

**Universidad ORT Uruguay**  
**Facultad de Ingeniería**

**Arquitectura de Software en la Práctica**  
**Obligatorio 2**

**Integrantes:**

Rafael Pirotto (199390)  
Victoria Moraes (207622)

**Docentes:**

Diego Romero  
Santiago Estrago

Jueves 10 de diciembre

# Índice

<b>Descripción de la arquitectura</b>	<b>4</b>
1.1. Vista de módulos	4
1.1.1. Vista de descomposición	4
Representación primaria	4
Catálogo de elementos	4
1.1.2. Vista de usos	6
Representación primaria	6
Catálogo de elementos	6
1.1.3. Vista de layers	7
Representación primaria	7
Catálogo de elementos	7
Decisiones de diseño	7
1.2. Vista de componentes y conectores	8
1.2.1. Vista de servicios	8
Representación primaria	8
Catálogo de elementos	8
Decisiones de diseño	9
Protocolo de comunicación: HTTP REST	9
Variables de entorno: archivo .env	9
Infraestructura de aplicación web: Express	9
Base de datos: MongoDB	9
Manejo de colas: Bull	9
Envío automático de emails: Nodemailer	10
Autenticación: JWT y bcrypt	10
Machine token: JWT	10
Caching: node-cache	10
Tareas programadas: Cron jobs con node-cron	11
Tareas programadas: Scheduling tasks	11
Tareas programadas creadas dinámicamente en base a las preferencias del usuario: Repeated jobs	11
1.2.2. Vista de secuencia	13
Representación primaria	13
1.3. Vista de asignación	14
1.3.1. Vista de despliegue	14
Representación primaria	14
Catálogo de elementos	14

Decisiones de diseño	14
<b>Justificaciones de diseño</b>	<b>16</b>
RNF1. Performance	16
RNF2. Confiabilidad y disponibilidad	16
RNF3. Configuración y manejo de secretos	16
RNF4. Autenticación y autorización	16
RNF5. Seguridad	17
RNF6. Código fuente	17
RNF7. Pruebas	17
RNF9. Estilo de arquitectura	20
RNF10. Integración continua	21
RNF11. Identificación de fallas	22
RNF12. Desplegabilidad	22
RNF13. Monitориabilidad	22
<b>Descripción del proceso de deployment y dev-ops</b>	<b>24</b>
3.1. Microservicios del backend	24
3.2. Frontend	27
<b>Links</b>	<b>29</b>
Link al repositorio de Github de la entrega	29
Hosting del frontend	29
Link al repositorio de Github del frontend	29
Hosting del microservicio “auth”	29
Link al repositorio de Github del microservicio “auth”	29
Hosting del microservicio “errors”	29
Link al repositorio de Github del microservicio “errors”	29
Hosting del microservicio “notifications”	29
Link al repositorio de Github del microservicio “notifications”	29
Hosting del microservicio “reports”	29
Link al repositorio de Github del microservicio “reports”	29
Hosting del microservicio “billing”	30
Link al repositorio de Github del microservicio “billing”	30
Hosting del microservicio “gateway”	30
Link al repositorio de Github del microservicio “gateway”	30
Link al repositorio de Github del SDK	30
<b>Anexo</b>	<b>31</b>
Resultados de pruebas de carga	31

Registrar usuario administrador	31
Login	31
Registrar usuario con invitación	31
Enviar invitación	31
Validar invitación	31
Crear api-key	31
Obtener preferencias	31
Editar preferencias	31
Monitorear salud del sistema (Auth)	32
Crear error	32
Obtener errores criticos	32
Editar error	32
Resolver error	32
Obtener errores	32
Obtener error	32
Monitorear salud del sistema (Errors)	32
Obtener reporte general	32
Obtener reporte top-10	33
Obtener reporte “no asignados”	33
Monitorear salud del sistema (Notifications)	33
Obtener factura	33
Obtener factura de un mes anterior	33

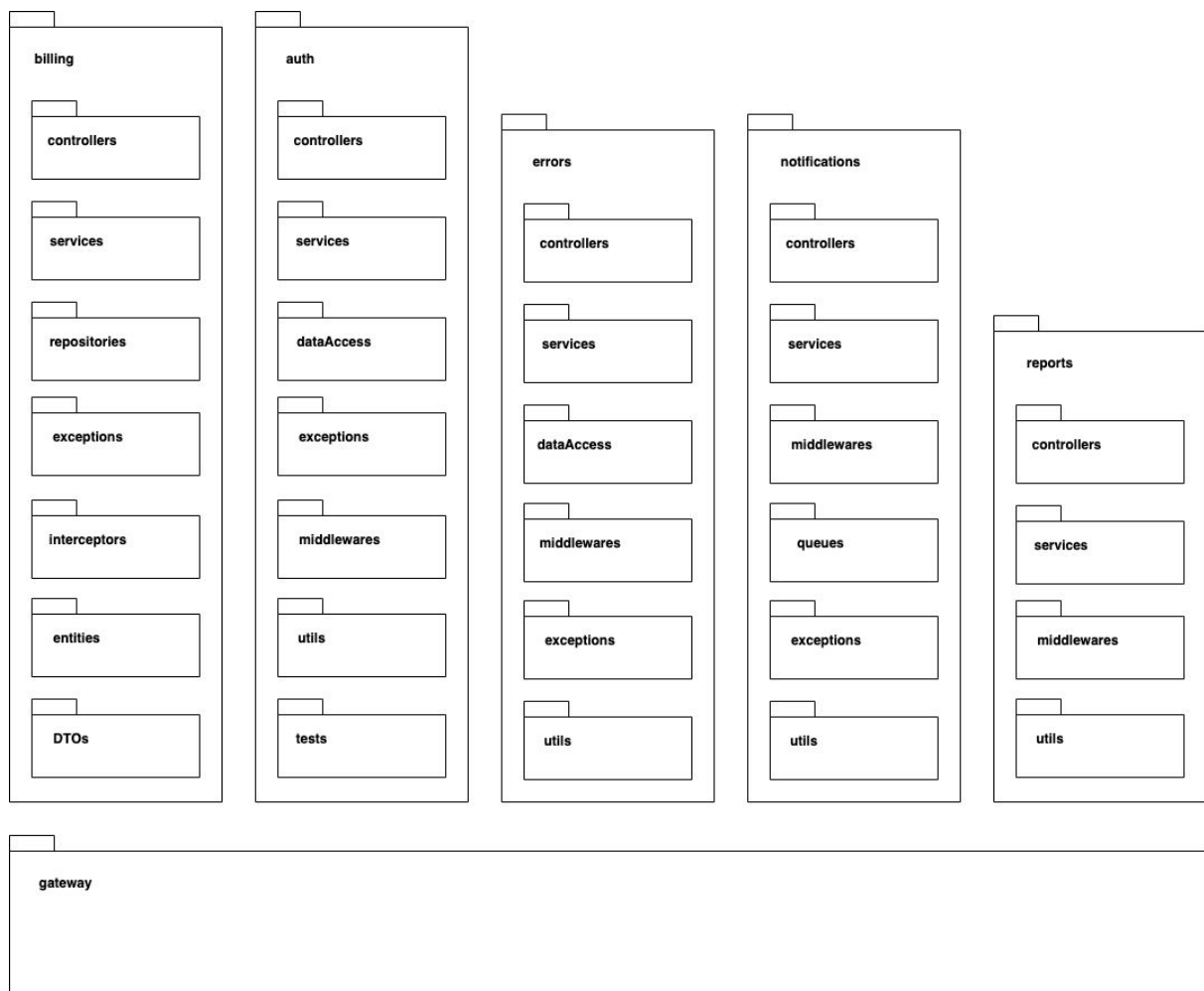
# 1. Descripción de la arquitectura

A continuación se presentan las diferentes vistas que describen la arquitectura de microservicios de nuestra aplicación. La justificación de la misma se encuentra más adelante en el documento en la sección “Justificaciones de diseño”.

## 1.1. Vista de módulos

### 1.1.1. Vista de descomposición

Representación primaria



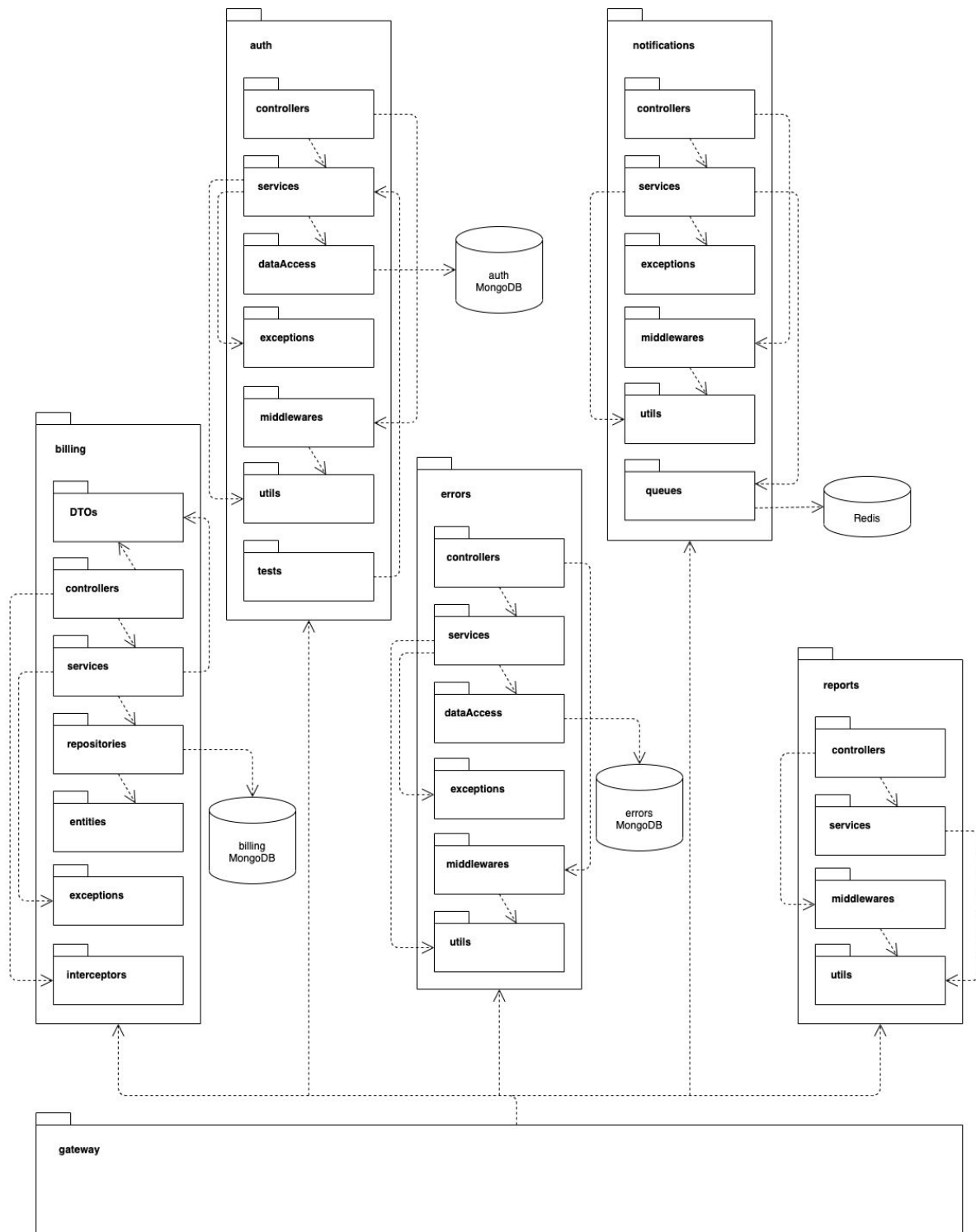
Catálogo de elementos

Elemento	Responsabilidades
gateway	Es el punto de entrada del sistema. Se encarga de

	redireccionar todos los requests a los servicios correspondientes.
auth	Se encarga de gestionar todo lo relativo a las cuentas de los usuarios.
notifications	Se encarga de gestionar todo lo relativo a las notificaciones enviadas por email.
errors	Se encarga de gestionar todo lo relativo a los errores.
reports	Se encarga de gestionar todo lo relativo a la generación de los diferentes reportes de errores.
billing	Se encarga de generar y almacenar las facturas.
middlewares	Contiene todos los middlewares requeridos para acceder a los distintos endpoints.
queues	Se encarga de gestionar todo lo relativo a las colas, permitiendo encolar, desencolar y procesar los objetos.
controllers	Se encarga de atender las request HTTP solicitadas a través de los endpoints.
services	Funciona como nexo entre los controllers y el acceso a datos. Contiene la lógica de negocio de la aplicación.
dataAccess	Se encarga de todo lo relacionado al acceso a datos. Contiene todas las operaciones necesarias para acceder a los datos y los modelos de los objetos que se almacenan en la base de datos (MongoDB).

## 1.1.2. Vista de usos

### Representación primaria



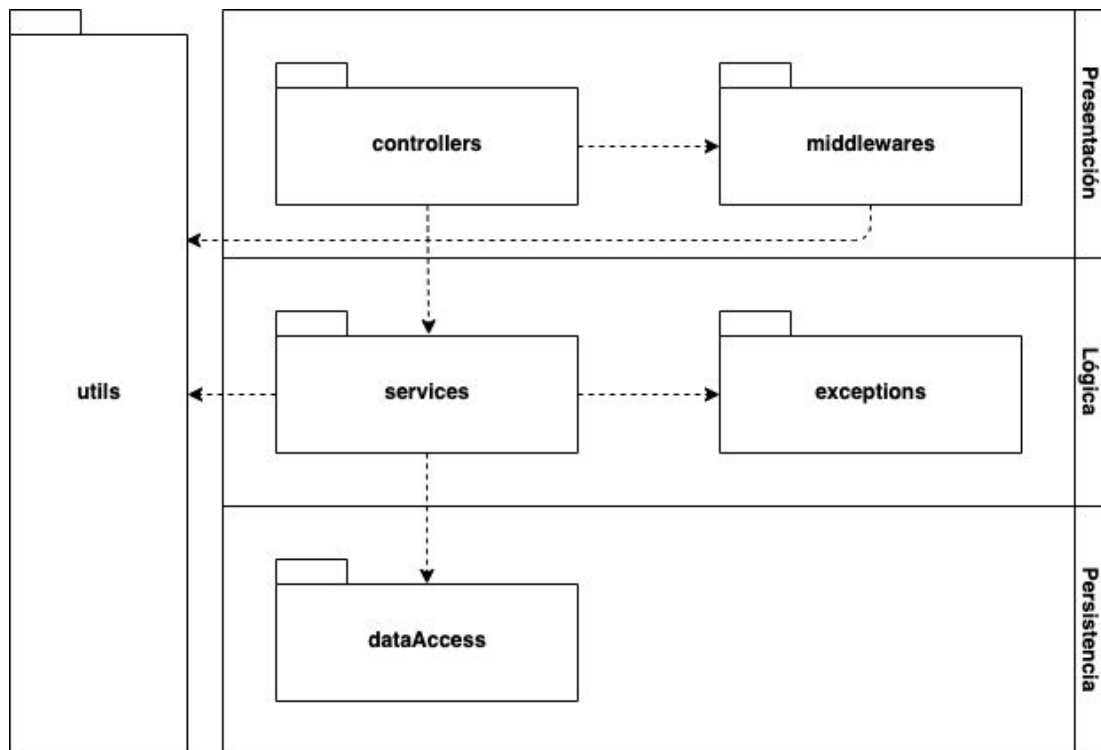
### Catálogo de elementos

Idem vista anterior.

### 1.1.3. Vista de layers

#### Representación primaria

Cada microservicio en su interior se divide en capas respetando la la estructura que se muestra a continuación.



#### Catálogo de elementos

Elemento	Responsabilidades
Capa de presentación	Contiene los puntos de entrada al backend.
Capa lógica	Contiene toda la lógica de negocio del sistema.
Capa de persistencia	Su función es el almacenamiento de los datos y el acceso a los mismos. Contiene los modelos de los objetos que se almacenan en la base de datos.

#### Decisiones de diseño

En esta vista podemos observar claramente la separación de responsabilidades internas del sistema. Priorizamos tener una buena separación de responsabilidades y capas bien definidas, con el fin de aumentar la mantenibilidad y el bajo acoplamiento, así como también contribuir con un

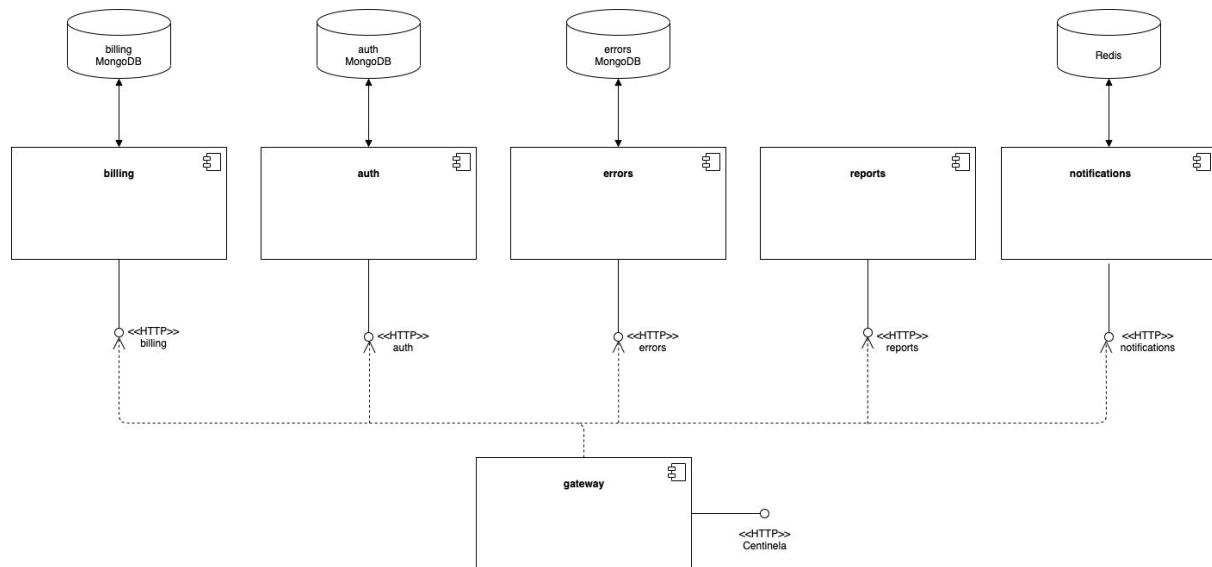


sistema escalable. Vemos como “utils” no pertenece a ninguna de las capas y que es común a todas.

## 1.2. Vista de componentes y conectores

### 1.2.1. Vista de servicios

#### Representación primaria



#### Catálogo de elementos

Elemento	Responsabilidades
auth	Este microservicio se encarga de gestionar todo lo relativo a las cuentas de los usuarios. Expone una interfaz que permite el consumo de sus servicios.
reports	Este microservicio se encarga de gestionar todo lo relativo a los reportes de errores de los usuarios. Expone una interfaz que permite el consumo de sus servicios.
errors	Este microservicio se encarga de gestionar todo lo relativo a los errores de los usuarios. Expone una interfaz que permite el consumo de sus servicios.
notifications	Este microservicio se encarga de gestionar todo lo relativo a las notificaciones enviadas por email. Expone una interfaz que permite el consumo de sus servicios.
billing	Este microservicio se encarga de gestionar todo lo relativo

	a las facturas de las organizaciones. Expone una interfaz que permite el consumo de sus servicios.
gateway	Este microservicio recibe todos los requests HTTP externos y los delega a cada uno de los microservicios.

## Decisiones de diseño

### Protocolo de comunicación: HTTP REST

Utilizamos el protocolo HTTP junto con un estilo de arquitectura REST, porque además de ser ideal para los requerimientos, es un protocolo seguro.

Esto hace que la interoperabilidad del sistema se vea ampliamente favorecida, ya que la aplicación adquiere la capacidad de funcionar con otros sistemas existentes sin restricciones de acceso o de implementación.

Mediante esta solución, logramos desacoplar la tecnología que se utiliza por detrás de la interfaz de los distintos microservicios, permitiendo el acceso a las distintas funcionalidades a través de URIs y parámetros predefinidos.

### Variables de entorno: archivo .env

Utilizamos variables de entorno en un archivo .env con el fin de desacoplar la configuración y evitar modificar el código de la aplicación cuando se requiera cambiar alguna de estas variables.

### Infraestructura de aplicación web: Express

La librería Express nos aportó una gran flexibilidad, así como también una amplia variedad de métodos de utilidad para HTTP. Esto nos permitió crear una API sólida y sencilla, al proporcionarnos una capa de características de aplicación web básica. Asimismo, su fácil integración y documentación clara nos trajo muchas ventajas a la hora de desarrollar el sistema.

### Base de datos: MongoDB

Una de las razones por las cuales utilizamos MongoDB es porque esta es muy compatible con nodejs al ser JavaScript el lenguaje para realizar las queries. Además estamos bastante familiarizados con ella, dado que la utilizamos en otras asignaturas, por lo que casi no necesitamos incorporar nuevos conocimientos o dedicar demasiado tiempo para poder utilizarla. Otra de sus ventajas es que cuenta con una amplia y detallada documentación. Finalmente creemos que su uso es adecuado para el problema dado que las entidades que se desean almacenar no están casi relacionadas como para optar por una base de datos relacional.

### Manejo de colas: Bull

Para el envío de emails decidimos implementar colas en el microservicio de notificaciones, de manera de que los emails se encolen y se vayan enviando poco

a poco y sin saturar el servidor. Al utilizar las colas y sus jobs logramos realizar estos procesos en segundo plano, evitando la mala experiencia del usuario, ya que el mismo no tendrá que esperar a que los emails sean enviados para seguir utilizando la aplicación.

Para el manejo de colas decidimos utilizar la biblioteca Bull, ya que nos provee un sistema de colas rápido y robusto, basado en Redis. Así la performance se ve favorecida, permitiéndonos procesar grandes cantidades de datos muy rápidamente.

### **Envío automático de emails: Nodemailer**

Para el envío automático de emails decidimos utilizar nodemailer, ya que es una herramienta muy sencilla de usar y que cumple con los requerimientos.

### **Autenticación: JWT y bcrypt**

Utilizamos la librería 'jsonwebtoken' ya que esta nos facilita la creación y verificación de tokens para los usuarios. En caso de realizar un request sin un token válido, el middleware responde antes de llegar al controlador del sistema. Además se utilizó la librería bcrypt de forma de guardar encriptadas las contraseñas de los usuarios.

### **Machine token: JWT**

Con el fin de proteger los endpoints de interacción interna entre los microservicios utilizamos un machine token, este es esencialmente un JWT creado a partir de una clave del sistema. En dichos endpoints se agregó un middleware específico que decodifica el token y verifica que la clave sea la del sistema. Dicha clave se encuentra almacenada como variable de entorno en el archivo .env del microservicio.

### **Caching: node-cache**

Para cumplir con el requerimiento no funcional de los errores críticos que establece que dicho endpoint debe responder en hasta 200ms para cargas de hasta 1200 req/m, implementamos cache con la librería node-cache. Esto nos permitió obtener la respuesta en un tiempo mucho menor que si se obtuviera accediendo a la base de datos del sistema directamente.

Para mantener la sincronización de los datos se utilizó la técnica “active expire”, esta nos pareció la más adecuada dado que era importante mantener los datos actualizados en todo momento. Es de esta forma, que cada vez que se crea un nuevo error, se obtienen los errores críticos y se guardan en cache para esa organización, además de borrarse el registro anterior. Luego, cuando se desean obtener los errores críticos, estos son tomados del caché por la clave de la organización.

Es importante aclarar que se realizaron pruebas de carga para este endpoint, sin embargo, como se detallará más adelante, nos ocurre en muchos de los endpoints que el tiempo de respuesta supera al requerido en la letra, al realizar numerosos accesos concurrentes al servidor de AWS. Por este motivo, si bien

implementamos esta táctica para reducir el tiempo, estamos limitados por la cota antes mencionada. En otras palabras, el caché si nos ayudó a reducir el tiempo de respuesta, pero debido a la latencia de red, el mismo supera los 200 ms.

### **Tareas programadas: Cron jobs con node-cron**

En el microservicio de errores, para cumplir el requerimiento de notificar al usuario cuando tiene errores asignados de más de dos días sin resolver (RF14), decidimos implementar tareas programadas. De esta forma, cada día se ejecuta una tarea que consulta en la base de datos si hay errores asignados con más de dos días sin resolver, si esto ocurre, se hace una llamada al microservicio de notificaciones para que el mismo se encargue de notificar al usuario por email. Para ello decidimos utilizar la librería node-cron, esta nos permite programar jobs o tareas que se ejecuten automáticamente en una fecha u hora determinada. Elegimos esta librería, dado que es muy sencilla de usar y nos permite cumplir con el requerimiento mencionado.

### **Tareas programadas: Scheduling tasks**

Con el objetivo de que los usuarios administradores puedan acceder a la factura de meses anteriores de su organización, necesitamos almacenar las facturas en la base de datos una vez que se terminaba el mes. Fue así que para realizar la tarea programada de guardar las facturas de las organizaciones a final de mes lo que hicimos fue usar las “Scheduling tasks” provistas por Spring Framework (dado que este microservicio fue desarrollado en Java Spring como se mencionó anteriormente). Esta nos permitió de manera sencilla ejecutar tareas programadas cada un cierto periodo de tiempo dado.

### **Tareas programadas creadas dinámicamente en base a las preferencias del usuario: Repeated jobs**

Con el fin de cumplir con el requerimiento de que el usuario pudiera optar por recibir una alerta en cierto momento del día (al horario que indique) con la cantidad de errores que se le asignaron durante ese día, lo que hicimos fue proveer un endpoint en el microservicio de notificaciones que se llame cada vez que el usuario elige un horario para que se le envíen los emails. Dicho endpoint lo que hace es encolar un job que se repite todos los días a la hora elegida por el usuario. Por otra parte, el job cuando se ejecuta, se encarga de consultar al microservicio de errores para saber si hubieron errores asignados para ese usuario en ese día, y, en el caso de que hayan, envía la notificación.

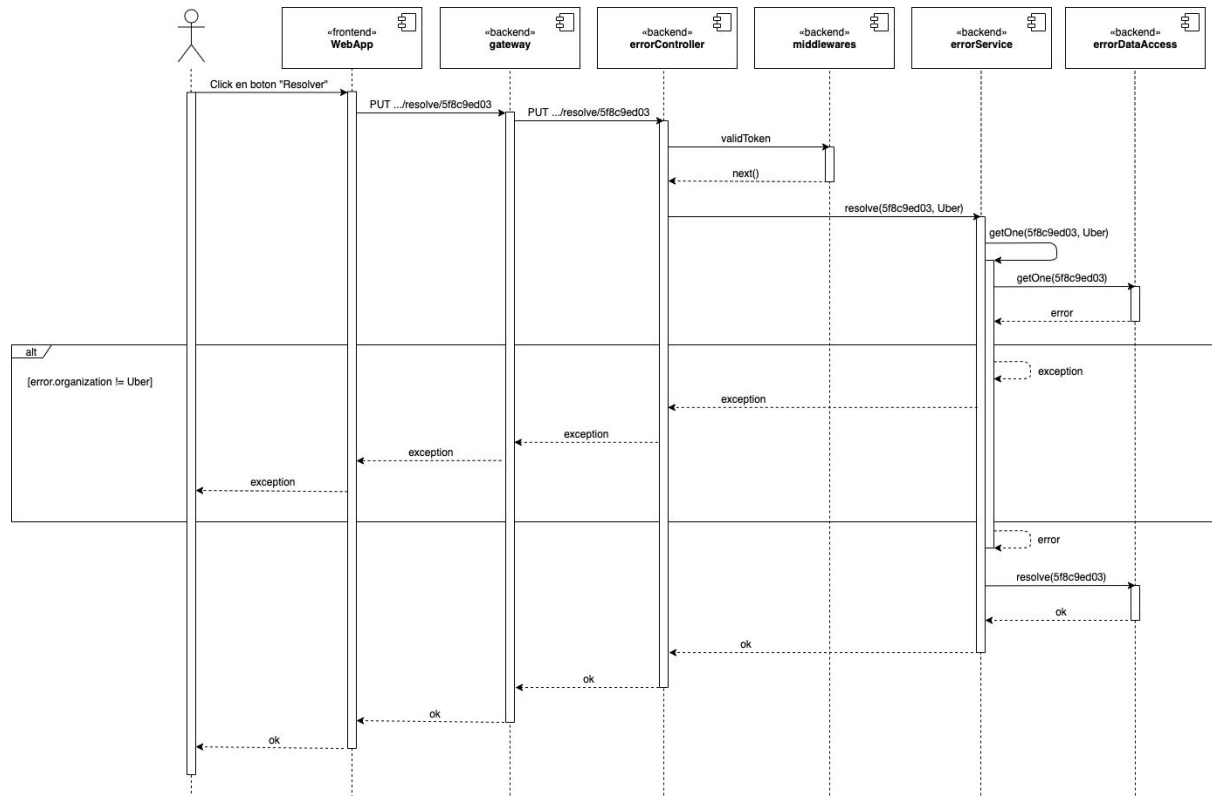
Por otra parte, se requería que la hora de envío de alerta pertenecieran al mismo huso horario que el usuario desarrollador. Esto lo resolvimos obteniendo la timezone del usuario desde el frontend con el método “getTimezoneOffset()” de la clase “Date” de Javascript. Una vez que detectamos la diferencia horaria entre la hora del usuario y la del servidor (siendo utc+0 la hora de aws), enviamos la información de la siguiente manera: si el usuario selecciono por ejemplo las 18 horas y su huso horario marca que tiene una hora menos respecto a la hora del

servidor, entonces se envia 18 como la hora seleccionada y un 1 como offset. Luego el backend recibe ambos datos y realiza la suma para convertir la hora de envío a la hora del servidor.

## 1.2.2. Vista de secuencia

En esta vista se puede observar el flujo de información a través de la infraestructura en tiempo de ejecución. En particular se detalla el flujo de una funcionalidad muy importante que es la de resolver un error, la cual es resuelta por el microservicio de errores. El flujo es análogo para el resto de los microservicios.

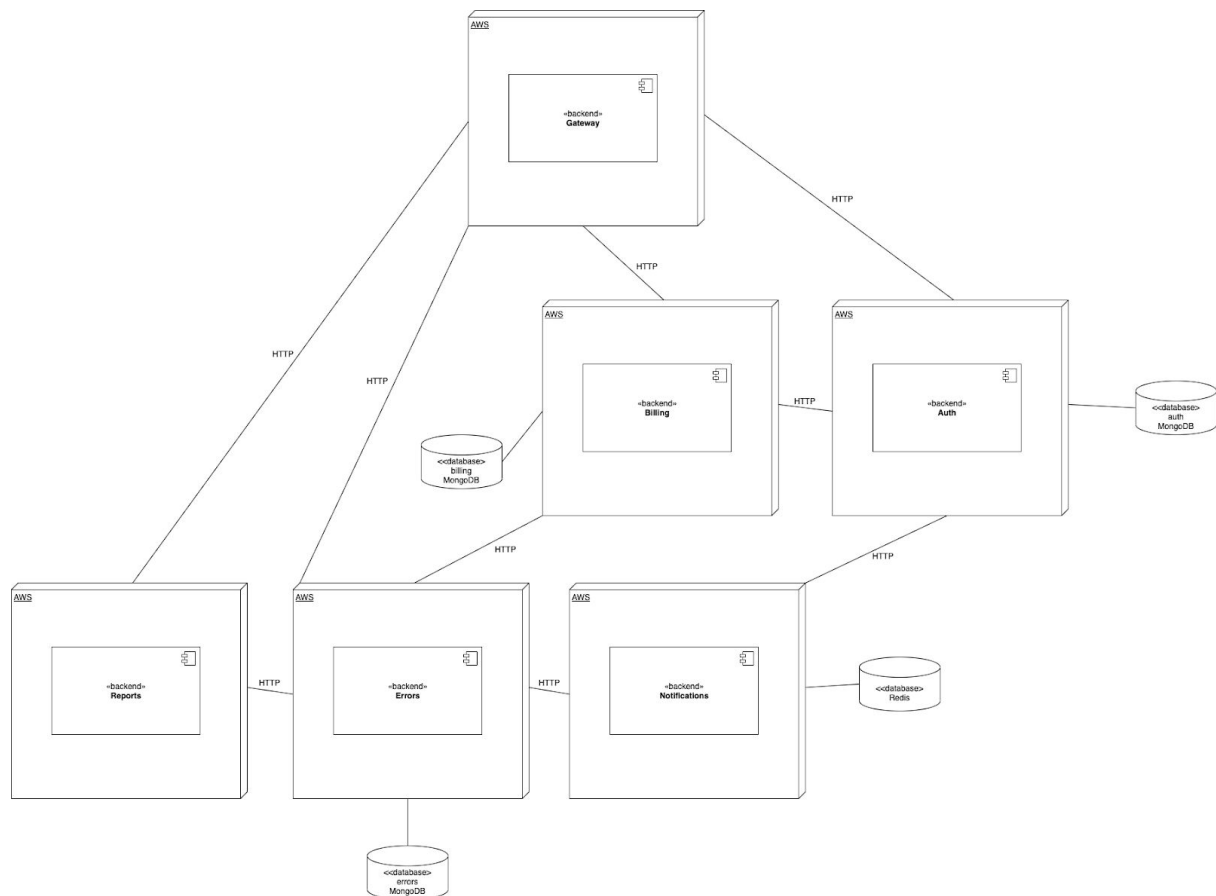
### Representación primaria



## 1.3. Vista de asignación

### 1.3.1. Vista de despliegue

#### Representación primaria



#### Catálogo de elementos

Elemento	Responsabilidades
AWS	Representa el ambiente donde se ejecuta el backend de los distintos microservicios. Estos se comunican mediante el protocolo HTTP. Los datos manejados por los microservicios son almacenados en sus respectivas bases de datos en MongoDB, y se utiliza Redis para la gestión de las colas de mensajes en el microservicio de notificaciones.

#### Decisiones de diseño

Para el deployment del backend de los microservicios se utilizó una infraestructura Platform as a Service. Esto fue principalmente debido a la inexperiencia en cuanto a dev-ops. Notamos que una infraestructura PaaS

requiere bastante menos configuración que utilizar una IaaS como es EC2. Es importante aclarar que se intentó realizar el deploy en EC2, pero dado que nos enfrentamos a muchas dificultades que no pudimos sortear, y a causa del poco tiempo, para esta instancia decidimos usar Elastic Beanstalk.



## 2. Justificaciones de diseño

### RNF1. Performance

Para cumplir con este requerimiento, hicimos que las tareas pesadas que no requieren que el usuario sepa su resultado o cuando fueron completadas, se realizarán en segundo plano. De esta forma, la respuesta es enviada al usuario, sin tener que esperar a que las dichas tareas se terminen de ejecutar. Para ello utilizamos colas de mensajes con la librería bull, como se mencionó anteriormente. Así pudimos correr background jobs para los casos en los cuales era necesario enviar emails, tanto cuando se envía una invitación, como cuando se crea un error y se requiere notificar por email a todos los usuarios de la organización que él mismo fue dado de alta.

De no haberse tomado esta decisión, existirían muchos casos en los que la respuesta tardaría mucho, dado que enviar un mail toma aproximadamente un segundo, por lo que se superaría ampliamente el tiempo de respuesta requerido (350 ms).

### RNF2. Confiabilidad y disponibilidad

Se implementó en cada microservicio el endpoint HTTP de acceso público requerido para monitorear la salud y disponibilidad del sistema. Este permite chequear el estado de la base de datos utilizada por las aplicaciones para persistir los datos (MongoDB) en los microservicios que necesitan almacenar datos, y el estado de Redis, que es la base de datos que utiliza Bull para el manejo de las colas de mensajes en el microservicio de notificaciones. Cabe aclarar que el microservicio de reportes no cuenta con este endpoint dado que no utiliza bases de datos.

### RNF3. Configuración y manejo de secretos

Todos los datos de configuración manejados en el código fuente se pueden especificar en tiempo de ejecución mediante variables de entorno al estar almacenadas en un archivo .env. Dicho archivo nunca fue subido al repositorio al ignorarse en el archivo gitignore. Así se logró desacoplar la configuración del código de la aplicación.

### RNF4. Autenticación y autorización

La aplicación cuenta con un control de acceso basado en roles, que distingue entre usuarios administradores y usuarios desarrolladores. De esta forma, se logra restringir el acceso a las funcionalidades según los permisos correspondientes al rol del usuario. Esto se implementó a través de middlewares que verifican los permisos antes de que la request llegue al controlador. Como se mencionó anteriormente, utilizamos JWT para la creación de tokens para los

usuarios. Estos luego nos permiten verificar la identidad del usuario y su rol para poder permitirle o no acceder a las distintas funcionalidades del sistema. El control de acceso está implementado de forma tal que los usuarios solo pueden acceder a información de su organización.

En cuanto a las claves de acceso, también fueron generadas utilizando JWT a partir del nombre elegido por el usuario, la organización del mismo y el secreto de nuestra aplicación. Estas son utilizadas para proteger los endpoints REST. Las mismas son pasadas en los requests por header a través de un par clave-valor.

## RNF5. Seguridad

Con respecto al manejo de errores, los microservicios estan implementados de manera tal, que responden con código 40X a cualquier request malformada o no reconocida, con código 50X frente a errores del servidor, y 200 en caso de éxito. De esta forma ningún endpoint queda expuesto.

Por otra parte, para la comunicación entre el frontend y el backend se utiliza HTTP que es un protocolo de transporte seguro. El mismo protocolo es utilizado para la comunicación entre los distintos microservicios.

En cuanto a la comunicación interna de cada microservicio, esta se realiza dentro de una red de alcance privado, dentro del VPC provisto por AWS.

## RNF6. Código fuente

Los microservicios fueron desarrollados utilizando nodejs, a excepción del microservicio de facturación el cual se desarrolló en Java Spring por requerimiento de letra. El frontend se implementó en React. Elegimos estas tecnologías, dado que ya las conocíamos y nos sentiamos cómodos trabajando con ellas. Además estas nos permiten cumplir con los requerimientos de la interfaz web así como también con los endpoints REST.

En cuanto al control de versiones utilizamos un repositorio en Git, en el cual utilizamos Gitflow para el manejo ordenado de las distintas branches que fuimos creando y utilizando durante el desarrollo.

El repositorio contiene en el archivo README.md una descripción con el propósito y alcance del proyecto, así como instrucciones para configurar un nuevo ambiente de desarrollo. También se incluye un README.md con el manual de instalación del SDK y un ejemplo de uso del mismo.

## RNF7. Pruebas

Se creó un script de generación de planes de prueba de carga utilizando la herramienta Apache jMeter. El mismo se realizó utilizando variables de entorno. Se configuró un grupo de hilos con 20 usuarios por segundo, para probar el requerimiento de performance que requería un tiempo promedio de respuesta de 350ms para cargas de hasta 1200 req/m.

### Thread Group

Name:

Comments:

Action to be taken after a Sampler error

☒ Continue ☐ Start Next Thread Loop ☐ Stop Thread ☐ Stop Test ☐ Stop Test Now

Thread Properties

Number of Threads (users):

Ramp-up period (seconds):

Loop Count: ☐ Infinite

☒ Same user on each iteration

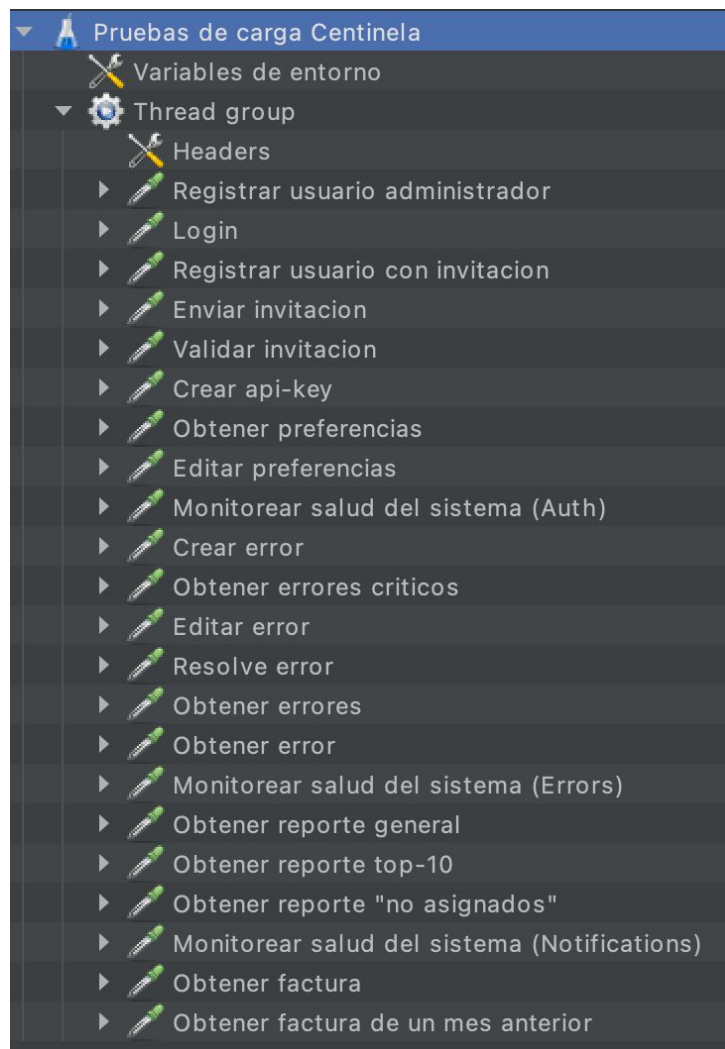
☐ Delay Thread creation until needed

☐ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Además se creó un reporte de árbol y otro de resumen por cada request HTTP a probar. Esto nos permitió visualizar tanto la correctitud de los requests y sus respuestas, como algunas estadísticas de los tiempos de respuesta, errores, etc.



En muchos casos, nos sucedió que el promedio del tiempo de respuesta de los requests excede el esperado, pero hay que tener en cuenta que en AWS va a haber unos milisegundos de latencia de red, dado que el servidor se encuentra en Virginia, USA. Creemos que una posible solución es escalar horizontal o verticalmente con el fin de lograr cumplir con los requisitos de tiempo. Nosotros en principio configuramos AWS con una capacidad pequeña dado que teníamos la restricción de los créditos que se nos otorgaron para consumir sus servicios, pero en caso de querer mejorar la escalabilidad esto podría aumentarse.

En el anexo se puede observar la evidencia de ejecución de las pruebas de carga para cada request, y en particular el reporte de resumen, para que se puedan apreciar en detalle los resultados. Por otra parte, el archivo .jmx, se encuentra en el repositorio.

En lo que respecta a las pruebas funcionales automatizadas para el RF9 y el RF10, estas fueron implementadas con los tests de Postman. Para ello se crearon test que validen casos de correcto funcionamiento, casos con errores, como inputs inválidos, y también en el caso del RF10, se validó que los errores estuvieran ordenados por severidad y que fueran cinco. La colección con las pruebas de

Postman y el ambiente con las variables de entorno utilizadas se encuentra en el repositorio.

## RNF9. Estilo de arquitectura

Como se pedía en este requerimiento, se implementó una arquitectura de microservicios. De esta forma, Centinela pasó a desarrollarse como una serie de pequeños servicios, cada uno ejecutándose de forma autónoma y comunicándose entre si mediante APIs HTTP. Cada servicio fue deployado de manera independiente en AWS.

Para el primero obligatorio se había implementado una arquitectura monolítica pero separada en módulos poco acoplados, y con responsabilidades diferentes. Esto nos ayudó a definir en esta instancia las fronteras de los servicios de forma de que estos estuvieran alineados con las distintas capacidades del negocio. Ya contábamos con una arquitectura bastante estable y estábamos familiarizados con el dominio del negocio. De esta manera, comenzamos a pensar cómo dividir los servicios de manera que cada uno de ellos fuera cohesivo e implementará un conjunto pequeño de funcionalidades.

Decidimos descomponer los servicios basándonos en las capacidades del negocio. Así fue que creímos adecuado contar con los siguientes microservicios:

- Microservicio “Auth”: Este cuenta con todas las funcionalidades referentes a los usuarios. Entre sus funcionalidades principales se incluyen el login y registro de usuario y la creación de api-keys.
- Microservicio “Errors”: Provee las funcionalidades referentes a la gestión de errores, entre ellas la creación, edición y resolución de errores.
- Microservicio “Reports”: Este obtiene información a partir del microservicio de errores, y permite obtener distintos tipos de reportes.
- Microservicio “Billing”: Permite generar la factura actual para una organización, obteniendo los datos de los microservicios “Auth” y “Errors”. Además, al final de cada mes, almacena automáticamente las facturas para cada organización, permitiendo al usuario consultar facturas de meses anteriores.
- Microservicio “Notifications”: Este microservicio se creó pensando en que el envío de notificaciones por mail era una funcionalidad requerida por varios de los otros microservicios, por lo que al equipo le pareció conveniente abstraerla a un nuevo microservicio que se encargara de gestionarlas. De esta manera evitamos repetir código, resolviendo el problema en un único lugar.
- Microservicio “Gateway”: Este microservicio se creó para resolver el problema de cómo acceden los clientes de Centinela a los microservicios individuales. Lo que hace entonces es definir una nueva API a partir de las APIs del resto de los microservicios. De esta manera logramos desacoplar la interfaz que ven los clientes de la implementación de los microservicios y sus APIs. También lo implementamos con el fin de evitar exponer los

servicios internos a los clientes externos, separando la API pública externa de las APIs internas de los microservicios.

El proceso de migración a microservicios se llevó a cabo de forma progresiva. Los microservicios se fueron extrayendo a medida que se realizaba un refactorio sobre el monolito. Luego se separaron las bases de datos y se crearon los nuevos esquemas en base a las nuevas necesidades. De esta forma, cada uno de los microservicios cuenta con una base de datos propia, evitando generar acoplamiento. De esta forma logramos que los datos de cada servicio se mantengan privados y solo accesibles a través de su API.

Es importante aclarar que el microservicio de reportes no cuenta con una base de datos, dado que todos los reportes son creados en el momento con los datos de los errores los cuales son obtenidos del microservicio de errores. Asimismo el microservicio de notificaciones tampoco utiliza una base de datos para almacenar información de la aplicación ya que su funcionalidad radica únicamente en el envío de notificaciones por email. Sin embargo, dado que este maneja colas de mensajes, utiliza una base de datos de Redis para almacenar los jobs que son encolados.

Esta arquitectura tiene el beneficio de permitir la heterogeneidad tecnológica, lo que nos permitió cumplir con el requerimiento de que uno de los microservicios estuviera implementando en un lenguaje diferente a los demás. Fue así que todos los microservicios que incluyen funcionalidades que ya estaban implementadas en el obligatorio anterior, están desarrollados en node js ya que era el lenguaje con el que veníamos trabajando. Sin embargo, el microservicio de facturación lo implementamos en Java Spring, dado que fue lo más conveniente para cumplir con el requerimiento mencionado, ya que era necesario implementarlo desde cero, al contener funcionalidades totalmente nuevas para el sistema. El lenguaje de este último microservicio fue elegido dado que el equipo ya contaba con algo de experiencia previa utilizando.

## RNF10. Integración continua

Para este requerimiento optamos por realizar las pruebas unitarias en los microservicios “Errors” y “Auth”. Para cumplir con lo pedido en la letra de cubrir tres requerimientos funcionales, realizamos pruebas sobre las siguientes funcionalidades: registrar un usuario y loguear un usuario, sobre el microservicio de Auth y finalmente el requerimiento de agregar un error sobre el microservicio de Errors.

Las pruebas fueron realizadas con la librería de node “mocha”. Para correrlas es necesario ejecutar el comando “npm test”. Finalmente, con el fin de no afectar la base de datos de producción, optamos por crear una base de datos de prueba, tanto para el microservicio de “Auth” como para el de “Errors”.

Una vez que tuvimos las pruebas funcionando correctamente nos enfrentamos al desafío de la integración continua. Para esto, usamos github actions. Esta funcionalidad de Github nos permite correr las pruebas en la nube luego de hacer un push a una cierta rama. Esto es útil en casos donde las pruebas pueden

demorar mucho, por lo que uno normalmente haría un push a develop por ejemplo y dejaría que las pruebas corran mientras avanza con otra cosa. Luego de que se terminen de correr las pruebas, Github nos envía un mail con el resultado. En caso de que haya habido una falla, se corrige y se comienza el ciclo de nuevo hasta que no hayan errores.

Con Github llevar a cabo este proceso es fácil, simplemente vamos al repositorio del proyecto al que queremos aplicarle la integración continua y Github nos provee varios templates según lo que queramos hacer. Dicho template nos permite agregar los comandos que queremos que se corran, por ejemplo “npm test”. También nos permite elegir las ramas sobre las que queramos que esta funcionalidad tenga efecto. Finalmente, una vez que lo configuramos, ya queda pronto para ser utilizado y tendrá efecto sobre el próximo push a las ramas seleccionadas.

## RNF11. Identificación de fallas

La identificación y detección de fallas se logró a través del servicio CloudWatch provisto por AWS. De esa forma se logró centralizar y retener los logs emitidos por todos los microservicios por un periodo de 24 horas. Se profundiza más en la sección de descripción del proceso de deployment y dev-ops.

## RNF12. Desplegabilidad

Con el fin de poder desplegar una versión de cualquier microservicio sin ocasionar downtime, el equipo decidió que iba a utilizar la estrategia de deploy blue-green.

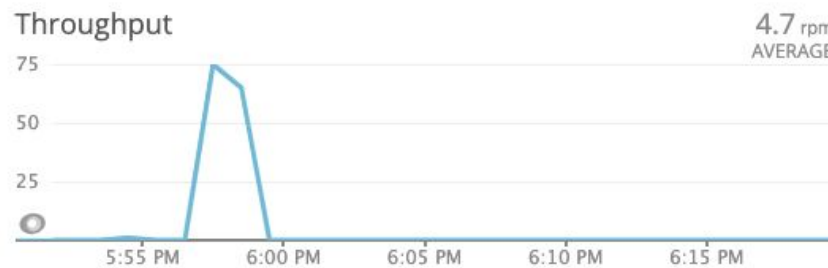
La idea es tener dos entornos idénticos. De esta forma, uno de los entornos aloja la aplicación de producción actual (entorno verde). Luego, cuando todo esté pronto para realizar un cambio en la aplicación y actualizarla, esto se hace en el otro entorno (la copia o entorno azul). Allí, se implementa la nueva versión de la aplicación, se ejecutan las pruebas, etc. Luego, si todo funciona correctamente, se cambia el balanceador de carga de forma que se dirija al nuevo entorno (entorno azul). Posteriormente se monitorea cualquier falla o excepción que pueda presentarse y, si todo se ve bien, eventualmente se podría cerrar el entorno con la versión anterior (entorno verde) y utilizarlo para organizar nuevos deploys. De lo contrario, si algo no funcionara bien, se puede retroceder rápidamente al primer entorno (entorno verde) apuntando el balanceador de carga hacia el mismo.

## RNF13. Monitoriabilidad

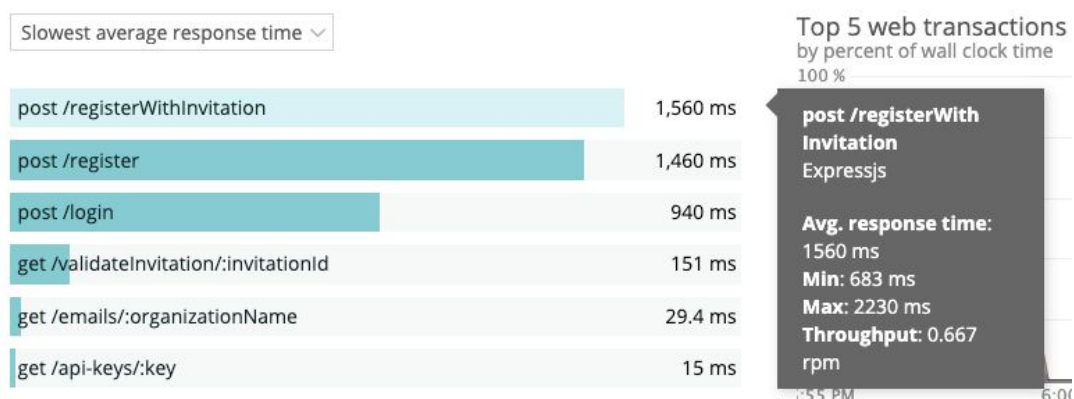
Para cumplir con este requerimiento utilizamos la herramienta Newrelic. Esta nos permite de una manera muy sencilla, tener acceso a una gran cantidad de estadísticas, métricas y gráficas sobre nuestros microservicios. La configuración es muy fácil y está debidamente explicada paso a paso en su documentación. Lo

único que hubo que hacer fue descargar la dependencia e instalarla. Una vez hecho eso, logramos que funcione correctamente.

En particular, la letra pedía obtener dos métricas. En primer lugar, las requests por minuto de cada microservicio y en segundo lugar el tiempo de respuesta de cada endpoint. A continuación se presentan algunos screenshots que evidencian el cumplimiento de lo requerido en la letra.



### Requests por minuto del microservicio de Errores.



### Tiempo de respuesta de cada endpoint del microservicio de Auth.



### 3. Descripción del proceso de deployment y dev-ops

#### 3.1. Microservicios del backend

El servicio de AWS que utilizamos para deployar cada microservicio fue Elastic Beanstalk. Para ello lo primero que hicimos fue crear una aplicación para cada uno.

Elastic Beanstalk > Create application

### Create new application

#### Application information

Application name

Maximum length of 100 characters, not including forward slash (/).

Description

#### Tags

Apply up to 50 tags. You can use tags to group and filter your resources. A tag is a key-value pair. The key must be unique within the resource and is case-sensitive. [Learn more](#)

Key	Value	
<input type="text"/>	<input type="text"/>	<button>Remove tag</button>

Add tag

50 remaining

CancelCreate

Luego creamos un ambiente también para cada uno donde utilizamos Docker como plataforma. Posteriormente subimos un .zip que contiene el Dockerfile (es por este motivo que previamente seleccionamos Docker como plataforma) y la carpeta raíz con el código fuente (sin la carpeta node\_modules en el caso de node). A continuación procedimos a configurar las variables de entorno (las cuales se encuentran detalladas en el archivo README.md presente en el repositorio de cada microservicio) con sus valores correspondientes como se muestra en la siguiente imagen.

**Environment properties**  
The following properties are passed in the application as environment properties. [Learn more](#)

Name	Value
ACCESS_TOKEN_SECRET	production
DB_URL	mongodb+srv://rafa:Centinela2020@cluster0.ic8e8.mongodb.net/<dbname>
FRONT_END_BASE_URL	http://centinela-bucket.s3-website-us-east-1.amazonaws.com
PORT	3002
REDIS_HOST	redis-16654.c52.us-east-1-4.ec2.cloud.redislabs.com
REDIS_PASSWORD	@Centinela2020
REDIS_PORT	16654
TOKEN_EXPIRATION	48h

Cancel Continue Apply

Finalmente, configuramos los logs en CloudWatch con un día de retención como se muestra en la imagen.

**Instance log streaming to CloudWatch Logs**  
Configure the instances in your environment to stream logs to CloudWatch Logs. You can set the retention to up to ten years and configure Elastic Beanstalk to delete the logs when you terminate your environment.

**Log groups**  
[/aws/elasticbeanstalk/CentinelaAuth-env](#)

**Log streaming**  
(Standard CloudWatch charges apply.)  
☒ Enabled

**Retention**  
1 days

**Lifecycle**  
Keep logs after terminating environment

De esta forma pudimos culminar el deploy de manera correcta. A continuación se adjunta prueba del correcto funcionamiento del mismo.

**CentinelaAuth-env**  
CentinelaAuth-env.eba-193gjp5t.us-east-1.elasticbeanstalk.com [🔗](#) (e-uczpiu3idi)  
Application name: Centinela-Auth

Refresh Actions

**Health**

Ok


Causes

**Running version**

centinela-auth-source-4

Upload and deploy

**Platform**

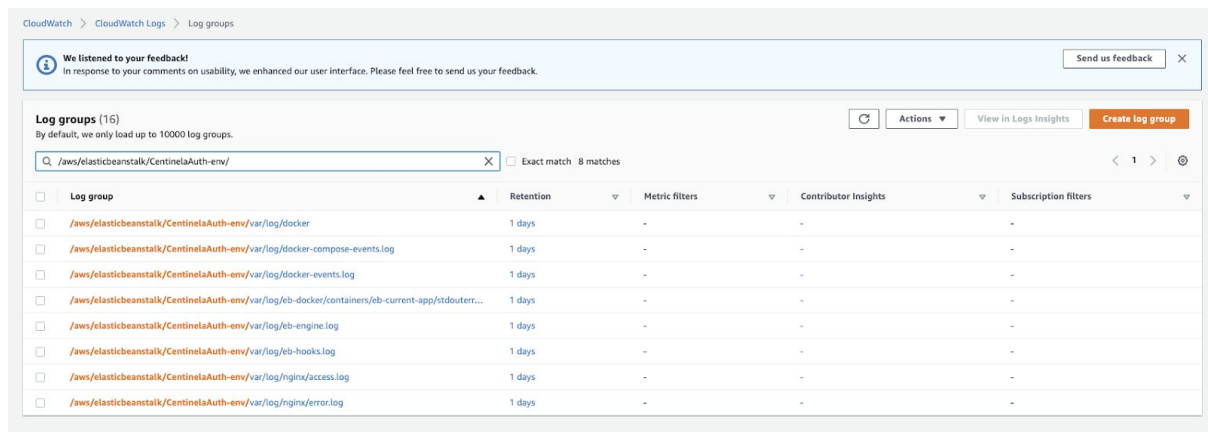


Docker running on 64bit Amazon Linux 2/3.2.1

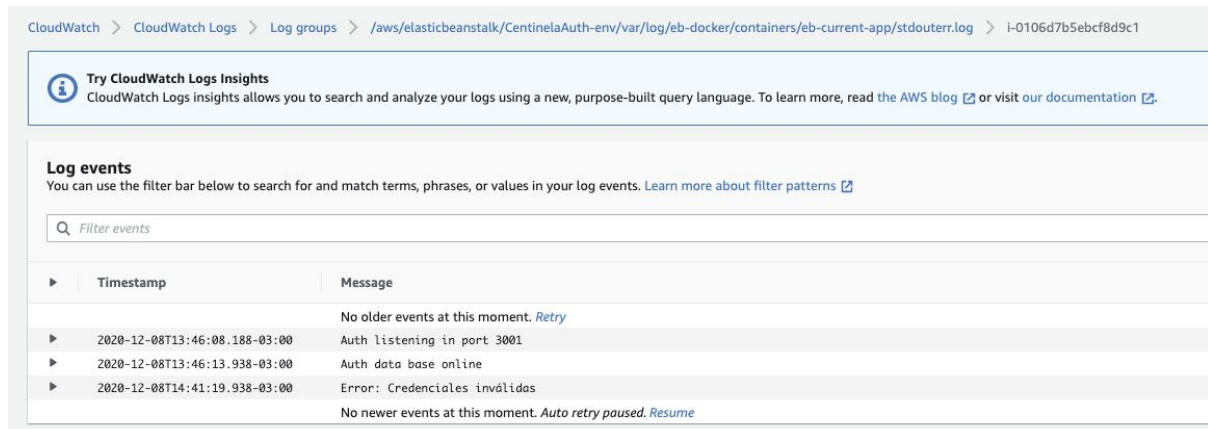
Platform update scheduled

Change

Para poder ver los logs, en Elastic Beanstalk, nos dirigimos al servicio de CloudWatch, luego a CloudWatch Logs y finalmente a Log Groups. Aquí podremos filtrar por microservicio para poder ver los logs que queramos.



Los logs nos permiten visualizar todos los registros (registros de docker, registros de nuestra aplicación, etc). Los logs de nuestra aplicación se ven como se muestra en la siguiente imagen. Para poder registrar los errores en CloudWatch utilizamos el standard output de nuestras aplicaciones a través del método “console.log()” en todas las ocasiones donde la aplicación lanzaba una excepción o registraba alguna acción.



## 3.2. Frontend

El servicio de AWS que utilizamos para deployar el frontend fue S3. En este servicio creamos un bucket y luego subimos los archivos de la carpeta build de nuestro proyecto. Para obtener esta carpeta ejecutamos el comando “npm run build” desde la terminal estando ubicados en la carpeta frontend de nuestro proyecto.

Amazon S3 > Create bucket

### Create bucket

Buckets are containers for data stored in S3. [Learn more](#)

#### General configuration

**Bucket name**

Bucket name must be unique and must not contain spaces or uppercase letters. [See rules for bucket naming](#)

**Region**

US East (N. Virginia) us-east-1

**Copy settings from existing bucket - optional**  
Only the bucket settings in the following configuration are copied.

Choose bucket

#### Bucket overview

Region	Amazon resource name (ARN)	Creation date	Access
US East (N. Virginia) us-east-1	arn:aws:s3:::centinela-frontend	December 8, 2020, 16:53 (UTC-03:00)	<a href="#">Objects can be public</a>

Objects | Properties | Permissions | Metrics | Management | Access points

Drag and drop files and folders you want to upload here, or choose **Upload**.

Objects (8)

Objects are the fundamental entities stored in Amazon S3. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Find objects by prefix

Name

Type

Last modified

Size

Storage class

<input type="checkbox"/>	asset-manifest.json	json	December 8, 2020, 17:03 (UTC-03:00)	849.0 B	Standard
<input type="checkbox"/>	favicon.ico	ico	December 8, 2020, 17:03 (UTC-03:00)	113.1 KB	Standard
<input type="checkbox"/>	index.html	html	December 8, 2020, 17:03 (UTC-03:00)	2.1 KB	Standard
<input type="checkbox"/>	manifest.json	json	December 8, 2020, 17:03 (UTC-03:00)	334.0 B	Standard
<input type="checkbox"/>	precache-manifest.b357057289ca7c1e07b024bf5b519756.js	js	December 8, 2020, 17:03 (UTC-03:00)	559.0 B	Standard
<input type="checkbox"/>	robots.txt	txt	December 8, 2020, 17:03 (UTC-03:00)	67.0 B	Standard
<input type="checkbox"/>	service-worker.js	js	December 8, 2020, 17:03 (UTC-03:00)	1.2 KB	Standard
<input type="checkbox"/>	static/	Folder	-	-	-

S3 nos resultó muy conveniente ya que para el frontend no hicimos uso de variables de ambiente, por lo que aprovechamos la funcionalidad de static website hosting y configuramos nuestro proyecto como tal. En caso de haber sido necesario utilizar variables de entorno, podríamos haber usado otro servicio como EC2 o Elastic Beanstalk.

Un aspecto a destacar, fue que tuvimos que apagar la opción que bloquea el acceso público, ya que de no hacerlo, no íbamos a poder acceder correctamente.

## Links

Link al repositorio de Github de la entrega

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Entrega>

Hosting del frontend

<http://centinela-frontend.s3-website-us-east-1.amazonaws.com>

Link al repositorio de Github del frontend

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Frontend>

Hosting del microservicio “auth”

<http://centinelaauth-env.eba-i93gjp5t.us-east-1.elasticbeanstalk.com>

Link al repositorio de Github del microservicio “auth”

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Auth>

Hosting del microservicio “errors”

<http://centinelaerrors-env.eba-aqxteytw.us-east-1.elasticbeanstalk.com>

Link al repositorio de Github del microservicio “errors”

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Errors>

Hosting del microservicio “notifications”

<http://centinelanotifications-env.eba-vz67vbyd.us-east-1.elasticbeanstalk.com>

Link al repositorio de Github del microservicio “notifications”

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Notifications>

Hosting del microservicio “reports”

<http://centinelareports-env.eba-mskafgbi.us-east-1.elasticbeanstalk.com>

Link al repositorio de Github del microservicio “reports”

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Reports>

Hosting del microservicio “billing”

<http://centinelabilling-env-3.eba-bdhmtqvv.us-east-1.elasticbeanstalk.com>

Link al repositorio de Github del microservicio “billing”

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Billing>

Hosting del microservicio “gateway”

<http://centinelagateway-env.eba-p8dbiuud.us-east-1.elasticbeanstalk.com>

Link al repositorio de Github del microservicio “gateway”

<https://github.com/ArqSoftPractica/Pirotto-Moraes-Gateway>

Link al repositorio de Github del SDK

<https://github.com/ArqSoftPractica/Pirotto-Moraes-SDK>

# Anexo

## Resultados de pruebas de carga

### 1. Registrar usuario administrador

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Registrar u...	20	2278	672	3562	1217,77	0,00%	4,7/sec	4,01	1,97	878,0
Total	20	2278	672	3562	1217,77	0,00%	4,7/sec	4,01	1,97	878,0

### 2. Login

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Login	20	698	308	1245	233,11	0,00%	4,5/sec	3,76	1,39	853,0
Total	20	698	308	1245	233,11	0,00%	4,5/sec	3,76	1,39	853,0

### 3. Registrar usuario con invitación

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Registrar u...	20	1672	903	2514	501,77	0,00%	4,2/sec	3,68	2,84	887,0
Total	20	1672	903	2514	501,77	0,00%	4,2/sec	3,68	2,84	887,0

### 4. Enviar invitación

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Enviar invit...	20	775	212	1720	541,85	0,00%	7,5/sec	2,48	3,85	337,0
Total	20	775	212	1720	541,85	0,00%	7,5/sec	2,48	3,85	337,0

### 5. Validar invitación

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Validar invi...	20	228	163	919	179,07	0,00%	4,9/sec	2,01	3,22	416,0
Total	20	228	163	919	179,07	0,00%	4,9/sec	2,01	3,22	416,0

### 6. Crear api-key

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Crear api-key	20	273	176	547	144,19	0,00%	5,3/sec	2,60	2,49	503,0
Total	20	273	176	547	144,19	0,00%	5,3/sec	2,60	2,49	503,0

### 7. Obtener preferencias

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener pr...	20	196	166	221	18,16	0,00%	6,1/sec	2,74	2,59	462,0
Total	20	196	166	221	18,16	0,00%	6,1/sec	2,74	2,59	462,0

### 8. Editar preferencias

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Editar pref...	20	267	172	689	106,18	0,00%	5,3/sec	2,39	3,25	462,0
Total	20	267	172	689	106,18	0,00%	5,3/sec	2,39	3,25	462,0



## 9. Monitorear salud del sistema (Auth)

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Monitorear ...	20	170	157	187	10,32	0,00%	5,3/sec	1,87	1,16	360,0
Total	20	170	157	187	10,32	0,00%	5,3/sec	1,87	1,16	360,0

## 10. Crear error

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Crear error	20	535	176	819	255,39	0,00%	4,7/sec	2,88	2,25	627,0
Total	20	535	176	819	255,39	0,00%	4,7/sec	2,88	2,25	627,0

## 11. Obtener errores criticos

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener err...	20	503	210	754	186,12	0,00%	4,2/sec	7,19	1,56	1763,0
Total	20	503	210	754	186,12	0,00%	4,2/sec	7,19	1,56	1763,0

## 12. Editar error

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Editar error	20	927	233	1412	323,10	0,00%	3,6/sec	1,29	2,19	362,0
Total	20	927	233	1412	323,10	0,00%	3,6/sec	1,29	2,19	362,0

## 13. Resolver error

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Resolve error	20	607	180	1236	343,63	0,00%	3,6/sec	1,28	1,69	362,0
Total	20	607	180	1236	343,63	0,00%	3,6/sec	1,28	1,69	362,0

## 14. Obtener errores

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener err...	20	1994	1518	2391	287,39	0,00%	2,3/sec	472,22	0,94	214455,6
Total	20	1994	1518	2391	287,39	0,00%	2,3/sec	472,22	0,94	214455,6

## 15. Obtener error

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener error	20	522	176	792	191,17	0,00%	3,2/sec	2,15	1,41	687,0
Total	20	522	176	792	191,17	0,00%	3,2/sec	2,15	1,41	687,0

## 16. Monitorear salud del sistema (Errors)

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Monitorear ...	20	204	161	312	49,22	0,00%	3,5/sec	1,22	0,76	360,0
Total	20	204	161	312	49,22	0,00%	3,5/sec	1,22	0,76	360,0

## 17. Obtener reporte general

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener re...	20	545	261	805	162,50	0,00%	2,3/sec	1,02	1,06	448,0
Total	20	545	261	805	162,50	0,00%	2,3/sec	1,02	1,06	448,0

## 18. Obtener reporte top-10

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener re...	20	497	293	838	137,39	0,00%	3,5/sec	1,19	1,50	344,0
Total	20	497	293	838	137,39	0,00%	3,5/sec	1,19	1,50	344,0

## 19. Obtener reporte “no asignados”

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener re...	20	516	208	799	197,13	0,00%	3,7/sec	1,26	1,60	344,0
Total	20	516	208	799	197,13	0,00%	3,7/sec	1,26	1,60	344,0

## 20. Monitorear salud del sistema (Notifications)

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Monitorear ...	20	173	162	249	21,98	0,00%	3,0/sec	1,03	0,67	357,0
Total	20	173	162	249	21,98	0,00%	3,0/sec	1,03	0,67	357,0

## 21. Obtener factura

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener fa...	20	669	321	1052	226,20	0,00%	3,6/sec	1,82	1,50	523,0
Total	20	669	321	1052	226,20	0,00%	3,6/sec	1,82	1,50	523,0

## 22. Obtener factura de un mes anterior

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Están...	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de B...
Obtener fa...	20	191	175	227	11,28	0,00%	6,1/sec	1,51	2,67	254,0
Total	20	191	175	227	11,28	0,00%	6,1/sec	1,51	2,67	254,0