

# Universidad ORT Uruguay

## Facultad de Ingeniería

Ing. Bernard Wand-Polak

### Centinela

*Arquitectura de Software en la Práctica*

*Obligatorio I*

Manuel Larrosa - 175136

Sebastian Zawrzykraj - 180110

Matias Settimo - 152946

### Profesores

Leticia Esperon

Alejandro Tocar

**2020**

# Declaración de autoría

Nosotros, Manuel Larrosa, Matias Settimo y Sebastián Zawrzykraj, declaramos que el trabajo que se presenta en la presente obra es de nuestra propia mano. Pudiendo asegurar que:

- La obra fue producida en su totalidad mientras realizamos la materia Arquitectura de Software en la Práctica.
- 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 contribuido por otros, y que fue contribuido por nosotros.
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Manuel Larrosa



Matias Settimo



Sebastian Zawrzykraj

# Indice

<b>1. Introducción</b>	4
1.1. Antecedentes	4
1.2. Propósito del documento	4
1.3 Propósito de la arquitectura	4
<b>2. Análisis</b>	5
2.1. Descripción de requerimientos	5
2.2. Desafíos y decisiones de diseño	6
2.3 Bosquejo de solución	7
2.3.1 Actores	7
2.3.2 Descripción general del diseño de la solución	7
<b>3. Documentación de la arquitectura</b>	9
3.1. Vista de módulos	9
3.1.1. Vista de Descomposición	9
3.1.1.1. Representación primaria	9
3.1.1.2. Catálogo de Elementos	10
3.1.2. Vista de Usos	11
3.1.2.1. Representación Primaria	11
3.1.2.2. Catálogo de Elementos	11
3.1.3. Modelo de Datos	12
3.1.3.1. Representación Primaria	12
3.1.3.2. Catálogo de Elementos	12
3.2. Vista de componentes y conectores	13
3.2.1 Vista de componentes y conectores del sistema	13
3.2.1.1 Representación primaria	14
3.2.1.2 Catálogo de elementos	15
3.2.1.3 Comportamiento	16
Creación de un bug	16
Creación de una invitación y registro	17
3.3. Vista de Asignación	18
3.3.1. Vista de Despliegue	18
3.3.1.1. Representación Primaria	18
3.3.1.2. Catálogo de Elementos	18
<b>4. Justificaciones de diseño</b>	20
4.1. Performance y testing	21
4.2. Confiabilidad y disponibilidad	26
4.3. Configuración y manejo de secretos	27
4.4. Autenticacion y autorizacion	27

4.5. Seguridad	28
4.6. Código fuente	28
4.7. Pruebas	28
4.8. Identificación de fallas	29
<b>5. Proceso de deployment y dev-ops</b>	<b>30</b>
<b>6. Resultados y conclusiones</b>	<b>39</b>
6.1 Proceso de trabajo y requerimientos funcionales	39
6.2 Requerimientos no funcionales	40
<b>7. Anexos</b>	<b>41</b>
7.1. Requerimientos y restricciones	42
7.1.1. Requerimientos Funcionales	42
7.1.2. Requerimientos No Funcionales	44
7.1.3. Restricciones	46
7.2. Sprints realizados	47
7.2.1 Sprint 0	47
7.2.1 Sprint 1	47
7.2.1 Sprint 2	47
7.2.1 Sprint 3	48
7.3. Api rest	48
7.4. URLs de la aplicación	49

# 1. Introducción

## 1.1. Antecedentes

A pedido de la empresa EnvíosYa, se desea implementar un sistema de seguimiento de errores que pueda ser contratado como un servicio SaaS, donde la empresa solicita abonar de acuerdo al volumen de uso de la herramienta.

Por tal motivo se desarrollará una solución, desde ahora llamada Centinela, que pueda resolver los requerimientos de la empresa mencionada, y a la vez pueda ser ofrecida a otros clientes, bajo el mismo régimen de contratación.

Dicho sistema proveerá las siguientes funcionalidades como registro de errores y bugs desde los sistemas de la empresa (con identificación de prioridad y ambiente que lo reporta), administración de usuarios (administradores y desarrolladores), administración de bugs (edición, asignación a usuarios, resolución, etc), notificaciones por correo y distintos reportes sobre los errores generados.

## 1.2. Propósito del documento

Este documento no es una especificación de requerimientos, sino un análisis que se ampara en los mismos.

El propósito tampoco es brindar un diseño de bajo nivel, sino proveer una especificación completa del diseño arquitectónico de Centinela, buscando detallar distintos aspectos de su implementación, decisiones de diseño, y tácticas de despliegue.

También se brindarán algunos detalles de la gestión del trabajo, herramientas utilizadas y otras decisiones tomadas por el equipo.

## 1.3 Propósito de la arquitectura

La arquitectura del sistema, que será la base para su elaboración, está basada en el análisis de sus requerimientos.

Toda decisión está amparada en los mismos, buscando favorecer los atributos de calidad solicitados por el cliente y deseados por el equipo de trabajo, a efectos de ofrecer una solución que resuelva de la mejor forma el problema, teniendo en cuenta ventajas y problemas que pueda introducir (*trade-offs*).

## 2. Análisis

### 2.1. Descripción de requerimientos

La especificación detallada de los requerimientos se encuentra en el apartado anexo [7.1 Requerimientos y restricciones](#).

Se pasará a una breve descripción de los mismos, a efectos de poder explicar algunas decisiones tomadas para el diseño de la solución.

#### 1. Requerimientos funcionales

- a. Registro web de usuarios administradores y desarrolladores (los usuarios administradores podrán crear usuarios y editar bugs; los usuarios desarrolladores solo podrán cerrar bugs).
- b. Autenticación de usuarios.
- c. Gestión de claves para que las distintas aplicaciones puedan crear errores.
- d. Gestión web de los errores para los usuarios (listado, visualización, edición)
- e. Reportes estadísticos de errores (web y rest).

#### 2. Principales requerimientos no funcionales y restricciones

- a. Performance con tiempos de respuesta promedio debajo de 350 ms. para cargas hasta 1200 req/m.
- b. Informe del funcionamiento del sistema mediante un endpoint HTTP para poder verificar la disponibilidad del sistema.
- c. Configuración de variables de entorno para los distintos ambientes para mejorar la seguridad y modificabilidad del sistema.
- d. Seguridad de acceso a la información de acuerdo a las distintas responsabilidades de tipo de usuario, aplicaciones, etc.
- e. Interfaz web, endpoints REST para la comunicación con el backend, utilización de un repositorio en Github con archivos README.md descriptivos.
- f. Centralizado de logs.
- g. Pruebas de carga con Apache JMeter.
- h. La aplicación debe ser desarrollada como un monolito.

## 2.2. Desafíos y decisiones de diseño

La solución requerida nos presentó varios desafíos, tanto tecnológicos como de diseño, en los cuales el equipo tenía nula experiencia previa.

En cuanto a los desafíos de diseño de la solución:

1. Diseñar sistemas distribuidos en la nube, con sus distintas particularidades y estrategias, como las recomendaciones *twelve factors*.
2. Generación de una solución monolito, que pueda atender los requerimientos, principalmente no funcionales como performance, disponibilidad, confiabilidad.

En cuanto a requerimientos tecnológicos:

1. Aprender distintas estrategias de despliegue en AWS (*Amazon Web Services*), entender su funcionamiento y cuál es la que más nos conviene, teniendo como restricción la cuenta *Educate* de AWS proporcionada por la Universidad Ort Uruguay, a efectos de realizar pruebas.
2. Tomar decisiones en cuanto a frameworks a utilizar para frontend y backend, herramientas de monitoreo, y herramientas gráficas para visualización de estadísticas.

Esto nos llevó a elegir algunas herramientas que ya conocíamos y confiábamos, tanto por su productividad como por ser orientadas a la web: NodeJs para backend, y Angular para frontend, entendiendo que con las mismas se nos facilitaría la implementación de la solución, pudiendo enfocarnos en las decisiones de nivel arquitectónico.

También tomamos la decisión de investigar la posibilidad de incorporar algún template de frontend que nos permita enfocarnos más en la usabilidad del mismo y no en la estética, dado que tampoco contábamos con un diseñador de interfaces experimentado en el equipo.

Por tal motivo, luego de una investigación previa, se tomó la decisión de experimentar con el template Nebular, de la empresa Akveo (<https://akveo.github.io/nebular/>). El mismo ofrece muchos componentes gratuitos desarrollados en angular, que simplificaron algunas tareas, como la generación de un menú, visualización de estadísticas, generación de estilos css homogéneos y atractivos, y algunos componentes customizados, como una tabla que nos ofrecía una manipulación de los eventos y comportamiento hacia el usuario bastante amigable, para la vista de bugs.

También se fueron tomando varias decisiones importantes a nivel del deploy, que dadas algunas restricciones que tenemos por la cuenta educativa, favorecieron la simplicidad del mismo, tal vez en detrimento de algún requisito de calidad, pero que entendemos podremos mejorar a futuro. Esto se detalla más adelante, en el apartado [5. Proceso de deployment y dev-ops](#).

2.3 Bosquejo de solución

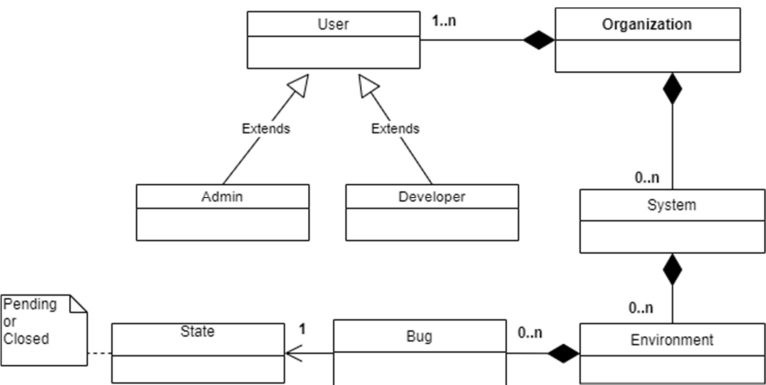
2.3.1 Actores

Para poder comprender cuál es la hoja de ruta hacia donde vamos, entendemos necesario un bosquejo rápido de la solución, a efectos de identificar actores externos del sistema, cómo se relacionan, y algunas decisiones de diseño tomadas para cumplir con las especificaciones.

En primer lugar, tenemos identificados los siguientes actores:

- **Organizaciones**, que quieren utilizar nuestro sistema para reportar **Bugs** de sus distintos **Sistemas**, y los **Ambientes** de esos sistemas que son los que pueden generar un bug (un ambiente de un sistema, podría ser por ejemplo el ambiente de Desarrollo, o de Producción).
- Usuarios **Desarrollador** y **Administrador**.

De esta manera presentamos el siguiente diagrama de dominio reducido, que busca representar qué es lo que necesita la organización que nos contrata:



Dado el requerimiento **RF1**, donde una organización es creada cuando el primer usuario administrador se registra en el sistema, tomamos como supuesto que cada usuario se pasará a identificar por su correo, y que este será único para todo el sistema, no pudiendo registrarse en más de una organización con el mismo correo. A su vez, cada organización para existir, requerirá de al menos un usuario.

2.3.2 Descripción general del diseño de la solución

Observando los requerimientos funcionales, pasamos a mostrar cómo se han resuelto los más relevantes.



Para esto se recomienda en primer lugar observar el [diagrama de componentes](#) detallado más adelante, que muestra cuáles son los principales sistemas que interactúan en la solución, la cual se compone de 2 componentes principales:

1. Una aplicación backend monolítica, representada en el diagrama como Centinela
2. Una aplicación **SPA** (*Single Page Application*), representada como CentinelaCli que se ofrece como frontend.

La aplicación frontend se encargará de interactuar con los usuarios, donde podrán ver listados y administrar bugs, administrar usuarios, y ver estadísticas (**RF1**, **RF2**, **RF3**, **RF4**, **RF5**, **RF6**, **RF7**, **RF8**, **RF11**).. La misma se comunicará con el backend a través de un cliente Rest de Angular, cumpliendo con la restricción **RES2**, de entablar comunicaciones REST.

La aplicación backend, en cambio, tendrá a cargo no solo la responsabilidad de atender distintos clientes frontend, sino que a la vez atenderá las solicitudes de registro de bugs de todas las organizaciones, y ofrecerá una endpoint con un reporte de errores (**RF9**, **RF10**).

Por tal motivo se tomó la decisión de utilizar 2 colas de mensajería para atender los procesos más importantes y que puedan generarle más estrés al sistema (la creación de bugs y el envío de correo), y a su vez se decidió separar algunas responsabilidades de la aplicación backend, manteniendo la solución monolítica, pero separándose en 3 servicios distintos, cada una con su puerto particular (con esto se busca cumplir con los requerimientos **RNF1** y **RNF2**) :

1. **Centinela**: finalmente la que se detalla en el diagrama de componentes, se encarga de atender todos los requests, encolar bugs para su posterior procesamiento, persistir y extraer información de la base de datos, y encolar mensajes de correo a usuarios.
2. **Sender**: un servicio que se encarga de extraer de la cola de mensajería los correos pendientes a enviar a usuarios y procesarlos.
3. **Monitoring**: un servicio que se encarga de monitorear el estado general del sistema: verifica estado del Sender, de la base de datos, las colas de mensajería y del Centinela.
4. **BugProcessor**: fue pensado como un proceso worker que realiza la tarea de atender los trabajos encolados por centinela para el procesamiento de bugs. Este desencola de la cola de trabajos "Bug Queue" y luego de la validación correspondiente, almacena los bugs en la base de datos del sistema.

Para ver ejemplos de cómo interactúan los componentes del sistema, puede entrar a los apartados [creación de un bug](#) y [creación de una invitación y registro](#), que se encuentran más adelante.

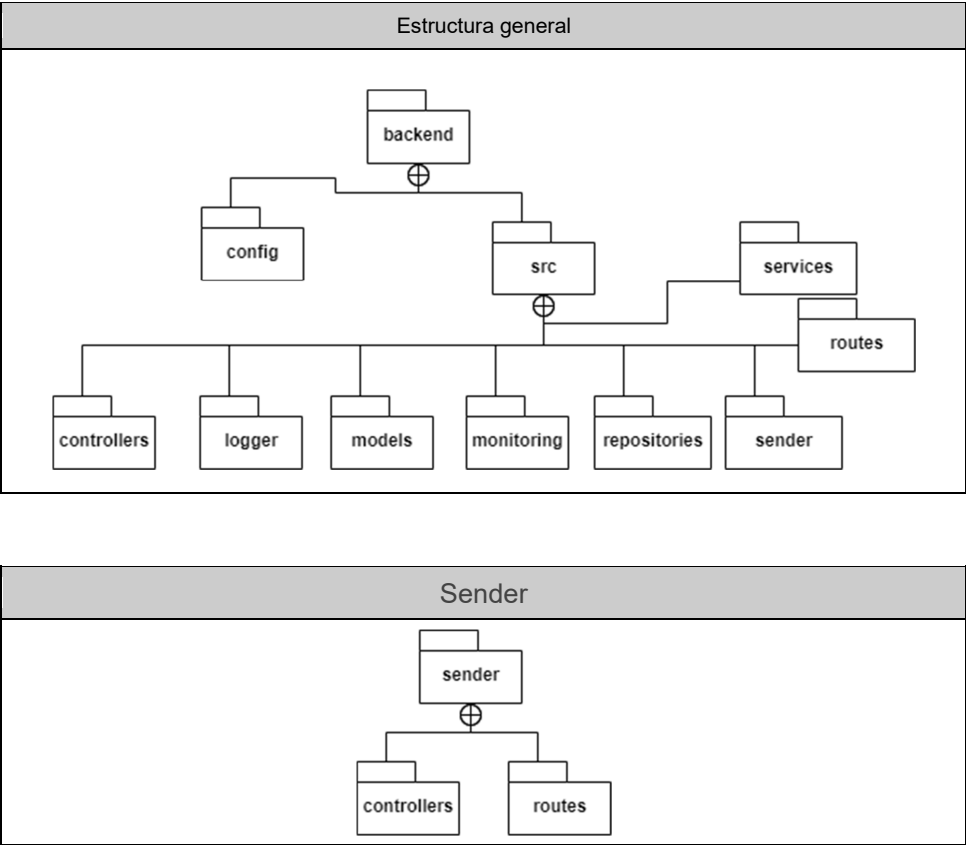
### 3. Documentación de la arquitectura

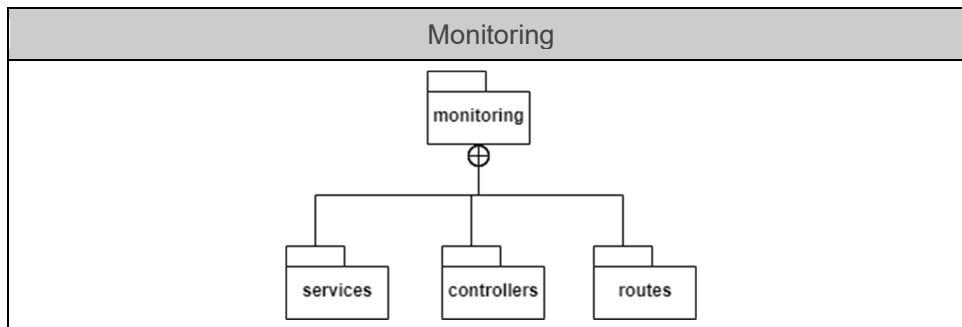
#### 3.1. Vista de módulos

Este tipo de vistas incorporan decisiones sobre cómo se estructura el sistema en un conjunto de unidades de código o datos.

##### 3.1.1. Vista de Descomposición

###### 3.1.1.1. Representación primaria





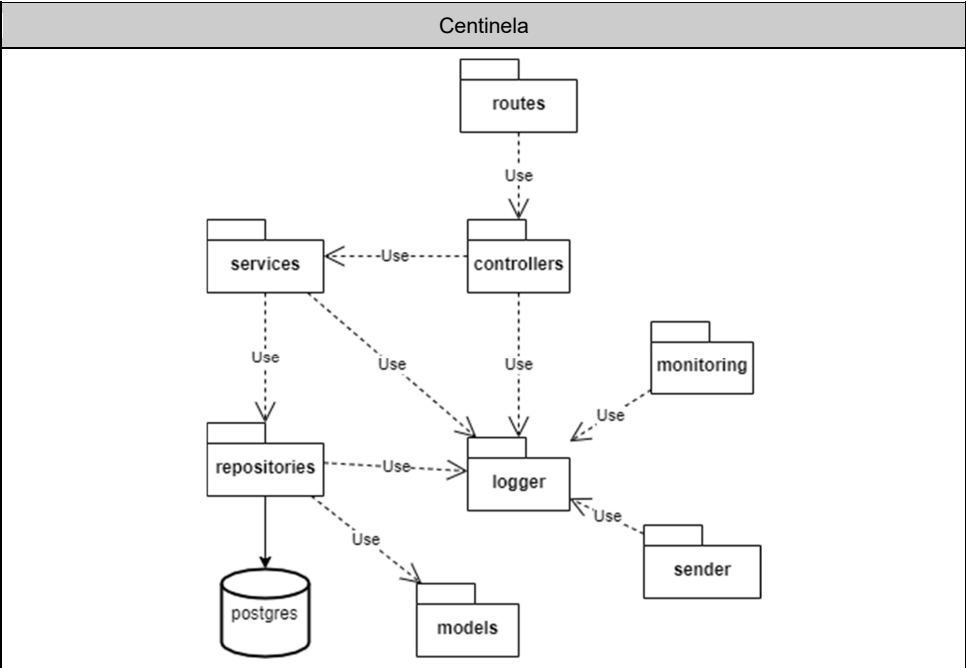
### 3.1.1.2. Catálogo de Elementos

Elemento	Responsabilidad
backend	Contiene los controllers, modelos, servicios, repositories y demás archivos de la aplicación.
config	Contiene archivos de configuración interna del sistema
src	Contiene todo el código base, lógica y archivos de la aplicación.
controllers	Contiene a los controladores encargados de atender los principales request HTTP, realizar acciones y responder.
services	Contiene la lógica para realizar las diferentes funcionalidades del sistema
models	Contiene los modelos de los datos que utilizaremos y serán persistidos en la base de datos.
repositories	Aquí se maneja todo lo relacionado con la base de datos, acceso a la misma, creación de tablas, restricciones.
routes	Contiene las diferentes rutas a los diferentes endpoints del sistema
sender	Contiene el mail sender, encargado del envío de los mails del sistema, ya sean las invitaciones, llegadas de un nuevo bug.
monitoring	Contiene el monitor, el cual está encargado de controlar

	cuando se lo solicitan el estado de las diferentes partes del sistema, respondiendo cuales están bien y cuáles no.
logger	Contiene el código para la generación de los log del sistema según los formatos deseados.

3.1.2. Vista de Usos

3.1.2.1. Representación Primaria



3.1.2.2. Catálogo de Elementos

Debido a que los elementos son todos los mismos que los de la [vista anterior](#) solo se detallarán los elementos nuevos.

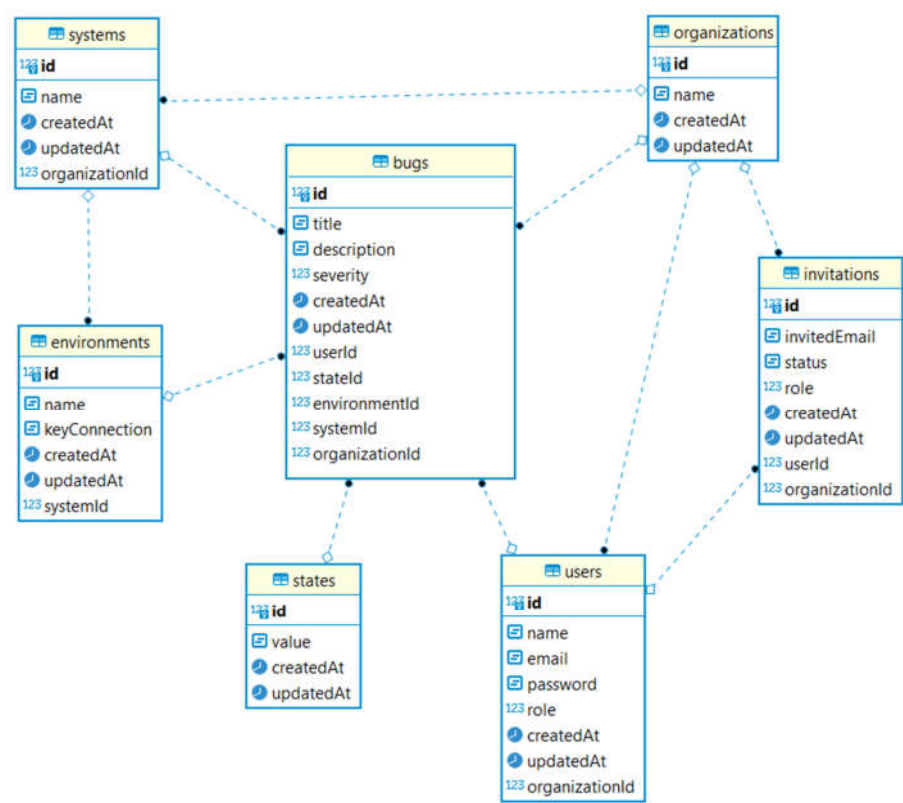
Elemento	Responsabilidad
----------	-----------------

postgres	Este es el tipo de base de datos utilizada para almacenar toda la información pertinente a los modelos del sistema.
----------	---

### 3.1.3. Modelo de Datos

A continuación pasaremos a mostrar y describir el Modelo entidad relación de la base de datos del sistema.

#### 3.1.3.1. Representación Primaria



#### 3.1.3.2. Catálogo de Elementos

Elemento	Responsabilidad
systems	Contiene todos los sistemas creados en la aplicación. Posee una foreign key con organización para saber a cual corresponde
environments	Corresponde a la tabla de ambientes para nuestro sistema, tiene una foreign key con los sistemas para saber a cual pertenece.
bugs	Contiene a todos los bugs creados en el sistema, las foreign key que tiene son con usuario, estado, environment, sistema y organización, para de esta forma saber a qué organización, sistema y ambiente pertenece, que usuario lo tiene asignado y en qué estado se encuentra.
organizations	Contiene todas las organizaciones del sistema
states	Contiene los estados en los cuales los bugs se podrán encontrar.
users	Contiene a los usuarios del sistema y posee una foreign key con la organización a la cual pertenece.
invitations	Contiene todas las invitaciones enviadas a los usuarios así como el estado actual de la misma (pendiente, cancelada, aceptada). Tiene dos foreign key, una con el usuario al cual fue enviada y otra con la organización para la cual pertenece esta invitación.

### 3.2. Vista de componentes y conectores

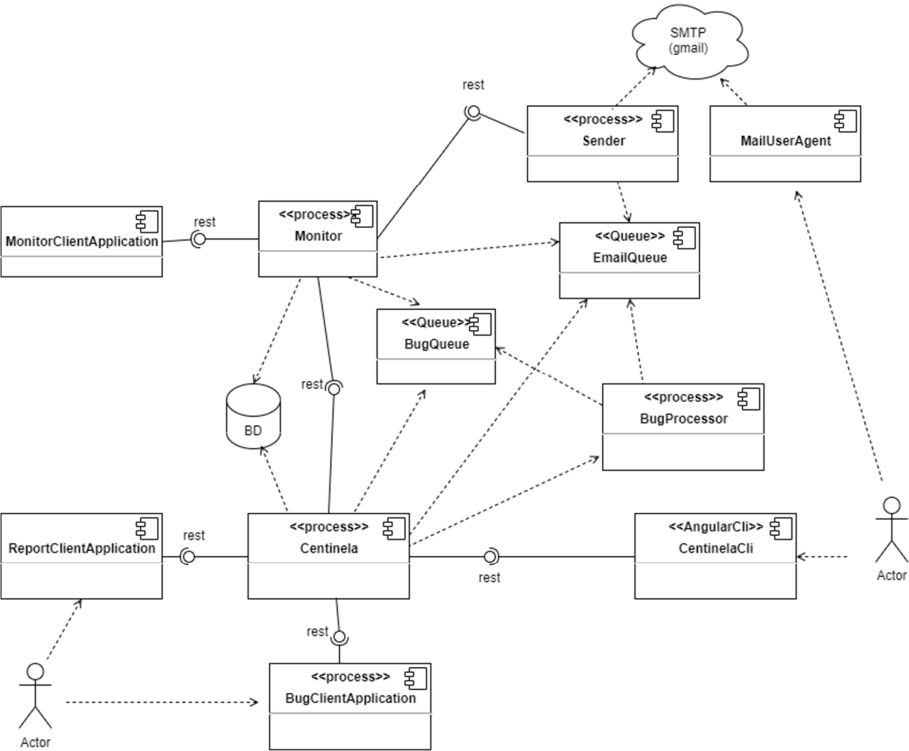
Esta vista tiene como objetivo mostrar en tiempo de ejecución cómo se relacionan los distintos componentes del sistema a través de sus conectores, básicamente mostrando qué componentes exponen interfaces (puertos) para que puedan ser consumidas por otros componentes.

#### 3.2.1 Vista de componentes y conectores del sistema

Esta vista contiene todos los componentes del sistema y los conectores que permiten su comunicación. A su vez todos los componentes externos al sistema que interactúan con el nuestro de alguna forma:

1. Aplicaciones que reportan bugs
2. Aplicaciones que consumen los reportes provistos (por ejemplo un cliente REST Postman)

3.2.1.1 Representación primaria



### 3.2.1.2 Catálogo de elementos

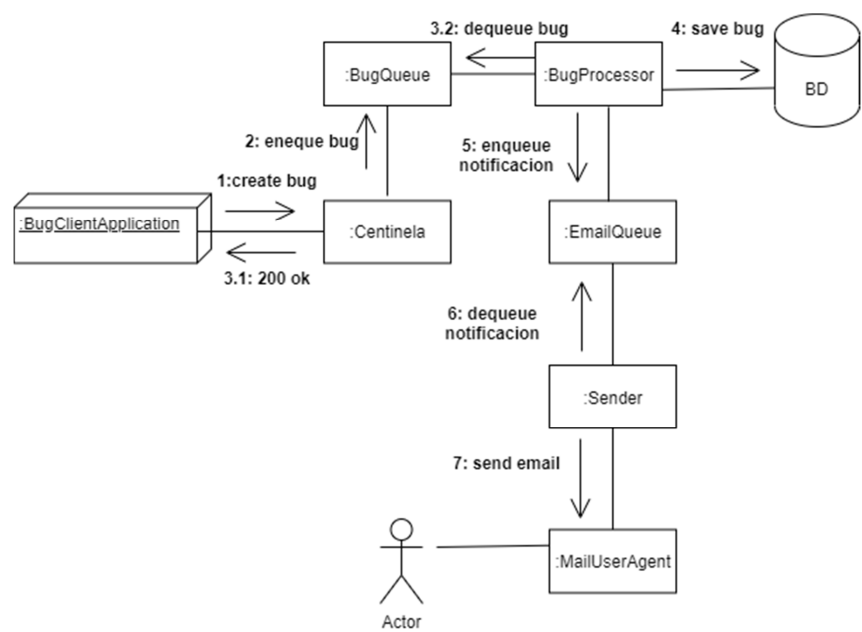
Componente/ conector	Responsabilidad
Centinela	Se encarga de procesar los errores que recibe de las aplicaciones, valida y guarda en la cola de mensajería para que se encargue el BugProcessor. A su vez la gestión de usuarios y sistemas. También ofrece una interfaz para que el monitor pueda chequear la vida del sistema. Por último envía notificaciones a una cola de mensajería de correos cuando se invita a un nuevo usuario.
Sender	Se encarga de enviar correo, por ejemplo cuando van arribando los errores, a los usuarios de la organización donde se originaron, o cuando se envía una invitación a un nuevo usuario.
EmailQueue	Es una cola de mensajería que se utiliza para que el sender pueda ir procesando las solicitudes y no se sobrecargue el sistema de correo.
MailUserAgent	La casilla de correo de un usuario
BugClientApplication	Una aplicación de una organización, que reporta errores de un ambiente a nuestro sistema
BugQueue	Una cola de mensajería a efectos de poder procesar los errores reportados sin sobrecargar el sistema y perder datos o provocar fallas
ReportClientApplication	Cualquier aplicación que posea un cliente rest, y que le interese consumir reportes de los errores que ofrece nuestra aplicación
Monitor	Proveer un servicio rest que se encargue de indicar el estado general de nuestro sistema. Posee un cliente a las colas de correo y la base de datos, para acceder a su estado, y se comunica vía rest con el Sender y el Centinela para conocer sus estados. Por último ofrece un endpoint rest para que otras aplicaciones puedan consumir los datos de estado que puede producir
BugProcessor	Se encarga de procesar los bugs encolados, los guarda en la base de datos, y encola solicitudes de correo para que el Sender comunique a usuarios la creación de un nuevo bug.



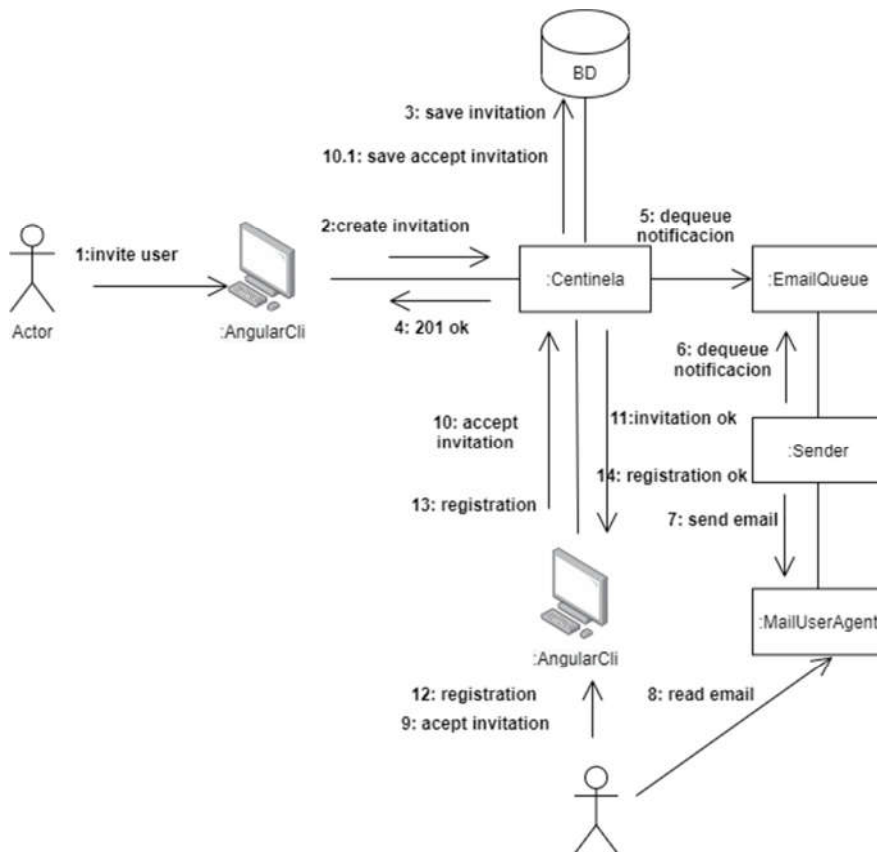
3.2.1.3 Comportamiento

Pasaremos ahora a mostrar cómo interactúan los distintos componentes, para los casos de uso más importantes del sistema.

a. Creación de un bug



b. Creación de una invitación y registro

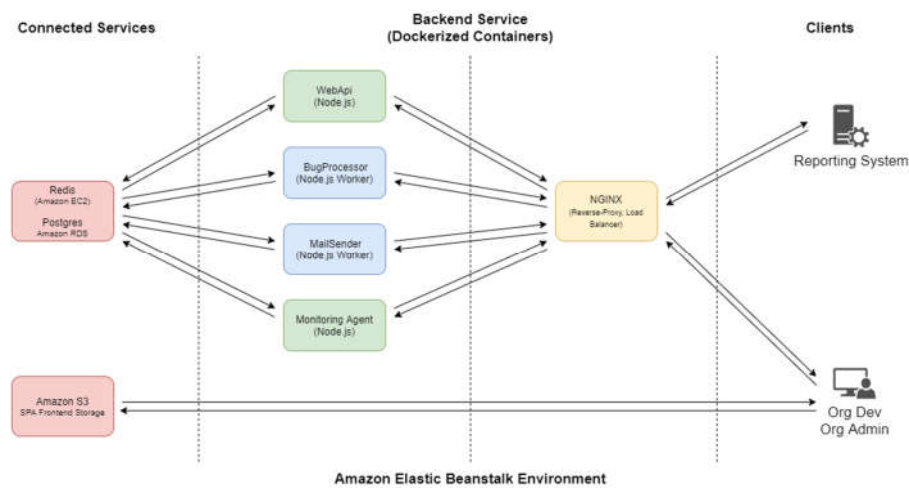


3.3. Vista de Asignación

En este tipo de vista, se incorporan las decisiones sobre cómo se relaciona el sistema con las estructuras que no son software en su ambiente (CPU, file systems, redes, equipos de desarrollo, etc). Muestran la relación entre los elementos del software y los elementos del entorno externo en el que se crea y ejecuta el software.

3.3.1. Vista de Despliegue

3.3.1.1. Representación Primaria



3.3.1.2. Catálogo de Elementos

Elemento	Responsabilidad
Redis (Amazon EC2) Postgres Amazon RDS	Aquí tenemos dos bases de datos, por un lado Redis que estará alojada en Amazon EC2 debido a las limitaciones de la cuenta educate y por otro lado a tenemos un Postgres alojado en Amazon RDS.
Amazon S3 SPA Frontend Storage	Utilizando Amazon S3 alojaremos el frontend de nuestra aplicación.
WebApi	El contenedor que aloja nuestra Api que está desarrollada en NodeJs.
BugProcessor	Contenedor que aloja el worker BugProcessor que se

	encargará de procesar los bug de la cola de mensajería.
MailSender	Contenedor que aloja el worker MailSender encargado de enviar los mails a los diferentes interesados.
Monitoring Agent	Contenedor que aloja nuestro agente de monitoreo el cual se encargará de monitorear los diferentes contenedores.
NGINX	Contenedor que aloja nuestro proxy reverso. Este se encargará de direccionar los distintos request REST al endpoint correspondiente.
Reporting System	Este será todo sistema externo de los clientes que solicitaron los reportes de bugs críticos
Org Dev/ Org Admin	Estos son los equipos o dispositivos de las organizaciones de los clientes, ya sean con el rol de developer o admin que van a interactuar con nuestro frontend directamente o mediante REST.

## 4. Justificaciones de diseño

El proyecto tecnológico a desarrollar debe usar tecnologías web y se deberá ajustar a las características de una aplicación cloud native. El equipo podrá optar por usar mayoritariamente la tecnología de "Plataforma como servicio" PaaS.

En nuestro caso, para la etapa de desarrollo y para el despliegue, los componentes principales de nuestro sistema fueron construidos para ser ejecutados como contenedores independientes. Esta decisión fue tomada con el fin de favorecer la portabilidad y escalabilidad de nuestro sistema, a lo que destacamos:

Los contenedores tienen la capacidad de ser agnósticos a la infraestructura en la cual están desplegados, permitiendo a nosotros hacer el despliegue de los mismos en cualquier plataforma que ofrezca el servicio de manejo de contenedores. Ya sea en nuestro pc de desarrollo como luego en producción.

A su vez, estos nuestros servicios contenerizados no manejan estado ( Stateless ) lo que habilita a que nuestro sistema tenga la capacidad de escalar de forma horizontal ( en número de instancias de estos contenedores ) como también en forma vertical ( mejorando la capacidad de la infraestructura que los aloja )

El uso de contenedores también favoreció a que el equipo de desarrollo pueda enfocarse por completo al diseño y codificación. Evitando así, que cada uno de nosotros tenga que realizar complicadas configuraciones en sus equipos para soportar la construcción del sistema.

Para el alojamiento de contenido estático, en nuestro caso el frontend como una single page application (SPA) en Angular. Utilizamos la plataforma como servicio de AWS llamada Amazon S3.

Si bien la tecnología de programación que utilizamos ofrece librerías para la publicación de contenido estático. El equipo tomó la decisión de alojar la SPA en una PaaS como lo es Amazon S3.

S3 nos da la ventaja de eliminar el consumo de recursos que pudiera causar la descarga de este contenido en el caso que fuera alojado en conjunto con nuestra capa de negocio. Permite además un desacoplamiento completo de lo que puedan ser las actualizaciones o cambios que pueda tener el frontend del backend.

Al mismo tiempo, Amazon S3 está preparado para soportar la transferencia de grandes volúmenes de datos sin la necesidad de tener que definir criterios de escalado o capacidad ofreciendo una tasa de disponibilidad superior 99.9% de uptime

La practicidad de su uso fue otra de las razones por la cual tomamos la decisión de utilizar S3. Todas las actualizaciones del frontend se pudieron realizar simplemente con una simple acción de Drag n' Drop de nuestro código compilado hacia el storage de S3.

Nuestro servicio depende también de servicios conectados o backing services como lo son nuestra base de datos relacional y el servicio de caché. Si bien ambos servicios son esenciales para el funcionamiento de nuestro sistema desarrollado, no son parte de la implementación del mismo, sino que solo son servicios que éste consume. Ambos backing services fueron pensados para ser consumidos desde servicios PaaS como los que ofrece Amazon. RDS para alojar nuestra base datos relacional y ElastiCache para el on-memory storage.

Aquí queremos notar que debido a las restricciones de nuestra cuenta Educativa en AWS, Elastic Caché con REDIS no era una opción permitida a utilizar, con lo que utilizamos un servicio IaaS como lo es Amazon EC2 para desplegar allí una máquina virtual con Ubuntu donde instalamos dentro un

servicio de caché REDIS y poder ofrecerlo como PaaS a nuestra aplicación.

La decisión principal para determinar la arquitectura de esta manera fué que una base de datos relacional era adecuada para el modelo de datos que se necesitaba manejar. Los requerimientos de performance se podían alcanzar y además nos permite mantener la integridad relacional de forma continua de los datos allí almacenados. Además utilizarlo como servicio conectado nos da la flexibilidad de que mañana pueda ser otro proveedor y/o que se pueda escalar verticalmente el motor de BD. A su vez, no hay necesidad de gestionar los backups y/o réplicas de los datos ya que están ofrecidos dentro del paquete PaaS.

La decisión sobre el uso de un caché dentro de la arquitectura de nuestro sistema se basó mayormente en que queríamos incrementar la performance usando tácticas para permitir un gran throughput en la publicación de bugs.

Debido a que estos bugs finalmente se persisten en una base de datos relacional. Los tiempos de acceso de esta podrían llegar a limitar la capacidad de respuesta de nuestro servicio a la creación de bugs. Se implementó entonces la táctica de "Limitar respuesta a eventos" para la cual el sistema al recibir la publicación de un nuevo bug encola un trabajo en un caché y responde al cliente. Luego este trabajo es realizado por un proceso worker que desencola el trabajo y lo almacena en la BD. Cualquier retraso en las operaciones de I/O a la base de datos son "amortiguadas" por nuestro caché y habilitan a que la aplicación pueda aceptar una mayor cantidad de req/s.

También se utilizó el servicio de colas de mensajería para el envío de correos electrónicos. Una vez que un bug es persistido en BD se dispara la notificación mediante correos electrónicos hacia los miembros de la organización. Este trabajo, que puede llegar a ser lento y además fallar debido a agentes externos, es realizado por un proceso secundario (worker).

Centinela encola trabajos de envío de correo electrónico y el worker se encarga de desencolar estos trabajos y enviar los correos. A su vez reintenta si el envío falla y almacena los envíos fallidos en una dead letter queue para su futuro análisis.

Se aplicó un algoritmo de reintento exponencial. El primer reintento ocurre a la hora de la primera falla y los siguientes 2 reintentos divididos exponencialmente en el tiempo.

A continuación pasaremos a detallar las diferentes tácticas que utilizamos para satisfacer los distintos RNFs pedidos en la letra.

#### 4.1. Performance y testing

Para lograr satisfacer que las respuestas a todas las operaciones públicas del sistema se encuentren por debajo de los 350 ms para una carga de hasta 1200 req/m, decidimos utilizar redis, la cual es una base de datos en memoria que nos permite cachear información que requiere bajos tiempos de latencia.

El uso específico que le damos a redis es para armar colas de mensajería. Estas colas son dos, una para recibir los request de bugs enviados por los sistemas externos y la otra para encolar los mails que va a ir enviando el mail sender.

En la siguiente imagen de de un test de carga generado con Apache JMeter, en el mismo fue generado con 102 clientes simultáneos más uno consultando el estado con el monitor del sistema que incorporamos.

Label	# Samples	Average	Throughput
Accept Invite	204	79	4.2/sec
Login as Admin	281	72	1.6/sec
Create Organization wit...	102	80	3.4/sec
Login as Developer	205	72	1.4/sec
Get Bugs Completed	426	6	2.7/sec
Post new Bug	715	5	3.9/sec
Get Critical Bugs	412	8	2.7/sec
Create Environment	102	18	2.7/sec
Get Bugs Pending	472	6	2.7/sec
Get System Status	22	25	6.3/min
Login	102	67	3.2/sec
Get Bug Open	204	5	1.6/sec
Invite new Dev User	204	17	4.3/sec
Update Bug User	236	15	1.4/sec
Update Bug State	204	15	1.6/sec
Get Users in Organization	256	4	1.5/sec
Get Bug Closed	203	5	1.6/sec
Create System	102	10	3.1/sec
Get Users	102	4	2.4/sec
<b>TOTAL</b>	<b>4554</b>	<b>21</b>	<b>20.8/sec</b>

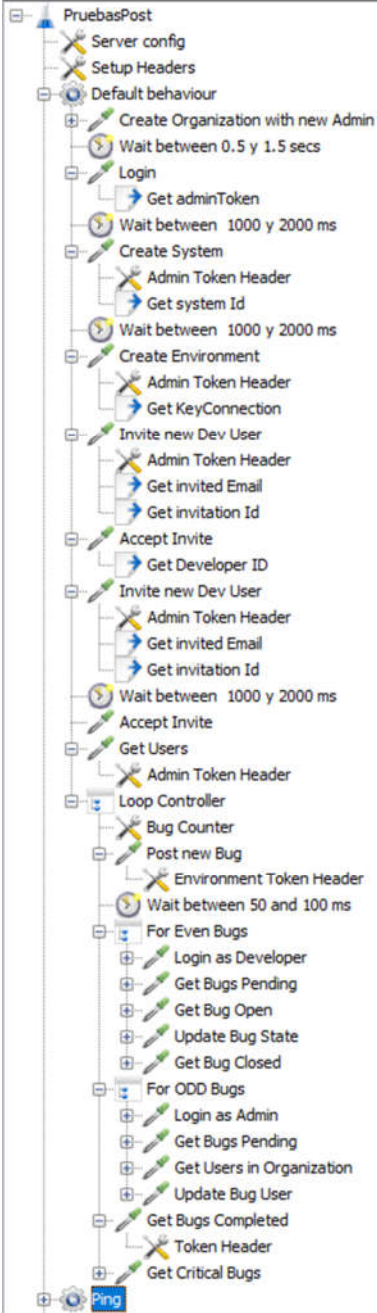
Como podemos apreciar el "throughput" (el cual se calcula como **número de request/unidad de tiempo**, siendo el tiempo calculado desde el inicio del primer sample hasta el final del último) nos da 20.8 req/sec lo cual se traduce a 20.8 requests por segundo, dándonos por lo tanto 1248 req/min para un promedio de 21 ms, superando ampliamente los 350 ms deseados en este requerimiento.

Para el testing que se realizó se desarrolló un flujo completo de la actividad que puede tener el sistema de forma diaria. A continuación se muestra una captura de la interfaz de JMeter y una explicación de las tareas que se realizaron.

Las ejecución de pruebas define 2 thread groups, un thread group (*Default behaviour*) que representará el comportamiento de los usuarios y los sistemas que envían bugs a la aplicación y otro



thread group (Ping) que estará monitoreando el estado general de la aplicación.



Default behaviour primero establece la creación de una organización con un nuevo usuario Administrador. Luego se simula el tiempo que le toma a este loguearse en el sistema. Se espera nuevamente un tiempo de entre 2 o 3 segundos para que el nuevo administrador logueado pueda crear un sistema y un ambiente.

Luego que el sistema y el primer ambiente son creados. Se hace la invitación a dos nuevos desarrolladores, los cuales a continuación aceptan la invitación. Finalmente se hace una solicitud para obtener todos los usuarios de la nueva organización.

A continuación de la etapa inicial de creación de la organización, sistema y ambiente y de invitar a los desarrolladores que usarán el sistema. Se definió en JMeter un Loop que se ejecuta por 70 ciclos. Para simular la carga de 70 bugs y el comportamiento de 70 solicitudes hacia nuestro servicio.

En el caso de los ciclos **pares**, el proceso continúa con el login al sistema como un usuario desarrollador. Luego se simula que el desarrollador obtenga la lista de bugs pendientes. De estos bugs pendientes, JMeter luego obtiene los detalles de uno de ellos. A el cual cambia su estado a cerrado. Una vez cerrado el bug, se lista un reporte de los bugs en estado cerrado.

Para el caso de los ciclos **impares**, JMeter realiza un login como un usuario administrador, obtiene los bugs pendientes, luego obtiene los usuarios que existen en la organización y luego asigna un bug a un desarrollador.

Al final de este loop, para los ciclos pares e impares, Jmeter solicita una lista de los bugs completados y luego el reporte de los 5 errores críticos del sistema.

El otro thread group (*Ping*) solamente realiza la tarea de lanzar el sanity check para corroborar el estado de cada uno de los componentes del sistema, este se repite cada 1 segundo.

4.2. Confiabilidad y disponibilidad

Para cumplir con este requerimiento, se creó un monitor con el cual se interactúa mediante un endpoint HTTP como lo solicita el requerimiento. Este se encarga de verificar el estado de los diferentes componentes del sistema retornando un json con el estado de cada uno de estos. A continuación dejamos la captura de dos consultas al monitor, una con todo prendido y otra con el mail sender y Centinela apagados :

Todo OK	Centinela y sender apagados
<pre>{   "status": "OK",   "data": {     "redis": "pong",     "postgresBd": "pong",     "senderEmailWorker": "pong",     "centinela": "pong",     "emailQueueStatistics": {       "delayed": 0,       "waiting": 0,       "active": 0,       "failed": 2,       "completed": 0,       "paused": 0     },     "bugQueueStatistics": {       "delayed": 0,       "waiting": 0,       "active": 0,       "failed": 0,       "completed": 0,       "paused": 0     }   } }</pre>	<pre>{   "status": "Alerted",   "data": {     "redis": "pong",     "postgresBd": "pong",     "senderEmailWorker": "Sender do not respond pong",     "centinela": "Centinela do not respond pong",     "emailQueueStatistics": {       "delayed": 0,       "waiting": 0,       "active": 0,       "failed": 2,       "completed": 0,       "paused": 0     },     "bugQueueStatistics": {       "delayed": 0,       "waiting": 0,       "active": 0,       "failed": 0,       "completed": 0,       "paused": 0     }   } }</pre>

Como se puede apreciar se devuelve el estado de cada parte del sistema, siendo **pong** respuesta exitosa y en caso contrario un mensaje indicando que no hubo respuesta. También podemos ver el estado de las colas de mail y de bugs, indicando que tan llenas se encuentran y los estados de los mensajes que tienen dentro.

Para gestionar la disponibilidad de los sistemas implementamos un reverse proxy que permitirá a un futuro poder balancear la carga en varias instancias de nuestra aplicación. Hoy el reverse proxy es un contenedor más dentro de nuestro ambiente y se encarga de distribuir los request a los contenedores correctos.

### 4.3. Configuración y manejo de secretos

Toda información de configuración no sensible y la cual no era necesario cambiarla en tiempo de ejecución se especificó en archivos de configuración internos a la aplicación en las carpetas config.

Por otro lado, toda la información de configuración sensible se maneja a nivel de variables de entorno, las cuales fueron cargadas en AWS y leídas desde ahí, no teniendo las cargadas en el código fuente de nuestro entorno.

La lista variables de entorno se encuentran en un archivo de nombre “**.env-examples**”, de esta forma cada vez que se creaba una nueva variable de entorno, se cargaba ahí para que los demás desarrolladores estuvieran al tanto de su existencia y a su vez llevar una lista de todas las variables de entorno utilizadas en el sistema.

### 4.4. Autenticacion y autorizacion

Para el uso de autenticación y autorización se utilizó manejo de tokens. La arquitectura elegida es la autenticación mediante uso de tokens de JSON Web Tokens en las request que requieran autorización. Esta permite que la autenticación de un actor ocurra sin la necesidad de enviar contraseñas en cada solicitud y habilita el manejo sesiones y la transferencia de datos dentro de estos tokens.

Cuando un cliente se registra, su contraseña es almacenada en la base de datos de forma encriptada. Esto ofrece la ventaja de que en caso de comprometerse la seguridad de nuestra base de datos, las contraseñas de nuestros usuarios no quedan comprometidas. El proceso de login ocurre comparando las versiones encriptadas de la contraseña, y en los casos que este login sea satisfactorio. El sistema retorna un web token que el cliente utilizará para cada uno de los siguientes requests que realice. Estos web tokens tienen una vigencia de 1hr (configurable) los cuales deben ser renovados para mantener la sesión activa. Además estos tokens llevan información útil del usuario y su organización Esta información es utilizada por nuestro sistema de backend y frontend y que en caso que de ser alterada su integridad provocaría un funcionamiento incorrecto del sistema. JSON Web Tokens permite validar que esta información que viaja en el token no sea alterada, ya que en caso que lo fuera, el token dejaría de ser válido.

El manejo de la autorización en nuestro sistema se basó en el uso del control de acceso basado en roles. Elegimos esta táctica para simplificar y a su vez asegurar el correcto control de acceso de los usuarios a los recursos. Cada una de los endpoints publicados tiene definido cuales son los roles que pueden utilizarlo y es mediante el uso de los tokens antes mencionados que se puede identificar y obtener el rol de un usuario.

Cada intento de login y cada verificación de token queda registrada en el log de la aplicación. Con el fin de poder determinar quienes accedieron al servicio y además poder hacer un diagnóstico en caso de error.

Tomamos la decisión de utilizar el mismo mecanismo para asignar las claves de aplicación. Cada vez que se crea un “entorno”, el sistema genera un JSON WebToken ( esta vez sin caducidad ) que

debe ser adjuntado en las request que los sistemas externos realizan al publicar sus bugs. Este token contiene además toda la información relevante del ambiente al cual debe ser publicado el bug con el fin de que nuestro sistema pueda rápidamente identificar la organización, del sistema y el entorno al cual pertenecerá el nuevo bug reportado.

#### 4.5. Seguridad

Para favorecer la seguridad, todos nuestros endpoints fueron diseñados para responder códigos de error acordes a la situación. Esto permite que, ya sea nuestro frontend como otros clientes que utilicen nuestro SaaS sepan interpretar y actuar acorde el tipo de error.

Para el control, toda actividad del sistema es logueada dentro de la nube de amazon mediante el stream de los mismos hacia Amazon Cloudwatch el cual almacena los logs por 7 días (configurable). Al utilizar todos servicios en el mismo proveedor cloud pudimos aprovechar y realizar la comunicación con todos los servicios conectados dentro de una red interna. además que estos servicios fueron configurados con contraseñas de acceso. Y estas contraseñas son solo configurables en variables del entorno y no están presentes en el código fuente.

Para la comunicación de el frontend hacia el backend debemos aclarar que no utilizamos protocolos seguros. Si bien la aplicación y sus librerías están preparadas para soportarlo, al ser ésta desarrollada sólo con fines académicos y debido a que existe un costo de contratación de los certificados SSL para permitir la comunicación segura, optamos por no implementarlo pero sí desarrollar todo para poder soportarlo en el futuro.

#### 4.6. Código fuente

Los lenguajes seleccionados para la codificación del sistema fueron **NodeJs** para el backend y **Angular** para el frontend. A su vez nos apoyamos en un template como explicamos en la sección [2.2.Desafíos y decisiones](#) para agilizar el desarrollo del frontend.

En cuanto a los estándares de codificación, nos basamos en las convenciones de codificación de NodeJS (<https://docs.npmjs.com/misc/coding-style>) para lo cual configuramos un formateador de texto que nos ayudará con dicha tarea.

Para el manejo de los branches utilizamos Gitflow como el requerimiento lo solicita, de forma comenzamos creamos dos branches, master y develop. Para el desarrollo de cada nueva funcionalidad se sale de develop llamando a la nueva rama "feature/nombre\_funcionalidad" mientras que los fix serán "fix/nombre\_fix".

Una vez finalizado el trabajo en esa feature se hace merge a develop.

#### 4.7. Pruebas

Para satisfacer con este requerimiento, se generaron los archivos correspondientes tanto las colecciones de postman como los scripts de JMeter. Estos se encuentran ubicados dentro del proyecto, en la carpeta **tests**, dentro de esta carpeta están identificadas con nombre las carpetas correspondientes a los archivos de las pruebas.

Para las pruebas de postman contamos con dos archivos, uno que carga la colección y otro que carga dos variables de entorno, url y url\_monitoring correspondientes con las url del sistema

Centinela y del monitor. En nuestro ambiente local por ejemplo eran: url: localhost:5000 y url\_monitoring: localhost:5001. Sin embargo para el sistema corriendo en AWS ya están configuradas en este archivo.

#### 4.8. Identificación de fallas

Como ya nos referimos en la sección [4.5. Seguridad](#), los logs son guardados amazon mediante el stream de los mismos hacia Amazon Cloudwatch donde los estamos almacenando por 7 días. Para crear las diferentes entradas de los registros de logs utilizamos un componente llamado logger el cual usa la librería winston para realizar las acciones de logeo. En logger definimos diferentes niveles para las entradas del log según lo que se quisiera indicar, estos niveles son: info, debug, warning, error y fatal; con ellos pudimos manejar los diferentes tipos de mensajes que queríamos loggear y así saber gracias a los niveles que había pasado. A su vez cada entrada del log fue armada de la siguiente forma:

[Fecha y hora en formato YYYY-MM-DD HH:MM:SS] [Nivel de información] [Archivo donde se está realizando la entrada del log] >>> Mensaje de la entrada del log.

A la hora de armar el mensaje de la entrada para el log, también se pueden mandar los tiempos de inicio y fin para de esta forma llevar un seguimiento de cuánto demoran las diferentes acciones en el sistema.

De la misma forma que podemos guardar los tiempos de inicio y fin de las operaciones, también se puede loggear en el mensaje un Guid para identificar cada entrada del log de manera que sea más fácil de buscar a la hora de mirar los log.

**Comentado [1]:** @sebastianzawrzykraj@gmail.com

Dale una mirada a ver si no metí cualquiera

## 5. Proceso de deployment y dev-ops

Tal como se ha mencionado anteriormente, parte de este proyecto educativo es el de la familiarización y utilización de plataformas cloud para el despliegue del sistema desarrollado.

La base de la construcción es el uso de contenedores con el fin de:

- Mejorar la eficiencia de la utilización de los recursos disponibles
- Permitir que existan ciclos de desarrollo y deployment mucho más rápidos.
- Garantizar la portabilidad, ya que son independientes a la infraestructura en la cual se despliegan.

### Construcción:

Para la construcción de nuestro software utilizamos la tecnología de contenedores docker con las herramientas docker run y docker-compose.

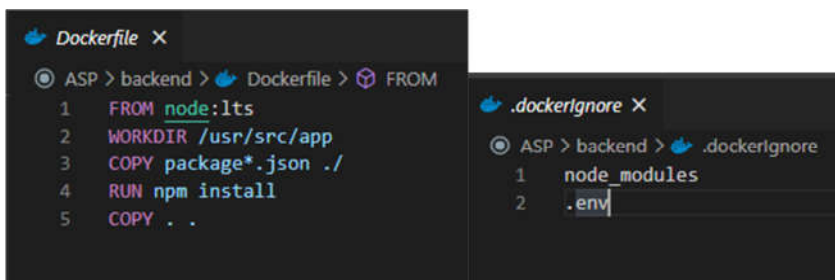
Para el proceso de desarrollo utilizamos Docker desktop para alojar los contenedores. En nuestro ambiente tenemos alojados los siguientes contenedores para ofrecer los servicios de los cuales la aplicación se conecta.

- Postgres - Contenedor que aloja una base de datos relacional
- Redis - contenedor que ofrece el Servicio de On-Memory caché
- PgAdmin - Contenedor que aloja un WebService para la gestión de base de datos Postgres.

Para la construcción del ambiente de producción, utilizamos docker-compose con el fin de orquestar la construcción de las imágenes de todos los contenedores necesarios y Dockerfile para definir las configuraciones de cada una de nuestras imágenes.

Las imágenes base para las cuales construimos los contenedores que luego se desplegarán en el entorno cloud son 2: **Backend y Nginx**.

- Backend contiene todo el codebase de nuestra aplicación. El DockerFile en esta imagen, que especifica el proceso de construcción de la imagen, es el siguiente.



```
Dockerfile X
ASP > backend > Dockerfile > FROM
1 FROM node:lts
2 WORKDIR /usr/src/app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .

.dockerignore X
ASP > backend > .dockerignore
1 node_modules
2 .env
```

El archivo Dockerfile da las instrucciones de obtener la imagen de NodeJS long term support. Luego da la instrucción copiar el archivo *package.json* para continuar con la instalación de las dependencias requeridas para nuestro código.

Al final copia todos los otros archivos del sistema, codebase y variables de entorno, en conjunto con el archivo *.dockerignore* para así evitar copiar las dependencias y archivos no necesarios en la imagen.

- La otra imagen que se construye durante el deployment es la del balanceador de carga NGINX.

En este caso, el *dockerfile* indica que se debe descargar la última versión de la imagen pública de NGINX y que luego se copie el archivo de configuración *default.conf* construido para que funcione en nuestro entorno.

```
Dockerfile X
ASP > nginx > Dockerfile > ...
1 FROM nginx
2 COPY ./default.conf /etc/nginx/conf.d/default.conf
```

```
Dockerfile default.conf X
ASP > nginx > default.conf
1 server {
2     listen 80;
3
4     location /monitoring/ {
5         proxy_pass http://centinela-monitor:5001/monitoring/;
6     }
7
8     location /sender/ {
9         proxy_pass http://centinela-sender:5002/sender/;
10    }
11
12    location / {
13        proxy_pass http://centinela-api:5000;
14    }
15 }
```

El orquestador de estas imágenes y el encargado de construir los contenedores en el ambiente cloud es el archivo *docker-compose.yml*.

Este crea varios contenedores que se utilizarán en el ambiente cloud utilizando la misma imagen construida *./backend*.



```

version: "3.7"
services:
  centinela-api:
    build: ./backend
    container_name: centinela-api
    command: ["node", "src/index.js"]
    env_file:
      - centinela.env
    ports:
      - "5000:5000"
    networks:
      - centinela-net

  centinela-monitor:
    build: ./backend
    container_name: centinela-monitor
    command: ["node", "src/monitoring.js"]
    env_file:
      - centinela.env
    ports:
      - "5001:5001"
    depends_on:
      - centinela-sender
      - centinela-api
      - centinela-bugprocessor
    networks:
      - centinela-net

  centinela-bugprocessor:
    build: ./backend
    container_name: centinela-bugprocessor
    command: ["node", "src/bugProcessor.js"]
    env_file:
      - centinela.env
    networks:
      - centinela-net

```

Finalmente el contenedor NGINX utilizando la imagen construida a partir de ./nginx

```

nginx:
  build: ./nginx
  container_name: nginx
  ports:
    - "80:80"
  depends_on:
    - centinela-monitor
  networks:
    - centinela-net

networks:
  centinela-net:
    name: centinela-net

```

### Deployment del backend.

Para el despliegue a producción utilizamos el servicio de Amazon Elastic Beanstalk.

Para ello subimos el código en un archivo zip.

La estructura del archivo comprimido es la siguiente.

```
— deploy.zip
  |— backend/
  |   |— src/
  |   |— config/
  |   |— dockerfile
  |   |— .dockerIgnore
  |   |— package.json
  |   |— package-lock.json
  |— nginx/
  |   |— dockerfile
  |   |— default.json
  |— docker-compose.yml
  |— centinela.env
```

Cada carpeta contiene lo referido a la imagen y su dockerfile para las instrucciones de construcción.

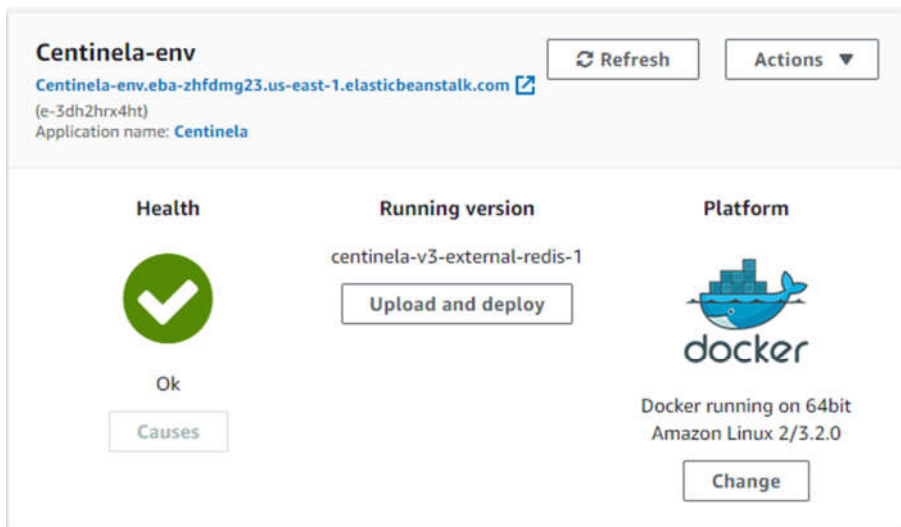
Este archivo zip se carga en Amazon Elastic Beanstalk y se configura un ambiente para su uso como host de contenedores Docker en una plataforma Amazon Linux 2 que [soporta ambientes multicontenedor](#).

The screenshot shows the 'Platform' configuration section of the Amazon Elastic Beanstalk console. It features two radio buttons: 'Managed platform' (selected) and 'Custom platform'. Below these are three dropdown menus: 'Platform' set to 'Docker', 'Platform branch' set to 'Docker running on 64bit Amazon Linux 2', and 'Platform version' set to '3.2.0 (Recommended)'.

Platform	
<input checked="" type="radio"/> Managed platform	Platforms published and maintained by AWS Elastic Beanstalk. <a href="#">Learn more</a>
<input type="radio"/> Custom platform	Platforms created and owned by you.
Platform	
Docker ▼	
Platform branch	
Docker running on 64bit Amazon Linux 2 ▼	
Platform version	
3.2.0 (Recommended) ▼	

La carga del archivo *zip*, que incluye los binarios y la configuración, crea un nuevo ambiente el cual lanza la construcción e inicio de todos los contenedores declarados en el archivo *docker-compose.yml* como fue explicado anteriormente.

Una vez que el ambiente se encuentra listo se puede apreciar su estado de Health OK



Elastic beanstalk deja a la vista la URL del ambiente por el cual nuestro servicio atenderá las request a través del contenedor NGINX.

Los contenedores requieren ciertas configuraciones que se especifican en un archivo de variables de entorno llamado *centinela.env*.

Este archivo es utilizado por ElasticBeanstalk para crear todas la variables de entorno y ofrecerlas a los contenedores que las necesiten.

Un ejemplo de estas variables es la definición de los parámetros para la conexión con el servicio RDS ofrecido por AWS para el hosting de nuestra base de datos relacional.

```
#DATABASE SETTINGS
DATABASE_USER=CentinelaUsr
DATABASE_PASSWORD=
DATABASE_NAME=CentinelaProd
DATABASE_HOST=centinelafree.cfsmohvyfeaq.us-east-1.rds.amazonaws.com
DATABASE_PORT=5432
```

Y para la conexión con el servicio de Elastic Caché - Redis

```
#Redis
REDIS_HOST='172.31.81.255'
REDIS_PORT=6379
REDIS_PASSWORD='[REDACTED]'
```

Sin embargo, tal como se explicó más arriba, no nos fue posible utilizar el servicio de ElastiCache debido a una restricción de la cuenta educativa.

Por tal motivo creamos una instancia de EC2 de una máquina virtual en Ubuntu 18 LTS en la cual instalamos el servicio de Redis Server 5.0.6 y lo configuramos para nuestra necesidad. Esta máquina virtual actúa como un backing service cloud para dar soporte a nuestro sistema.

Instance summary for i-05d723f17161c9471 (Redis-Ubuntu) Info

Connect Instance state

Instance ID i-05d723f17161c9471 (Redis-Ubuntu)	Public IPv4 address 34.226.122.127   open address	Private IPv4 addresses 172.31.81.255
Instance state Running	Public IPv4 DNS ec2-34-226-122-127.compute-1.amazonaws.com   open address	Private IPv4 DNS ip-172-31-81-255.ec2.internal
Instance type t2.micro	Elastic IP addresses -	VPC ID vpc-fa44bb87
IAM Role -	Subnet ID subnet-164ae337	

```
ubuntu@ip-172-31-81-255: ~
ubuntu@ip-172-31-81-255:~$ service redis status
● redis-server.service - Advanced key-value store
   Loaded: loaded (/lib/systemd/system/redis-server.service; enabled; vendor preset: enable
   Active: active (running) since Wed 2020-10-21 21:22:41 UTC; 21h ago
     Docs: http://redis.io/documentation,
           man:redis-server(1)
   Process: 900 ExecStart=/usr/bin/redis-server /etc/redis/redis.conf (code=exited, status=
   Main PID: 951 (redis-server)
      Tasks: 4 (limit: 1140)
   CGroup: /system.slice/redis-server.service
           └─951 /usr/bin/redis-server 0.0.0.0:6379

Oct 21 21:22:41 ip-172-31-81-255 systemd[1]: Starting Advanced key-value store...
Oct 21 21:22:41 ip-172-31-81-255 systemd[1]: redis-server.service: Can't open PID file /va
Oct 21 21:22:41 ip-172-31-81-255 systemd[1]: Started Advanced key-value store.
lines 1-14/14 (END)
```

### Monitoreo y Actualizaciones.

Para monitorear el estado del sistema, Elastic Beanstalk hace PINGs constantes hacia la URL del sistema. En este caso, si el contenedor NGINX contesta, el estado de salud del sistema será considerado como saludable.

```
▼ 2020-10-22T16:30:03.723-03:00 nginx | 172.31.42.2 - - [22/Oct/2020...
nginx | 172.31.42.2 - -
[22/Oct/2020:19:29:57 +0000] "GET / HTTP/1.1" 200 20 "-" "ELB-
HealthChecker/2.0" "-"
```

Copy

Para tener un mejor control de todas las partes del sistema, adicionado al monitoreo de Elastic Beanstalk, utilizamos dos tácticas que ya fueron explicadas arriba.

Una es la táctica de por la cual ofrecemos un **Sanity Check** que comprueba el estatus de cada componente del sistema.

Este sanity check se invoca utilizando la táctica de PING hacia el endpoint /monitoring/v1/ping. Previo a dar una respuesta, el sanity check verifica que todos los componentes del sistema estén activos y responde de la siguiente manera:

```
{
  "status": "OK",
  "data": {
    "redis": "pong",
    "postgresBd": "pong",
    "senderEmailWorker": "pong",
    "centinela": "pong",
    "emailQueueStatistics": {
      "delayed": 0,
      "waiting": 0,
      "active": 0,
      "failed": 5,
      "completed": 0,
      "paused": 0
    },
    "bugQueueStatistics": {
      "delayed": 0,
      "waiting": 0,
      "active": 0,
      "failed": 0,
      "completed": 0,
      "paused": 0
    }
  }
}
```

### Gestion de Logs

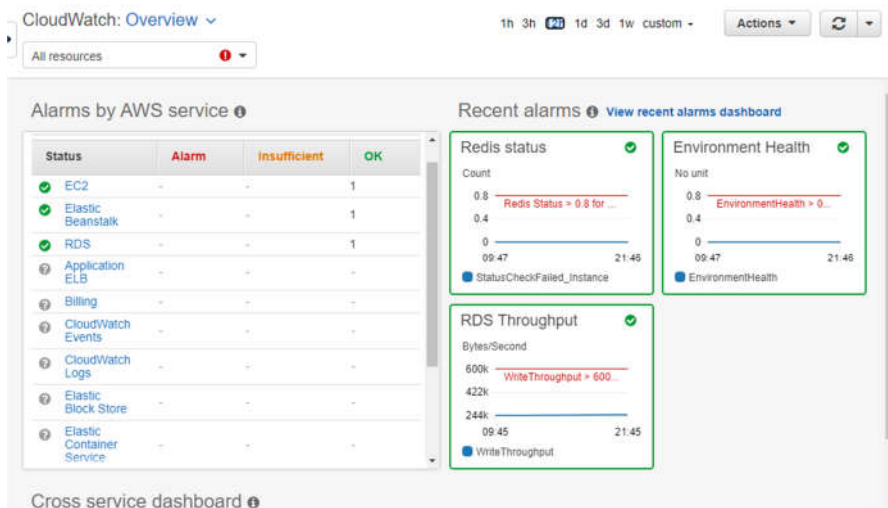
Para la gestión de Logs, configuramos nuestro ambiente para hacer uso del servicio CloudWatch de Amazon, en el cual se envían todos los logs generados por todos los procesos del sistema hacia este repositorio. Allí podemos observar en tiempo real todo lo que sucede en el ambiente desplegado en elastic beanstalk.

Aquí un ejemplo:

▶	2020-10-21T19:45:08.984-03:00	centinela-api		[2020-10-21 22:45:08][info][.../controllers/invitation-con...	
▶	2020-10-21T19:45:09.734-03:00	nginx		172.31.42.2 - - [21/Oct/2020:22:45:08 +0000] "POST /api/v1/invite...	
▶	2020-10-21T20:15:33.772-03:00	nginx		172.31.42.2 - - [21/Oct/2020:23:15:33 +0000] "OPTIONS /api/v1/inv...	
▶	2020-10-21T20:15:33.772-03:00	centinela-api		[2020-10-21 23:15:33][debug][.../controllers/invitation-co...	
▶	2020-10-21T20:15:34.023-03:00	nginx		172.31.42.2 - - [21/Oct/2020:23:15:33 +0000] "GET /api/v1/invitat...	
▶	2020-10-21T20:15:50.290-03:00	nginx		172.31.42.2 - - [21/Oct/2020:23:15:50 +0000] "OPTIONS /api/v1/inv...	
▶	2020-10-21T20:15:50.290-03:00	centinela-api		[2020-10-21 23:15:50][debug][.../services/invitation-servi...	
▶	2020-10-21T20:15:50.540-03:00	centinela-api		[2020-10-21 23:15:50][debug][.../services/invitation-servi...	
▶	2020-10-21T20:15:50.540-03:00	centinela-api		[2020-10-21 23:15:50][debug][.../services/invitation-servi...	
▼	2020-10-21T20:15:50.540-03:00	centinela-api		[2020-10-21 23:15:50][debug][.../services/invitation-servi...	
		centinela-api		[2020-10-21 23:15:50][debug][.../services/invitation-service.js] >>> This	Copy
		invitation		{	
				"id": 114,	
				"invitedEmail": "msettino@edinet.com.uy",	
				"status": "Pending",	
				"role": 2,	
				"userId": 1,	
				"organizationId": 1	
				}	
				is being set as: Confirmed	
▶	2020-10-21T20:15:53.043-03:00	nginx		172.31.42.2 - - [21/Oct/2020:23:15:50 +0000] "POST /api/v1/invite...	
▶	2020-10-21T20:18:02.918-03:00	nginx		172.31.42.2 - - [21/Oct/2020:23:18:02 +0000] "OPTIONS /api/v1/inv...	

Cloudwatch además permite configurar alertas para múltiples métricas en las cuales podemos ser notificados de cambios en el estado del sistema.

En nuestro caso tenemos definidas 3 alertas para el monitoreo y control de la instancia de Redis, de la instancia de Elastic Beanstalk y el estado general de RDS.



En cuanto a la modalidad de actualización o deployment policies de nuevas versiones de nuestro sistema. En esta primera entrega el equipo decidió utilizar la política de roll out “*All at once*”. Sabemos que esto puede generar interrupciones del servicio al momento de hacer los roll out, pero se debe tener especial consideración, ya que dada la demanda que posee este servicio, al ser simplemente educativo, solo una instancia se encuentra en ejecución y en este caso, no importaría cuál sea la política de roll out. En todas ellas existiría una interrupción del servicio. Idealmente en casos de negocio reales, si se eligiera el mecanismo de roll out “*All at once*” se debería incluir como parte de acuerdo de servicio con nuestros clientes, ventanas de mantenimiento programado, con el fin de poder hacer las actualizaciones.

## 6. Resultados y conclusiones

Pasamos ahora a comunicar los resultados que hemos obtenido, y como hemos trabajado para llegar a los mismos.

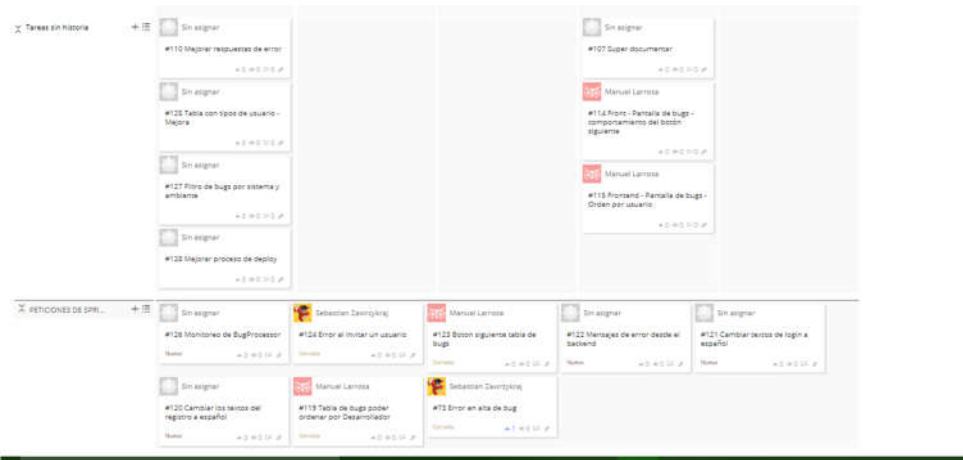
### 6.1 Proceso de trabajo y requerimientos funcionales

En cuanto al proceso de trabajo, el equipo decidió tomar a Scrum como metodología de trabajo. Se dividió el trabajo en 4 Sprints, en los cuales progresivamente se fue mejorando la dinámica y los resultados, como se puede ver en el apartado [sprint realizados](#).

El resultado estado final del trabajo, con sus tareas pendientes, se encuentra en el último sprint:

<https://tree.taiga.io/project/zebasetas-bug-tracking/taskboard/sprint-3-6755?kanban-status=1922560>

De acuerdo a los resultados, se entiende que se llegó prácticamente al 99.9% de los requerimientos funcionales solicitados, quedando un remanente de algunas mejoras menores que quedarán pendientes para futuros sprints:



Dentro de las mejoras a realizar, los apartados que nos quedaron pendientes más relevantes:

1. Mejoras en los mensajes de error:  
El equipo dejó como pendiente mejorar los distintos mensajes que entrega la aplicación en los casos de posibles errores que puedan darse en la aplicación. Por ejemplo, para validaciones



no cumplidas o requests malformados, los mensajes no son 100% informativos a efectos de que sea más fácil para el frontend tomar algunas decisiones.

2. Monitoreo del worker BugProcessor: se realiza un monitoreo de todos los componentes, colas mensajería y base de datos, quedando pendiente hacerle un ping/echo a este componente independiente.
3. Mejorar el proceso de deploy en AWS buscando ir hacia la integración continua: Entendemos que las limitaciones que nos ofrece la cuenta educativa y la restricción de mantener todo el código de forma monolítica en el mismo codebase colaboró a que el despliegue sea también en forma combinada. Para futuras versiones intentaremos aplicar tareas de codedeploy ofrecidos por AWS para desplegar los servicios de forma totalmente independiente.
4. Si bien la codificación del sistema fue totalmente en inglés, al ser un servicio pensado para utilizar en uruguay, el frontend debería mostrar su interfaz de login y registro en idioma español. El resto del frontend está desarrollado en español. Idealmente se podría construir agnóstico al idioma o al menos en los dos idiomas mas comunes de nuestros clientes.

Algunas actividades del proceso que nos fueron de gran ayuda:

1. Se trabajó de utilizando Gitflow, se intentó tener una nomenclatura de branches que se puede ver en nuestra wiki (que aún resta pulir):

<https://tree.taiga.io/project/zebasetas-bug-tracking/wiki/2-propuesta-de-gestion-de-repositorios>

2. Se realizaron inspecciones de código cruzadas.
3. Se estableció que una tarea no estaba terminada hasta tener el test de JMeter, y a la vez que un compañero no le diera el visto bueno.
4. Se realizaron tests de integración luego de cada cierre de Sprint
5. Se intentó acordar estas pautas en el definition of done <https://tree.taiga.io/project/zebasetas-bug-tracking/wiki/1-definition-of-done>

## 6.2 Requerimientos no funcionales

En cuanto a los requerimientos no funcionales se detalla su cumplimiento

1. Se cumplió con los requerimientos de performance (**RNF1**) documentados en el apartado [performance](#).
2. Se cumplió con el requerimiento de confiabilidad y disponibilidad (**RNF2**), como se indica en el apartado de [confiabilidad](#).
3. Se cumple con el configuración y manejo de secretos (**RNF3**), como se indica en el apartado [configuración y manejo de secretos](#).
4. Se cumple con el requerimiento de autenticación (**RNF4**), como se indica [autenticación y autorización](#).
5. Se cumple con el apartado de seguridad (**RNF5**), indicado en [seguridad](#)
6. Se cumple con el apartado de código fuente (**RNF6**). Se deja un archivo Readme en el repositorio, indicando cómo configurar un ambiente de desarrollo. También se puede acceder

a esa información en nuestra wiki <https://tree.taiga.io/project/zebasetas-bug-tracking/wiki/3-configuracion-y-ambientes>

También se puede ver el apartado rest [api rest](#), donde puede ver cómo generamos nuestra api (o visitar directamente el cliente swagger configurado (<http://centinela-env.eba-zhfdmg23.us-east-1.elasticbeanstalk.com/api/v1/docs/>))

7. Se cumple con el requerimiento de identificación de fallas (**RNF8**), como se indica en [identificación de fallas](#)

## 7. Anexos

### 7.1. Requerimientos y restricciones

#### 7.1.1. Requerimientos Funcionales

ID Requerimiento	Descripción	Actor
RF1 - Registro de usuario administrador	Los usuarios administradores podrán registrarse vía web como administrador de su organización. Para ello, deberá ingresar nombre, email, nombre de organización (único en el sistema) y contraseña. Al registrarse de esa manera, el administrador habrá creado una organización nueva. Luego, podrá invitar otros usuarios administrador y usuarios desarrolladores ( RF2 ).	<ul style="list-style-type: none"><li>• Usuario Administrador</li></ul>
RF2 - Registro de usuarios mediante invitación	Un usuario administrador deberá poder invitar a otros usuarios administradores y desarrolladores a la plataforma, enviándole un correo electrónico que contenga un link al registro anterior ( RF1 ), quitando el campo de “nombre de organización” y con el agregado de un mensaje que indique a qué organización se estaría uniendo.	<ul style="list-style-type: none"><li>• Usuario Administrador</li><li>• Usuario Desarrollador</li></ul>
RF3 - Autenticación de usuario	Dado un usuario no autenticado cuando está presente en la página principal del sistema y hace click en “Ingresar” entonces se despliega un formulario de ingreso que solicita su correo electrónico y una contraseña de acceso. Una vez autenticado el usuario el sistema debe redireccionarlo a la pantalla de listado de errores.	<ul style="list-style-type: none"><li>• Usuario Administrador</li><li>• Usuario Desarrollador</li></ul>
RF4 - Gestión de clave de aplicación	El sistema debe permitir a los usuarios administradores crear claves de acceso de aplicación. Estas claves únicas de acceso son las que el cliente debe utilizar para invocar los servicios REST provistos por el sistema. Ver RNF4. De las claves de acceso simplemente se solicita un nombre, o “entorno” ( staging, production ) , para identificarlas.	<ul style="list-style-type: none"><li>• Usuario Administrador</li></ul>
RF5 - Gestión de errores	El sistema debe permitir a los usuarios administradores editar errores mediante un formulario en la web. De los mismos se puede editar el título, descripción (opcional), severidad (opcional, número entre 1 y 4) y desarrollador asignado (opcional). Los errores son creados mediante el	<ul style="list-style-type: none"><li>• Usuario Administrador</li></ul>

	endpoint REST mencionado en RF9.	
RF6 - Listado de errores	El sistema debe permitir a todos los usuarios ver un listado web de los errores que existen para su organización. La vista por defecto deben ser los errores no resueltos , ordenados por más crítico primero (es decir, severidad 1 antes que severidad 2, etc.). También se debe poder tener la opción de mostrar errores ya resueltos. En el listado se tiene que poder ver el nombre, severidad, estado (resuelto o no resuelto) y desarrollador asignado.	<ul style="list-style-type: none"> <li>• Usuario Administrador</li> <li>• Usuario Desarrollador</li> </ul>
RF7 - Detalles de error	El sistema debe permitir a todos los usuarios clicar en un error del listado ( RF6 ), y esto debe dirigirlos a una vista en la que se muestra la información del error (título, descripción, severidad, estado, desarrollador asignado), y el botón que se especifica en RF8.	<ul style="list-style-type: none"> <li>• Usuario Administrador</li> <li>• Usuario Desarrollador</li> </ul>
RF8 - Resolución de error	El sistema debe permitir a todos los usuarios clicar un botón en la vista de RF7 que marque el error como resuelto.	<ul style="list-style-type: none"> <li>• Usuario Administrador</li> <li>• Usuario Desarrollador</li> </ul>
RF9 - Creación de error (REST)	Se deberá disponibilizar un endpoint REST que permita crear errores. Este endpoint será utilizado en los sistemas de los clientes de nuestro sistema, para que cuando ocurre una excepción en su código, se reporte automáticamente en nuestro producto. Los atributos que se pueden enviar para crear el error, son los mismos que se mencionan en RF5 , excepto “desarrollador asignado”.  <b>Importante:</b> Cuando un error es creado mediante este endpoint, se debe enviar un email a todos los miembros de la organización con información básica del mismo, para que estén al tanto de las fallas que ocurren en sus sistemas.	<ul style="list-style-type: none"> <li>• Sistema Externo</li> </ul>
RF10 - Errores críticos (REST)	Se deberá disponibilizar un endpoint REST que permita obtener los 5 errores no resueltos con mayor severidad para la organización. Este endpoint será integrado y utilizado por sistemas de los clientes, como dashboards de monitoreo y en canales de comunicación, como por ejemplo Slack. Este endpoint debe responder en hasta 200 ms para cargas de hasta 1200 req/m ya que es crítico obtener con rapidez los errores más graves.	<ul style="list-style-type: none"> <li>• Sistema Externo</li> </ul>

RF11 - Reporte de estadísticas de errores	El sistema debe permitir a los usuarios administradores obtener un reporte web que muestre cuántos errores ocurrieron en un rango de fechas dado, y cuántos de ellos fueron resueltos. También debe mostrar cuántos errores hubo por cada tipo de severidad.	<ul style="list-style-type: none"> <li>• Usuario Administrador</li> </ul>
--	--	---

### 7.1.2. Requerimientos No Funcionales

ID Requerimiento	Descripción	Atributo de calidad
RNF1 - Performance	El tiempo de respuesta promedio para todas las operaciones públicas del sistema en condición de funcionamiento normal deberá mantenerse por debajo de los 350 ms. para cargas de hasta 1200 req/m.	Performance
RNF2 - Confiabilidad y disponibilidad	Con el fin de poder monitorear la salud y disponibilidad del sistema, se deberá proveer un endpoint HTTP de acceso público que informe el correcto funcionamiento del sistema (conectividad con bases de datos, colas de mensajes, disponibilidad para recibir requests, etc.).	Disponibilidad
RNF3 - Configuración y manejo de secretos	Relativo al desarrollo, todo dato de configuración sensible que se maneje en el código fuente deberá poder especificarse en tiempo de ejecución mediante variables de entorno, manteniendo todo valor fuera del repositorio. Esto incluye credenciales de acceso a APIs, URLs de servicios externos, y cualquier otra configuración específica del sistema.	Modificabilidad
RNF4 - Autenticación y autorización	Se deberá establecer un control de acceso basado en roles, distinguiendo entre usuarios administradores y usuarios desarrolladores. Los permisos otorgados a los mismos deberán restringir el acceso a sus correspondientes funcionalidades, prohibiendo la interacción con cualquier otra operación pública del sistema. Los usuarios de una organización no	Seguridad

	<p>pueden bajo ningún concepto poder acceder a datos de otra organización.</p> <p>Las claves de aplicación deben ser generadas como JSON Web Tokens y el sistema debe validar que el token sea válido y tenga los permisos necesarios para acceder a los endpoints especificados en RF9 y RF10.</p>	
RNF5 - Seguridad	<p>El sistema deberá responder con código 40X a cualquier request mal formada o no reconocida por el mismo, no dejando expuesto ningún endpoint que no sea explícitamente requerido por alguna funcionalidad.</p> <p>A su vez, toda comunicación entre clientes front end y componentes de back end deberán utilizar un protocolo de transporte seguro. Por otra parte, la comunicación entre componentes de back end deberá en lo posible realizarse dentro de una red de alcance privado; de lo contrario deberán también utilizar un protocolo de transporte seguro autenticados por una clave de autenticación.</p>	Seguridad
RNF6 - Código fuente	<p>El sistema deberá ser desarrollado con un lenguaje, framework y/o librerías elegidas por el equipo, que sean tecnologías web que permitan cumplir con los requerimientos que necesitan una interfaz web, como también los que necesitan ser endpoints REST.</p> <p>Por otra parte, el control de versiones del código se deberá llevar a cabo con repositorios Git debidamente documentados, que contienen en el archivo README.md una descripción con el propósito y alcance del proyecto, así como instrucciones para configurar un nuevo ambiente de desarrollo. Para el manejo de branches, se deberá utilizar Gitflow.</p>	Modificabilidad
RNF7 - Pruebas	<p>Se deberá mantener un script de generación de planes de prueba de carga utilizando la herramienta Apache JMeter, con el objetivo de ejercitar todo el sistema simulando la actividad de múltiples</p>	Testeabilidad

	<p>usuarios concurrentes. De necesitar incluir claves o secretos, el script deberá recibir dichas configuraciones por variables de entorno.</p> <p>Además de las pruebas de carga, se deberá contar con pruebas funcionales automatizadas para el RF9 y RF10.</p>	
RNF8 - Identificación de fallas	Para facilitar la detección e identificación de fallas, se deberán centralizar y retener los logs emitidos por la aplicación en producción por un período mínimo de 24 horas.	Disponibilidad

### 7.1.3. Restricciones

ID Reestrcción	Descripción
RES1 - Diseño y estructura	La aplicación debe ser desarrollada y diseñada como un monolito.
RES2 - Servicios	Los servicios expuestos deberán ser Rest
RES3 - Repositorio	El código fuente, documentación y artefactos deberán ser gestionados en Git.
RES4 - Pruebas	Deberemos utilizar la herramienta Apache JMeter para realizar las pruebas de carga.
RES5 - Flujo de trabajo	Para el manejo de las ramas en el repositorio se deberá utilizar GitFlow.

7.2. Sprints realizados

7.2.1 Sprint 0

Link de taiga: <https://tree.taiga.io/project/zebasetas-bug-tracking/taskboard/sprint-0-1780?kanban-status=1922560>

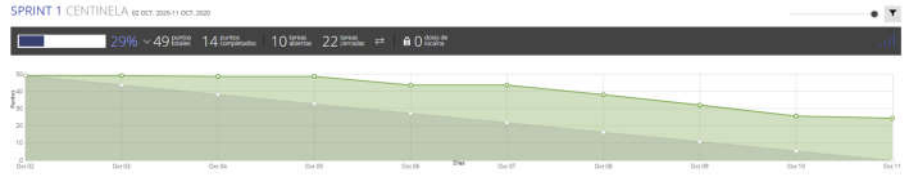
Burndown chart:



7.2.1 Sprint 1

Link de taiga: <https://tree.taiga.io/project/zebasetas-bug-tracking/taskboard/sprint-1-17674?kanban-status=1922560>

Resultado:



7.2.1 Sprint 2

Link de taiga: <https://tree.taiga.io/project/zebasetas-bug-tracking/taskboard/sprint-3-6681?kanban-status=1922560>





7.2.1 Sprint 3

Link de taiga: <https://tree.taiga.io/project/zebasetas-bug-tracking/taskboard/sprint-3-6755?kanban-status=1922560>

Resultado:



7.3. Api rest

Login		▼
POST	/api/v1/login	Login
Bugs		▼
POST	/api/v1/bugs	Create bug
GET	/api/v1/bugs	Get all bugs
PUT	/api/v1/bugs/:id	Update bug
GET	/bugs/:id	Get bug
Invitations		▼
GET	/api/v1/invitations/:id	Get invitation
POST	/api/v1/invitations	Create invitation
Is alive		▼
GET	/api/v1/ping	Is alive
Organizations		▼
GET	/api/v1/organizations/:id	Get organization
Users		▼
POST	/api/v1/users	Create user with organization
GET	/users	Get users

Reports		▼
GET	/api/v1/reports/critical	Critical errors report
GET	/api/v1/reports/statistics	Statistics errors report
Systems		▼
POST	/api/v1/systems	Create system
GET	/api/v1/systems	Get systems
POST	/api/v1/systems/:id/environments	Create an environment in a system
GET	/api/v1/systems/:id/environments	Get environments from a system
Monitoring		▼
GET	/monitoring/v1/ping	Monitoring

## 7.4. URLs de la aplicación

URL de Swagger con la descripción general de la API:

<http://centinela-env.eba-zhfdmg23.us-east-1.elasticbeanstalk.com/api/v1/docs/>

URL del sistema de backend:

<http://centinela-env.eba-zhfdmg23.us-east-1.elasticbeanstalk.com/>

URL del frontend:

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

URL del sistema de gestión del proyecto:

<https://tree.taiga.io/project/zebasetas-bug-tracking/backlog>

URL del repositorio de GitHub:

<https://github.com/ASP-N8A/Larrosa-Settimo-Zawrzykraj>