Universidad ORT Uruguay

Obligatorio

Arquitectura de Sistemas

Docente: Gerardo Quintana

Agustín Ferrari 240503

Joaquín Meehroff 247096

Graziano Pascale 186821

Link al repositorio:

https://github.com/ORTArqSoft/240503-247096-186821

2022

Declaraciones de autoría

Nosotros, Agustín Ferrari, Joaquín Meerhoff y Graziano Pascale declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano.
Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizamos Arquitectura de Software;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuído por otros, y qué fue contribuído por nosotros;

Ninguna parte de este trabajo ha sido publicada previamente a su entrega

Introducción

Éste trabajo fue realizado para demostrar los conceptos aprendidos durante el curso de Arquitectura en Sistemas de la carrera de Ingeniería en Sistemas, Universidad ORT del Uruguay.

La cátedra propuso el desafío de crear una plataforma de voto electrónico, teniendo en cuenta las distintas implicancias que éste presenta.

La solución diseñada fue producto únicamente del trabajo de los integrantes del equipo.

Propósito

El propósito del presente documento es proveer una especificación completa de la arquitectura de la plataforma de voto electrónica diseñada, AppEv.

Antecedentes

Propósito del sistema

AppEv es una plataforma de voto electrónico que permite llevar a cabo elecciones para el usuario principal del sistema, el ente regulador de las elecciones. La plataforma busca cubrir las necesidades que el ente pueda tener y adaptarse a los distintos escenarios que éste proponga, con una sencilla configuración. La solución diseñada asegura la integridad y seguridad del voto, así como la privacidad del votante. A su vez, se busca una alta disponibilidad de la plataforma, de forma que esté abierta a su uso cuando el usuario los disponga, respondiendo rápidamente a sus pedidos.

El sistema periódicamente cargará nuevas elecciones, que una vez iniciadas, sus votantes podrán votar a su candidato preferido. Al finalizar la elección, se suspende la votación, y su resultado es enviado a un conjunto de usuarios elegidos por la Autoridad Electoral.

El sistema tiene distintos tipos de usuarios: Autoridad Electoral (ente regulador de elecciones), Agente de Consulta, Consultor y Votante. Éstos podrán realizar distintos tipos de consultas sobre el estado de las elecciones, como su resultado hasta el momento, los horarios con mayor frecuencia de votos, entre otras consultas. Los Consultores podrán realizar configuraciones tanto en tiempo de ejecución como de desarrollo para adaptar la plataforma a las necesidades de la Autoridad Electoral.

Nota: Es importante destacar que como cualquier proyecto real, durante el desarrollo del obligatorio, el equipo tuvo que tomar decisiones importantes respecto a la administración de sus recursos, en algunos casos optando por reducir alcances o implementar versiones minimalistas que permitieran verificar al menos la viabilidad de las ideas. Estos casos se encontrarán claramente expresados en la sección correspondiente.

Requerimientos significativos de arquitectura

Resumen de Requerimientos Funcionales

A continuación detallaremos el principal alcance de la solución propuesta. Ideamos éstos requerimientos funcionales que creemos que describen la mayor parte de la funcionalidad del sistema.

1. Inicio y cierre de elección

El sistema es capaz de agregar, iniciar y cerrar elecciones obteniéndolas a través de la API Rest de la Autoridad Electoral con la que se esté trabajando. Una vez inicie la elección se permiten realizar votos hasta que la misma termine.

Cuando se obtiene una nueva elección el sistema se encarga de hacer validaciones sobre los datos de la misma y guardarla en caso de ser válida.

Tanto en el inicio como el cierre de elección, se envían actas a una lista de usuarios previamente definidas.

2. Emisión del voto

El sistema permite a los votantes habilitados, votar en cada elección en las que estén habilitados siempre y cuando la elección esté activa y se cumplan con los siguientes puntos:

- El votante está habilitado para votar en la elección
- El votante vota en el circuito al que pertenece
- El votante votó solo un candidato
- El candidato es uno de los habilitados en la elección
- El votante votó una única vez si la modalidad de voto es única

El voto es contabilizado al candidato especificado y no se guarda relación entre voto y votante, salvo en elecciones de modalidad de voto repetido. En casos de elecciones de voto repetido, se contabiliza el último voto del usuario

Al emitir el voto, el usuario recibe una confirmación del mismo.

3. Configuración de la plataforma

El sistema permite realizar configuraciones de las validaciones a realizar para emisión de voto e inicio y cierre de elección.

Además los Consultores AppEv pueden realizar los siguientes cambios en la configuración de la elección en tiempo de ejecución a través de un endpoint en **QueryAPI:**

- Cantidad máxima de votos que puede hacer un votante por elección en caso de ser repetida
- Cantidad máxima de pedidos de constancia de voto por votante
- Los usuarios a los que el sistema debe notificar las distintas alertas que provee la plataforma

4. Notificación a usuarios

El sistema implementa un sistema de notificaciones que permite intercambiar fácilmente los medios de comunicación de las mismas, permitiendo en primera instancia realizarlas a través de email y mensaje de texto pero pudiendo extenderse a otros medios en el futuro como podría ser WhatsApp.

5. Generación de consultas

Los usuarios del sistema son capaces de consultarle a éste sobre el estado de las distintas elecciones cargadas en el sistema. Las consultas se hacen a través de **QueryAPI**

En concreto, se pueden realizar las siguientes consultas, según el rol del usuario:

Roles permitidos: Autoridad Electoral

- Consulta de voto
- Consulta del resultado de la elección

Roles permitidos: Autoridad Electoral y Consultor appEv

- Consulta de la configuración de la plataforma

Roles permitidos: Todos

- Horarios más frecuentes de votación
- Cobertura de votos de cada circuito
- Cobertura de votos por departamento

6. Autenticación y autorización de usuarios

El sistema cuenta con un mecanismo para poder autenticar y autorizar a los usuarios de los siguientes roles:

- Autoridad Electoral

- Votante
- Agente de consulta
- Consultores AppEv

Este mecanismo fue elaborado pensando en minimizar las vulnerabilidades del sistema. Se designó la responsabilidad de autenticar y autorizar roles al subsistema **AuthSubSystem,** con el que los usuarios se comunican a través de una API para obtener su token JWT. Luego el usuario le presenta el token al subsistema que le interesa, y éste le consulta a AuthSubsystem sobre la validez del token presentado, mediante GRPC

7. Gestión de errores y fallas

El sistema está construido considerando que se le va a dar un uso intensivo al mismo, y que deberá soportar una alta demanda. En estos escenarios es común el surgimiento de errores inesperados, por lo que nuestro sistema cuenta con un registro estricto de Logs, que mantienen una traza de ejecución que podrá ser inspeccionada en caso de fallos, permitiendo un análisis rápido y eficaz de las causas.

Utilizamos el node module *winston* para la implementación de los logs.

Resumen de Requerimientos No Funcionales

Los principales atributos de calidad que identificamos a partir de los requerimientos funcionales en la sección anterior son Disponibilidad, Seguridad y Performance. En un segundo plano podría agregarse Modificabilidad.

Disponibilidad

- **RF5. Generación de consultas**

  Las consultas deben estar disponibles de uso en todo momento para que los usuarios del sistema puedan acceder a la información de la misma. Podría existir un script que emule el resultado de una elección en tiempo real, constantemente consultando a la plataforma. No debería existir un problema en el sistema al manejar ésto.

- **RF1. Inicio y cierre de elección**

Existen procesos de validación y notificación altamente relacionados con el inicio y el cierre de la elección. Es imprescindible que el sistema esté disponible en éstas instancias, dentro de cada elección.

- **RF2. Emisión del voto**

La plataforma debe estar disponible en el momento que el usuario desee votar. Este requerimiento se fundamenta por el hecho que la votación se da en circuitos, el votante debe movilizarse para llegar a ellos, no podemos admitir que no esté disponible a la hora de votar.

Seguridad

- **RF6. Autenticación y autorización de usuarios**

El sistema permite autenticar y autorizar a los usuarios dependiendo de sus respectivos roles de manera segura, otorgando un token a partir de las credenciales del usuario que podrá usar para acceder a los endpoints que requieran inició de sesión.

- **RF2. Emisión del voto**

El sistema recibe la información del voto encriptada por una clave pública del servidor junto a una firma de su voto creada con la clave privada única del votante. De ésta forma procuramos la integridad del mismo, permitiendo que solo el sistema pueda desencriptar el mensaje con su clave privada, y pudiendo verificar que el votante emitió el mensaje con la firma. También favorecemos la confidencialidad del voto, al no guardar trazabilidad entre votante y voto.

Performance

- **RF2. Emisión del voto**

El sistema devuelve al usuario una respuesta en un margen de 1 a 2 segundos a la hora de realizar un voto. En caso de no poder responder al usuario en este margen, el sistema envía un mensaje de error.

- **RF5. Generación de consultas**

Algunas de las consultas especificadas en este requerimiento deben poder correr en una ventana de tiempo de entre 1 y 2 segundos:

- Consulta de voto.
- Consulta del resultado de la elección.
- Consulta de la configuración de la plataforma.

También hay que tener en cuenta otras consultas dentro del rango de 2 a 10 segundos:

- Horarios más frecuentes de votación
- Cobertura de votos de cada circuito
- Cobertura de votos por departamento

En este requerimiento se identifica un mayor grado de importancia en el atributo de calidad performance para las queries con respecto a los commands.

Modificabilidad

- **RF3. Configuración de la plataforma**

  El sistema permite realizar modificaciones sobre las validaciones en tiempo de despliegue, realizando cambios en archivos de configuración que serán utilizados una vez se redespliegue el código.

  Para las configuraciones de elecciones, como la cantidad máxima de votos; constancias o los usuarios a quienes notificar se permiten modificar en tiempo de ejecución a través de un endpoint de configuración donde se reciben los datos y se actualizan las configuraciones, surtiendo efecto una vez guardadas

- **RF4. Notificación a usuarios**

  Se permite intercambiar fácilmente la implementación del sistema de notificación a usuarios. En la solución entregada, se pueden encontrar el envío de notificaciones por SMS y email, pero sencillamente se pueden agregar nuevos mecanismos de notificación.

- **RF7. Gestión de errores y fallas**

  Se permite intercambiar fácilmente la implementación del sistema de registro de logs al implementarse en un módulo independiente siguiendo el patrón adapter sobre la librería winston, desacoplando de los lugares en donde se utilizará directamente.

Restricciones

1. Tecnología Node.js. Una de las restricciones planteadas por el cliente para éste proyecto fue la elección de la tecnología. Se seleccionó el framework de javascript Node.js como lenguaje de programación para llevar a cabo el proyecto. Ninguno de los integrantes del equipo acumulaba experiencia en ésta tecnología, lo que supuso un reto. El uso Javascript supone grandes restricciones a la hora de introducir concurrencia, lo cual es vital en nuestra solución.

2. Tiempos. Otra restricción del proyecto fue ajustarse al cronograma de trabajo del curso. A pesar de que técnicamente se dio inicio al mismo en el mes de Abril, fuimos incorporando herramientas para lograr el propósito en las semanas siguientes, por lo que el margen de tiempo para realizar el trabajo terminó siendo acotado.

3. Forma de trabajo virtual. Debido a que parte del equipo vive en una región distinta del país, la totalidad del proyecto tuvo que ser llevada adelante por medios virtuales. No fue

posible en ningún momento la modalidad de trabajo presencial, lo que supuso un desafío extra para un proyecto de semejante magnitud, requiriendo mejor organización, comunicación y división de tareas.

4. Composición de los equipos. Aunque hubo libertad para elegir los equipos, una restricción era el número de 3 integrantes por equipo.

Documentación de la Arquitectura

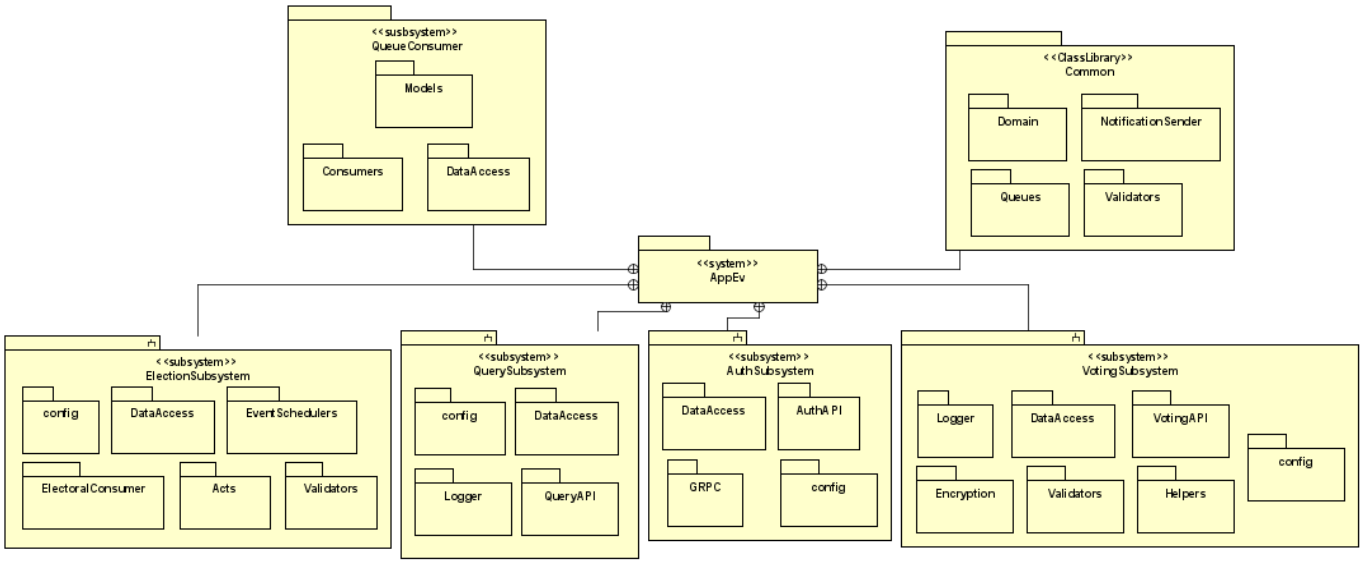
Elegimos el estándar Views and Beyond elaborado por el SIE (Software Engineering Institute) de Carnegie Mellon University para documentar la arquitectura de nuestra solución. A continuación iremos analizando la solución desde un punto de vista arquitectónico, pasando por las distintas vistas de la plataforma.

Nota: Los diagramas se pueden encontrar con mayor resolución en el repositorio en la ruta ./Documentation/Diagrams

Vistas de Módulos

Vista de Descomposición

Representación primaria



Catálogo de elementos

Elemento	**Tipo**	**Responsabilidad**
ElectionSubsystem	Subsistema	Encargado de recibir las nuevas elecciones,

		procesarlas y validarlas. Controla sus cambios de estado. Utiliza un scheduler para consultar periódicamente la API electoral para luego validarlas usando Pipes&Filters configurable y en caso de ser válidas enviar jobs para agregarlas.
QuerySubsystem	Subsistema	Expone una API en la que se pueden hacer distintos tipos de consultas sobre el estado de las elecciones. Se comunica a través de gRPC con el Federated Identity para validar tokens JWT y autorizar los usuarios con sus respectivos roles.
AuthSubsystem	Subsistema	Se encarga de la autenticación de los distintos usuarios que usan la plataforma. Actúa como Federated Identity para autenticar el acceso por los distintos puntos del sistema. Expone un servidor de autorización que recibe tokens a través de gRPC y los autentica.
VotingSubsystem	Subsistema	Expone una API dedicada exclusivamente para la votación y es encargado de manejar la validación y procesamiento de cada voto. Para la validación se utiliza un Pipes&Filters configurable, para la autenticación se consulta el Federated Identity a través de GRPC
QueueConsumer	Subsistema	Contiene a los consumidores que consumen trabajos de la cola Bull, controlando el acceso a la base de datos MySql. Tanto las queues como los consumidores están separadas en command y query, para poder escalar de manera independiente siguiendo el patrón arquitectónico CQRS.
Common	Librería	Contiene clases del dominio y módulos utilizados por todo el sistema.
config	Configuración	Contiene archivos de configuración requeridos para la modificabilidad de cada subsistema.
DataAccess	Paquete	Contiene las clases necesarias para transaccionar con las distintas bases de datos. Se expone una interfaz para comandos (Command) y otra separada para consultas (Query), aplicando el patrón CQRS.
Encryption	Módulo	Encargado de desencriptar la información de los votos que llegan al sistema, con la clave pública de sus votantes, favoreciendo la integridad del voto.
EventSchedulers	Módulo	Encargado de programar trabajos a ejecutarse a

		futuro. En particular, los procesos relacionados al inicio y fin de las elecciones. Utiliza el módulo: node-scheduler
ElectroralConsumer	Módulo	Módulo encargado de consumir la API que expone la Autoridad Electoral, que trae información sobre nuevas elecciones
Logger	Módulo	El Logger es el encargado de registrar logs sobre eventos como accesos al sistema, requests de los distintos usuarios, errores en el sistema. Nos ayudan a auditar a la plataforma. Implementado con el módulo npm *winston*. Se utiliza Adapter para desacoplar la implementación del código de terceros
NotificationSender	Módulo	Mecanismo para enviar notificaciones a los usuarios del sistema. Si bien implementamos dos tipos de *senders* (SMS, Email), gracias al patrón Strategy es altamente modificable y extensible, de forma que se pueden agregar nuevos.
Validators	Módulo	Módulo encargado de realizar la validación de diferentes entidades del sistema como votos o elecciones, implementado a través del patrón Pipes&Filters, usando el patrón de diseño Strategy y un validator manager que actúa como pipe conteniendo cada filter y su orden.

Decisiones de diseño

Decidimos optar por una Service Oriented por sobre monolítica:

Ésto fue producto de analizar detenidamente las responsabilidades que debería cumplir la plataforma. Identificamos responsabilidades muy separadas, consecuentemente se les asignó estas responsabilidades a subsistemas separados. Éstas responsabilidades fueron descritas previamente en el catálogo de elementos.

Con esta decisión estamos aplicando las **tácticas de modificabilidad: Reducir tamaño de módulo e incrementar la cohesión**

Identificamos distintos atributos de calidad a priorizar en cada subsistema, los que pudimos atender separando los mismos en distintos subsistemas.

Por ejemplo, al evaluar el atributo de calidad Performance, el mismo tiene mayor importancia en los requerimientos como voto y consultas, por lo cual es lógico intentar darle más prioridad a estos requerimientos sobre los que no tienen ningún requisito de performance específico. Es por esto que se decidió crear una arquitectura orientada a servicios, que permite escalar horizontalmente cada subsistema de forma independiente, creando nuevas instancias.

Defer binding con archivos config

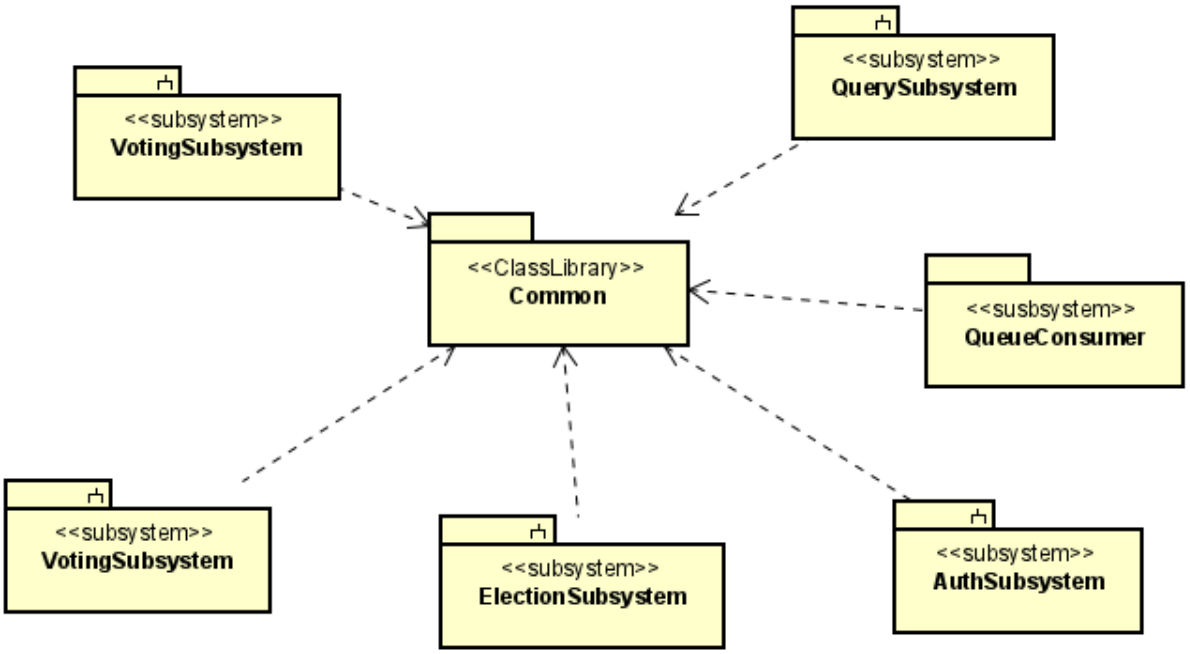
Aplicamos la táctica de **Defer Binding** en los distintos subsistemas para facilitar la **Modificabilidad** de los mismos. Utilizamos defer binding en tiempo de desarrollo/deploy como en tiempo de ejecución. El primero es utilizado para configurar las validaciones, podemos utilizar los distintos validadores, alternando su orden, su ejecución asíncrona y agregar nuevos si deseamos. Ésto se configura en las carpetas *config* de los Validators.

Por otro lado, también podemos configurar en tiempo de desarrollo/deploy el tipo de notificación que queremos enviar en los distintos subsistemas.

Podemos configurar elecciones en tiempo de ejecución haciendo un POST a la QueryAPI /elections/{id}/config

Vista de Uso

Representación primaria



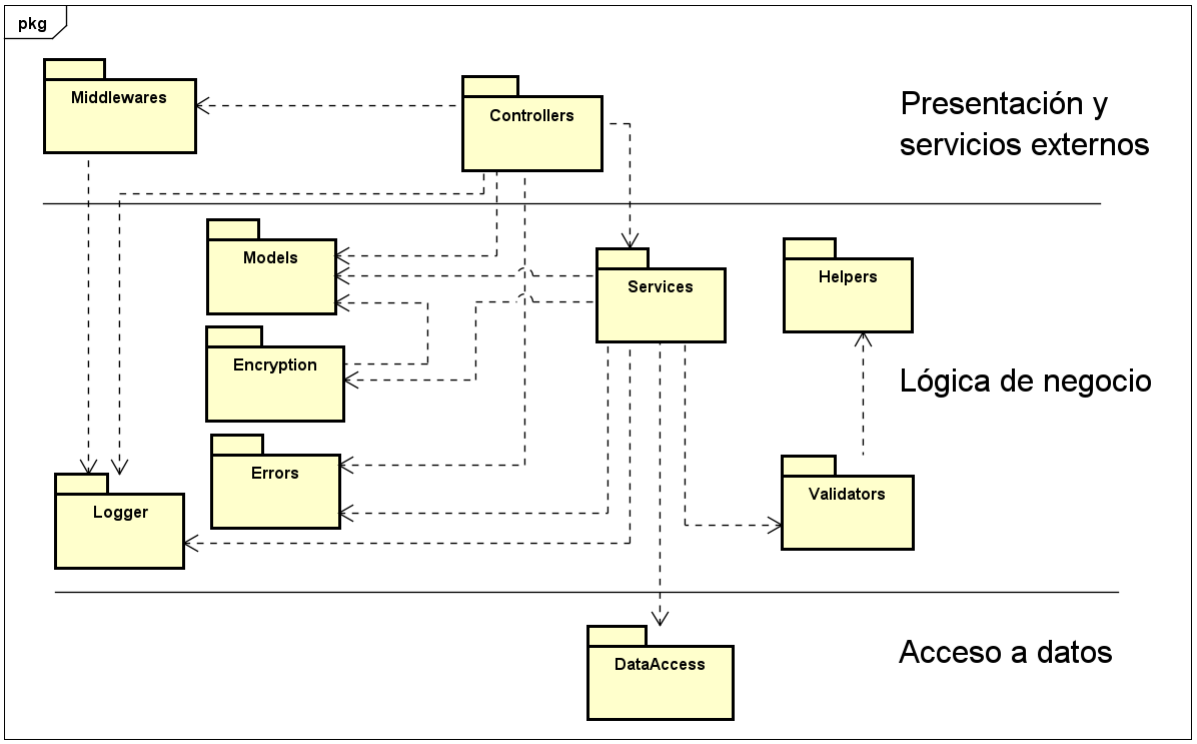
Decisiones de diseño

Paquete Common. Decidimos crear una biblioteca de clases con elementos que son utilizados a través de los distintos subsistemas. Ésto implica los elementos del dominio, la base de los validadores implementados con **Pipes & Filters**, la interfaz de las colas de Bull y los enviadores de notificaciones.

Entendemos que ésto implica una gran dependencia de la plataforma al paquete Common, idealmente a la hora de desplegar el sistema, se analizaría qué elementos del Common mantener en cada subsistema.

Vista de layers

Representación primaria



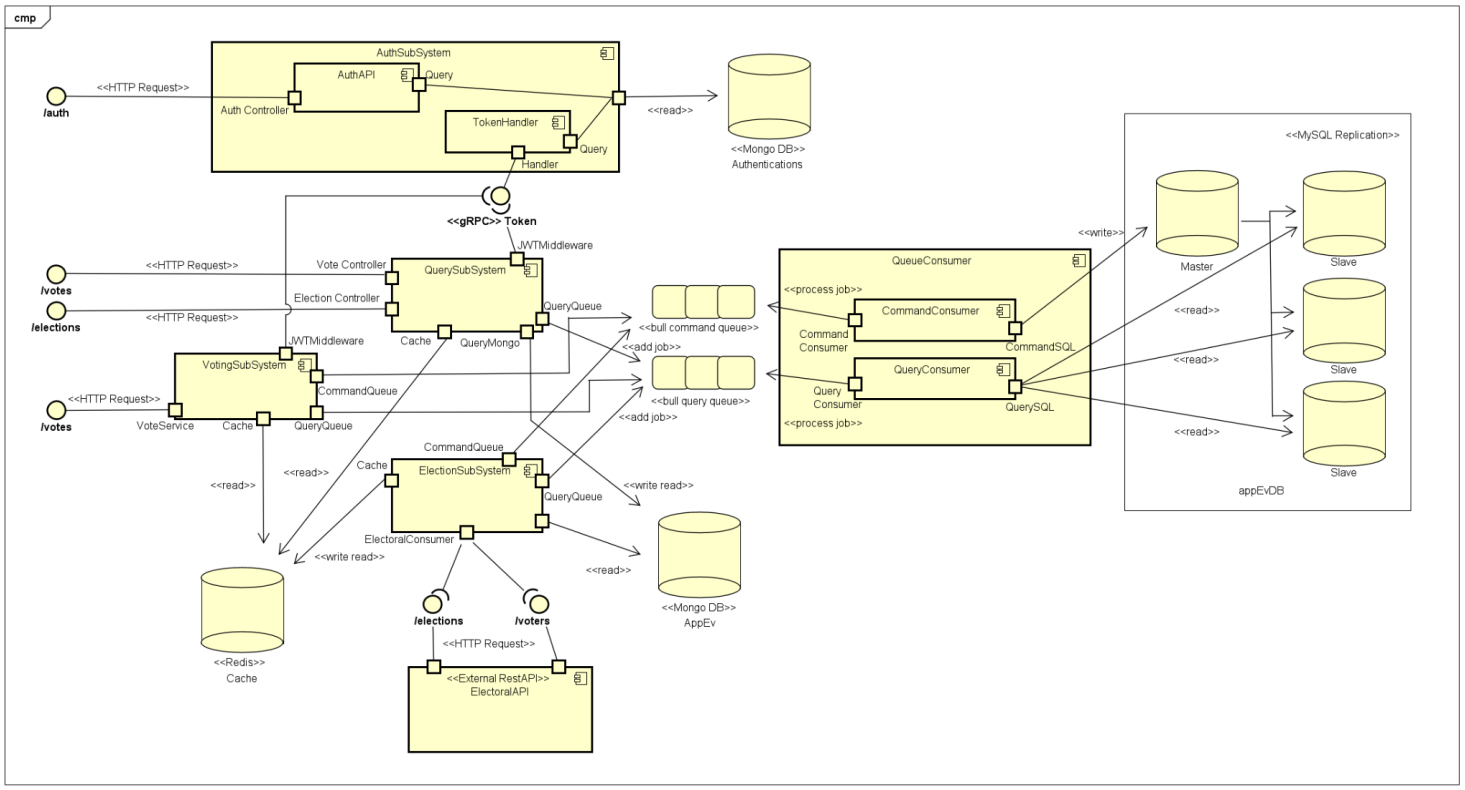
Decisiones de diseño

Decidimos seguir el patrón **Layers** porque nos permitió desarrollar de manera separada, favoreciendo la modificabilidad y reuso por menor interacción entre las partes, además de ser el patrón al que nos conduce Express a la hora de desarrollar.

Vistas de Componentes y Conectores

Vista externa del sistema

Representación primaria



Catálogo de elementos

Se describen los elementos que no han sido mencionados anteriormente en otros catálogos.

Elemento	**Tipo**	**Responsabilidad**
AuthAPI	Módulo	Expone la API de autenticación, con un endpoint de logueo que devuelve un token a partir de las credenciales del usuario
TokenHandler	Módulo	Recibe un token por gRPC y devuelve un usuario con sus datos y rol.
CommandConsumer	Módulo	Consume la cola de commands y es responsable de comunicarse con el master de la base de datos.
QueryConsumer	Módulo	Consume la cola de queues y hace consultas a los slaves de la base de datos.
appEvDB [MySQL]	Base de datos	Encargado de guardar los datos a través del

		master y replicarlos en todos los slaves
Bull comand queue	Cola de prioridad	Recibe jobs de command y los encola utilizando la prioridad recibida para ordenar la forma en que se consumen
Bull query queue	Cola de prioridad	Recibe jobs de query y los encola utilizando la prioridad recibida para ordenar la forma en que se consumen

Interfaces

Interfaz: Token [gRPC]	
Paquete que implementa: TokenHandler (AuthSubSystem)	
Servicio	**Descripción**
Validate	Recibe una TokenRequest que contiene el token como un string y devuelve una TokenResponse que contiene el email, la cédula y el rol del usuario. Esta comunicación se realiza a través de gRPC.

Interfaz: /auth [HTTP]	
Paquete que implementa: AuthAPI (AuthSubSystem)	
Servicio	**Descripción**
/login [POST]	Recibe una request HTTP con el email y la password del usuario y devolvemos un token JWT que le servirá como autenticación y autorización para otros subsistemas.

Interfaz: /votes [HTTP]	
Paquete que implementa: VoteController (QuerySubSystem)	
Servicio	**Descripción**
/ [GET]	Recibe una request HTTP para obtener la

	fecha en que un usuario votó para una elección. Se reciben electionId y voterCI en el body de la request. Roles autorizados: Autoridad Electoral.
/proof [GET]	Recibe una request HTTP para obtener la constancia de voto. Se reciben voteId y voterCI en el body de la request. Roles autorizados: Votante

Interfaz: /elections/:id [HTTP]	**Nota:** Para los servicios contenidos en esta tabla, se envía el id de la elección sobre el que se va a trabajar en la ruta.
Paquete que implementa: ElectionController (QuerySubSystem)	
Servicio	**Descripción**
/config [GET]	Recibe una request HTTP para obtener la configuración de la elección indicada. La respuesta devuelve un objeto JSON con la máxima cantidad de votos por votante, en caso de que la elección sea de modo repetido. También la máxima cantidad de pedidos de constancia de voto por usuario, como la lista de usuarios a los que el sistema debe enviar notificaciones Roles autorizados: Autoridad electoral, consultor appEv
/config [POST]	Recibe una request HTTP para configurar la elección provista en la URI. Se pueden modificar los parámetros previamente descritos en la ruta GET. Éstos son enviados en formato JSON por el body de la request. En caso de no especificar un cambio en uno de los parámetros, éste no se ve modificado. Roles autorizados: Consultor appEv
/vote-frequency [GET]	Recibe una request HTTP para obtener los datos acerca de la frecuencia de votos dividida por fechas para una elección dada. Se devuelve una objeto JSON con una lista de horas y la cantidad de votos que se registraron en cada una.

	Roles autorizados: Todos
/circuit-info [GET]	Recibe una request HTTP para obtener los datos de los circuitos de la elección. En el body se reciben los siguientes parámetros: - minAge: [opcional] edad mínima de los votantes - maxAge: [opcional] edad máxima de los votantes - rangeSpace: rango de agrupación de edades. EJ:8, se agrupan los votos por edad de votantes [18-25] [26-31] … Se devuelve una lista que contiene para cada circuito la cantidad de votos agrupada por las edades en los rangos indicados, separadas por género. Roles autorizados: Todos
/state-info [GET]	Recibe una request HTTP para obtener los datos de los departamentos de la elección. En el body se reciben los siguientes parámetros: - minAge: [opcional] edad mínima de los votantes - maxAge: [opcional] edad máxima de los votantes - rangeSpace: rango de agrupación de edades. EJ:8, se agrupan los votos por edad de votantes [18-25] [26-31] … Se devuelve una lista que contiene para cada departamento la cantidad de votos agrupada por las edades en los rangos indicados, separadas por género. Roles autorizados: Todos
/ [GET]	Recibe una request HTTP para obtener un resumen del resultado de la elección hasta el momento. Devuelve un objeto JSON con: ● Cantidad de habilitados para votar ● Cantidad de votantes habilitados por departamento ● Cantidad total de votos por departamento ● Cantidad total de votos ● Cantidad de votos por candidato ● Cantidad de votos por partido Roles autorizados:Autoridad Electoral

Interfaz: /votes [HTTP]	
Paquete que implementa: VoteService (VotingSubSystem)	
Servicio	**Descripción**
/ [POST]	Recibe un request HTTP para emitir un voto en una elección dada. Dada la precondición de que el usuario fue loggeado con AuthAPI y tiene su token. Se envía en el body: ● Identificador de la Elección (idElección) ● Identificador del Circuito (idCircuito) ● Documento de Identidad del votante ● Documento de Identidad del candidato votado La información se envía firmada digitalmente con la clave privada del votante. Rol autorizado: Votante

Relación con elementos lógicos

Componente	**Paquetes**
QuerySubsystem	DataAccess, Logger, QueryAPI
VotingSubsystem	DataAccess, Encryption, Logger, Validators, VotingAPI
ElectionSubsystem	DataAccess, Acts, ElectoralConsumer, Errors, EventSchedulers, Models, Validators
AuthSubsystem	DataAccess, GRPC, AuthAPI
AuthAPI	Controllers, Helpers, Models, Routes
QuerueConsumer	DataAccess, Models, CommandConsumer, QueueConsumer

Decisiones de diseño

CQRS

Desde un principió se identificó que la performance requerida para las lecturas era mayor a la necesaria para las escrituras, de hecho ninguna de las escrituras tenía un requisito de performance, mientras que la mayoría de las lecturas si lo tienen.

Es por esto que desde una etapa muy temprana del proyecto decidimos usar el patrón CQRS, ya que él mismo nos permite escalar independientemente las lecturas y escrituras. Algunas de las decisiones de diseño que serán mencionadas posteriormente también tienen que ver con la decisión de seguir este patrón, como el uso de queues separadas en command y query y la replicación de base de datos.

Evaluando esta decisión en retrospectiva, ya habiendo probado el sistema y analizado las métricas[1], estamos seguros de que fue una de las decisiones más importantes y beneficiosas para el proyecto, ya que tuvo un gran impacto en dos de los atributos de calidad que consideramos más importantes, disponibilidad y performance.

MySQL Replication

Se decidió usar la táctica de **múltiples copias de datos** para la base de datos principal del sistema, ya cuando comenzamos a hacer load testing del sistema[2], nos dimos cuenta que nuestro mayor cuello de botella para las consultas y validaciones (que hacen también consultas sobre la base de datos) se debía a que estamos corriendo una única instancia de MySQL, sobre la que se hacían todas las inserciones y consultas, siendo este además un único punto de falla.

Se decidió pasar a un modelo Master-Slave, donde se hacen las inserciones se hacen en el Master y las consultas en los Slaves, teniendo así múltiples copias de datos y mejorando la performance del sistema.

Colas de prioridad y acceso a datos

Otro de los problemas con los que nos encontramos fue el acceso a la base de datos de appEv, ya que como se puede ver en el diagrama, la mayoría de los subsistemas accedería a la misma.

---

[1] Anexo de métricas
[2] Anexo de métricas

En un principio se pensó que cada subsistema tenga su propio acceso a datos, comunicándose con la base de datos appEvDB directamente a través de sequelize. Rápidamente nos dimos cuenta que esto podría ser un problema ya que esta solución acoplaba a todos los subsistemas a tener que usar sequelize y conocer todos los modelos de la base de datos. Esto desfavorece la **modificabilidad** del sistema, ya que cualquier cambio en los modelos implicaría tener que propagarlos en todos los subsistemas que lo utilicen. O aún peor, si un día se decide cambiar de MySQL a SQL Server o Mongo por ejemplo, esto implicaría un cambio gigantesco que afectaría también a todos los subsistemas.

El primer paso para resolver este problema fue crear un servicio que se encargue de enviar commands y queries a appEvDB, utilizando la táctica de **modificabilidad usar un intermediario y abstraer servicios comunes**. Éste servicio sería el único responsable de acceder a los datos, desacoplando todos los otros subsistemas del uso de sequelize o conocer incluso que base de datos estamos utilizando. Éste servicio dio origen a lo que terminaría siendo QueueConsumer pero en un principio solo era un DataAccess.

Una vez definimos la creación de este servicio utilizando la táctica de **agregar un intermediario,** comenzamos a evaluar qué otros atributos de calidad serían afectados por esta decisión. De los tres principales atributos (Disponibilidad, Seguridad y Performance), el que se vería afectado era **performance**, ya que al agregar un intermediario se agregaría overhead en cada command o query a appEvDB. Mientras estábamos evaluando el impacto que podría tener el intermediario nos dimos cuenta de que teníamos una mayor amenaza a la performance, esto era que si hay un job con gran impacto a la base de datos como agregar una nueva elección junto con sus votantes, todos los demás accesos a la misma se verían retrasados.

Teniendo en cuenta que no todos los requerimientos tienen los mismos requisitos de performance decidimos implementar una cola de prioridad antes del servicio DataAccess anteriormente mencionado, siguiendo la táctica de **performance** de **priorizar eventos**. De esta manera los eventos que necesitan acceso a datos y tienen requisitos de performance tienen más prioridad que los demás. Además también podemos darle más prioridad a los que necesitan respuestas de entre 1 y 2 segundos como la emisión de voto o algunas queries sobre las queries que necesitan entre 2 y 10 segundos.

De esta manera DataAccess pasaría a llamarse QueueConsumer, elemento mostrado en el diagrama, y tendría la nueva responsabilidad de consumir desde la cola de prioridad y procesar los jobs que le sean asignados haciendo commands o queries a la base de datos.

Nota: un problema que encontramos en las colas de bull fue que si activamos el borrado automático de jobs, los mismos se volvían inconsistentes y perdíamos la mayoría, por lo cual optamos por activar la opción de que se eliminen los jobs una vez se pasan los 10000 terminados para que el sistema pueda trabajar correctamente.

Carga de elecciones

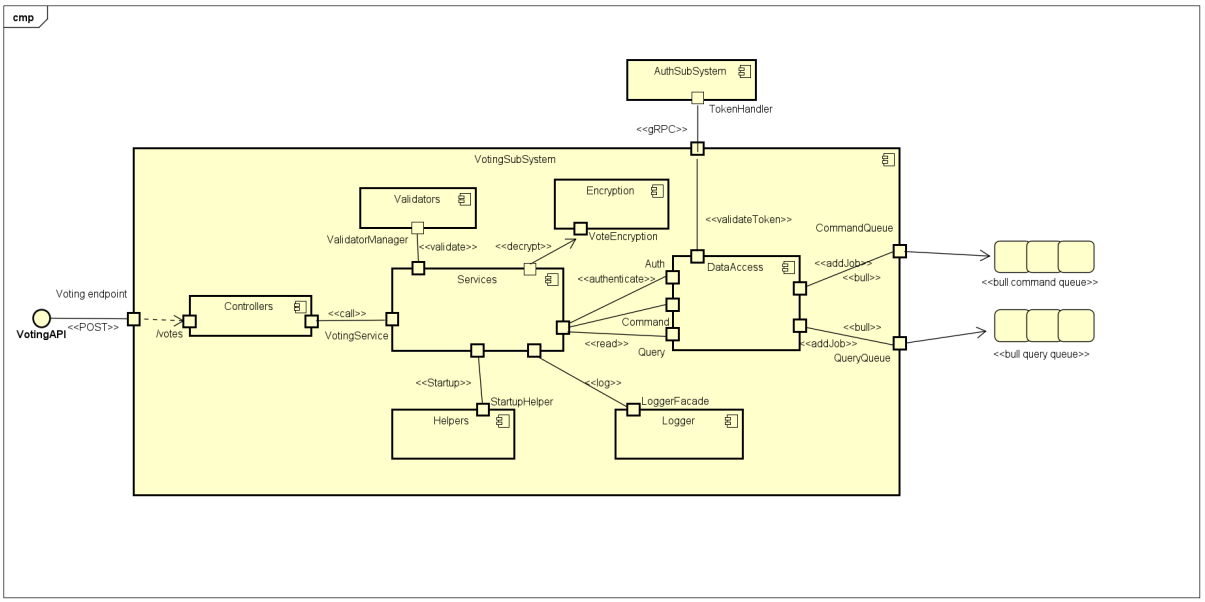
El subsistema encargado de cargar elecciones al sistema es *ElectionSubsystem*. La carga de elecciones fue una de las principales decisiones de diseño tomadas por el equipo. La disyuntiva era decidir entre cargar las elecciones enteras con sus votantes, que pueden ser millones, en nuestra base de datos, frente a cargar la información básica de cada elección y consultar por los votantes a la API externa.

Analizamos detenidamente éste problema, y optamos por guardar toda la información de las elecciones en nuestra base de datos, inclusive los votantes. Ésta decisión fue tomada analizando las reglas de negocio: la operación de carga de elecciones debería estar programada con antelación, ya que las elecciones no suceden de imprevisto. Además la API externa no está bajo nuestro control, por lo que no podemos asegurar su disponibilidad ni performance a la hora de validar un votante.

A efectos prácticos en el uso del sistema, el sistema irá en busca de nuevas elecciones una vez por mes (se puede configurar la fecha), por lo que sería contraproducente poner elecciones en ese día de carga. Puede ser visto como una tarea de mantenimiento de la plataforma.

Vista Interna de VotingSubsystem

Representación primaria



Relación con elementos lógicos

Componente	**Paquetes**
Validators	Filters
Encryption	VoteEncryption
Controllers	Server
Services	VotingService
DataAccess	Auth, Command, Query
Helpers	StratupHelper, RequestCountHelper
Logger	AccessLoggers, RequestLoggers

Decisiones de diseño

Federated Identity

En un principio pensamos en utilizar el sistema de autenticación de Google como **Federated Identity,** pero decidimos crear nuestro propio sistema para favorecer la Testeabilidad del sistema, así creamos AuthSubsystem. Para reducir la complejidad del mismo decidimos enviar las credenciales del usuario sin encriptar en el body de la request, pero somos conscientes de que en caso de implementar el sistema en un caso real, se deberían encriptar estos datos o utilizar un servicio de Federated Identity como Google por ejemplo, favoreciendo el atributo de calidad de seguridad.

El subsistema nos permite **autenticar** y **autorizar** a los distintos usuarios del sistema, así como **identificarlos**, favoreciendo el atributo de calidad de **Seguridad.** Como implementamos autorización por roles en los distintas APIs que usan éste servicio, **limitamos el acceso** a los endpoints de las mismas.

Conexión gRPC con Federated Identity

Para la comunicación con el Federated Identity, se decidió usar gRPC ya que es un protocolo de comunicación más rápido que HTTP/REST, se investigaron diferentes artículos[3] antes de tomar la decisión de usar gRPC. Como se necesita autorización y autenticación para varios de los requerimientos que necesitan favorecer el atributo de calidad Performance (Emisión de voto y consultas), decidimos utilizar el protocolo más performante en cuanto a velocidad de respuesta.

Seguridad en gRPC

Otro detalle con respecto a la conexión gRPC, es que también se tuvo en cuenta la Seguridad de la misma, ya que estamos manejando datos de los usuarios y queremos favorecer su privacidad. Investigamos los posibles tipos de encriptación que ofrece gRPC y descubrimos que el mismo ofrece encriptación de tres tipos: SSL/TLS, ALTS y Token-based authentication with Google.

Consideramos que el uso de estos tipos de conexión escapan del alcance del obligatorio, ya que no disponemos de certificados SSLs ni tenemos experiencia no los otros tipos de conexión ofrecidos por gRPC, por lo cual decidimos usar la conexión de forma insegura (desencriptada), pero somos conscientes de que la misma debería cambiarse a una conexión encriptada si el sistema se quiere poner en producción.

Logger para auditar la información

Implementamos un logger que loggea los accesos al sistema, y la actividad de los usuarios, manteniendo su confidencialidad. Se implementó usando el módulo *winston*, utilizando el patrón Adapter, de forma de desacoplar su implementación favoreciendo Modificabilidad. El Logger es usado en todos los subsistemas con los que los usuarios interactúan, generando un historial de la actividad del subsistema. Con ésta información podremos **auditar el sistema** en escenarios adversos, para orientarnos a la falla, favoreciendo la **Seguridad** del mismo

[3] https://albertsalim.dev/post/grpc-vs-rest-node-js/ y https://daily.dev/blog/grpc-vs-rest

Validación

Para la validación de los votos se decidió usar el patrón de arquitectura **Pipes&Filters,** al ser un requisito que estos sean fácilmente intercambiables en tiempo de despliegue. Este patrón nos permite crear una pipeline a partir del Validation Manager con los filtros configurados a partir de un archivo de configuración. También se está aplicando la táctica de **modificabilidad defer binding**, para poder configurar los filtros a utilizar en tiempo de despliegue en el archivo anteriormente mencionado.

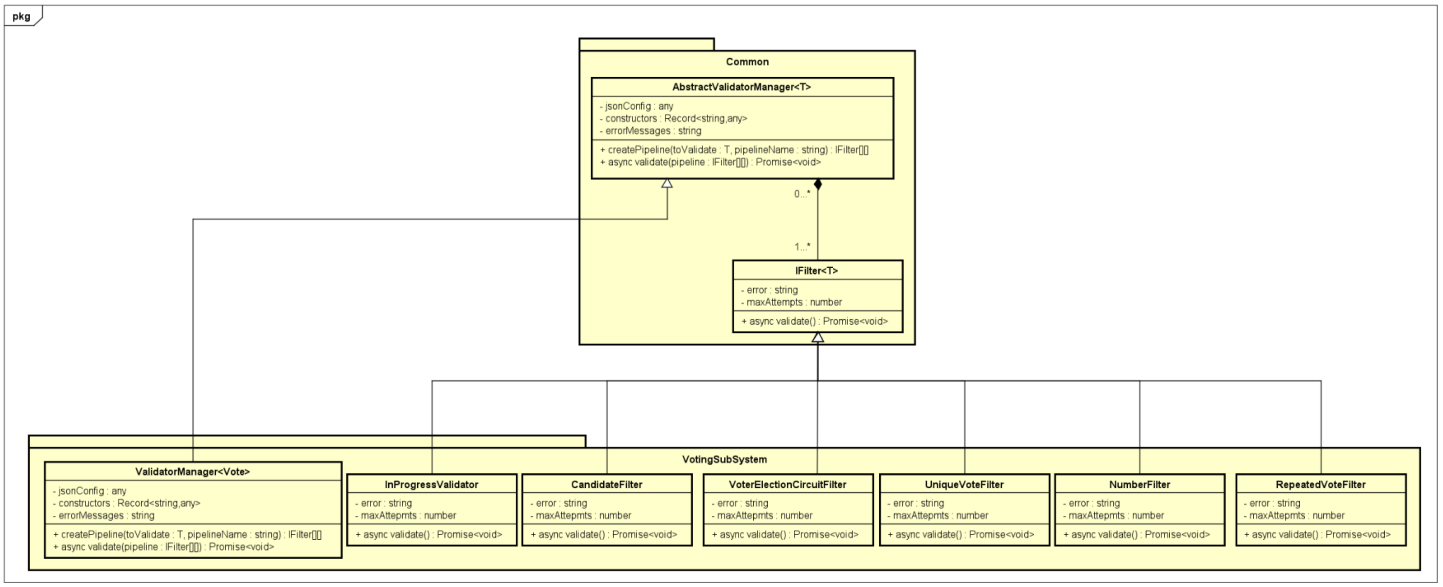
El formato del archivo de configuración es JSON y su estructura es la siguiente:

```
{
“nombre de pipeline”:[
            {
            “filters”: [
                  ]
            },
            {
            “filters”: [
                  ],
            },
            ….
            …
            {
            “filters”: [
                  ]
            },
      ]
}
```

Dentro de cada filters se agregan los filtros a correr, por ejemplo:

```
{
  "class": "UniqueVoteFilter",
  "parameters": {
     "maxAttempts": 1,
     "key1": "voterCI",
     "key2": "electionId",
     "errorMessage": "The voter has voted already",
      "description": "Filter voters that have voted already"
     }
 }
```

Para la implementación del patrón **Pipes&Filters** se utilizó el patrón de diseño strategy, creando una clase IFilter, de donde todos los filtros heredan e implementan el método validate, para poder hacer la pipeline agnóstica a la implementación de los mismos, solo sabiendo de la existencia del método validate.



Encriptación

Dada la sensibilidad de la información transferida durante la emisión del voto, decidimos utilizar la táctica de **Seguridad de encriptación de la información.**

Para asegurar la **privacidad de los votantes** al momento de enviar los votos, se hace uso del paquete *crypto* de npm para:

- la creación de pares de claves del servidor y votantes
- la firma y encriptación de un voto dado
- la desencriptación y validación de la integridad del mensaje en el servidor.

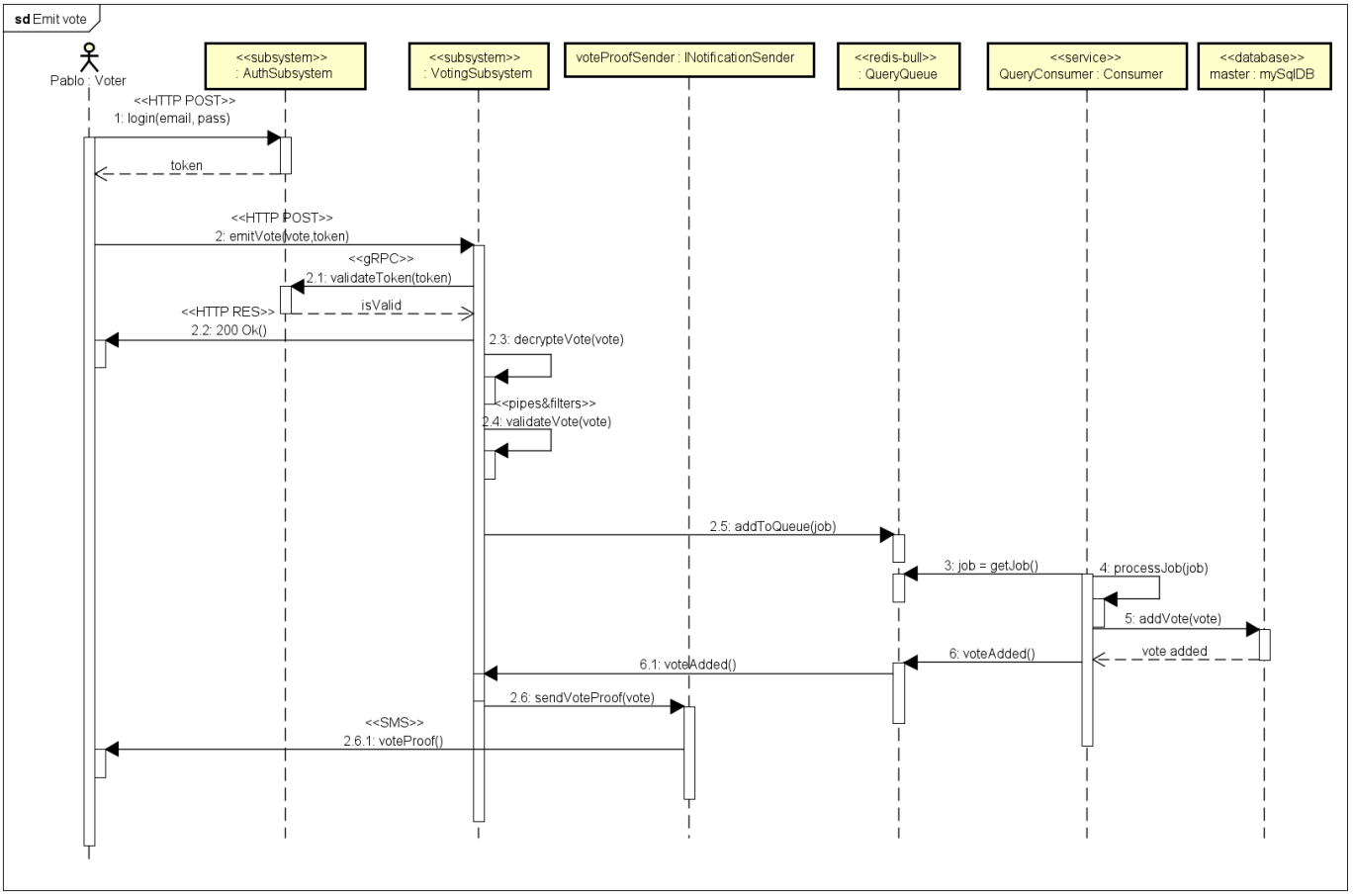
Para esto se generó un par de clave pública y privada para todos los votantes y el servidor, donde la pública del servidor es usada por los votantes para encriptar sus votos.
Todos los datos de un voto son encriptados excepto la cédula de identidad del votante, para poder buscar la clave pública asociada a esa cédula.

Dentro del objeto encriptado se incluye el id de la elección, el id del circuito donde se votó, la cédula de identidad del candidato al que se está votando y el momento en el que se votó. Estos últimos cuatro datos son usados para generar una **firma con la clave privada del votante** incluida dentro del objeto encriptado con la clave pública del servidor.

Haciendo uso de los cuatro datos de la firma y la clave pública del votante se puede seguir la táctica de **Seguridad de verificar la integridad del mensaje** del voto para asegurarse de que alguien no vote por otra persona.

Comportamiento

Emisión de voto



Decidimos priorizar la claridad del diagrama, optando por omitir los procesos que suceden a bajo nivel en el sistema.

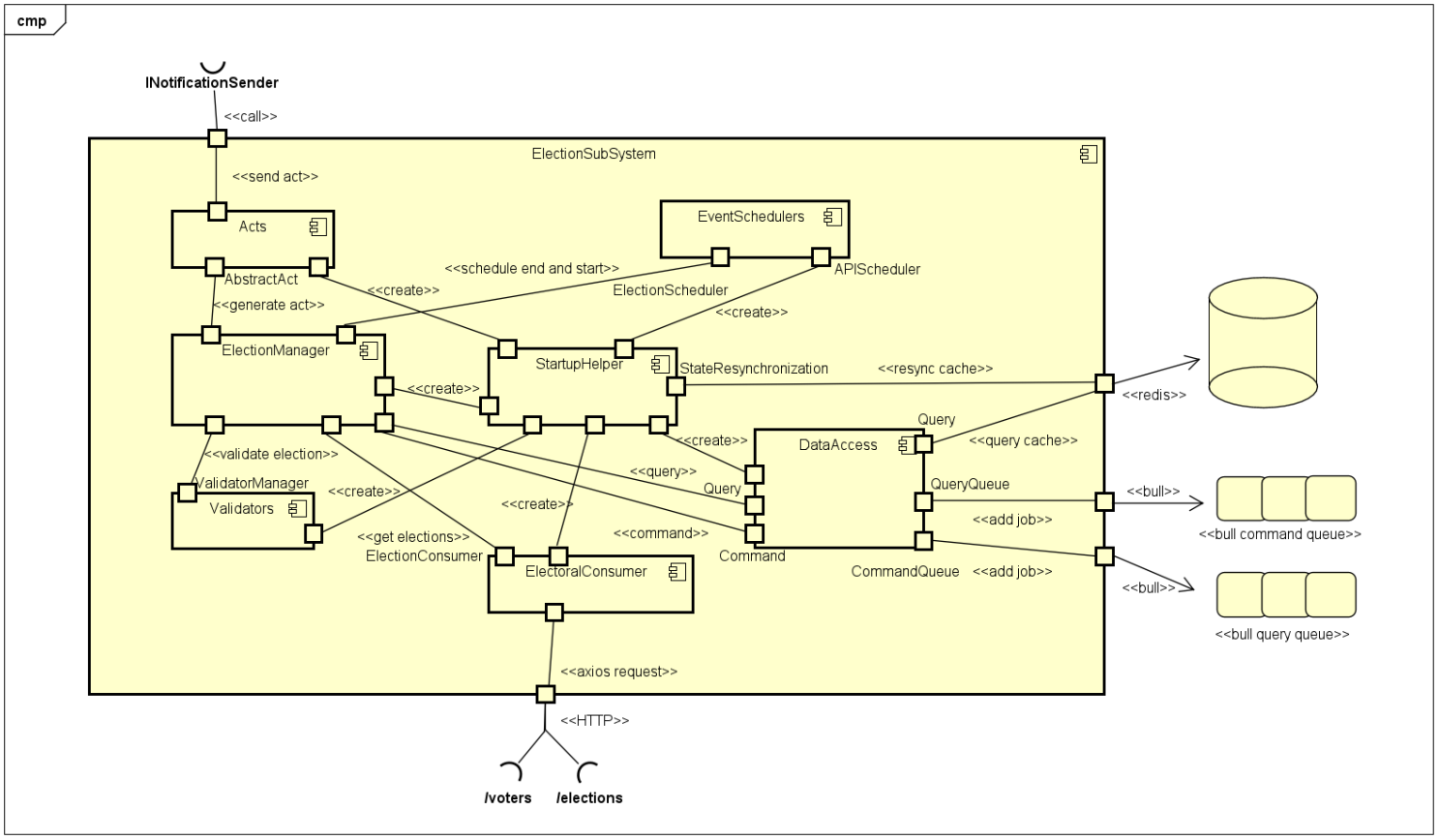
En el diagrama de secuencia se muestra únicamente el caso de voto exitoso, también para aportar claridad. Existe la posibilidad de que el votante no se pueda autenticar, que la petición haga timeout (seteado en 2 segundos, aplicando la táctica de **limitar el tiempo de ejecución**), o que el voto no cumpla con el proceso de validación. Esos casos están cubiertos en la implementación de la funcionalidad

Decidimos que responda la petición HTTP apenas el token es validado. Luego el voto pasa por un proceso de validación, que en caso de fallar, se le notifica al usuario por SMS. De ésta forma aplicamos la táctica de **recuperación ante fallas,** favoreciendo a la **Disponibilidad.**

Por último el voto es agregado a base de datos por intermedio de una cola Bull, cuándo recibimos la confirmación de que el voto fue agregado, se le envía la constancia de voto al votante.

Vista Interna de ElectionSubSystem

Representación primaria



Catálogo de elementos

Elemento	**Tipo**	**Responsabilidad**
ElectionManager	Clase	Coordina todos los procesos asociados a la carga de las elecciones.
StartupHelper	Clase	Contiene los procesos para iniciar éste subsistema, que es el core de la plataforma. Recuperándose ante fallas y favoreciendo **Disponibilidad**
Acts	Módulo	Encargado de crear las actas de inicio y fin de elección

Relación con elementos lógicos

Componente	Paquetes
DataAccess	Command, Query
ElectoralConsumer	APIConsumer
ElectionSchedulers	APIScheduler, ElectionScheduler
Validators	Filters

Decisiones de diseño

State Resynchronization

Una parte importante del sistema es el cache guardado en redis, el mismo es utilizado para evitar hacer llamadas constantemente a la base de datos, en el guardamos información de las elecciones que es pedida en varios puntos del sistema.

El problema con el cual nos topamos fue que la base de datos en Redis trabaja sobre memoria, por lo cual en caso de tener algún problema o que se caiga la instancia de la misma, todos los datos que están almacenados se perderían.

Como parte de los subsistemas dependen de que el caché está cargado con la información básica de las elecciones, es crucial que éste tenga los datos correctamente cargados y en caso de un reinicio pueda volver a obtenerlos para asegurar tener una mayor **disponibilidad** del sistema. Se decidió usar la táctica de **state resynchronization** para restablecer los datos del cache si el mismo se pierde en algún momento.

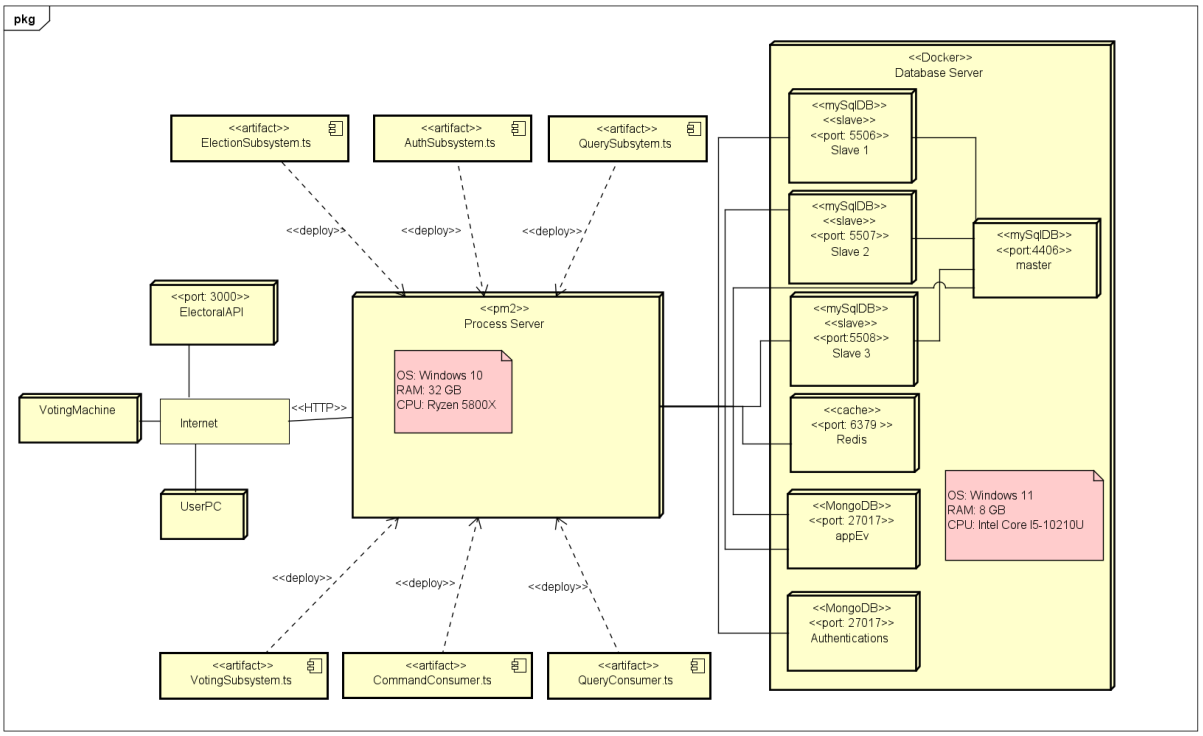
Notificaciones

Para favorecer la modificabilidad del sistema de notificaciones, se hizo uso de tácticas como "**Defer Binding**" y **abstraer servicios comunes** de reducir acoplamiento teniendo interfaces que exponen la funcionalidad a ser implementada por notificadores concrectos. La solución a la que se llegó es configurable en tiempo de despliegue por archivos de configuración, donde dependiendo del subsistema, se puede pasar por string el notificador de usuarios entre los existentes (SMS o EMAIL) o por inyección de dependencia se puede cambiar el utilizado en la lógica de negocio.

Vistas de Asignación

Vista de Despliegue

Representación primaria



Nota: Esta estructura de despliegue fue la utilizada durante el desarrollo del proyecto, en un ambiente académico, en caso de desplegar en producción el sistema, sería recomendable utilizar servidores independientes para cada artefacto a desplegar, de esta manera se puede facilitar la asignación de recursos a los mismos.

Catálogo de elementos

Elemento	**Tipo**	**Responsabilidad**
PM2	Nodos	Permite la replicación de instancias de los distintos subsistemas

Decisiones de diseño

Introducción de concurrencia y múltiples instancias de cómputo.

PM2 nos permite tener distintas instancias de los distintos subsistemas corriendo simultáneamente de manera que éstas se distribuyen la carga computacional entre sí. Éstos puntos favorecen a la **Performance** del sistema.

Recuperación de fallas

PM2 introduce la posibilidad de hacer uso de **Retry** en los procesos ante fallas de las instancias, favoreciendo la **Disponibilidad** de los subsistemas.

Reflexión final

Ya culminado el desarrollo del proyecto, tomamos conciencia del trabajo logrado en el correr de éstos últimos meses. Se logró implementar una solución que resuelve un problema con gran aplicación en el mundo, como es el voto electrónico. Se analizó detenidamente los atributos de calidad en los que nos deberíamos focalizar para llegar a una solución que podría usarse en la realidad.

Utilizamos una tecnologías en la que no teníamos experiencia alguna como lo son Node.js, Redis o Bull. Ésto nos permitió expandir nuestro conocimiento técnico. Nos apoyamos en el inmenso mundo de *npm* para resolver problemas comunes en el desarrollo como lo son por ejemplo, el scheduling de jobs o logging, para los que utilizamos node-scheduler y winston.

Llegamos a una arquitectura orientada a servicios, la cual nos extendió el concepto de sistema. Hasta el momento hemos trabajado en sistemas monolíticos, diseñar un sistema que esté comprendido por distintos sistemas independientes que colaboran entre sí fue un logro inmensurable para el equipo.

Fue muy positivo lograr una sinergia en el trabajo colectivo, potenciamos las habilidades de cada uno para el bien en común del equipo. Ocurrió una clara distinción de roles, tal cual como se da en el mundo profesional. Nos apoyamos como equipo en los momentos más estresantes, y celebramos las pequeñas victorias.

Quedamos muy conformes con el trabajo realizado, a nivel técnico y de gestión. Pudimos poner en práctica distintos aspectos de la ingeniería de software visto en otros cursos.

Anexo

Métricas

Las métricas fueron tomadas haciendo uso del paquete autocannon de npm que permite realizar requests concurrentes o por una duración de tiempo. Solamente se ejecutó de la segunda manera en los requerimientos que buscaban capacidad en 10 segundos.
En los primeros tests se muestra la diferencia entre el uso de MySQL distribuido y no distribuido, y en todos los tests se corre además de las instancias mostradas, el resto de los subsistemas necesarios para el funcionamiento correcto de del sistema appEV.
En los pedidos se indica el tiempo promedio (Time AVG ms) de cada request.

Requerimientos de 2 segundos:

Votos

Con MySQL distribuido:

Instancias VotingSubSystem	Query Consumers	Command Consumers	Vote Count	Guardados en db	Res HTTP	Time (AVG ms)	Process ms
1er Intento							
5	6	3	1000	1000	1000	600	2909.137
5	6	3	2000	2000	2000	1422	4259.880
5	6	3	3000	3000	3000	3189.79	6056.7737
2do Intento							
5	6	3	1000	1000	1000	900	2974.312
5	6	3	2000	2000	2000	1622	4579.271
5	6	3	3000	3000	3000	4019.69	7455.8683
Promedio para configuraciones							
5	6	3	1000	1000	1000	750	2941.7245
5	6	3	2000	2000	2000	1522	4419.5755
5	6	3	3000	3000	3000	3604.74	6756.321

Sin MySQL distribuido:

Instancias VotingSubSystem	Query Consumers	Command Consumers	Vote Count	Guardados en db	RES HTTP	Time (AVG ms)	Process ms
5	6	3	1000	1000	1000	580.91	3636.396
5	6	3	2000	2000	2000	2798.04	5354.664
5	6	3	3000	3000	2718	3985.19	6961.357

Se llega a la conclusión que se pueden procesar entre 2000 y 3000 votos concurrentes en menos de 2000 ms con la configuración planteada, y que se nota una diferencia en el tiempo de procesamiento y en la desviación estándar del tiempo de las requests cuándo se hace uso de MySQL distribuido. Teniendo bases de datos distribuidas se pasa de más de 10 segundos como máxima desviación estándar a menos de 2000 milisegundos.

Req 8: Cantidad de votos

GET : (http://localhost:3003/votes)
Body:

```
{
   "electionId": electionId,
   "voterCI": "voterCI"
}
```

Con MySQL distribuido:

Instancias QuerySubSystem	Query Consumers	Query Count	Res HTTP	Time (AVG ms)
1er Intento				
5	12	100	100	220.16
5	12	500	500	496.53
5	12	1000	1000	712.12
5	12	2000	2000	1575.48
5	12	3000	3000	2119.83
5	12	10000	9994 (+6 code: 400)	5048.25
2do Intento				
5	12	1000	1000	751.78
5	6	2000	2000	1189.52
5	6	3000	3000	1791.22
Promedio para las configuraciones:				
5	6	1000	1000	751.78
5	6	2000	2000	1382.5
5	6	3000	3000	1955.52

Sin MySQL distribuido:

Instancias QuerySubSystem	Query Consumers	Query Count	Res HTTP	Time (AVG ms)
5	12	1000	1000	1004.85
5	12	2000	2000	1253.03
5	12	3000	3000	2155.74
5	12	10000	3729	9503.39

Se llega a la conclusión de que para esta query el máximo posible de pedidos concurrentes está entre 2000 y 3000 y que nuevamente se nota una diferencia con la presencia de MySQL distribuido, notando que o mantiene el performance o lo mejora en algunos casos.

Se puede medir ya sea por desviación estándar (1500ms en no distribuido a diferencia de distribuido con menor a 1200ms) o por mayor consistencia incluso en requests más grandes (como por ejemplo en la de 10000 concurrentes) o mantener el response time promedio.

Esto podría indicar que una mayor cantidad de instancias del QuerySubSystem podría llegar a permitir más requests.

Se toma la decisión por las pruebas anteriores de mantener el uso de MySQL distribuido de aquí en adelante.

Req 9: Información de elección

GET http://localhost:3003/elections/{electionId}

Instancias QuerySubSystem	Query Consumers	Query Count	Res HTTP	Time (AVG ms)
1er Intento				
5	12	100	100	159.66
5	12	500	500	887.75
5	12	1000	1000	2151.75
5	12	2000	2000	4043.78
2do Intento				
5	12	100	100	148.89
5	12	500	500	868.53
5	12	1000	1000	2376.36
5	12	2000	2000	4246.86
Promedio para las configuraciones:				
5	12	100	100	154.275
5	12	500	500	878.14
5	12	1000	1000	2264.055
5	12	2000	2000	4145.32

Se concluye que se puede contestar alrededor de 500 requests concurrentes para esta query para cumplir el requerimiento de 2 segundos. Por lo observado se podría quizás mejorar los resultados aumentando la cantidad de instancias del subsistema y consumidores.

Req 10: Configuración de elección

GET http://localhost:3003/elections/{electionId}/config

Con MySQL distribuido:

Instancias QuerySubSystem	Query Consumers	Query Count	Res HTTP	Time (AVG ms)
1er Intento				
5	12	100	100	56.46
5	12	500	500	190.05
5	12	1000	1000	350.37
5	12	2000	2000	688.08
5	12	3000	3000	890.8
5	12	5000	5000	1585.91
5	12	6000	6000	1593.17
5	12	6500	6500	1965.48
2do Intento				
5	12	5000	5000	1469.66
5	12	6000	6000	2050.32
5	12	6500	5739	3105.46
Promedio				
5	12	5000	5000	1527.85
5	12	6000	6000	1821.45
5	12	6500	6119.5	2535.47

Por los resultados anteriores, a por debajo de 6000 request concurrentes se puede dar respuesta en menos de 2000 ms.

Requerimientos de 1 a 10 segundos

En las siguientes queries se mide el tiempo transcurrido desde la primera request hasta la respuesta de la última. Para los casos en que algunas requests no den el valor esperado, se indica con paréntesis mostrando la cantidad de 404 por no responder a las requests correctamente por sobrecarga.

Req 11: Frecuencia de Voto

GET http://localhost:3003/elections/{electionId}/vote-frequency

Con MySQL distribuido:

Instancias QuerySubSystem	**Query Consumers**	**Conexiones Simultaneas**	**Query Count**	**Res HTTP**	**Time (AVG ms)**	**Tiempo transcurrido (segundos)**
5	12	1000	21229	15989 [5240 x (404)]	361.67	9.2
6	15	3000	19669	19669	1312.18	9.41
6	15	5000	21586	21586	1865.09	9.98
6	15	4500	26825	26825	1591.21	10.68

El paso a 15 consumidores de queries y 6 APIs logró reducir a 0 la cantidad de respuestas erróneas del servidor, llegando a un máximo de 21586 respuestas en 10 segundos.

Req 12: Frecuencia por Circuito

GET http://localhost:3003/elections/{electionId}/circuit-info

Con MySQL distribuido:

Instancias QuerySubSystem	Query Consumers	Conexiones Simultaneas	Query Count	Res HTTP	Time (AVG ms)	Tiempo transcurrido (segundos)
6	15	100	5481	5481	80.74	10.05
6	15	100	4124	4093 [31 x (404)]	215.32	9.06
9	15	1000	4534	4446 [88 x (404)]	195.49	9.07
9	15	20	4178	4178	42.65	9.05
6	15	20	4970	4970	35.78	9.05

A pesar de aumentar la cantidad de APIs no se logró aumentar la cantidad de pedidos que se puede aceptar en 10 segundos consistentemente, por lo cual puede ser una limitación del ambiente en el que se está corriendo las pruebas (compartiendo recursos entre el que pide request y el que manda la response).

Se estaría llegando a poder responder a 20 conexiones simultáneas 4970 veces en 9.05 segundos, ya que en varios de los ejemplos no se logra responder a todos los pedidos correcta y consistentemente.

Req 13: Información por departamento

GET http://localhost:3003/elections/{electionId}/circuit-info

Con MySQL distribuido:

Instancias QuerySubSystem	Query Consumers	Conexiones Simultaneas	Query Count	Res HTTP	Time (AVG ms)	Tiempo transcurrido (segundos)
6	15	20	5450	5450	32.5	9.03
6	15	100	6385	6347 [38 x (404)]	139.26	9.05
6	15	90	7204	7148 [56 x (404)]	122.65	10.03
6	15	80	7009	7009	113.27	10.04

Se llega a la conclusión de que se pueden aceptar a un poco menos de 80 conexiones concurrentes cerca de 7000 pedidos en 10 segundos.

Requests variados entre todos los requerimientos:

Las siguientes pruebas son un caso más cercano a la realidad donde no solo se realizan pedidos a un endpoint, sino a varios en las queries.

Instancias:

Instancias QuerySubSystem	Query Consumers	Conexiones Simultaneas
6	15	20

Resultados:

Pedidos a Req 8,9,10,11,12 y 13

Conexiones de cada Requerimiento	Cant. Req 8	Cant. Req 9	Cant. Req 10	Cant. Req 11	Cant. Req 12	Cant. Req 13	Tiempo transcurrido (segundos)
10	1175	1915	12313	249	909	1048	10
20	1879 2 * (404)	1366 4 * (404)	8091	1529 1 * (404)	999 3 * (404)	1549 3 * (404)	9.43

Como comienzan a haber errores 404 porque express no es capaz de responder a todas las requests, consideramos que el límite de conexiones simultáneas posible para cada requerimiento es 20, donde cuando una request termina, se comienza una siguiente.

Postman

Se incluye en el archivo “ARQ_SOFT_TESTS.postman_collection.json” en el directorio Documentation una colección de pruebas de postman a las APIs del sistema. Cada método incluye una prueba con su resultado donde se puede visualizar el tiempo de respuesta de los pedidos. Para levantar las pruebas se ingresa a Postman y se elige “Importar”.

Guías de uso e instalación

Se incluyen dos guías de uso en formato markdown:

- CommandGuide.md
- QuickStartGuide.md

CommandGuide.md es una guía detallada donde cada script tiene una explicación de sus configuraciones y usos, mientras que QuickStartGuide.md explica brevemente los comandos necesarios para tener el sistema en un estado al que se puedan ejecutar pruebas.