistema complejo, está compuesto por una serie de módulos en donde cada uno de ellos tiene una responsabilidad en específico.

- ***Controllers****:* Se encarga de recibir las requests por parte del cliente. Una vez que obtiene la misma, delega el procesamiento a los elementos pertenecientes al módulo Services, y una vez que obtiene la respuesta se la devuelve al cliente.

- ***Services***: Recibe la solicitud por parte del controlador, y utilizando los servicios provistos por el módulo ReservationProccesor, determina si los campos de la reserva son válidos o no. En el caso de no ser válidos simplemente retorna el error. En otro caso, realiza lo que es el procesamiento de la reserva a fin de saber si la misma debe ser o no confirmada en función de si hay o no cupos en el departamento y zona escogida.

- ***ReservationProccesor*** este módulo se encarga del procesamiento de las reservas que llegan al sistema, determinando si los campos que tienen las mismas son o no valores válidos según lo definido mediante la API de configuración de las validaciones sobre los campos de reserva. A su vez, este módulo está compuesto por otros módulos en donde cada uno de ellos tiene una responsabilidad particular en este proceso.

    - ***Services*** este módulo posee el servicio que se comunica mediante HTTP con el proveedor de identificación civil a fin de poder determinar si la persona que está queriendo realizar reserva se encuentra en la base de datos poblacional.

    - ***RequestChecker*** este módulo posee la responsabilidad de preparar la estructura que permite realizar la validación de cada uno de los campos (filtros), así como también, de cargar cada uno de los parámetros de configuración de los campos de validación que se ingresan mediante la API de configuración.

    - ***Filters:*** este módulo contiene cada uno de los filtros que se utilizan para realizar las validaciones sobre los campos de las reservas que ingresan. Los mismos, tienen un orden en particular que se determina colocando un número al comienzo del nombre de cada uno de ellos, implicando de que si un filtro posee un número menor a otro significa que el mismo se ejecutará antes.

  En concreto, los filtros con los que cuenta al momento la plataforma, son los que consideramos necesarios para el caso de Uruguay, y se describen a continuación.

    1. **1_checkMissingData:** Se encarga de verificar que la reserva introducida posee los campos requeridos, es decir, que contiene: documento de identidad, celular, fecha de reserva, horario de vacunación, departamento y zona.

    2. **2_checkDocumentId:** Se encarga de verificar que el documento de identidad ingresado sea correcto en cuanto al largo requerido (mediante la configuración ingresada), y además, se encarga de verificar que el dígito verificador sea correcto.

    3. **3_checkCellphone:** Se encarga de verificar que tanto el largo como el prefijo del celular ingresado sean los requeridos según lo ingresado mediante la API de configuración de validación de parámetros.

    4. **4_checkReserveDate:** Se encarga de validar que la fecha de reserva ingresada sea correcta, es decir, que sea posterior a la fecha actual del día, y a su vez, en este filtro también se transforma la fecha al formato introducido mediante la API de configuración, que para el caso de nuestro país es: “DD/MM/YYYY”.

    5. **5_checkScheduleTime:** Se encarga de validar que el horario de vacunación ingresado sea correcto, tomando en cuenta lo indicado desde la API de configuración.

6. **6_checkUbication:** Se encarga de validar que la ubicación (departamento y zona) ingresada sean valores correctos, tomando en cuenta las validaciones ingresadas mediante la API de configuración.

- ValidationLine: Este módulo, contiene una estructura que permite almacenar cada uno de los filtros, por lo que se tiene una función que permite agregar filtros a la estructura, así como otra que permite correr cada uno de los filtros de acuerdo a la convención de nombres previamente descrita.
Cabe destacar, que ante la detección de un campo inválido por parte de uno de los filtros, se corta la ejecución y no se continúa con las validaciones en los restantes filtros. Concretamente, la función que realiza la ejecución de cada uno de los filtros, termina devolviendo la reserva tal como se ingresó desde un inicio si todos los campos son correctos, y un objeto de error en otro caso.

Emitter*:* este módulo tiene la responsabilidad de recibir las reservas luego de que las mismas fueron procesadas, y se haya validado que los campos que se ingresaron son correctos. Una vez que sucedió eso, las reservas llegan a este módulo, que como su nombre lo indica, se encarga de emitir las reservas a los lugares necesarios, en el caso de esta plataforma, se encarga de enviar las reservas al emulador externo de envío masivo de SMS, así como también, de persistir la reserva tanto en la base de datos de *VacPlanner* como en la de *VacQuery (*para poder hacer consultas complejas).

VacQueryTool: este módulo tiene la responsabilidad de exponer una interfaz que permite realizar consultas complejas sobre la base de datos sobre el plan de vacunación. Es decir, recibe las requests provenientes del cliente y realiza cierto procesamiento para poder dar respuesta a esas solicitudes. Esto dependiendo la consulta que se deba realizar, ya que un ejemplo sobre una consulta compleja existente es que, dado un rango de fechas, listar por departamento y por horario la cantidad de vacunas aplicadas. Entonces esta request es recibida, y procesada determinando qué operación se debe realizar, para al final del procesamiento consultar sobre la base de datos sí para el rango de fechas recibido hay vacunas aplicadas agrupandolas por departamento y horario.

Tools*:* Este módulo engloba todo lo referido a módulos que se utilizan en diversas partes del código.

- ***Connections:*** Este módulo se encarga de manejar las conexiones a la base de datos. Su responsabilidad es inicializar la base de datos y cargar los diferentes modelos presentes en cada una de ellas, en donde esto favorece nuevamente la modificabilidad ya que si se quiere realizar un cambio en un modelo es el único lugar en dónde se debe modificar, así cómo también si se quiere modificar la conexión a la base de datos es el único punto donde se debe modificar.

- ***Logger****:* Este módulo engloba parte de la funcionalidad del requerimiento número 11. Se implementa un sistema de logging con librerías específicas para que todos los módulos que lo utilizan sean independientes de la misma y sólo requieran la implementación que dicha librería utiliza, escondiendo la implementación del sistema de logging de terceros. Esta implementación puede ser cambiada por cualquier otra sin tener ningún impacto en alguno de los módulos que la utilizan. Esto último favorece la modificabilidad, cumpliendo lo establecido por el requerimiento, dónde se indicaba que el sistema debía contemplar la posibilidad de poder cambiar las herramientas o librerías que almacenan la información generada por los errores y fallas. Además de contemplar la reutilización de esta solución, se encarga de registrar fallas o cualquier tipo de error, permitiendo que el sistema provea toda la información necesaria para que los administradores del mismo puedan realizar un diagnóstico rápido y preciso de la causa de los errores.

- ***ActivityTracker****:* Este módulo también engloba parte de la funcionalidad del requerimiento número 11. Pero a diferencia del logger, se enfoca en que el sistema deba proveer suficiente información para poder conocer el detalle de actividad de los usuarios administradores y vacunadores. Entonces como no tienen las mismas responsabilidades, lo correcto fue separarlo en dos módulos diferentes, en donde este módulo se enfoca en el atributo de calidad Seguridad. Se enfoca en dicho atributo ya que para los usuarios administrados y vacunadores, para cada operación que realicen se guarda en detalle qué fue lo que se realizó, quién lo realizó y cuando se realizó, actuando como una pista de auditoría (auditory tail). Con dicha información almacenada permite determinar las acciones ocurridas e identificar atacantes, para a futuro generar nuevas herramientas de Seguridad en el sistema.

3.1.1.3 Decisiones de diseño

Separación de responsabilidades VacPlanner y VacPlannerForAdmins

Una de las decisiones de diseño más importantes que se tuvo que tomar es aquella con respecto a la separación de responsabilidades entre los módulos VacPlanner y VacPlannerForAdmins.
Observando las responsabilidades de las operaciones a realizar encontramos que ninguna de ellas tenían algún punto en común para tenerlas en único módulo, debido a que la mitad de las responsabilidades estaban asociados a los usuarios autorizados únicamente por la Autoridad Sanitaria, además de que dichas operaciones requieren autenticación, por lo cual no tendrían sentido para un ciudadano.
Es por esto que encontramos la necesidad de que estuvieran separadas, ya que si ocurría un cambio en alguna de las funcionalidades de los usuarios administrador y vacunadores, se verían también afectadas las de los usuarios, de forma tal que para favorecer la **modificabilidad** encontramos la utilización de la táctica ***Split Module****,* la cual aplica en su totalidad, dado que si todas las operaciones se mantuvieran en único módulo, sería un módulo de gran capacidad, en donde el costo de realizar una modificación sería alto. También, encontramos la aplicación de la táctica de Modificabilidad ***Incrementar la coherencia semántica****,* ya que las responsabilidades asociadas no tienen el mismo propósito (unas están asociadas a la creación de reserva, visualización y eliminación de la misma, mientras que las otras operaciones existentes se encuentran asociado a la creación de vacunatorios, agregado de periodos, etc, las cuales requieren autenticación para realizarse), de forma tal que basándonos en el argumento mencionado dichas operaciones deben ubicarse en módulos diferentes. Es por esto, que tomamos la decisión de separarla en dos módulos.

Proveer interfaz que permita configurar la validación sobre los campos de la reserva.

Otra decisión de diseño realizada en pos de favorecer la **modificabilidad**, y en particular lo indicado respecto a que los datos están basados en Uruguay y que deben ser fácilmente modificables, fue la de proveer una interfaz que permita configurar la validación sobre los campos de la reserva. En este proceso de reserva, se tuvieron en cuenta varios aspectos que ayudan a favorecer la modificabilidad, entre los que podemos destacar que al leer los filtros que realizan las validaciones se utiliza una suerte de reflection, por lo que, estamos aplicando la táctica ***Defer Binding,*** ya que estamos leyendo y decidiendo en tiempo de ejecución al leer un assembly, lo que implica que si se desean agregar nuevos filtros pueda hacerse fácilmente sin necesidad de modificar los módulos existentes.

Cabe destacar, que cada uno de estos filtros posee dos operaciones: *checkConditions* y *setParameters.* La primera de ellas, como su nombre lo indica, se encarga de chequear las condiciones de la reserva dependiendo de cada uno de los filtros, mientras que la segunda, recibe los parámetros provistos por la API de configuración de validación sobre los campos de reserva, y guarda dichas validaciones en variables locales para poder utilizarlas posteriormente.
Por lo mencionado, si en el futuro se desea agregar un nuevo filtro de validación, simplemente se debería agregar un nuevo archivo en el módulo *Filters,* conteniendo las dos operaciones que se

describieron previamente, sin necesidad de modificar nada de lo codificado en los módulos existentes.

Introducción de nuevos clientes

Otra decisión de diseño realizada en pos de favorecer la **modificabilidad**, es asociada a que la plataforma va a utilizar una o más APIS externas y se desea minimizar el costo de codificación al cambiar o incorporar una nueva API. Es por esto, que se tomó la decisión de que dichas API´s fueran fácilmente modificables, dónde se provee una interfaz que permite el agregado y actualización de clientes. Para este proceso de configuración se tomó un aspecto clave tanto en Performance (hablaremos más adelante sobre esto) como modificabilidad, ya que a la hora de utilizar estos clientes los mismos se setean en único punto, lo cual si ocurre un error, hay un único lugar en el que se debe modificar, en lugar de tener múltiples archivos de configuración de donde se leerían las URL de los clientes externos.

Filtros extensibles

Otro aspecto a destacar es que buscamos favorecer la **modificabilidad** en lo que es la estructura que contiene los filtros y realiza la ejecución de los mismos, ya que mediante la aplicación de la táctica ***Defer Binding***, contamos con un *AbstractLine* que presenta las dos operaciones que debe contener la estructura: *run* (que permite ejecutar cada uno de los filtros), y *use* (que permite agregar un nuevo filtro a la estructura), y se decide en tiempo de ejecución que *Line* concreto se va a utilizar. Para nuestra solución, se implementó un *DirectLine* concreto con las operaciones antes mencionadas, dejando abierta la posibilidad de que en un futuro se puedan agregar nuevos tipos de *Lines* concretos, sin necesidad de realizar modificaciones en los módulos ya existentes.

Criterios extensibles

Otra decisión de diseño importante tomada con el fin de favorecer la **modificabilidad**, y en particular para permitir que el algoritmo de asignación de cupos sea modificable permitiendo en un futuro agregar nuevos cupos, fue la de tener una clase criterio abstracta (*abstract-criteria*) e implementar criterios concretos que extiendan de ella que para el caso de esta solución son: *etary-criteria* y *priority-criteria.*

Para eso, al llegar una nueva reserva al sistema, se utilizan los criterios de cada uno de los periodos de los vacunatorios en el departamento y zona escogida, y se busca asignar el cupo a la persona que reserva. Lo que se realiza es traer todos los vacunatorios que tienen lugar disponible en la ubicación elegida y en la fecha solicitada. Una vez que se tienen estos vacunatorios, lo que se hace es iterar sobre los periodos de vacunación de cada uno de ellos. Cada uno de estos periodos tiene un criterio, el cual se toma y por medio de la aplicación de la aplicación de la táctica ***Defer Binding*** se genera una nueva instancia de clase la cual extiende de un criterio abstracto, que tiene dos operaciones: *isPersonCompriseInCriteria(reserve, parameters)* que permite saber si una persona está habilitada para ese periodo de vacunación de acuerdo a lo establecido en el criterio, y *selectReserveToConfirm(reserves)* que dada una lista de reservas las ordena a fin de saber cual debe tener mayor prioridad a la hora de asignar nuevos cupos (esto se explicará luego con mayor detalle).

Entonces, haciendo uso de la operación *isPersonCompriseInCriteria* se determina para cada periodo de vacunación si la persona está habilitada o no para vacunarse. En el caso de que la persona esté habilitada, se confirma la reserva en el vacunatorio, disminuyendo la cantidad de lugares disponibles en el mismo para el periodo en cuestión. En caso de que la persona no esté habilitada para ninguno de los periodos, la reserva queda pendiente a la espera de que se agreguen nuevos cupos para poder confirmarla.

Entonces, de la forma en que se encuentran diseñados los criterios, si en un futuro se desean agregar nuevos se podrá hacer fácilmente sin necesidad de modificar los módulos existentes, ya que simplemente se debe crear una nueva clase para el criterio que extienda del criterio abstracto y que implemente la dos operaciones antes mencionadas. Un aspecto importante a destacar es que el nombre de esta clase debe ser: “nombre_del_criterio-criteria”, donde nombre-del-criterio es el valor introducido al ingresar un nuevo periodo de vacunación, ya que a partir del mismo se genera en tiempo de ejecución una instancia del criterio concreto de acuerdo a lo que se indica en el campo mencionado.

Manejo de errores

Utilizamos la táctica de manejo de errores para favorecer la **disponibilidad** en todo el sistema. Los errores son “catcheados” en todas las capas y finalmente manejados a nivel de Api el cual diferencia entre el tipo de errores para finalmente enviar la respuesta al usuario (Error 400, 404, 500, etc).

Para diferenciar estos errores utilizamos el atributo “type” del objeto Error y dependiendo de éste se envía un Status Code diferente al usuario.

Ejemplo:

```
} catch (ex) {
  if (ex.type === "validation") {
    res.status(400);
    res.body = ex.toString();
  } else if (ex.type === "not-found") {
    res.status(404);
    res.body = ex.toString();
  } else {
    res.status(500);
    res.body = {
      message: `Ha ocurrido un error interno dentro del sistema.
                Por favor intentelo mas tarde.`,
    };
    logger.log(ex);
  }
```

Un aspecto que sería bueno mejorar en un futuro sería que los errores no dependieran de un string en particular sino utilizar Custom Errors para manejar los mismos.

Creación de Tools

Es una ventaja tener la existencia de este módulo ya que evita la duplicación de código favoreciendo la **modificabilidad**. Cuando hablamos de duplicación, hablamos del tipo de duplicación conocido cómo **“real”**, donde el código no sólo es idéntico, sino que además sirve para lo mismo. Sucediendo entonces que si se produce un cambio, requiere que todos los sitios donde esté ese código se modifiquen de la misma manera, por tanto tiene una única razón para cambiar.

3.1.2 Vista de Uso

En esta sección veremos cómo se describen las dependencias de usos entre los módulos del sistema.

3.1.2.1 Representación primaria

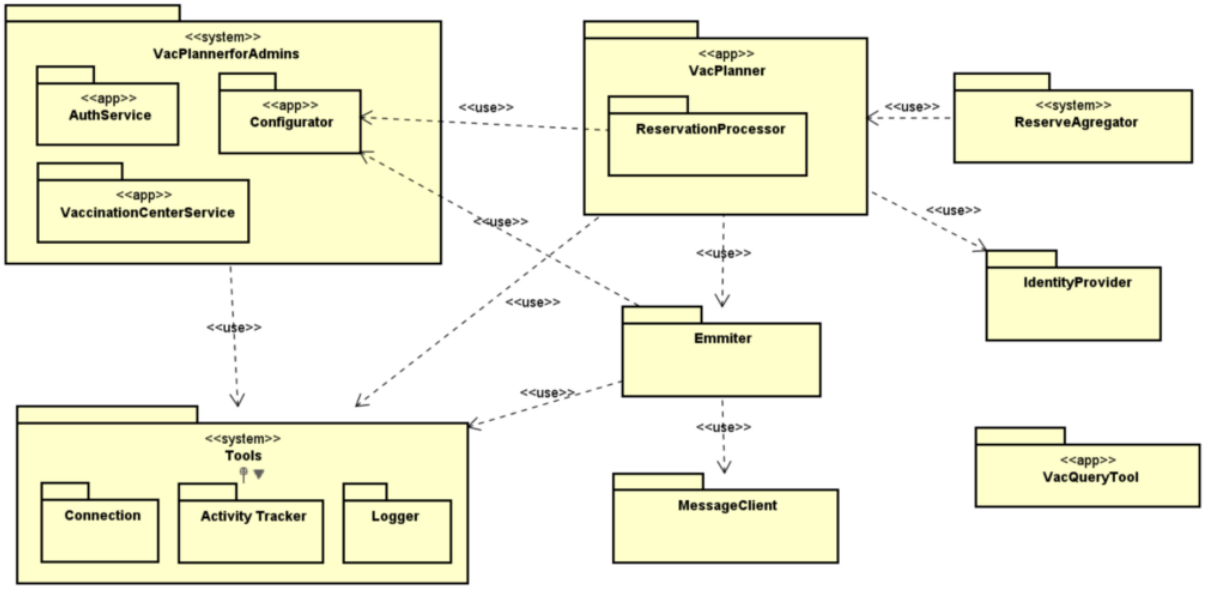

Figura 2. Diagrama de uso de la solución.

Dentro de esta vista encontramos que en el sistema que se ha creado se pueden observar dos diagramas de uso. Primero aquel en que los módulos son encargados de proveer interfaces (VacPlanner, VacPlannerForAdmins y VacQueryTool) estando compuestos principalmente por los submódulos controllers, services y repositories. Tal cómo se puede observar en la siguiente figura.

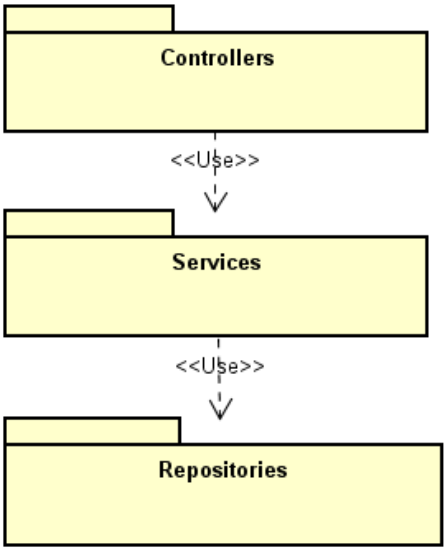

Figura 3. Diagrama de uso con submódulos controllers, services y repositories.

Por otro lado, contamos con la otra vista mencionada (figura 4), que al no proveer una interfaz no cuenta con el submódulo controller, componiendose únicamente por los submódulos services y repositories, como es el caso del módulo Emitter.

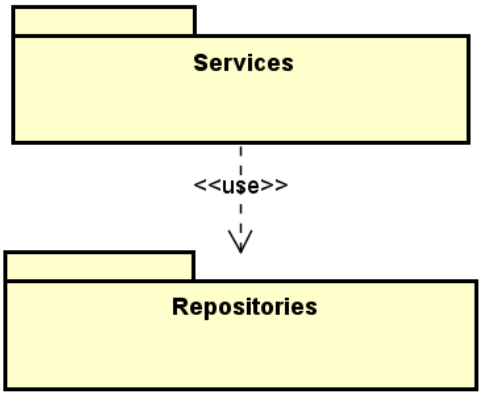

Figura 4: Diagrama de uso con submódulos services y repositories

3.1.2.2 Decisiones de diseño

Una decisión de diseño que se consideró para favorecer la **modificabilidad**, y contemplar la posibilidad de poder cambiar las herramientas o librerías que almacenan la información generada por los errores y las fallas, es hacer uso de la táctica ***Introducir Intermediario***, permitiendo que ante un cambio en la librería de terceros utilizada, los módulos que la utilizan no se vean afectados debido a que no se encuentran dependiendo de ella, sino que dependen de un intermediario como se puede observar la dependencia hacia el módulo *Tools* en la figura 2.

En esta figura también se puede observar cómo estos módulos fueron separados teniendo en cuenta la categoría de Modificabilidad ***Reducir acoplamiento***. Ya que reduciendo el acoplamiento entre dos módulos, se disminuye el costo esperado de cualquier modificación que afecte al que depende del otro.

Por otro lado, observando la figura 3 se puede apreciar la aplicación de la táctica de ***Uso de intermediarios***, además de lo ya explicado, se observa claramente que en el módulo de servicios, el mismo tiene la responsabilidad de realizar lógica sobre los datos, aunque en ocasiones, cuando dicha lógica no es requerida, tiene la función de “pasamanos” entre el controller y el repositorio, dejando así la posibilidad de que si en un futuro se desea agregar lógica, se puede realizar sin problemas actuando de tal forma como un intermediario.

También otra de las tácticas que se aplicó dentro de la categoría Modificabilidad fue la de ***Restringir dependencias***, la cual será explicada en la siguiente sección “Vista de Layers”, ya que queda más claro cuando se ven los módulos separados en capas.

Cabe destacar que se decidió separar de esta forma para ***Incrementar la coherencia semántica*** ya que aunque el módulo de servicios, se podría encargar también de acceder a los recursos del sistema, entendemos que son responsabilidades separadas.

3.1.3 Vista de Layers

3.1.3.1 Representación primaria

Principalmente podemos visualizar dos vistas de layers diferentes, dependiendo de la responsabilidad del módulo. Todos los módulos del sistema exceptuando Configurator, AuthService y VaccinationCenterService (módulos de administradores y vacunadores) tendrán el siguiente diagrama:

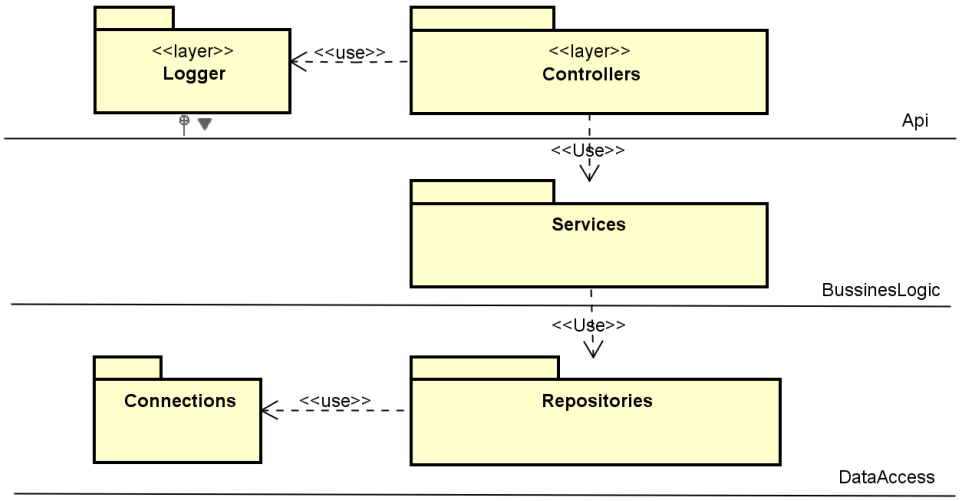

Figura 5: Diagrama de Layers.

Por otro lado, para los módulos de administradores y vacunadores previamente mencionados se tendrá la siguiente estructura.

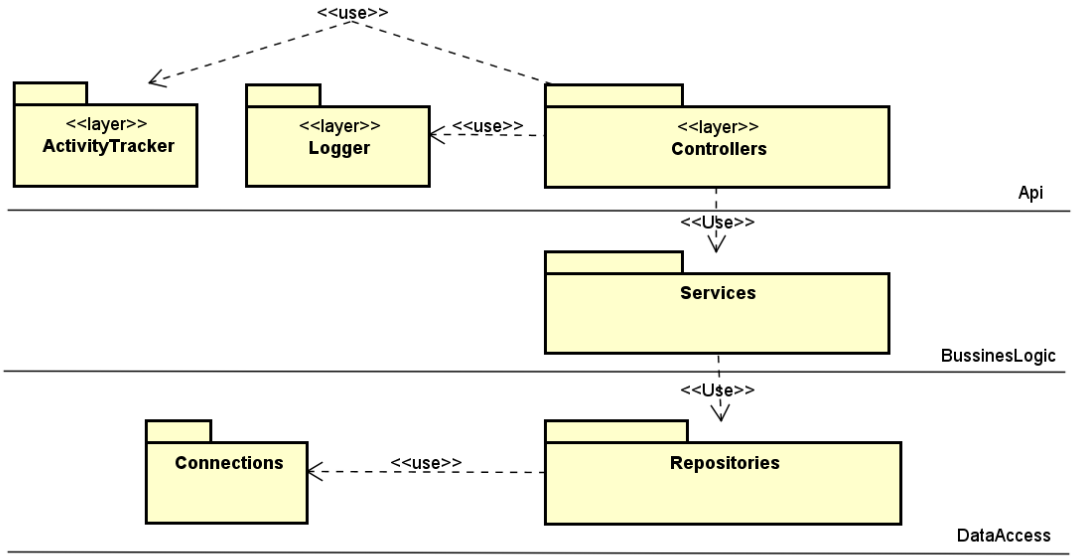

Figura 6: Diagrama de Layers.

3.1.3.2 Decisiones de diseño

Con el fin de poder desarrollar y evolucionar las partes del sistema de manera independiente, se debe dividir al mismo para que los módulos se desarrollen y evolucionen por separado, aportando así portabilidad, modificabilidad y reutilización. Para dar solución a esto, se aplicó el ***Layered Pattern*****,** permitiendo dividir al software en unidades llamadas capas, en donde cada una de ellas es una agrupación de módulos que ofrece un conjunto cohesivo de servicios.

Mediante el uso de este patrón, se permite organizar nuestros módulos y poder definir mejor las responsabilidades, ***incrementando la coherencia semántica*** debido a que como cada capa trabaja por objetivos comunes, se disminuyen los impactos de cambios.

Además, tal como se observa en la figura, cada capa usa los módulos que se encuentren en la capa adyacente inmediatamente inferior, o los que se encuentran en su misma capa, por lo que estaríamos aplicando la técnica de ***Restringir dependencias***, ya que por lo expresado, se restringen las comunicaciones y caminos de comunicación, disminuyendo dependencias y efectos secundarios en cambios.

- Los módulos pertenecientes a la capa de **Api**, se encargaran de interactuar con un agente externo. Estos serán accedidos mediante una request, donde extraerán la información almacenada en dicha request, y se la otorgaran al servicio para que la procese y de esa forma generar una response.

- En siguiente lugar los módulos dentro de la capa de **BusinessLogic** se encargarán de procesar información y realizar la lógica de las funcionalidades. Usualmente, deberán requerir de ayuda de los repositorios para interactuar con los datos almacenados requeridos para procesar la información de la forma solicitada.

- En tercer lugar, los módulos de **DataAccess** serán los encargados de extraer, almacenar o actualizar información a la que el sistema debe acceder de forma continua. Esta interacción puede referirse según nuestro criterio, tanto a bases de datos, como a colas, como en archivos JSON. En esta capa, se presenta además el servicio llamado *Connections* que maneja todo lo relativo a la conexión con la base de datos, creación de esquemas, entre otros. Debido a que *Connections* es usado en todos los módulos *repositories* del sistema, decidimos aplicar la táctica ***Abstraer servicios comunes***, implementado este servicio en un solo lugar, minimizando el impacto del cambio.

- Finalmente tenemos dos **servicios** de utilidad los cuales se encuentran en la capa de **Api.**
  En primer lugar el Logger, en el cual decidimos que solo la capa más alta se encargue de loggear información mediante el manejo de errores. Por otro lado, contamos con el servicio de *ActivityTracker* para los usuarios administradores los cuales necesitan tener un registro de actividad sobre las acciones que toman sobre el sistema, estos también se registran únicamente en la capa Api.
  Debido a que *ActivityTracker* y *Logger* son utilizados en todos los módulos del sistema, se aplicó con el fin de favorecer la **modificabilidad**, la táctica ***Abstraer servicios comunes***, implementando estos servicios una única vez en un solo lugar, permitiendo que si en el futuro algo de estos módulos cambia, las modificaciones se concentren en un solo lugar.

3.1.4 Comportamiento

A continuación mostraremos los diagramas de flujo más relevantes de nuestro sistema.

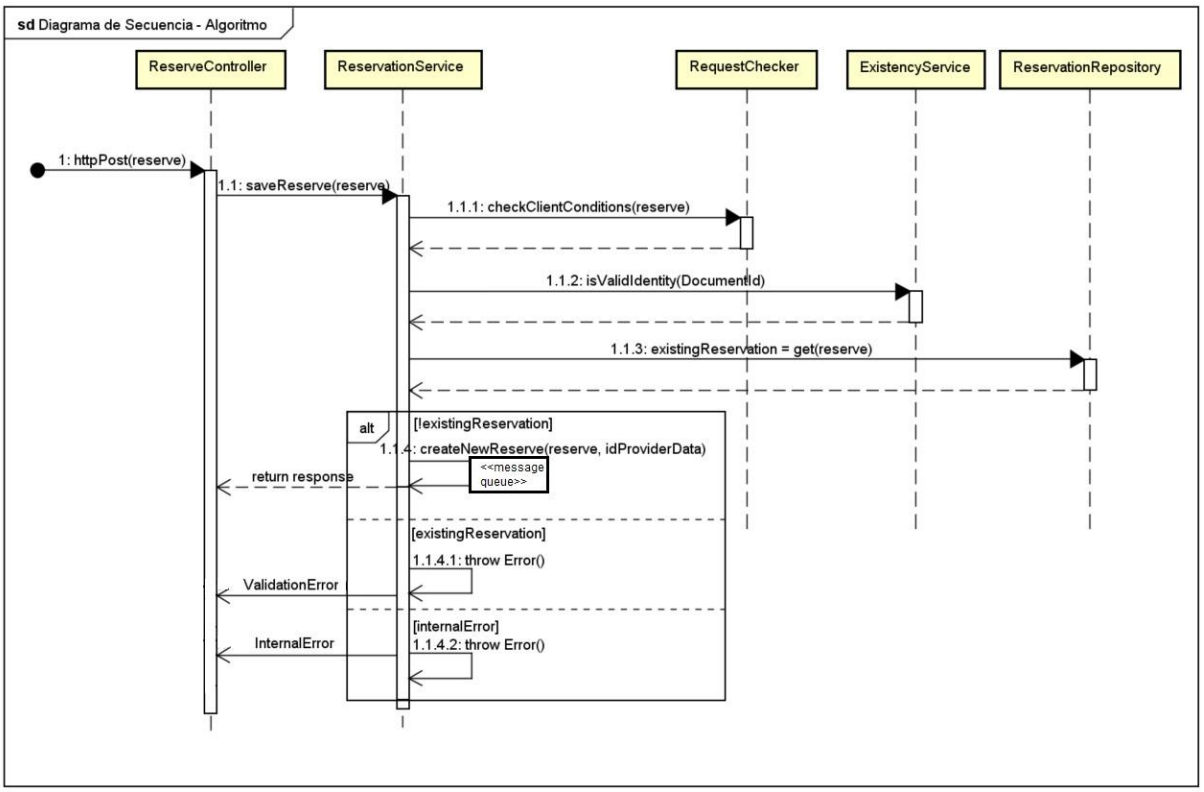

Figura 7: Diagrama de Secuencia, creación de una reserva.

Mecanismo

- Mediante un POST HTTP llega la reserva a ReserveController (el envío masivo de reservas se realizará mediante el emulador ReserveAggregator).
- Una vez las reservas son recibidas por el controlador, las mismas son traspasadas a ReservationService, el cual se encarga de verificar si cumple con las condiciones establecidas por medio del RequestChecker (encargado de transmitir la reserva por los diferentes filtros que realizan las validaciones de la misma con los parámetros correspondientes los cuales son seteadas *únicamente* al inicio de la ejecución del sistema leyendolas de un archivo de configuración, el cual es modificable mediante el subsistema configurador presente en VacPlannerForAdmins).
    - En caso de que los campos ingresados no sean correctos, se lanza un error que impide continuar con la ejecución, pero dicho caso se omite en el diagrama para no complejizar el mismo y poder mostrar los aspectos más relevantes del caso de uso.
    - En caso de que sean correctos, se continúa con la ejecución.
- El siguiente paso es validar si el documento proporcionado es válido, es decir, sí es una persona existente en los datos provistos por el emulador de identificación civil, el proceso continúa.
    - En caso de que el documento ingresado no pertenezca a nadie dentro del proveedor de identificación civil, se lanza un error que impide continuar con la ejecución.
- De manera que si la identidad brindada es válida se continúa verificando si dicho documento no tiene una reserva existente dentro del sistema.
    - En caso de que tenga una reserva previa dentro del sistema, se lanza un error indicando que el usuario ya posee una reserva previa.

- Si no existe una reserva previa, se procede a la creación de la misma, donde una vez confirmada, de forma inmediata es enviada al usuario a través de la API externa de envío masivo de SMS, además de ser enviada a una cola de mensajes para que se vaya procesando de forma asincrónica para almacenarse en la base de datos.
- Mientras que si a la hora de realizar el guardado ocurre un error interno también se lanza un error (diferente tipo al de cuando la reserva existe), tal cómo se puede observar en el diagrama en el fragmento ALT que se ve en la parte inferior.

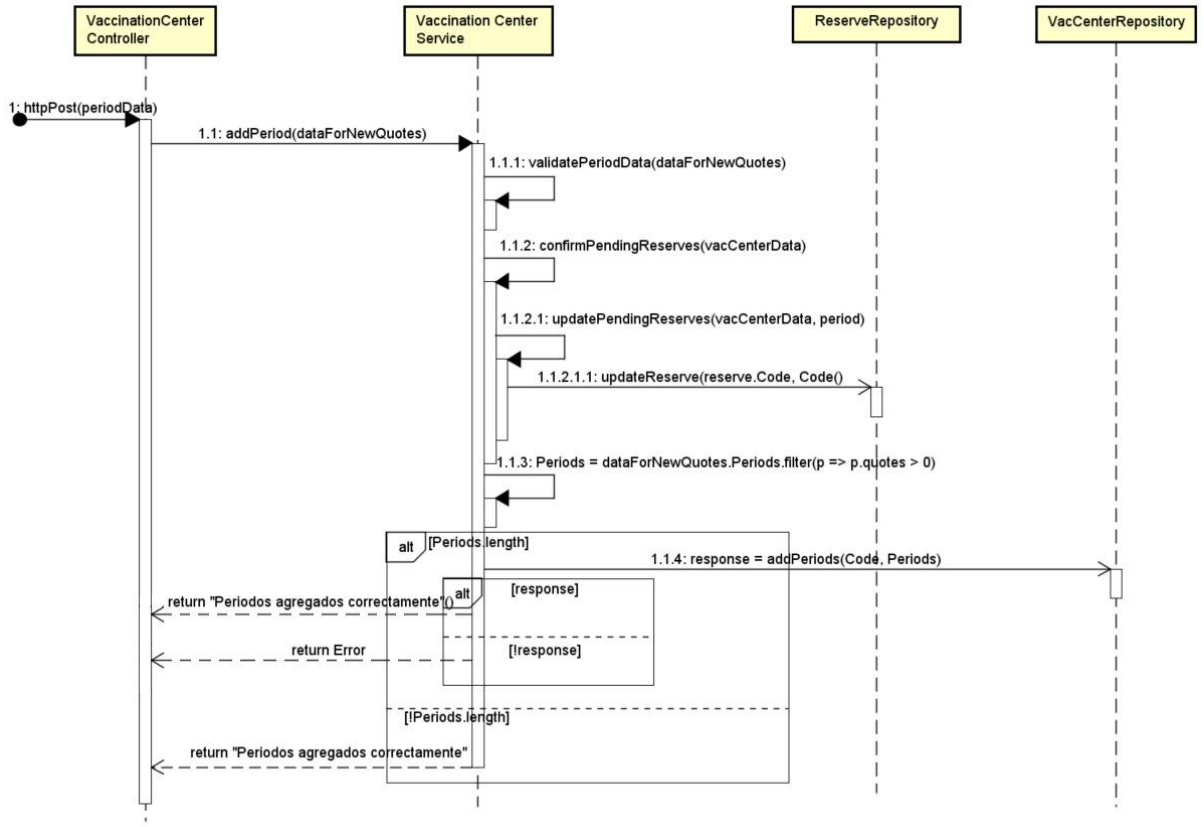

Figura 8: Diagrama de Secuencia, agregar un nuevo periodo.

Mecanismo

- El caso de uso comienza con una solicitud HTTP POST con la data necesaria para agregar un periodo de vacunación.
- Esta información es recibida por el *VaccinationCenterController*, quien por medio de la invocación al método *addPeriod* de *VaccinationCenterService* delega el procesamiento de la *request* a este último.
- En primer lugar, lo que se hace es validar que los datos ingresados sean correctos por medio del método *validatePeriodData*, esto es, que el código de vacunatorio corresponda a un vacunatorio válido para el departamento y zona ingresada.
    - En caso de que los campos ingresados no sean correctos, se lanza un error que impide continuar con la ejecución, pero dicho caso se omite en el diagrama para no complejizar el mismo y poder mostrar los aspectos más relevantes del caso de uso.
    - En el caso de que los campos ingresados sean correctos, se continúa con la ejecución invocando al método *confirmPendingReserves* el cual itera sobre cada uno de los periodos que se desean agregar e invoca para cada uno de ellos al método *updatePendingReserves.* Este método lo que hace es en primer lugar traer de la base de datos todas aquellas reservas que se encuentren en estado pendiente con el

departamento, zona y fecha que se indican en el periodo a agregar. Luego de esto, se toma el criterio del periodo de vacunación a agregar, y por medio de la aplicación de la táctica ***Defer Binding*** se genera una nueva instancia de clase la cual extiende de un criterio abstracto, que tiene dos operaciones: *isPersonCompriseInCriteria(reserve, parameters)*, y *selectReserveToConfirm(reserves)* que dada una lista de reservas las ordena a fin de saber cual debe tener mayor prioridad a la hora de asignar nuevos cupos. Entonces, haciendo uso de la operación *selectReservesToConfirm se ordenan las reservas pendientes de acuerdo al criterio, y luego se seleccionan las primeras n de ellas, donde n es el número de cupos habilitados para el periodo a crear.*

- *A continuación, por medio de la función updateReserve se actualizan cada una de estas reservas pasando a estado confirmado y se le asigna a cada una de ellas el vacunatorio para el cual se está agregando el periodo.*
- Una vez hecho eso, lo que se realiza es filtrar a aquellos períodos cuya cantidad de cupos sea mayor a cero (porque es posible que al agregar n cupos la totalidad de los mismos se hayan asignado únicamente a reservas previas pendientes).
- Finalmente, aquellos períodos cuya cantidad de cupos es mayor a cero, se agregan a la base de datos por medio de la función *addPeriods*, para que los mismos puedan ser asignados próximamente cuando arriben nuevas reservas.

3.2 Vistas de Componentes y conectores

En esta sección se describe como el sistema está estructurado en un conjunto de elementos que están en tiempo de ejecución así como la interacción entre estos. Estos elementos se definen como componentes mientras que las interacciones son los conectores. Los conectores son relaciones las cuales deben denotarse agregando un estereotipo que las identifique.

3.2.1 Diagrama de componentes y conectores

3.2.1.1 Representación primaria

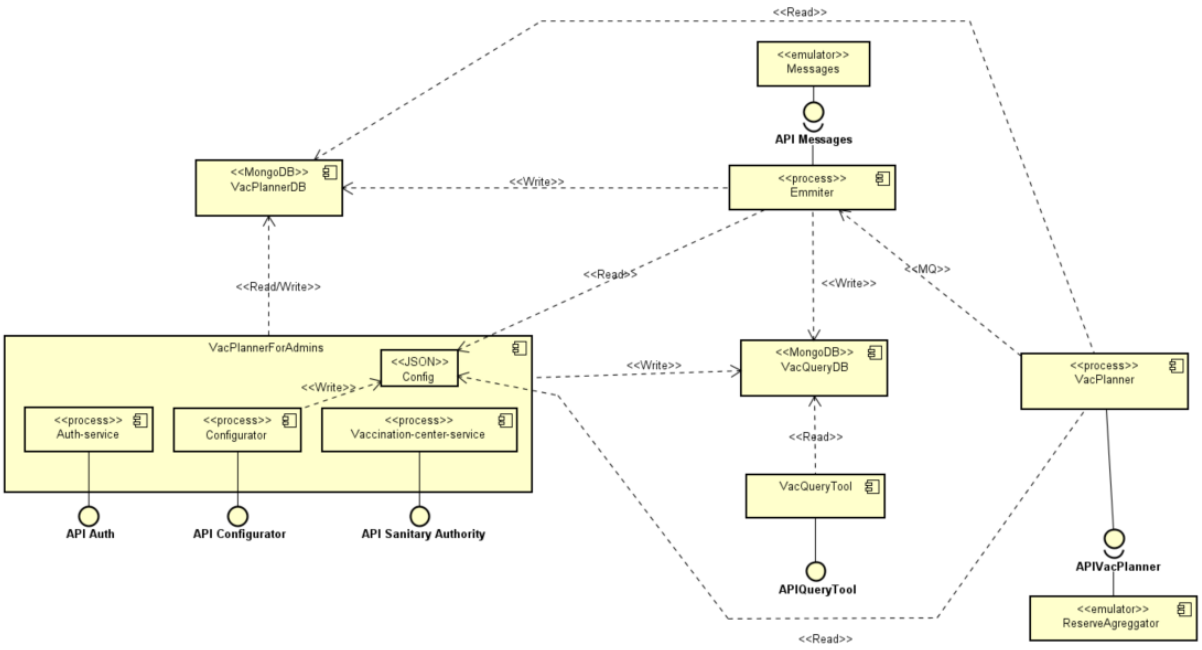

Figura 9: Diagrama de componentes.

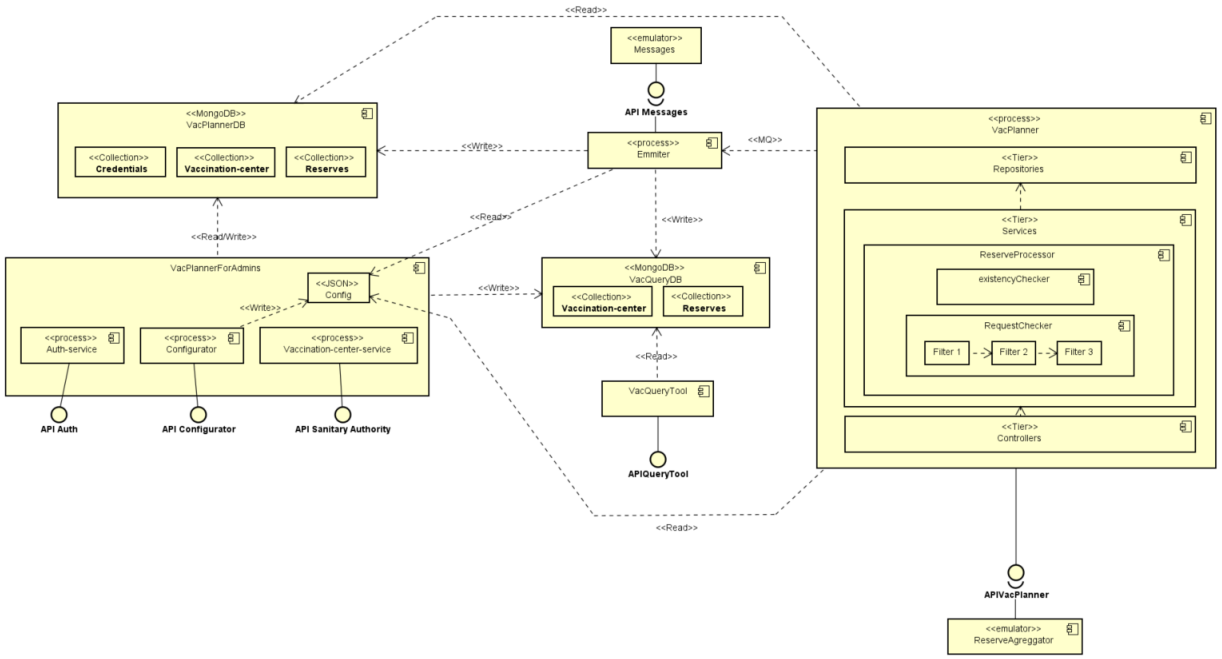

Figura 10: Diagrama de componentes detallando componentes importantes.

3.2.1.2 Catálogo de elementos

Componente/conector	Tipo	Descripción
VacPlanner	Componente	Se encarga de recibir las reservas por la interfaz expuesta API VacPlanner, una vez procesadas, las valida y verifica la no existencia de una reserva para la persona. Una vez validada la envía a la message queue EmmiterMQ para su próximo procesamiento en Emmiter.
VacQueryTool	Componente	Expone una interfaz APIQueryTool la cual recibe consultas, estas son procesadas y posteriormente solicitadas a la base de datos.
Emmiter	Componente	Este componente se encarga de leer los diferentes trabajos que tiene que procesar guardados en la message queue EmmiterMQ, y distribuirlos a los diferentes servicios que la necesitan. En este caso, a la base de datos y a los clientes externos. Los clientes externos a los cuales emite, son leídos desde el archivo de configuración almacenado en VacPlannerForAdmins (debido a que la configuración de los mismos se realiza en el componente configurador).
Auth-service	Componente	Expone una interfaz API Auth en la cual los usuarios se “loguean” al sistema, así obteniendo un token con diferentes permisos para realizar acciones sobre Configurator y Vaccination-center-service.
Configurator	Componente	Expone una interfaz API Configurator la cual requiere autenticación por parte del usuario administrador que va a hacer uso de ella. En este componente los administradores podrán modificar tanto el archivo de configuración de los parámetros del sistema como las API´s externas.
VaccinationCenterService	Componente	Expone una interfaz API Sanitary Authority la cual requiere autenticación por parte del usuario administrador y el vacunador que va a hacer uso de ella. Este componente es el encargado de realizar el mantenimiento del sistema, realizando acciones como agregar vacunatorios, agregar periodos, crear actas de vacunación, etc.
VacPlannerDB	Componente	Repositorio de datos el cual almacena todas las colecciones del sistema.
VacQueryDB	Componente	Repositorio de datos secundario el cual almacena únicamente algunas colecciones del repositorio principal. (las necesarias para poder satisfacer las consultas requeridas).

Read/Write	Conector	Muestra que dicho componente tiene la capacidad de escribir y leer sobre algún repositorio del sistema.
MQ	Conector	Muestra que el mecanismo de comunicación entre los componentes son colas de mensajes, las cuales fueron implementadas usando Bull
Interfaces	Conector	Muestra que se expone una API REST como medio de comunicación.

3.2.1.3 Interfaces

Nuestro subsistema VacPlanner, provee una interfaz, la cual contiene tres endpoints, que se presentan a continuacion.

Recurso	Verbo	Responsabilidades
/reserves	POST	Permite crear una reserva a un cliente con los datos brindados en VacPlanner.
/reserves/{documentId}	GET	Permite obtener una reserva existente para el documento brindado por path dentro del sistema de VacPlanner.
/reserves/?documentId=XXXXXX&code=YYYYYY	DELETE	Permite eliminar una reserva para el documento de identidad y código de reserva provistos.

Por otro lado, el subsistema *VacPlannerForAdmins*, provee tres interfaces, donde la primera de ellas es *vaccination-center-service* y cuyos endpoints se describen a continuación.

Recurso	Verbo	Responsabilidades
/vaccinationCenter	POST	Permite crear un nuevo vacunatorio con los datos brindados.
/vaccinationPeriod	POST	Permite agregar período/s de vacunación al centro de vacunación que se indica en la solicitud.
/vaccinationAct	POST	Permite registrar un acto vacunal para los datos brindados en la solicitud.
/vaccinationCenter/vaccines	GET	Permite dado un vacunatorio y una fecha, obtener los datos acerca de la cantidad de vacunas dadas hasta el momento y la cantidad de vacunas remanentes.

/vaccinationCenter/pendingReserves	GET	Permite obtener la cantidad de reservas pendientes de asignar por departamento y zona.

***Nota**: Para hacer uso de los endpoints de VacPlannerForAdmins que se indican arriba, es necesario proveer en la request un token de autorización.*

VacPlannerForAdmins provee además una interfaz configurator, cuyos endpoints se describen a continuación.

Recurso	Verbo	Responsabilidades
/clients	POST	Permite agregar una nueva API externa para ser usada en el sistema.
/clients	PUT	Permite modificar una API externa existente dentro del sistema para ser usada por el mismo.
/params	POST	Permite configurar las validaciones sobre los campos de la reserva.

Finalmente, VacPlannerForAdmins provee una interfaz auth-service que permite a los usuarios administradores y vacunadores autenticarse.

Recurso	Verbo	Responsabilidades
/login	POST	Permite a los usuarios administradores y vacunadores autenticarse en el sistema.

Continuando, dentro de nuestra solución encontramos el subsistema VacQueryTool.

Recurso	Verbo	Responsabilidades
/vaccines/applied	GET	Permite obtener la cantidad de vacunas aplicadas, dado un rango de fechas y de edades, listadas por departamento y zona. Permite, dado un rango de fechas, obtener la cantidad de vacunas aplicadas por departamento y horario.
/reserves/pending	GET	Permite listar la cantidad de reservas pendientes por Departamento.

Para el subsistema de VacQueryTool tuvimos que tomar una decisión importante a la hora de implementar la interfaz, dicha decisión estuvo vinculada a la cantidad de endpoints.
Esto se debe a que inicialmente pensamos en implementar 3 endpoints, una para cada query, pero luego de cierto análisis, pensamos que esto no sería ideal ya qué tendríamos 3 HTTP GET con parámetros similares (y en caso de tener exactamente los mismos parámetros en el futuro agregando una nueva query, no sería posible implementarlo). Cabe aclarar que para las consultas solicitadas dos de ellas son las que se encuentran relacionadas (las que se encuentran en el primer endpoint).

Debido a esto, nos propusimos encontrar una alternativa, en dónde encontramos un patrón asociado a las consultas que se realizan. Notamos que dos de las consultas eran en un periodo de tiempo, es decir una fecha inicial y final, además de que la segunda de ellas agregaba una cierta condición (rango de edad), donde como resultado se obtienen las vacunas aplicadas agrupadas según un criterio. Es por esto, que llegamos a la conclusión de que lo más eficiente y reutilizable, es decir la API que mejor se adecua a la situación, sería tener un único endpoint para aquellas consultas que reciban por query params un rango de fechas y deban devolver como resultado la cantidad de vacunas aplicadas según un criterio. Es por esto, que pudimos realizar un único endpoint para las dos consultas existentes, ya que la variación que tienen dentro de los parámetros recibidos es el rango de edad, de tal manera que para la primera de las consultas (“Dado un rango de fechas. Listar por departamentos y por horario la cantidad de vacunas aplicadas) se verifica que los únicos parámetros sean el rango de fechas, ya que si viniera un rango de edades entraría en la segundo caso.

Para que el sistema *VacPlanner* funcione, requiere de una interfaz externa de identificación civil que se utiliza para saber si la cédula asociada a la reserva ingresada corresponde a una persona real.

Recurso	Verbo	Responsabilidades
/population/documentId	GET	Permite obtener los datos de la persona (en caso de que exista), brindando su documento de identidad.

Modelos:

- *ReserveModel*:
  - **Code**: Código único que identifica a la reserva (este valor se autogenera y no debe pasarse en la request de creación).
  - **DocumentId**: Documento de identidad de la persona que realiza la reserva.
  - **Cellphone**: Celular de la persona que realiza la reserva.
  - **ReservationDate**: Fecha de reserva para la vacunación.
  - **Schedule**: Horario escogido para vacunarse.
  - **State**: Departamento donde desea vacunarse.
  - **Zone**: Zona del departamento en donde desea vacunarse.
  - **DateOfBirth**: Fecha de nacimiento de la persona que realiza la reserva. Este dato se obtiene por medio de la API externa de identificación civil.
  - **Priority**: Prioridad (1-4) de la persona que realiza la reserva. Este dato se obtiene por medio de la API externa de identificación civil.
  - **Confirmed**: Es un valor booleano que toma *true* si la reserva fue confirmada, y *false* en otro caso.
  - **VaccinationCode**: Es el código del vacunatorio asociado a la reserva de la persona (en el caso de que la misma haya sido confirmada).
- *UserModel*:
  - **WorkerNumber**: Número que identifica al administrador o vacunador.
  - **Password**: Contraseña de la cuenta del administrador o vacunador.
  - **Permissions**: Permisos con los que cuenta el administrador o vacunador.
- *VaccinationAct*:
  - **VaccinationCode**: Código único que identifica al vacunatorio al cual se va a asociar el acto vacunal.
  - **DocumentId**: Documento de identidad de la persona a la cual se le registrará el acto vacunación.
  - **VaccinationDate**: Fecha en la cual se registra el acto vacunal (vacunación a una persona).
- *VaccinationCenter:*
  - **State**: Departamento en donde se encuentra el vacunatorio a crear.
  - **Zone**: Zona del departamento en donde se encuentra el vacunatorio a crear.
  - **Code**: Código único que identifica al vacunatorio a crear.
  - **Name**: Nombre del vacunatorio a crear.
  - **Schedule**: Horario en el que vacunará el vacunatorio a crear.
  - **Periods**: Es un array que contiene los diferentes periodos de vacunación que se abren para el vacunatorio que se está creando. Cada uno de ellos contiene: dateFrom (fecha de comienzo), dateTo (fecha de finalización), un criterio que contiene: nombre y los parámetros propios del criterio, y por último, un atributo quotes para indicar la cantidad de cupos que se abren para cada periodo.

3.2.1.4 Comportamiento

A continuación mostraremos los diagramas de flujo más relevantes de nuestro sistema, pero enfatizando en los componentes y sus relaciones.

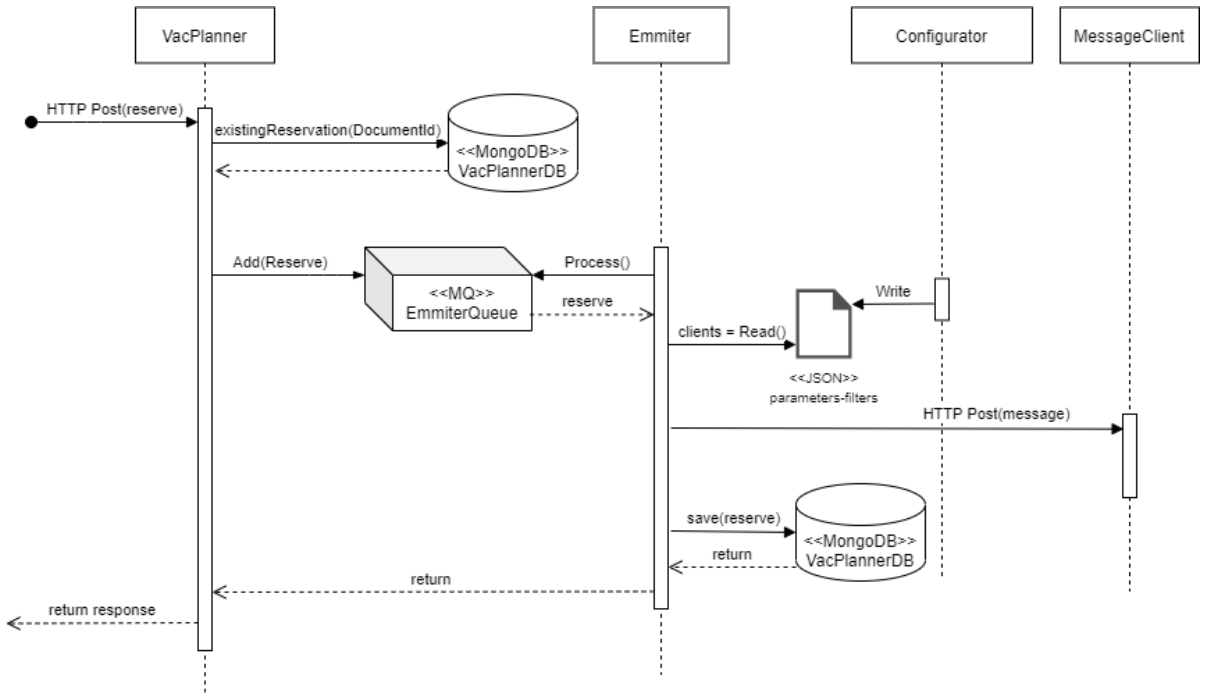

Figura 11: Diagrama de secuencia, creación de una reserva a nivel de componentes.

El comportamiento es el mismo que el descrito en la Figura 7, simplemente lo que se trata de mostrar aquí es la interacción a nivel de componentes.

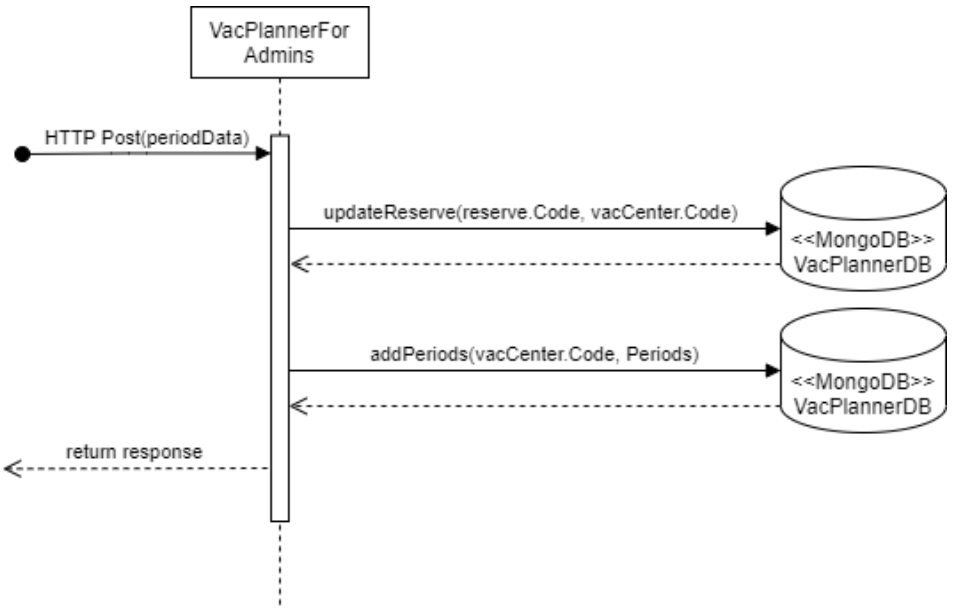

Figura 12: Diagrama de secuencia, agregar un nuevo periodo a nivel de componentes.

El comportamiento es el mismo que el descrito en la Figura 8, simplemente lo que se trata de mostrar aquí es la interacción a nivel de componentes.

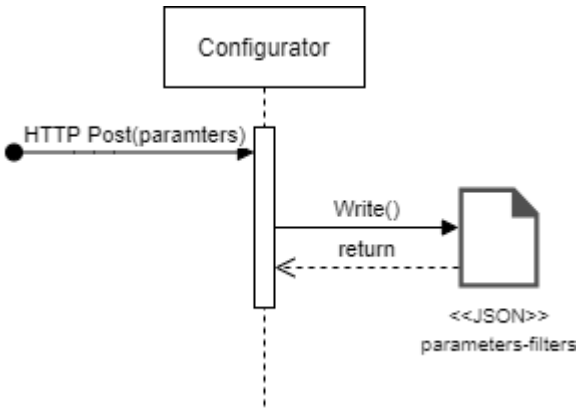

Figura 12: Diagrama de secuencia, actualizar parámetros del sistema a nivel de componentes.

Mecanismo

- El caso de uso comienza cuando se reciben los parámetros de configuración mediante una solicitud HTTP Post.
- Una vez que se reciben los parámetros configuración, los mismos son validados verificando que tenga todas las propiedades necesarias. Dónde las validaciones que se realizan, es que para cada uno la propiedad exista y no sea vacía, además de por ejemplo, verificar que el largo mínimo pasado para la cédula sea mayor a cero, y el máximo sea mayor o igual al mínimo.
- Una vez completada esta validación los datos son guardados en un archivo sobre el mismo módulo dentro del paquete config.
- De una forma similar sucede con la configuración de las APIS externas, donde una vez recibido los datos, tanto para guardar uno nuevo como para actualizar uno existente, lo que se hace es verificar la existencia de dicho cliente, produciéndose dos casos en ambas operaciones. En la operación de agregado de un nuevo cliente, sucede que si el mismo no existe, se guarda dichos datos en un json dentro de la carpeta config dentro del módulo, mientras que sí se desea agregar un cliente ya existente en el sistema, se muestra un error indicando que el cliente que desea agregar ya se encuentra registrado.
- Por otro lado, con la actualización sucede algo similar, siendo que si uno quiere actualizar un cliente el mismo tiene que existir, por lo cual sí existe, se sobrescribe el archivo modificando los datos de dicho cliente, mientras que si no existe se lanza un error indicando que el cliente a actualizar no existe.

3.2.1.5 Relación con elementos lógicos

Componente	Paquetes
VacPlanner	*VacPlanner*
Emmiter	*Emmiter*
Auth-service	*VacPlannerForAdmins*
Vaccination-center-service	*VacPlannerForAdmins*
Configurator	*VacPlannerForAdmins*
VacQueryTool	*VacQueryTool*

3.2.1.6 Decisiones de diseño

Autenticación

Con respecto al requerimiento 12 (Protección de datos y accesos al sistema) el cual está asociado a el atributo de calidad **seguridad**, decidimos resolver este requerimiento basándonos en la categoría **resistir ataques**, táctica ***autenticar actores*** utilizando JWT.
Utilizamos Json Web Tokens debido a que pueden ser utilizados para propagar la identidad de usuarios como parte del proceso de autenticación entre un proveedor de identidad (en nuestro caso la base de datos Credentials) y un proveedor de servicio (en nuestro caso las operaciones las cuales requieren uso por personal autorizado). Para esto, se genera un par clave pública y privada utilizando el algoritmo RS256, esta clave privada es instalada en el módulo de autenticación y nunca es compartida, ya que es utilizada para firmar. Mientras que la clave pública es compartida con los módulos que utilizan el mecanismo de autenticación (VaccinationCenterService y Configurator) para autenticar la solicitud recibida, y determinar si tiene permiso o no.

Separación entre VacPlanner y Emitter

Otra decisión que influyó positivamente en la **performance** fue la decisión de realizar la separación de los componentes VacPlanner y Emitter. Es decir separar lo que es la distribución a los clientes y escritura de datos en MongoDB, del componente que recibe las reservas y las valida (VacPlanner).

Con esta separación obtuvimos las siguientes ventajas:

- Responder al usuario rápidamente antes de persistir la reserva en la base de datos y enviar el SMS de confirmación de la misma. Esto es un aspecto fundamental, debido a que si esperamos a que la reserva se persista y se envíe la confirmación de la misma por SMS (lo que implica una llamada HTTP a una API externa) para recién en ese momento dar respuesta al cliente, los tiempos de respuesta aumentan sustancialmente.

- Poder escalar cada proceso de forma independiente.

Otra de los atributos de calidad en el cual influyó positivamente fue en la **disponibilidad**. Sucede que la indisponibilidad del sistema de continuar recibiendo request puede deberse a que un recurso está fuera de línea o hay una falla en algún componente, es por esto que si todo estuviera agrupado en un único componente y el mismo fallase encontraríamos un único punto de fallas en dónde el sistema dejaría de poder recibir las reservas, validarlas y almacenarlas, por lo cual una vez que dicho componente falla, dentro del sistema no se podría realizar ninguna operación más. Es por esto, que una solución que encontramos, es realizar una ***separación de responsabilidades*** entre componentes, donde VacPlanner justamente se encargaría de recibir las reservas y validarlas, para luego enviarlas por medio de una message queue al componente Emitter, el cual se encargaría de hacer la distribución a los diferentes clientes y la escritura en la base de datos, esto claramente favorece la **disponibilidad** ya que si alguno de los componentes tuvieran una falla, las reservas que han sido procesadas y enviadas hacia el emitter no se perderían, ya que las mismas se encontraría disponibles entre la cola intermedia entre VacPlanner y Emitter (es decir reservas procesadas y validadas, las cuales deben ser almacenadas en la base de datos).

Manejo de error en Emitter. Decidimos que si falla algo en emitter, al ya haberle respondido previamente al usuario, la solución que encontramos fue enviarle un SMS con el siguiente mensaje *“Ha ocurrido un error al confirmar su reserva, por favor intente agendarse nuevamente”*. Esta decisión fue tomada teniendo en cuenta que este componente tendrá fallos mínimos, por lo tanto, no habría problema de que algunas solicitudes de reserva se reintenten en caso de error.

Separación VacPlanner de VacPlannerForAdmins

Otra decisión importante para mejorar aún más la **performance** del proceso de reservas fue independizar VacPlanner de las operaciones de Administración. Finalmente, se creó un nuevo componente llamado VaccinationCenterService (dentro de VacPlannerForAdmins) el cual realiza todas las operaciones de mantenimiento de vacunatorios en el sistema.

Con esto se hace más eficiente el replicamiento del componente VacPlanner utilizando la táctica ***mantener múltiples copias de computo***, es decir poder replicar cada componente por separado a demanda, en otras palabras, si vemos que algún componente se encuentra con una mayor demanda de carga a procesar, podemos replicarlo sin tener que replicar a los demás innecesariamente

Otro motivo que nos llevó a separarlo es que las operaciones de mantenimiento requieren seguridad, por lo tanto nos pareció conveniente agruparlo con otros componentes que necesitaban autenticación como es Configurator.

Separación Bases de Datos de escritura y consultas.

Uno de los requerimientos no funcionales solicitados es el de la independencia de consultas de VacQueryTool y escrituras por parte de *VacPlanner*, con el fin de que los componentes que implementan operaciones de ingreso de datos puedan gestionarse y desplegarse de forma independiente de los componentes que implementan operaciones de consulta de datos. Este punto se cumple con tan solo tener el servicio que ingresa reservas en un componente (*VacPlanner*) separado del servicio que hace consultas sobre las reservas (*VacQueryTool*). Pero esto, no garantiza lo requerido en cuanto a **performance** tanto en el ingreso de reservas desde el componente *VacPlanner,* como también en las consultas de VacQueryTool donde se busca optimizar la latencia.

Decimos que esto no garantiza la performance, porque si bien son dos componentes independientes, están impactando sobre la misma base de datos compitiendo las escrituras por un lado y las lecturas por otro.

Por esa razón, nos planteamos diferentes opciones para resolver este problema y poder garantizar la **performance**, entre las que se encuentran: aplicar *sharding* para realizar una partición horizontal de la base de datos, limitar la cantidad de *jobs que pueda procesar el emitter para que la* base de datos no sea vea tan sobrecargada, aplicar el patrón **CQRS**, o simplemente tener copias de las tablas necesarias para realizar consultas en otra base de datos diferente. Analizando todas las opciones, y por cuestiones de tiempo y complejidad, nos decantamos por la última de las opciones nombradas, es decir, al momento de realizar una escritura, la misma se hace tanto en la base de datos de VacPlanner llamada VacPlannerDB como también en la de VacQuery llamada VacQueryDB, para que al momento de realizar alguna consulta compleja en este último, la respuesta sea rápida y no se vea comprometida por los frecuentes ingresos de reservas debido a que son dos bases de datos totalmente independientes.

Un aspecto que notamos es que debido a que al momento de guardar reservas estamos escribiendo en dos bases de datos diferentes, podría ocurrir la situación de que la inserción en una de las bases se produzca de manera satisfactoria, mientras que en la otra no, generando una desincronización entre ambas bases de datos. En ese momento, ante la ocurrencia de un error en una de las inserciones, lo que habría que hacer es un *rollback* deshaciendo los cambios realizados, aspecto que no fue realizado por temas de tiempo y complejidad de la implementación, pero que sería bueno poder implementarlo en futuras versiones de la plataforma.

Configurator parte de VacPlannerForAdmins.

Uno de los aspectos que el sistema debe proveer es la posibilidad de poder configurar la mayor cantidad de aspectos del país y el plan de vacunación. Es por esto que una manera posible de realizarlo es exponiendo una API, que permita configurar gran parte de los parámetros definidos en el plan de vacunación, minimizando la codificación de nuevos componentes. Esto tal como se mencionó antes, se encuentra dentro del componente Configurator, el cual se encarga de configurar los tipos de campos de la reserva y las validaciones que la plataforma debe realizar al recibir una solicitud de reserva, además de permitir agregar o modificar las APIS externas minimizando el costo de codificación.
Entonces a la hora de construir dicha API, nos cuestionamos si debería estar disponible para todo público o para administradores del sistema. Como las configuraciones a realizar son sobre datos que utiliza el sistema para las APIS externas, así cómo también los campos de validaciones de las reservas, la decisión que se tomó es que fueran únicamente para administradores, es por esto que tuvimos que diseñar una capa de seguridad sobre la cual apoyarnos y elegir de las diversas formas de autenticación una de ellas.

¿Cuáles son los métodos que se pueden utilizar para conectarse a la API?
En la actualidad, principalmente se pueden diferenciar tres tipos de autenticación, cada método nos ofrece distintas opciones de flexibilidad de uso, así como también nos permiten definir el grado de autorización que tendrá el cliente para acceder a los datos.
Estos son: API Keys, JSON Web Tokens y OAuth. En particular nos enfocamos en la utilización de JWT debido a que ya lo habíamos utilizado para las operaciones referentes a los administradores por lo cual para seguir con la misma idea se continuó utilizando el mismo. La ventaja de su utilización frente a los demás métodos son dos:

1. JWT se puede usar en más situaciones más allá de la autenticación ya que es un token que nos permite cualquier tipo de información que no sea sensible, por lo cual se puede combinar con otros métodos de autenticación.

2. Permite mantener un token que se puede enviar por la red sin ocupar mucho espacio.

Única lectura del archivo de configuración.
Para mejorar la **performance** de los componentes de VacPlanner decidimos leer una única vez el archivo de configuración al levantar este servidor y guardarlo localmente. De esta forma cada vez que necesita la configuración accede directamente sin tener que volver a leer el archivo.

Esto trajo consigo un problema menor en el cual debemos bajar y subir el servicio al realizar cambios en configuración y clientes, ya que no desarrollamos un mecanismo de sincronización en tiempo de ejecución, pero esto no genera un problema mayor ya que los cambios en los archivos de configuración no son muy frecuentes a la hora de realizar una ejecución habitual del sistema.

Defer binding - config
Una táctica de modificabilidad que usamos es la de Defer Binding, particularmente en tiempo de inicialización. Esto lo conseguimos mediante el uso del archivo de configuración que contiene los valores que son propensos al cambio. Los distintos atributos son accedidos desde el resto de los módulos requiriendo la librería config, lo que implica que en lugar de tener los valores fijos en el código, al inicializar el sistema se cambian los valores del archivo por los deseados y el resto de los módulos se sincronizan con dicha configuración.

Manejo de carga

Para probar la arquitectura del sistema, realizamos un load test, saturando al mismo con una gran cantidad de reservas y evaluando cómo responde a este tipo de situaciones. Para eso, se probó el sistema enviando un total de 150.000 requests, enviando 1.000 de ellas por segundo, y se obtuvo respuesta para las *150.000 en un total de 9 minutos,* lo que hace en promedio un tiempo de respuesta igual a *3.6 ms*, el cual es un tiempo de respuesta que consideramos bueno dada la gran cantidad de reservas recibidas por segundo.

Al mismo tiempo en que se estaban ingresando las reservas, se probaron hacer consultas complejas mediante *VacQueryTool*, obteniendo en el peor de los casos una latencia igual a 0.7 segundos, lo cual es aceptable considerando que se esperaba que en promedio las consultas tengan una latencia menor a 2 segundos.

3.3 Vistas de Asignación

En esta sección se describen las vistas de asignación para representar relaciones entre las unidades de software y los elementos del entorno tales como, hardware, sistemas de archivos o la organización de los equipos de desarrollo de software.

3.3.1 Vista de Despliegue

3.3.1.1 Representación primaria

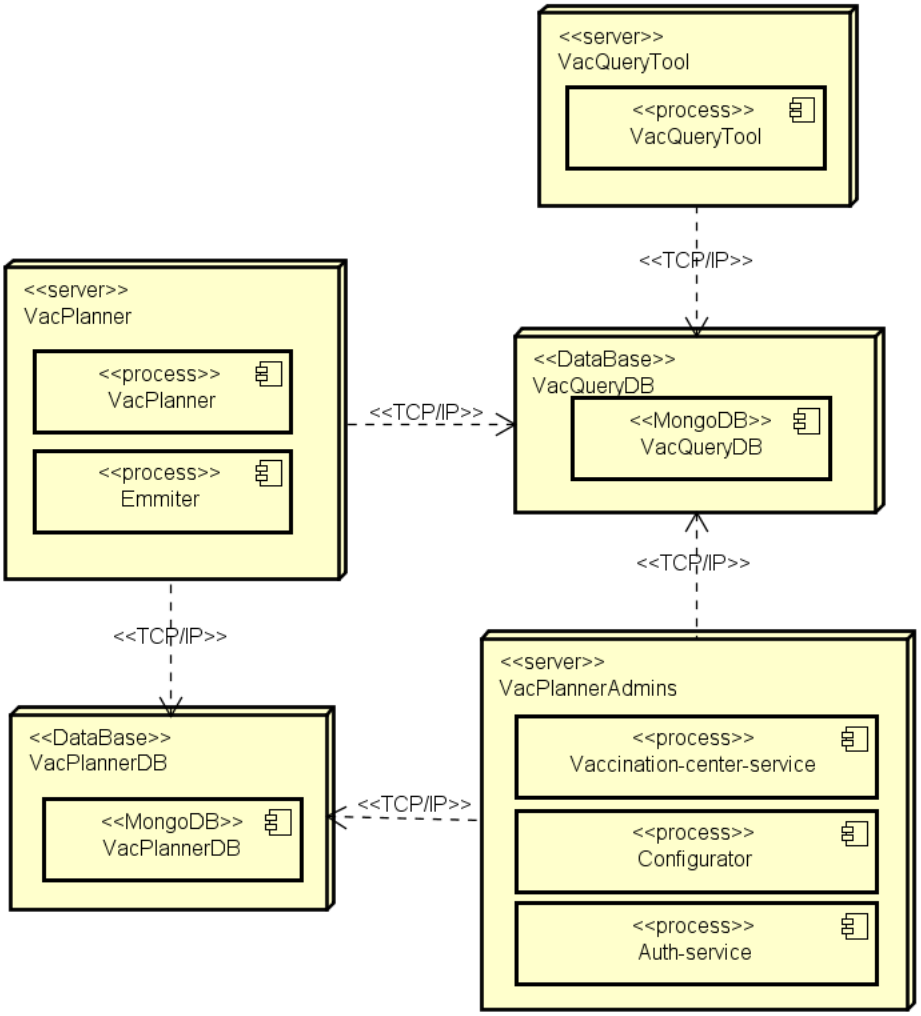

Figura 13: Diagrama de despliegue.

3.3.1.2 Decisiones de Diseño

Independencia VacPlanner y Emitter
Como se puede observar en este diagrama decidimos desplegar en un nodo aparte a VacPlanner y Emmiter. Esta decisión fue tomada teniendo en cuenta que estos componentes se encargan de procesar las reservas entrantes y tienen una alta demanda de recursos por lo tanto aislarlos en un módulo aparte nos permitirá utilizar *Clustering (pudiendo ser por medio de la herramienta PM2, la cual a la hora de realizar la ejecución es utilizada permitiendo la replicación)* y así mejorar su performance.

Independencia VacQueryTool
En este diagrama también se puede observar la distribución del servicio VacQueryTool en otro Nodo y por ende el componente del mismo se podría ejecutar en otro ambiente de ejecución. Esto favorece a la seguridad del sistema ya que al realizar esta división entramos dentro de la táctica *Separate Entities* dentro de la categoría resistir ataques del atributo de calidad seguridad.

Uso de MongoDB
La decisión de almacenar los datos del sistema en una base de datos no relacional como MongoDB trajo las siguientes ventajas:

- Documento modificable, esta fue una de las principales ventajas por las cuales elegimos este modelo. Nos permite almacenar diferentes estructuras de datos en una misma colección como en el caso de los criterios utilizados.

- Fácil y eficientes consultas sobre documentos JSON, esto se debe a que MongoDB es una base de datos orientada a documentos BSON, el cual es un formato bastante similar a JSON.

- Se potenció la perfomance de las consultas utilizando la función aggregate ofrecida por esta base.

- Con respecto a la disponibilidad podemos decir que MongoDB la favorece ya que a partir de la versión 4.0, posee transacciones ACID.

- En un futuro, en el que se desee potenciar la escalabilidad, MongoDB permite escalar de forma horizontal usando sharding.

4. Anexo

4.1 Justificación de las dependencias a librerías

- **fs**: librería utilizada para leer la clave pública y la clave privada que se encuentran almacenadas dentro de archivos, así como también escribir los diferentes archivos de configuración de clientes externos y parámetros de las validaciones para las reservas.

- **express**: Express es el framework web más popular de Node, y es la librería subyacente para un gran número de otros frameworks web de Node populares. Proporciona mecanismos de escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas). Se utiliza generalmente en los ruteadores, justamente para rutear a los controladores correspondientes.

- **sequelize**: es una poderosa librería en Javascript que facilita la administración de una base de datos SQL. Sequelize es un mapeador relacional de objetos, lo que significa que mapea la sintaxis de un objeto en los esquemas de nuestra base de datos, utilizando la sintaxis de objetos de NodeJs y Javascript para realizar el mapeo. Se utiliza para el emulador de reservas (por defecto se tienen los distintos dataset almacenados en diferentes tablas en una base de datos en MySql), así como el almacenamiento de datos para el proveedor de identificación civil (se encuentran todos las identidades de los usuarios que tienen datos válidos). También se utiliza para almacenar cada una de las pistas del auditory tail (es decir almacena cada uno de los “movimientos” de los usuarios que requieren autenticación).

- **config**: Esto le da a la aplicación una interfaz de configuración consistente compartida entre una lista creciente de módulos npm que también usan node config.

- **cors**: Es un paquete node.js para proporcionar un middleware Connect / Express que se puede utilizar para habilitar CORS con varias opciones.

- **axios**: Axios es un cliente HTTP basado en promesas para NodeJs y el navegador. Cómo es basado en promesas, nos permite código asincrónico para realizar solicitudes muy fácilmente.

- **bottleneck**: es un programador de tareas y limitador de velocidad de request, liviano y sin dependencias para NodeJs y él navegador.

- **moment**: Se utilizó esta librería para manejar los diferentes formatos de fechas que se piden a lo largo del sistema. Moment es una biblioteca de fechas de JavaScript liviana para analizar, validar, manipular y formatear fechas.

- **mongoose**: Mongoose es una herramienta de modelado de objetos de MongoDB diseñada para funcionar en un entorno asincrónico. Mongoose admite promesas y devoluciones de llamada. En este caso fue utilizada para el almacenamiento de las reservas así como también los vacunatorios con los respectivos períodos, además del registro del acto vacunal y las credenciales para los usuarios administradores y vacunadores.

- **sentry**: El SDK de Sentry permite la notificación automática de errores, excepciones y transacciones. Utilizándose para el logueo de los diferentes errores, brindando a los desarrolladores la capacidad de ver el código fuente de Node en cada marco y obtener un seguimiento de contexto adecuado al modelo de concurrencia de Node.

- **bull**: Bull es una biblioteca de Node que implementa un sistema de cola rápido y robusto basado en redis. En particular se utiliza para enviar las reservas ya procesadas y confirmadas al emitter quién se encarga de emitir los mensajes a los clientes así cómo guardar la reserva en la base de datos.

- **glob**: Un glob es una cadena de caracteres literales que se utilizan para hacer coincidir las rutas de archivo. Globbing es el acto de localizar archivos en un sistema de archivos utilizando uno o más globs.

- **jsonwebtoken**: utilizado para realizar la autenticación de VacPlannerForAdmins, para las operaciones las cuales requieren autenticación.