



Universidad ORT Uruguay
Facultad de Ingeniería

Arquitectura de Software en la Práctica

Obligatorio 2

Profesores

Guillermo Areosa
Nahuel Biladóniga

Estudiante

Santiago Díaz - 240926

Repositorio:

<https://github.com/SantiagoDiaz9822/ASP-obligatorio>

1. Descripción de la Arquitectura.....	2
1.1 Distribución Física.....	2
Justificaciones y explicaciones del diagrama:.....	4
1. División en microservicios.....	4
2. Microservicios y sus responsabilidades.....	4
3. Conexiones entre microservicios.....	5
4. Bases de datos separadas.....	5
5. Interacción con bases de datos.....	6
6. Razonamiento detrás de la conexión Auditoría → Usuarios.....	6
7. Integración con Auditoría y AWS SQS.....	6
Beneficios de esta estructura.....	6
1.2 Flujo de Información.....	6
2. Justificación de Diseño para los Requerimientos No Funcionales.....	8
RNF1. Performance.....	8
RNF2. Confiabilidad y disponibilidad.....	8
RNF3. Observabilidad.....	9
RNF4. Autenticación, autorización y tenancy.....	9
RNF5. Seguridad.....	10
RNF6. Código fuente.....	10
RNF7. Integración continua.....	11
RNF8. Pruebas de carga.....	15
RNF9. Identificación de fallas.....	16
RNF10. Portabilidad.....	16
RNF11: Estilo de arquitectura.....	16
RNF12: Desplegabilidad.....	17
3. Cumplimiento con los 12 factores de una app.....	18
4. Descripción del proceso de Deployment.....	19
Plataforma.....	19
Proceso de Producción.....	20

1. Descripción de la Arquitectura

1.1 Distribución Física

La arquitectura del sistema está compuesta por varios microservicios y componentes que interactúan entre sí para proporcionar una solución integral.

Estos incluyen:

- Frontend:
 - Implementado en React que interactúa con los microservicios backend a través de APIs RESTful.
 - Conectado con los microservicios mediante servicios de red Elastic Beanstalk.
- Backend (Microservicios):
 - Servicio de Empresas: Gestiona las empresas.
 - Servicio de Usuarios: Gestiona los usuarios, su autenticación y autorización.
 - Servicio de Proyectos: Gestiona los proyectos asociados a las empresas.
 - Servicio de Features: Gestiona las features de los proyectos.
 - Servicio de Reportes: Proporciona informes sobre los llamados de las evaluaciones de las features.
 - Servicio de Auditoría: Registra acciones críticas de los diferentes microservicios y agrega un mensaje a la cola de mensaje sqs para enviar las notificaciones por email.
- Bases de Datos:
 - MySQL (RDS): Utilizado para almacenar los datos relacionados con empresas, usuarios, proyectos, funcionalidades, reportes y auditoría.
 - Redis (ElastiCache): Implementado para mejorar el rendimiento de la aplicación almacenando el caché de reportes.
- Almacenamiento de Archivos:
 - AWS S3: Se utiliza para almacenar los logos de las empresas.
- Cola de mensaje
 - AWS SQS: se utiliza para generar colas de mensaje para el envío de notificación.
- Procesamiento de cola de mensaje:
 - AWS Lambda: se utiliza para procesar la cola de mensaje y enviar los mail de notificación a los usuarios correspondientes.

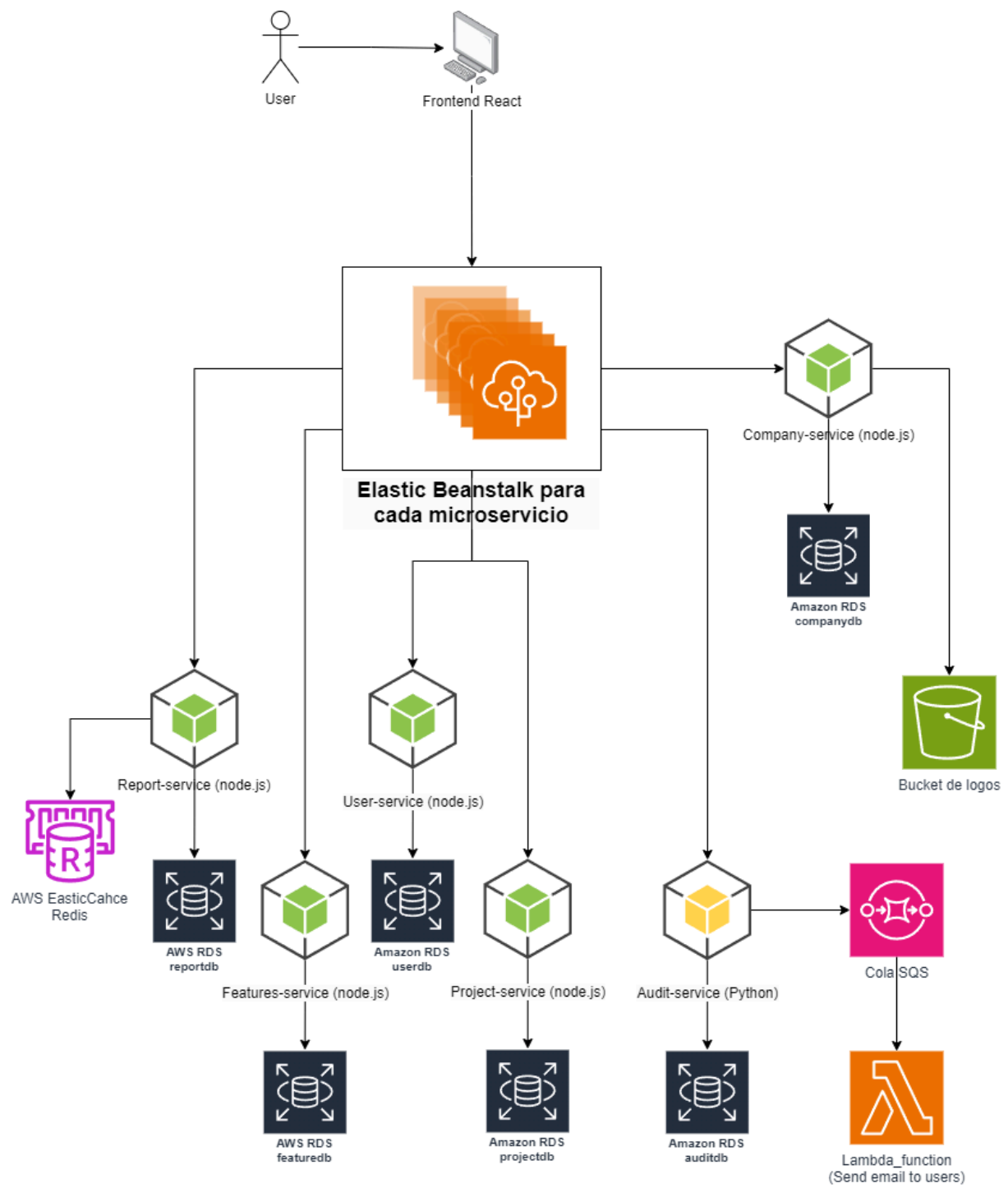
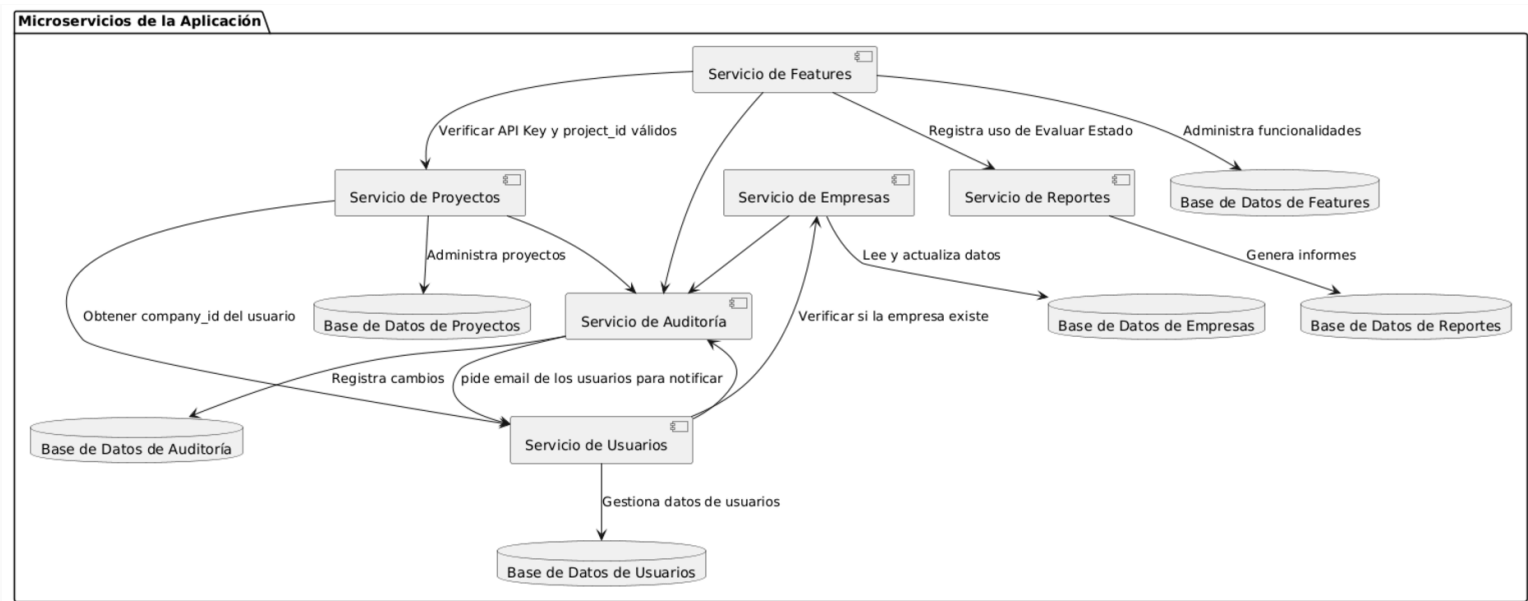


Diagrama de Microservicios y Bases de Datos:



Justificaciones y explicaciones del diagrama:

1. División en microservicios

La arquitectura de microservicios permite:

- Independencia de desarrollo y despliegue: Cada servicio se desarrolla, implementa y escala de manera independiente.
- Especialización: Cada servicio tiene pocas responsabilidades y claramente definidas.
- Escalabilidad: Todos los microservicios pueden escalar sin impactar a otros.

2. Microservicios y sus responsabilidades

- Servicio de Empresas:
 - Gestiona las empresas y sus datos relacionados.
 - Interactúa con la Base de Datos de Empresas para leer y almacenar información de las empresas.
- Servicio de Usuarios:
 - Gestiona autenticación y autorización.
 - Antes de asignarle una empresa a un usuario valida si las empresas existen mediante el Servicio de Empresas.
 - Interactúa con la Base de Datos de Usuarios para leer y almacenar información de los usuarios.
- Servicio de Proyectos:
 - Gestiona los proyectos y sus datos relacionados.
 - Consulta al Servicio de Usuarios para verificar el company_id del usuario.
 - Interactúa con la Base de Datos de Proyectos para leer y almacenar información de los proyectos.
- Servicio de Features:

- Gestiona las Features y sus datos relacionados.
- Antes de ser asignada a un proyecto, verifica datos clave como API Key y project_id con el Servicio de Proyectos.
- Interactúa con el Servicio de Reportes para registrar el uso de evaluaciones.
- Interactúa con la Base de Datos de Features para leer y almacenar información de las features.
- Servicio de Reportes:
 - Genera informes basados en datos de uso y evaluación.
 - Interactúa con la Base de Datos de Reportes para leer y almacenar información de los reportes.
- Servicio de Auditoría:
 - Registra cambios en funcionalidades y acciones clave de los otros microservicios.
 - Interactúa con la Base de Datos de Auditoría para leer y almacenar información de las auditorías.
 - Solicita emails de usuarios al Servicio de Usuarios para notificaciones.
 - También interactúa con AWS SQS para enviar mensajes para notificar los cambios de las features por mail.

3. Conexiones entre microservicios

- Empresas, Usuarios, Proyectos, Features → Auditoría:
 - Se registran las acciones críticas (crear, eliminar y modificar) de cada servicio.
- Usuarios → Empresas:
 - Verifica si la empresa que se le intenta asociar a un usuario existe.
- Proyectos → Usuarios:
 - Consulta al Servicio de Usuarios para obtener el company_id asociado a un proyecto para asociar esa empresa al proyecto que el usuario está creando.
- Features → Proyectos:
 - Verifica la validez de la API Key y el project_id del proyecto donde se quiere crear la feature.
- Features → Reportes:
 - Registra el uso del servicio de evaluación de estados en el Servicio de Reportes.
- Auditoría → Usuarios:
 - Solicita el email de los usuarios para enviar notificaciones relacionadas a las features.

4. Bases de datos separadas

Cada servicio tiene su propia base de datos:

- Esto asegura independencia entre los microservicios y facilita el escalado horizontal.
- Evita conflictos de datos y permite optimizar cada base para su servicio específico.

5. Interacción con bases de datos

- Cada servicio accede exclusivamente a su propia base de datos para operaciones de lectura y escritura, manteniendo un modelo desacoplado:
 - Ejemplo: *La Base de Datos de Empresas* es gestionada únicamente por el Servicio de Empresas.

6. Razonamiento detrás de la conexión Auditoría → Usuarios

El Servicio de Auditoría necesita información de contacto (emails) para enviar notificaciones por cambios registrados. En lugar de acceder directamente a los datos de los usuarios, utiliza el Servicio de Usuarios, promoviendo encapsulamiento y seguridad.

7. Integración con Auditoría y AWS SQS

- El Servicio de Features registra las acciones realizadas en el Servicio de Auditoría.
- El Servicio de Auditoría utiliza AWS SQS para desacoplar la generación de notificaciones, evitando así que el usuario tenga que esperar a que el sistema le envíe el mail a todos los usuarios suscritos de la empresa cada vez que se realiza algún cambio en el Servicio de Features.

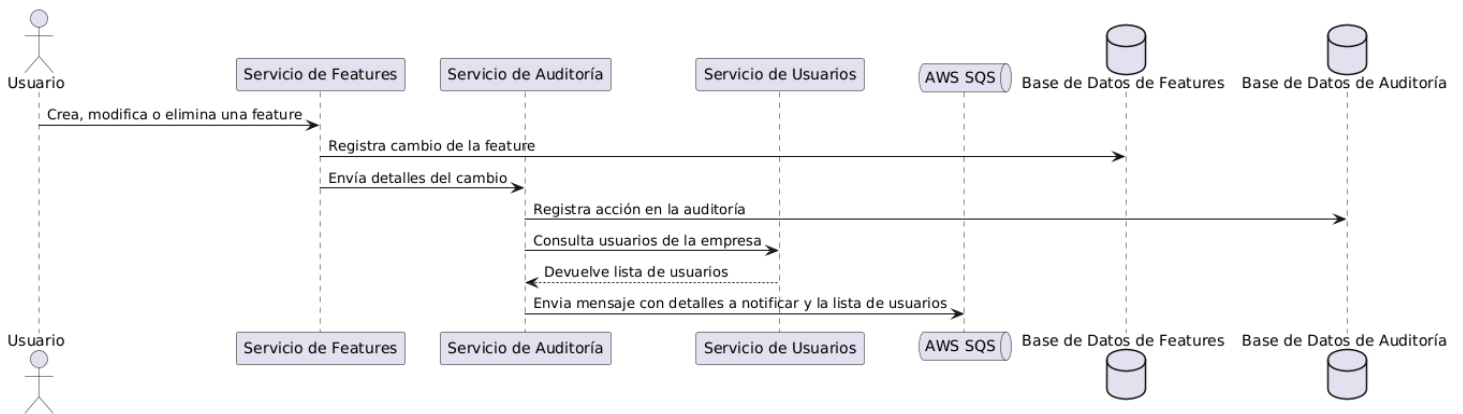
Beneficios de esta estructura

- Escalabilidad: Cada servicio puede escalar según sus necesidades de uso.
- Modularidad: Facilita el mantenimiento y la introducción de nuevos servicios o funcionalidades.
- Seguridad: Las responsabilidades bien definidas y la separación de bases de datos evitan el acceso no autorizado.
- Resiliencia: Si un servicio falla, otros pueden continuar funcionando sin interrupción.

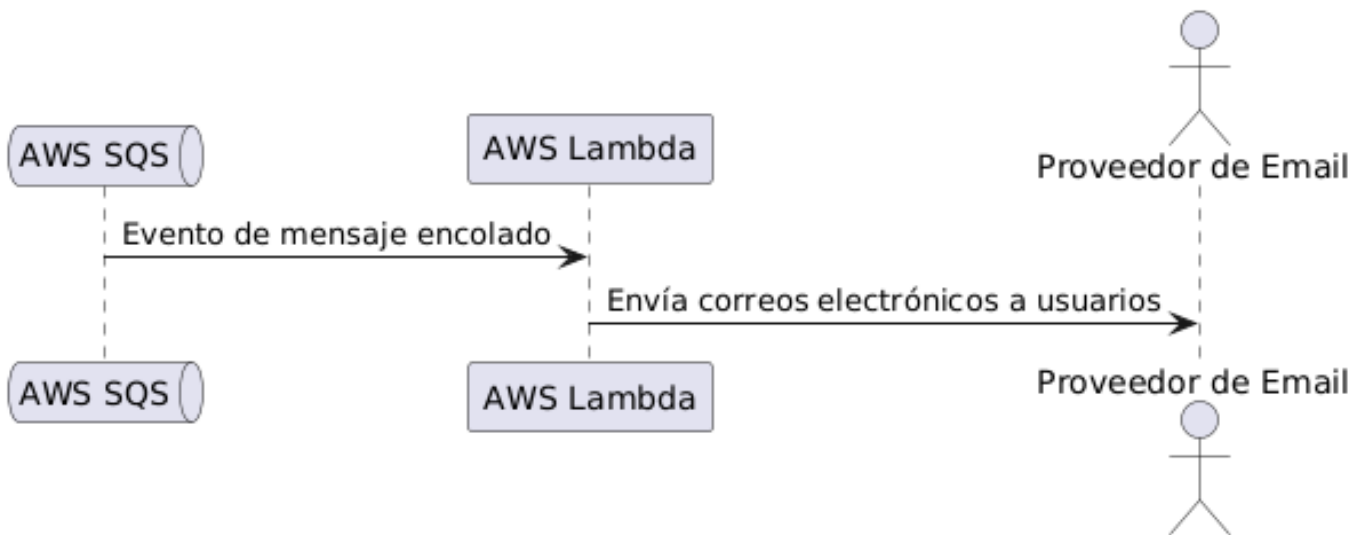
1.2 Flujo de Información

1. El frontend envía solicitudes HTTP a los microservicios a través de Elastic Beanstalks.
2. Los microservicios interactúan entre sí mediante RESTful APIs para proporcionar la lógica de negocio necesaria, tales como la creación de empresas, la asignación de usuarios y la gestión de proyectos.
3. Redis es utilizado para almacenar datos en caché, como los reportes de evaluación de las features.
4. Las bases de datos RDS MySQL se encargan de almacenar la información de manera persistente.
5. S3 se utiliza para almacenar los logos de las empresas.
6. El servicio de auditoría mantiene un registro de todas las acciones realizadas sobre los datos de la aplicación.
7. El servicio de auditoría encola un mensaje SQS cuando se realiza una acción en alguna feature y luego una función lambda (que se “dispara” cada vez que se manda un mensaje en la cola sqs) es la encargada de enviar la notificación por mail a todos los usuarios de la empresa.

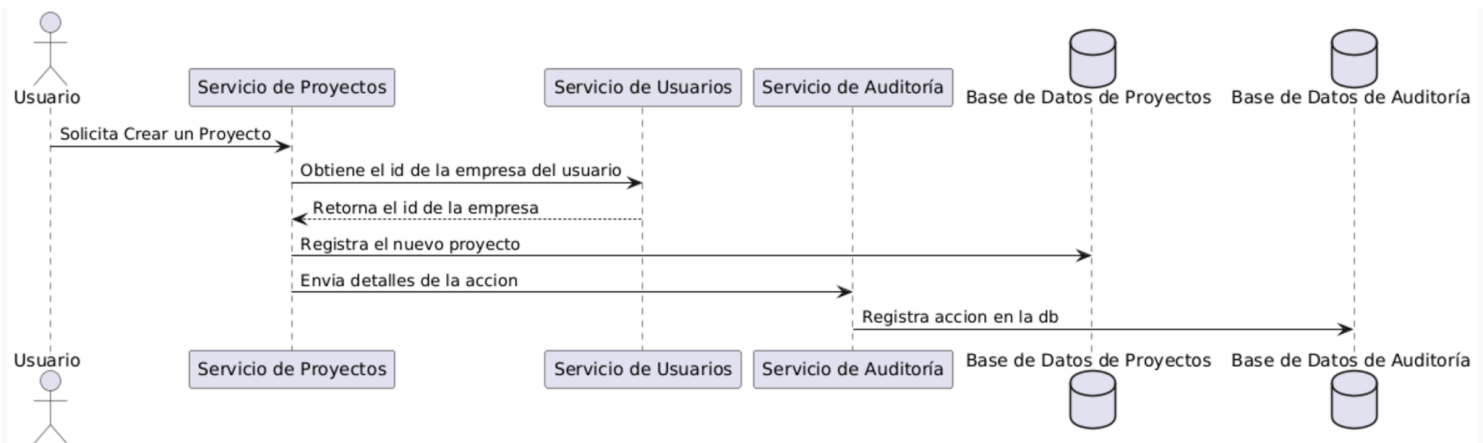
Flujo cuando se realiza algún cambio en una feature:



Y luego de que se envía el mensaje a la cola SQS se dispara la lambda que envía los correos a los usuarios suscritos:



Otro flujo básico es cuando la acción crítica no es de una feature:



Como se puede ver, este flujo sólo registra la acción, pero no envía ningún mensaje a la cola SQS

2. Justificación de Diseño para los Requerimientos No Funcionales

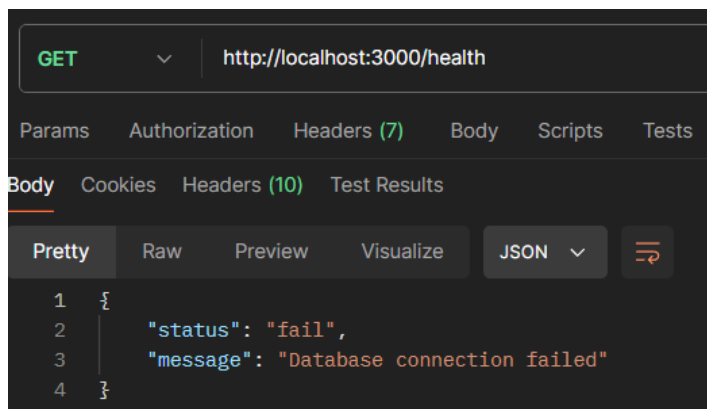
RNF1. Performance

- **Diseño aplicado:** Para cumplir con el requisito de responder las consultas del RF6 en menos de 100 ms para cargas de hasta 1200 req/m, se implementó un sistema de caching utilizando Redis (Amazon ElastiCache). Redis almacena temporalmente los resultados de consultas que requieren mucho procesamiento, lo que reduce la carga en la base de datos y mejora significativamente los tiempos de respuesta.
- **Tácticas aplicadas:** Redis se utiliza como una capa de almacenamiento rápido en memoria. Además, la estructura de la base de datos ha sido optimizada con índices en los campos de búsqueda clave. Finalmente, se han realizado ajustes de configuración en AWS para asegurar que la infraestructura pueda manejar el tráfico previsto.

RNF2. Confiabilidad y disponibilidad

- **Diseño aplicado:** Se implementó un endpoint HTTP de acceso público `/health` que verifica la conectividad con la base de datos MySQL, Redis y otros componentes clave. Este endpoint permite el monitoreo de la disponibilidad del sistema.
- **Tácticas aplicadas:** Este endpoint hace una llamada simple a los servicios principales y retorna un estado HTTP 200 si todo funciona correctamente, o un 500 si hay algún problema.

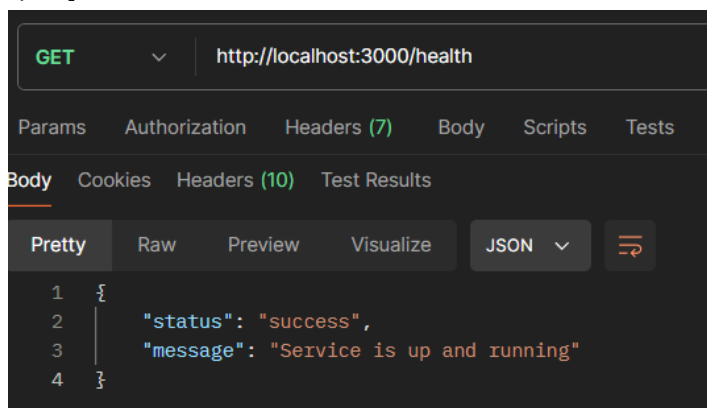
Ejemplo base de datos no corriendo:



A screenshot of a REST client interface showing a GET request to `http://localhost:3000/health`. The response body is displayed in JSON format, indicating a failure:

```
1 {
2   "status": "fail",
3   "message": "Database connection failed"
4 }
```

Ejemplo sistema corriendo correctamente:



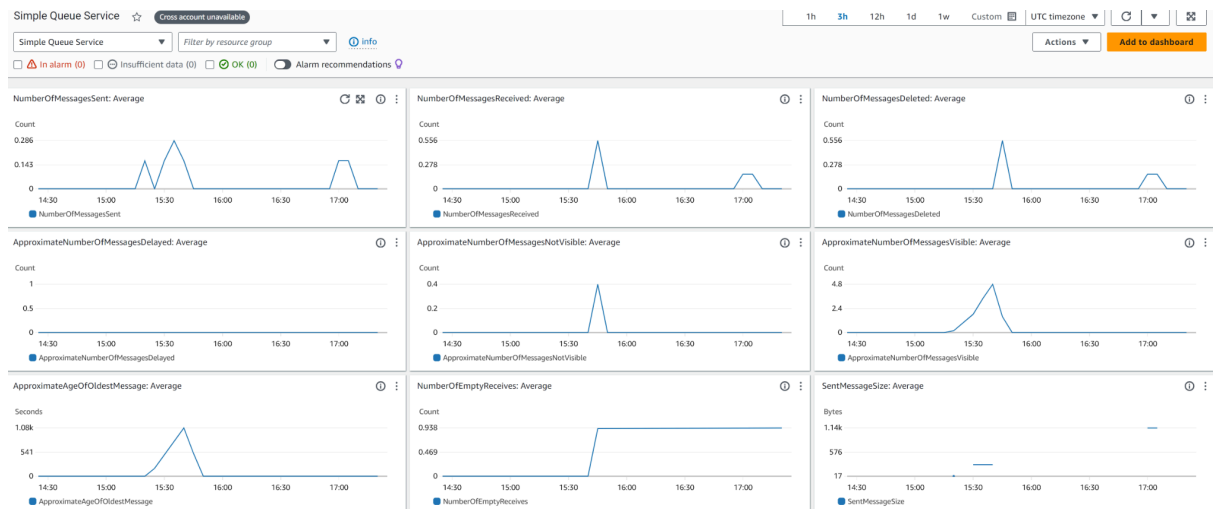
A screenshot of a REST client interface showing a GET request to `http://localhost:3000/health`. The response body is displayed in JSON format, indicating a successful status:

```
1 {
2   "status": "success",
3   "message": "Service is up and running"
4 }
```

RNF3. Observabilidad

- **Diseño aplicado:** Para monitorear el rendimiento y el uso del sistema, se integró la herramienta de monitoreo AWS CloudWatch que recolecta métricas de rendimiento.
- **Tácticas aplicadas:** Se configuraron logs detallados para capturar métricas de las peticiones y tiempos de respuesta en cada endpoint de los microservicios. Estas métricas se envían a una herramienta de monitoreo que las procesa y almacena.

Ejemplo - Métricas CloudWatch del uso de la cola de mensajes:



RNF4. Autenticación, autorización y tenancy

- **Diseño aplicado:** El sistema implementa control de acceso basado en roles, utilizando JSON Web Tokens (JWT) para la autenticación y bcrypt para hashear las contraseñas de los usuarios, asegurando que la información sensible esté protegida. Además, se utiliza middleware personalizado para proteger las rutas y autorizar el acceso a cada una de las funcionalidades según los roles de los usuarios.
- **Tácticas aplicadas:** Las rutas del backend están protegidas por un middleware que verifica si el token JWT del usuario es válido y tiene los permisos adecuados para acceder al recurso solicitado. Este middleware también asegura que los usuarios solo puedan acceder a los datos de su propia empresa. El uso de bcrypt para el hashing de contraseñas garantiza que estas se almacenen de forma segura y no sean recuperables en caso de acceso no autorizado a la base de datos.

Ejemplo de contraseñas hasheadas:

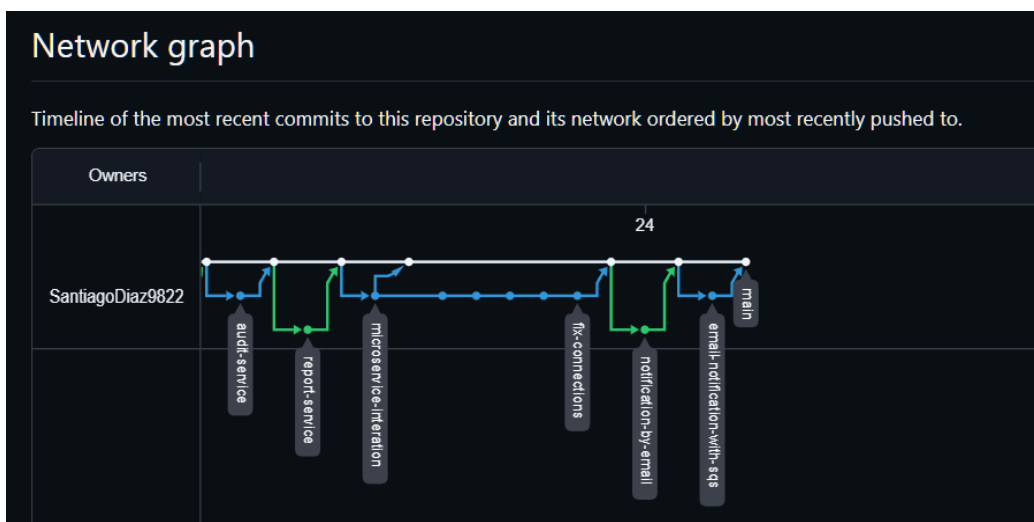
```
[
  {
    "id": 1,
    "company_id": 1,
    "email": "admin@example.com",
    "password_hash": "$2a$10$jPakKpcf2lIsaRi1tzstduVvFE93B4irnYBGMa0LvFI0SFAZ/0Jk.",
    "role": "admin",
    "first_login": 1,
    "created_at": "2024-11-23T21:48:09.000Z",
    "updated_at": "2024-11-24T05:59:04.000Z",
    "is_subscribed": 0
  },
]
```

RNF5. Seguridad

- **Diseño aplicado:** Actualmente, el frontend se comunica con los microservicios desplegados en Elastic Beanstalk a través de HTTP utilizando el dominio. Esto implica que las comunicaciones no están cifradas, lo que representa un riesgo de seguridad.
- **Tácticas aplicadas:** Para cumplir con este requerimiento en producción, se planea configurar HTTPS en Elastic Beanstalk. Esto se logrará adquiriendo y configurando un certificado SSL/TLS para habilitar la comunicación segura. El uso de HTTPS garantizará que las transmisiones de datos entre el frontend y los microservicios estén protegidas frente a interceptaciones.

RNF6. Código fuente

- **Diseño aplicado:** El proyecto se gestiona con GitHub, siguiendo la metodología GitHub Flow. El archivo **README.md** incluye instrucciones detalladas sobre cómo configurar el ambiente de desarrollo y las convenciones de manejo de branches.
- **Tácticas aplicadas:** El uso de Pull Requests y revisiones de código asegura la calidad del código. La metodología GitHub Flow permite un ciclo de vida de desarrollo simplificado y una integración continua ágil.



RNF7. Integración continua

- **Diseño aplicado:** Se configuró un pipeline de integración continua (CI) que ejecuta pruebas unitarias automáticas para el microservicio de empresas y el microservicio de reportes, asegurando una cobertura del 100%. Las pruebas se ejecutan automáticamente cada vez que se integra un commit a la rama principal.
- **Tácticas aplicadas:** Para esto, se configuró la herramienta de CI/CD GitHub Actions, que ejecuta pruebas de cada funcionalidad crítica del sistema en cada commit. Esto garantiza que las funcionalidades se mantengan estables durante el desarrollo.

Cobertura client-service controller:

```
PASS controllers/companyController.test.js
Company Controller
  createCompany
    ✓ should create a company and return success (34 ms)
    ✓ should handle error when query fails (67 ms)
    ✓ should handle error when audit service fails (17 ms)
  deleteCompany
    ✓ should delete a company and return success (6 ms)
    ✓ should handle error when query fails (13 ms)
    ✓ should handle error when audit service fails (8 ms)
  getAllCompanies
    ✓ should return all companies (1 ms)
    ✓ should handle error when query fails (12 ms)
  getCompanyById
    ✓ should return a company by id (3 ms)
    ✓ should return 404 if company not found (1 ms)
    ✓ should handle error when query fails (13 ms)

-----|-----|-----|-----|-----|-----
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files |    100 |    91.66 |    100 |    100 |
companyController.js |    100 |    91.66 |    100 |    100 |      8
-----|-----|-----|-----|-----|-----

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   0 total
Time:        1.383 s, estimated 11 s
Ran all test suites.
```

Cobertura report-service controller.js

```
PASS controllers/reportController.test.js
reportController
  registerUsageLog
    ✓ debe registrar un log de uso exitosamente (8 ms)
    ✓ debe devolver error 400 cuando los datos son inválidos (28 ms)
    ✓ debe devolver error 500 cuando hay un error al registrar el log (75 ms)
  getUsageReport
    ✓ debe devolver el reporte de uso correctamente (3 ms)
    ✓ debe devolver error 400 si falta el rango de fechas (1 ms)
    ✓ debe devolver error 500 si hay un error al obtener el reporte (13 ms)
    ✓ debe aplicar filtro por feature_id si se pasa en la solicitud (2 ms)

-----|-----|-----|-----|-----|-----
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files |    100   |    100   |    100   |    100   |
reportController.js |    100   |    100   |    100   |    100   |
-----|-----|-----|-----|-----|-----

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        1.053 s
Ran all test suites.
```

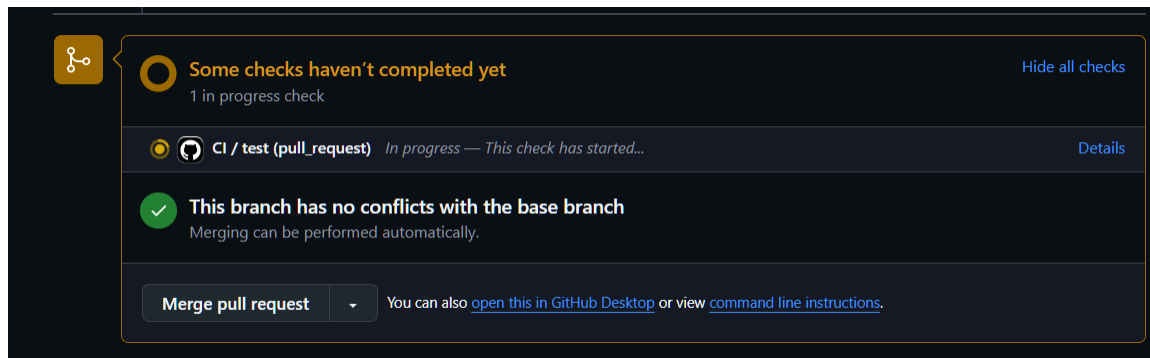
Workflow file:

Workflow file for this run

[.github/workflows/main.yml](#) at 520a737

```
1  name: CI
2
3  on:
4    push:
5      branches:
6        - main
7    pull_request:
8      branches:
9        - main
10
11  jobs:
12    test:
13      runs-on: ubuntu-latest
14
15      steps:
16        - name: Checkout code
17          uses: actions/checkout@v2
18
19        - name: Setup Node.js
20          uses: actions/setup-node@v2
21          with:
22            node-version: "14"
23
24        - name: Install dependencies for company-service
25          working-directory: ./company-service
26          run: npm install
27
28        - name: Run tests for company-service
29          working-directory: ./company-service
30          run: npm test
31
32        - name: Install dependencies for report-service
33          working-directory: ./report-service
34          run: npm install
35
36        - name: Run tests for report-service
37          working-directory: ./report-service
38          run: npm test
```

Ejemplo de uso con pull request:



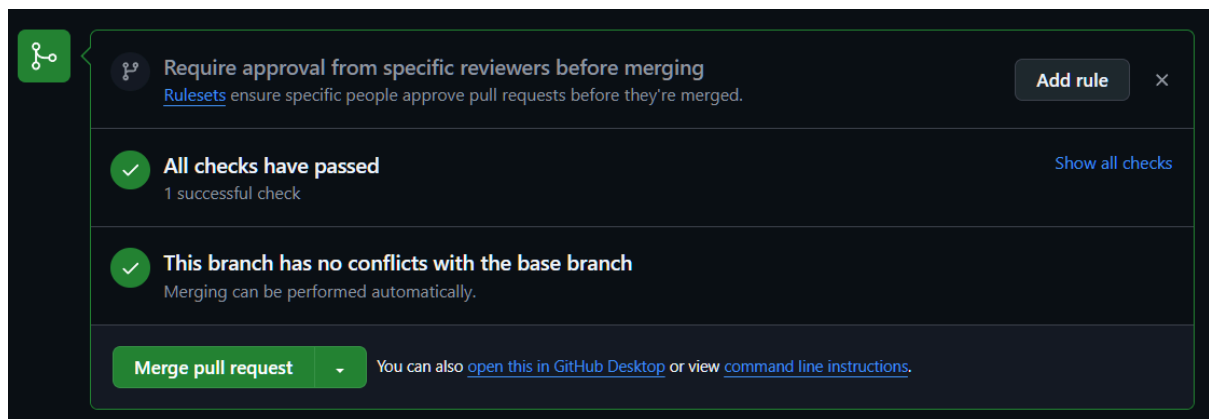
This screenshot shows a pull request status bar with a yellow icon. It indicates that some checks have not completed yet, with one check in progress. The progress bar is partially filled. The check is labeled 'CI / test (pull_request)' and is in progress. Below this, a green checkmark indicates that the branch has no conflicts with the base branch, allowing for automatic merging. A 'Merge pull request' button is visible at the bottom.

Some checks haven't completed yet Hide all checks
1 in progress check

CI / test (pull_request) *In progress* — *This check has started...* Details

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request ▼ You can also [open this in GitHub Desktop](#) or view [command line instructions](#).



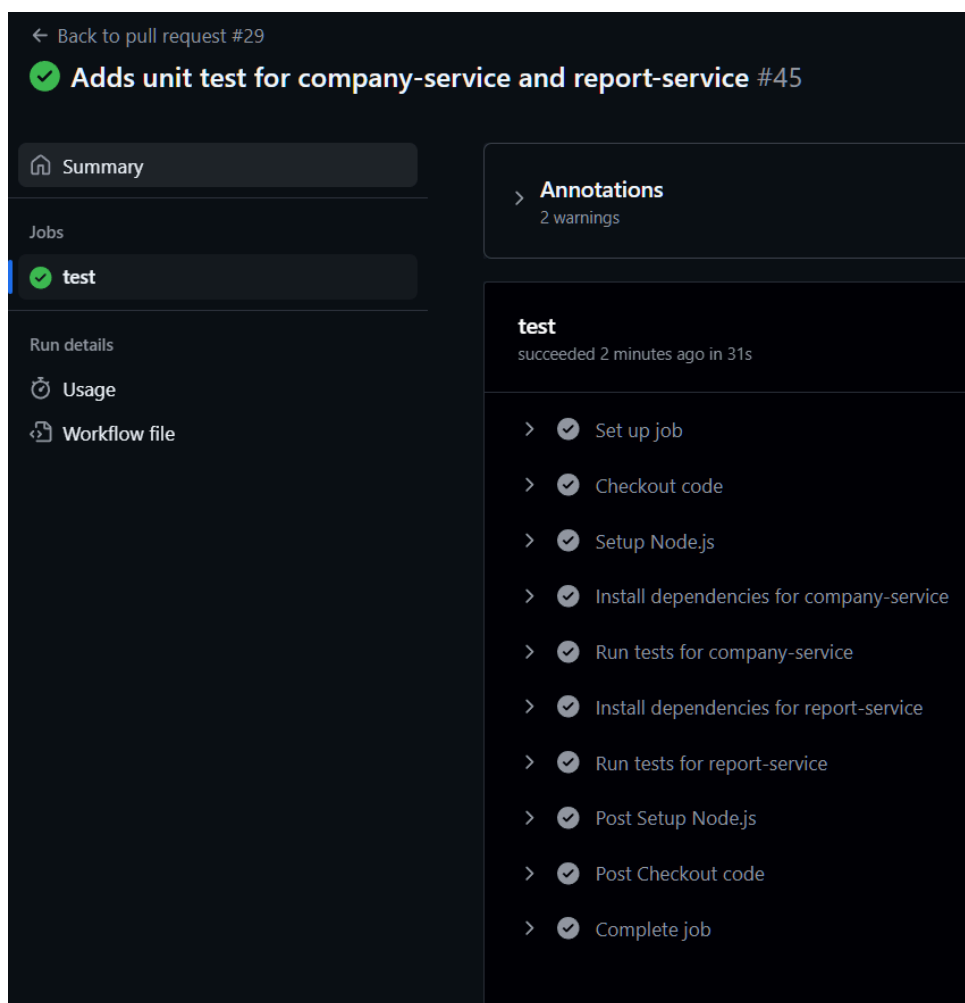
This screenshot shows a pull request status bar with a green icon. It indicates that all checks have passed, with one successful check. The progress bar is fully green. The check is labeled 'All checks have passed' and is successful. Below this, a green checkmark indicates that the branch has no conflicts with the base branch, allowing for automatic merging. A 'Merge pull request' button is visible at the bottom.

Require approval from specific reviewers before merging Add rule ×
[Rulesets](#) ensure specific people approve pull requests before they're merged.

All checks have passed Show all checks
1 successful check

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request ▼ You can also [open this in GitHub Desktop](#) or view [command line instructions](#).



This screenshot shows the details of a pull request. The top section shows a green checkmark and the title 'Adds unit test for company-service and report-service #45'. Below this, there is a sidebar with 'Summary', 'Jobs', 'Run details', 'Usage', and 'Workflow file'. The 'Jobs' section is expanded, showing a list of jobs. The 'test' job is highlighted, showing a green checkmark and the status 'succeeded 2 minutes ago in 31s'. The job steps are listed below, each with a green checkmark indicating success.

[← Back to pull request #29](#)

Adds unit test for company-service and report-service #45

[Summary](#)

Jobs

- test**

Run details

Usage

Workflow file

Annotations
2 warnings

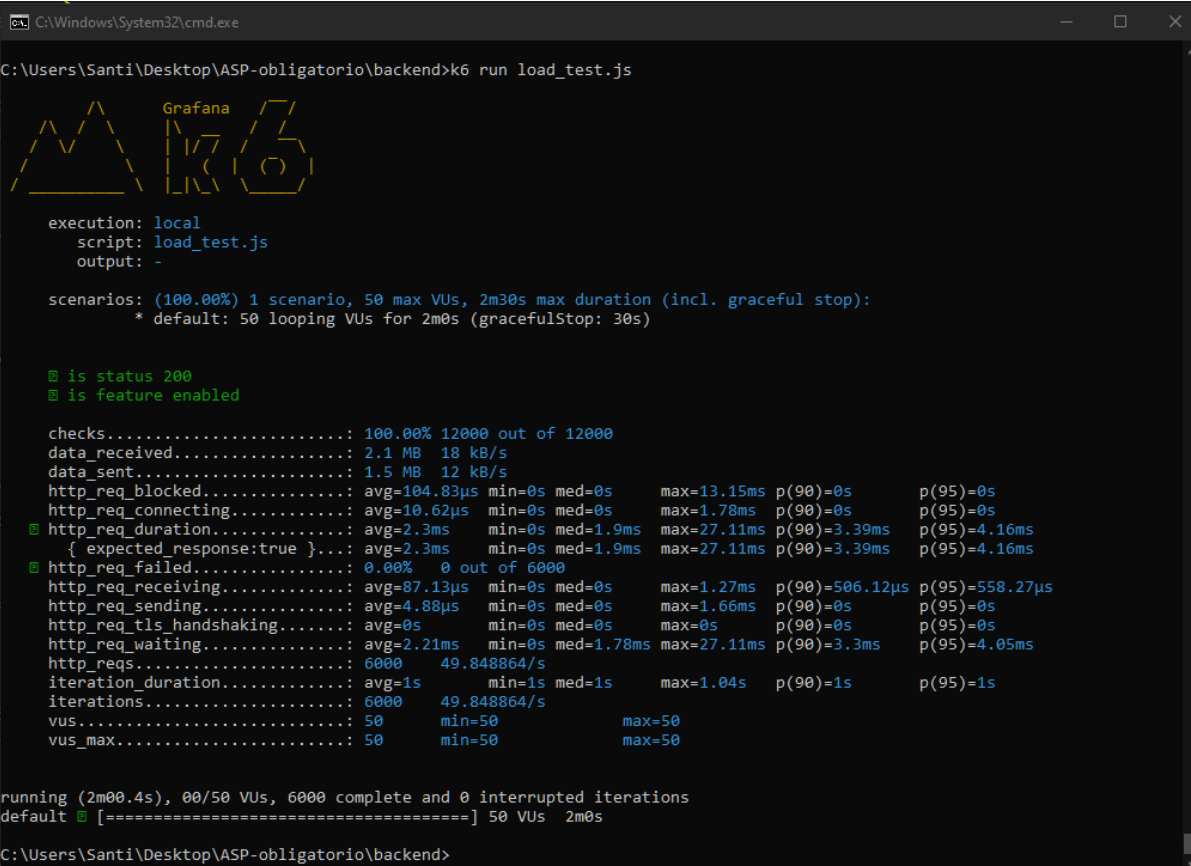
test
succeeded 2 minutes ago in 31s

- > **Set up job**
- > **Checkout code**
- > **Setup Node.js**
- > **Install dependencies for company-service**
- > **Run tests for company-service**
- > **Install dependencies for report-service**
- > **Run tests for report-service**
- > **Post Setup Node.js**
- > **Post Checkout code**
- > **Complete job**

RNF8. Pruebas de carga

- **Diseño aplicado:** Se realizaron pruebas de carga utilizando la herramienta k6.io, simulando el comportamiento de la aplicación bajo distintas cargas de usuarios para asegurar que pueda manejar las 1200 req/m con un tiempo de respuesta menor a 100 ms.
- **Tácticas aplicadas:** Se crearon scripts que generan múltiples solicitudes simultáneas a los endpoints críticos, evaluando el comportamiento del sistema bajo condiciones de alta demanda y ajustando parámetros de la infraestructura si fuera necesario.

Evidencia prueba de carga:



```
C:\Windows\System32\cmd.exe
C:\Users\Santi\Desktop\ASP-obligatorio\backend>k6 run load_test.js

Grafana

execution: local
script: load_test.js
output: -

scenarios: (100.00%) 1 scenario, 50 max VUs, 2m30s max duration (incl. graceful stop):
* default: 50 looping VUs for 2m0s (gracefulStop: 30s)

[ ] is status 200
[ ] is feature enabled

checks.....: 100.00% 12000 out of 12000
data_received.....: 2.1 MB 18 kB/s
data_sent.....: 1.5 MB 12 kB/s
http_req_blocked.....: avg=104.83µs min=0s med=0s max=13.15ms p(90)=0s p(95)=0s
http_req_connecting.....: avg=10.62µs min=0s med=0s max=1.78ms p(90)=0s p(95)=0s
[ ] http_req_duration.....: avg=2.3ms min=0s med=1.9ms max=27.11ms p(90)=3.39ms p(95)=4.16ms
{ expected_response:true }...: avg=2.3ms min=0s med=1.9ms max=27.11ms p(90)=3.39ms p(95)=4.16ms
[ ] http_req_failed.....: 0.00% 0 out of 6000
http_req_receiving.....: avg=87.13µs min=0s med=0s max=1.27ms p(90)=506.12µs p(95)=558.27µs
http_req_sending.....: avg=4.88µs min=0s med=0s max=1.66ms p(90)=0s p(95)=0s
http_req_tls_handshaking.....: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
http_req_waiting.....: avg=2.21ms min=0s med=1.78ms max=27.11ms p(90)=3.3ms p(95)=4.05ms
http_reqs.....: 6000 49.848864/s
iteration_duration.....: avg=1s min=1s med=1s max=1.04s p(90)=1s p(95)=1s
iterations.....: 6000 49.848864/s
vus.....: 50 min=50 max=50
vus_max.....: 50 min=50 max=50

running (2m00.4s), 00/50 VUs, 6000 complete and 0 interrupted iterations
default [ ] [=====] 50 VUs 2m0s

C:\Users\Santi\Desktop\ASP-obligatorio\backend>
```

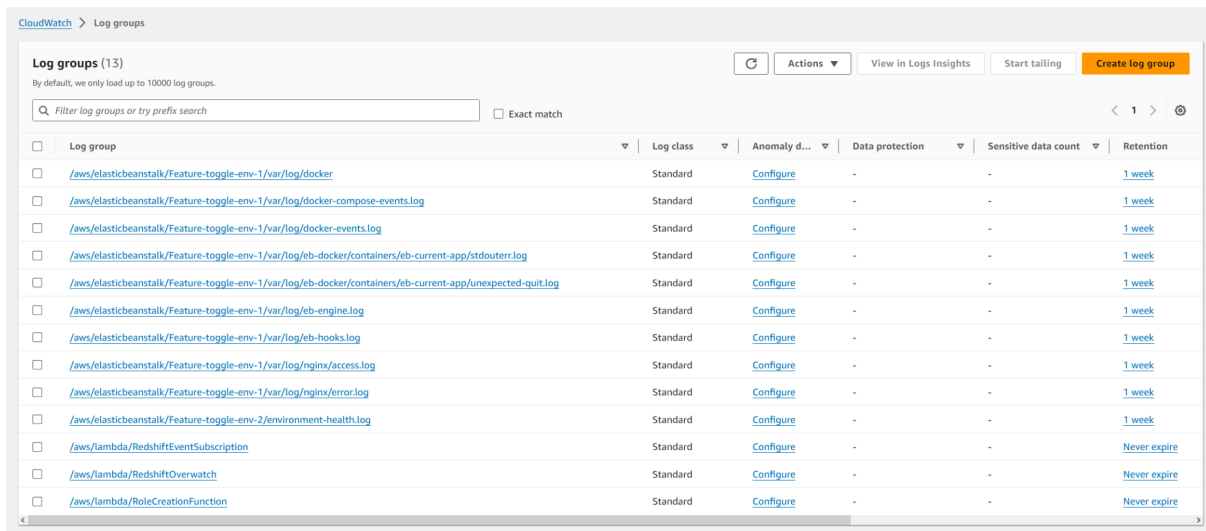
Resultados:

- Total de solicitudes: 6000
- Tasa de éxito: 100%
- Tiempo promedio de respuesta: 2.3 ms
- Tiempos de respuesta máximos: 27.11 ms
- Porcentaje de solicitudes por debajo de 200 ms: 100%
- Tasa de fallos: 0%

Los resultados demuestran que el sistema es capaz de manejar la carga esperada sin problemas significativos de rendimiento.

RNF9. Identificación de fallas

- **Diseño aplicado:** Los logs de los microservicios en producción se centralizan utilizando AWS CloudWatch Logs, reteniendo estos logs durante un período de al menos 24 horas. Estos logs ayudan a identificar fallas de manera eficiente.
- **Tácticas aplicadas:** Cada microservicio genera logs detallados de errores, accesos, y fallos en un formato que permite su análisis posterior. Los logs se almacenan y se pueden consultar a través de una interfaz gráfica para diagnóstico en caso de fallas.



The screenshot shows the AWS CloudWatch 'Log groups' page. It displays a list of 13 log groups. The table has columns for Log group, Log class, Anomaly detection, Data protection, Sensitive data count, and Retention. The log groups are for various services including Elastic Beanstalk, Lambda, and Redshift.

Log group	Log class	Anomaly d...	Data protection	Sensitive data count	Retention
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/docker	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/docker-compose-events.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/docker-events.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/eb-docker/containers/eb-current-app/stdouterr.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/eb-docker/containers/eb-current-app/unexpected-quit.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/eb-engine.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/eb-hooks.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/nginx/access.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-1/var/log/nginx/error.log	Standard	Configure	-	-	1 week
/aws/elasticbeanstalk/Feature-toggle-env-2/environment-health.log	Standard	Configure	-	-	1 week
/aws/lambda/RedshiftEventSubscription	Standard	Configure	-	-	Never expire
/aws/lambda/RedshiftOverwatch	Standard	Configure	-	-	Never expire
/aws/lambda/RoleCreationFunction	Standard	Configure	-	-	Never expire

RNF10. Portabilidad

- **Diseño aplicado:** Se proporcionan archivos de configuración para Docker que permiten que cualquier persona que clone el repositorio pueda levantar el sistema localmente usando un solo comando (`docker-compose up`).
- **Tácticas aplicadas:** El uso de contenedores Docker asegura que los microservicios se ejecuten de manera consistente en cualquier entorno, y el archivo `docker-compose.yml` facilita la configuración automática de todos los servicios necesarios (bases de datos, microservicios, frontend).

RNF11: Estilo de arquitectura

1. Microservicios:

La arquitectura de la aplicación se organiza mediante microservicios, cada uno con fronteras claras y responsabilidades definidas. Los microservicios están diseñados para ser independientes, lo que permite que cada servicio se despliegue, se escale y se mantenga por separado. Esto se logra mediante la creación de diferentes servicios para manejar aspectos como la gestión de usuarios, la gestión de empresas, los proyectos, las features, los reportes y la auditoría.

2. Comunicación entre Microservicios:

- Todos los microservicios se comunican a través de llamadas HTTP.

3. **Comunicación mediante de cola de mensaje:**

- El servicio de auditoría utiliza una cola de mensaje para enviar la notificación de acciones de una feature al mail de los usuarios correspondientes.

4. **Lenguajes diferentes:**

- El microservicio de auditoría está desarrollado en Python, mientras que el resto de los microservicios utilizan Node.js.

RNF12: Desplegabilidad

1. **Despliegue Independiente de Microservicios:**

La arquitectura de microservicios permite el despliegue independiente de cada servicio. Cuando se actualiza un microservicio, sus despliegues no afectan el funcionamiento de los otros microservicios. Esto se logra mediante la configuración de los microservicios para que se ejecuten de manera independiente en contenedores, y cada uno puede ser desplegado sin causar interrupciones en los demás.

2. **Despliegue sin Downtime:**

Se utilizan técnicas como rolling updates y blue/green deployment para asegurar que el despliegue de nuevas versiones de un microservicio no cause tiempos de inactividad. AWS Elastic Beanstalk, por ejemplo, permite realizar despliegues continuos con poco o ningún impacto en la disponibilidad de la aplicación.

3. **Docker y docker-compose:**

Todos los microservicios están configurados para ser ejecutados localmente usando Docker y docker-compose. Cada microservicio tiene su propio **Dockerfile**, y la orquestación de todos los contenedores se gestiona mediante un archivo **docker-compose.yml** que facilita el levantamiento de los microservicios en cualquier entorno. Esto asegura que cada microservicio se ejecute de manera coherente en entornos de desarrollo, prueba y producción.

4. **Instrucciones Claras para Despliegue:**

El **README.md** de cada repositorio incluirá las instrucciones detalladas para levantar el microservicio localmente con Docker, asegurando que cualquier miembro del equipo pueda replicar el entorno de producción sin complicaciones.

3. Cumplimiento con los 12 factores de una app

1. Codebase:

- Se utiliza un único repositorio que mantiene tanto el frontend como los microservicios. Esto facilita la gestión del código y asegura que ambas partes del sistema estén sincronizadas.

2. Dependencies:

- Las dependencias de cada microservicio están explícitamente declaradas en sus respectivos archivos package.json o requirement.txt.

3. Config:

- Las configuraciones de los microservicios se gestionan a través de variables de entorno, utilizando un archivo .env. Esto permite cambiar la configuración sin modificar el código, facilitando así la adaptabilidad de la aplicación a diferentes entornos (desarrollo, pruebas, producción).

4. Backing services:

- Se utilizan servicios externos, como bases de datos y Redis, configurados desde la infraestructura en AWS. Esto permite que los componentes de la aplicación sean independientes y que se puedan gestionar, escalar y actualizar sin afectar al resto del sistema.

5. Build, release, run:

- El proceso de construcción, liberación y ejecución de la aplicación está bien definido, permitiendo una clara separación entre estos pasos y facilitando un despliegue continuo.

6. Processes:

- Cada microservicio están diseñados para ser ejecutados en uno o más procesos stateless, lo que significa que no se mantiene estado en la memoria del servidor entre las solicitudes.

7. Port binding:

- Los microservicios exportan servicios a través de un puerto específico, permitiendo que se puedan ejecutar de forma independiente y que otros microservicios puedan interactuar entre ellos.

8. Concurrency:

- Cada microservicio puede escalar y manejar múltiples instancias de procesos concurrentes, lo que mejora la capacidad de respuesta y la disponibilidad del sistema.

9. Disposability:

- La aplicación está diseñada para iniciar y detenerse rápidamente, lo que permite un despliegue ágil y la capacidad de recuperarse de fallos.

10. Dev/prod parity:

- Se hace un esfuerzo consciente por mantener la paridad entre los entornos de desarrollo y producción, asegurando que los cambios se prueben en un entorno lo más similar posible al de producción.

11. Logs:

- Los microservicios generan logs de salida estándar, lo que facilita la supervisión y el análisis de la actividad del sistema.

12. Admin processes:

- Los procesos administrativos, como migraciones de bases de datos y scripts de mantenimiento, se pueden ejecutar de manera ad-hoc en el entorno de producción.

4. Descripción del proceso de Deployment

Plataforma

Para el despliegue de la aplicación, se utiliza Amazon Web Services (AWS), aprovechando varios de sus servicios clave:

- Elastic Beanstalk: Facilita la implementación y escalado de aplicaciones web. Permite gestionar automáticamente la infraestructura subyacente, lo que permite a los desarrolladores centrarse en escribir código en lugar de preocuparse por la configuración del servidor.
- RDS (Relational Database Service): Utilizado para la gestión de bases de datos relacionales. AWS RDS simplifica la configuración, operación y escalado de bases de datos en la nube, asegurando alta disponibilidad y respaldo automático.
- ElastiCache: Utilizado para mejorar el rendimiento de la aplicación mediante el almacenamiento en caché de datos en memoria. Esto ayuda a reducir la latencia y a mejorar la respuesta de la aplicación, especialmente para datos que se consultan con frecuencia.

La elección de AWS se justifica por su escalabilidad y confiabilidad, permitiendo manejar picos de tráfico y ofreciendo opciones robustas para asegurar la disponibilidad y el rendimiento de la aplicación.

Pasos para el Despliegue

1. Creación de Instancias:
 - Inicia sesión en tu consola de AWS.
 - Navega a Elastic Beanstalk y crea una aplicación para cada microservicio.
 - Selecciona la plataforma deseada (Docker, en este caso) y configura el entorno.
2. Carga del Código de cada microservicio:
 - En tu entorno de Elastic Beanstalk, selecciona la opción para cargar el código.
 - Puedes subir tu aplicación directamente como un archivo ZIP o conectarte a un repositorio de código fuente como GitHub.
 - Elastic Beanstalk se encargará de compilar el código y lanzar la aplicación en el entorno configurado.
3. Configuración de Variables de Entorno de cada microservicio:
 - En la consola de Elastic Beanstalk, ve a la configuración de tu entorno y agrega las variables de entorno necesarias (por ejemplo, credenciales de base de datos).
4. Configuración de RDS de cada microservicio:
 - Ve a RDS y crea una nueva base de datos por microservicio.
 - Elige el motor de base de datos (MySQL).
 - Configura la instancia, el tamaño, las credenciales de acceso y las opciones de red.
 - Asegúrate de habilitar la conectividad entre tu instancia de Elastic Beanstalk y RDS.

5. Configuración de ElastiCache:

- Ve a ElastiCache y crea un clúster de Redis.
- Configura los parámetros necesarios, como el tamaño del nodo y las opciones de seguridad.
- Asegúrate de que tu aplicación pueda conectarse al clúster de Redis utilizando el endpoint proporcionado.

Proceso de Producción

● **Despliegue Manual:**

Actualmente, el proceso de despliegue es manual. Cada vez que se necesita actualizar la aplicación, se sigue el proceso mencionado anteriormente para cargar la nueva versión del código de cada microservicio en Elastic Beanstalk.

Esto implica:

- Crear un archivo ZIP con la nueva versión del código de cada microservicio .
- Cargarlo a través de la consola de Elastic Beanstalk.
- Configurar las variables de entorno necesarias y realizar las pruebas pertinentes.

● **Migraciones:**

Las migraciones de las bases de datos se gestionan a través de scripts que se ejecutan manualmente cuando se despliegan las nuevas versiones, asegurando que las bases de datos estén siempre actualizadas con la última estructura requerida por la aplicación.

● **Downtime:**

El proceso de despliegue manual puede generar un breve periodo de downtime, ya que la aplicación puede no estar disponible durante el tiempo que tarda en desplegarse y reiniciarse. Se recomienda coordinar los despliegues en horarios de menor tráfico para minimizar el impacto en los usuarios.