

Universidad ORT Uruguay

Facultad de ingeniería

Arquitectura de software en la práctica Obligatorio I

Mauro Teixeira - 211753
Giovanni Ghisellini - 205650
Rodrigo Hirigoyen - 197396

Backend:

<https://github.com/ArqSoftPractica/211753-205650-197396-Backend>

Frontend:

<https://github.com/ArqSoftPractica/211753-205650-197396-Frontend>

Demo:

https://youtu.be/_dVdvCXXbTE

URI (Acceso web público):

<https://gestion-inventario-ort.herokuapp.com/login>

Docentes

Guillermo Areosa
Nicolas Fornaro

Mayo 2023

Índice

1. Introducción	2
2. Antecedentes	2
2.2 Requerimientos significativos de Arquitectura	2
2.2.1 Requerimientos Funcionales:	2
2.2.2 Requerimientos no Funcionales:	3
3. Descripción de Arquitectura	5
3.1 Vistas de módulos	5
3.1.1 Vista de capas	6
3.1.2 Vista de usos	8
3.2 Vista de componentes y conectores	8
3.2.1 Representación primaria	9
3.2.2 Catálogo de elementos	9
3.2.3 Descripción de controladores expuestos en la API REST	10
3.3 Vista de despliegue	11
3.3.2 Catálogo de elementos	12
4. Decisiones generales de diseño	13
4.1 Logs	13
4.2 Exceptions y manejo de errores	13
4.3 Base de datos	14
4.4 Variables de configuración	14
4.5 Cache	14
4.6 Requerimientos no Funcionales	14
5. Frontend	26
6. Cumplimiento con 12 factor app	28
7. Descripción del proceso de deployment	29

1.Introducción

El objetivo del documento es que sea una guía para lograr entender la arquitectura y funcionamiento del sistema GestionInventario. Se documentará y explicará las decisiones de diseño tomadas para realizar el sistema como sus principales vistas, requerimientos funcionales y no funcionales.

2.Antecedentes

2.2 Requerimientos significativos de Arquitectura

2.2.1 Requerimientos Funcionales:

ID Requerimiento	Descripción	Actor
RF1. Registro de usuario administrador	Registro de administrador	Usuario administrador
RF2. Registro de usuarios mediante invitación	Registro de administrador y empleado mediante invitación	Usuario administrador y usuario empleado
RF3. Autenticación de usuario	Login de la aplicación	Usuario empleado y administrador
RF4. Gestión de productos	Alta, modificación y baja de producto	Usuario administrador
RF5. Gestión de proveedores	Alta, modificación y baja de proveedores	Usuario administrador
RF6. Registro de compras	Registro de compra	Usuario administrador
RF7. Registro de ventas	Registro de venta	Usuario empleado y administrador
RF8 Gestión de inventario	Visualización de inventario de la empresa	Usuario empleado y administrador
RF9. Página de inicio de la empresa	Ver ventas registradas para una determinada fecha	Usuario empleado y administrador
RF 10. Top 3 de productos con más ventas (REST).	Ver 3 top productos mas vendidos dentro de la empresa	Usuario empleado y administrador
RF 11. Consulta de compras para un proveedor (REST).	Consulta de las compras de de un proveedor	Usuario empleado y administrador

2.2.2 Requerimientos no Funcionales:

ID Requerimiento	Descripción
RNF 1. Performance	Su equipo ha acordado que los endpoints públicos deben responder rápidamente para no perjudicar la performance de las aplicaciones que lo consumen. Los endpoints de los requerimientos funcionales 10 y 11 deben responder con una máxima de 300 ms para cargas de hasta 1200 req/m (p95).
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.).
RNF3. Observabilidad	Para facilitar el diagnóstico ante eventuales fallas, usted y su equipo deberán poder monitorear al menos las siguientes métricas del sistema: <ul style="list-style-type: none">• Peticiones por minuto• Tiempos de respuesta de distintos endpoints
RNF 4. Autenticación, autorización y tenancy	Se deberá establecer un control de acceso basado en roles, distinguiendo entre usuarios administradores y usuarios empleados. Los permisos otorgados a los mismos deberán restringir el acceso a sus correspondientes funcionalidades, prohibiendo la interacción con cualquier otra operación pública del sistema.
RNF 5. Seguridad	Toda comunicación entre clientes front end y componentes de back end deberán utilizar un protocolo de transporte seguro o describir qué sería necesario para configurarlo si por razones de la plataforma no fue posible para el equipo. Por otra parte, la comunicación entre componentes de back end deberá realizarse dentro de una red de alcance privado. De lo contrario, deberán utilizar un protocolo de transporte seguro autenticado mediante el mecanismo que consideren necesario.
RNF 6. Código fuente	El control de versiones del código se deberá llevar a cabo con repositorios Git

	<p>debidamente documentados, que contengan en el archivo README.md una descripción con el propósito y alcance del proyecto, así como instrucciones para configurar un nuevo ambiente de desarrollo</p>
RNF 7. Integración continua	<p>Se deberá contar con pruebas unitarias para al menos dos requerimientos funcionales de su elección. Estos dos requerimientos deben tener una cobertura de 100%. Las pruebas unitarias se deben correr automáticamente cada vez que un nuevo commit se integra a la rama principal.</p>
RNF 8. Pruebas de carga	<p>Se deberá diseñar y ejecutar un script de generación de planes de prueba de carga utilizando la herramienta Apache jMeter, con el objetivo de demostrar el cumplimiento con los requerimientos de performance.</p>
RNF 9. Identificación de fallas	<p>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.</p>
RNF 10. Portabilidad	<p>El repositorio debe contar con los archivos necesarios para que cualquier persona que se clone el repositorio y tenga Docker corriendo, pueda levantar el sistema de manera local con solo un comando.</p>

3.Descripción de Arquitectura

En esta sección se detallan algunas vistas para explicar la arquitectura del Backend y Frontend. Entre ellas se encuentran: módulos, componentes y conectores y asignaciones para el backend y para el frontend vista de módulos.

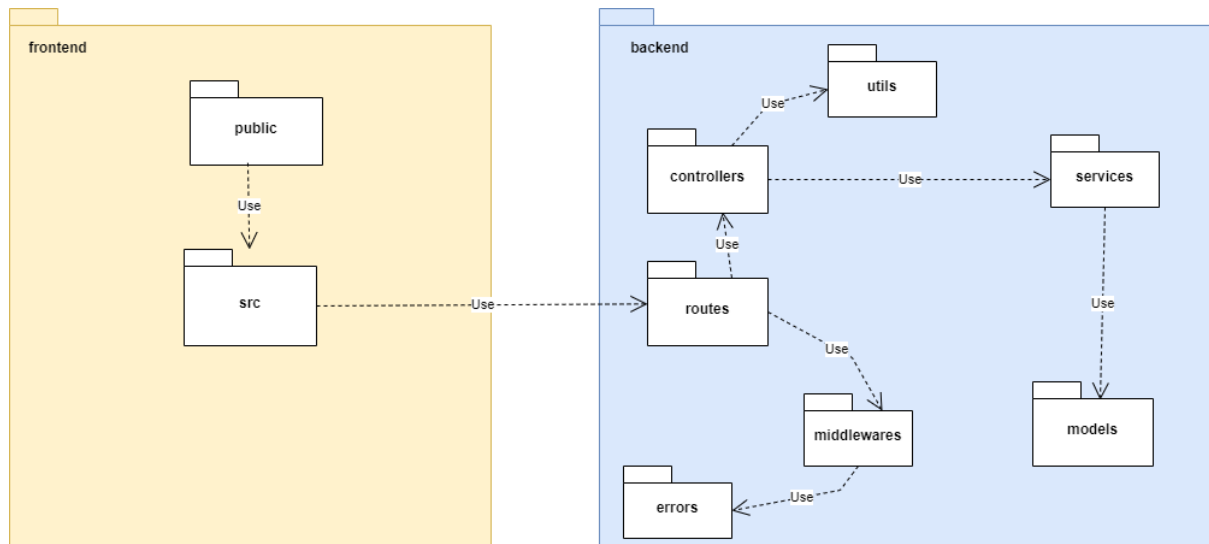
Backend

Comenzamos con una descripción general del sistema, en la que se presentarán todos los módulos que lo conforman, junto con sus respectivas relaciones y su vínculo con el Frontend. De esta manera, pretendemos ofrecer una visión panorámica de nuestro sistema, con el objetivo de que se familiarice con la solución que hemos propuesto para abordar el problema planteado.

3.1 Vistas de módulos

Representación general

Nuestro sistema se divide en dos partes principales: en primer lugar, un servidor que alberga la lógica del sistema (backend), y en segundo lugar, un servidor web desarrollado en angular (frontend), que permite a los usuarios interactuar con nuestro sistema de manera visual. De esta manera, el usuario puede acceder a todas las funcionalidades de nuestro sistema a través del frontend, mientras que el backend se encarga de procesar y gestionar los datos para asegurar un correcto funcionamiento del sistema.

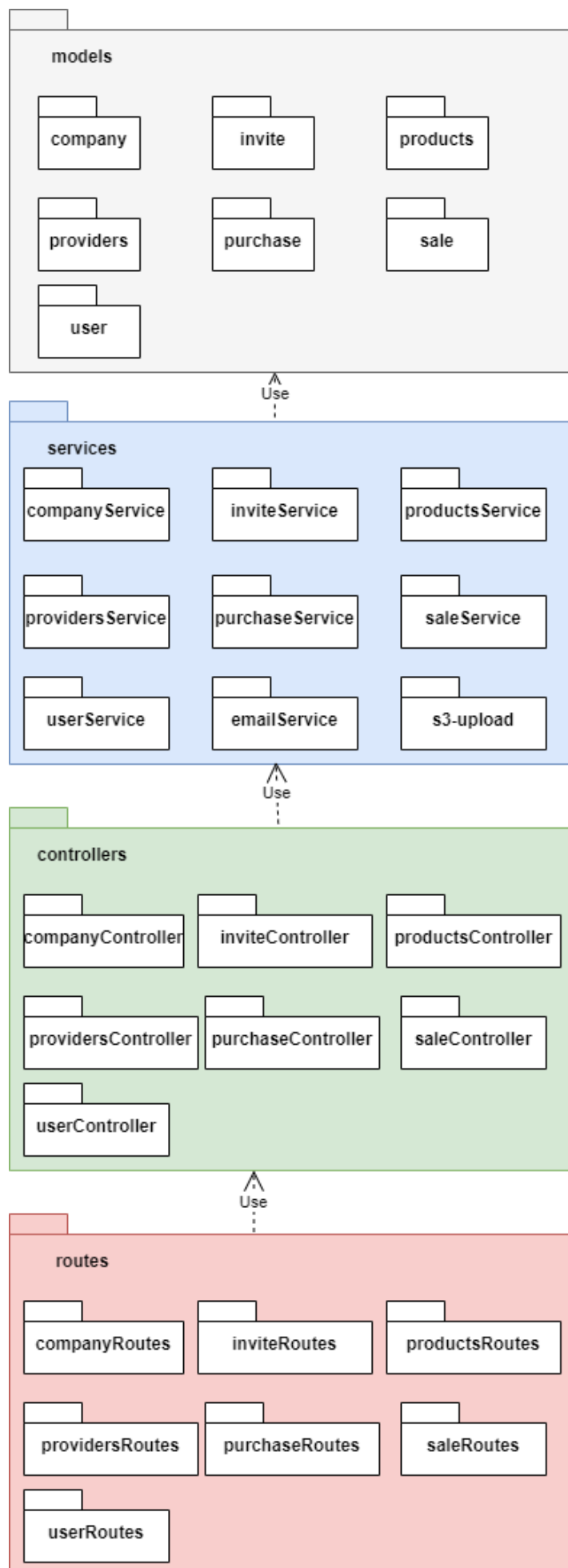


3.1.1 Vista de capas

Como primera presentación de nuestro sistema, mostraremos la arquitectura de nuestro backend, que como se puede apreciar en el diagrama antes presentado, utilizamos una solución orientada a capas, favoreciendo el patrón de diseño Layered, agrupando la solución en módulos cohesivos.

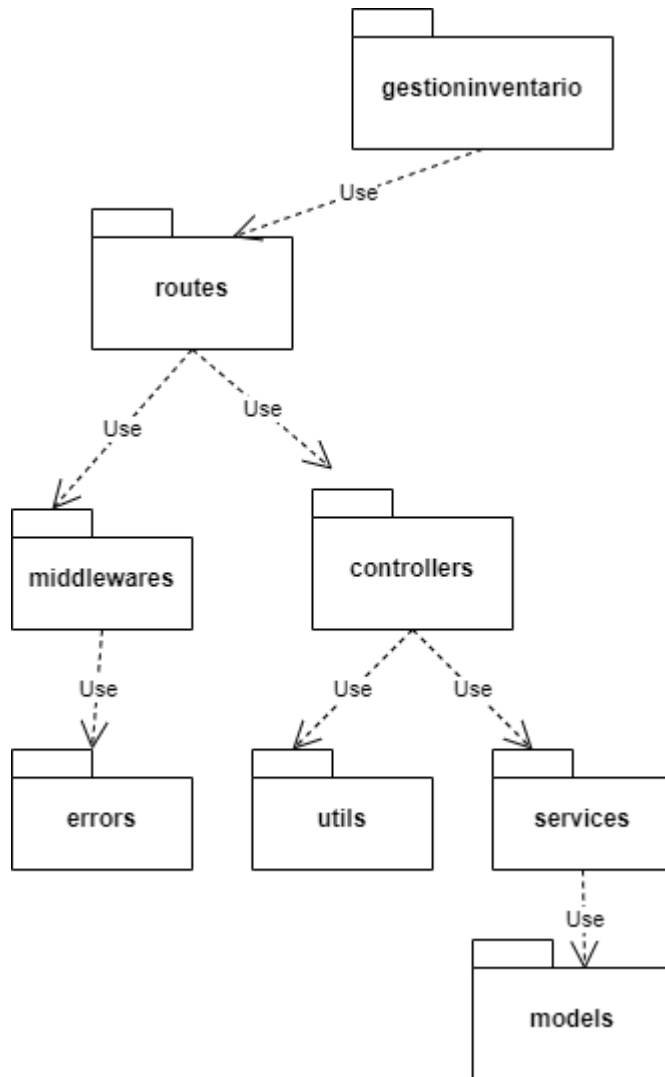
El patrón Layered proporciona una serie de beneficios clave en el diseño de sistemas, incluyendo la abstracción, el encapsulamiento y la definición clara de capas de funcionalidad. Esto conduce a una alta cohesión, reutilización de componentes y un acoplamiento débil, lo que a su vez reduce los riesgos y minimiza el impacto de los cambios en las tecnologías o soluciones. Estas características son altamente valoradas, ya que nos permite contar con un sistema robusto y adaptable que puede evolucionar a medida que cambian las necesidades del negocio.

Para ilustrar mejor cómo funcionan los distintos módulos de nuestro sistema, nos gustaría presentar una vista de usos que muestra cómo se relacionan entre sí. Esta vista proporciona una imagen clara y detallada de cómo los distintos componentes del sistema interactúan, lo que es esencial para garantizar la coherencia y la eficiencia en el funcionamiento del sistema. Al presentar esta vista de usos, esperamos que tengan una mejor comprensión de cómo funciona nuestro sistema



3.1.2 Vista de usos

Para que el lector pueda comprender mejor las decisiones tomadas y visualizar la dependencia de uso de los paquetes en nuestro sistema, presentamos el diagrama más relevante de uso del mismo. De esta manera, resultará más sencillo entender cómo se relacionan los distintos paquetes y cómo interactúan entre sí.

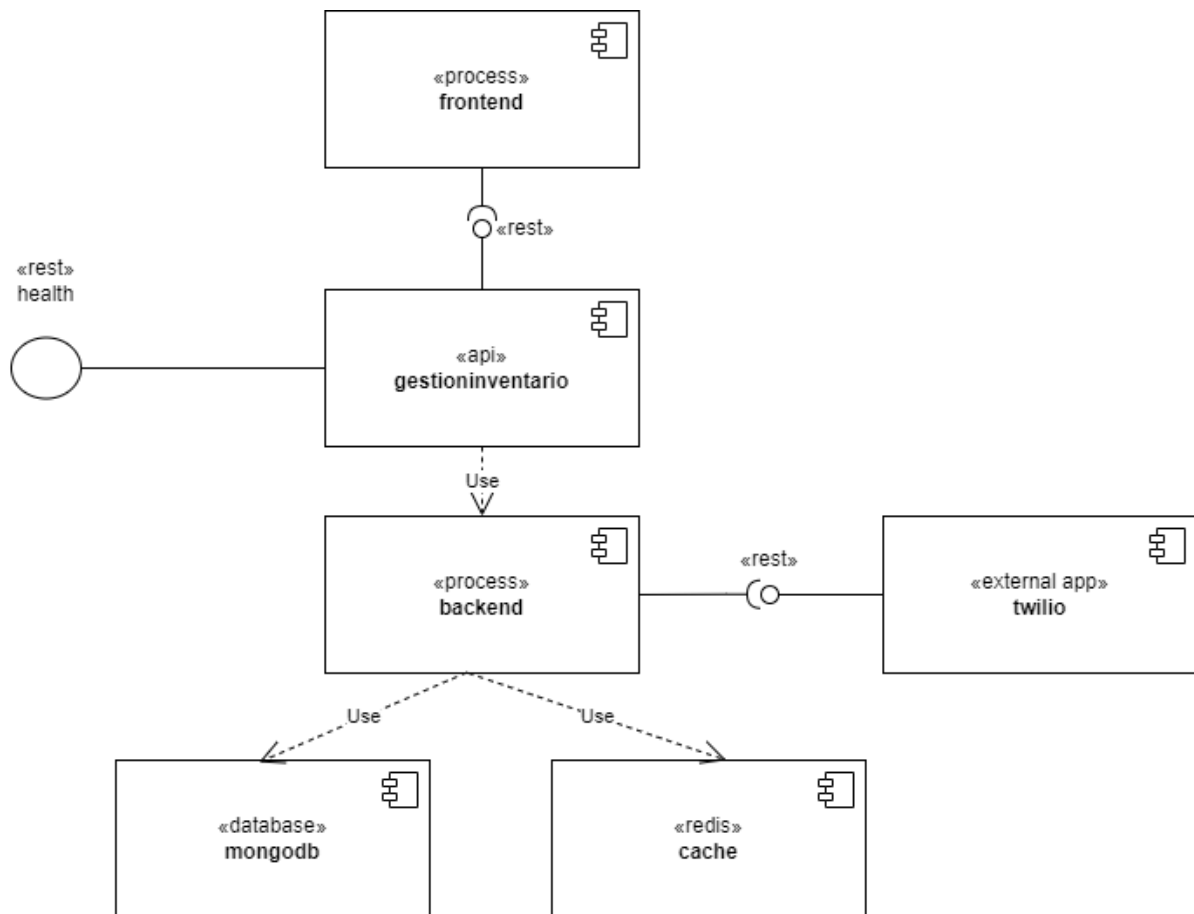


3.2 Vista de componentes y conectores

La vista que presentamos tiene como objetivo describir el sistema a partir de la especificación de elementos que presentan características específicas de ejecución, tales como procesos, clientes, servidores, entre otros.

El propósito de esta vista es describir un conjunto de funcionalidades que pueden ser invocadas en tiempo de ejecución por otros elementos del sistema. Los conectores, por su parte, son los responsables de transmitir tanto la petición desde el solicitante hacia el proveedor. En resumen, esta vista nos ayuda a entender cómo se relacionan los distintos elementos del sistema y cómo se comunican entre sí.

3.2.1 Representación primaria



3.2.2 Catálogo de elementos

Componente/Conector	Tipo	Descripción
Frontend	App	Componente donde se renderiza la UI para el cliente, es el encargado de solicitar la comunicación con el backend de forma apropiada para el mismo
GestionInventario	Api	Componente el cual expone servicios para consumirlos desde el frontend siempre que se cuente con un token válido, como también expone otros servicio públicos sin necesidad de token.
Backend	App	Procesa las peticiones provenientes del API, y

		evalúa la correctitud. Para aquellas que sí cumplan, delega los pedidos a la base de datos. Para las que no satisfagan las reglas de negocio, se informa al cliente de la situación a través del API.
MongoDB	Database	Es el repositorio de datos de la aplicación, aloja la información de todo el sistema. Aprovechamos su naturaleza relacional haciendo uso de llaves foráneas que facilitan la navegación entre entidades/tablas.
Cache(Redis)	Cache	Se utiliza para almacenar los datos de los del REQ 10 y 11.
TwilioAPI	Api	Se utilizó este servicio externo para el envío de correos a los nuevos usuarios que se desean invitar tanto usuario como administrador.
<<rest>>	Conector	Notación que empleamos para denotar la comunicación vía HTTP o HTTPS a una api rest expuesta.

3.2.3 Descripción de controladores expuestos en la API REST

Para hacer más fácil la comprensión del diagrama anterior, se eliminó información sobre las funciones que cada interfaz realiza. Sin embargo, se proporciona una tabla detallada que describe las posibilidades que se manejan en cada interfaz, con el objetivo de brindar una mejor comprensión al lector.

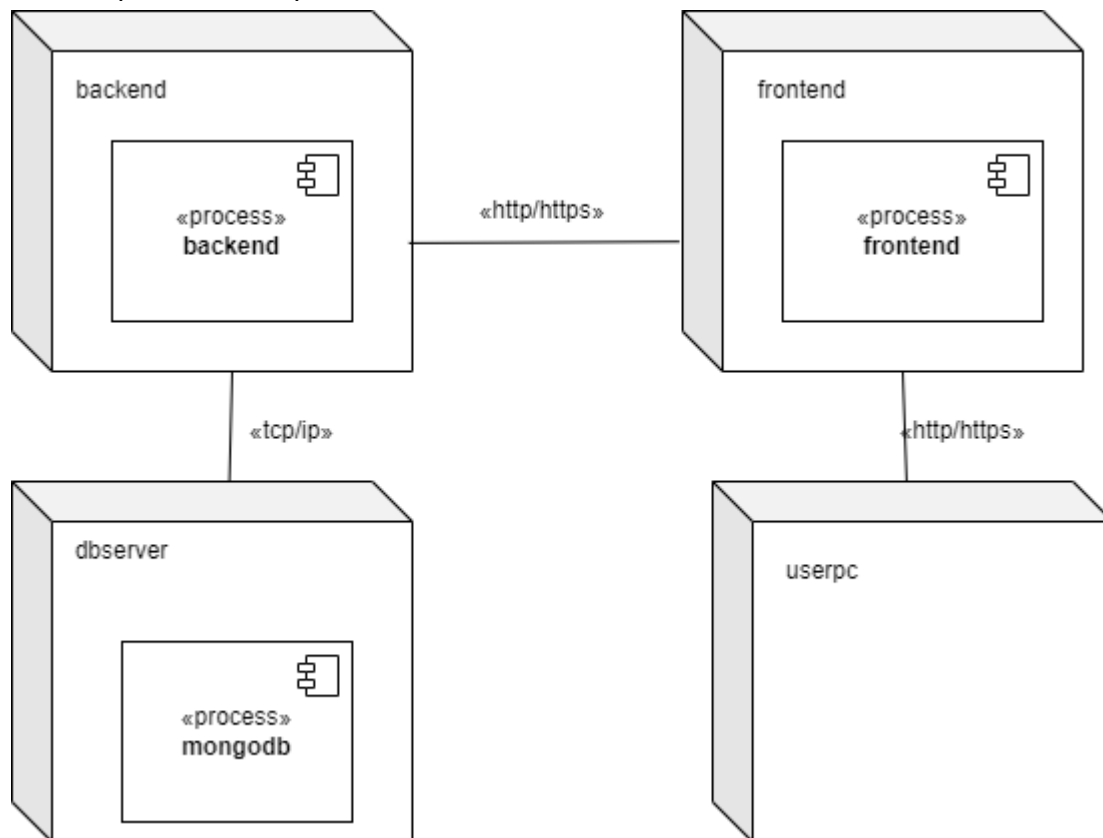
Componente GestionInventario	expuesto por	Responsabilidad
ProductController		Permite realizar el CRUD de productos.
ProviderController		Permite realizar el CRUD de proveedores.
PurchaseController		Permite la creación de compras y obtener las compras para un proveedor.

SaleController	Permite la creación de una venta y obtener las ventas.
UserController	Permite la autenticación del usuario en el sistema y obtener un token de acceso.
InviteController	Permite la creación de usuarios (administrador o empleado) vía email.
HealthController	Se expone un endpoint específico para observar el funcionamiento del sistema.

3.3 Vista de despliegue

El uso del patrón Multi-Tier ha llevado a que el software se divida en tres capas distintas de componentes. La de presentación web, que es el frontend de nuestra solución, donde se encuentra la interfaz de usuario que los usuarios finales interactúan. La otra capa es la de aplicación, que contiene el back-end de la solución, donde se procesan las solicitudes del usuario y se realizan las operaciones de lógica de negocios. Finalmente, última capa es la de base de datos, que compone el nivel más bajo de la solución, y es responsable de la gestión y almacenamiento de los datos utilizados por la aplicación

3.2.1 Representación primaria



3.3.2 Catálogo de elementos

Nodo	Descripción
frontend	Nodo dedicado a renderizar componentes hacia el usuario.
backend	Nodo el cual contiene toda la lógica del negocio.
mongodb	Nodo el cual representa el almacenamiento de datos.
userpc	Nodo el cual representa al cliente, el mismo interactúa con nuestro sistema por medio del navegador web.

Conector	Descripción
HTTP/HTTPS	Este conector es utilizado para poder comunicar al usuario a nuestro sistema, y nuestro frontend con nuestro backend.
TCP/IP	Este conector es mediante el cual se comunica nuestra app con el servicio de base de datos.

4. Decisiones generales de diseño

Como se mencionó anteriormente se decidió utilizar el patrón Layers, este promueve la separación de responsabilidades y la modularidad en el código. Al dividir el sistema en capas o niveles, cada capa se encarga de una funcionalidad específica y solo se comunica con la capa adyacente. Esto ayuda a minimizar la dependencia entre las diferentes partes del sistema, lo que a su vez facilita la escalabilidad, el mantenimiento, la evolución del software a largo plazo, y que la misma se adapte no solo a nuevas tecnologías, sino a nuevos problemas que la industria requiera.

Además, al utilizar el patrón Layers, es más fácil de entender y modificar el código, ya que cada capa tiene una responsabilidad clara y definida. Esto ayuda a simplificar el proceso de depuración y mejora la calidad del software.

4.1 Logs

Utilizamos Winston para nuestros logs para registrar e informar los eventos de nuestro sistema, estos son útiles para determinar la causa de los problemas. La decisión en elegir Winston es que él mismo nos ofrece más funciones, a continuación detallaremos algunas de ellas:

- **Secuencias:** Es una forma eficaz de fragmentar el procesamiento de datos. Las secuencias dividen las tareas en partes pequeñas y manejables.

- **Transportes:** De forma predeterminada, `console.log` va a su archivo de salida estándar. Winston permite transmitir sus registros a una ubicación central o almacenarlos en una base de datos para consultarlos más adelante. Lo bueno de su utilización es que se pueden tener tantos transportes como se deseen.

Se procuró cumplir con un determinado proceso de logs a lo largo de nuestro sistema, el cual consiste en tres niveles de información de auditoría. Estas tres categorías están divididas en el código por tres métodos (`logger.debug`, `logger.error` y `logger.info`). El log de errores lo utilizamos justamente para dar información sobre errores, excepciones en las que pueda caer el código, `logger.info` lo utilizamos para dar información sobre acciones completadas con éxito, mientras que `logger.debug` lo usamos para dar información sobre valores intermedios o tareas que no han sido terminadas todavía.

4.2 Exceptions y manejo de errores

Para lo que es el manejo de errores, utilizamos bloques `try-catch` a lo largo de toda la aplicación con el objetivo de poder cachear los errores en el caso de que ocurran. Además, en cualquiera de los casos (éxito o error) estamos loggeando, como se mencionó anteriormente, concretamente cuál fue el resultado o el error obtenido.

Por otro lado, también utilizamos los códigos de respuesta de HTTP acordes a cada estado:

- Utilizamos el código de respuesta 200 para cuando devolvemos datos, el código 204 cuando la respuesta es correcta pero sin datos y 201 creado correctamente.
- El código 400 para Bad Request, 422 para cuando los datos son correctos pero fallan las validaciones, y 404 cuando no encontramos recursos.

- Finalmente, el código 500 para los errores del servidor.

4.3 Base de datos

Decidimos utilizar una base de datos para cumplir de la mejor forma posible los requerimientos solicitados por el cliente.

Utilizamos una base de datos no relacional MongoDB, esta decisión se basó en que esta nos permite almacenar grandes volúmenes de información de forma rápida. A su vez la indexación, se puede indexar los campos como primarios o secundarios esto ayuda a mejorar el rendimiento de la búsqueda, aumentando su velocidad. Además este tipo de base de datos es altamente escalable, por lo que realizar cambios en su diseño sería bastante fácil de implementar

También para la persistencia de imágenes se utilizó S3 bucket de AWS, es un contenedor de objetos, similar a una carpeta o directorio, en el que se pueden almacenar y recuperar archivos o datos de forma segura y eficiente. Cada objeto en un S3 bucket tiene una clave única que lo identifica, y se pueden configurar políticas de acceso para controlar quién puede acceder y realizar operaciones en los objetos. Luego para relacionarlo con nuestra base de datos en MongoDB guardamos el ID de la imagen de AWS en la colección de producto en MongoDB.

4.4 Variables de configuración

Utilizamos archivos .env permite personalizar las variables de entorno de trabajo individuales y Docker los carga como variables de entorno. Esto nos permite potenciar la modificabilidad del sistema.

4.5 Cache

Utilizamos caché para los endpoints de consultas solicitados en los REQ 10 y 11 ya que es estos deben responder en un tiempo acordado, por lo cual para mejorar considerablemente la performance evitando acceder a la base de datos cada vez que se necesiten los candidatos y así poder aumentar la velocidad de la request, este fue implementado con Redis. La decisión se basa en favorecer el atributo de calidad de performance. Esto permite reducir significativamente el tiempo de procesamiento, ya que el acceso a un dato de caché en Redis es más rápido que acceder a los mismos datos en MongoDB.

4.6 Requerimientos no Funcionales

Durante este documento, se mencionaron varios atributos de calidad que se consideraron importantes para garantizar un código de alta calidad. Sin embargo, no se profundizó en estos atributos en detalle en todo momento. En esta sección, se volverá a abordar algunos de estos atributos para entenderlos mejor y comprender por qué se consideraron importantes al tomar decisiones relevantes en el desarrollo de la aplicación.

- **RNF. 1 - Performance**

Se ha acordado que los endpoints públicos deben responder rápidamente para no perjudicar la performance de las aplicaciones que lo consumen. Los endpoints de los requerimientos funcionales 10 y 11 deben responder con una máxima de 300 ms para cargas de hasta 1200 req/m (p95). Para lograr esta solicitud se aplicaron tácticas que van desde la replicación de datos, usando cache (REDIS). Al desarrollar el backend de la aplicación en Node.js, uno de los aspectos destacados es su capacidad para manejar la asincronía de manera eficiente. Este utiliza un modelo de programación asíncronico y basado en eventos que permite que varias operaciones se realicen de manera concurrente sin bloquear el hilo de ejecución principal.

En lugar de esperar a que una operación finalice antes de pasar a la siguiente, Node.js puede continuar ejecutando otras operaciones mientras espera que una operación se complete. Esto mejora la eficiencia y la escalabilidad de la aplicación, lo que es especialmente útil en aplicaciones de alta carga o en tiempo real.

Además, utiliza devoluciones de llamada (callbacks) y promesas para manejar la asincronía, lo que facilita la escritura de código asíncronico y reduce la complejidad.

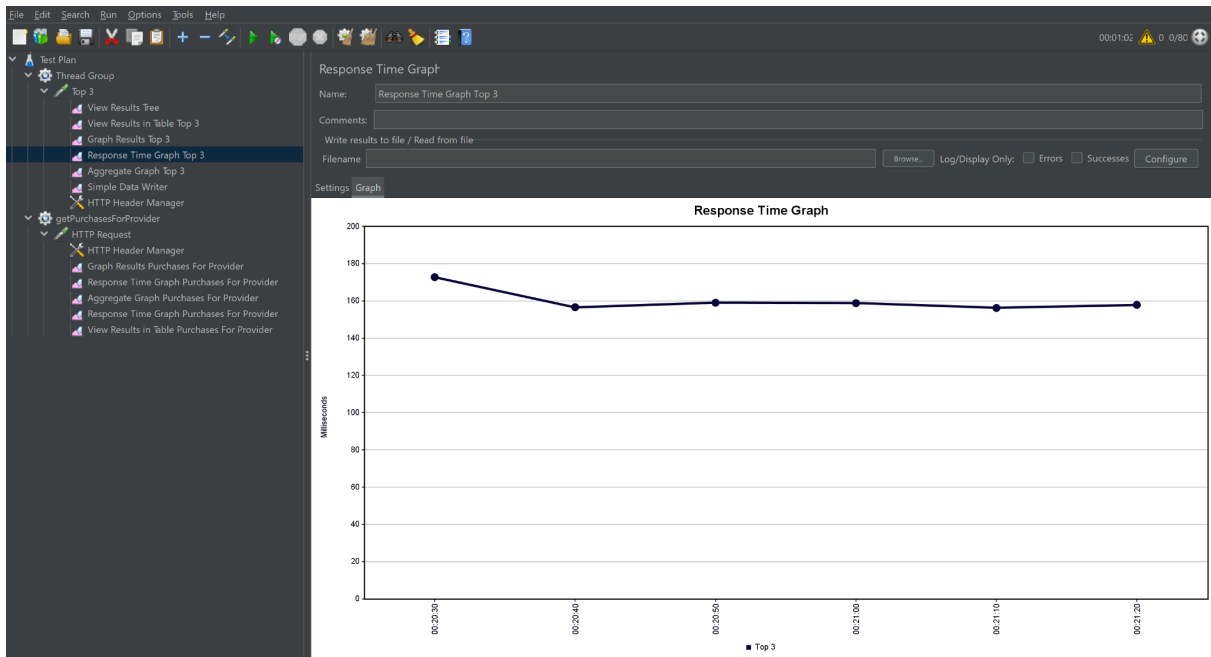
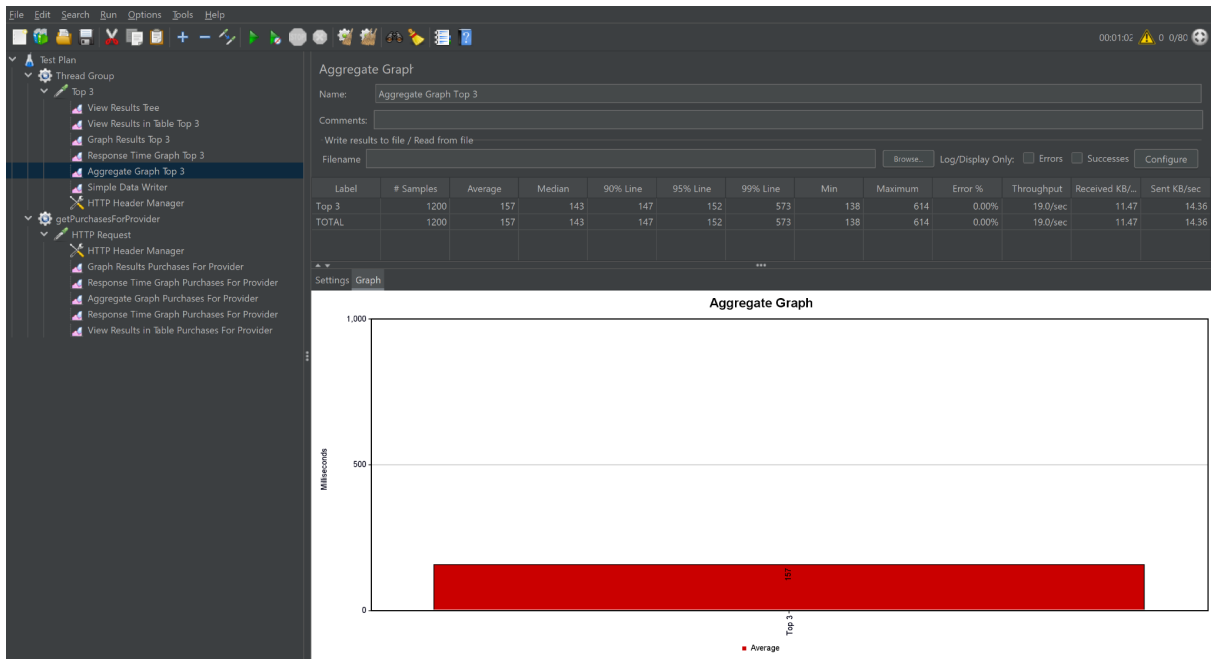
Para visualizar la performance de los endpoints mencionados anteriormente se utilizó la herramienta Apache JMeter, y así observar los tiempos de respuesta obtenidos.

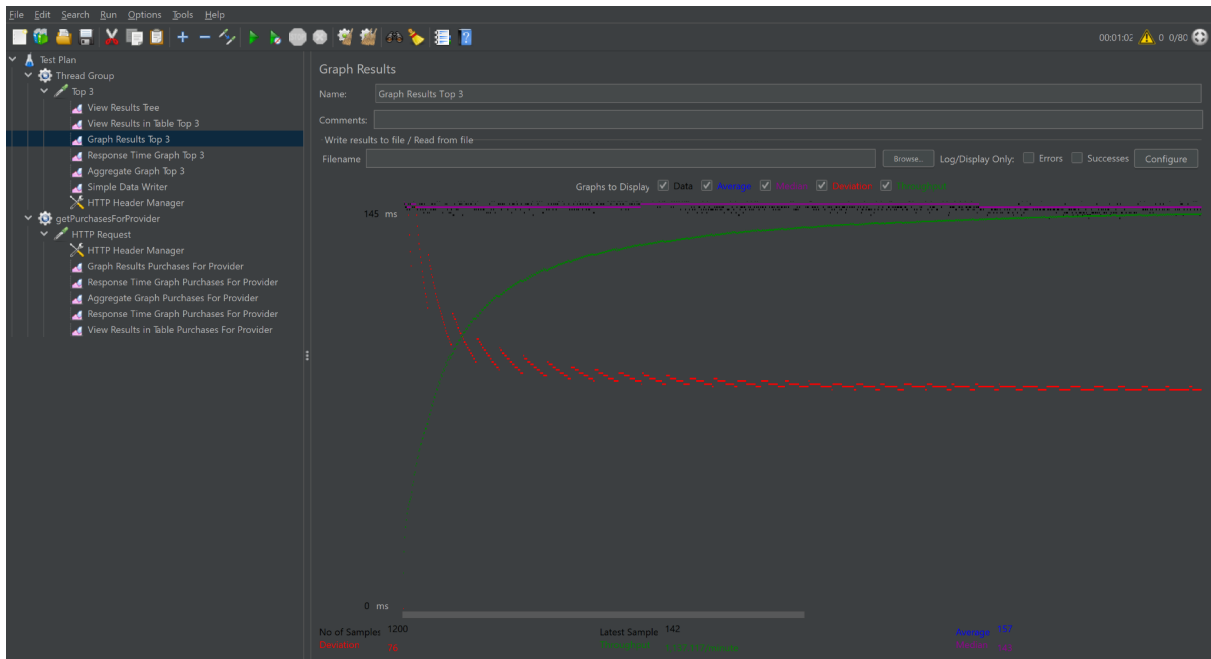
Estas pruebas fueron ejecutadas de forma local. A continuación vamos a mostrar los resultados con la aplicación deployada, cumpliendo los requerimientos solicitados:

- Aplicación deployada en Heroku

1. Top 3 de productos con más ventas

A continuación mostramos la gráfica de aggregate,response, los resultados y los datos utilizados para las pruebas de top 3 de productos con más ventas, como se puede ver en las gráficas se cumple con lo requerido con una carga de 1200 req/m el tiempo de respuesta medio es de 157 que es mucho menor que 300 que el máximo estipulado en el requerimiento. Para finalizar en la última gráfica se puede ver los resultados de todas las pruebas realizadas.





File Edit Search Run Options Tools Help

0001:02 0 0/80

Test Plan

- Thread Group
 - Top 3
 - View Results Tree
 - View Results in Table Top 3
 - Graph Results Top 3
 - Response Time Graph Top 3
 - Aggregate Graph Top 3
 - Simple Data Writer
 - HTTP Header Manager
 - getPurchasesForProvider
 - HTTP Request
 - HTTP Header Manager
 - Graph Results Purchases For Provider
 - Response Time Graph Purchases For Provider
 - Aggregate Graph Purchases For Provider
 - Response Time Graph Purchases For Provider
 - View Results in Table Purchases For Provider

View Results in Table

Name: View Results in Table Top 3

Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes ☐ Configure

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	00:20:35.403	Thread Group 1-1	Top 3	569	✓	620	776	569	424
2	00:20:35.972	Thread Group 1-1	Top 3	145	✓	620	776	145	0
3	00:20:36.117	Thread Group 1-1	Top 3	144	✓	620	776	144	0
4	00:20:36.262	Thread Group 1-1	Top 3	143	✓	620	776	143	0
5	00:20:36.405	Thread Group 1-1	Top 3	143	✓	620	776	143	0
6	00:20:36.549	Thread Group 1-1	Top 3	142	✓	620	776	142	0
7	00:20:36.691	Thread Group 1-1	Top 3	143	✓	620	776	143	0
8	00:20:36.835	Thread Group 1-1	Top 3	144	✓	620	776	143	0
9	00:20:36.979	Thread Group 1-1	Top 3	142	✓	620	776	142	0
10	00:20:37.121	Thread Group 1-1	Top 3	144	✓	620	776	144	0
11	00:20:37.265	Thread Group 1-1	Top 3	144	✓	620	776	144	0
12	00:20:36.903	Thread Group 1-2	Top 3	608	✓	620	776	608	465
13	00:20:37.409	Thread Group 1-1	Top 3	143	✓	620	776	143	0
14	00:20:37.512	Thread Group 1-2	Top 3	145	✓	620	776	145	0
15	00:20:37.552	Thread Group 1-1	Top 3	143	✓	620	776	143	0
16	00:20:37.658	Thread Group 1-2	Top 3	144	✓	620	776	144	0
17	00:20:37.695	Thread Group 1-1	Top 3	143	✓	620	776	143	0
18	00:20:37.802	Thread Group 1-2	Top 3	143	✓	620	776	143	0
19	00:20:37.838	Thread Group 1-1	Top 3	142	✓	620	776	142	0
20	00:20:37.945	Thread Group 1-2	Top 3	143	✓	620	776	143	0
21	00:20:37.980	Thread Group 1-1	Top 3	143	✓	620	776	143	0
22	00:20:38.088	Thread Group 1-2	Top 3	144	✓	620	776	144	0
23	00:20:38.124	Thread Group 1-1	Top 3	142	✓	620	776	142	0
24	00:20:38.232	Thread Group 1-2	Top 3	144	✓	620	776	144	0
25	00:20:38.266	Thread Group 1-1	Top 3	144	✓	620	776	144	0
26	00:20:38.376	Thread Group 1-2	Top 3	142	✓	620	776	142	0

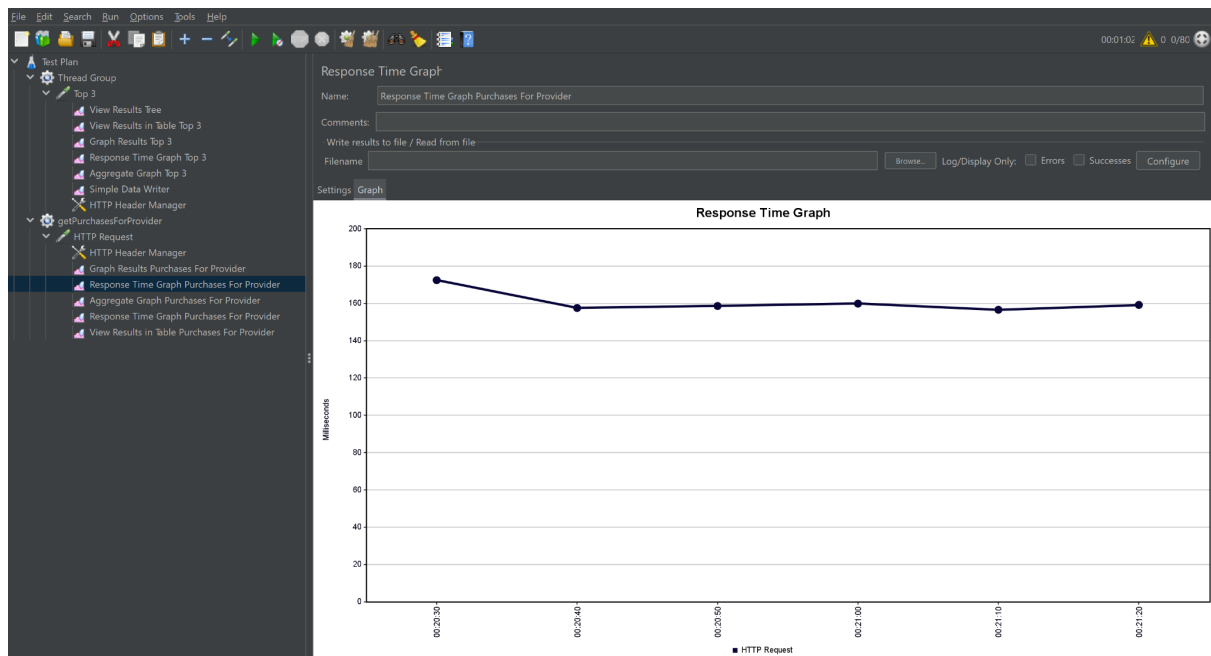
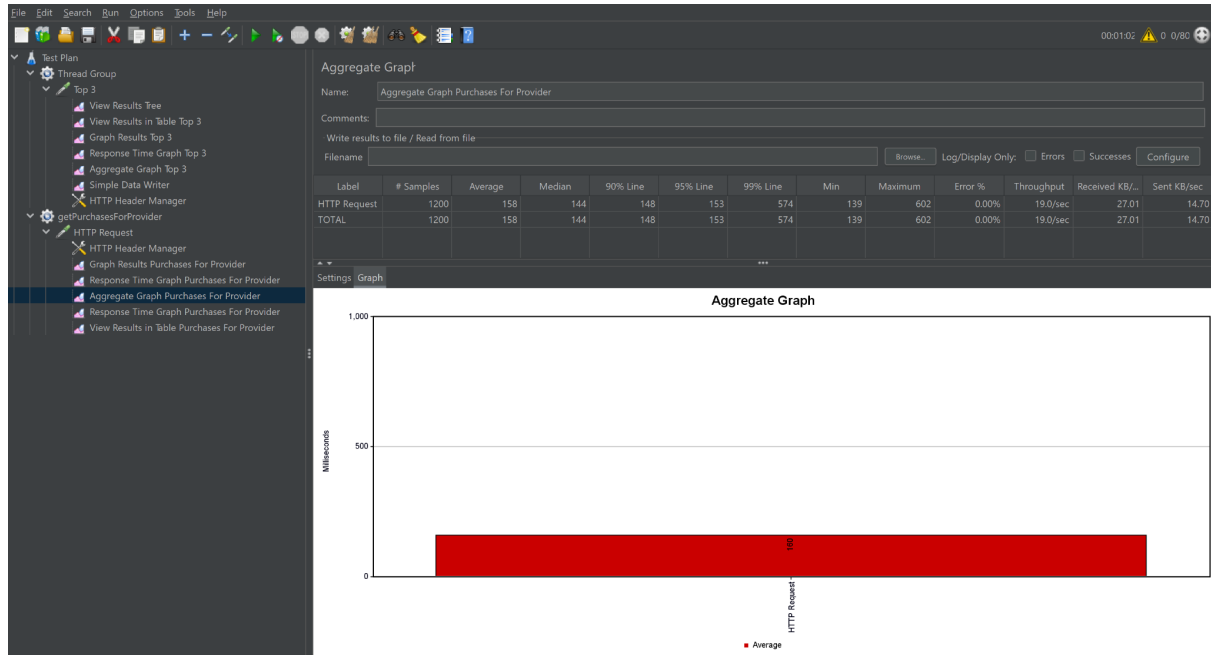
☐ Scroll automatically? ☐ Child samples?

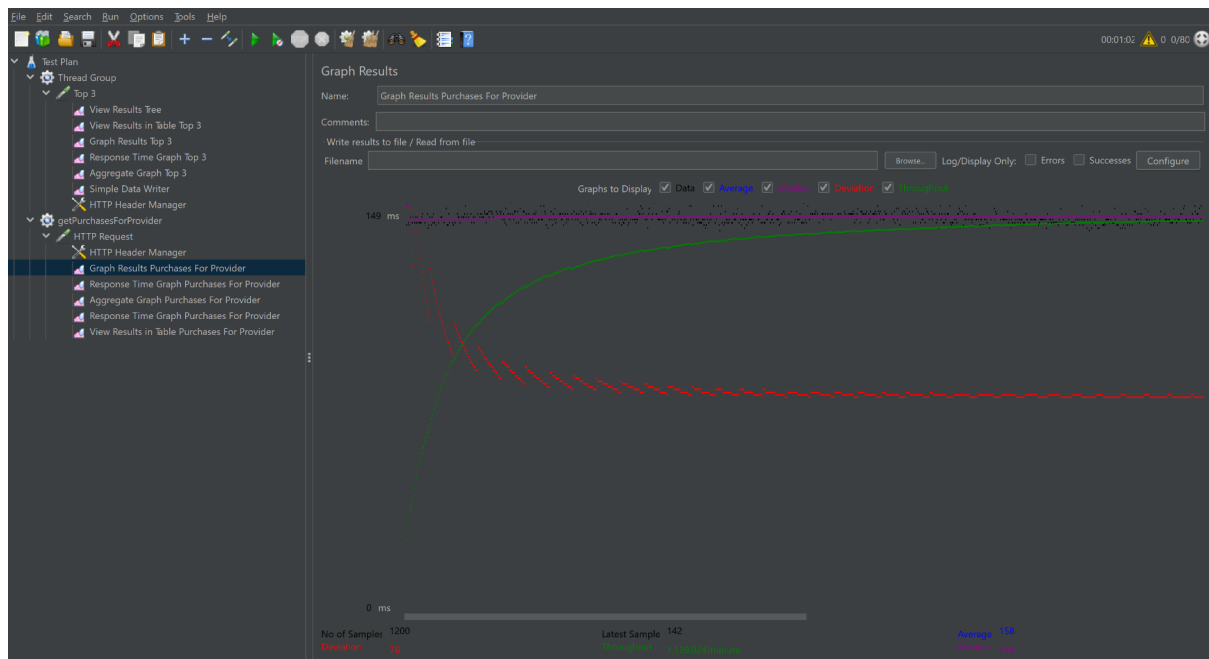
No of Samples: 1200 Latest Sample: 142

Average: 137 Deviation: 78

2. Consulta de compras para un proveedor

A continuación mostramos la gráfica de aggregate, response, los resultados y los datos utilizados para las pruebas de compras para un proveedor, como se puede ver en las graficas se cumple con los requerido con una carga de 1200 req/m el tiempo de respuesta medio es de 158 que es mucho menor que 300 que el máximo estipulado en el requerimiento. Para finalizar en la última gráfica se puede ver los resultados de todas las pruebas realizadas.





Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	00:20:35.408	getPurchasesFor...	HTTP Request	564	✓	1457	793	564	422
2	00:20:35.972	getPurchasesFor...	HTTP Request	148	✓	1457	793	148	0
3	00:20:36.121	getPurchasesFor...	HTTP Request	141	✓	1457	793	141	0
4	00:20:36.262	getPurchasesFor...	HTTP Request	149	✓	1457	793	149	0
5	00:20:36.411	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
6	00:20:36.553	getPurchasesFor...	HTTP Request	141	✓	1457	793	141	0
7	00:20:36.694	getPurchasesFor...	HTTP Request	143	✓	1457	793	143	0
8	00:20:36.837	getPurchasesFor...	HTTP Request	144	✓	1457	793	144	0
9	00:20:36.981	getPurchasesFor...	HTTP Request	143	✓	1457	793	143	0
10	00:20:37.124	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
11	00:20:37.266	getPurchasesFor...	HTTP Request	144	✓	1457	793	144	0
12	00:20:36.909	getPurchasesFor...	HTTP Request	602	✓	1457	793	602	460
13	00:20:37.410	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
14	00:20:37.512	getPurchasesFor...	HTTP Request	144	✓	1457	793	144	0
15	00:20:37.552	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
16	00:20:37.657	getPurchasesFor...	HTTP Request	144	✓	1457	793	144	0
17	00:20:37.695	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
18	00:20:37.801	getPurchasesFor...	HTTP Request	143	✓	1457	793	143	0
19	00:20:37.837	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
20	00:20:37.945	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
21	00:20:37.979	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0
22	00:20:38.088	getPurchasesFor...	HTTP Request	144	✓	1457	793	144	0
23	00:20:38.121	getPurchasesFor...	HTTP Request	145	✓	1457	793	145	0
24	00:20:38.232	getPurchasesFor...	HTTP Request	143	✓	1457	793	143	0
25	00:20:38.266	getPurchasesFor...	HTTP Request	143	✓	1457	793	143	0
26	00:20:38.375	getPurchasesFor...	HTTP Request	142	✓	1457	793	142	0

- **RNF. 2 - Confiabilidad y disponibilidad**

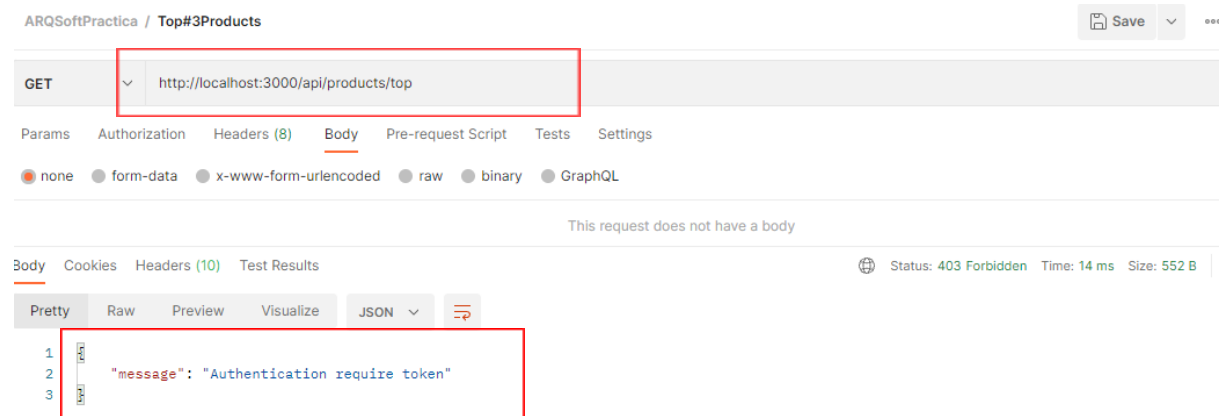
En este caso se dispone un endpoint que permite verificar el estado del sistema(bases de datos, servicios, etc). Este devuelve un mensaje especificando “Ok” o “Error” y cual sistema se encuentra desconectado o con fallas. A nivel de logs el backend disponibiliza en su terminal la impresión de aquellos eventos de interés elevado que puedan causar fallos en el sistema.

- **RNF. 3 - Observabilidad**

Para este requerimiento se usó winston, una librería de node.js que permite la utilización de logs, con una función que está constantemente guardando todas las peticiones, tiempos de respuesta y mensajes del backend pudimos cumplir con lo pedido que es monitorear todo el sistema ante eventuales fallas . A su vez se utilizó new relic , es una plataforma de monitoreo de rendimiento y análisis de aplicaciones que proporciona visibilidad en tiempo real de los sistemas en ejecución. Es una herramienta útil para el monitoreo de logs e información . New Relic proporciona una visibilidad completa de tus sistemas, lo que te permite identificar problemas de rendimiento y errores en tu aplicación en tiempo real. Además, da una visión clara de cómo tus sistemas están interactuando con otros sistemas, lo que te permite tomar decisiones informadas sobre cómo optimizar tus sistemas.

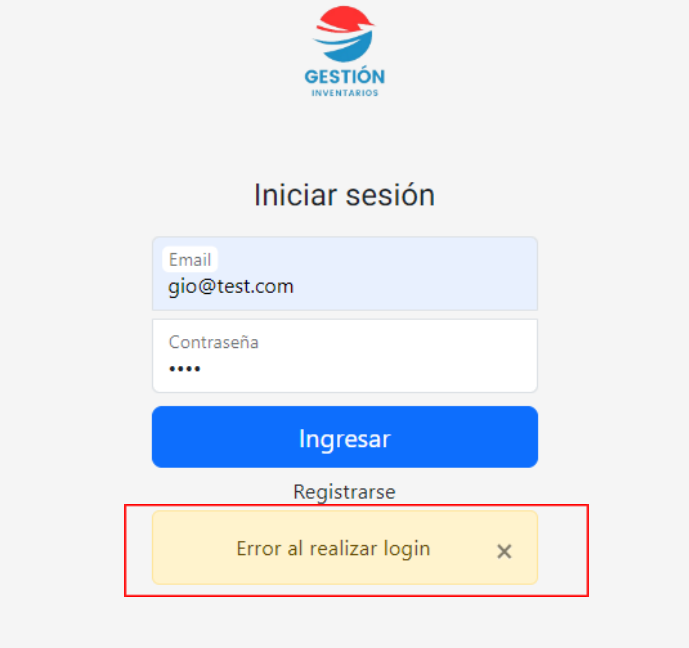
- **RNF. 4 y 5 - Autenticación, autorización y tenancy. Seguridad**

El sistema cuenta con un sólido sistema de acceso basado en roles que se otorga mediante un token JSON Web Token (JWT) con una firma criptográfica avanzada. Esto garantiza la autenticidad e integridad de los datos del token y previene manipulaciones o falsificaciones. Para eso utilizamos un middleware que verifica el rol asignado para ese endpoint. En el siguiente caso se intenta acceder al siguiente endpoint sin Autorización:



Además, el sistema ha sido diseñado para limitar la exposición de información sensible al usuario. Por ejemplo, si un usuario ingresa credenciales incorrectas al iniciar sesión, el sistema mostrará un mensaje de error genérico como "Error al realizar login" en lugar de revelar información adicional que pueda ayudar a un atacante a identificar el origen del fallo.

De esta manera, se garantiza la privacidad y la seguridad de los datos de los usuarios.



Si bien no se realizó un escaneo completo del Frontend utilizando Burp Suite, podemos visualizar inconvenientes en **Seguridad** que se deberían tratar en futuros releases. Se debería agregar encabezados de respuesta como:

- HTTP Strict Transport Security (HSTS)
- Content Security Policy (CSP)
- X-Frame-Options
- A su vez sería necesario ocultar información del server como: nombre, versiones, etc

En el caso del inicio de sesión no existe ningún mecanismo de defensa que contemple el ataque de fuerza bruta, por lo que se podría agregar un captcha o autenticación en dos pasos para prevenir este tipo de ataques. Si bien por letra se solicita cierto tipo de contraseña la misma no cumple con los estandares de NIST, ya que debe tener un largo mínimo de 8 caracteres, letras mayúscula, minúscula, números y caracteres especiales.

El sistema ha sido diseñado para manejar adecuadamente las URL malformadas, lo que ayuda a garantizar la seguridad y la usabilidad del sistema. Sin embargo, al discutir este enfoque, se ha identificado un posible tradeoff entre la usabilidad del sistema y la seguridad. Porque podría ser más fácil para un atacante intentar enumerar directorios de la aplicación y detectar vulnerabilidades en el sistema

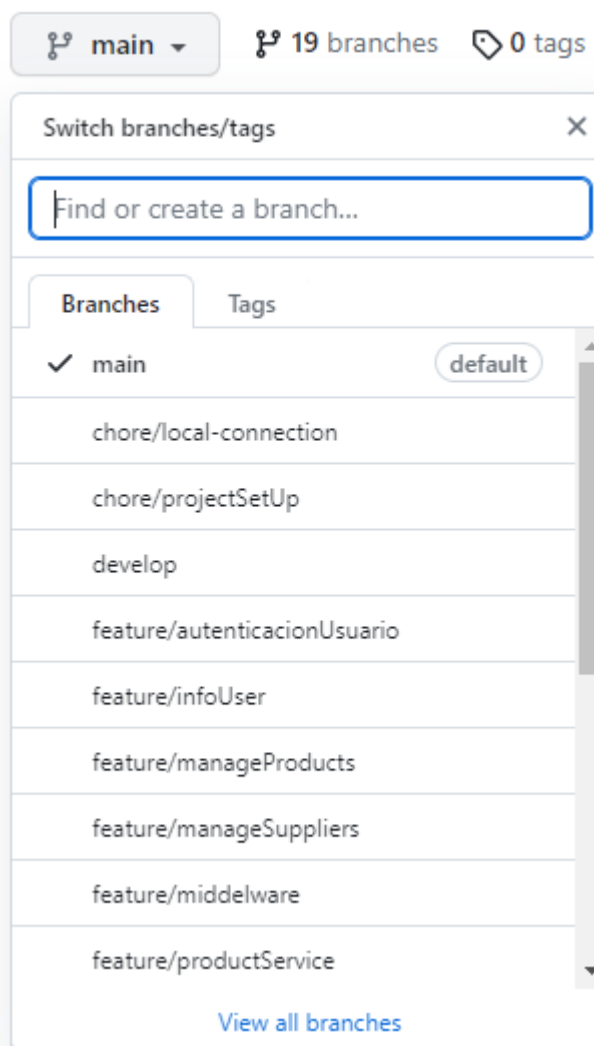
404 Not Found

[Volver](#)

- **RNF 6. Código fuente**

El control de versiones se llevó a cabo utilizando un único repositorio para el backend y otro para el frontend. El proyecto cuenta con un archivo README que detalla el propósito y los objetivos del proyecto. Además, se ha utilizado la estrategia de desarrollo conocida como git-flow, que se enfoca en mantener un flujo constante de trabajo y en la creación de ramas específicas para cada función o tarea.

Además, antes de realizar cambios significativos en la rama principal (develop), se ha empleado la práctica de pull requests. Esto implica que cualquier cambio importante en el código debe ser revisado y aprobado por otros miembros del equipo antes de ser fusionado con la rama principal. De esta manera, se garantiza que el código sea revisado y evaluado antes de ser implementado, lo que ayuda a prevenir errores y asegura la calidad del proyecto.



- **RNF 7. Integración continua**

En este caso el atributo favorecido es el de testeabilidad, realizamos pruebas unitarias RF5 y RF2. Para esto se utilizó Jest, proporcionando una plataforma completa para escribir y ejecutar pruebas unitarias. Las pruebas unitarias son una técnica de prueba de software que se utiliza para comprobar que una unidad de código, como una función o un método, funciona correctamente. La idea es aislar una pequeña parte del código y probarla de forma independiente, para detectar errores y problemas antes de integrarse en el resto del sistema. Las pruebas unitarias deben ser fáciles de escribir y ejecutar, y deben ser rápidas y precisas. En este caso las pruebas nos dieron un porcentaje de cobertura del 100%. A continuación mostraremos este resultado:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
src/config	100	100	100	100	
logger.ts	100	100	100	100	
src/controllers	100	100	100	100	
providerController.ts	100	100	100	100	
src/errors	100	100	100	100	
httpError.ts	100	100	100	100	
src/models	100	100	100	100	
product.ts	100	100	100	100	
provider.ts	100	100	100	100	
tests	100	100	100	100	
mocks.ts	100	100	100	100	
setup.ts	100	100	100	100	

- **RNF 8. Pruebas de carga**

Los resultados de esto fueron vistos en el requerimiento de Performance y el cual se pudo visualizar y realizar en jMeter realizando una carga de 1200 req/min, cumpliendo con los requerimientos que fueron pactados. La prueba se realizó directamente en esa herramienta. Y en un ambiente de producción.

- **RNF 8. Identificación de fallas**

Para facilitar la detección e identificación de fallas utilizamos Winston, es una biblioteca de registro (logging) para Node.js que se utiliza para registrar eventos y errores en una aplicación. Los registros son útiles para depurar y solucionar problemas en una aplicación. La biblioteca Winston es muy flexible y configurable, lo que permite personalizar la configuración de registro según las necesidades específicas de la aplicación. A su vez fue configurada para que los logs queden retenidos por un mínimo de 24hs.

Cada endpoint cuenta con un log para poder identificar y facilitar las fallas del sistema. A continuación mostraremos un ejemplo en nuestro sistema. A su vez contamos con logs para ver cuanto tiempo demora la respuesta de un endpoint y cuantas request por minuto está procesando el sistema.

```

async createSale(req: CustomRequest<any>, res: Response, next: NextFunction): Promise<void> {
  try {
    const company = req.user.company;
    const { products, client } = req.body;
    const newSale = await createSale(company, products, client);
    res.status(201).json(newSale);
    logger.info(`New sale created`);
  } catch (error: any) {
    logger.error(`Error in createSale: ${error.message}`);
    next(new HttpError(400, error.message));
  }
}

```

- **RNF 10. Portabilidad**

Se destaca la portabilidad del sistema, se necesita ir al directorio del proyecto e ingresar el comando. El proyecto cuenta con un archivo docker-compose.yml en donde están definidos los servicios.

☐ **docker-compose up --build**

Esto levanta tanto el backend,frontend, mongodb y redis.

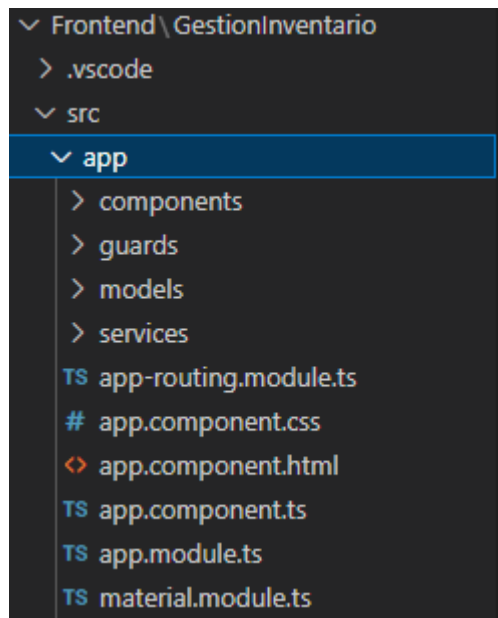
```
glowen@DESKTOP-K378880 MINGW64 ~/OneDrive/Escritorio/Facultad/ASP/197396-211753-205650-Backend (main)
$ docker-compose up --build
[*] Building 41.5s (18/18) FINISHED
-> [internal] load build definition from Dockerfile
-> -- transferring dockerfile: 465B
-> [internal] load .dockerignore
-> -- transferring context: 34B
-> [internal] load metadata for docker.io/library/node:14
-> [internal] load build context
-> -- transferring context: 2.59kB
-> [1/5] FROM docker.io/library/node:14@sha256:a158d3b9d4c3fa813fa68c590b8f8a868ed815ad4e59b0ce5744d2f6fd8461aa
-> CACHED [2/5] WORKDIR /app
-> [3/5] COPY package*.json ./
-> [4/5] RUN npm ci
-> [5/5] COPY . .
-> exporting to image
-> -- exporting layers
-> -- writing image sha256:a69f24dec356ca172bcd7424f895630ce791686e9d8a866003e34a20ff2d1
-> -- naming to docker.io/library/197396-211753-205650-backend-app
[*] Running 3/3
- Container 197396-211753-205650-backend-mongo-1 Created
- Container 197396-211753-205650-backend-redis-1 Created
- Container 197396-211753-205650-backend-app-1 Recreated
Attaching to 197396-211753-205650-backend-app-1, 197396-211753-205650-backend-mongo-1, 197396-211753-205650-backend-redis-1
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.705 # oO0000000000: Redis is starting oO0000000000
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.705 # redis version:7.0.11, bits=64, commit=00000000, modified=0, pid=1, just started
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.705 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.705 * monotonic clock: POSIX clock_gettime
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.708 * Running mode=standalone, port=6379.
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.708 # Server initialized
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.708 # WARNING Memory overcommit must be enabled! Without it, a background save or replication may fail under low memory condition. Being disabled, it can also cause failures without low memory condition, see https://github.com/jemalloc/jemalloc/issues/1328. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.710 * Loading RDB produced by version 7.0.11
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.710 * RDB age 163185 seconds
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.710 * RDB memory usage when created 0.82 Mb
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.710 * Done loading RDB, keys loaded: 0, keys expired: 0.
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.710 * DB loaded from disk: 0.000 seconds
197396-211753-205650-backend-redis-1 | 1:M 04 May 2023 23:40:54.710 * Ready to accept connections
197396-211753-205650-backend-app-1 |
197396-211753-205650-backend-app-1 | > obligatorie@1.0.0 start /app
197396-211753-205650-backend-app-1 | > node --max-old-space-size=4096 dist/app.js
197396-211753-205650-backend-app-1 |
```

5. Frontend

El frontend fue desarrollado utilizando Angular en su versión 15.2.6. Utiliza type-script, por lo que nuestro código se encuentra fuertemente tipado. Esta decisión está basada en que este framework nos provee un marco de aplicación de página única (SPA) una buena performance, con fácil extensibilidad pensando en el crecimiento del proyecto.

Angular proporciona una estructura predefinida del proyecto. La estructura de directorios y archivos se crea automáticamente cuando se genera un nuevo proyecto con la herramienta de línea de comandos de Angular (Angular CLI). Una de las ventajas de este framework es que sigue el patrón MVC (Modelo-Vista-Controlador). Con la vista separada de la lógica de negocio. A su vez está basado en componentes lo que nos permite reutilizar algunos de ellos en nuestra aplicación un ejemplo son el de las Alertas. Decidimos crear una arquitectura que nos permitiera organizar nuestro proyecto en directorios.

Estructura principal de nuestra aplicación:



Los modelos, servicios y componentes son artefactos clave que se utilizan para desarrollar aplicaciones web. Cada uno de ellos tiene un propósito específico en la arquitectura de la aplicación.

Modelos (Models): Los modelos son clases que representan la estructura de datos de la aplicación. Estos modelos se utilizan para definir objetos y su estructura en la aplicación.

Servicios (Services): Los servicios son clases que proporcionan funcionalidad compartida en toda la aplicación. Estos servicios se utilizan para separar la lógica de la vista y se encargan de tareas como la autenticación, la gestión de datos y la comunicación con la API del servidor. Los servicios también se utilizan para almacenar y compartir datos en toda la aplicación.

Componentes (Components): Los componentes son clases que representan la lógica de la vista de la aplicación. Cada componente tiene una plantilla HTML y una clase TypeScript que se encarga de la lógica de la vista. Los componentes se utilizan para dividir la interfaz de usuario en piezas más pequeñas y manejables y para proporcionar una estructura modular a la aplicación.

Guardas(Guards): Son una característica que se utiliza para proteger las rutas de la aplicación y para controlar el acceso a diferentes partes de la misma. Los guards se utilizan para interceptar las solicitudes de navegación antes de que se procesen y se utilizan para verificar si un usuario tiene permiso para acceder a una ruta determinada.

Decidimos crear una arquitectura “KISS”, es decir “Keep it simple”, esto para favorecer el atributo de calidad de Mantenibilidad y Extensibilidad. Algunos de las decisiones tomadas fueron las siguientes:

- Mantener los métodos pequeños.
- Nombre nemotécnicos, para funciones, variables y clases.
- Aplicación de principios SOLID.
- No abusar de los comentarios.

Para mejorar la UI se utilizó Angular Material, esta biblioteca proporciona una serie de componentes de UI preconstruidos y bien diseñados que se pueden utilizar para crear aplicaciones web modernas y atractivas. Los componentes de UI preconstruidos también se pueden personalizar para adaptarse a las necesidades específicas de nuestro proyecto “GestionInventario”. Esto nos permitió acelerar el proceso de desarrollo de la aplicación web y mejorar la coherencia y la calidad de la interfaz de usuario.

6. Cumplimiento con 12 factor app

A continuación detallaremos cómo se desempeña 12 factor en nuestro sistema.

1. Codebase: Tenemos un único repositorio para Backend como para Frontend.

2. Dependencies: Las dependencias en el Frontend se manejan de forma aislada en un package.json, de forma que el que desea ejecutar el Frontend deberá instalar las dependencias ejecutando el comando npm install. En el caso del Backend como explicamos en la parte de justificación de diseño (npm install).

3. Config: En el caso del Frontend se utiliza un archivo environment.ts el cual se tiene una variable de configuración para poder conectarse con el Backend, y por parte del Backend se utiliza archivos .env variables de configuración.

4. Backing services: Creamos en docker-compose un container con una instancia de redis y una instancia de mongo para deployar el backend con el comando docker-compose up --build.

5. Build, release, run: Ambos sistemas tanto Frontend como Backend cumplen con este requisito.

6. Procesos: Se cumple con este requisito de forma natural, ya que al utilizar una arquitectura REST, la misma ya cumple con contar con procesos stateless, los cuales no comparten estados. De todas formas nuestros procesos son independientes y no dependen de los estados de otros.

7. Port binding: A nivel de docker si cumplimos con este requisito, ya que se detalla en qué puerto queremos levantar y hacer el binding a nuestra máquina, esto lo hacemos solamente para exponer el Frontend, ya que no queremos que nadie acceda desde afuera a nuestro Backend o base de datos.

8. Concurrency: En ambos sistemas utilizamos la concurrencia ya que ambos usan la asincronía.

9. Disposability: El tiempo que tarda la aplicación en iniciarse puede variar ligeramente dependiendo de diversos factores. En el caso del backend de la aplicación, se ha discutido sobre el equilibrio entre el rendimiento del inicio y la facilidad de realizar modificaciones en el futuro. El equipo ha llegado a la conclusión de que se encuentra en un punto intermedio, donde se ha logrado un equilibrio adecuado entre ambas consideraciones.

10. Dev/prod parity: durante el desarrollo, únicamente se utilizó el ambiente de Dev, el cual siempre estaba actualizado. Si bien no se utilizaron diferentes ambientes, siempre fuimos conscientes de tratar de mantener Dev lo más actualizado posible, utilizando los Pull-Request para una correcta actualización en

Dev.

11. Logs: utilizamos la biblioteca Winston, esto se encuentra comentado en la sección anterior de logs.

12. Admin processes: La creación de la base de datos puede requerir permisos de root como visto en el docker-compose.

7. Descripción del proceso de deployment

El equipo realizó el deploy en **Heroku**.

Para realizar el deploy en Heroku lo primero que se hizo fue crear una cuenta en Heroku. Luego de crear la cuenta en Heroku, el equipo se descargó la Heroku CLI (Command Line Interface) para poder hacer login desde la misma.

Una vez que eso está hecho, los pasos a seguir son:

Proceso de deployment de la app

1. Navegar a la carpeta raíz del proyecto dentro de la CLI
2. Crear una app de Heroku mediante la interfaz.
3. Crear un archivo llamado 'Procfile' en la ruta del proyecto. Este archivo le dice a heroku como iniciar la aplicación, en este caso, el archivo contiene: npm start.
4. Para deployar el frontend, se editó el objeto configurations dentro del archivo 'angular.json' para que cuando se corre el proyecto con el parámetro de producción (production) se reemplaza el archivo de env por el archivo de env de production. Este archivo env de production tiene la ruta de la app de heroku que se creó mediante la interfaz.
5. Para deployar tanto el frontend como el backend juntos, dentro de la raíz del backend se creó una carpeta llamada 'public' en la cual se incluyó la carpeta 'dist' del frontend.
6. Una vez que ya está el código listo para ser deployado, se utiliza el siguiente comando 'git push heroku main' para que se haga el deploy a heroku con los últimos cambios.
7. Una vez hecho el deploy, se puede acceder a todo el proyecto desde una URL, la cual es: <https://gestion-inventario-ort.herokuapp.com/login>

Proceso de deployment base de datos y redis

MongoDB

1. Es necesario crear una cuenta de MongoDB Atlas, para luego en panel de control de Atlas, crear un nuevo proyecto.
2. Una vez creado el proyecto, se le da al botón "Crear un clúster". Se elige el proveedor de la nube y la región, y la versión de MongoDB.
3. Haga clic en el botón "Crear clúster" para iniciar el proceso de despliegue.
4. Después de que se crea el clúster, se navega hasta la pestaña "Colecciones" y se clickea "Crear base de datos".
5. Se elige el nombre y se hace clic en "Crear" para crear la nueva base de datos y colecciones.
6. Una vez que está creada, se obtiene del dashboard la mongo URI.

Luego, en el código, se crea el archivo config.ts en el cual se define la cadena de conexión en la constante "connectionString", el nombre de usuario, la contraseña, la dirección y el puerto. Se define además la función "connectDB", que se encarga de conectar a la base de datos MongoDB y la función "disconnectDB" que se utiliza para desconectar de la base.

Redis

El equipo optó por red en la nube, por lo cual es necesario crear una cuenta en app.redislabs.com.

Una vez creada la cuenta, en el panel de control, se hace click en "Create New Database".

Se elige el plan gratuito y la región, y se le da el nombre deseado.

Luego de dar click en "Create Redis Database" queda creado y del dashboard obtenemos la url, el usuario y la password.

Una vez obtenida la url con su usuario y password, dentro del archivo cache.ts en el proyecto de backend, se define la URL de Redis en la constante "redisUrl", además del nombre de usuario, la contraseña, la dirección y el puerto. Se crea una nueva instancia del cliente Redis utilizando la URL. Además se exportan tres funciones "getAsync", "setAsync" y "setexAsync" que se utilizan para leer y escribir datos en la base de datos Redis.