Universidad ORT Uruguay

Obligatorio

Arquitectura de Software

Agustina Disiot 221025

Iván Monjardin 239850

Francisco Rossi 219401

23 de Junio de 2022

Índice:

1. Introducción:

El Documento de Descripción de arquitectura (DA), está orientado a arquitectos de software, por lo que se presentarán algunos términos y definiciones que se asumen conocidos. El documento cuenta con una estructura dividida en descripción del sistema y diferentes vistas, presentando diagramas para cada una y sus explicaciones correspondientes.

1.1. Propósito

El propósito del presente documento es proveer una especificación completa de la arquitectura del AppEv. En la siguiente sección se identificarán requerimientos tanto funcionales como no funcionales que servirán para entender nuestro diseño de arquitectura y para ser de guía siguiendo las diferentes vistas del sistema del modelo 'Views & Beyond"

1.2. Repositorio

El siguiente link hace referencia al repositorio utilizado por el equipo. Cabe aclarar que se utilizó trunk-based development por lo que se observan muchos commits a main y ramas de corta vida. Se decidió trabajar con esta metodología ya que fue presentado en otro curso paralelo a esta materia y nos pareció interesante llevarlo a la práctica en un proyecto más complejo.
Link al repositorio:
https://github.com/ORTArqSoft/239850_221025_219401

2. Descripción del Sistema:

2.1. Propósito del sistema

El sistema provee diferentes servicios con el objetivo de proponer una solución para crear una plataforma de votación online. Se puede votar, realizar consultas sobre las votaciones e integrar el sistema con otros sistemas electorales ya existentes.

2.2 Requerimientos significativos de Arquitectura

2.2.1 Resumen de Requerimientos Funcionales:

1. **Configuración de la elección:**

   El sistema permite recibir una elección de un servicio electoral externo y en base a la información brindada, configurar fechas de inicio y fin de la elección. Esto activará un sistema de notificación para indicarle a ciertas personas cuando una elección ha comenzado y finalizado respectivamente.
   Se pueden configurar parámetros como qué elección solicitar por medio de un endpoint con un identificador único o que se notifique por email a ciertas personas en caso de exceder un límite en el ejercicio de votar o consultar demasiados certificados de votos.

2. **Votar y obtener constancia**

   La principal función del sistema es que permite votar dado ciertos datos valores requeridos. El sistema validará estos datos y en función de esas validaciones, las acciones que tomará. En caso de proceder correctamente, se notificará por SMS al usuario el resultado de la acción tomada.
   Cabe destacar que consideramos el voto como información sensible y no pública, por lo que es un de los datos que se envía encriptado para que no hayan ataques de hombres en el medio modificando lo que el usuario envió al sistema.

3. **Autenticación y Gestión de Permisos**

   Se creó un sistema de autenticación para la protección de acceso a las funcionalidades. Los usuarios se deberán registrar con usuario, contraseña y rol para obtener un token de nuestro servicio de autenticación. Luego con ese token podrán acceder a diferentes consultas.
   Este es un desglose de los roles y sus permisos:

   Autoridad Electoral:
   - Consultar un voto específico
   - Consultar el resultado de la elección en cualquier momento.
   - Consulta de la configuración de la plataforma

   Consultores de AppEv
   - Consulta de la configuración de la plataforma

   Todos:
   - Horarios más frecuentes de votación
   - Cobertura de votos de cada circuito, discriminados por grupo de edades predefinidas y sexo.
   - Cobertura de votos de cada departamento, descriminados por grupo de edades predefinidas y sexo.

Votante:

- Votar y pedir certificado de votación

4. **Generación de estadísticas**

   Dado que el sistema es de interés nacional y que la información proporcionada es de suma importancia, decidimos implementarlo de tal manera que genere información actualizada lo más rápido posible. Para obtener esta información, creamos un servicio de consultas al que luego de registrarse y obtener un token acorde al rol correspondiente, se puede acceder a información actualizada, como puede ser el resultado de la elección, detallada por candidato o por partido políticos, o las horas mas populares de los votos, entre otras consultas. Se buscó que fueran óptimas estas consultas y actualizadas en tiempo real.

5. **Gestión de errores y fallas**

   Nuestro sistema está pensado para funcionar en situaciones donde se hará un uso intensivo del mismo. Para ellos se creó una librería propia de logueo la cuál se puede utilizar a lo largo del sistema y discriminado según es un error, información relevante o simplemente una advertencia en caso de que así se desee informar. Esto genera un archivo específico para cada servicio para tener una idea clara de que ocurrió en cada servicio específico.

2.2.2 Resumen de Requerimientos de Atributos de Calidad

Nuestro sistema fue diseñado con varios atributos de calidad (ACs) en mente. Tuvimos que priorizar los que consideramos más importantes dado los requerimientos solicitados. Entre los más importantes se encuentran seguridad y performance. Luego vendría la interoperabilidad, mantenibilidad y disponibilidad. Por último, aunque también son importantes se encuentra escalabilidad y testeabilidad.

- **Performance**
- **Seguridad**
- **Interoperabilidad**
- **Mantenibilidad**
- **Disponibilidad**
- **Escalabilidad**
- **Testeabilidad**

3. Documentación de la arquitectura

3.1 Vista de Módulos

3.1.1 Vista de Descomposición

3.1.1.1 Representación Primaria

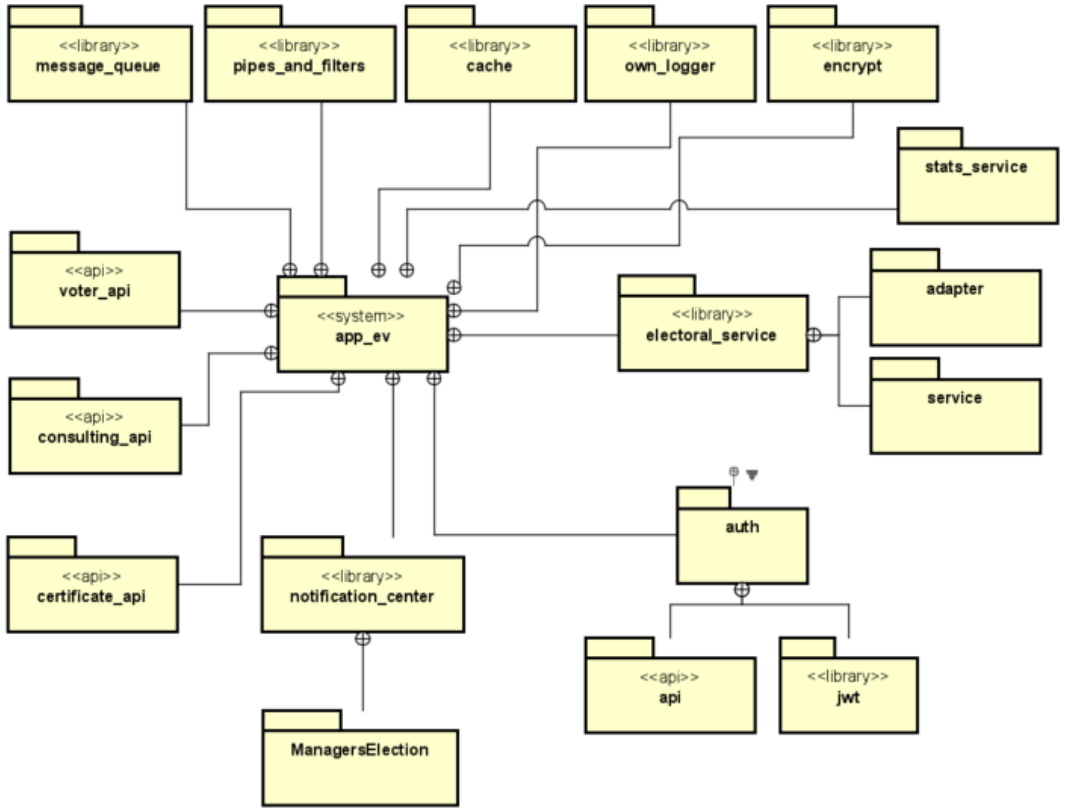


Figura 3.1: Diagrama de descomposición de paquetes.[1]

3.1.1.2 Decisiones de diseño

Servicios

Decidimos implementar un sistema orientado a servicios (SOA), donde cada servicio utiliza librerías internas como son own_logger, pipes_and_filter y también externas como fiber para exponer APIs como medio de comunicación por medio de interfaces. Se buscó apuntar a la mantenibilidad, por lo que se crearon varios servicios que se pueden desplegar independientemente y así mantenerlos sin afectar tantos aspectos del sistema en conjunto.

[1] Diagrama encontrado en Documentation/Diagrams/Descomposition.svg

Además, dividir el sistema en varios servicios y librerías agilizo el desarrollo ya que cada integrante podía tocar sistemas distintos estando seguro que no iba a ver conflictos.

Adapter

Para cumplir con el atributo de calidad de interoperabilidad es necesario que se puede cambiar el Servicio de Autoridad Electoral de manera sencilla. Para eso ponemos un intermediario entre la lógica principal del electoral_service y la comunicación con la api del sistema electoral. Entre medio hay un adapter el cual implementa una interfaz, la lógica del electoral_service usa cualquier struct que implemente la interfaz. Además cambiar qué adapter usar es bastante fácil, incluso se pueden agregar varios adapters al sistema y elegir cual usar mediante una variable de entorno configurable en el despliegue, sería un patrón Strategy.

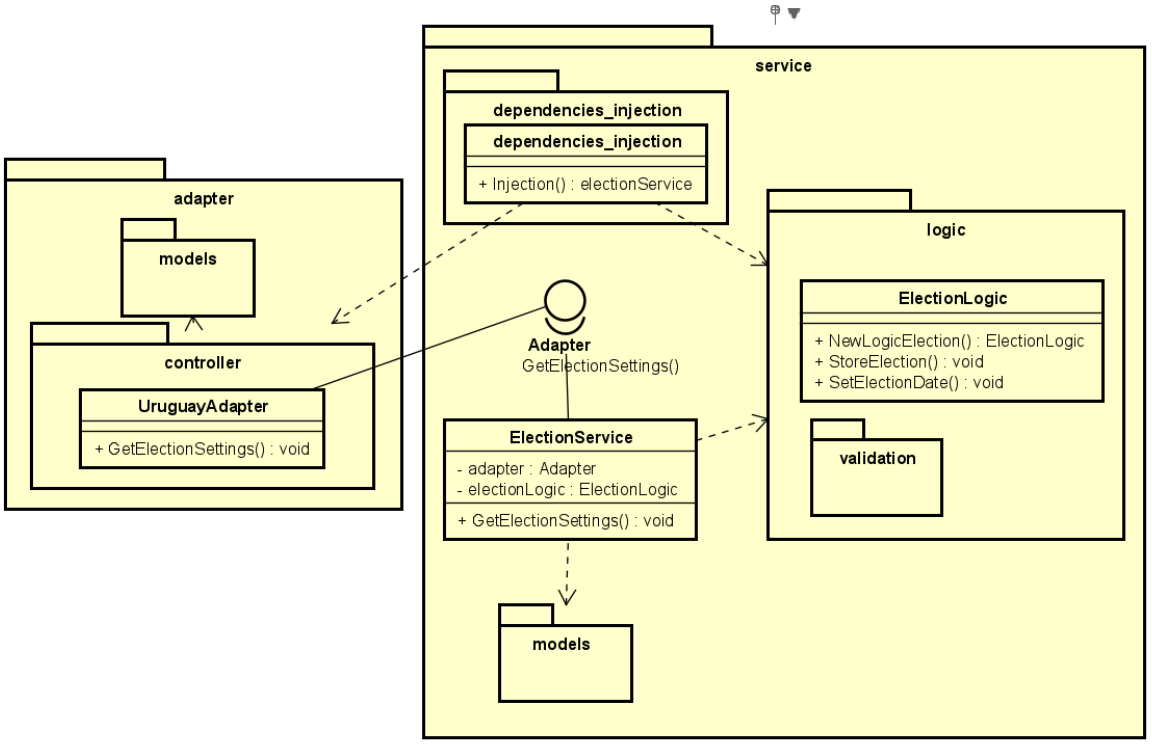


Figura 3.2: Diagrama de clases electoral_service.[2]

Modelos

Además de usar un adapter para recibir los datos de un sistema externo, también es necesario pasarlo a modelos soportados por el sistema. Por ejemplo para pasar los datos de un votante del sistema externo a appEV, para ello definimos un struct con datos consideramos esenciales o básicos, y el resto los guardamos en la base de datos mapeandolos en un único objeto "OtherFields". Se tendría que codificar un nuevo adapter que lo que tendría que hacer es solicitar los datos de la elección particular del país y enviar un struct para que se pueda mapear a nuestro struct principal.

[2] Diagrama encontrado en Documentation/Diagrams/ClassElectoralService.svg

Pipes and filters

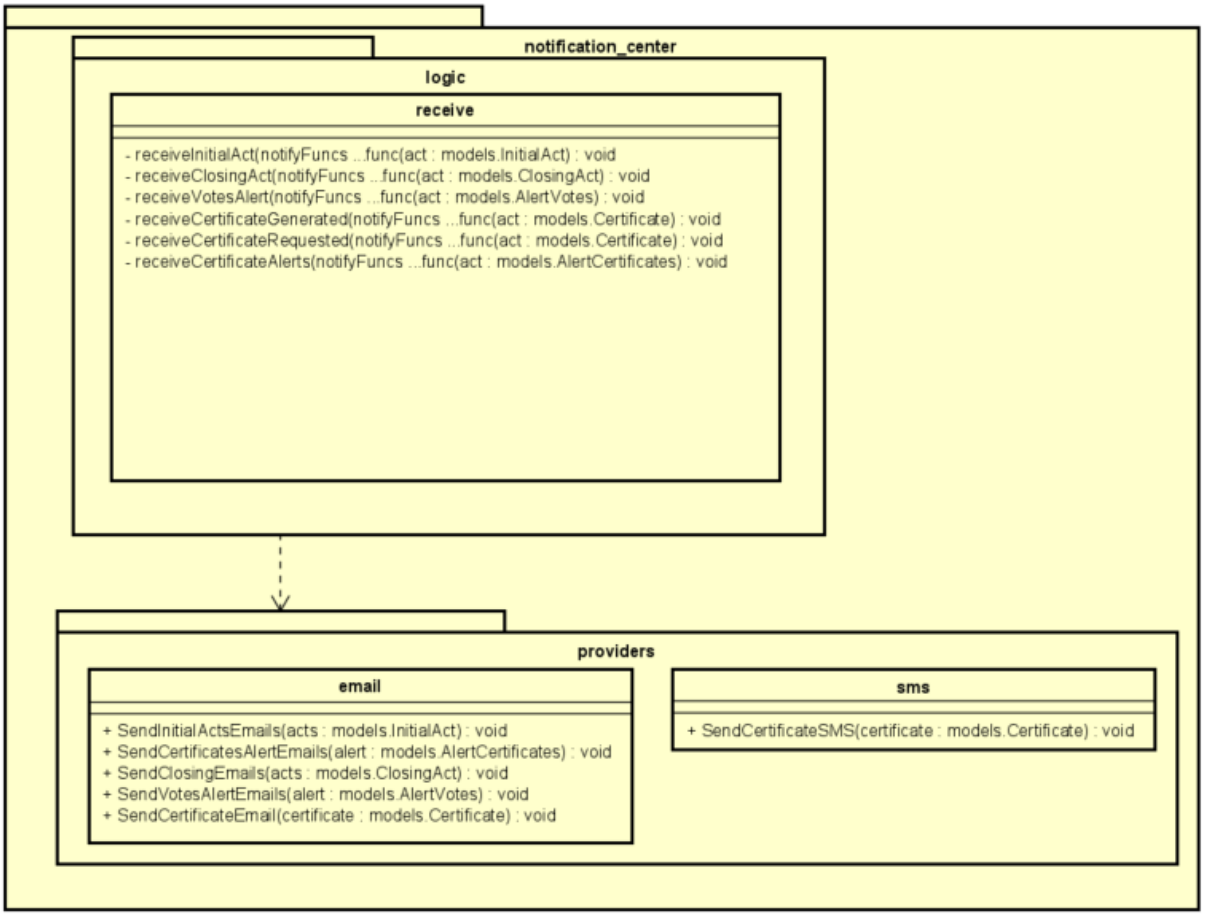
Utilizamos el patrón pipes and filters el cual se va a más en detalle en la vista de componentes. Sin embargo, queríamos destacar la facilidad de implementar nuevos filtros ya que requiere escribir código. En resumen, cada filtro es una función que recibe por parámetro: la data de tipo any y más parámetros los cuales se leen desde un archivo de configuración.
Además el pipes and filter es una librería por lo cual se puede importar en toda la solución. La única restricción es que los nuevos filtros tengan la misma firma de función. Como esta librería se iba a usar en varios servicios se buscó que fuera fácil de modificar.

own_logger

Creamos un proyecto de go el es importado por el resto de los módulos y utilizado para logear eventos y actividades de los usuarios de cada uno de los procesos. Como es un único módulo para todos los otros proyectos, se puede modificar es un único lugar fácilmente. Ej. si se quiere guardar los logs en una base de datos en vez de en un archivo, se puede realizar el cambio y que aplique a todos los proyectos fácilmente, mejorando la mantenibilidad del sistema. También se podría haber usado versionado del own_logger así si se realiza un cambio en el futuro, cada proyecto puede elegir que versión usar. Otra ventaja que tiene el uso de logs es para seguridad y disponibilidad lo cual vamos a hablar más adelante en la vista de componentes.

notification_center

Notification center es el encargado de recibir mensajes y enviarlos a través de sms, mail y cualquier otro tipo de proveedor que se puede agregar en un futuro. Para esto se definieron funciones por cada tipo de mensajes, cada función recibe una lista de funciones, y al momento de llegar un mensaje se ejecuta cada función para notificar a los respectivos proveedores.
Los paquetes de proveedores son los responsables de comunicarse con las APIs externas de cada proveedor, actualmente nos faltó emular un proveedor usando una api externa, de todos modos no cambia la arquitectura.



[3]Figura 3.3: Diagrama de Usos

[3] Diagrama encontrado en Documentation/Diagrams/Class_Notification_Center.svg

3.1.2 Vista de Uso

3.1.2.1 Representación primaria

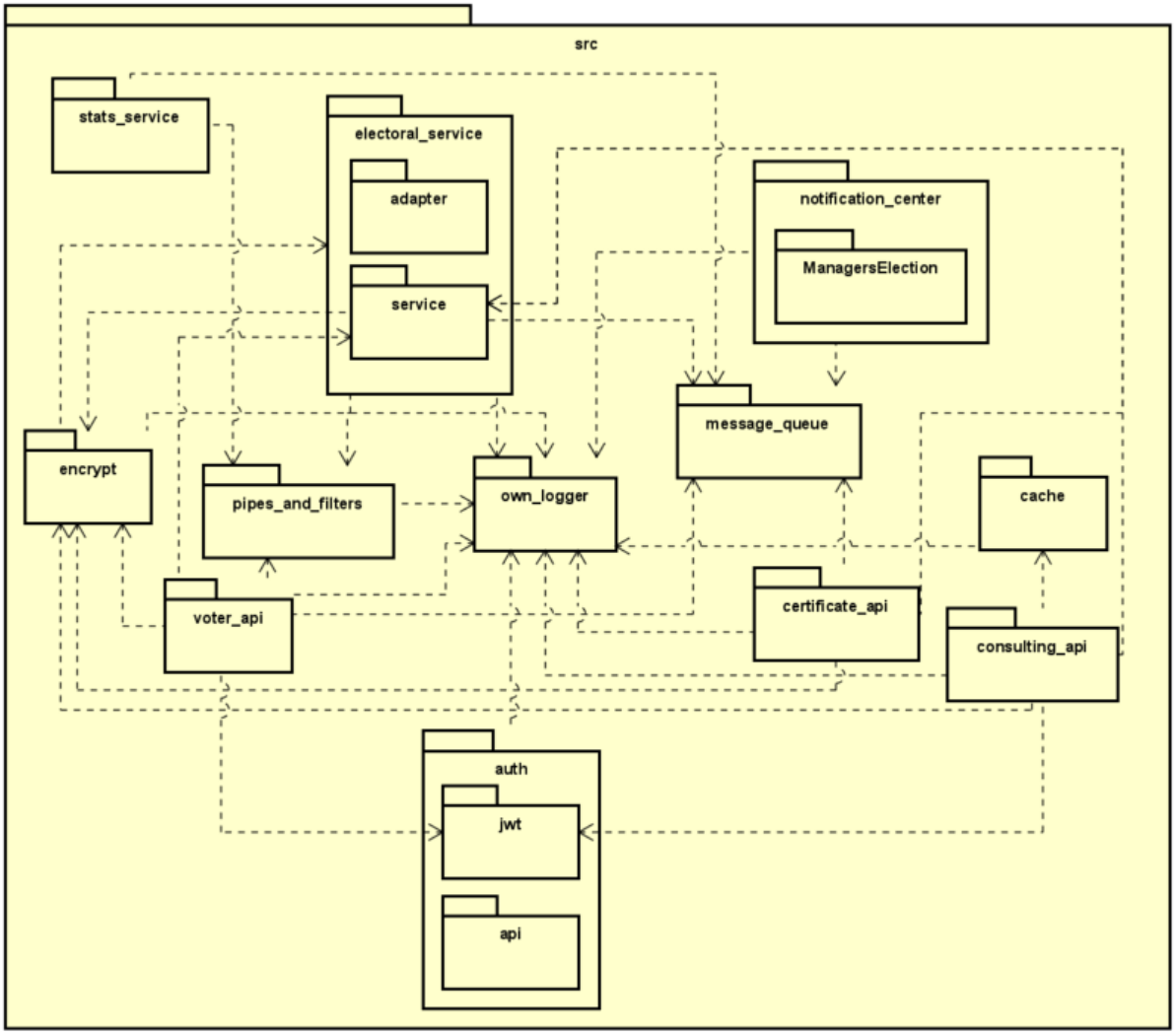


Figura 3.4: Diagrama de Usos.[4]

3.1.2.2 Decisiones de diseño

Modificabilidad:

Se decidió crear librerías internas como son own_logger, pipe_and_filters, auth, encrypt, cache, message_queue, la cuáles fueron creadas para ser usadas y salvó algunas excepciones que no "usen" a nadie, o no dependen de otro servicio. Esto implica diferentes tácticas, como reducir el tamaño de un módulo e incrementar la coherencia semántica ya que los librería cumplen un objetivo claro que puede ser brindado a varios servicios.

También se podría considerar message_queue como un intermediario, ya que desacopla por ejemplo voter_api con certificate_api para que no dependan directamente de la información que proporciona cada uno de estos servicios, sino que cada uno provea su información a su

[4] Diagrama encontrado en Documentation/Diagrams/Uses.svg

debido tiempo y puedan seguir intercambiando datos entre sí. Esto último tiene que ver con el Requerimiento funcional 2, donde luego de validar y procesar el voto para obtener una constancia se debe comunicar con un servicio especializado en esto. A su vez certificate_api utiliza notification_center donde quisimos centralizar nuestro proveedores para que sean fáciles de cambiar a futuro.

Seguridad:
En este diagrama se puede observar también cómo se da el intercambio para el Requerimiento 3 sobre autenticación. Podemos tomar consulting_api que se especializa en tomar las consultas de los usuarios con credenciales específicas y se comunica con jwt directamente, nuestra librería para chequear que el token corresponda a la solicitud que esté siendo procesada.

3.1.3 Vista de Layers

3.1.3.1 Representación primaria

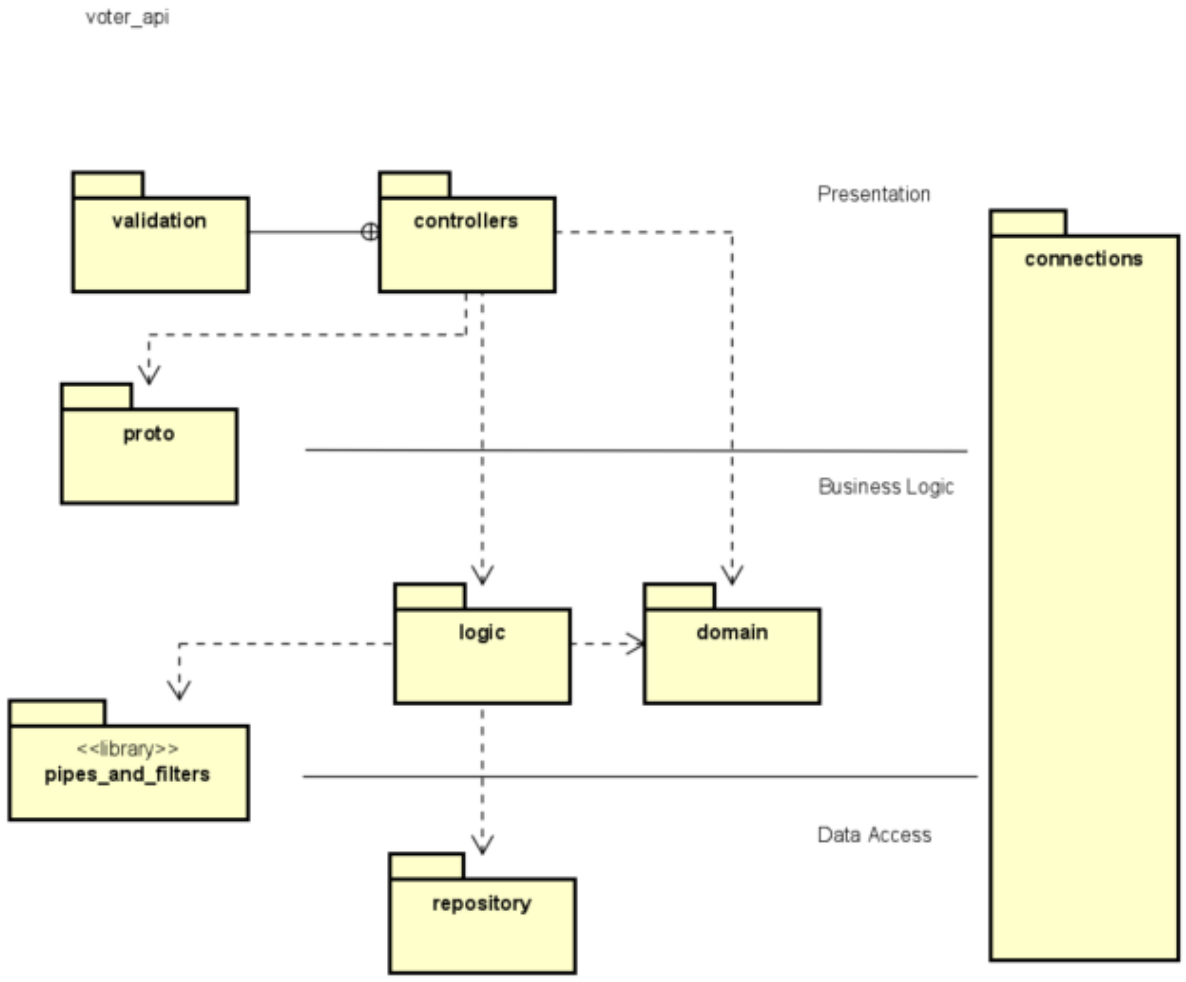


Figura 3.5: Diagrama de Layers voter_api[5]

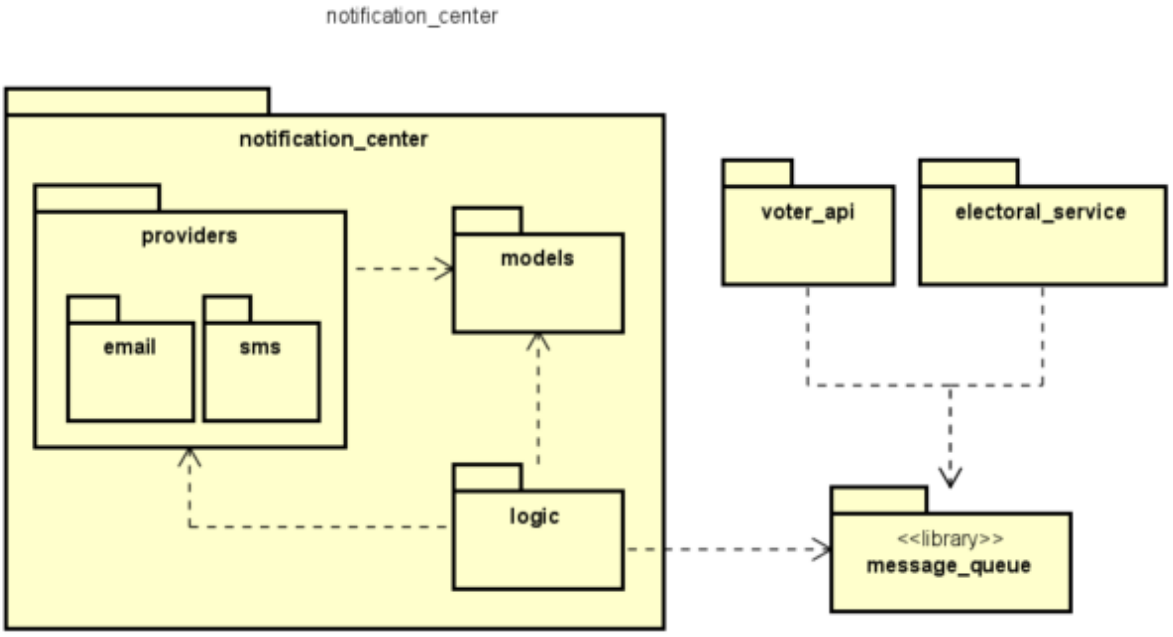


Figura 3.6: Diagrama de Layers/Modulos notification_center[6]

[5] Diagrama encontrado en Documentation/Diagrams/LayersVoterApi.svg
[6] Diagrama encontrado en Documentation/Diagrams/Notification.svg

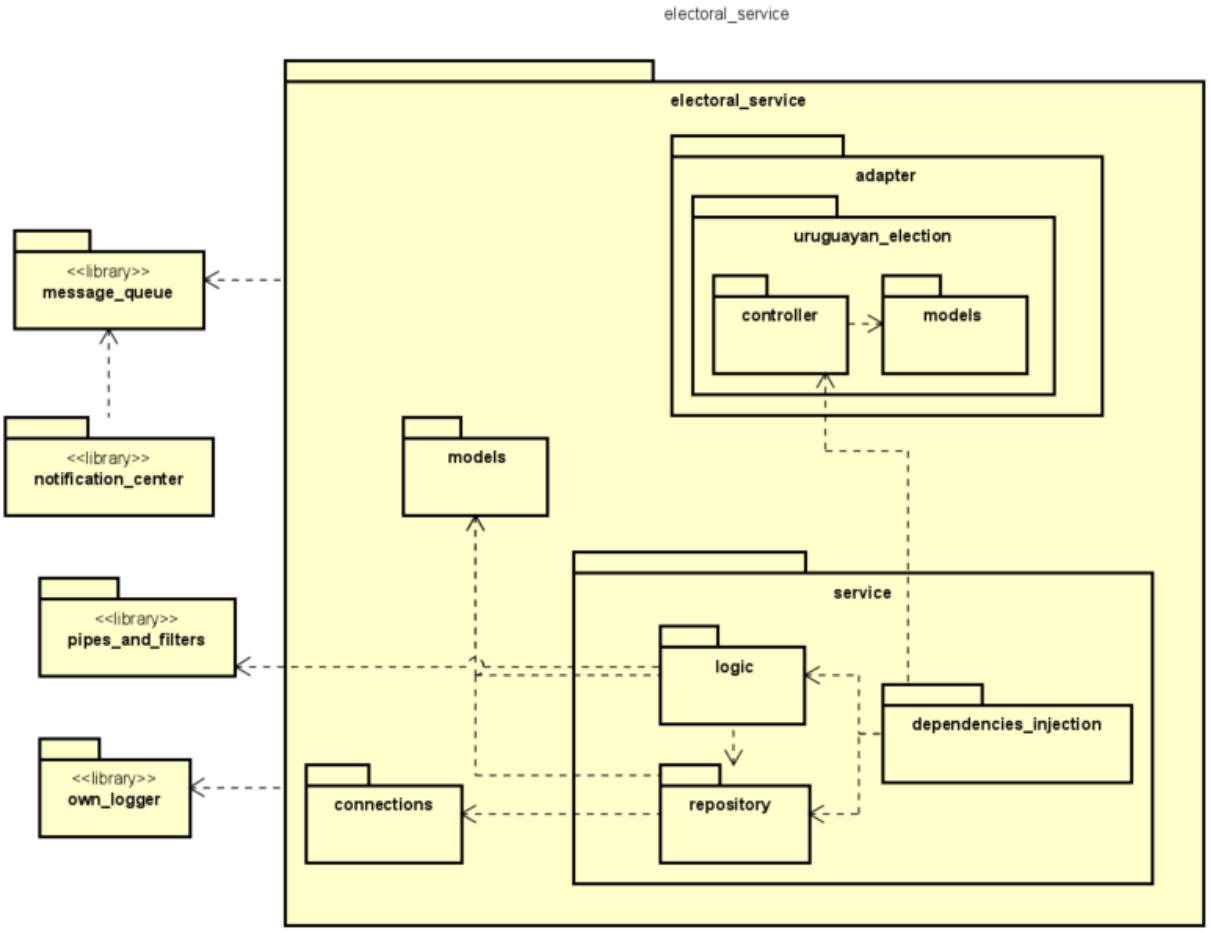


Figura 3.7: Diagrama de Layers electoral_service[7]

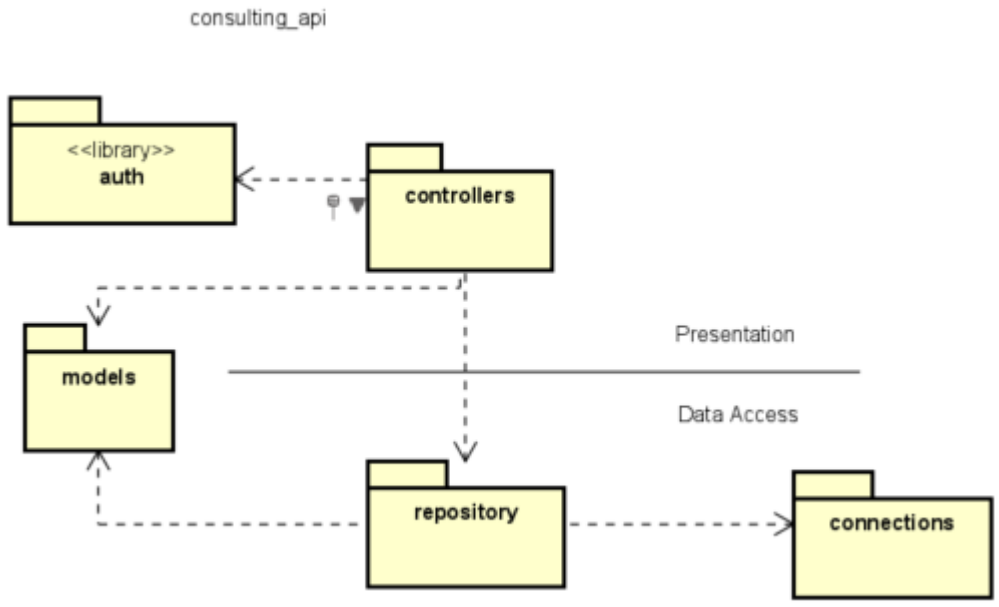


Figura 3.8: Diagrama de Layers consulting_api[8]

[7] Diagrama encontrado en Documentation/Diagrams/ElectoralService.svg
[8] Diagrama encontrado en Documentation/Diagrams/ConsultingApi.svg

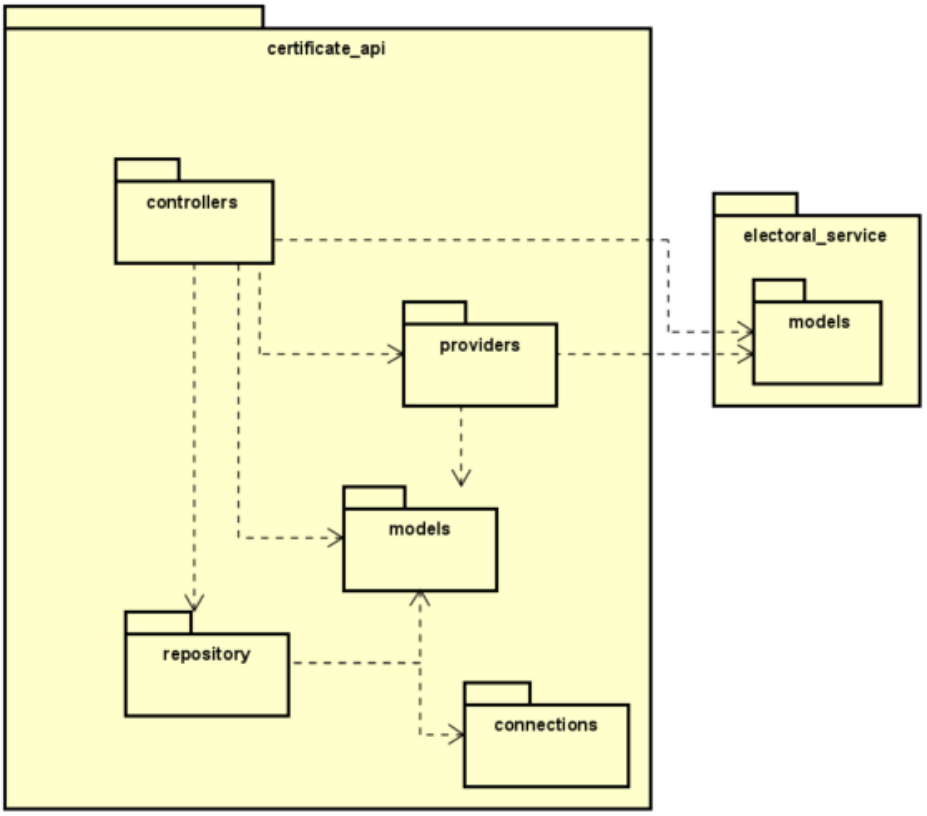


Figura 3.9: Diagrama de Layers/Módulos certificate_api[9]

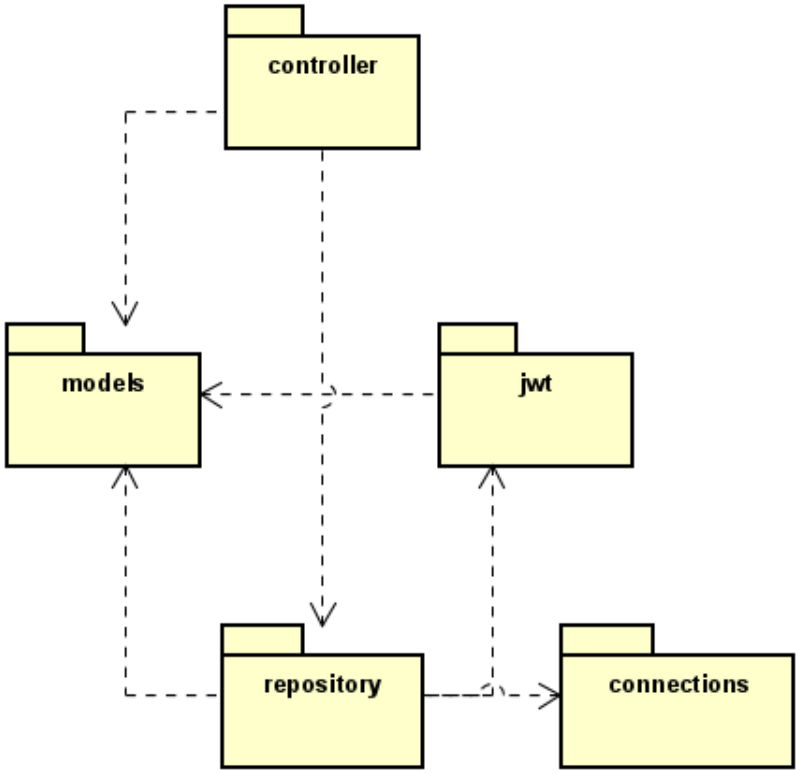


Figura 3.10: Diagrama de Layers/Módulos auth[10]

[9] Diagrama encontrado en Documentation/Diagrams/CertificateApi.svg
[10] Diagrama encontrado en Documentation/Diagrams/Auth.svg

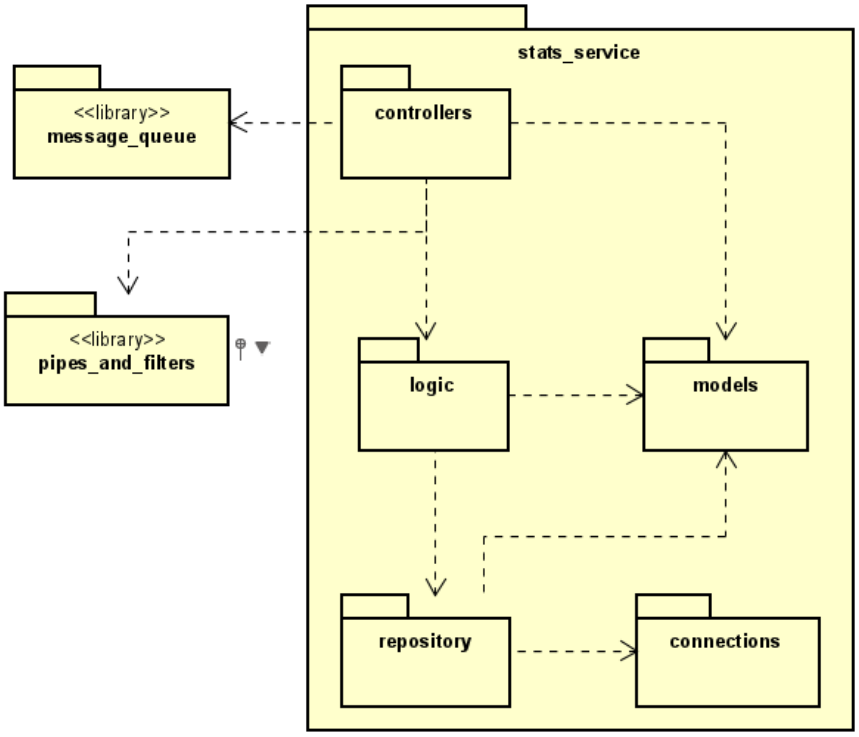


Figura 3.11: Diagrama de Layers/Módulos stats_service[11]

3.1.3.2 Decisiones de diseño

Nuestro diseño de los servicios que exponen APIs se trató de mantener constante durante todo el sistema. Nos basamos en un diseño por capas, donde las controladoras exponen una interfaz de comunicación, la lógica procesa la información y el repositorio se encarga de guardar y traer la información de la base de datos. Decidimos incluir en cada uno de los módulos un archivo especializado a las conexiones. En nuestro caso se repite la información de la misma para conectarse a mongo por ejemplo, pero lo implementamos de esta manera porque creemos que si se quisiera tener diferentes instancias de mongo de esta manera sería bastante sencillo.

Tomamos estas decisiones para crear un mejorar la mantenibilidad del sistema ya que los cambios en capas superiores no afectan las capas inferiores. Además cambios en capas no adyacentes tampoco provocan cambios en la misma capa.

3.1.4 Catálogo de elementos

Elemento	Responsabilidades

[11] Diagrama encontrado en Documentation/Diagrams/Stats.svg

src/certificate_api	WebApi que se encarga de recibir request de constancias de voto, crear constancias y almacenar ambas. Enviar asincrónicamente los certificados por sms cuando se vota y por email cuando se solicita
src/consulting_api	WebApi que se encarga de recibir request de todo tipo de consultas sobre la elección, configuracion, horas más votadas, votos por región o por circuito, si determinado usuario voto, etc
src/voter_api	Api mediante grpc que se encarga registrar los votos que realiza un votante y encolar los certificados
src/notification_center	Library que se encarga de manejar las notificaciones de alertas y actas escuchando una cola de mensajes
src/auth	Este módulo contiene las herramientas necesarias para la autenticación de usuario
src/auth/api	WebApi que se encarga de recibir requests de registro y login de usuario
src/auth/jwt	Librería de jwt que crea y verifica el token de autenticación
src/electoral_service	Módulo encargado de recibir los datos y configuraciones de una elección
src/electoral_service/adapter	Se encarga de poder adaptarse a cualquier elección de cualquier país
src/electoral_service/service	Contiene la lógica sobre las validaciones y filtros sobre una elección y el almacenamiento
src/encrypt	Módulo encargado de encriptar la información de los votos y votantes
src/pipes_and_filters	Módulo encargado de manejar las pipelines de validaciones y transformaciones (filtros)
src/own_logger	Se encarga de los logs de info, warning y errores del sistema
src/cache	Módulo que utiliza redis como caché para almacenar algunos datos de la elección para más rapido acceso

src/message_queue	Módulo para cola de mensajes
src/stats_service	Actualiza las estadísticas sobre las coberturas de votos por departamentos y circuitos según la cantidad de votantes.

Nota: Módulo lo usamos como sinónimo de librería

3.1.5 Interfaces

Interfaz:	
Paquete que la implementa:	
Servicio	**Descripción**
voter_api	Se utilizó gRPC para exponer esta interfaz. Esto se realizó como modo de investigación para ver si mejoraba la performance pero trajo algunos problemas de cantidad de conexiones soportadas al mismo tiempo. Parametrizamos un límite de go routines para procesar en batches asíncronos los votos.
certificate_api	Esta interfaz además de exponer una api para solicitar un certificado, se consume por medio de cola de mensaje certificados luego de que se haya procesado el voto y de esta manera enviar un SMS al votante o email según corresponda.

3.1.6 Comportamiento

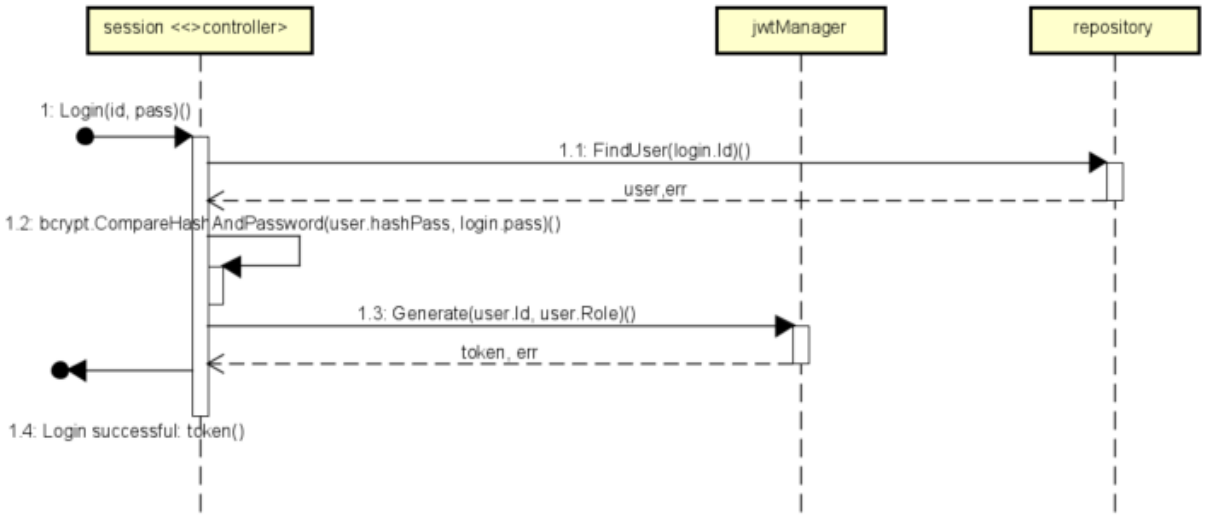


Figura 3.12: Diagrama de secuencia login[12]

En este diagrama se puede observar los pasos que se siguen cuando se quiere obtener un token para determinado usuario con un rol específico, cuando sus credenciales ya fueron registradas en la base de datos.

[12] Diagrama encontrado en Documentation/Diagrams/Login.svg

3.2 Vistas de Componentes y conectores

3.2.1 Representación primaria

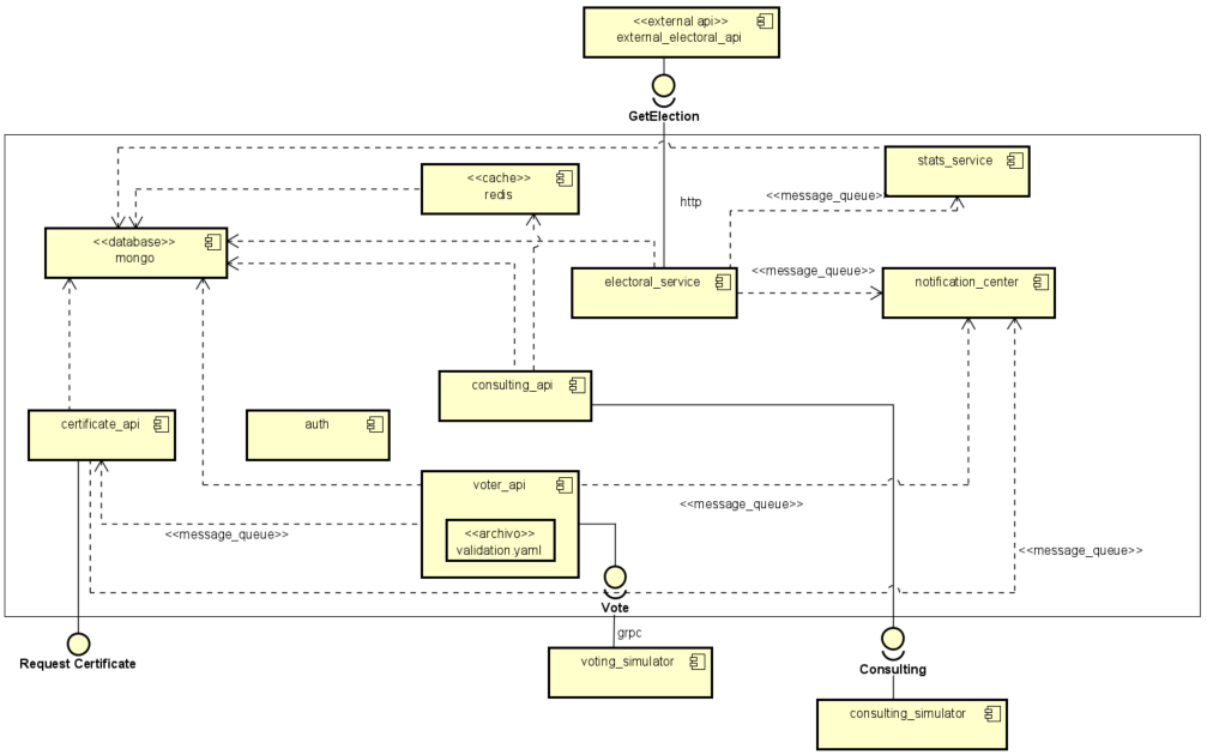


Figura 3.13: Diagrama de componentes y conectores[13]

3.2.2 Catálogo de elementos

Componente/conector	Tipo	Descripción
voter_api	Componente	Servicio encargado de recibir, validar y procesar cada voto del sistema y actualizar la información de las base de datos, como por ejemplo la cobertura de votos por departamento cada vez que se vota, o el resultado de la elección. Antes de finalizar envía el certificado a una cola de mensajes.
certificate_api	Componente	Servicio que consume los certificados y los envía por SMS al votante. También expone una API para realizar una solicitud de certificado para ser enviada por email.
auth	Componente	Encargado de generar token, exponiendo APIs de registro y Logeo al sistema. Funciona como librería para verificar el token para otros servicios que requieran autenticación.
consulting_api	Componente	Expone endpoints para realizar consultas de la información que tiene almacenada el sistema. Algunos ejemplos son: horas más populares de votación, resultado de elección en tiempo

[13] Diagrama encontrado en Documentation/Diagrams/C&C.svg

		real, configuración de elección, etc. Varias de estas requieren un token con permisos correspondientes.
electoral_service	Componente	Es el encargado de manejar las elecciones, su comienzo y fin al obtener la información de un servicio externo. También maneja los partidos políticos con sus candidatos y los votantes para determinada elección. Utiliza un modelo que consideramos básico, adaptándolo a cada país.
redis	Componente	Cumple el propósito de mejorar la performance para distintas consultas hacia la base de datos que pueden ser frecuentes guardando la información en memoria.
mongo	Componente	Bases de datos donde se guarda la información del sistema en general. Existen varias bases de datos siguiendo el AC de seguridad, pero para mejorar aún más este aspecto se debería tener varias instancias de mongo separadas.
stats_service	Componente	Consumidor de una cola de mensaje al cuál publican electoral_service y voter_api. La primera setea la capacidad por departamento y circuito para cada grupo predefinido de edad y sexo. Voter_api manda para actualizar los votos en cada grupo. Cabe destacar que este grupo son modificables mediante un archivo.
notification_center	Componente	Consumidor principal de las actas publicadas por electoral_service y alertas que son enviadas si ocurren ciertos eventos específicos, como por ejemplo que un votante vote muchas veces.
RabbitMQ	Componente	Instancia que administra todas las colas usadas para la comunicación asincrónica entre componentes
grpc	Conector	Usado en para enviar los votos a la voter_api, por la posible mejora de performance que tenga
http/api rest	Conector	Usado para la comunicación con cualquier sistema externo (cliente y servicios electoral), excluyendo la comunicación de envio de votos
amqp	Conector	Protocolo usado para comunicarse con la instancia de RabbitMQ

3.2.3 Comportamiento

En los siguientes diagramas se muestra las comunicaciones del certificate_api y los dos casos que existen:

1. Recibir un voto para guardar en la base de datos luego de que voter_api lo haya procesado y validado.

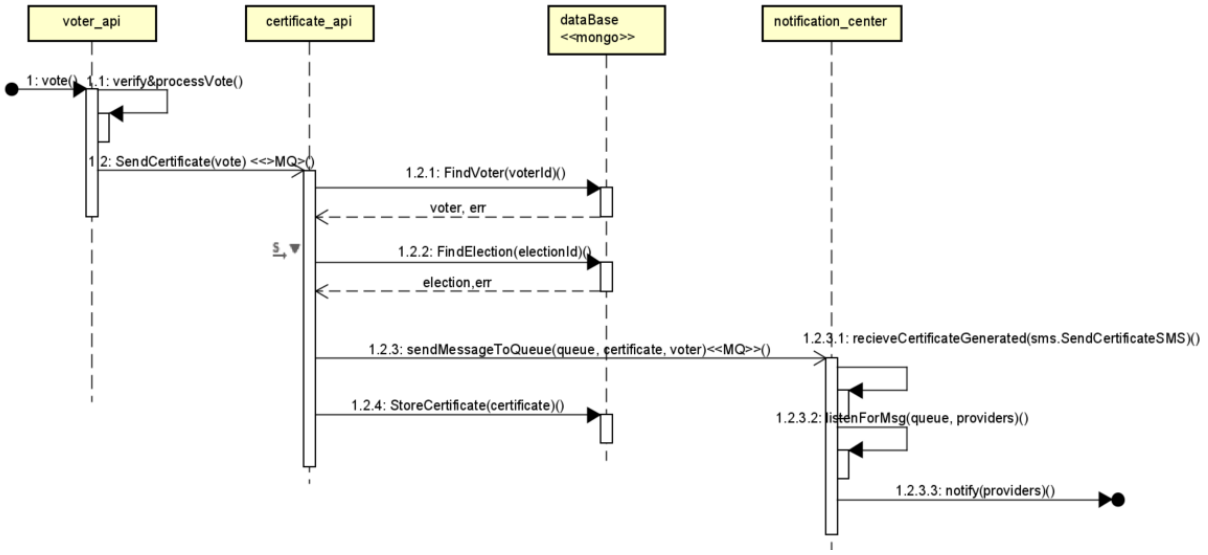


Figura 3.14: Diagrama de secuencia mensaje sms[14]

2. Recibe un pedido por medio de una API Rest para obtener un certificado por email. Este guarda el pedido en una base de datos y luego manda a otra cola de mensaje para que maneje el envío por email.

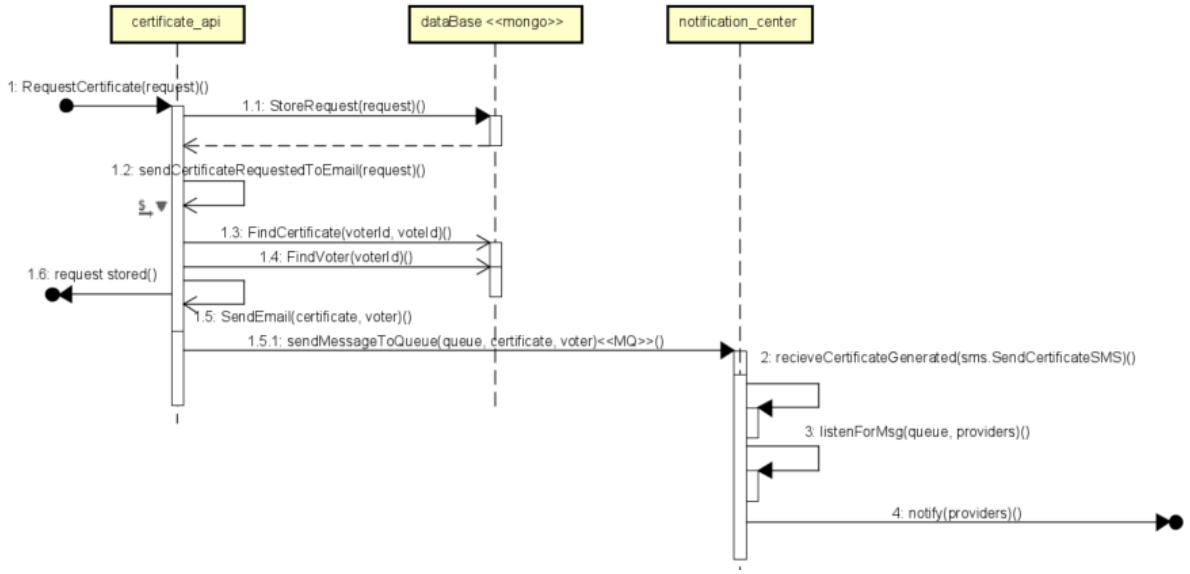


Figura 3.15: Diagrama de secuencia certificado[15]

[14] Diagrama encontrado en Documentation/Diagrams/CertificateMQ.svg
[15] Diagrama encontrado en Documentation/Diagrams/CertificateAPI.svg

3.2.4 Decisiones de diseño

Grpc y votos

Uno de los puntos más importantes es la performance. Esto se planteó como objetivo en un principio para verlo reflejado en la funcionalidad más importante que es la votación. El equipo decidió investigar una nueva tecnología como lo es gRPC como protocolo de transporte para observar si podía ayudar a disminuir los tiempos de comunicación. Llegamos a la conclusión de que no es el mejor escenario para usar esta tecnología dado a que al crear una nueva conexión entre ambas partes y ser muy orientado a tcp, soportaba hasta un máximo no muy alto de conexiones simultáneas y rechazaba nuevas conexiones. Se hicieron varias pruebas limitando la cantidad de coroutines activas y optimizando el procesamiento de la llamada grpc, pero el resultado no fue tan bueno como en otros servicios donde usamos http apis.

A pesar de esto, encontramos un límite de 330 conexiones simultáneamente, que pueden procesar votos en en menos de 1 segundo, pero al probar 10000 votos simultáneos se pierden pocas conexiones de gRPC que quedan sin procesar. Creemos que si contáramos con máquinas más potentes, estos números podrían ser mayores, así que lo consideramos en parte una limitación de hardware.

En nuestra solución implementamos la táctica de “Limit event response”, donde utilizamos un canal al recibir los votos y que procese cuando alguna go routine queda libre.
También realizamos muchos procesos en paralelo, como el actualizar estadísticas y mandar a una cola de mensaje, aquí utilizamos “Introduce concurrency.”

Cache

Para mejorar la performance, mantenemos múltiples copias de datos al utilizar un caché para no tener que acceder a la base de datos todo el tiempo.
Se utiliza en la consulting_api donde puede suceder que se pidan datos que ya se han consultado recientemente. Para las consultas que traen más datos la diferencia de tiempo llegaba a ser 20 más rápido desde el cache (ej. 2ms vs 20ms)
Se probó utilizar caché para la voter_api sin embargo las diferencias de tiempo no eran significativas (corriendo localmente), no mejoraron la performance y tampoco había tantos datos que se consultaban repetidas veces. En un sistema con mayor latencia entre la base de datos y la voter_api se podría implementar el cache.
Además, para mejorar la disponibilidad, como el cache no es un proceso crítico de la aplicación, se diseñó para que la aplicación funcioné aunque no esté disponible el redis (deshabilitando el cache). Se sigue la táctica Retry, reintentando conectarse al cache cada minuto. Mientras no esté disponible las consultas al cache automáticamente devuelve error, esto mejora la performance ya que no intenta conectarse con el redis todo el tiempo.

El siguiente diagrama muestra cómo se realice una consulta que no está en el cache, se va a buscar a la base de datos y se guarda en el cache para que el próximo la obtenga desde ahí.

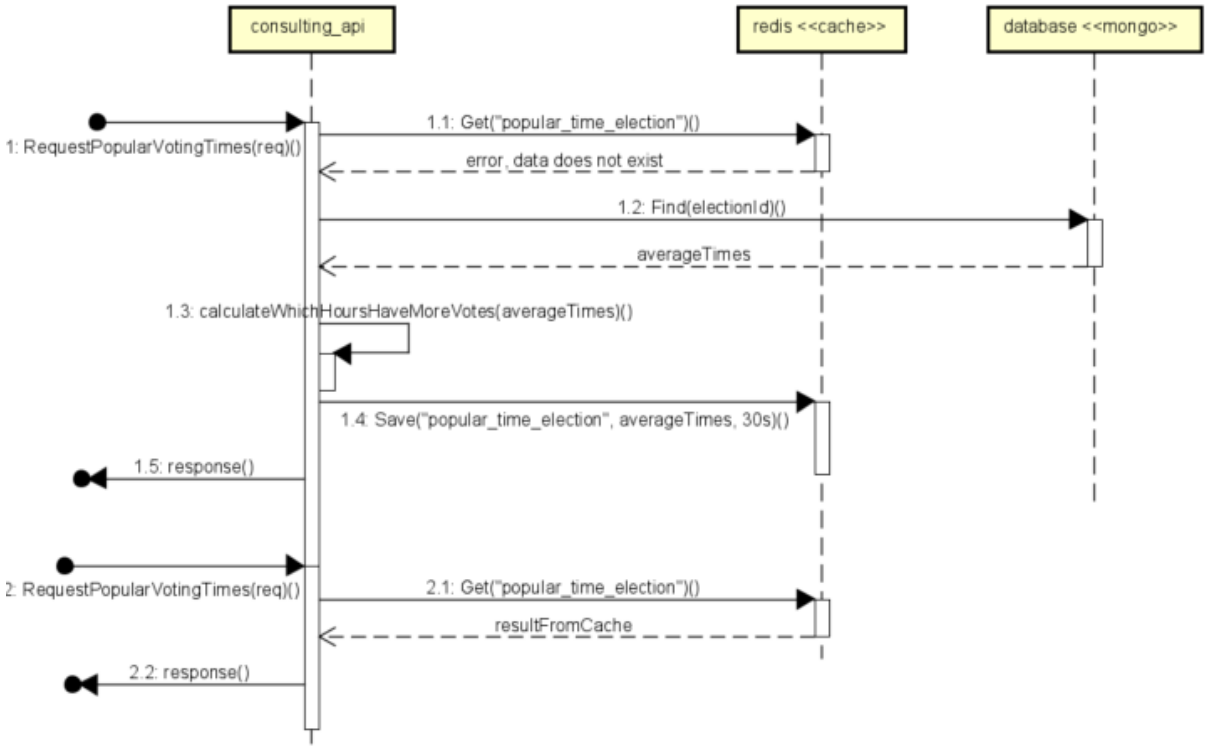


Figura 3.16: Diagrama de secuencia consultas[16]

Ping

Seguimos la táctica de disponibilidad y los distintos sistemas hacen un ping a la base de datos para asegurarnos que la conexión haya sido exitosa.

Exception handling

A lo largo del sistema se utiliza esta táctica para chequear los errores devolviéndole al usuario un mensaje y código en caso que corresponda. Pocas veces finaliza un proceso al ocurrir un error.

Encriptación y verificación

Al tratarse de un sistema de votación online, la seguridad fue otro de los puntos claves a tratar.

Se implementaron varias técnicas para poder cumplir el objetivo como por ejemplo verificar la integridad del mensaje al verificar la firma digital del voto con claves pública y privada. También autenticamos y autorizamos los actores que tengan un rol específico y envíen un token correcto al logearse al sistema. Esto nos permite comprobar la auditoría, cualquier acceso al sistema es logueado, pudiendo ver quien utilizó el sistema (siguiendo la táctica de Identificar actores) y que acciones tomó.

[16] Diagrama encontrado en Documentation/Diagrams/Cache.svg

Utilizamos la táctica de encriptar los datos para que si hay un ataque exitoso al sistema, los datos de los usuarios en la base de datos y durante el viaje estén protegidos.

Instancias de bases de datos

Nuestro objetivo fue tener bases de datos distintas para que la información esté distribuida, favoreciendo el limitar la exposición de los datos, por ejemplo los votantes por un lado y los votos por otro. Dado que nuestras máquinas no tienen demasiados recursos, se corren todas las base de datos bajo un mismo servidor mongo, lo cuál fuertemente recomendamos se cambie si se llegará a desplegar. En caso de tener los recursos necesarios estas serían las bases de datos de mongo recomendables:

- Por seguridad: Una única instancia de Mongodb para servicio *auth* encargado de guardar las contraseñas hasheadas. A esa instancia solo puede acceder dicho servicio.
- Por performance: Separar las actuales bases de datos en distintas instancias de mongo, donde cada instancia tendría una sola base de datos, serían: votes, statistics, election. Las tres bases de datos tienen distintas responsabilidades y se accede a la información con distinta frecuencia por lo tanto es importante que las consultas a la bd de estadísticas no afecten la escritura de nuevos votos.

Archivos .env

Todos los procesos cuentan con sus propios archivos .env donde se pueden modificar valores que cambian el comportamiento del código al momento de ejecutarse. Esto sigue la táctica de Defer binding.
Por ejemplo, las URI para las conexión de la base de datos se encuentran en los .env al igual que valores que limitan la cantidad de gorutinas de la voter_api para asegurarse que poder controlar mejor la performance. También se incluyen los nombres de las colecciones y bases de datos.

Emuladores

Si bien no se muestran en los diagramas, para probar la funcionalidad del sistema (principalmente la performance) se crearon sistemas que emulan una posible carga real. Usando la táctica de Sandboxing de Testing, pudimos llegar a los valores que dijimos anteriormente.

Patron federated Identity

Delega la autenticación a un proveedor de identidad externo. Esto puede simplificar el desarrollo, minimizar los requisitos de administración de usuarios y mejorar la experiencia de usuario de la aplicación. Pero el atributo de calidad más relevante es el de seguridad, con este patrón el sistema de autentificación se puede despleguar en un ambiente más seguro que el

resto. Además, al evitar que cada sistema realice la autenticación, es más fácil ver los logs y auditar el sistema.
El servicio de auth no guarda las contraseñas en texto plano, sino que las hashea y cuando manda la contraseña un usuario, compara los hashea de la base de datos y del usuario. De esta manera si hay una falla de seguridad y alguien accede a la base de datos, no pueden saber las contraseñas.
Actualmente para votar no se pide un token, ya que como se firma y encripta los datos, el simple hecho de validar la firma ya permite autenticar al votante.

Uso de JWT

Para cumplir con el requerimiento no funcional de seguridad, elegimos utilizar un par de llaves público-privada. Todas las consultas que se pueden realizar sobre el sistema de votaciones, datos de la elección, viajan con un Bearer token (implementado con jwt). Este token está cifrado con una llave privada conocida sólo por autenticación y la api consultora. De esta manera, se puede verificar el rol del usuario que quiere hacer la consulta, sin exponer datos sensibles a quienes no tienen autorización.

Single-tenancy

El sistema fue diseñado para funcionar de manera single-tenant, es decir, no va a haber elecciones de varios países en el mismo sistema, para cada país se tiene que hacer un despliegue. Esto mejora la seguridad y la performance ya que hay menos usuarios usando el sistema y menos usuarios con roles con varios permisos. Esto permitiría limitar más el acceso al sistema mejorando la seguridad.

Indices

Se utilizaron índices para las base de datos de mongo, siguiendo la táctica de aumentar la eficiencia. Los índices hay que configurarlos cada vez que se despliega el sistema.

Pipes and Filters

Se utilizó el patrón pipes and filters para facilitar la modificabilidad de las validaciones a lo largo del sistema, se puede cambiar los filtros a utilizar, los parámetros de los mismo y cuantas veces reintentar en caso de vaya, todo utilizando un archivo .yaml. Dicho archivo es configurable al momento de despliegue siguiendo la táctica de defer binding.
Ya que solo se utiliza el patrón para validaciones o filtros que devuelven errores, la ejecución de los filtros es asincrónica utilizando goroutines y siguiendo el patrón fan out. Los filtros devuelven errores los cuales son juntados usando un fan it.

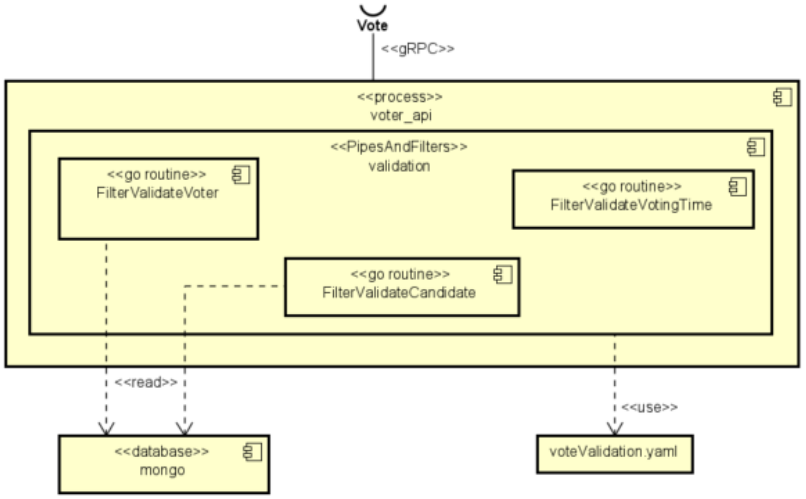


Figura 3.17: Diagrama de Pipes and Filters[17]
Nota: En el diagrama no se representan todos los filtros que hay

Publish subscribe

A lo largo del sistema se utilizaron varias colas de mensajes siguiendo el patrón publish-subscribe. Su principal objetivo es permitirnos la comunicación asincrónica entre distintos servicios de nuestra solución.
Mejora la performance porque nos permite procesar datos fuera de servicios críticos como voter_api. A diferencia de métodos sincrónicos, el procesamiento de los datos se realiza a medida que puede el consumidor limitando la cantidad de mensajes entrantes (táctica de Manage event rate). Por lo tanto en vez de tener un único sistema que sea capaz de procesar todos los datos, podemos tener dos sistemas uno más potente para los datos que requieran respuesta más rápida.
Otra ventaja a nivel de disponibilidad es que si algún consumidor se cae, se pueden seguir publicando mensajes y cuando vuelva a funcionar el consumidor, se procesan todos los mensajes. Además para la testabilidad, no es necesario tener todos los servicios ejecutándose para probarlos.

[17] Diagrama encontrado en Documentation/Diagrams/PipesAndFilters.svg

3.3 Vistas de Asignación

3.3.1 Vista de Despliegue

3.3.1.1 Representación primaria

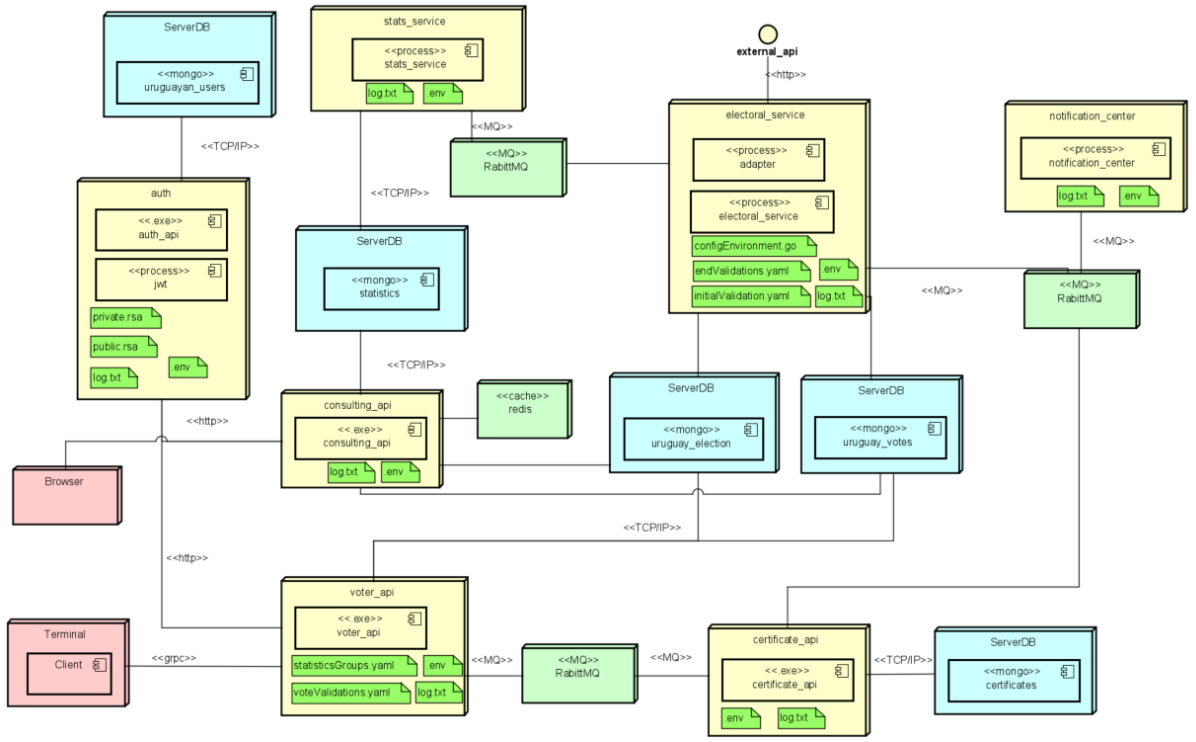


Figura 3.18: Diagrama de Despliegue[18]

Notas: Si bien nosotros desplegamos cada uno de los servicios en nuestra única computadora, se recomienda desplegar en contenedores o servidores diferentes para mejorar la performance, seguridad y disponibilidad. Cada servicio cuenta con su propio.
Dependiendo de cómo hagan el despliegue de las bases de datos y de las colas de mensajes pueden tener una o varias instancias de cada servicio, en el diagrama se muestran separadas para simplificar pero ServerDB y RabbitMQ pueden ser el mismo servidor. No es necesario cambiar el código para realizar estos cambios se puede modificar el .env de cada servicio.

Diagrama encontrado en Documentation/Diagrams/Deployment.svg

3.3.1.2 Catálogo de elementos

[18] Diagrama encontrado en Documentation/Diagrams/C&C.svg

Nodo	Descripción
auth	Sistema que provee los endpoints para la autenticación. Generación y verificación de token
consulting_api	Sistema que provee los endpoint para las consultas sobre datos de la elección
voter_api	Sistema que provee endpoint para poder realizar el voto
certificate_api	Sistema que provee los endpoint para la solicitud de un certificado
notification_center	Sistema encargado de las notificaciones
electoral_service	Sistema encargado de recibir y validar elecciones
stats_service	
mongoDB	Motor de base de datos noSQL
redis	Motor de base de datos en memoria que se usa como cache
rabbitMQ	Sistema encargado de colas de mensajería