OBLIGATORIO ARQUITECTURA DE SOFTWARE

DESCRIPCIÓN DE LA ARQUITECTURA DE APPEV

APPEV

23 de Junio de 2022

Alejo Garat (219610)
Juan Pedro da Silva (229475)

Link Repositorio GitHub: [ORTArqSoft/219610_229475: Obligatorio Arquitectura de Software appEV (github.com)]

ÍNDICE

1. Introducción

El presente documento es una descripción de la arquitectura del sistema de votación electrónica. En la misma el lector podrá encontrar los diferentes requerimientos del sistema, la identificación de atributos de calidad y las diferentes descripciones de las vistas de módulos, componentes y conectores, y de asignación. Por otro lado, se detallan decisiones de diseño tomadas y descartadas que se tuvieron en cuenta para la versión actual del sistema y pensando en versiones futuras.

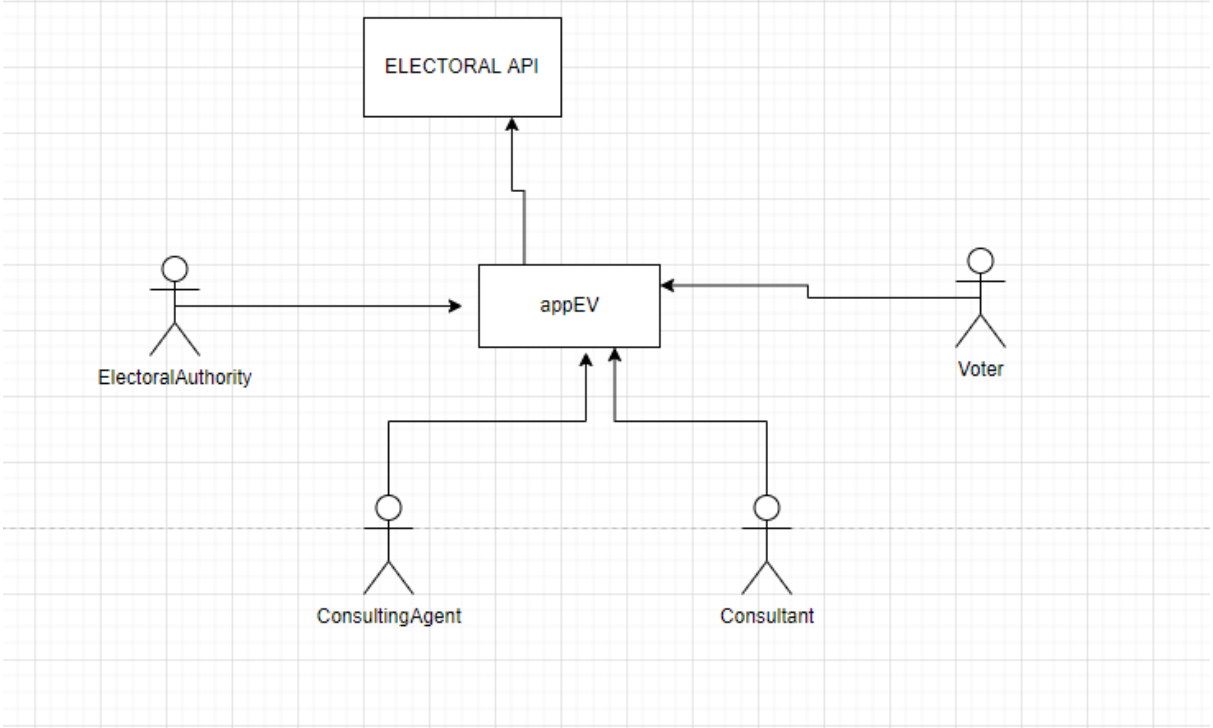
1.1 Propósito

El propósito del presente documento es proveer al arquitecto una especificación completa de la arquitectura del ***sistema de votación electrónica appEV.***

2. Antecedentes

2.1 Propósito del sistema

El sistema *appEV* es un sistema de votación electrónica donde existen cuatro tipos de usuarios y APIs externas, como se puede observar en el siguiente diagrama de contexto.



- **Votante**: usuario que emite votos y solicita constancias de los mismos.
- **Autoridad electoral**: usuario que realiza consultas.
- **Agente de consulta**: usuario que realiza consultas.
- **Consultor de la aplicación**: usuario que modifica las configuraciones del sistema y realiza consultas.
- **API Autoridad electoral**: API de la cual appEV obtiene los datos de una elección.

2.2 Requerimientos significativos de Arquitectura

2.2.1 Resumen de Requerimientos Funcionales

ID Requerimiento	Descripción	Actor
RF 1 Inicio de la elección	*Permite iniciar la elección para que los votantes queden habilitados a emitir votos*[1]	*Autoridad Electoral*
RF 2 Cierre de la elección	*Permite finalizar la elección*	*Autoridad Electoral*
RF 3 Emitir voto	*Permite a los votantes emitir votos*	*Votante*
RF 4 Constancia de voto	*Permite a los votantes solicitar una constancia de voto vía mail*	*Votante*
RF 5 Procesamiento de información y validaciones	*Permite a los consultores de la aplicación agregar, quitar o intercambiar filtros para validar peticiones que ingresan a la plataforma*	*Consultor*
RF 6 Configuración de alertas	*Permite a los consultores modificar valores esperados respecto a la cantidad de votos y cantidad de constancias*	*Consultor*

[1] Dependiendo de la modalidad de voto de la elección el mismo podrá ser único o no.

	emitidas y solicitadas respectivamente por un votante. Por otro lado permite modificar el conjunto de destinatarios a los que les debe llegar cualquier desvío de los valores esperados,	
RF 7 Consulta de voto	*Permite consultar las fechas y horas en las que un determinado votante votó*	*Autoridad Electoral*
RF 8 Consulta del resultado de la elección	*Permite consultar el resultado de la elección*	*Autoridad Electoral*
RF 9 Consulta de la configuración de la plataforma	*Permite consultar los valores esperados y conjunto de destinatarios de alertas (ver RF 6)*	*Autoridad Electoral, Consultor*
RF 10 Horarios más frecuentes de votación	*Permite consultar los horarios más frecuentes de votación*	*Autoridad Electoral, Consultor, Agentes de Consulta*
RF 11 Cobertura de votos de cada circuito	*Permite obtener para cada circuito de una elección la cantidad de votantes habilitados y votos según edad y sexo.*	*Autoridad Electoral, Consultor, Agentes de Consulta*
RF 12 Cobertura de votos de cada departamento	*Permite obtener para cada departamento de una elección la cantidad de votantes habilitados y votos según edad y sexo.*	*Autoridad Electoral, Consultor, Agentes de Consulta*

2.2.2 Resumen de Requerimientos de Atributos de Calidad

ID Requerimiento	ID Requerimiento de Calidad o restricción	Descripción
RF 2 Emitir voto	*AC 1 Performance*	Debido a la carga de datos es fundamental que el votante pueda recibir una pronta respuesta del procesamiento de su voto. Para ello se utilizaron gorutinas permitiendo enviarle una respuesta al usuario sin necesariamente agregar su voto a la base de datos, lo único que se realiza son las validaciones para efectivamente asegurarse que el voto puede entrar al sistema, todo lo demás se realiza luego de enviarle la respuesta al votante. Esto afecta a la pérdida de datos, esto se explica en el RNF 5 Manejo de carga.
RF 7 Consulta de voto *RF 8 Consulta de resultado de elección*	*AC 1 Performance*	Para favorecer la performance de estas consultas se utiliza una base de datos en memoria

RF 9 Consulta de la configuración de la plataforma *RF 10 Horarios más frecuentes de votación* *RF 11 Cobertura de votos de cada circuito* *RF 12 Cobertura de votos de cada departamento*		que provee los datos rápidamente así como la técnica CQRS creando modelos de lectura específicos para las mismas.
RF 4 Constancia de voto	*AC 1 Performance*	Para favorecer la performance una vez que se valida la información de voto lo primero que se hace es enviar la constancia para no tener que esperar a que se termine de procesar el registro del voto. También se creó un servicio único para el envío de la constancias por sms, separándolo del servicio general de comunicación.
RNF 1 Gestión de errores y fallas	*AC 2 Modificabilidad* *AC 4 Testeabilidad*	Debido a que en el futuro se quiere poder modificar de herramienta para recibir información de fallas en el sistema impactando lo menos posible en la solución, se implementó un servicio en el cual se desencola información de errores y se guardan en un .txt las diferentes fallas detectadas en los servicios del sistema. Dentro del servicio, se podrá cambiar de herramienta ya que aplicó el patrón de diseño Strategy. Por otro lado, al querer conocer rápidamente las fallas estamos testeando al sistema.
RNF 2 Cambio de proveedor de mensajería para enviar constancia de voto	*AC 2 Modificabilidad*	Se utilizó el patrón de diseño Strategy para poder cambiar fácilmente de proveedor de mensajería, para agregar un nuevo proveedor solamente se debería agregar el código del mismo teniendo en cuenta la interfaz provista.
RNF 3 Recepción de datos de elecciones de cualquier país (Asociado con RF 1)	*AC 2 Modificabilidad*	Para la recepción de datos se utilizó el patrón de diseño Adapter así permitiendo recibir datos de la elección de cualquier país solamente agregando como es el formato de los mismos junto con un conversor hacia el Adapter.
RNF 4 Protección de los datos	*AC 3 Seguridad*	Para proteger la confidencialidad se guardaron los datos sensibles (la contraseña) de los usuarios aplicándoles un hash, también para los votantes se aplicó la

		misma técnica modificando el id del votante y el id del candidato cuando se guarda el voto en las bases de datos. En cuanto a la integridad se decidió hacer múltiples bases de datos las cuales tienen accesos restringidos únicamente para los que los usuarios que tienen permisos, como por ejemplo a la base de datos de los votos solamente pueden acceder los votantes y la autoridad electoral.
RNF 5 Manejo de carga	*AC 1 Performance*	El sistema debe ser capaz de procesar la mayor cantidad de votos sin perder datos en el camino. Por ello fue imprescindible reintentar la información que no se pudo agregar por saturación del sistema mediante un cron[2].
RF 5 Procesamiento de información y validaciones	*AC 2 Modificabilidad*	Para la modificación de las validaciones que se realizan tanto en el inicio de la elección, cierre de la elección y votación se tomaron en cuenta dos grandes aspectos, primero para que los consultores puedan cambiar el orden de los filtros, dejar usar filtros y usar nuevos filtros que ya pertenecen al sistema decidimos guardar los filtros que se van a usar en una base de datos memoria leyéndolos cada vez que se van a ejecutar de esta forma cuando los consultores realicen las modificaciones desde la api estos filtros la próxima vez que sean ejecutados se van a ver modificados. En cuanto al segundo aspecto se utilizó el patrón Strategy para los filtros los cuales se ejecutan indiferentemente sin saber en particular cual se está ejecutando pero retornando el error específico del filtro si es necesario.

3. Documentación de la arquitectura

Como se mencionaba al comienzo del documento, el lector podrá encontrar en esta sección las diferentes vistas en las que se describen mediante diagramas y justificaciones las

[2] Golang Cron (linuxhint.com)

diferentes decisiones de diseño tomadas que ayudaron al equipo a satisfacer los atributos de calidad propuestos.

En primer lugar comenzamos con la vista de módulos, donde el objetivo es proveer un "plano" de cómo está construido el sistema y facilitar el análisis de impacto ante modificaciones.

Posteriormente seguimos con la vista de componentes y conectores cuyo propósito es mostrar al sistema en tiempo de ejecución.

Finalmente concluimos con la vista de asignación donde se muestra cómo están desplegados los diferentes componentes.

Cabe aclarar que todos los elementos pertenecientes al sistema fueron desarrollados con el lenguaje de programación Golang[3]. Como se va a poder apreciar, existen procesos que se pueden correr en diferentes computadoras haciendo posible el cambio de lenguaje. Sin embargo, existen decisiones y acciones ejecutadas en base a herramientas que el lenguaje provee, y por ende la modificabilidad se ve afectada en este sentido.

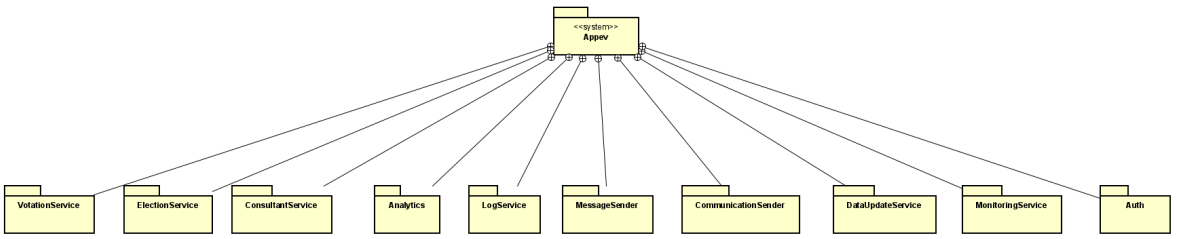
3.1 Vistas de Módulos

3.1.1 Vista de Descomposición

En la presente vista se describe la estructura jerárquica del sistema.
En la misma el lector podrá encontrarse con la representación primaria, el catálogo de elementos y diferentes decisiones de diseño tomadas.

3.1.1.1 Representación primaria

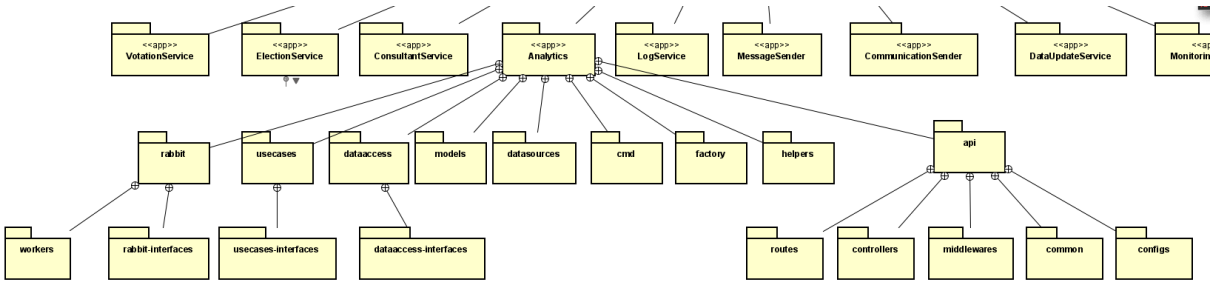
Esta representación primaria fue fragmentada para una correcta visualización por parte del lector. Nuestro sistema está formado por diez servicios[4] los cuales se pueden observar en la siguiente representación.



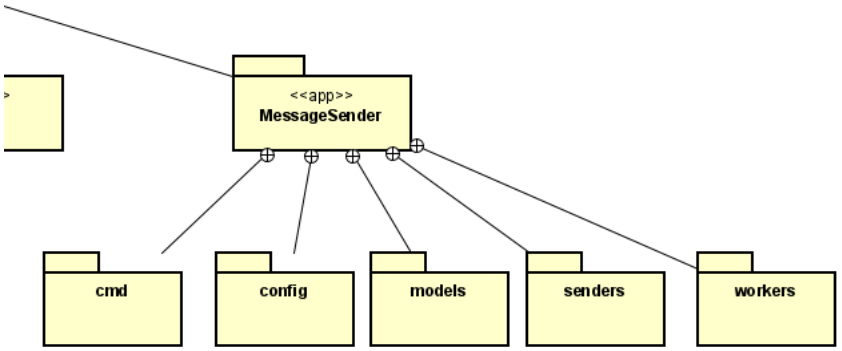
El sistema está formado por APIs y otros procesos, de los cuales algunos requieren acceso a datos y otros no. En cuanto a las APIs, la continuación de la representación primaria es similar puesto que comparten la mayor parte de directorios[5]. Mostraremos a modo de ejemplo la continuación de la representación primaria de *Analytics* (API) y *MessengerSender* (proceso sin acceso a datos).

---

[3] The Go Programming Language
[4] Ver con mayor detalle la vista de componentes y conectores.
[5] Si se visualizan las carpetas del proyecto, se puede encontrar por ejemplo que en *VotationService* se incluye por ejemplo el directorio *Messenger* por el eventual cambio del proveedor de mensajería para enviar la constancia de voto.



Continuación de representación primaria de Analytics

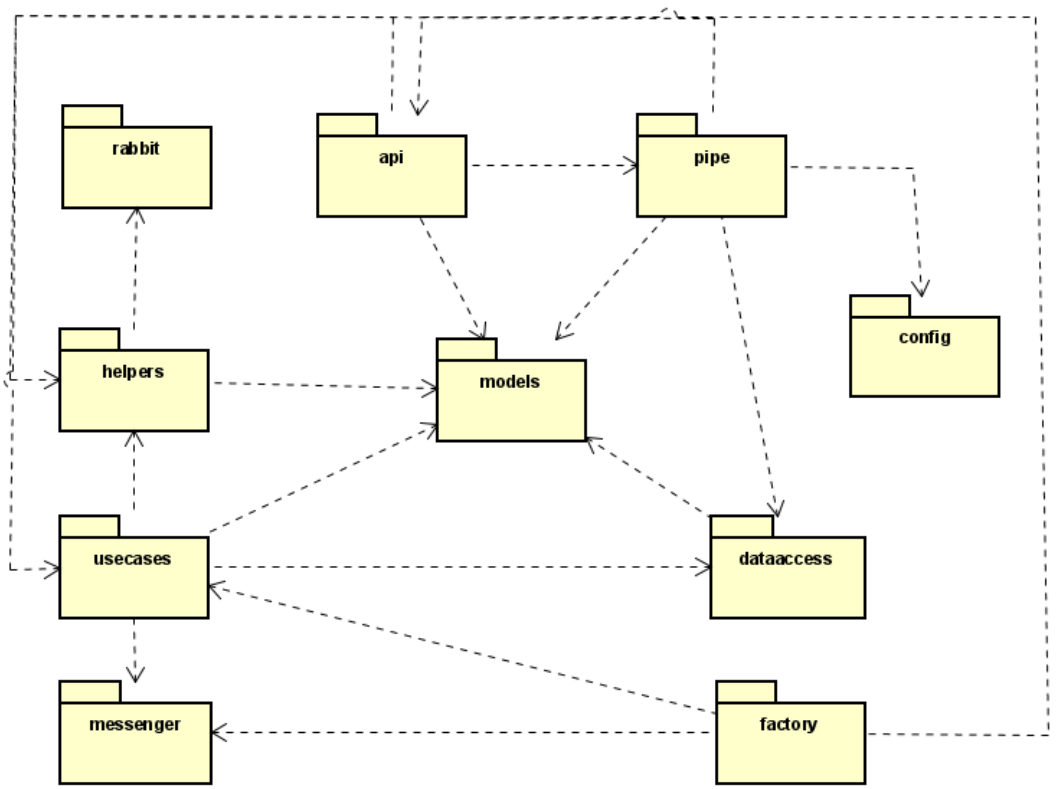


Continuación de representación primaria de MessageSender

3.1.2 Vista de Uso

En la presente vista el lector se encontrará con la descripción de las dependencias de usos entre los módulos del sistema. Se utilizará *VotationService* para detallar la representación primaria y vista de layers.

3.1.2.1 Representación primaria



Representación primaria del servicio VotationService

3.1.2.2 Decisiones de diseño

Se decidió utilizar el patrón *layered pattern*, es decir, dividimos cada servicio en unidades, donde cada una agrupa un conjunto de módulos que ofrecen un conjunto de servicios cohesivos de modo que los módulos puedan ser desarrollados y evolucionados de forma separada con la mínima interacción entre las partes. Las capas (unidades) fueron separadas por **responsabilidad**.

Hoy en día la solución utiliza *RabbitMQ* y es por eso que con el objetivo de impactar lo menos posible ante un cambio de proveedor se creó el paquete helpers el cual llama a las diferentes tipos de comunicación con rabbit (colas). Por lo tanto *usecases* no posee una dependencia directa con rabbit, reduciendo el acoplamiento mediante el uso de intermediarios.

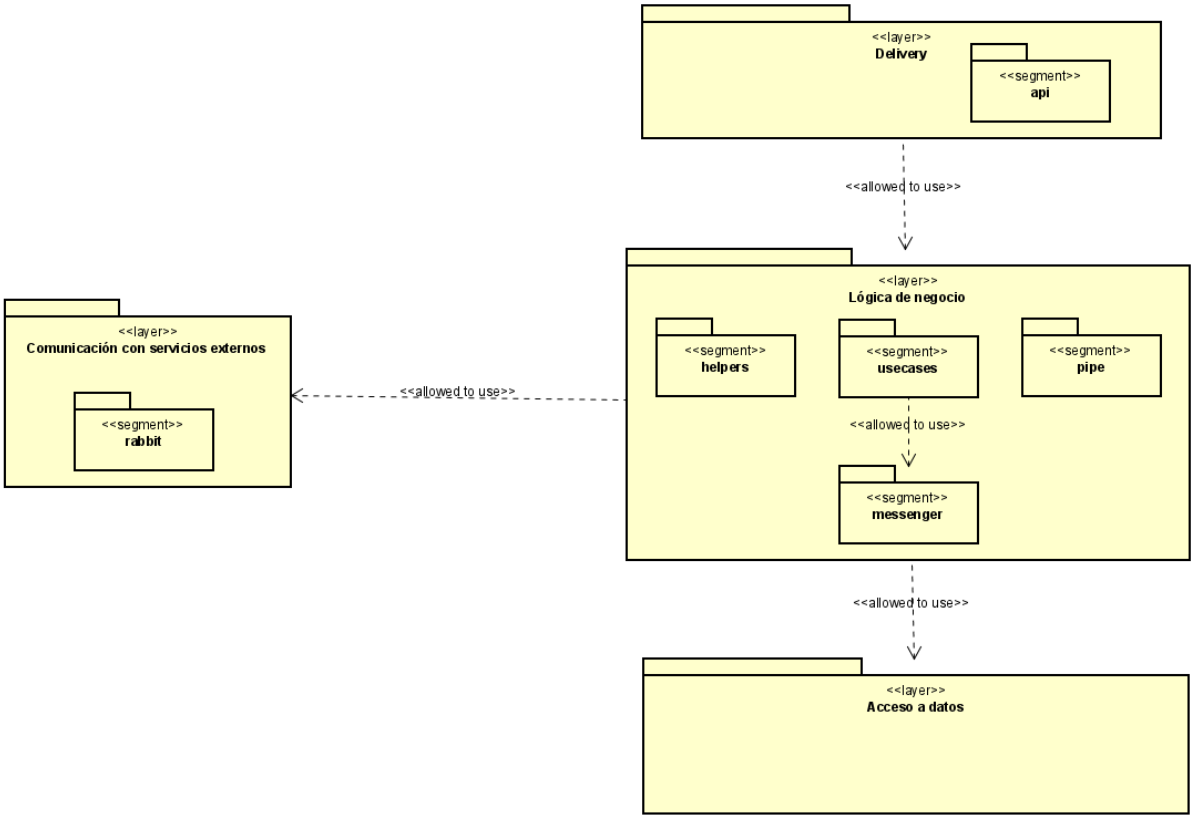
Se puede observar en gran cantidad de servicios el paquete *Rabbit*, la cual posee una interfaz llamada *rabbit-communication-interface*. Las clases que requieren la interfaz son las que se describen a continuación.

- *LogHelper*: encargada de llamar a la clase concreta de *Rabbit* que envía la información de los logs.
- *DataUpdateHelper*: responsable de enviar la información del voto para su posterior comunicación con el servicio encargado de actualizar datos en la base de datos electoral.
- *MessageSenderHelper*: encargada de llamar a la clase concreta de *Rabbit* que envía la constancia de voto a la cola de sms (proveedor de mensajería actual).

- *CommunicationSenderHelper*: encargada de llamar a la clase concreta de *Rabbit* que envía la constancia de voto a la cola de mails.

3.1.3 Vista de Layers

3.1.3.1 Representación primaria


<<layer>>
Delivery
<>
api
<<allowed to use>>
<<layer>>
Lógica de negocio
<>
helpers
<>
usecases
<>
pipe
<<allowed to use>>
<>
messenger
<<layer>>
Comunicación con servicios externos
<>
rabbit
<<allowed to use>>
<<allowed to use>>
<<layer>>
Acceso a datos

Vista de layers del servicio VotationService

3.1.3.2 Catálogo de elementos

Elemento	Responsabilidades
Delivery	Capa que se encarga de recibir la petición y enviarla a la lógica de negocio a realizar el procesamiento y validaciones pertinentes.
Comunicación con servicios externos	Capa destinada a reunir las clases necesarias para realizar llamadas a otros servicios externos.
Lógica de negocio	Capa que se encarga de recibir la información por parte de la capa Delivery y realizar con ella el procesamiento y las validaciones, para luego en caso de éxito comunicarse con la capa de acceso a datos.
Acceso a datos	Capa que almacena y consulta información en la base de datos.

3.1.3.3 Decisiones de diseño

Decidimos incluir la capa de *comunicación con servicios externos* ya que es importante que si el día de mañana se quiere cambiar o agregar servicios externos como lo son colas de mensajes, se pueda hacer impactando lo menos posible en la solución.

3.1.3.4 Otras vistas de módulos

Como parte del *RF 5 Procesamiento y validaciones*, se utilizó una interfaz de modo que la clase Pipeline posee una lista de filtros y los utiliza indistintamente.

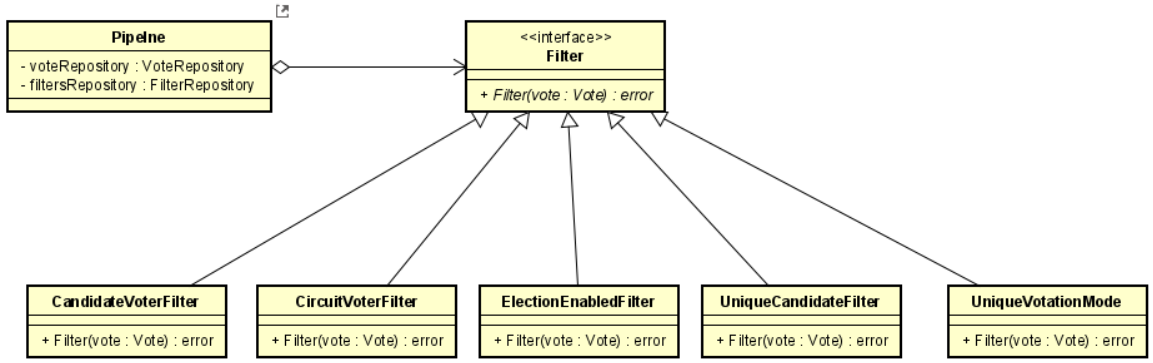


Diagrama de clases de los filtros

Como parte del *RNF 2 Modificar proveedor de mensajería* (servicio *VotationService*), es importante mostrar la herencia realizada en el paquete *Messenger*, en la cual aplicando el patrón de diseño **Strategy** si en el futuro se quiere cambiar de SMS a Whatsapp debemos instanciar en el paquete *Factory* la clase *Whatsapp*.

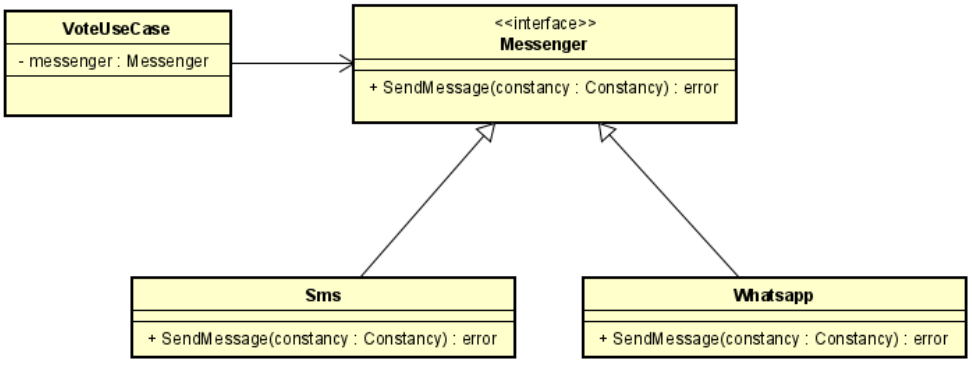


Diagrama de clases de proveedor de mensajería

Similar al caso anterior para el *RNF 1 Gestión de errores y fallas* (servicio *LogService*), se creó la interfaz *LoggerInterface* para que las clases hijas implementen el método *Log*.

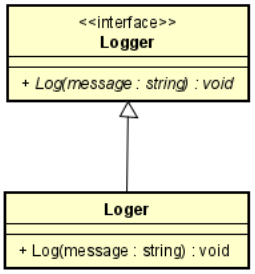


Diagrama de clases de logger

En cuanto a la elección, la solución debía poder adaptar fácilmente datos provenientes de otros países, y es por ello que se utilizó el patrón *adapter* de modo de convertir la interfaz de la elección en otra interfaz que espera el dominio del servicio. Esto se realizó basados en el *RNF 3 Poder recibir datos de elecciones de cualquier país*.

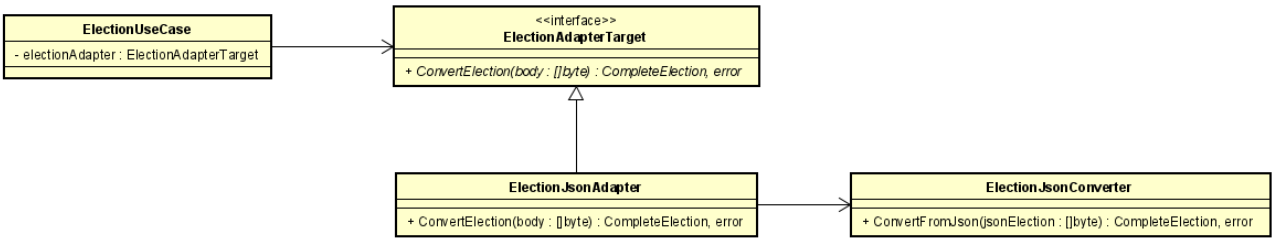


Diagrama de clases del patrón Adapter en ElectionService

3.1.4 Catálogo de elementos - Procesos que son APIs

El siguiente catálogo de elementos describe los diferentes módulos de los servicios que son APIs como *VotationService* que se pueden observar en la representación primaria. No todas las APIs poseen actualmente los mismos módulos. A pesar de ello, se espera que todas tengan la estructura jerárquica que fue descrita en la representación primaria de *Analytics*.

Elemento	Responsabilidades
Api	Se encarga de recibir las peticiones de los clientes e implementar middlewares que permitan realizar validaciones antes de que la petición ingrese al controller.
Usecases	Es donde se realiza la lógica de negocio luego de haberse realizado las validaciones del voto en Pipe. Se comunica con *dataaccess* para obtener o agregar datos de la base de datos.
Pipe	Se encarga de aplicar los filtros para validar el contenido del voto. Allí se pueden ver el uso del patrón *FanOut*

Dataaccess	Responsable de agregar o modificar datos de la base de datos de cada elección.
MemoryDataaccess	En algunos servicios como ElectionService el acceso a los datos en memoria fue separado de los datos que se acceden a Mongo. En otros esto no fue posible ya que existen métodos que utilizan ambas bases de datos.
Models	Allí se encuentran los modelos utilizados en todas las capas. Representa el dominio, aunque no se realizan más que atributos de clase y constructores.
Messenger	Paquete responsable de reducir el impacto de modificación ante un cambio en el proveedor de mensajería para enviar la constancia de voto.
Rabbit	Paquete que se encarga de enviar información a diferentes colas de mensajes.
Helpers	Es un intermediario que evita el acoplamiento entre Usecases y Pipe con el servicio de comunicación externa para encolar mensajes (actualmente Rabbit).
Factory	Es una fábrica que se encarga de instanciar las diferentes clases de usecases, dataaccess y controllers. Se realizó con el objetivo de que si se desea agregar un controller nuevo, al tener que crear su clase lógica de negocio y acceso a datos, solamente se tenga que crear una clase y crear las instancias allí de modo que solamente se realiza una llamada desde el "main" de la aplicación.
Config	Posee clases de configuración (destinado a los consultores específicamente). Se realizó también ya que si el día de mañana los filtros no se quieren acceder más desde Redis se puedan acceder desde un archivo de configuración.
Datasources	Posee las instancias que inicializan la conexión con las bases de datos.

3.1.5 Catálogo de elementos - Procesos que necesitan acceso a datos

Dichos procesos cuentan con los módulos de lógica de negocios (*usecases*), modelos (*models*), fuente de datos (*datasources*) y acceso a datos (*datasources*). Hoy en día al utilizar *Rabbit* existe el siguiente elemento:

Elemento	Responsabilidades

Workers	Responsable de recibir datos de una cola de mensajes para actualizar información en las bases de datos, ya sea o no en memoria.

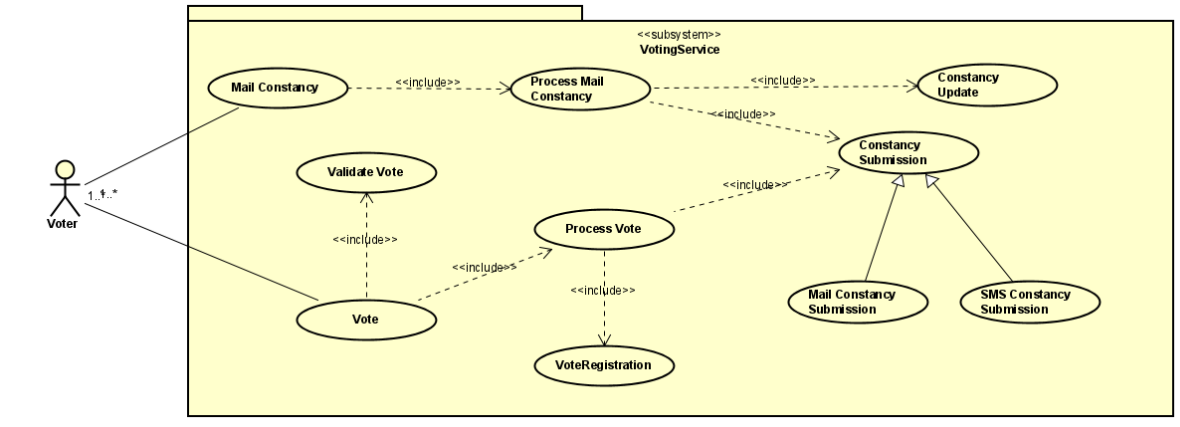
3.1.6 Catálogo de elementos - Procesos que no necesitan acceso a datos

Los procesos que no deben acceder a datos precisan al igual que los anteriores el elemento *Workers* y otros elementos que dependen según la responsabilidad del componente. A modo de ejemplo, existen servicios que necesitan un módulo donde se reúnen las clases encargadas del envío de información.

Elemento	Responsabilidades
Senders	Responsable de hacer el envío de mensajes, mails, entre otros. Esto hoy en día se hace mediante un print pero se espera en futuras versiones que se haga mediante proveedor de mensajería, SMTP, entre otros.

3.1.7 Comportamiento

3.1.7.1 Diagrama de comportamiento de la emisión de voto y solicitud de constancia vía mail



En el siguiente diagrama podemos ver por un lado la emisión del voto, donde el subsistema *VotingService* recibe la información del voto y en primera instancia la válida, dejando sin efecto las operaciones siguientes en caso de error. Si la validación es exitosa, se le otorga la respuesta al cliente y se procede a procesar el voto. Dicho procesamiento consiste en preparar la constancia y enviarla a otro subsistema (*MessageSenderService*), registrar el

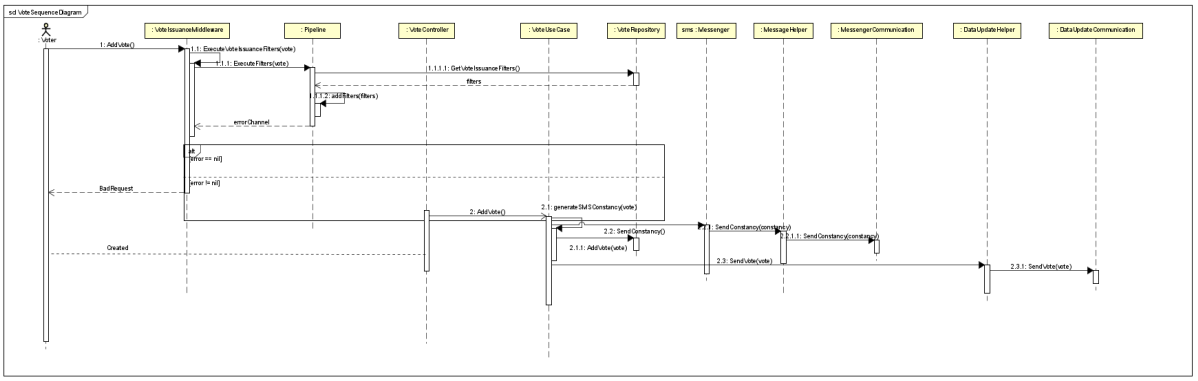
voto y enviar la información al subsistema que actualiza datos electorales (*DataUpdateService*).

Por otro lado, el votante cuando solicita la constancia de voto, se procesan los datos enviados (no se aplican filtros) y en caso de éxito se envía la información necesaria al subsistema responsable de enviar las constancias vía mail (*CommunicationSenderService*).

La comunicación con otros subsistemas que no se pueden ver en este diagrama al tratarse de un diagrama perteneciente a la vista de módulos se pueden ver en el diagrama de comportamiento de la vista de componentes y conectores.

3.1.7.2 Diagrama de secuencia de emisión de voto

El objetivo es observar cómo se valida la información del voto mediante un middleware que utiliza un pipeline con filtros extraídos de Redis (cargados por un consultor).

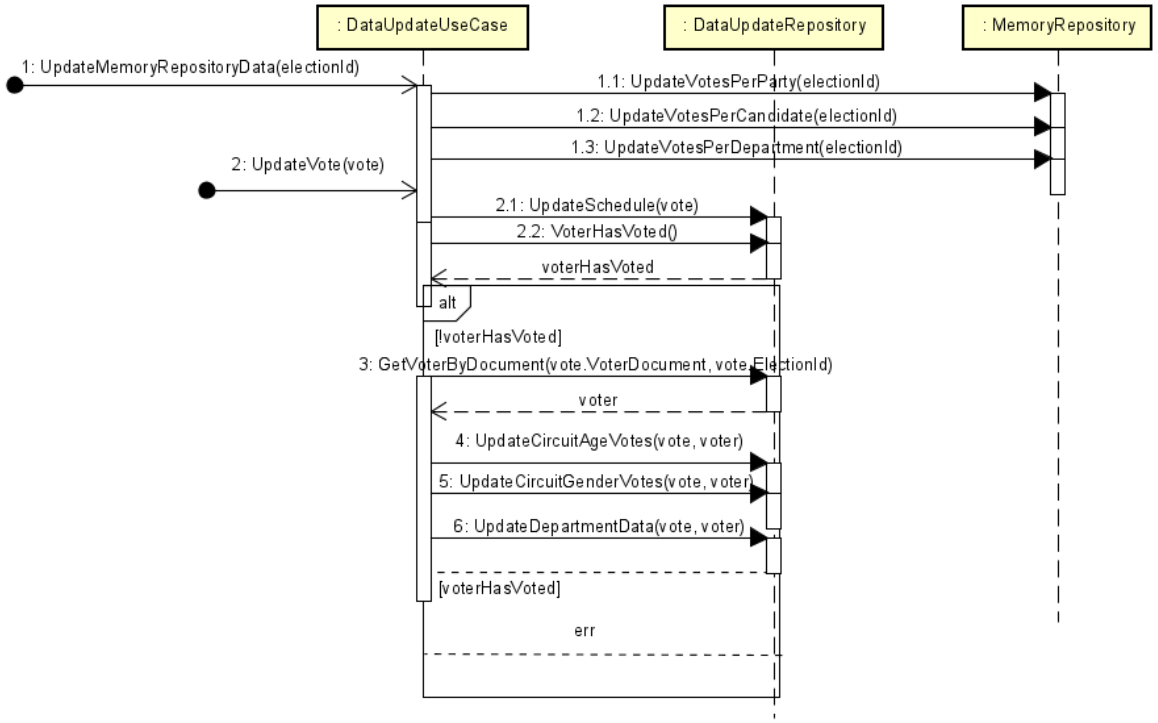


Una vez que ingresa la petición, la misma es recibida por un middleware encargado de realizar las validaciones del voto (o datos de elección si fuera en *ElectionService*). Los filtros se obtienen de la base de datos en memoria (en este caso Redis). Como se describirá en la vista de componentes y conectores, se utilizó el patrón de concurrencia *FanOut* en el cual se ejecutan los filtros de forma concurrente. Si alguno de los filtros arroja error, se le da la respuesta con el código de estado 400 al votante.
En caso de que el voto logre pasar los filtros, ingresa al controller que llama a la lógica de negocios para que se realicen los siguientes procesos:

- Generar la constancia y encolarla en la cola de mensajes.
- Agregar el voto en la base de datos de la elección.
- Encolar la información del voto para comunicarse con el servicio *DataUpdateService* quien actualiza la información en la base de datos electoral.

Antes que eso (hay un error en el diagrama), ya se le da la respuesta al votante (favoreciendo la performance), y se deja haciendo asincrónicamente los procesos mencionados anteriormente, lanzando go routines.

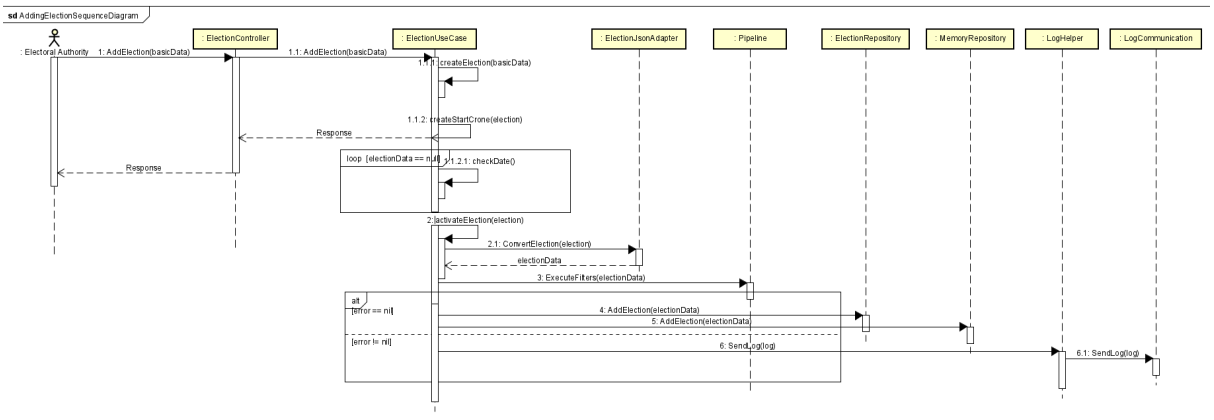
3.1.7.3 Diagrama de secuencia de actualización de datos en *DataUpdateService*



El propósito del diagrama es mostrar la responsabilidad del servicio *DataUpdateService*. Cada cierto tiempo el servicio actualiza los datos de Redis para cada elección, haciendo posible la lectura de los datos actualizados por parte de los usuarios que realizan consultas (votos por partido, departamento y candidato). Por otro lado desencola votos y actualiza en la base de datos electoral. Siempre actualiza los horarios frecuentes de votación, pero dependiendo si ya votó anteriormente (almacenamos una lista de horarios en los que votó), no se tienen que actualizar los datos de votos por edad y sexo en circuito y departamento.

3.1.7.4 Diagrama de secuencia del envío de datos básicos de elección por parte de una autoridad electoral y recepción de datos para comenzar la elección

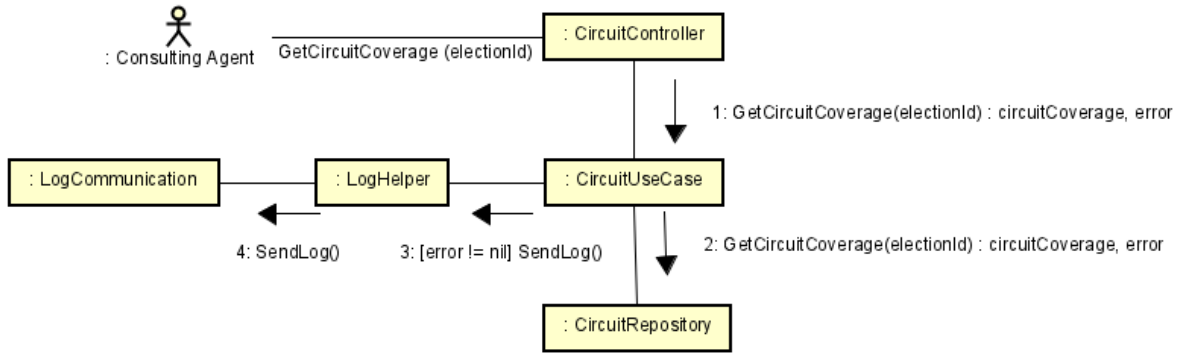
El equipo consideró pertinente en este punto aclarar el procedimiento de cómo el sistema inicia una elección. En primer lugar, la autoridad electoral enviará a *appEV* una descripción de la elección, la url para obtener los datos completos y la fecha de inicio.



Como se puede apreciar, en primera instancia una autoridad electoral envía los datos básicos de una elección, cuyos datos de mayor importancia son la URL para poder obtener el resto de los datos en algún momento y la fecha de comienzo. Una vez que ingresa la petición se crea la elección con dichos datos y se utiliza un cron para - cada cierto tiempo - consultar el resto de los datos. Hecho esto se devuelve la respuesta y asincrónicamente se consultan los datos. Cuando estos llegan se utiliza el adapter para transformar los datos que la autoridad electoral envió a *appEV*. Dichos datos son validados a través de un pipeline con filtros, en los que si alguno arroja error la elección no podrá comenzar. Si los datos pasan los filtros, se podrán agregar los datos a la base de datos tanto de la elección como la electoral (general para todas las elecciones). En dicho punto está representado el uso de la gestión de fallas, donde por ejemplo si la llamada a la base de datos resulta en error, a través del *LogHelper* se comunica con *LogCommunication* quien finalmente encola el error para que luego el servicio de logs lo desencole.

3.1.7.5 Diagrama de comunicación de la obtención de la cobertura de votos por circuito en *Analytics*

El objetivo principal de este diagrama es mostrar cómo se da la comunicación para enviar al servicio de logs (*LogService*) la información con la descripción del error.

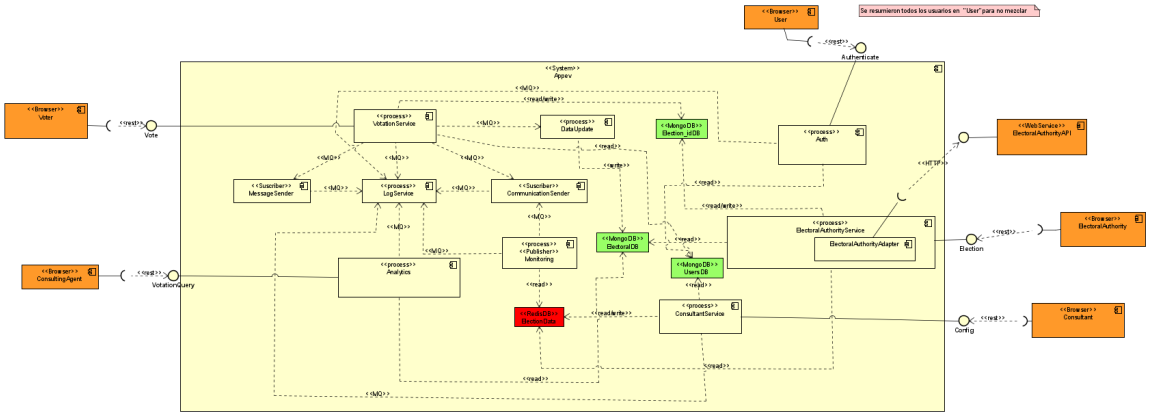


Como se puede observar, una vez que ingresa la petición al controller, este se comunica con la lógica de negocio, donde si la llamada al repositorio retorna error, se encola la información a la cola de mensajes de log (recibida por *LogService*) utilizando como intermediario a *LogHelper*.

3.2 Vistas de Componentes y conectores

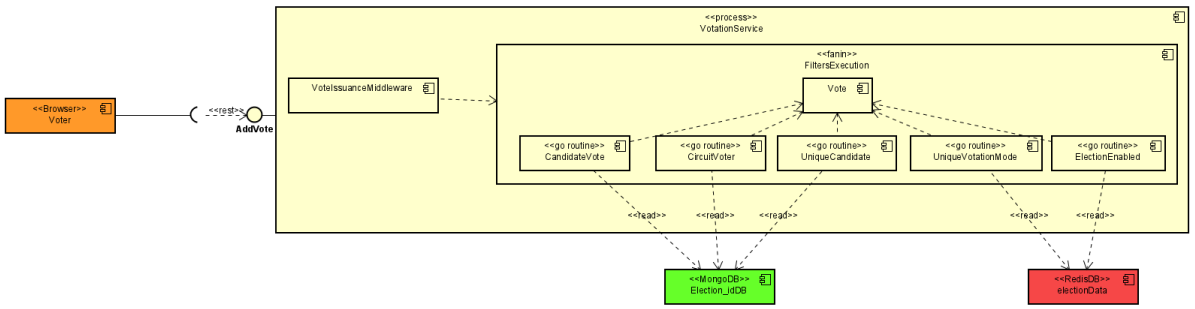
En esta sección se describen las vistas de componentes y conectores con el objetivo de comunicar la visión del sistema en tiempo de ejecución.

3.2.1 Representación primaria



3.2.2 Representación primaria - Profundización filtros aplicados en *VotationService*

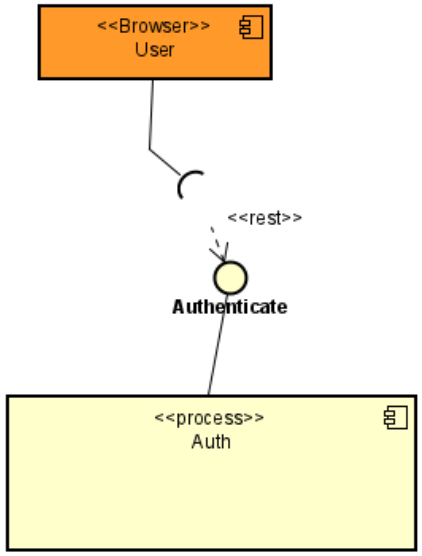
Para favorecer la performance es clave la introducción de concurrencia. Tanto en la emisión del voto como los datos provistos por la API de autoridad electoral para iniciar y cerrar la elección se deben aplicar una serie de filtros, lo cual hacerlo sincrónicamente no lograría el cometido. Por ello utilizamos un patrón de concurrencia de Go llamado *Fan-out* en el cual múltiples funciones se ejecutan concurrentemente recibiendo como parámetro un voto. Si alguno retorna un error, se dejan de correr los filtros restantes.



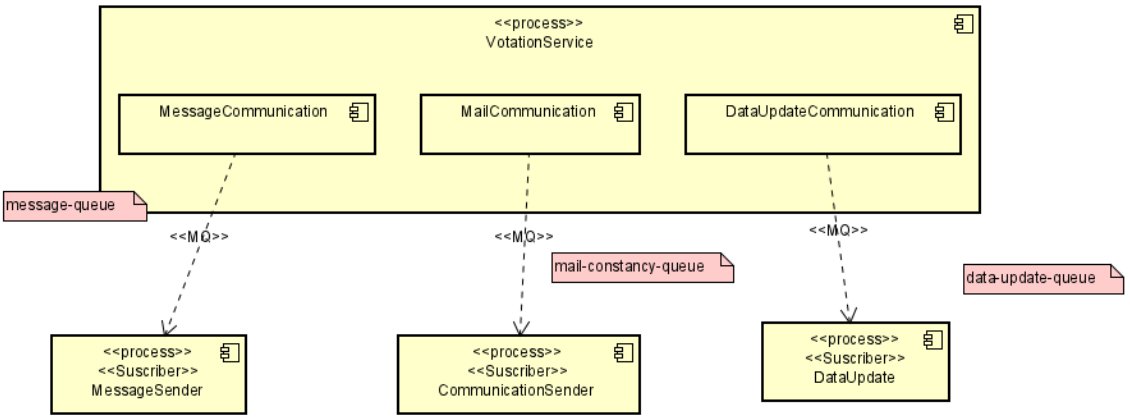
3.2.3 Representación primaria - Autenticación con servicio *Auth*

El propósito del siguiente diagrama es mostrar que los usuarios del sistema se deben autenticar con un servicio perteneciente a nuestro sistema responsable de realizar la autenticación, entregando una token al usuario en caso de éxito. Este token debe ser presentado por el usuario al servicio con el cual desea comunicarse. La decisión de utilizar el patrón de **identidad federada** radica principalmente en la seguridad y modificabilidad ya

que delegar la autenticación a un proveedor de identidad (en este caso interno) simplifica el desarrollo puesto que nos evita tener que crear un mecanismo por cada servicio que requiere autenticación.



3.2.4 Representación primaria - Profundización uso de MQ para actualizar información de votos



A la hora de la actualización de los datos de los votos se utilizaron colas de mensajes para que la estas actualizaciones no afecten la performance de votación. Se decidió usar comunicación asincrónica ya que estas actualizaciones no se tienen que realizar al instante de realizado el voto. Esta decisión también ayuda a que no se sature el sistema de votación ya que el sistema de actualización de datos, el sistema de mensajería y el de comunicación son externos al sistema de votación.

3.2.5 Catálogo de elementos

Componente/conector	Tipo	Descripción

VotationService	***Componente***	Es una api que permite a los votantes emitir votos y poder solicitar la constancia vía mail.
ElectoralService	***Componente***	Funciona en primer lugar como receptor de los datos de diferentes elecciones de diferentes autoridades electorales para comenzar una determinada elección así como finalizarla. Por otro lado es una api que permite al usuario autoridad electoral realizar consultas respecto a una elección.
Analytics	***Componente***	Es una api que permite a los agentes de consulta obtener datos de una elección
ConsultantService	***Componente***	Es una api que permite a los consultores de la aplicación realizar determinados ajustes
MessageSenderService	***Componente***	Recibe las constancias mediante MQ y envía mediante un servicio de mensajería (hoy en día SMS) la constancia al votante. Este servicio está separado de CommunicationSender para no saturar a un único servicio ya que los mensajes sms pueden ser muchos al mismo tiempo y se precisan enviar lo más rápido posible.
CommunicationSender	***Componente***	Recibe: ● Información para enviar la constancia de voto por mail. ● Información para enviar alerta de desvío de valores a los consultores. ● Información para enviar los certificados de apertura y cierre de una elección. En un principio pensamos que lo mejor sería separar estas responsabilidad en componentes separados. Luego de analizar, llegamos a la conclusión de que por cuestiones de performance reducir la comunicación entre componentes sería una buena idea.
MonitoringService	***Componente***	Monitorea constantemente si hay desviación de los valores esperados seteados por los consultores de la aplicación. De haberlos, se comunica mediante MQ con *CommunicationSenderService* para que pueda enviar mail a los consultores.
LogService	***Componente***	Recibe información de errores de todos los servicios de la aplicación y los loguea en un txt.

DataUpdateService	***Componente***	Por un lado recibe información sobre votos y actualiza la información de la base de datos ElectoralDB. Por otro lado, cada 30 minutos actualiza información en la base de datos en memoria.
Auth	***Componente***	Es el proveedor de identidad. El usuario se intenta loguear y si lo consigue se le da un token. Con el mismo podrá efectuar operaciones sobre el servicio que le corresponde.
MQ	***Conector***	Representa una comunicación mediante cola de mensajes.
REST	***Conector***	Representa una comunicación mediante protocolo de transferencia de hipertexto.
Election_idDB	**Componente**	Es una base de datos en Mongo la cual contiene datos de una elección en particular (por ello el "id").
ElectoralDB	**Componente**	Es una base de datos en Mongo la cual contiene datos de todas las elecciones (a excepción de votos y votantes).
UsersDB	**Componente**	Es una base de datos en Mongo la cual contiene a los usuarios del sistema.
RedisDB	**Componente**	Es una base de datos en memoria en la cual se aloja información de la que se quiere tener rápido acceso, como información general de una elección, votos de la misma, filtros y valores esperados.

3.2.6 Interfaces

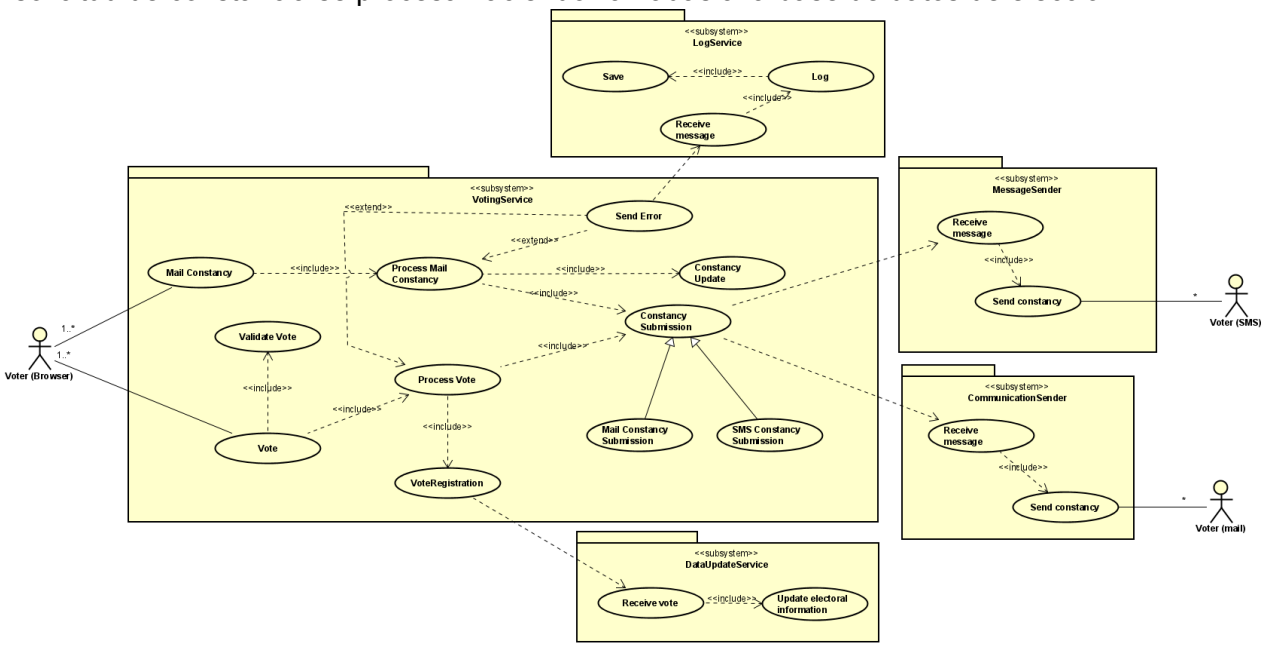
Servicio	***Descripción***
WebService	Cada servicio que posee este estereotipo en el diagrama de componentes y conectores expone un servicio web a los usuarios con los cuales se comunica mediante HTTP.
Rabbit MQ (MQ)	Hoy en día los componentes que no son bases de datos se comunican asíncronamente mediante MQ para favorecer la performance.

3.2.7 Comportamiento

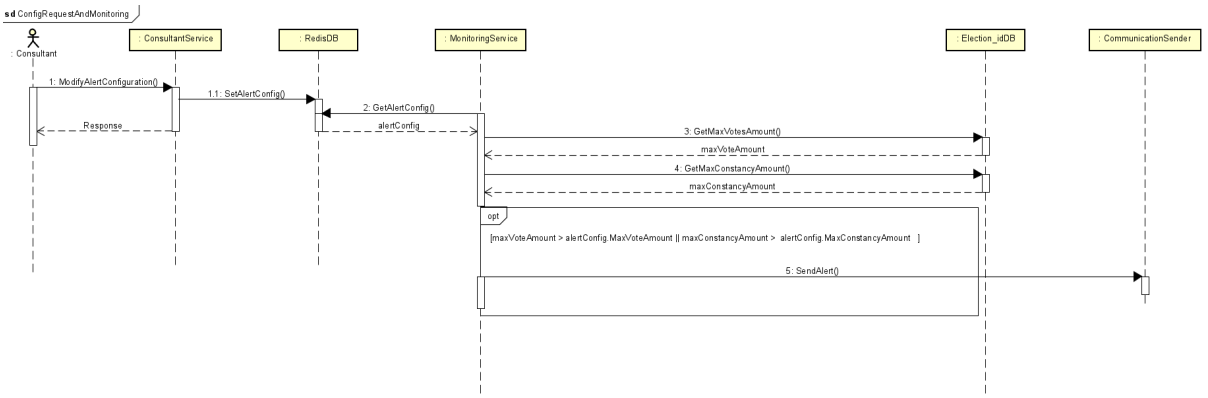
3.2.7.1 Diagrama de comportamiento de la emisión de voto y solicitud de constancia de voto vía mail

El siguiente diagrama intenta explicar dos diferentes casos de uso: emitir voto y solicitar constancia vía mail. Cuando el votante emite un voto se realizan las validaciones y procesamientos pertinentes para registrar el voto y preparar la constancia para su encolamiento a la cola de mensajes, quien desencola el subsistema *MessageSenderService* para enviar el SMS al votante. Si algún paso falla en el procesamiento se envía el error para que *LogService* lo guarde. Por otro lado se encola la información del voto para que luego el *DataUpdateService* actualice la información en la base de datos electoral.

La solicitud de constancia se procesa haciendo llamadas a la base de datos de elección

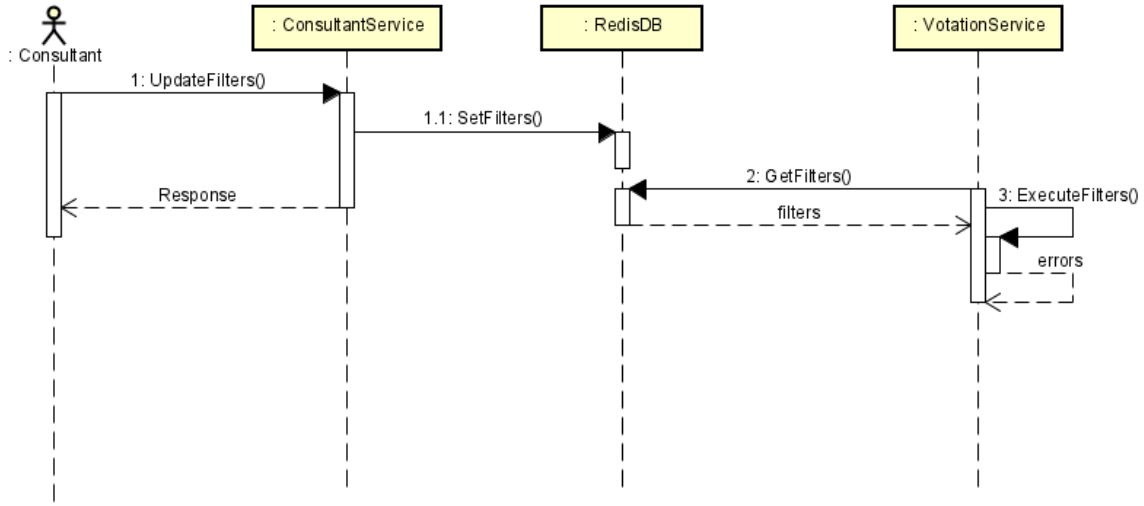


3.2.7.2 Diagrama de secuencia de la actualización de las configuraciones de alerta y posterior monitoreo y envío de mails en caso de desvío



MonitoringService está constantemente monitoreando el desvío para enviar de forma rápida a los consultores la alerta correspondiente. Para ello vemos como el consultor realiza una modificación de la configuración y el sistema de monitoreo con frecuencia (tiempo configurable) consulta a Redis los valores esperados y por otro lado obtiene los valores reales. Si hay un desvío, encola el mensaje para que *CommunicationSender* lo desencole y envíe los mails a los consultores.

3.2.7.3 Diagrama de secuencia de la modificación de filtros



Una vez que un consultor de *appEV* modifica los filtros para validar datos (ya sea de elección o voto), se agregan en memoria, para que luego el servicio (en este caso de votación) los obtenga y los cargue para ejecutarlos.

3.2.8 Relación con elementos lógicos

Componente	Paquetes
VotationService	*Api, Usecases, Pipe, Messenger, Dataaccess, Datasources, Factory, Helpers, Rabbit, Config*
ElectionService	*Api, Usecases, Pipe, Dataaccess, MemoryDataAccess, Datasources, Factory, Helpers, Rabbit, Config*
Analytics	*Api, Usecases, Dataaccess, Datasources, Factory, Helpers, Rabbit*
ConsultantService	*Api, Usecases, Dataaccess, Datasources, Factory, Helpers, Rabbit*
MessageSenderService	*Senders, Workers, Models, Config*
CommunicationSender	*Senders, Workers, Models, Config*
MonitoringService	*Rabbit, Helpers, Usecases, Dataaccess, Datasources, Models*
LogService	*Workers, Logger*
DataUpdateService	*Workers, Usecases, Dataaccess, MemoryDataAccess, Datasources, Models*
Auth	*Api, Usecases, Dataaccess, Models, Factory, Helpers, Datasources,*

3.2.9 Decisiones de diseño y análisis realizados

En primer lugar, identificamos que debíamos tener cuidado con la performance en la emisión del voto, ya que la **latencia** en promedio no podía exceder los 2 segundos. Es por ello que decidimos que la respuesta al votante sea rápida, para ello lo único realizando antes de darle una respuesta al usuario es verificar que el voto es válido para nuestro sistema (cuando pasa los filtros sin tirar ningun error), luego de esto utilizamos ***introduce concurrency*** usando go routines para continuar con el resto de las operaciones sin retrasar la respuesta. También decidimos que dentro del servicio de votación la única operación que se hace es el agregado a la base de datos de la elección del voto, esta decisión fue para proteger la integridad de los datos accediendo desde este servicio a la base de datos de la elección y a su vez para no saturar al mismo sistema de votación con múltiples operaciones. Luego se publicó la información del voto en una cola de mensajes, la cual el servicio *DataUpdate* se encarga de desencolar para luego actualizar la información de la base de datos electoral (información que consultan los demás usuarios). Es decir, estamos ***priorizando eventos*** y ***limitando la respuesta*** con el objetivo de aumentar la performance de voto.

Por otro lado, el SMS enviado al votante también debía ser rápido y es por ello que utilizando la técnica ***introduce concurrency*** en la cual mediante el uso de las go routines que proporciona el lenguaje Golang hicimos que la constancia sea preparada y encolada inmediatamente para luego procesar diferentes secuencias de eventos en diferentes subprocesos.

Al existir consultas en las que también la **performance** es un atributo de calidad exigido, decidimos **mantener múltiples copias de datos** y usar ***CQRS*** creando modelos para lectura, tanto en la base de datos de memoria como la base de datos que no está en memoria, de modo que se pueda tener un rápido acceso a la información relevante para las mismas. El servicio *ElectoralService* lo hace específicamente por el *RF 8 Consulta del resultado de la elección* y *MonitoringService* debido a que puede alertar con mayor rapidez a los consultores del desvío de valores. Es preciso aclarar que los filtros se guardan en Redis puesto que al emitir un voto se llegó a la conclusión que obtenerlos de la memoria iba a mejorar la performance ya que obtenerlo de un archivo de configuración iba a enlentecer el procesamiento.

En términos de **seguridad**, se decidió crear una base de datos **por elección** con los datos del voto, de modo de restringir el acceso, tanto de lectura como de escritura, a personas no autorizadas a información **sensible** como la es el voto y así favoreciendo la **confidencialidad** y la **integridad** de las mismas. Los datos del voto fueron encriptados mientras que los datos de los votantes al limitar el acceso.
Las variables de configuración como las urls para conectarse a mongo, redis o rabbit así como las claves privadas para la desencriptación de datos y puertos de conexión fueron guardadas en un archivo de entorno.

El equipo tuvo diferentes discusiones en algunos aspectos. Uno de ellos es que los requerimientos de consultas 10,11 y 12 se repetían para varios tipos de usuarios. Una opción que discutimos fue favorecer la modificabilidad *reduciendo el acoplamiento* con la técnica de *abstracción de servicios comunes*. Al pensar en dónde podríamos abstraer el servicio, lo ideal hubiera sido hacerlo en el servicio *ElectoralService* puesto que es el que reúne otras consultas. Sin embargo, este servicio al estar destinado a las autoridades electorales pueden existir momentos en el que recibe una gran cantidad de datos y eso podría enlentecer las respuestas a los usuarios. Sumado a que el *RF 7 Consulta de voto* y *RF 7 Consulta del resultado de una elección* deben tener una latencia no mayor a los dos segundos, el riesgo era grande.

Otra opción fue comunicarse mediante gRPC o HTTP con un servicio extra de donde se obtendría la información necesaria. No obstante, la cantidad de intermediarios sería grande también impactando en la performance. Es por ello que finalmente decidimos que cada servicio evite comunicaciones y por ello puede verse gran cantidad de código repetido.

Otro aspecto discutido fue el de cómo registrar las **fallas del sistema**. Se discutieron dos opciones:

- En un equipo donde trabajen más integrantes, sería factible que hubiera diferentes equipos responsables de servicios en concreto. Es por ello que al ser aplicaciones o procesos que se ejecutan de forma separada cada una de ellas podría guardar logs. Esto favorece la performance ya que se evita el uso de intermediarios.
- Al ser servicios que forman parte del mismo sistema, se podría crear un servicio que posea la responsabilidad de guardar los logs, favoreciendo la modificabilidad pero desfavoreciendo la performance.

El atributo de calidad que terminó de definir nuestra decisión fue la modificabilidad, ya que el *RNF 1 Gestión de errores y fallas* nos indica que el cambio de herramientas para registrar las fallas no debe tener un alto impacto en la solución. Haber guardado logs en servicios separados hubiera sido costoso en términos de modificabilidad ya que se tendrían que modificar todos los componentes pertenecientes al sistema.

En cuanto a las bases de datos elegidas, se eligió MongoDB (base de datos no relacional) para aquellas bases de datos donde se guardan grandes cantidades de datos. Esta decisión fue tomada en base a:

- **Alta disponibilidad**.
- **Flexibilidad**. Como *appEV* es un proyecto greenfield y los requerimientos fueron cambiando, el mapeo de objetos rápido que ofrece MongoDB fue clave para el equipo.
- **Alta escabilidad**.

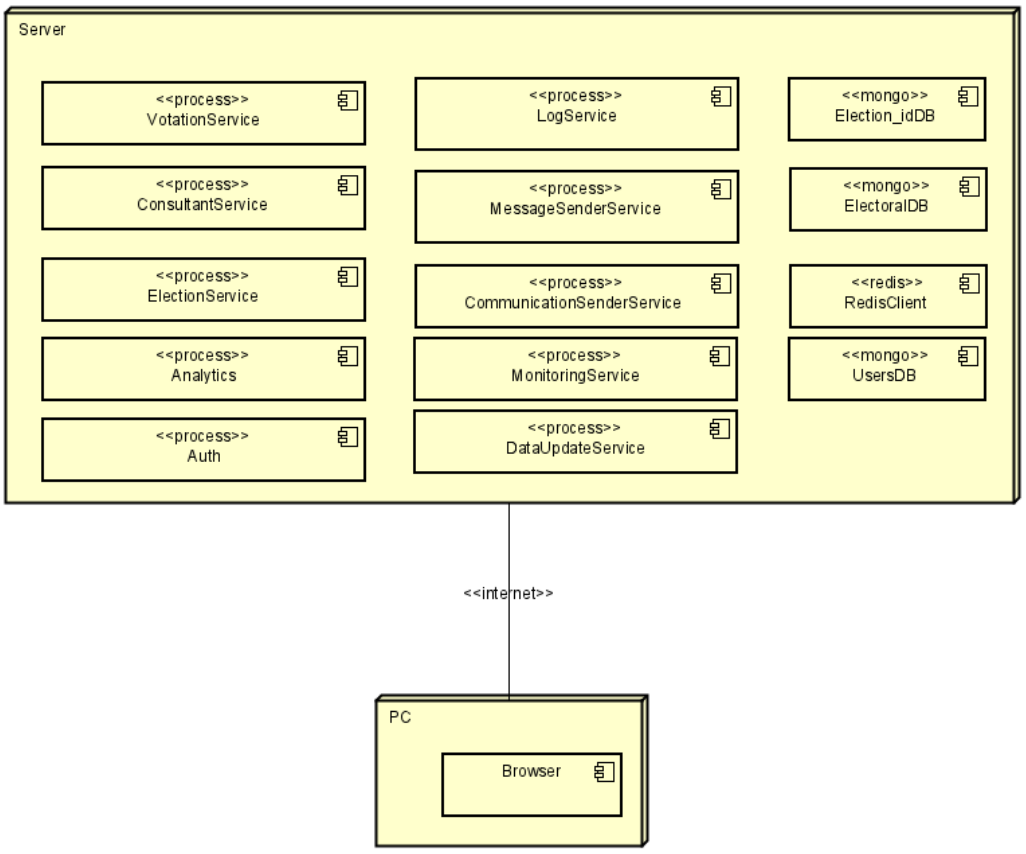
Para la búsqueda de datos en memoria la base de datos elegida fue RedisDB ya que fue la enseñada en clase y el equipo desconocía este tipo de motor. Como se explicó anteriormente, el acceso a datos en memoria es algo clave para mejorar la performance al obtener datos y es por ello que el equipo decidió utilizarla.

3.3 Vistas de Asignación

En esta sección el lector podrá encontrar una descripción de cómo los elementos de software se relacionan con su entorno.

3.3.1 Vista de Despliegue

3.3.1.1 Representación primaria



3.3.1.2 Catálogo de elementos

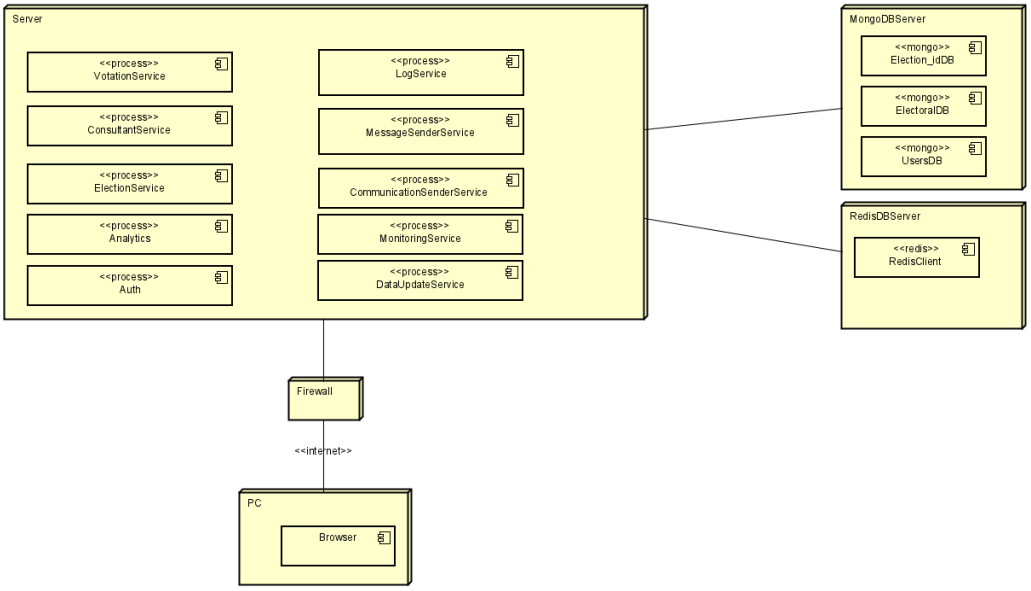
Nodo	Características	Descripción
Server	Memoria RAM 16 GB, CPU Intel i5 8va generación, 200 mb/s de velocidad de bajada	Esta es la computadora de uno de los integrantes del equipo. Como podemos observar, las características del servidor no están aptas para procesar grandes cantidades de peticiones sin que comience a fallar. Una mejora sería mejorar el hardware y comenzar a mantener múltiples copias de cómputo para mejorar la performance.
PC	Memoria RAM 1 GB, 1 mb/s de velocidad	La PC únicamente realizará requests HTTP por lo que no debe contar con grandes requisitos a nivel de hardware.

Conector	Características	Descripción
Internet	1 mb/s de bajada	No se requiere una gran conexión a internet para realizar las peticiones.

3.3.1.3 Decisiones de diseño y análisis

Se decidió utilizar un único servidor haciendo uso de *Docker* el cual nos permite tener contenedores con las bases de datos utilizadas. Esta decisión es visiblemente mala ya que al tener limitaciones en el hardware del servidor y correr todo en el mismo nodo la performance no va a ser alcanzada para una carga considerable.

Es por ello que una buena idea para una futura versión sería tener un servidor separado con los procesos que pueden correr en computadoras separadas y un servidor por cada sistema de base de datos.En términos de seguridad, se podría contar con un **firewall** para proteger el acceso no autorizado al servidor.



Contando con un presupuesto mayor, lo ideal sería tener servidores separados para componentes críticos (servicios de votación, elección, analytics, configuraciones, autenticación) y otro servidor conteniendo procesos como el servicio de logs o el responsable de enviar constancias.

4. Prueba de carga

Para estas pruebas no se utilizó la autenticación jwt pero los filtros para validar el voto fueron realizados.
Se realizó un emulador de votos y se hicieron las siguientes pruebas:

- Primero se probó asincrónicamente la ejecución de peticiones de 1000 votantes votando al mismo tiempo, el sistema logró responder abajo de los 2 segundos a 106 de los votantes.
- Luego se decidió reducir la cantidad de requests enviadas al mismo tiempo y se enviaron 100 grupos al mismo tiempo de 10 votantes, el sistema logró responder bajo de los 2 segundos a 149 de los votantes.
- Para esta prueba se decidió enviar 50 grupos al mismo tiempo de 20 votantes cada uno y el sistema logró responder abajo de los 2 segundos a 213 de votantes.
- Continuamos probando con 10 grupos al mismo tiempo de 100 votantes cada uno y el sistema le logró responder a cada uno en menos de 2 segundos.
- Por último también se decidió probar el envío sincrónico de 1000 votantes y el sistema logró responderle a cada uno en menos de 2 segundos.

Notamos que estos resultados fueron realizados en la computadora de uno de los integrantes del equipo y no en un servidor específico con todos los recursos que se creen necesarios para dar mejores resultados. Si el sistema se lleva a producción se espera que estos resultados mejoren.