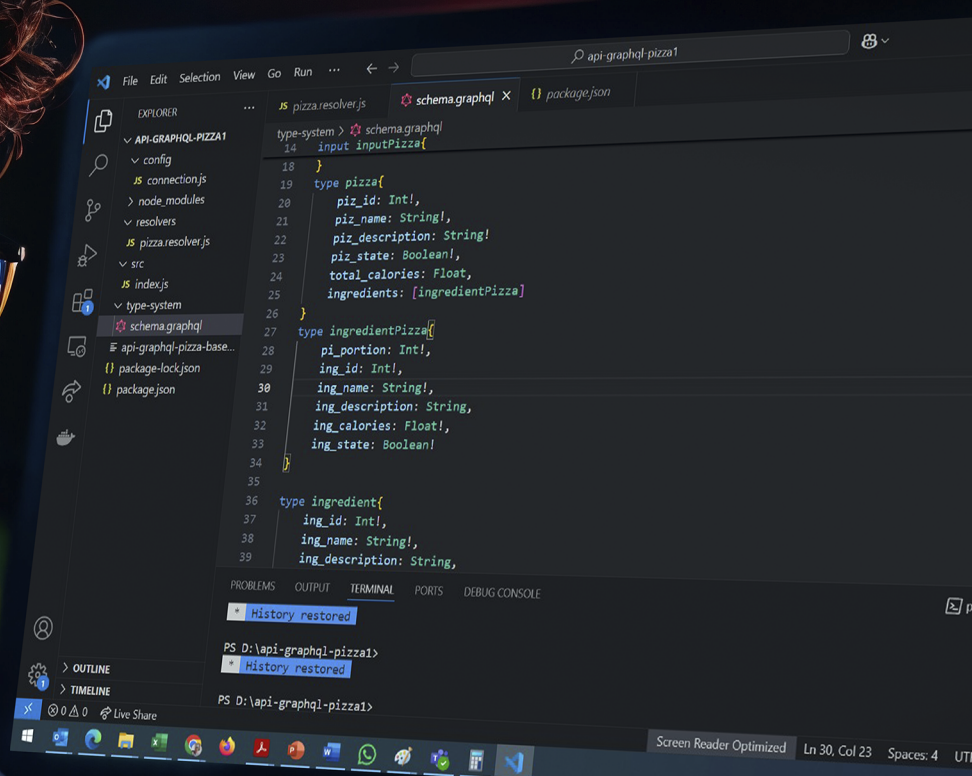


GraphQL

DESARROLLO FULLSTACK

Desarrollo web orientado a microservicios con GraphQL



Antonio Quiña-Mera
Cathy Guevara-Vega
Jorge Carcelén



GraphQL: Desarrollo FullStack

GraphQL: Desarrollo FullStack

Desarrollo web orientado a microservicios con GraphQL

Antonio Quiña-Mera
Cathy Guevara-Vega
Jorge Carcelén



CRÉDITOS

Edita

Editorial Universidad Técnica del Norte
Av. 17 de Julio, 5-21
y Gral. José María Córdova
Campus El Olivo
IBARRA - IMBABURA - ECUADOR
www.utn.edu.ec
editorial@utn.edu.ec



Autores

Antonio Quiña-Mera, Ph.D.
Grupo de Investigación de Ciencias en Red (e-CIER)
Universidad Técnica del Norte, Ecuador
aquina@utn.edu.ec
<https://orcid.org/0000-0003-2516-9016>

Cathy Guevara-Vega, Ph.D.
Grupo de Investigación de Ciencias en Red (e-CIER)
Universidad Técnica del Norte, Ecuador
cguevara@utn.edu.ec
<https://orcid.org/0000-0002-2470-8287>

Jorge Carcelén Barahona, Ing.
Universidad Técnica del Norte, Ecuador
jacarcelenb@utn.edu.ec
<https://orcid.org/0009-0004-3703-0024>

Proyecto de Investigación



Pares revisores externos

Víctor Hugo Saquicela, Ph. D
Universidad de Cuenca, Ecuador
victor.saquicela@ucuenca.edu.ec

Geovanny Raura Ruiz, Ph. D
Universidad de las Fuerzas Armadas, Ecuador
graura@espe.edu.ec

Revisión de estilo

Silvia Arciniega Hidrobo, MSc.
Universidad Técnica del Norte, Ecuador
srarciniega@utn.edu.ec

Concepto de diseño

Eduardo Alejandro Díaz Ruano,
Portada y Contraportada
Portadillas generadas con ChatGPT
Logotipo de la colección
Carrera de Diseño Gráfico
Universidad Técnica del Norte

Grupos de Investigación



GISIA
Grupo de Investigación
Ingeniería de Software
e Inteligencia Artificial

© de los textos y fotografías:
Sus respectivos autores
© de esta edición:
Editorial Universidad Técnica del Norte
1ª edición, digital: Noviembre de
2025

e-ISBN: 978-9942-572-30-1

DOI: 10.53358/libfica/JHBO4815

Prohibida la reproducción total o parcial de esta obra
sin la previa autorización escrita
de la Editorial Universidad Técnica del Norte

ÍNDICE

PRESENTACIÓN	14
CAPÍTULO 1 Introducción	16
Arquitecturas de software	16
Importancia de la arquitectura de software	17
Patrones de arquitectura de software	17
Microservicios	22
FullStack	25
Backend	26
Frontend	26
APIS	27
Breve Historia	27
¿Qué es una API?	28
¿Cómo funcionan las APIs?	28
Protocolos para APIs	29
Antecedentes	30
REST	30
API REST	30
¿Cómo funcionan las APIs REST?	30
Diferencias entre REST y GraphQL	32
Resumen	34
CAPÍTULO 2 Fundamentos de GraphQL	36
Componentes principales de GraphQL	36
Lenguaje GraphQL	36
Sistema de tipos (Type System)	38
Introspección (Introspection)	39
Validación (Validation)	40
Ejecución (Execution)	40
Ejemplo de Ejecución	41
Resolvedores (Resolvers)	42
Componentes del Sistema de Tipos	42
Esquemas de datos	42
Tipos	42
Ejemplo de Tipos en GraphQL	43
Operaciones de GraphQL	44
Consultas	44
Mutaciones	45
Suscripciones	46
Fragmentos	47
Resumen	50

CAPÍTULO 3 Backend con GraphQL: Implementación de APIs	52
Configuración del ambiente de desarrollo	52
Herramientas para el desarrollo de la API	52
Instalación de las herramientas para el desarrollo de la API	54
Creación del proyecto	56
Conexión a la base de datos	60
Sistema de tipos	63
Schemas	63
Operaciones	67
Consultas	70
Mutaciones	71
Resumen	73
Actividades	73
CAPÍTULO 4 Frontend con GraphQL: Implementación del Cliente y Consumo de APIs	75
Configuración del ambiente de desarrollo en React	75
Características clave de React	76
Creación del proyecto en React	77
Operaciones	82
Creación del servicio message-service	82
Consultas	85
Creación del servicio pizza-service	85
Creación del servicio ingredient-service	89
Creación del servicio util-service	90
Creación de componentes	91
Creación del componente shared	93
Creación del componente home	94
Creación del componente pizza-form	95
Creación del componente pizza-list	104
Creación del componente ingredient-form	111
Creación del componente ingredient-list	117
Resumen	125
Actividades	125
CAPÍTULO 5 GraphQL Hands-On: Retos Prácticos para Desarrolladores	127
ORM (Mapeo Objeto-Relacional)	128
ORM: Sequelize	129
Implementación del ORM en un proyecto Backend	129
Seguridad en GraphQL	146
JWT y Autenticación	146
Menús dinámicos	182
Creación del proyecto backend en NodeJS	184
Manejo avanzado de tablas	239
Modificación del proyecto frontend creado	239
Reportería	243
Consumo e integración con APIs REST	264
Consumo del API REST de GEMINI	264
GraphQL en la Nube	295
Hasura	295
Resumen	303
Actividades	304

ÍNDICE DE FIGURAS

1.1	Patrones de arquitectura de Software	18
1.2	Arquitectura por capas	19
1.3	Arquitectura basada en eventos	20
1.4	Arquitectura microkernel	21
1.5	Arquitectura de microservicios	22
1.6	Arquitectura basada en el espacio	23
1.7	Características destacadas de las APIs	29
1.8	Características API REST	31
2.1	Componentes del paradigma GraphQL	37
3.1	Descarga de Node.js	54
3.2	Instalación de Node.js	55
3.3	Descarga de Visual Studio Code	56
3.4	Descarga de PostgreSQL	56
3.5	Creación de la carpeta del proyecto	57
3.6	Abrir carpeta en Visual Studio Code	57
3.7	Abrir terminal en VS-Code	58
3.8	Verificación del funcionamiento de Node	58
3.9	Inicialización del proyecto	59
3.10	Package.json	59
3.11	Archivo index.js	60
3.12	Estructura de proyecto	60
3.13	Creación de la base de datos	61
3.14	Servidor de Apollo	70
3.15	Consulta en Apollo Server	71
3.16	Consulta con argumentos en Apollo Server	71
3.17	Mutación para crear pizzas en Apollo Server	72
3.18	Mutación para actualizar pizzas en Apollo Server	72
4.1	Inicialización del proyecto	78
4.2	Configuración de las propiedades del proyecto	78
4.3	Selección del framework del proyecto	78
4.4	Selección la variante del proyecto	79
4.5	Selección la variante del proyecto	79
4.6	Ejecución del proyecto	80
4.7	Estructura del proyecto	80
4.8	Componente principal del proyecto	81
4.9	Borrar los estilos del proyecto	81
4.10	Importación de dependencias	82
4.11	Vista principal	108
4.12	Listado Pizza	108

4.13	Crear Pizzas	109
4.14	Pizza creada	109
4.15	Actualizar Pizzas	110
4.16	Pizza actualizada	110
4.17	Eliminar Pizzas	111
4.18	Pizza eliminada	111
4.19	Vista principal	120
4.20	Listado de ingredientes	121
4.21	Crear Ingredientes	121
4.22	Ingrediente creado	122
4.23	Actualizar ingredientes	122
4.24	Ingrediente actualizada	123
4.25	Eliminar Ingredientes	123
4.26	Ingrediente eliminado	124
5.1	Inicialización del proyecto	130
5.2	Package.json	130
5.3	Componente principal del aplicación	145
5.4	Componente Pizzas	145
5.5	Componente Ingredientes	146
5.6	Base de datos modificada	148
5.7	Estilos Primeract	163
5.8	Login	171
5.9	Inicio de Sesión	181
5.10	Página principal	181
5.11	Menú lateral	182
5.12	Modelo modificado de la base de datos	183
5.13	Tabla Menu	184
5.14	Tabla Rol Menu	184
5.15	Menús del Usuario	238
5.16	Menús del usuario administrador	238
5.17	Manejo de Tablas	242
5.18	Manejo de Tablas Filtrado	242
5.19	Opciones para generar el reporte PDF	252
5.20	Reporte PDF	252
5.21	Gráficos Dashboard	263
5.22	Gráficos Dashboard	263
5.23	Gráficos Dashboard	264
5.24	Página Oficial de Gemini	265
5.25	Ingreso a Google AI Studio	266
5.26	Crear clave del API	266
5.27	Clave del API	267
5.28	Clave del API	267
5.29	Estructura de carpetas	268
5.30	Aplicación Frontend	274
5.31	Aplicación Frontend	274
5.32	Aplicación Frontend	275
5.33	Estructura proyecto	276
5.34	Componente Popular Videos	286
5.35	Vista Principal	293
5.36	Búsqueda de videos	294

5.37	Búsqueda de canales	294
5.38	Plataforma Hasura	296
5.39	Vista Plataforma Hasura	296
5.40	Consola Hasura	297
5.41	Proyecto Creado	297
5.42	ElephantSQL	298
5.43	Creación base de datos	298
5.44	Información base de datos	299
5.45	Conexión a la base de datos	299
5.46	Cambio de credenciales en la base de datos	300
5.47	Seleccionar entidades de la base de datos	300
5.48	Prueba de la API creada en Hasura	301
5.49	Prueba de la API en Postman	301
5.50	Resultados de la consulta de la API en Postman	302

ÍNDICE DE TABLAS

I	Características de los microservicios	26
II	Diferencias entre backend y frontend	27
III	Métodos para peticiones REST	32
IV	Diferencias entre las propiedades de REST y GraphQL	33

Este libro está dirigido a profesionales, investigadores, docentes y estudiantes que deseen profundizar en el desarrollo de software con un enfoque en tecnologías emergentes como GraphQL. A lo largo de sus páginas, los autores presentan un conjunto de experiencias, estrategias innovadoras y metodologías respaldadas por investigaciones y buenas prácticas. Con un enfoque didáctico y orientado a la práctica, este libro no solo busca fortalecer la formación académica, sino también preparar a los estudiantes universitarios para enfrentar los desafíos tecnológicos actuales y futuros. A través de ejemplos concretos, los lectores adquieren una comprensión profunda y aplicable de GraphQL y su papel en la evolución de la ingeniería del software.



Presentación

El desarrollo de aplicaciones web ha evolucionado significativamente en las últimas décadas, impulsado por la necesidad de optimizar el acceso a los datos y mejorar la eficiencia en la gestión de servicios digitales. En este contexto, GraphQL ha emergido como una alternativa innovadora al tradicional modelo REST, proporcionando mayor flexibilidad en las consultas y un control más eficiente sobre la información transferida entre clientes y servidores.

El paradigma GraphQL ha experimentado una adopción creciente por parte de grandes empresas como GitHub, Netflix y Twitter; aún existen desafíos en la recopilación de evidencia empírica sobre su impacto en la industria y su implementación en entornos gubernamentales. Asimismo, se identifican áreas poco exploradas dentro del ecosistema GraphQL, lo que abre nuevas líneas de investigación para el futuro.

Este libro tiene como objetivo presentar la importancia de las arquitecturas de software, el uso de los microservicios y la aplicación de APIs mediante el lenguaje GraphQL contemplando su impacto en la industria del software y la investigación académica. Para ello, se realiza una descripción detallada de los tipos de arquitecturas de software, características de los microservicios, el desarrollo FullStack tanto en Backend como en Frontend y el funcionamiento de las APIs.

A lo largo de este libro, los lectores encontrarán una descripción detallada del paradigma de GraphQL, sus principios de diseño, su aplicación en arquitecturas de microservicios y su creciente adopción en la industria. Además, se presentan comparaciones entre GraphQL y REST, ilustrando cómo GraphQL soluciona problemas de sobrecarga y subcarga de datos en las APIs tradicionales.

Este libro está dirigido tanto a desarrolladores, arquitectos de software e investigadores, como a cualquier persona interesada en comprender a fondo el ecosistema GraphQL. El propósito de los autores es proporcionar un recurso riguroso y actualizado que sirva como referencia para la comunidad técnica y académica.

Se espera que este libro sirva como una guía para la adopción de GraphQL en proyectos tecnológicos y que inspire futuras investigaciones y aplicaciones en el campo de las APIs modernas.

Con agradecimiento sincero y el anhelo de un futuro transformador,

Los autores



CAPÍTULO 1

Introducción

«Todos los libros, los materiales de aprendizaje y las evaluaciones deberían llevarse a cabo de manera digital e interactiva, adaptada a cada estudiante de forma que pudiera recibir información sobre su progreso en tiempo real.»

Einstein, citado en Calaprice, 2000, p. 313.

La arquitectura de software es fundamental para el desarrollo de aplicaciones informáticas efectivas y escalables. A lo largo de las últimas décadas han surgido diversos patrones arquitectónicos para abordar los desafíos y requisitos cambiantes de las soluciones empresariales modernas. Uno de los enfoques innovadores y prometedores es la arquitectura de microservicios, que promueve la descomposición de aplicaciones monolíticas en conjuntos de servicios independientes y especializados. Esta arquitectura ofrece beneficios como equipos ágiles enfocados, despliegues incrementales sin interrupciones, escalamiento elástico, mayor resiliencia, flexibilidad tecnológica y facilidad para la innovación continua. En este contexto, el enfoque fullstack cobra relevancia al abarcar, por un lado, el desarrollo frontend centrado en interfaces atractivas y, por otro, el backend robusto que soporta la lógica de negocio y gestión de datos. Mientras que las interfaces de programación de aplicaciones (APIs) desempeñan un papel clave al permitir la comunicación entre componentes siguiendo protocolos estandarizados como REST o el más reciente GraphQL, que busca abordar limitaciones de las arquitecturas REST tradicionales.

ARQUITECTURAS DE SOFTWARE

La arquitectura de software es un conjunto estructurado de prácticas, principios y frameworks que organizan la descripción, diseño y evolución de una aplicación informática [1]. Estas estructuras incluyen los elementos de software, sus interconexiones y las propiedades inherentes a ambos, proporcionando una base para el diseño, evolución y mantenimiento del sistema [2].

La arquitectura de software es un medio en constante crecimiento dentro de la ingeniería del software debido a que facilita la comunicación entre las personas interesadas en automatizar un proceso y los desarrolladores de software. Además, permite mejorar aspectos como la trazabilidad entre los requisitos y las soluciones informáticas, permitiendo que se puedan reducir los riesgos durante el ciclo del desarrollo del software [3].

La arquitectura de software representa las abstracciones de una aplicación informática, definiendo su estructura, comportamiento y propiedades clave. Incluye componentes de software y su relación con las condiciones del entorno [4].

Importancia de la arquitectura de software

La arquitectura de software es crucial debido a su impacto directo en la capacidad de un sistema informático para cumplir con los atributos de calidad, como el rendimiento, la usabilidad y la escalabilidad. Estos atributos son parte de los requisitos no funcionales; deben expresarse de manera cuantitativa para ser evaluados objetivamente. La estructura del software influye en la satisfacción de estos atributos; por ejemplo, un software con una estructura que requiere muchas transiciones entre componentes puede tener un rendimiento deficiente [5].

Además de influir en los atributos de calidad, la arquitectura guía el desarrollo al dividir el sistema en componentes, asignando tareas a individuos o grupos. Esta estructuración es esencial para la planificación del proyecto. No obstante, los diseños arquitectónicos pueden ser reutilizados, lo que reduce costos y mejora la calidad, especialmente si han sido exitosos en aplicaciones informáticas previas. En resumen, la arquitectura de software no solo impacta la funcionalidad del sistema informático, sino que también juega un papel crucial en la eficiencia, adaptabilidad y reutilización en el desarrollo del software [5].

Patrones de arquitectura de software

Se conoce como un patrón a una solución recurrente a un problema de diseño en un contexto específico [6]. Los patrones de arquitectura ofrecen soluciones a los desafíos que surgen durante el desarrollo de software, evitando la duplicación de código fuente y favoreciendo su reutilización. Proporcionan una estructura clara que facilita la organización del código fuente, permitiendo desarrollar software de calidad, mejorar su mantenimiento y optimizar la comprensión del código fuente a través de la modularidad [6].

Es común observar que gran parte de los desarrolladores empiecen a construir una aplicación informática sin una arquitectura establecida de manera formal, clara y bien definida. La mayoría de los desarrolladores suelen escoger el patrón de arquitectura tradicional en capas (también llamado arquitectura en N capas), creando capas implícitas al separar los módulos de código fuente en paquetes. Sin embargo, a menudo esa práctica resulta ser códigos de archivos fuente que carecen de organización y presenta problemas que pueden afectar la calidad del producto de software que se va a desarrollar [7].

La Fig. 1.1 muestra los patrones que se utilizan para establecer las diferentes arquitecturas de software que se pueden utilizar en el desarrollo de aplicaciones informáticas.

Arquitectura por capas

La arquitectura por capas es un enfoque de diseño de software que extiende el patrón de capas tradicional, separando la lógica del negocio en módulos independientes y estructurados. Se centra en el dominio del problema, asegurando que las reglas y procesos empresariales sean el núcleo del sistema, mientras que las capas técnicas (como la presentación y persistencia de datos) se organizan de manera desacoplada para facilitar el mantenimiento, la escalabilidad y la evolución del software [7].

Este patrón se ha convertido en el estándar para la mayoría de las aplicaciones empresariales Java EE, por lo tanto, es ampliamente conocido y utilizado por la mayoría de arquitectos, diseñadores y programadores de software. La popularidad del patrón de arquitectura por capas se debe en gran medida a que su estructura en niveles separados refleja de manera muy similar la estructura jerárquica y de comunicación que existe tradicionalmente en los departamentos de tecnología de la información de las empresas, así como en la propia organización de la mayoría de las compañías. Esta similitud hace que la arquitectura por capas sea una opción natural y familiar para la mayoría de los esfuerzos de desarrollo de aplicaciones empresariales [7].

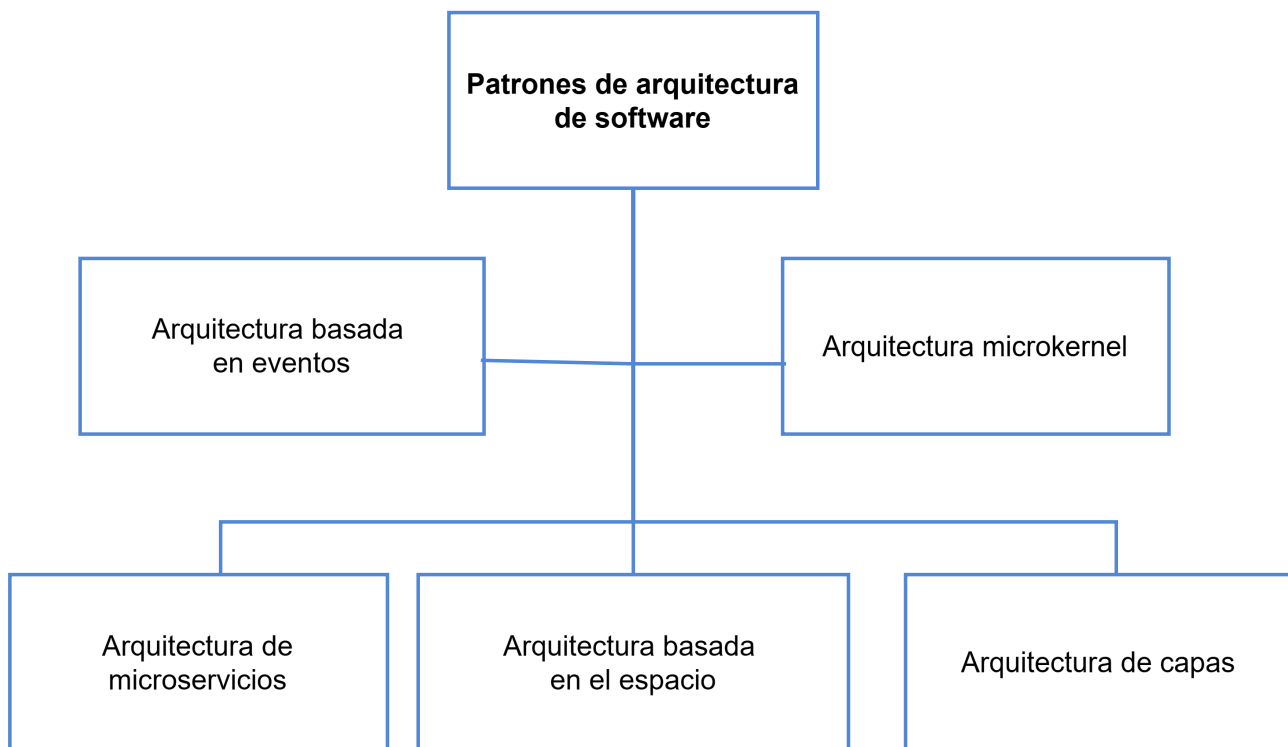


Fig. 1.1. Patrones de arquitectura de Software

La Fig. 1.2 muestra un ejemplo básico de la arquitectura de capas. Cabe mencionar que el número de capas se determina con base en las necesidades del negocio o de las funciones que se necesiten implementar en la aplicación informática. Por ejemplo, pueden existir casos en donde la capa de negocio se subdivida en secciones que abarquen distintos aspectos de la lógica de negocio [7].

Arquitectura basada en eventos

La arquitectura basada en eventos (EDA) es un popular patrón arquitectónico asíncrono y distribuido que se utiliza para construir aplicaciones altamente escalables. Es adaptable y puede emplearse en aplicaciones pequeñas, medianas y grandes. La EDA está compuesta por componentes de procesamiento de eventos desacoplados y de propósito único que reciben y procesan eventos de forma asíncrona. Cada componente ejecuta una función específica desencadenada por eventos entrantes, sin necesidad de conocer el estado general de la aplicación. Esta característica permite escalar fácilmente añadiendo o quitando instancias de componentes según sea necesario [7]. Existen dos topologías principales de EDA: i) el mediador y ii) el corredor. En la topología del mediador, los componentes se suscriben a un mediador central que recibe todos los eventos y los enruta a los componentes suscritos. El corredor, en cambio, permite una comunicación directa entre los componentes mediante el intercambio de eventos a través de un bus de mensajes. La EDA promueve bajos acoplamientos, alta cohesión, escalabilidad inherente y resiliencia. Por ello se ha convertido en una opción muy popular para sistemas críticos y procesamiento de datos a gran escala, como aplicaciones de comercio electrónico, financieras, de internet de las cosas (IoT), etc. Permite construir pipelines de procesamiento de eventos de forma flexible y eficiente.

La Fig. 1.3 muestra un ejemplo de la arquitectura basada en eventos. No hay un componente central mediador de eventos que controle y orqueste el evento inicial; en su lugar, cada componente procesador de eventos es responsable de procesar un evento y publicar un nuevo evento que indica la acción que acaba de realizar. Por ejemplo, un procesador de eventos que gestiona el equilibrio de una cartera de acciones puede recibir un evento inicial llamado “división de acciones”. A partir de este evento, el procesador puede

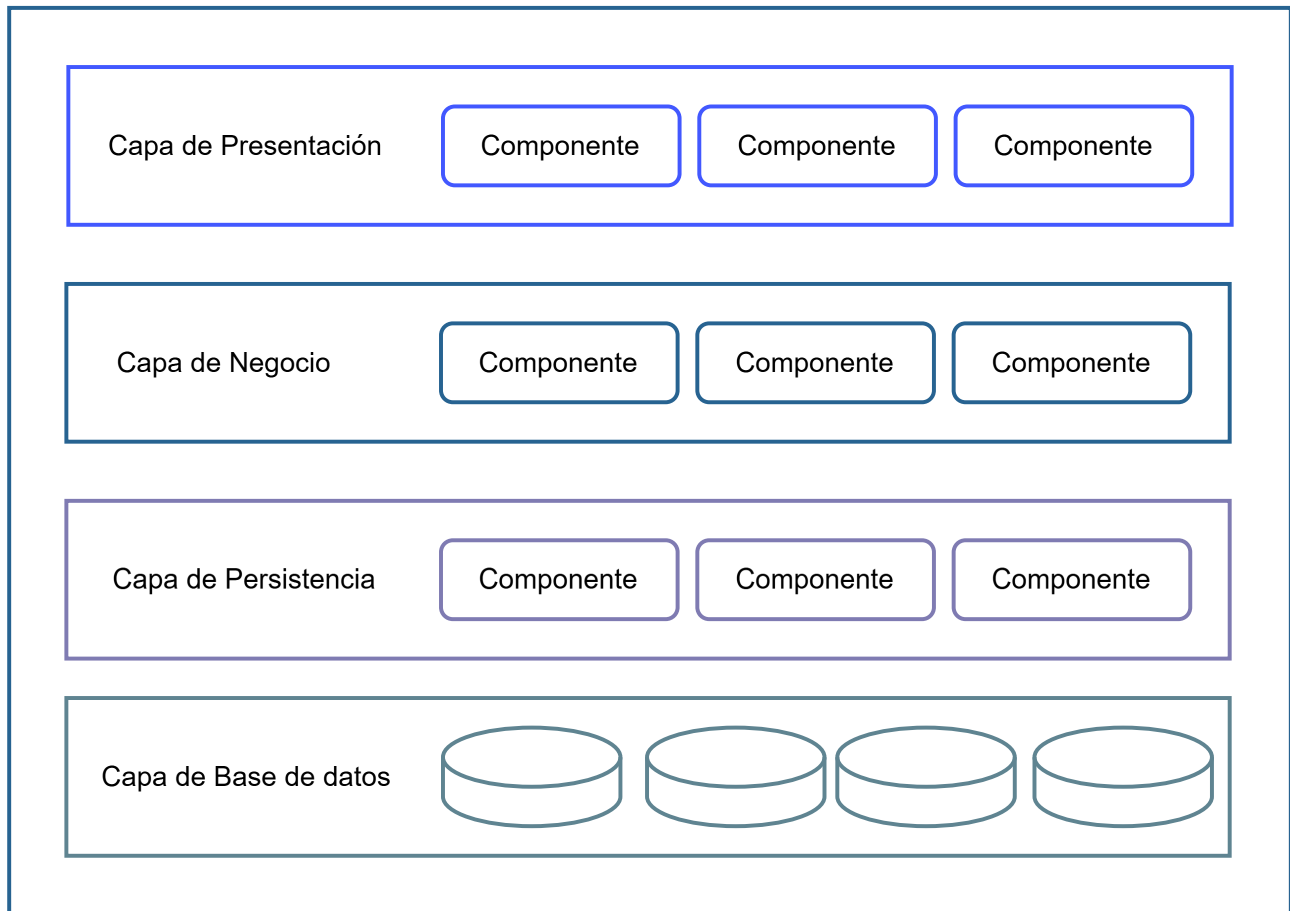


Fig. 1.2. Arquitectura por capas

ajustar la cartera y luego publicar un nuevo evento denominado “reajuste de cartera”, que será recogido por otro procesador de eventos. Cabe señalar que, en algunos casos, un procesador de eventos puede publicar un evento que no sea recogido por ningún otro procesador. Esto es común cuando se está evolucionando una aplicación o preparando su funcionalidad para futuras extensiones [7].

Arquitectura microkernel

El patrón de arquitectura de microkernel (traducido al español como micronúcleo) es una opción natural para implementar aplicaciones empaquetadas como productos de software para su distribución y descarga. Muchas compañías también desarrollan sus aplicaciones empresariales internas como productos, con versiones, notas de lanzamiento y características. Este tipo de aplicaciones encaja bien con este patrón arquitectónico. La arquitectura de micronúcleo permite agregar funcionalidades adicionales a la aplicación principal en forma de plugins, brindando extensibilidad y aislamiento. El núcleo contiene la funcionalidad común, mientras que los plugins proveen características específicas y opcionales. Esta separación promueve bajo acoplamiento y alta cohesión. Los beneficios clave del patrón de micronúcleo son la capacidad de extender fácilmente la aplicación, actualizar partes concretas de forma independiente, personalizar la funcionalidad para distintos usuarios y distribuir nuevas características sin necesidad de rediseñar o testear desde cero. Esto agiliza el mantenimiento y desarrollo. Por ello es ideal para productos empresariales multiusuario distribuidos como servicios [7].

En la Fig. 1.4, los módulos de complemento (componente conectado) son componentes independientes y autónomos que contienen procesamiento especializado, funciones adicionales y código personalizado diseñado para mejorar o extender el sistema central, agregando capacidades empresariales adicionales.

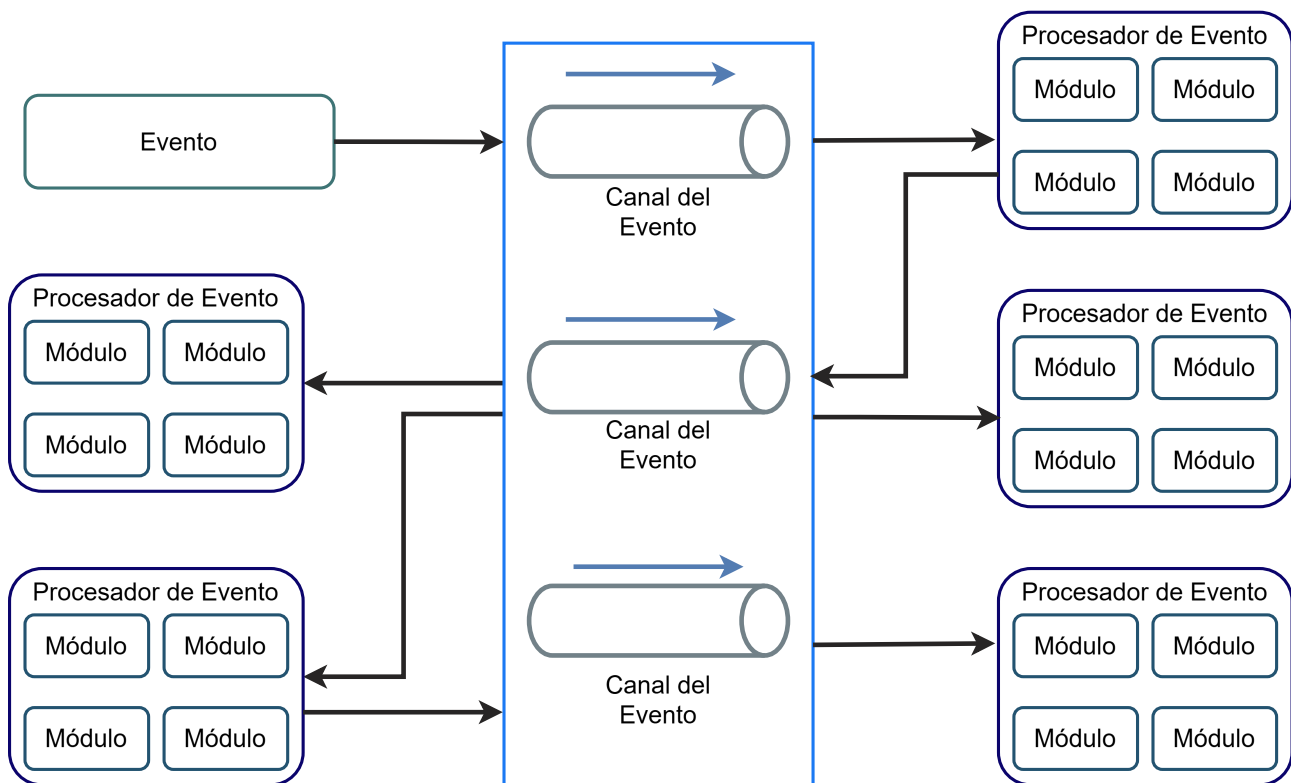


Fig. 1.3. Arquitectura basada en eventos

En general, los módulos de complemento deben ser independientes entre sí, aunque es posible diseñar complementos que requieran la presencia de otros. De cualquier forma, es importante minimizar la comunicación entre ellos para evitar problemas de dependencia [7]. El sistema central debe conocer qué módulos de complemento están disponibles y cómo acceder a ellos. Una forma común de implementar esto es mediante un registro de complementos, que almacena información sobre cada módulo, como su nombre, contrato de datos y detalles del protocolo de acceso remoto (según cómo se conecte al sistema central) [7]. Por ejemplo, un complemento de software fiscal que identifica posibles auditorías de alto riesgo podría tener una entrada en el registro con el nombre del servicio (AuditChecker), su contrato de datos (datos de entrada y salida) y el formato del contrato (XML). Si el complemento se accede mediante SOAP, el registro también podría incluir un archivo WSDL (Web Services Definition Language).

Arquitectura de microservicios

Los microservicios son un estilo arquitectónico que permite desarrollar aplicaciones mediante servicios independientes, escalables y colaborativos, adaptándose a entornos complejos. Surgieron como alternativa a los sistemas monolíticos, que encapsulan toda la funcionalidad en una sola unidad lógica, compartiendo recursos y dificultando la escalabilidad, flexibilidad y mantenimiento. Entre sus principales desventajas, los sistemas monolíticos presentan falta de modularidad, donde cualquier cambio afecta toda la aplicación; dificultad para escalar, ya que no es posible aumentar la capacidad de forma parcial; complejidad en el despliegue, pues las actualizaciones requieren redistribuir todo el sistema, generando tiempos de inactividad; y baja adaptabilidad a nuevas tecnologías, lo que dificulta la integración con servicios modernos [8].

En la Fig. 1.5 cada componente de la arquitectura de microservicios se implementa como una unidad separada, lo que permite un despliegue más sencillo mediante un pipeline de entrega eficiente y optimizado, mayor escalabilidad y un alto grado de desacoplamiento entre la aplicación y sus componentes [7]. Un

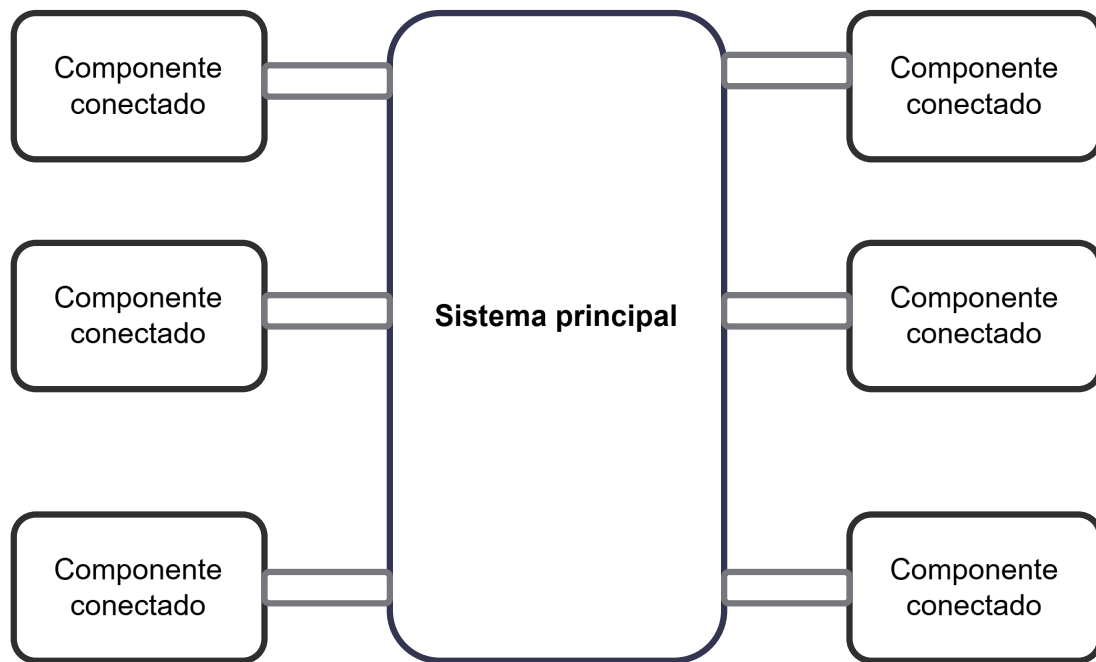


Fig. 1.4. Arquitectura microkernel

concepto clave en esta arquitectura es el “componente de servicio”. En lugar de pensar en servicios como unidades dentro de una arquitectura de microservicios, es más adecuado pensar en componentes de servicio, que pueden variar en su granularidad, desde un solo módulo hasta una gran parte de la aplicación. Los componentes de servicio contienen uno o más módulos (como clases Java) que representan funciones específicas (por ejemplo, proporcionar el clima de una ciudad) o partes independientes de una aplicación empresarial grande (como el registro de transacciones de acciones o la determinación de tarifas de seguros de automóviles). Diseñar la granularidad adecuada de los componentes de servicio es uno de los mayores desafíos dentro de una arquitectura de microservicios, lo cual se discute más a fondo en la subsección de orquestación de componentes de servicio.

Arquitectura basada en el espacio

La mayoría de aplicaciones web corporativas siguen un flujo común: una solicitud desde un navegador llega al servidor web, luego al servidor de aplicaciones y finalmente a la base de datos. Este patrón funciona bien para poco tráfico, pero los cuellos de botella empiezan a aparecer a medida que crece el número de usuarios, primero en el servidor web, luego en el de aplicaciones y después en la base de datos. Lo habitual para solucionar los cuellos de botella provocados por más usuarios es escalar verticalmente agregando más recursos a los servidores web. Esto es relativamente fácil y barato, y a veces resuelve el problema. Pero en situaciones de mucho tráfico, escalar los servidores web solo traslada el cuello de botella al servidor de aplicaciones.

La Fig. 1.6 ilustra el patrón básico de arquitectura basado en espacio y sus componentes arquitectónicos principales. El componente de unidad de procesamiento contiene los componentes de la aplicación (o porciones de ellos). Esto incluye componentes basados en la web, así como lógica de negocio de backend [7]. El contenido de la unidad de procesamiento varía según el tipo de aplicación. Aplicaciones web más pequeñas probablemente se desplegarían en una sola unidad de procesamiento, mientras que aplicaciones más grandes podrían dividir la funcionalidad en varias unidades de procesamiento basadas en áreas funcionales. La unidad de procesamiento generalmente contiene los módulos de la aplicación, junto con una cuadrícula de datos en memoria y un almacenamiento persistente asíncrono opcional para conmutación por error. También incluye un motor de replicación que es utilizado por el middleware virtualizado para

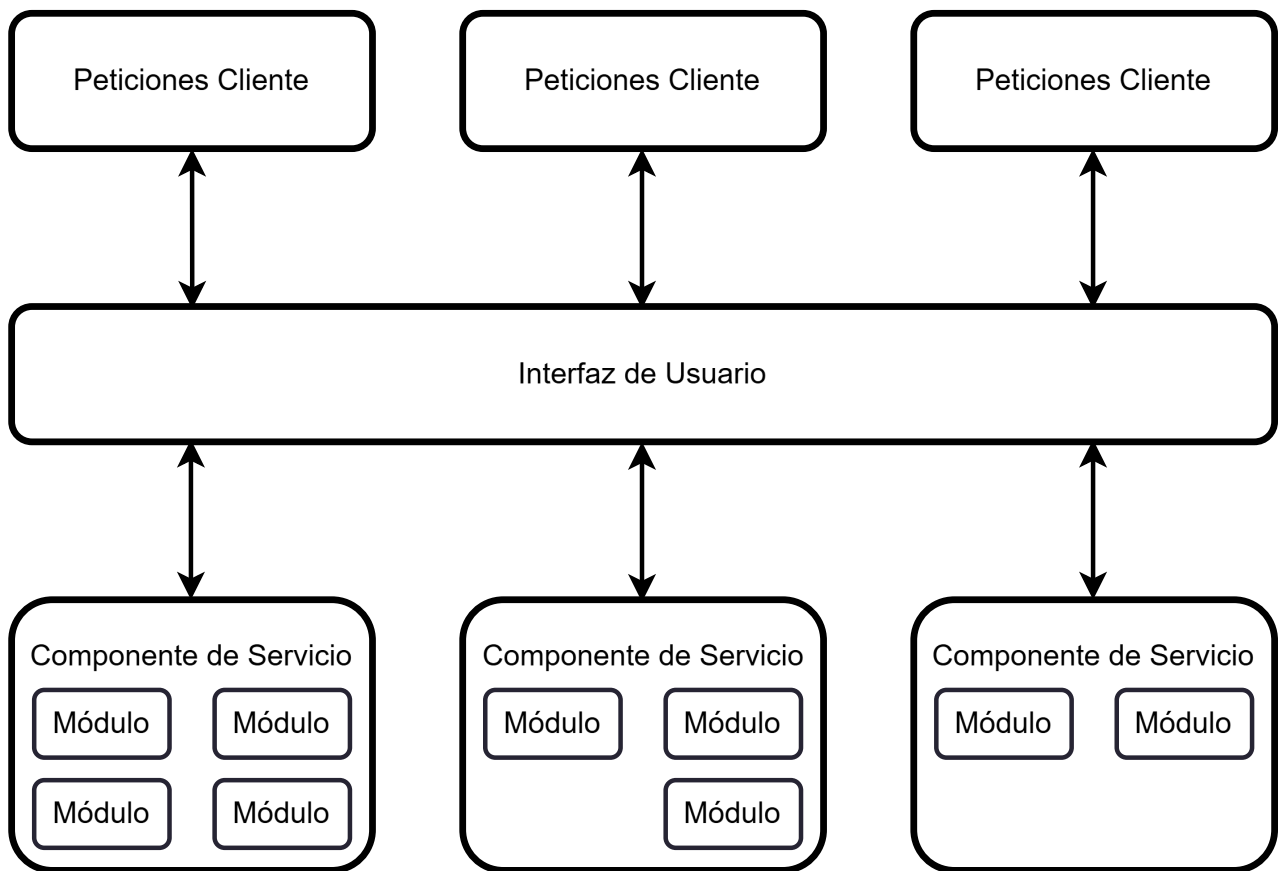


Fig. 1.5. Arquitectura de microservicios

replicar los cambios de datos realizados por una unidad de procesamiento a otras unidades activas.

Microservicios

Los microservicios constituyen un enfoque innovador desde una perspectiva tanto arquitectónica como organizacional en el desarrollo del software empresarial. Este paradigma se caracteriza por la descomposición de aplicaciones monolíticas tradicionales en conjuntos de pequeños servicios independientes, los cuales se enfocan en realizar una función de negocio específica. A diferencia de un modelo monolítico, en el cual toda la lógica de la aplicación reside desplegada en una sola unidad, la arquitectura de microservicios permite subdividir una aplicación en servicios de propósito único e independientes entre sí. Estos servicios se comunican entre ellos a través de APIs claramente definidas y contratos interservicios. Algunos beneficios clave de este estilo arquitectónico son: equipos más pequeños y ágiles enfocados en un solo servicio, despliegues independientes sin impactar la aplicación como conjunto, escalado específico solo donde se necesite, mayor resiliencia al confinar las fallas, capacidad de usar la tecnología más adecuada para cada microservicio y mayor velocidad de innovación. Sin embargo, una arquitectura de microservicios bien implementada requiere prácticas avanzadas de integración y entrega continuas, monitorización, descentralización de los datos, automatización de infraestructura y pruebas robustas, entre otros aspectos clave [9].

Una de las principales ventajas que ofrece la arquitectura de microservicios es la autonomía que permite entregar a los equipos responsables de cada microservicio. A diferencia de un enfoque monolítico tradicional, en donde un solo equipo grande debe encargarse del desarrollo y mantenimiento de toda una aplicación. Los microservicios promueven que cada pieza pequeña de la funcionalidad global sea gestionada por un equipo independiente. Esto posibilita que se formen equipos pequeños, ágiles y enfocados, con capacidad

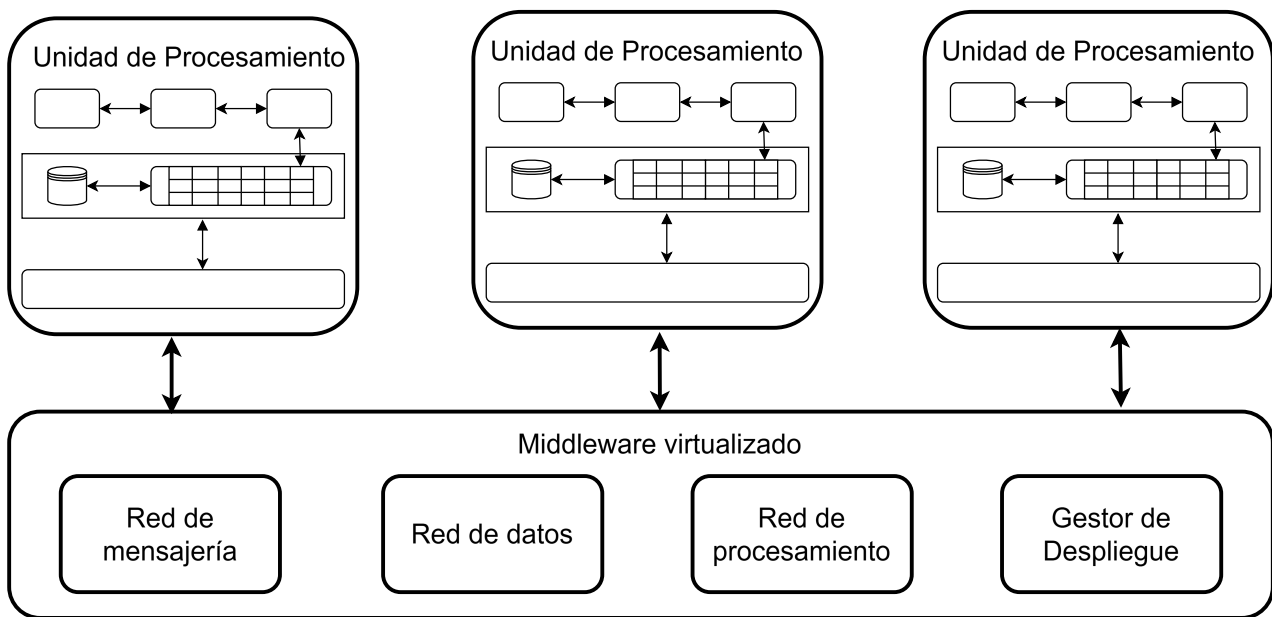


Fig. 1.6. Arquitectura basada en el espacio

de tomar decisiones rápidas y eficientes sobre un único servicio. Al estar aislados del resto de la aplicación, estos equipos tienen la libertad de elegir las tecnologías y lenguajes de programación adecuados para implementar las necesidades específicas de su propio microservicio. Esto fomenta la diversidad tecnológica, la innovación y la adaptabilidad; En contraste con una aplicación monolítica que obliga a utilizar una misma pila tecnológica en todos los componentes. La arquitectura de microservicios promueve construir cada pieza con la tecnología idónea según su propósito, sin acoplamientos ni restricciones por decisiones tomadas en otros servicios. Esto optimiza el desarrollo y desempeño del desarrollo del software. [9].

Características de los microservicios

■ Autonomía

Una característica distintiva clave de la arquitectura de microservicios es que cada componente que implementa una función específica se construye como un servicio independiente, el cual puede ser desarrollado, desplegado, operado y escalado de forma aislada, sin que sus procesos o decisiones de ciclo de vida afecten al resto de servicios con los que compone la aplicación final; [8] A diferencia de un enfoque monolítico tradicional, en donde todo el código fuente se comparte en un solo proyecto, en la arquitectura de microservicios cada equipo es dueño de un código fuente aislado que no necesita ser compartido con ningún otro equipo o microservicio. Esto promueve el desacoplamiento y la modularidad. Cada microservicio expone sus funcionalidades a través de interfaces bien definidas como API REST, mientras que su implementación interna y tecnologías son un detalle oculto al resto de la aplicación. Esto brinda libertad para adaptar el stack tecnológico según requerimientos técnicos y de negocio específicos.

■ Especializados

Una característica esencial de la arquitectura de microservicios es el meticuloso diseño orientado que debe realizarse para cada microservicio [8]. A diferencia de un monolito, donde toda la aplicación está contenida en una sola pieza, en esta arquitectura cada servicio está pensado y delimitado desde el principio para resolver una capacidad específica y acotada. El objetivo detrás de este enfoque es que cada microservicio aborde de forma óptima y enfocada un conjunto concreto y minimalista de funcionalidades, las cuales contribuyen de forma agregada a resolver una necesidad más amplia del negocio o sistema global. Se busca intencionalmente que cada servicio resuelva un solo problema muy bien definido. Esta

meticulosa delimitación de propósito, funciones y responsabilidades para cada pieza promueve servicios altamente cohesivos y desacoplados entre sí, características críticas para desarrollar sistemas modulares, escalables y fáciles de mantener, permitiendo que cada equipo resuelva de forma focalizada objetivos específicos, el todo se vuelve más que la suma de las partes, contribuyendo a la eficiencia y productividad en el desarrollo de aplicaciones empresariales complejas. Así, el diseño es absolutamente clave para maximizar los beneficios de esta arquitectura.

■ Flexibilidad tecnológica

Un aspecto diferenciador fundamental que distingue a la arquitectura de microservicios frente a enfoques monolíticos es la capacidad que tiene cada microservicio de ser construido de forma independiente, pudiendo elegir el lenguaje de programación y las tecnologías más apropiadas para las necesidades específicas de ese servicio, sin ataduras ni restricciones debido a decisiones tecnológicas tomadas en otros componentes. [8]

■ Diseño de unidad

Una ventaja importante que provee la arquitectura de microservicios a los equipos de desarrollo es la flexibilidad para elegir el lenguaje de programación y stack tecnológico más adecuado para implementar cada microservicio de forma individualizada. A diferencia de las aplicaciones monolíticas tradicionales, donde usualmente se adopta de forma estándar un mismo lenguaje y framework para todos los componentes por razones de uniformidad técnica, los microservicios permiten evaluar las necesidades específicas de cada pieza de la funcionalidad global y seleccionar las herramientas idóneas para desarrollar ese preciso microservicio, sin depender de decisiones tomadas en otros equipos. Esta diversidad tecnológica es invaluable en un contexto de aplicaciones empresariales modernas, donde diferentes microservicios pueden tener requisitos distintos de rendimiento, escalabilidad, uso de memoria o procesamiento en función de la especificidad de cada capacidad de negocio. Permitir la adopción de la tecnología más apropiada según cada necesidad agiliza y optimiza el desarrollo [8].

■ Reutilización de funcionalidad

A diferencia del enfoque de componentes fuertemente acoplados en aplicaciones monolíticas tradicionales, los microservicios están deliberadamente desacoplados entre sí, interactuando únicamente a través de estas interfaces. Este diseño orientado a servicios fuerza por convención que toda comunicación y colaboración entre microservicios se lleve a cabo mediante llamados estandarizados a APIs diseñadas intencionalmente para proveer los recursos e información necesarios de ese servicio. Estas APIs expuestas con protocolos universales y autodescriptivos como REST permiten automatizar la interacción entre microservicios de forma simple, confiable y eficiente, habilitando agilidad en el desarrollo de aplicaciones empresariales. Además, facilitan la reutilización de funcionalidades y la evolución independiente de los servicios [8].

■ Despliegue

Una capacidad distintiva central de la arquitectura de microservicios es poder realizar despliegues de nuevas versiones de cada microservicio en producción de forma completamente independiente, sin que ello repercuta o tenga impacto alguno en el funcionamiento del resto de microservicios con los que compone la aplicación final. En contraste de las aplicaciones monolíticas tradicionales, donde cualquier cambio o actualización involucra reconstruir y desplegar la aplicación entera, la naturaleza modular de los microservicios permite implementar mejoras o arreglos sobre microservicios específicos e inclusive introducir nuevas piezas de funcionalidad incrementales, sin necesidad de modificar nada en los demás servicios. Esta flexibilidad sin precedentes para desplegar cada componente por separado facilita enormemente las iteraciones y la innovación continua al evolucionar una aplicación empresarial. Permite agilizar la entrega de valor sin riesgos de regresión [8].

■ Mantenimiento

Varias cualidades distintivas del enfoque de microservicios, como su simplicidad, focalización funcional y capacidad de ser gestionados de forma aislada, se combinan para aportar beneficios significativos en cuanto al mantenimiento de aplicaciones a lo largo del tiempo [8].

■ Implementación sencilla

La implementación de microservicios constituye una estrategia avanzada y altamente eficiente en el desarrollo del software, al ofrecer una arquitectura modular que promueve la flexibilidad y la escalabilidad.[8]

■ Resistencia

La resistencia se convierte en un pilar esencial al abrazar la arquitectura de microservicios, desempeñando un papel crucial en la capacidad de la aplicación para mantener su funcionamiento óptimo incluso en condiciones adversas [8]. Esta resistencia se deriva en gran medida de la independencia inherente de cada servicio dentro de la arquitectura de microservicios, lo que contribuye significativamente a la robustez y estabilidad del sistema en su conjunto. En contraste con una arquitectura monolítica, donde un fallo en un único componente puede desencadenar consecuencias catastróficas en toda la aplicación. Los microservicios ofrecen una mayor tolerancia a los fallos. La modularidad de los microservicios permite que cada uno opere de manera autónoma, lo que significa que un fallo en un servicio no necesariamente afectará al funcionamiento de otros servicios dentro del software.

Esta capacidad de aislar y contener fallos limita su impacto y ayuda a prevenir la propagación de problemas en toda la aplicación. Además, la independencia de los microservicios facilita la implementación de estrategias de recuperación ante fallos más efectivas. Los equipos de desarrollo pueden diseñar cada servicio con mecanismos de autorecuperación y redundancia incorporados, lo que les permite detectar y mitigar rápidamente los problemas sin afectar el funcionamiento global de la aplicación. Esta capacidad de respuesta rápida y adaptativa es fundamental para garantizar la disponibilidad continua de la aplicación, incluso en situaciones de alto estrés o condiciones imprevistas. Otra ventaja de la arquitectura de microservicios en términos de resistencia es su capacidad para escalar de manera más eficiente. Al poder escalar cada servicio de forma independiente según sea necesario, los equipos pueden distribuir la carga de trabajo de manera más equitativa y mitigar el riesgo de sobrecarga en cualquier punto del software. Esto no solo mejora el rendimiento general de la aplicación, sino que también reduce la probabilidad de fallos debido a la saturación de recursos.

La Tabla I muestra cada una de las características que se mencionaron anteriormente sobre los microservicios.

FULLSTACK

El desarrollo fullstack originalmente se centraba en dominar las habilidades de programación necesarias para construir tanto el frontend (lado del cliente) como el backend (lado del servidor) de una aplicación web o sitio dinámico. Sin embargo, en la evolución reciente de la tecnología y la acelerada adopción de la nube, el significado de fullstack se ha expandido notablemente. Hoy en día, se espera que un desarrollador fullstack no sólo domine el desarrollo web en cliente y servidor, sino que también posea conocimientos más amplios relacionados con la infraestructura en la nube, contenedores, bases de datos, redes, almacenamiento, seguridad, monitorización, automatización y despliegue continuo.

En otras palabras, se ha pasado de un perfil centrado exclusivamente en habilidades de codificación a uno mucho más integral, que abarca todo el ciclo de vida del desarrollo de software moderno. Esto incluye aspectos como aprovisionamiento de infraestructura, operaciones en producción y mantenibilidad.

Algunas de las áreas clave que se deben dominar para ser un desarrollador fullstack en la actualidad son: nube pública (AWS, Azure, Google Cloud), contenedores y orquestación (Docker, Kubernetes), CI/CD, infraestructura como código (Terraform, CloudFormation) y monitoreo/logs (Prometheus, Grafana, ELK).[10].

TABLA I
Características de los microservicios

Características	
Autonomía	Se pueden desarrollar, implementar, operar y escalar sin afectar el funcionamiento de otros servicios.
Especializados	Cada servicio está diseñado para un conjunto de capacidades y se enfoca en resolver un problema específico.
Flexibilidad tecnológica	Se pueden desarrollar utilizando un lenguaje de programación que mejor se adapte al problema a resolver.
Diseño de unidad	La aplicación consiste en servicios poco acoplados. Cada servicio admite una única tarea de negocio.
Reutilización de funcionalidad	Los microservicios definen API que exponen su funcionalidad a cualquier cliente.
Despliegue	Cada microservicio se despliega de forma independiente, sin afectar a los otros microservicios de la aplicación.
Mantenimiento	Los microservicios son simples, enfocados e independientes. Así que la aplicación es más fácil de mantener.
Implementación sencilla	La implementación de microservicios ofrece una solución eficiente y flexible para el desarrollo del software.
Resistencia	La resistencia se traduce en la capacidad de la aplicación para mitigar y gestionar los errores de manera eficiente.

Stack

El término stack o pila tecnológica se refiere al conjunto integrado de lenguajes de programación, frameworks, librerías, herramientas y demás tecnologías que se utilizan para construir una aplicación informática. Un stack provee una aproximación estructurada y bien definida que permite a los desarrolladores aprovechar eficientemente un conjunto de herramientas y soluciones existentes, en lugar de tener que construir todo desde cero. De esta forma se agiliza y simplifica significativamente el proceso de desarrollo del software [10].

Backend

El backend de una aplicación juega un papel fundamental al gestionar la funcionalidad global de la aplicación web. Cuando los usuarios interactúan con el frontend, esta interacción se traduce en una solicitud al backend [11]. El backend utiliza una variedad de protocolos y tecnologías de comunicación para llevar a cabo eficientemente las solicitudes. Además, debe gestionar múltiples solicitudes simultáneamente, lo que implica implementar técnicas avanzadas como la distribución de solicitudes en varios servidores, el uso de almacenamiento en caché para optimizar el acceso a datos frecuentemente utilizados y la duplicación de datos para garantizar la disponibilidad y redundancia.

Frontend

El término frontend hace referencia a la interfaz gráfica de usuario (GUI) con la que los usuarios pueden interactuar de forma directa, como los menús de navegación, los elementos de diseño, los botones, las imágenes y los gráficos. En términos técnicos, una página o pantalla que el usuario ve con varios componentes de la interfaz de usuario se denomina modelo de objetos del documento (DOM) [11].

Ambos aspectos, frontend y backend, colaboran estrechamente para ofrecer una experiencia integral al

TABLA II

Diferencias entre backend y frontend

Diferencias	Frontend	Backend
Simultaneidad	Cada usuario tiene su propia copia de alguna aplicación, por lo que el frontend no tiene que gestionar los problemas de concurrencia.	El backend utiliza varias estrategias para gestionar miles de solicitudes de usuarios al mismo tiempo.
Almacenamiento en caché	Los navegadores o las aplicaciones cliente almacenan en caché los archivos de la aplicación y los utilizan para mejorar el rendimiento.	Los sistemas de backend almacenan en caché los archivos en diferentes servidores.
Seguridad	Los desarrolladores de frontend se centran principalmente en los flujos de trabajo de validación de entradas y autenticación de usuarios.	La seguridad en el desarrollo del backend se logra mediante el cifrado, los sistemas de autenticación seguros y las prácticas de codificación seguras.
Objetivos de desarrollo	El desarrollo del frontend se centra en crear interfaces de usuario totalmente funcionales, con buena capacidad de respuesta y bien diseñadas.	El desarrollo del backend implica la creación de una arquitectura fiable que respalde el desarrollo del frontend.
Habilidades de desarrollo	Los desarrolladores de frontend conocen HTML, CSS y JavaScript. Pueden utilizar marcos de frontend y crear páginas visualmente atractivas.	Los desarrolladores de backend cuentan con habilidades de codificación y administración de bases de datos.

usuario. La interfaz atractiva del frontend se comunica con el backend de manera eficiente, asegurando que las interacciones del usuario se traduzcan en operaciones lógicas y actualizaciones de datos. Esta interconexión sin problemas entre ambos componentes es esencial para el rendimiento global de la aplicación y la satisfacción del usuario, creando así un equilibrio armonioso entre la apariencia visual y la funcionalidad subyacente. En conjunto, el frontend y el backend desempeñan roles cruciales en la creación y el mantenimiento exitoso de aplicaciones modernas [11].

La Tabla II muestra las diferencias que existen entre el backend y el frontend.

APIS

Breve Historia

Las APIs (Interfaces de Programación de Aplicaciones) surgieron en los inicios de la informática, mucho antes de que las computadoras personales se convirtieran en dispositivos comunes en los hogares y oficinas. En sus primeras etapas, las APIs eran mayormente utilizadas como bibliotecas dentro de los sistemas operativos, proporcionándole a los desarrolladores un conjunto de funciones y rutinas para facilitar

la creación de aplicaciones dentro de un entorno local. En este período, las APIs estaban diseñadas para ser ejecutadas en las propias aplicaciones. Su uso se limitaba principalmente al funcionamiento interno de los sistemas, aunque ocasionalmente se utilizaban para permitir la transmisión de mensajes entre grandes computadoras centrales (mainframes), las cuales dominaban la infraestructura tecnológica de la época.

A lo largo del tiempo, y tras varios avances en la tecnología, las APIs comenzaron a expandirse más allá de estos entornos locales. Cerca de tres décadas después de su aparición, las APIs empezaron a ser más flexibles y accesibles, permitiendo que diferentes aplicaciones y servicios pudieran interactuar entre sí a través de redes más amplias. Esto sentó las bases para lo que serían futuras arquitecturas distribuidas. A principios del nuevo milenio, alrededor del año 2000, las APIs evolucionaron para convertirse en una tecnología esencial para la integración remota de datos, facilitando la comunicación entre sistemas que no compartían la misma infraestructura física. Esto permitió la creación de servicios basados en la web. Este cambio fue fundamental para el desarrollo de la arquitectura orientada a servicios (SOA) y, posteriormente, para la era de la web moderna, donde las APIs juegan un papel crucial en la interoperabilidad de aplicaciones distribuidas a través de internet. [12].

¿Qué es una API?

Las APIs son fundamentales en el mundo de la programación moderna, ya que proporcionan mecanismos claros y bien definidos para que diferentes componentes de software puedan comunicarse de manera efectiva. A través de un conjunto de definiciones, reglas y protocolos estandarizados, las APIs permiten que las aplicaciones interactúen de forma coherente y predecible, sin importar las diferencias tecnológicas o de plataforma entre ellas. Esta estandarización es clave, ya que reduce la complejidad y el tiempo de desarrollo al proporcionar una forma común de acceso a funcionalidades y servicios. En lugar de que los desarrolladores tengan que crear soluciones complejas desde cero para permitir que sus aplicaciones trabajen juntas. Las APIs actúan como "puentes" que conectan distintos sistemas, permitiendo el intercambio de información y la ejecución de tareas específicas entre ellos. Este enfoque no solo mejora la eficiencia, sino que también fomenta la interoperabilidad, facilitando la integración de servicios y la creación de sistemas más dinámicos y escalables. Las APIs permiten, por ejemplo, que una aplicación de compras en línea acceda a servicios de pago, que un sistema de gestión de clientes se conecte a herramientas de análisis de datos, o que diferentes plataformas móviles puedan interactuar con bases de datos de manera segura y eficiente. Además, las APIs no solo permiten el intercambio de datos, sino que también habilitan el acceso a funcionalidades completas, como procesamiento en la nube, servicios de almacenamiento, autenticación de usuarios, entre otros. Esto ha sido fundamental para la evolución de arquitecturas basadas en servicios, microservicios y sistemas distribuidos, donde las aplicaciones son desglosadas en pequeñas unidades funcionales que se comunican entre sí a través de APIs. De esta forma, las APIs contribuyen a la modularidad y flexibilidad, permitiendo que los desarrolladores creen aplicaciones que no solo sean más fáciles de mantener, sino también más adaptables a cambios y nuevas tecnologías [13].

¿Cómo funcionan las APIs?

La arquitectura de las APIs se describe comúnmente en términos de una relación cliente-servidor, donde el cliente realiza solicitudes y el servidor responde. En el contexto del ejemplo meteorológico, la base de datos del instituto actúa como el servidor, mientras que la aplicación móvil representa el cliente, solicitando y recibiendo datos meteorológicos [13].

Protocolos para APIs

El uso de las API ha ido en aumento. Se han ideado ciertos protocolos para ofrecer a los usuarios un conjunto de normas definidas, o especificaciones de API, que determinan tipos de datos, comandos y sintaxis aceptadas. Estos protocolos de API facilitan el intercambio estandarizado de información [13].

De acuerdo con IBM, se consideran como protocolos de APIs a los siguientes:

SOAP (Simple Object Access Protocol): SOAP, desarrollado con XML, permite que los puntos finales transmitan y reciban datos mediante protocolos como SMTP y HTTP. Las API basadas en SOAP facilitan el intercambio de información entre aplicaciones o componentes de software, incluso cuando operan en entornos distintos o están programados en diferentes lenguajes.

XML-RPC (Remote Procedure Call XML): El protocolo XML-RPC se basa en un formato XML específico para la transferencia de datos. Aunque XML-RPC es más antiguo que SOAP, es considerablemente más sencillo y relativamente ligero, ya que utiliza un ancho de banda mínimo.

JSON-RPC: Similar a XML-RPC, JSON-RPC es una llamada de procedimiento remoto, pero utiliza JSON (JavaScript Object Notation) en lugar de XML para la transferencia de datos.

REST (Representational State Transfer): REST es un conjunto de principios de arquitectura de API web. Las API REST (también conocidas como API RESTful) son APIs que siguen ciertas restricciones arquitectónicas REST. Es posible implementar API RESTful con protocolos SOAP, pero estos dos estándares suelen considerarse como especificaciones competidoras.

La Fig. 1.7 muestra las características más destacables que las APIs poseen.

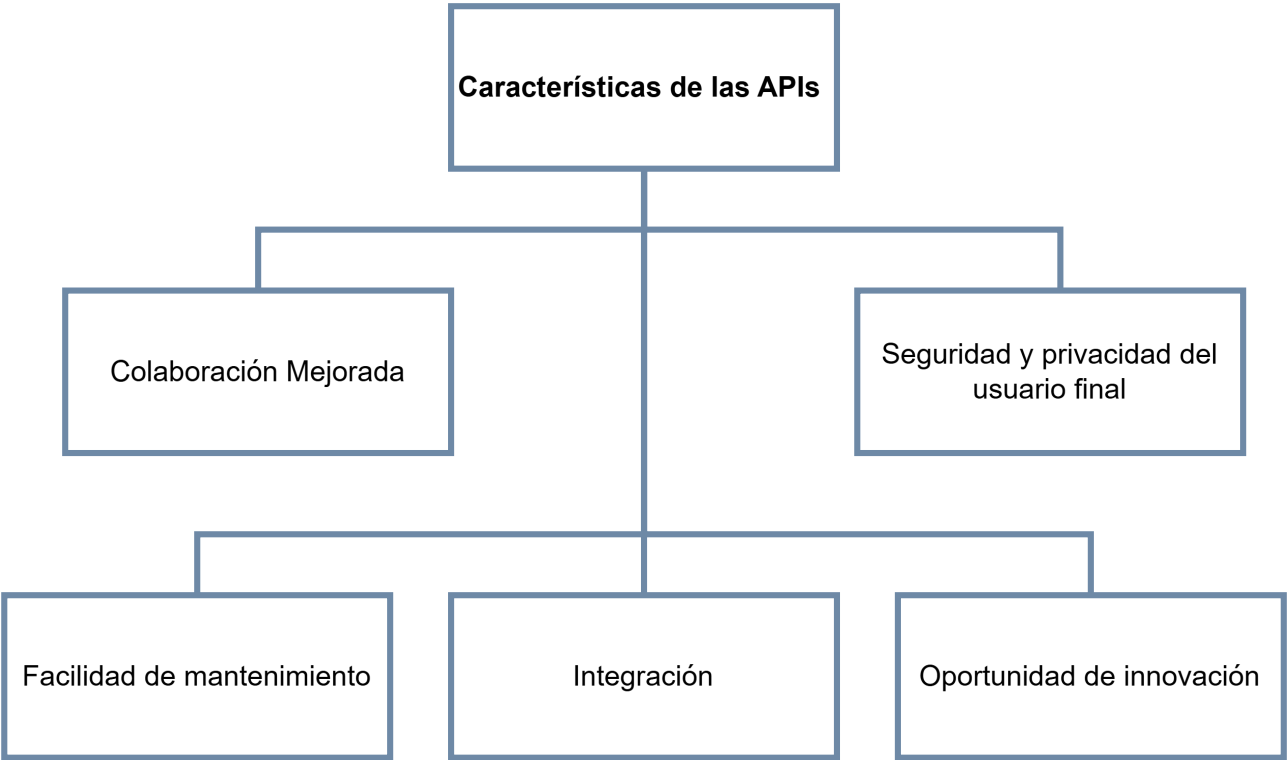


Fig. 1.7. Características destacadas de las APIs

ANTECEDENTES

La evolución en el desarrollo de servicios web ha sido marcada por la adopción de diversas tecnologías a lo largo del tiempo. Antes de que REST se convirtiera en el estándar predominante, otras arquitecturas y protocolos desempeñaron roles destacados en la creación de aplicaciones distribuidas. Desde el uso extensivo de SOAP y XML-RPC hasta arquitecturas como Common Object Request Broker Architecture (CORBA), el panorama de las tecnologías web ha experimentado una diversidad significativa. En este contexto, se exploran algunas de las alternativas que precedieron a REST, destacando sus características distintivas y su impacto en el desarrollo de sistemas distribuidos.

REST

REST, por sus siglas en inglés de Representational State Transfer, es un estilo arquitectónico para sistemas hipermedia distribuidos, creado por Roy Thomas Fielding en 2000. REST es un conjunto de conceptos de diseño basados en características y requisitos del software de red, que tiene como objetivo reducir la complejidad del desarrollo, mejorar la escalabilidad de los sistemas y reducir la carga de la comunicación.

Las limitaciones mencionadas han sido clave para el éxito de la arquitectura REST en la Web; no obstante, REST ha mostrado varios problemas, en donde GraphQL se presenta como una alternativa para mejorar varios de estos problemas. Por tal razón, a continuación, se describen las diferencias entre las características de estos dos paradigmas [14].

API REST

La API REST se caracteriza por utilizar los principios de diseño de REST, que es un estilo de arquitectura para sistemas de software distribuidos. Es comúnmente conocida como API RESTful debido a su conformidad con estos principios. En esencia, REST establece pautas para la creación de servicios web que sean escalables, flexibles y fáciles de mantener. Esto se logra mediante la manipulación de recursos a través de una interfaz uniforme y predefinida, utilizando métodos HTTP estándar como GET, POST, PUT y DELETE. Además, las API REST suelen ser stateless, lo que significa que no mantienen ningún estado de sesión entre las solicitudes del cliente, lo que las hace altamente escalables y adecuadas para entornos distribuidos y de alta concurrencia [15].

La Fig. 1.8 describe las características que poseen las APIs REST.

¿Cómo funcionan las APIs REST?

Cuando un cliente necesita acceder a un recurso, establece comunicación con el servidor a través de la API. Los desarrolladores de la API proporcionan instrucciones sobre cómo el cliente debe interactuar con la API REST en la documentación asociada al servidor de la aplicación [16]. A continuación, se describen los pasos generales para cualquier llamada a la API REST:

- El cliente envía una solicitud al servidor, siguiendo las pautas establecidas en la documentación de la API para formatear correctamente la solicitud.
- El servidor autentica al cliente y verifica que esté autorizado para realizar la solicitud.
- Una vez recibida, el servidor procesa la solicitud internamente.

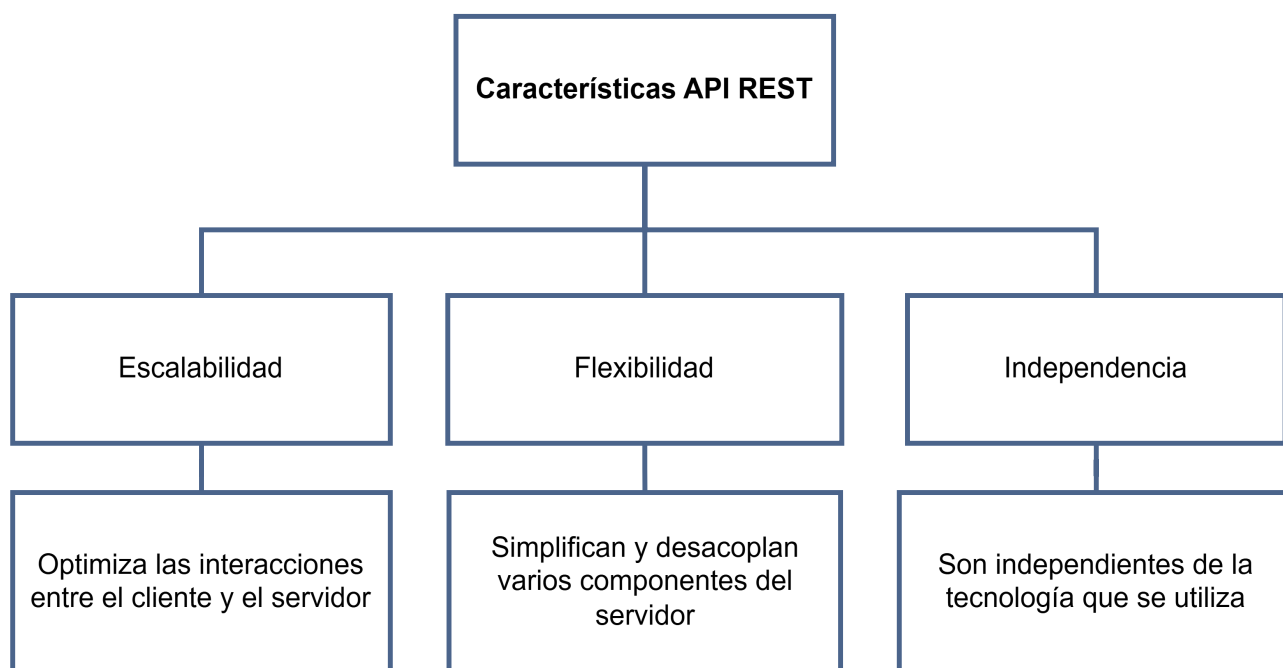


Fig. 1.8. Características API REST

- Posteriormente, envía una respuesta al cliente que indica si la solicitud se ha procesado correctamente, junto con cualquier información solicitada por el cliente.

Una parte importante dentro del funcionamiento de las APIs REST son los métodos HTTP que nos permiten realizar distintas acciones para obtener, modificar e incluso eliminar los que provienen de una fuente específica. A continuación se describen los principales métodos HTTP que se utilizan con las APIs REST.

GET.- Solicita al servidor un objeto, recurso o documento.

POST.- Envía una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.

PUT.- Permite actualizar la información existente; es similar a la sentencia UPDATE de las bases de datos.

DELETE.- Permite eliminar un recurso específico; por lo general es información existente. El funcionamiento de este método es similar a la sentencia DELETE de las bases de datos.

En la Tabla III se observa un ejemplo de las distintas peticiones que se pueden realizar mediante un API REST. El método GET permite obtener un listado de los productos mediante la ruta /products, el método POST permite crear nuevos productos, el método UPDATE permite actualizar la información de un producto existente y el método DELETE, como se observa en la tabla, permite eliminar un producto específico.

TABLA III
Métodos para peticiones REST

Método	Ruta	Resultado
GET	/products	Listar todos los productos
GET	/products/id	Listar un producto por su id
POST	/products	Crear un producto
PUT	/products/id	Actualizar un producto
DELETE	/products/id	Eliminar un producto

Nota: Es necesario mencionar que los métodos descritos anteriormente son los métodos comunes cuando se utilizan API REST para el desarrollo del backend.

Diferencias entre REST y GraphQL

Antes de mencionar las diferencias, se describen las similitudes entre estos patrones arquitectónicos. El sistema de tipos de GraphQL es similar a la definición de recursos en la arquitectura REST, donde no hay consultas ad hoc, pero se definen de antemano los tipos del sistema, especificando los posibles parámetros y el formato de salida. Tanto GraphQL como REST son independientes del lenguaje de programación subyacente y de los backends de bases de datos utilizados. La comunicación con el cliente se basa en un patrón de intercambio de mensajes petición-respuesta [14].

GraphQL surgió en 2012 como una solución a la necesidad de mejorar la velocidad en las plataformas de redes sociales emergentes. Los desarrolladores se dieron cuenta de que las arquitecturas de API existentes, como REST, resultaban demasiado verbosas y estructuradas para generar fuentes de noticias de manera eficiente [17].

A continuación, se describen las principales limitaciones que posee REST:

Intercambio de datos con estructura fija: La API de REST requiere que las solicitudes de los clientes sigan una estructura fija para recibir un recurso. Aunque esta estructura es fácil de usar, no siempre es el método más eficiente para intercambiar los datos únicamente necesarios.

Exceso e insuficiencia de datos: Las API de REST devuelven siempre un conjunto completo de datos. Por ejemplo, al solicitar un objeto de persona, recibirás su nombre, fecha de nacimiento, dirección y número de teléfono, incluso si solo necesitas el número de teléfono.

Entre las principales diferencias de REST con GraphQL se destacan las siguientes: REST es un estilo arquitectónico, y GraphQL es un lenguaje de consulta y tiempo de ejecución de APIs. La interfaz REST proporciona múltiples puntos finales (endpoints) de la API, uno para cada recurso; en el caso de GraphQL tiene un único punto final. REST utiliza generalmente las acciones POST, GET, PUT y DELETE de HTTP para manipular los datos; en cambio, GraphQL utiliza consultas (queries) y mutaciones (mutations) con la acción POST sobre su único endpoint. REST admite el almacenamiento en caché y documentos parciales; estas características, así como la libertad de elección con respecto a los formatos de intercambio de datos utilizados, no son posibles con GraphQL [14].

En el rendimiento de aplicaciones móviles existen diferencias notables entre GraphQL y REST. En el caso de REST, al depender de múltiples endpoints, genera over-fetching y under-fetching, lo que impacta el consumo de recursos y la latencia. En contraste, GraphQL permite recuperar solo los datos necesarios en una única consulta, reduciendo el tráfico de datos y optimizando el rendimiento [18].

En la Tabla IV se muestra un resumen de las diferencias entre las propiedades de REST y GraphQL.

TABLA IV
Diferencias entre las propiedades de REST y GraphQL

Propiedad	REST	GraphQL
Tecnología	Estilo arquitectónico	Lenguaje de consulta
Manipulación de datos	Métodos HTTP	queries, mutations
Puntos finales de la API	Múltiple	Único
Formato para las consultas	Basado en cadenas	Cadenas
Formato para las mutaciones	Cualquiera	Cadenas
Formato para la respuesta	Cualquiera	JSON
Sistema de tipos	Débilmente tipado	Fuertemente tipado

Diversas investigaciones han demostrado que, en determinadas condiciones, GraphQL puede ofrecer un mejor rendimiento que REST. A continuación, se presentan algunos casos que ilustran esta ventaja

Un estudio comparativo sobre APIs, publicado en Proceedings of International Conference on Information Technology and Applications, analizó las diferencias clave entre REST y GraphQL en términos de eficiencia en la recuperación de datos [19]. Por ejemplo, en una aplicación de comercio electrónico, una API REST necesitaría múltiples solicitudes para obtener información de un producto, su categoría y reseñas, lo que genera over-fetching (más datos de los necesarios) y under-fetching (varias llamadas para consolidar datos) [19]. En cambio, con GraphQL, un solo endpoint permite una consulta específica que devuelve únicamente los datos requeridos en una sola solicitud, reduciendo la latencia y optimizando el rendimiento. El estudio concluye que GraphQL es más eficiente y flexible, mientras que REST, aunque funcional, puede volverse ineficiente en consultas complejas. Además, se evaluó un GraphQL Wrapper sobre REST, una alternativa que mejora el rendimiento sin necesidad de reconstruir completamente una API REST existente [19].

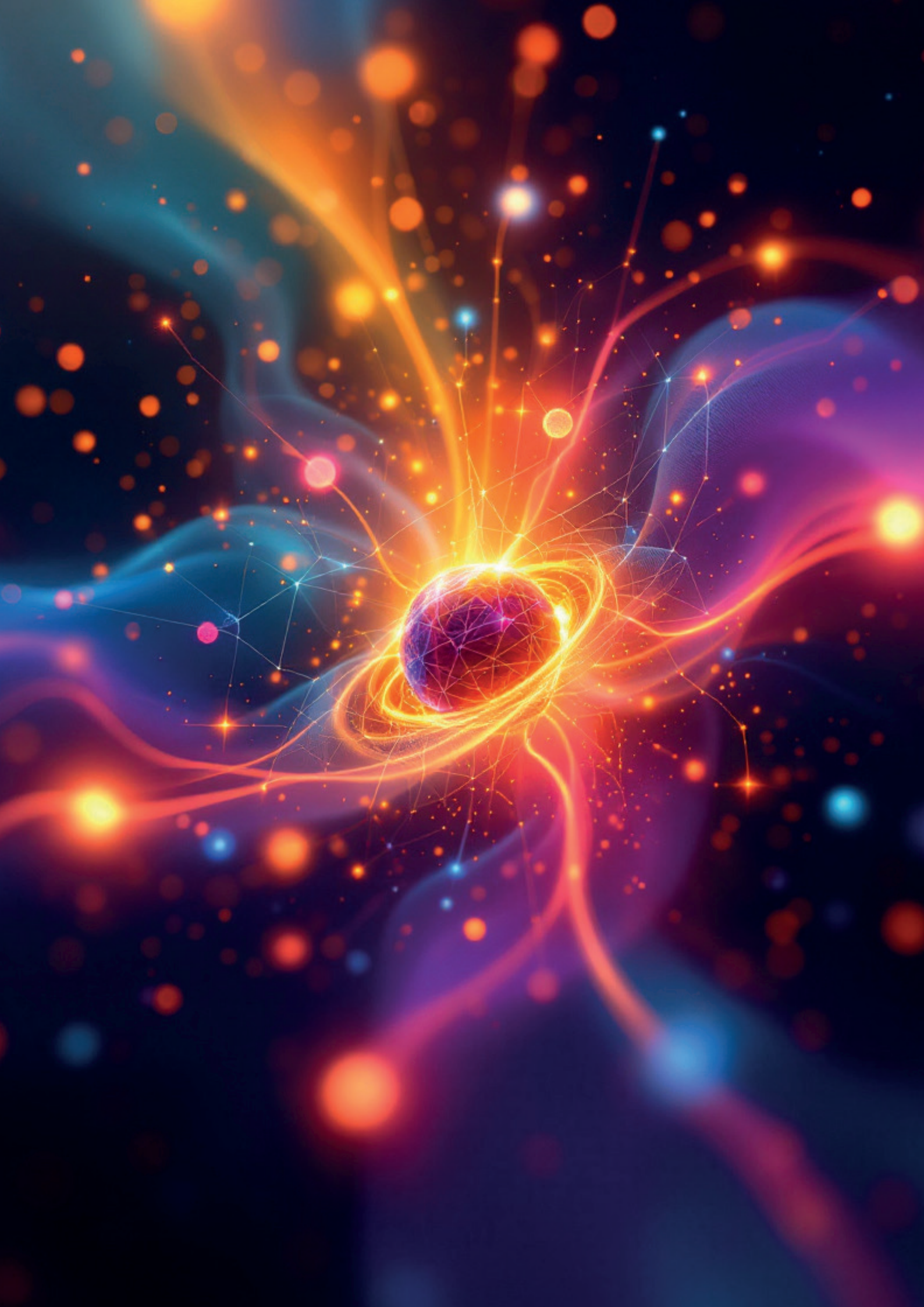
Otro estudio comparativo sobre la eficiencia de microservicios, publicado en Data and Metadata, analizó el desempeño de APIs GraphQL y REST en entornos Docker [20]. En un experimento computacional, se midió el tiempo de respuesta de ambas arquitecturas bajo distintas cargas de datos y niveles de complejidad. Los resultados mostraron que Docker proporciona un rendimiento significativamente superior, y que GraphQL supera a REST en consultas complejas al optimizar la recuperación de datos y reducir la latencia. Por ejemplo, en escenarios con múltiples relaciones en bases de datos, GraphQL permite recuperar datos de manera más eficiente con una única consulta, mientras que REST requiere múltiples solicitudes, generando sobrecarga en el procesamiento [20]. Sin embargo, en operaciones de inserción de datos dentro de Docker, REST obtuvo mejores tiempos de respuesta. El estudio concluye que la combinación de Docker y GraphQL es la opción más eficiente para arquitecturas de microservicios orientadas a consultas de datos, aunque REST sigue siendo útil en escenarios específicos [20].

RESUMEN

A lo largo de este capítulo se subraya la relevancia de la arquitectura de software como un pilar esencial en el desarrollo de aplicaciones modernas. Se enfatiza cómo los diferentes patrones arquitectónicos han evolucionado a lo largo del tiempo para responder a los desafíos que plantea la construcción de sistemas cada vez más complejos, tales como la escalabilidad, el mantenimiento y la eficiencia. Este proceso de evolución no solo refleja la adaptación de las prácticas de ingeniería de software a las demandas del mercado y los usuarios, sino también la búsqueda constante de soluciones que garanticen la sostenibilidad de los proyectos en el largo plazo.

En este contexto, la arquitectura de microservicios se consolida como una de las propuestas más robustas y versátiles. Su capacidad para dividir un sistema en componentes independientes permite mejorar la resiliencia, facilitar el despliegue continuo y ofrecer una mayor flexibilidad tecnológica al integrar diferentes lenguajes y herramientas dentro de un mismo ecosistema. Este capítulo resalta que estas características convierten a los microservicios en una alternativa que responde de manera efectiva a las necesidades de organizaciones que requieren aplicaciones escalables, distribuidas y fáciles de mantener en entornos cambiantes.

Del mismo modo, se destaca la importancia del desarrollo FullStack como un enfoque integral que unifica backend y frontend, favoreciendo una visión holística del ciclo de vida de las aplicaciones. En este marco, las APIs se presentan como piezas clave para mediar en la comunicación entre ambas capas, garantizando una interacción fluida y eficiente. Finalmente, el capítulo plantea una comparación entre REST y GraphQL, donde este último se posiciona como una solución innovadora al superar las limitaciones tradicionales, proporcionando un acceso más preciso, eficiente y adaptable a la información, en consonancia con las demandas actuales de flexibilidad y personalización en el desarrollo de software.



CAPÍTULO 2

Fundamentos de GraphQL

«Un buen programador es alguien que siempre mira en ambas direcciones antes de cruzar una calle de un solo sentido.»

Raymond, E. S., 1999, p. 47.

GraphQL es un lenguaje de consulta y motor de ejecución que está transformando la forma en que las aplicaciones cliente-servidor interactúan y comunican datos. A diferencia de las arquitecturas tradicionales como REST, GraphQL ofrece un enfoque flexible, eficiente y declarativo para describir las capacidades y requisitos de modelos de datos. Este paradigma permite a los clientes solicitar exactamente los datos que necesitan, sin redundancias innecesarias, al tiempo que facilita la evolución continua de las API para adaptarse a las necesidades cambiantes. Entre sus componentes más robustos están el propio lenguaje GraphQL, el sistema de tipos, las operaciones ejecutables, la introspección y los resolvers. GraphQL brinda a los desarrolladores una forma estandarizada e independiente de la implementación de construir, depurar y mantener sistemas de manera efectiva.

COMPONENTES PRINCIPALES DE GRAPHQL**Lenguaje GraphQL**

GraphQL se define como un lenguaje de consulta y un motor de ejecución que permite describir las capacidades y requisitos de modelos de datos en aplicaciones cliente-servidor [21]. En esencia, GraphQL proporciona una manera flexible y eficiente de comunicarse entre el cliente y el servidor sin la necesidad de atarse a un lenguaje de programación o un mecanismo de persistencia específicos. Esto significa que, al implementar servidores de aplicaciones GraphQL, no es necesario utilizar un conjunto particular de herramientas; en su lugar, GraphQL permite codificar las capacidades del modelo de datos basándose en un sistema de tipos utilizando un lenguaje uniforme y coherente. Esta capacidad de describir las capacidades del modelo de datos de manera independiente de la implementación subyacente hace que GraphQL sea adaptable e integrable en una amplia variedad de sistemas y tecnologías.

GraphQL ofrece una descripción exhaustiva y fácilmente comprensible de los datos disponibles en su API, lo que permite a los clientes solicitar precisamente lo que necesitan sin redundancias innecesarias. Además, facilita la evolución continua de las API a lo largo del tiempo, adaptándose a las necesidades cambiantes de los usuarios y del sistema. GraphQL proporciona un conjunto de herramientas poderosas y flexibles para los desarrolladores, lo que les permite construir, depurar y mantener sistemas de manera eficiente y efectiva [22].

El lenguaje GraphQL emplea una gramática sintáctica para establecer un sistema de tipos o para realizar consultas a un servicio GraphQL mediante documentos. Estos documentos GraphQL pueden incluir varias instancias de definiciones, como definiciones ejecutables o de extensión de sistema de tipos.

La Fig. 2.1 ilustra el proceso de instanciación del paradigma GraphQL en los siguientes pasos: 1) las herramientas utilizadas durante el desarrollo emplean el lenguaje GraphQL, su gramática y el lenguaje de descripción de interfaces (IDL) para establecer un sistema de tipos o una extensión del mismo para un servicio GraphQL. 2) La creación del servicio del API GraphQL se complementa con la implementación de *resolvers* en algún lenguaje de programación para devolver las propiedades definidas en el sistema de tipos. 3) Desde las herramientas del cliente, se elabora un documento que contiene definiciones de operaciones ejecutables como query, mutation, subscription o fragments para enviar una solicitud de ejecución al servicio GraphQL. Antes de enviar la solicitud, las herramientas del cliente llevan a cabo una validación de la sintaxis utilizando la introspección del servicio GraphQL (indicada con un círculo entrecortado con la palabra “INT” en la sección de “Herramientas del cliente”). El servicio GraphQL recibe la petición en el motor de validación para asegurar que el documento sólo incluya definiciones ejecutables y que la solicitud esté libre de errores. 4) El servicio GraphQL devuelve la respuesta de la ejecución.

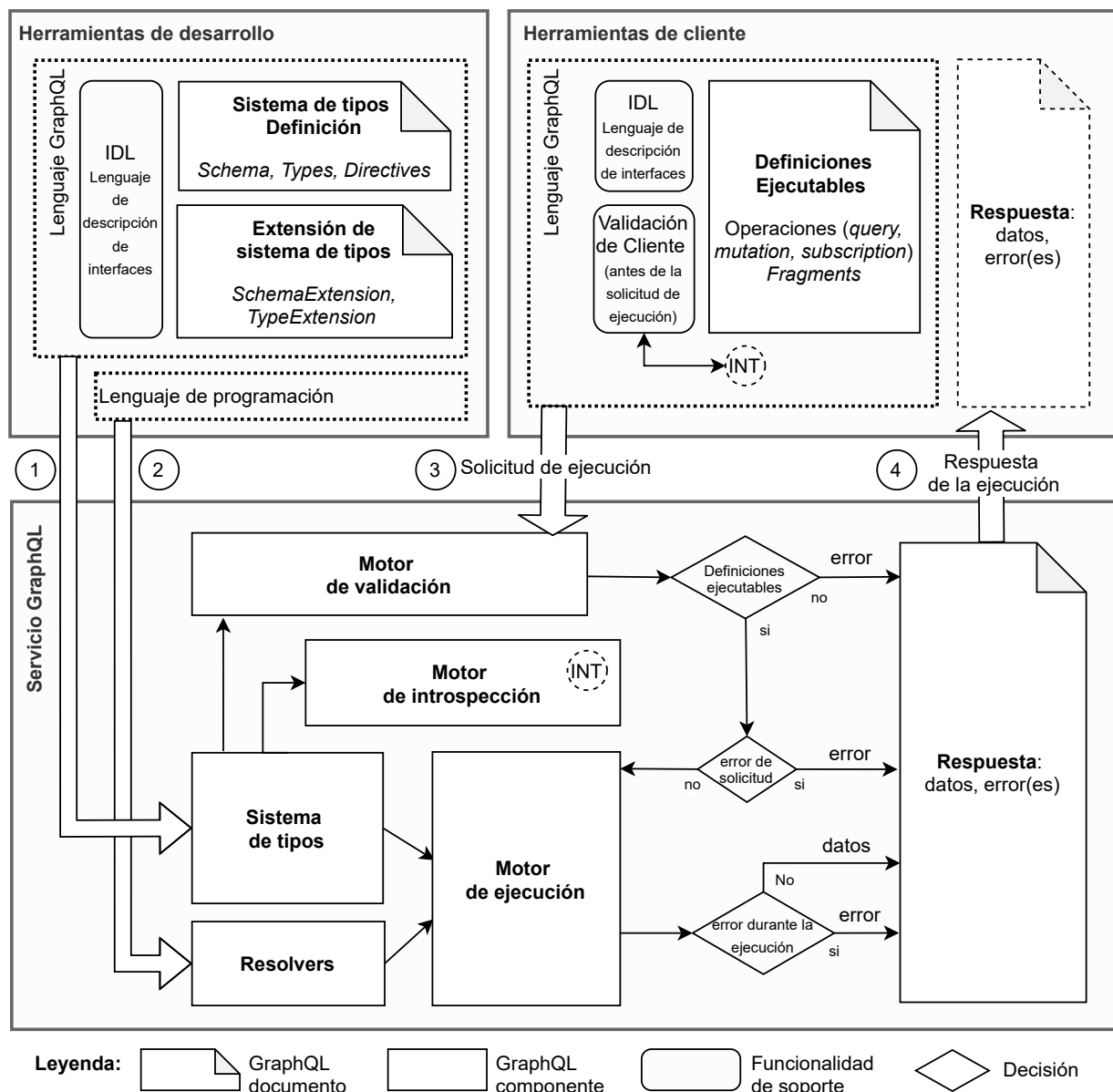


Fig. 2.1. Componentes del paradigma GraphQL

Sistema de tipos (Type System)

El sistema de tipos en un servidor GraphQL define las capacidades fundamentales de los datos que pueden ser manejados a través de la API. Este sistema de tipos proporciona una representación precisa y estructurada de los datos disponibles, permitiendo que los clientes comprendan y utilicen eficazmente la API [21].

GraphQL proporciona una serie de tipos básicos y personalizados que ayudan a definir la estructura de la información. Entre estos tipos se encuentran los escalares, los objetos, las listas y las interfaces. Además, la capacidad de definir relaciones entre los tipos permite modelar datos de una manera clara y organizada, facilitando su consulta y manipulación.

El uso de tipos de entrada para las variables de consulta otorga a los clientes flexibilidad en la forma en que envían datos al servidor. Gracias a esta estructura, es posible realizar solicitudes más precisas y reutilizables, lo que mejora la eficiencia y la claridad en la comunicación entre el cliente y el servidor.

A continuación, se muestra el código fuente 2.1 donde se presenta un ejemplo de definición de tipos en GraphQL para la gestión usuarios y publicaciones:

```
type Usuario {  
  id: ID!  
  nombre: String!  
  email: String!  
  publicaciones: [Publicacion]  
}  
  
type Publicacion {  
  id: ID!  
  titulo: String!  
  contenido: String!  
  autor: Usuario!  
}  
  
input NuevaPublicacionInput {  
  titulo: String!  
  contenido: String!  
  autorId: ID!  
}  
  
type Mutation {  
  crearPublicacion(input: NuevaPublicacionInput!): Publicacion  
}  
  
type query {  
  usuario(id: $id): Usuario  
}
```

Código Fuente 2.1: Definición de tipos en GraphQL

En este esquema de GraphQL, se definen los siguientes tipos:

- **Usuario:** Representa a un usuario del sistema con los campos `id` (identificador único), `nombre` (nombre del usuario), `email` (correo electrónico) y `publicaciones` (una lista de publicaciones asociadas a ese usuario).
- **Publicacion:** Define una publicación con los campos `id`, `titulo`, `contenido` y `autor` (referencia al usuario que creó la publicación).
- **NuevaPublicacionInput:** Es un tipo de entrada que permite recibir datos al momento de crear una publicación. Incluye `titulo`, `contenido` y `autorId` (el identificador del usuario que la escribe).
- **Mutation:** Define la mutación `crearPublicacion`, la cual permite agregar una nueva publicación al sistema tomando como entrada un objeto del tipo `NuevaPublicacionInput`.
- **Query:** Permite recuperar información sobre un usuario en función de su `id`, devolviendo el objeto `Usuario` con sus respectivas publicaciones.

Introspección (Introspection)

La introspección en GraphQL es una funcionalidad esencial que permite a los clientes obtener información detallada sobre el esquema del servidor. Mediante esta capacidad, un cliente puede consultar la API GraphQL para conocer los tipos de datos disponibles, los campos de cada tipo, las relaciones entre ellos y las operaciones admitidas. La introspección se realiza a través de consultas especiales dentro del propio lenguaje GraphQL, lo que permite explorar la estructura del esquema sin depender de documentación externa [21].

Esta característica es especialmente útil para desarrolladores que trabajan con APIs GraphQL, ya que facilita la comprensión de la estructura del sistema sin necesidad de acceder al código fuente o a archivos de especificaciones. Gracias a la introspección, es posible generar documentación automática de la API, detectar cambios en el esquema y construir herramientas de desarrollo más eficientes.

Las herramientas como GraphiQL, GraphQL Playground y Apollo Studio aprovechan la introspección para proporcionar entornos interactivos donde los desarrolladores pueden examinar y probar la API de manera intuitiva. Estas herramientas muestran en tiempo real la estructura de los tipos, consultas y mutaciones disponibles, lo que reduce significativamente la curva de aprendizaje y mejora la experiencia de desarrollo.

Además, la introspección es clave para la interoperabilidad entre diferentes servicios y clientes, ya que permite que las aplicaciones descubran automáticamente los recursos disponibles en un servidor GraphQL. Esto facilita la integración con frameworks de frontend y backend, así como con herramientas de generación de código fuente y validación de esquemas.

A continuación, se muestra un ejemplo de una consulta GraphQL de introspección que permite obtener información sobre los tipos definidos en el servidor; ver código fuente 2.2:

```
query Introspeccion {
  __schema {
    types {
      name
      kind
      fields {
        name
        type {
          name

```

```
      kind
    }
  }
}
```

Código Fuente 2.2: Consulta de introspección en GraphQL

Esta consulta devuelve una lista de los tipos definidos en el esquema, junto con sus nombres, categorías (por ejemplo, objeto, escalar o interfaz) y los campos que contienen. La introspección también puede utilizarse para obtener información sobre las consultas y mutaciones disponibles, los argumentos que aceptan y los valores de retorno que generan.

Validación (Validation)

La validación en GraphQL es un proceso fundamental que garantiza que una solicitud sea correcta antes de su ejecución. Este proceso verifica que la consulta esté bien formada, respetando la sintaxis y las reglas del esquema GraphQL. La validación se realiza automáticamente antes de ejecutar la solicitud, asegurando que todos los tipos, campos y argumentos utilizados en la consulta existan y sean compatibles con la estructura definida en el esquema [21].

Un servicio GraphQL puede omitir la validación si tiene la certeza de que una solicitud ha sido verificada previamente, como en casos donde la misma consulta se reutiliza constantemente y ha sido validada en un paso anterior. Sin embargo, ejecutar una solicitud sin validarla puede generar errores inesperados si la estructura de la API cambia o si la consulta contiene errores.

Es importante destacar que las definiciones del sistema de tipos y sus extensiones en GraphQL no son ejecutables. Esto significa que, aunque un esquema defina reglas y relaciones entre tipos, estas definiciones no afectan la ejecución directa de las consultas. La validación también permite optimizar el rendimiento del servicio al evitar la ejecución de consultas inválidas y proporcionar retroalimentación inmediata a los clientes, lo que facilita la depuración y el desarrollo eficiente de aplicaciones GraphQL.

Ejecución (Execution)

La ejecución en GraphQL es un proceso clave en el ciclo de vida de una consulta, ya que se encarga de evaluar y resolver las operaciones solicitadas por los clientes. Este proceso se basa exclusivamente en las definiciones ejecutables del esquema, como consultas, mutaciones y fragmentos, ignorando cualquier elemento que no esté diseñado para su ejecución directa [21].

Cuando un cliente envía una solicitud GraphQL, el servidor la procesa en el contexto del sistema de tipos definido en su esquema. Si la consulta ha pasado previamente las validaciones sintácticas y semánticas, el motor de ejecución resuelve cada campo solicitado accediendo a los datos correspondientes en la fuente de datos del servidor. En caso de que la solicitud contenga errores de validación, estos se incluyen en la lista de errores de la respuesta, y la consulta no se ejecuta.

Es importante señalar que, aunque una consulta pase la validación, pueden ocurrir errores durante la ejecución. Estos errores pueden deberse a problemas como la falta de datos en la base de datos, restricciones de acceso o fallos en servicios externos. En estos casos, GraphQL devuelve los datos que pudieron resolverse correctamente junto con un informe detallado de los errores encontrados, lo que permite a los clientes gestionar adecuadamente la información parcial recibida.

Ejemplo de Ejecución

A continuación, el código fuente 2.3 muestra un ejemplo de una consulta GraphQL y su posible respuesta en caso de una ejecución parcial con errores.

```
query ObtenerUsuario($id: ID!) {  
  usuario(id: $id) {  
    nombre  
    email  
    publicaciones {  
      titulo  
      contenido  
    }  
  }  
}
```

Código Fuente 2.3: Consulta GraphQL para obtener información de un usuario

Si la ejecución de esta consulta se realiza con éxito y los datos están disponibles, el servidor devolverá una respuesta similar al código fuente 2.4.

```
{  
  "data": {  
    "usuario": {  
      "nombre": "Juan Pérez",  
      "email": "juanperez@example.com",  
      "publicaciones": [  
        {  
          "titulo": "Introducción a GraphQL",  
          "contenido": "GraphQL es un lenguaje de consultas para APIs..."  
        }  
      ]  
    }  
  }  
}
```

Código Fuente 2.4: Respuesta exitosa de la consulta

Sin embargo, si durante la ejecución ocurre un error, como la ausencia del usuario en la base de datos, la respuesta incluirá información sobre el problema como muestra en código fuente 2.5.

```
{  
  "data": {  
    "usuario": null  
  },  
  "errors": [  
    {  
      "message": "No se encontró el usuario con el ID proporcionado",  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ]  
    }  
  ]  
}
```

```
],  
  "path": [  
    "usuario"  
  ] ]}]}
```

Código Fuente 2.5: Respuesta con error de la consulta

Este mecanismo permite a los clientes manejar errores de manera estructurada, diferenciando entre datos recuperados y problemas encontrados, lo que mejora la robustez de las aplicaciones que utilizan GraphQL.

Resolvedores (Resolvers)

Los resolvedores (resolvers) en GraphQL son funciones esenciales que determinan cómo se obtiene el valor de cada campo en un esquema GraphQL. Cada tipo y cada campo en GraphQL tienen un resolver asociado, el cual es responsable de recuperar y devolver los datos apropiados en función de la consulta realizada por el cliente [21].

Cuando un cliente envía una consulta, el motor de GraphQL invoca los resolvedores correspondientes para cada campo solicitado. Estos resolvedores pueden acceder a bases de datos, llamar a servicios externos o ejecutar lógica de negocio para construir la respuesta final. Si no se proporciona un resolver personalizado, GraphQL utiliza un resolvedor predeterminado que simplemente devuelve la propiedad correspondiente del objeto padre.

Los resolvedores se implementan en el servidor utilizando algún lenguaje de programación compatible con GraphQL, como JavaScript, Python, Java o Go. Una correcta implementación de los resolvedores es clave para garantizar que los datos se recuperen y procesen de manera eficiente, optimizando el rendimiento de la API.

COMPONENTES DEL SISTEMA DE TIPOS

Esquemas de datos

Representa el conjunto completo de funcionalidades del sistema de tipos de un servicio GraphQL. Se define mediante la especificación de tipos, directivas y tipos de operaciones principales, que son query, mutation y subscription [21].

Tipos

El sistema de tipos en GraphQL define las especificaciones detalladas de los datos disponibles en la API. Estos tipos pueden ser concretos, con nombres específicos, o abstractos, permitiendo flexibilidad y organización en la estructura de los datos [21].

Los tipos con nombre son los elementos fundamentales del esquema de GraphQL y se pueden clasificar en diferentes categorías.

- **Escalares (Scalars):** Representan valores primitivos, como:

- Int: Enteros.
- Float: Números de punto flotante.

- **String:** Cadenas de texto.
- **Boolean:** Valores lógicos (verdadero o falso).
- **ID:** Identificadores únicos.
- **Enumeraciones (Enums):** Especifican un conjunto predefinido de valores posibles, lo que mejora la validación y restringe las opciones dentro de un campo.
- **Objetos (Objects):** Definen un conjunto de campos donde cada campo puede ser de cualquier otro tipo del sistema. Esto permite la creación de estructuras de datos complejas y jerárquicas. Los campos pueden aceptar argumentos que modifiquen su comportamiento, funcionando como funciones parametrizables dentro del esquema.
- **Objetos de entrada (Input objects):** Son similares a los objetos, pero están diseñados exclusivamente para ser utilizados como entradas en consultas o mutaciones. Los campos dentro de un objeto de entrada pueden ser escalares, enumeraciones u otros objetos de entrada.

Además de los tipos concretos, GraphQL también admite tipos abstractos, los cuales deben resolverse en un tipo de objeto específico durante la ejecución:

- **Interfaces (Interfaces):** Definen un conjunto de campos que deben ser implementados por uno o más tipos de objetos. Permiten reutilizar estructuras de datos comunes entre diferentes tipos.
- **Uniones (Unions):** Agrupan varios tipos de objetos en una sola entidad, lo que permite que un campo pueda devolver múltiples tipos posibles.

Por último, GraphQL proporciona dos tipos modificadores que pueden aplicarse a otros tipos:

- **Lista (List):** Permite que un campo devuelva una colección de valores en lugar de un solo valor.
- **No nulo (Non-null):** Garantiza que un campo nunca devuelva un valor nulo, mejorando la seguridad de los datos.

Ejemplo de Tipos en GraphQL

A continuación, el código fuente 2.6 muestra un ejemplo de un esquema GraphQL que define diferentes tipos, incluyendo escalares, objetos, enumeraciones y objetos de entrada.

```
type Usuario {
  id: ID!
  nombre: String!
  email: String!
  tipo: TipoUsuario!
  publicaciones: [Publicacion]
}

type Publicacion {
  id: ID!
  titulo: String!
  contenido: String!
  autor: Usuario!
}

enum TipoUsuario {
  ADMIN
```

```
EDITOR
LECTOR
}

input NuevaPublicacionInput {
  titulo: String!
  contenido: String!
  autorId: ID!
}

type Mutation {
  crearPublicacion(input: NuevaPublicacionInput!): Publicacion
}

query ObtenerUsuario($id: ID!) {
  usuario(id: $id) {
    nombre
    email
    tipo
    publicaciones {
      titulo
      contenido
    }
  }
}
```

Código Fuente 2.6: Definición de tipos en GraphQL

OPERACIONES DE GRAPHQL

Consultas

En GraphQL, el proceso de obtención de datos se realiza mediante consultas. A diferencia de las API REST tradicionales, donde cada endpoint devuelve una estructura de datos predefinida, GraphQL permite a los clientes especificar exactamente qué datos necesitan y en qué formato desean recibirlos. Esto optimiza el uso de recursos y minimiza la cantidad de datos transferidos, mejorando el rendimiento de la aplicación y reduciendo el ancho de banda necesario para las comunicaciones entre el cliente y el servidor [21].

Las consultas en GraphQL se basan en la estructura del esquema definido en el servidor. Cada consulta puede incluir múltiples campos anidados, lo que permite obtener información relacionada en una sola petición en lugar de realizar múltiples llamadas a diferentes endpoints. Además, GraphQL permite el uso de variables en las consultas, lo que facilita la reutilización y parametrización de las solicitudes.

Otro aspecto importante de las consultas en GraphQL es su naturaleza declarativa. En lugar de definir cómo obtener los datos, el cliente simplemente describe el resultado esperado, lo que permite una mayor modularidad y flexibilidad en la construcción de la API. Además, GraphQL proporciona mecanismos para manejar errores de manera eficiente y ofrecer respuestas parciales en caso de que solo una parte de la consulta falle.

Ejemplo de Consulta en GraphQL

A continuación, el código fuente 2.7 presenta un ejemplo de una consulta GraphQL utilizada para obtener información sobre un usuario y sus publicaciones.

```
query ObtenerUsuario($id: ID!) {  
  usuario(id: $id) {  
    nombre  
    email  
    tipo  
    publicaciones {  
      titulo  
      contenido  
    }  
  }  
}
```

Código Fuente 2.7: Ejemplo de consulta en GraphQL

En este ejemplo, se realiza las siguientes consultas:

- La consulta solicita información sobre un usuario específico mediante su identificador `id`.
- Se recuperan los campos `nombre`, `email` y `tipo` del usuario.
- También se obtienen las publicaciones asociadas a ese usuario, incluyendo sus títulos y contenidos.
- El uso de la variable `id` permite reutilizar la consulta con diferentes valores sin modificar su estructura.

Mutaciones

En GraphQL, una mutación es una operación fundamental que permite a los clientes modificar datos en el servidor. A través de las mutaciones, se pueden realizar diversas acciones como la inserción de nuevos datos, la actualización de información existente o la eliminación de registros. Este mecanismo es crucial para modificar el estado de la aplicación y garantizar la integridad de los datos en el sistema [21].

Las mutaciones en GraphQL pueden considerarse equivalentes a las operaciones HTTP POST, PUT, PATCH y DELETE en arquitecturas REST. A diferencia de las consultas, que solo recuperan información, las mutaciones permiten realizar cambios específicos en la base de datos a través de una estructura bien definida en el esquema. Al estar fuertemente tipadas, las mutaciones deben seguir un esquema preestablecido, asegurando que las modificaciones sean consistentes y libres de errores.

Existen diferentes tipos de mutaciones en GraphQL, dependiendo de la operación que se desee realizar. Estas pueden clasificarse en tres grupos principales:

- **Inserción (Insert Mutations):** Permiten agregar nuevos datos al sistema, como crear nuevos usuarios, registrar productos o añadir elementos a una lista.
- **Actualización (Update Mutations):** Se utilizan para modificar datos existentes, por ejemplo, actualizar la información de un usuario o cambiar el contenido de una publicación.
- **Eliminación (Delete Mutations):** Facilitan la eliminación de datos del sistema, como borrar un usuario de una base de datos o eliminar publicaciones de un blog.

Cada uno de estos tipos de mutaciones desempeña un papel clave en la gestión de datos dentro de una API GraphQL, proporcionando a los clientes un mecanismo estructurado y flexible para modificar información según sea necesario.

Ejemplo de Mutación en GraphQL

A continuación, el código fuente 2.8 presenta un ejemplo de una mutación en GraphQL para la creación de una nueva publicación en un sistema de gestión de contenido.

```
mutation CrearPublicacion($input: NuevaPublicacionInput!) {  
  crearPublicacion(input: $input) {  
    id  
    titulo  
    contenido  
    autor {  
      nombre  
      email  
    }  
  }  
}
```

Código Fuente 2.8: Ejemplo de mutación en GraphQL

En este ejemplo se realiza las siguientes mutaciones:

- La mutación `crearPublicacion` recibe un objeto de entrada `NuevaPublicacionInput` con los datos de la nueva publicación.
- La respuesta de la mutación devuelve el `id`, `titulo` y `contenido` de la publicación creada.
- Además, se incluye información del autor de la publicación, como su `nombre` y `email`.
- El uso de variables en la mutación permite reutilizar la misma estructura para diferentes solicitudes sin modificar la consulta.

Suscripciones

Las suscripciones en GraphQL representan una poderosa herramienta que permite a los clientes suscribirse a fuentes reactivas de datos en el servidor. Esta capacidad transforma la forma en que los clientes interactúan con la API, ya que les permite recibir actualizaciones en tiempo real en lugar de tener que realizar consultas repetitivas para obtener información actualizada [21].

Cuando un cliente se suscribe a una fuente reactiva, como por ejemplo un flujo de eventos, una cola de mensajes o un canal de chat, activa una consulta GraphQL sobre esos datos. Esto significa que el servidor ejecuta la consulta GraphQL correspondiente a medida que los nuevos datos llegan a la fuente reactiva. Luego, los resultados de esta consulta se transmiten al cliente, proporcionándole actualizaciones en tiempo real sobre los cambios en los datos.

Este modelo de suscripción basado en eventos resulta útil en escenarios donde la información cambia con frecuencia o donde es crucial recibir actualizaciones en tiempo real, como en aplicaciones de mensajería instantánea, sistemas de monitoreo en tiempo real o aplicaciones colaborativas. Las suscripciones en GraphQL ofrecen una manera eficiente y escalable de implementar este tipo de funcionalidad, permitiendo una experiencia de usuario más dinámica y enriquecida.

Ejemplo de Suscripción en GraphQL

A continuación, el código fuente 2.9 muestra un ejemplo de una suscripción en GraphQL para recibir notificaciones en tiempo real cuando se crea una nueva publicación.

```
subscription NuevaPublicacion {  
  nuevaPublicacion {  
    id  
    titulo  
    contenido  
    autor {  
      nombre  
      email  
    }  
  }  
}
```

Código Fuente 2.9: Ejemplo de suscripción en GraphQL

En este ejemplo, el cliente se suscribe a la creación de nuevas publicaciones. Cada vez que una nueva publicación es agregada al sistema, el servidor envía automáticamente los datos relevantes a todos los clientes suscritos. Esto permite que los clientes reciban actualizaciones en tiempo real sin necesidad de hacer consultas repetidas, optimizando así el rendimiento y mejorando la experiencia del usuario.

Fragmentos

Los fragmentos en GraphQL permiten reutilizar partes de consultas al definir conjuntos de campos que pueden ser compartidos entre múltiples consultas o mutaciones. Esto evita la repetición de código, mejora la claridad de las consultas y facilita el mantenimiento en aplicaciones que consumen APIs GraphQL [22].

En GraphQL, los fragmentos se definen con la palabra clave `fragment`, se les asigna un nombre y se especifica el tipo al que pertenecen. Posteriormente, estos fragmentos pueden ser utilizados dentro de consultas o mutaciones mediante la sintaxis de propagación [22].

En el código 2.10 se han definido dos ejemplos de fragmentos en base al esquema de tipos del código 2.1.

DetallesUsuario: Contiene los datos principales de un usuario.

```
fragment DetallesUsuario on Usuario {  
  id  
  nombre  
  email  
}
```

Código Fuente 2.10: Ejemplo de fragmento para los datos de un usuario

DetallesPublicacion: Contiene los datos principales de una publicación y su autor.

```
fragment DetallesPublicacion on Publicacion {  
  id  
  titulo  
  contenido  
  autor {  
    ...DetallesUsuario  
  }  
}
```

Código Fuente 2.11: Ejemplo de fragmento para los datos de una publicación

Consultas utilizando los fragmentos en GraphQL

Una de las principales ventajas de los fragmentos en GraphQL es la capacidad de estructurar consultas de manera eficiente y evitar la repetición de código. A continuación, veremos en detalle cómo se puede obtener información de un usuario junto con todas sus publicaciones utilizando fragmentos.

```
query ObtenerUsuario($id: ID!) {  
  usuario(id: $id) {  
    ...DetallesUsuario  
    publicaciones {  
      ...DetallesPublicacion  
    }  
  }  
}
```

Código Fuente 2.12: Ejemplo de consulta utilizando fragmento

En el código 2.12, se puede observar los fragmentos utilizados para los detalles del usuario y para el detalle de la publicación, el fragmento utilizado para el detalle del usuario se encarga de extraer información esencial de un usuario, incluyendo su identificador único (ID), nombre y dirección de correo electrónico. Su propósito es facilitar la reutilización en cualquier consulta que requiera datos del usuario, eliminando la redundancia y mejorando la eficiencia del código. Al centralizar esta lógica, se optimiza el mantenimiento y se garantiza la consistencia en la recuperación de información en diferentes partes del sistema, el fragmento utilizado para el detalle de la publicación se encarga de extraer los datos fundamentales de una publicación, incluyendo su identificador único (ID), título y contenido. Además, incorpora información detallada sobre el autor de la publicación mediante la reutilización del fragmento DetallesUsuario. Esto permite estructurar y organizar mejor la información, evitando la duplicación de código y garantizando la coherencia en la recuperación de los datos del usuario en distintas consultas.

En el código 2.13 se observa el ejemplo de la respuesta obtenida del ejemplo de la consulta en donde se utilizan los fragmentos.

```
{
  "data": {
    "usuario": {
      "id": "1",
      "nombre": "Juan Pérez",
      "email": "juan@example.com",
      "publicaciones": [
        {
          "id": "101",
          "titulo": "Introducción a GraphQL",
          "contenido": "GraphQL es un lenguaje de consulta para APIs...",
          "autor": {
            "id": "1",
            "nombre": "Juan Pérez",
            "email": "juan@example.com"
          }
        },
        {
          "id": "102",
          "titulo": "Avanzando con GraphQL",
          "contenido": "En este artículo exploraremos los fragmentos...",
          "autor": {
            "id": "1",
            "nombre": "Juan Pérez",
            "email": "juan@example.com"
          }
        }
      ]
    }
  }
}
```

Código Fuente 2.13: Ejemplo de la respuesta de la consulta

RESUMEN

GraphQL, se conoce como un paradigma que transforma la forma en que cliente y servidor interactúan. A diferencia de los enfoques tradicionales, GraphQL se sustenta en un sistema de tipos fuertemente tipado y en un lenguaje declarativo de consultas (SDL), lo que aporta claridad y consistencia en la definición de datos. Esta base conceptual no solo otorga mayor seguridad al desarrollo de aplicaciones, sino que también establece una estructura sólida que guía la evolución de las APIs, reduciendo ambigüedades y mejorando la comunicación entre equipos de desarrollo.

La introducción a los elementos fundamentales de GraphQL como consultas, mutaciones y suscripciones permite comprender cómo este enfoque optimiza la precisión en la recuperación de datos y otorga un mayor control sobre las operaciones. Mientras las consultas garantizan la obtención de información exacta y personalizada, las mutaciones permiten modificar el estado de los datos de forma eficiente, y las suscripciones facilitan la actualización en tiempo real. Este modelo no solo mejora la experiencia del cliente al reducir sobrecarga de información, sino que también asegura que las APIs evolucionen sin romper la compatibilidad con versiones anteriores, un aspecto clave en entornos de software dinámicos.

Asimismo, este capítulo aborda características avanzadas como la introspección y el uso de fragmentos, que refuerzan la capacidad de descubrimiento, exploración y reutilización de los esquemas definidos. Estas funcionalidades convierten a GraphQL en una herramienta flexible y adaptable, capaz de integrarse en escenarios donde los datos son complejos, distribuidos y en constante cambio. De esta manera, se consolida como una alternativa robusta frente a otros modelos de comunicación, ofreciendo a los desarrolladores un lenguaje potente que equilibra precisión, eficiencia y escalabilidad.



CAPÍTULO 3

Backend con GraphQL: Implementación de APIs

*«La tecnología avanza tan rápido como las ideas
de quienes la crean. El software es el motor
del futuro.»*

Torvalds, L., 2001, p. 189.

Este capítulo se enfoca en el desarrollo backend de APIs utilizando GraphQL, adoptando un enfoque práctico que combina la teoría y la aplicación. A través de un proyecto de desarrollo de software en donde se construye una API básica para la gestión de una pizzería, se profundiza y se aplican los conceptos fundamentales de esta tecnología. El objetivo es proporcionar al lector una guía integral que facilite la comprensión de la implementación de GraphQL en proyectos reales, construyendo una base para su aplicación en entornos modernos de desarrollo. La sección 3 describe la configuración del entorno de desarrollo para el proyecto de gestión de una pizzería. La sección 3 explica la definición del sistema de tipos (estructura del esquema de la API). Finalmente, la sección 3 muestra la implementación de las principales operaciones de GraphQL: consultas (obtener datos), mutaciones (modificar datos) y suscripciones (recibir actualizaciones en tiempo real).

CONFIGURACIÓN DEL AMBIENTE DE DESARROLLO

Herramientas para el desarrollo de la API

En el desarrollo backend, existen diversas herramientas y tecnologías diseñadas para facilitar la construcción, mantenimiento y optimización de APIs. Estas herramientas permiten gestionar la lógica del negocio, la interacción con bases de datos, la seguridad, el rendimiento y la documentación de los servicios expuestos.

A continuación, se detallan las principales herramientas que se utilizarán en el desarrollo de APIs en el backend dentro de este capítulo.

Node.js

Node.js es una plataforma que se basa en el motor V8 de Google Chrome y permite la ejecución de JavaScript en el servidor. Su principal fortaleza radica en su modelo de entrada/salida asíncronico y en

su arquitectura basada en eventos. Node.js es una opción óptima para aplicaciones con altos niveles de concurrencia [23].

A diferencia de soluciones tradicionales como PHP o Java, que dependen de un enfoque multihilo para procesar solicitudes. Node.js emplea un modelo monohilo con operaciones asincrónicas. Esto minimiza el costo asociado al cambio de contexto y optimiza el rendimiento [23].

Beneficios de Node.js

Conexiones rápidas al servidor

Node.js¹ emplea un bucle de eventos (event loop) y un modelo monohilo asincrónico; esto permite procesar múltiples solicitudes de manera eficiente sin recurrir a la administración de múltiples hilos. En comparación con PHP, que sigue un enfoque síncrono, Node.js destaca por su rendimiento optimizado en entornos con alta concurrencia [24].

Uso de un único lenguaje de programación

A diferencia de otras tecnologías que requieren conocimientos en distintos lenguajes para el desarrollo frontend y backend (por ejemplo, PHP para el backend y JavaScript para el frontend). Node.js permite utilizar JavaScript en ambos entornos, lo que simplifica el proceso de desarrollo y reduce la curva de aprendizaje [24].

Alta popularidad y comunidad activa

JavaScript es uno de los lenguajes de programación más usados a nivel global, lo que facilita su aprendizaje y adopción en Node.js. Además, cuenta con una comunidad de desarrolladores extensa y el respaldo de grandes compañías tecnológicas, lo que garantiza un ecosistema en constante evolución y con abundantes recursos de soporte [24].

Apollo Server

Apollo Server es un servidor GraphQL de código abierto diseñado para simplificar la construcción, despliegue y mantenimiento de APIs GraphQL. Ofrece compatibilidad con diversas fuentes de datos, como bases de datos relacionales y APIs REST, lo que permite a los desarrolladores crear APIs eficientes y escalables. Su diseño permite la personalización mediante la definición de tipos, resolvers y directivas personalizadas, facilitando la creación de esquemas GraphQL adaptados a necesidades específicas. Además, es compatible con cualquier cliente GraphQL, destacándose Apollo Client como una de las opciones más utilizadas [25].

Uno de los principales beneficios de Apollo Server es su flexibilidad y capacidad de extensión, siendo adecuado para una amplia variedad de casos de uso. Los desarrolladores pueden adaptarlo fácilmente a diferentes arquitecturas y entornos, optimizando su rendimiento según las necesidades del proyecto. Su ecosistema también incluye herramientas avanzadas para monitoreo, caché y optimización de consultas, lo que mejora la eficiencia en aplicaciones de gran escala. Gracias a estas características, Apollo Server se ha convertido en una opción popular dentro de la comunidad de GraphQL [25].

PostgreSQL

PostgreSQL² es un sistema de gestión de bases de datos objeto-relacional de código abierto, diseñado para ofrecer alta eficiencia, escalabilidad y rendimiento en el manejo de datos estructurados. Su modelo

¹Node.js: <https://nodejs.org/en>

²PostgreSQL: <https://www.postgresql.org/>

relacional garantiza integridad y consistencia. Esto permite la gestión de grandes volúmenes de información con consultas complejas. Además, soporta múltiples tipos de índices avanzados, como BTree, Hash, GIN y GiST, optimizados para mejorar el acceso y procesamiento de datos [26].

Instalación de las herramientas para el desarrollo de la API

Para la configuración del ambiente de desarrollo con respecto al proyecto de gestión de una pizzería, es necesario realizar los siguientes pasos que se describen a continuación:

- El primer paso que se debe realizar es instalar Node JS que es un entorno de código abierto y multiplataforma diseñado para ejecutar código fuente en el servidor.

La Fig. 3.1 muestra el sitio oficial para realizar la descarga de Node.js. Es importante tener en cuenta que existen dos versiones de descarga: las versiones LTS y las versiones actuales o current. Lo más aconsejable cuando se realiza la descarga es escoger las versiones LTS, debido a que estas suelen recibir mayor soporte y son versiones más estables que permiten evitar errores e inconvenientes al momento de desarrollar aplicaciones para el servidor con Node.js.

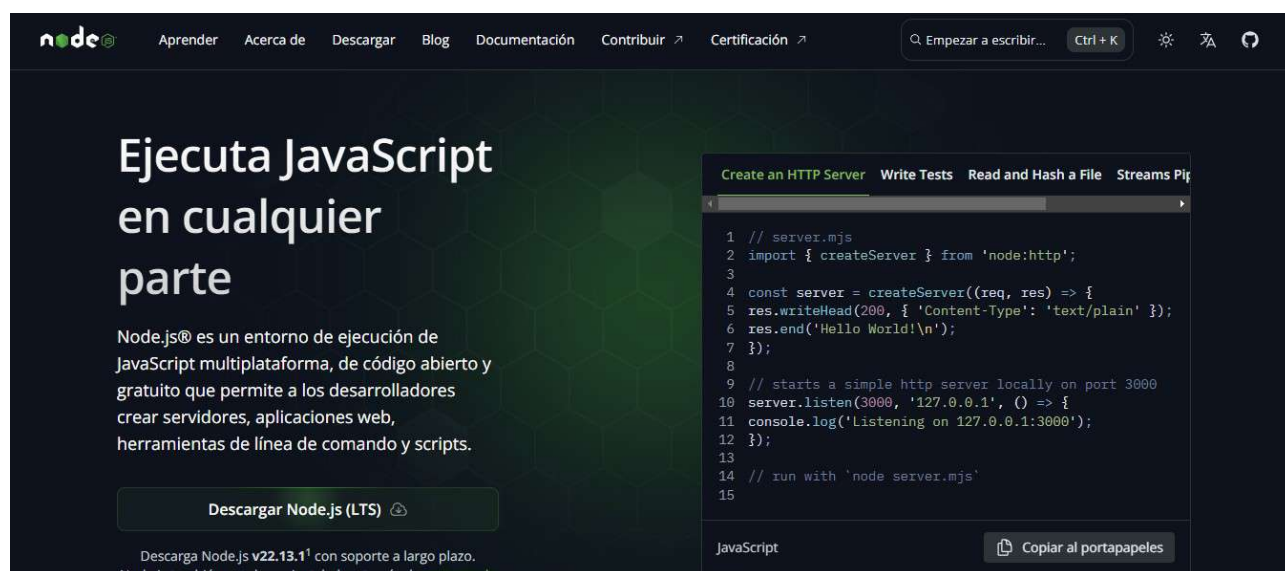


Fig. 3.1. Descarga de Node.js

- Posteriormente, como se muestra en la Fig. 3.2, una vez que se haya descargado la herramienta, se necesita realizar la instalación. Para lo cual se debe ejecutar el archivo .msi en el caso de sistemas operativos como Windows. En el caso de los sistemas operativos de distribuciones Linux y Ubuntu, las extensiones serán distintas, pero el proceso de instalación es similar.

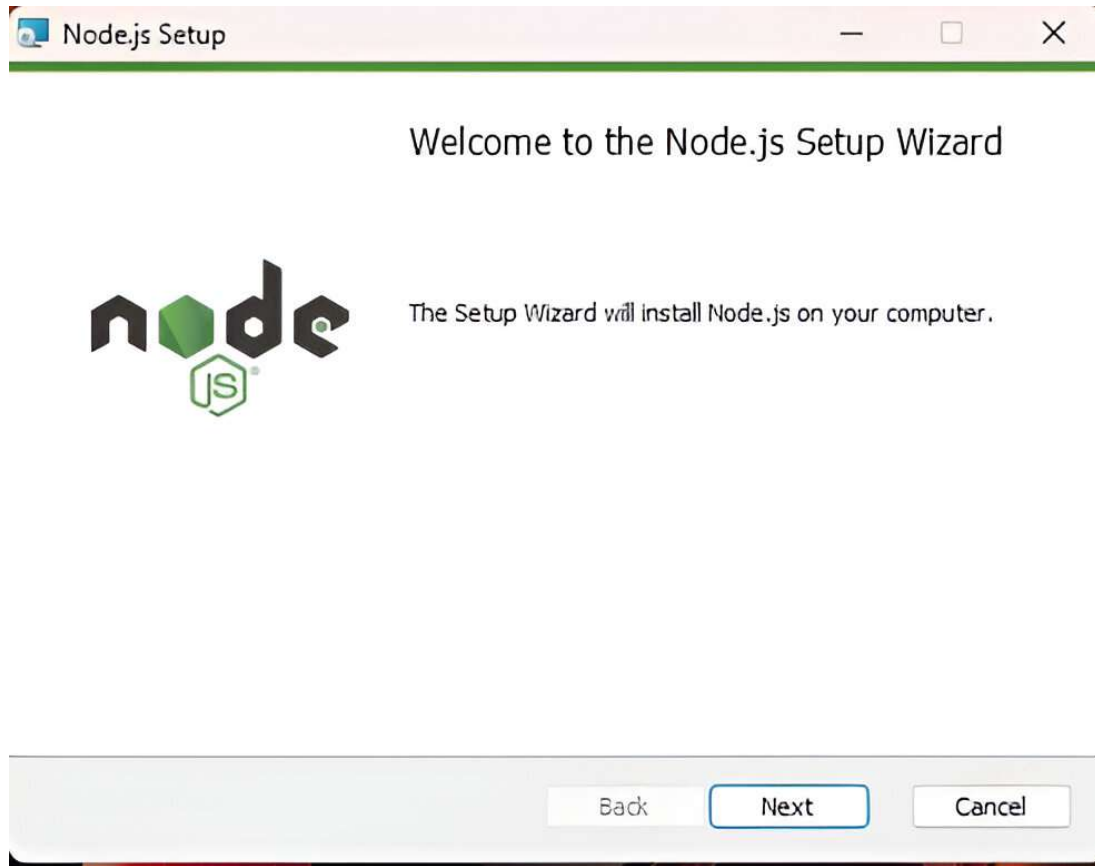


Fig. 3.2. Instalación de Node.js

Nota

Para verificar que la instalación de Node.js fue exitosa y que no existió ningún problema, puede ejecutar el siguiente comando en su terminal para verificar la versión instalada:

node -v

- La siguiente herramienta que se requiere instalar es el editor de código Visual Studio. Esta herramienta es un editor de código desarrollado por Microsoft que está disponible para Windows, Linux, macOS y la web. Ofrece una amplia gama de características, que incluyen soporte para depuración, control integrado de Git, resaltado de sintaxis, finalización automática de código fuente, fragmentos y refactorización de código fuente.

La Fig. 3.3 muestra el sitio oficial para la descarga del editor. No es necesario realizar ninguna configuración antes de su descarga; solo se debe ejecutar el archivo descargado y seguir cada una de las instrucciones durante el proceso de instalación.

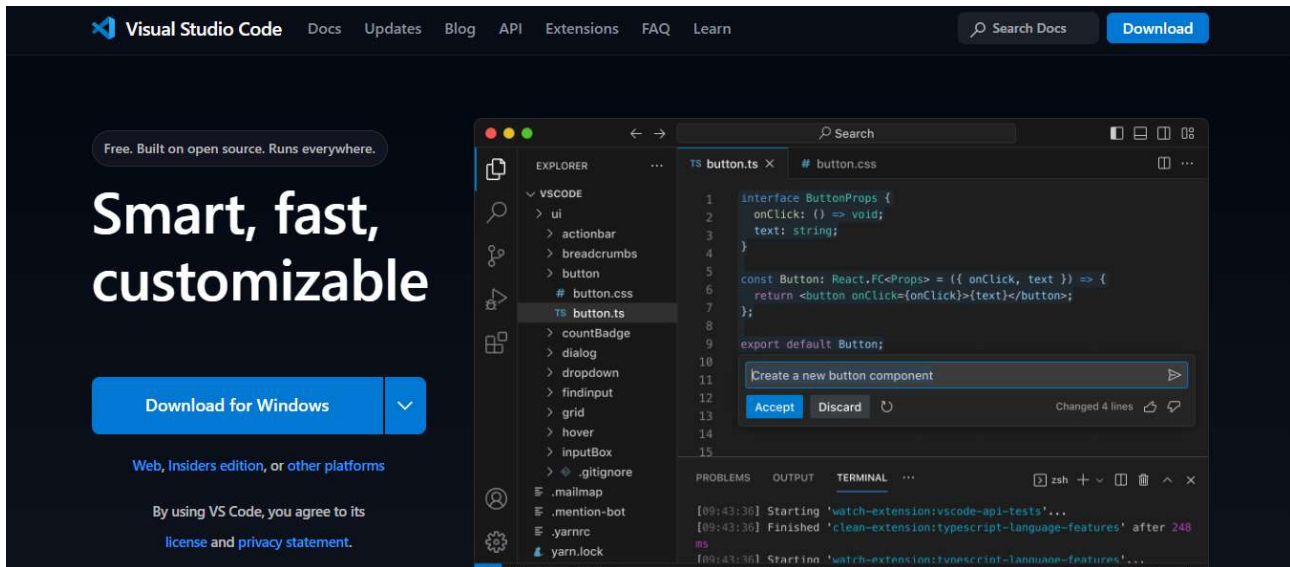


Fig. 3.3. Descarga de Visual Studio Code

- Como siguiente paso en la configuración del ambiente de desarrollo, es descargar el sistema de gestión de base de datos o también denominado SGBD. Existen distintos tipos, pero durante el desarrollo de la API de GraphQL se utilizará PostgreSQL debido a que es una opción confiable y popular dentro de las bases de datos relacionales.

La Fig. 3.4 muestra el sitio oficial para la descarga de la base de datos PostgreSQL que se utilizará a lo largo del desarrollo de esta API en GraphQL.

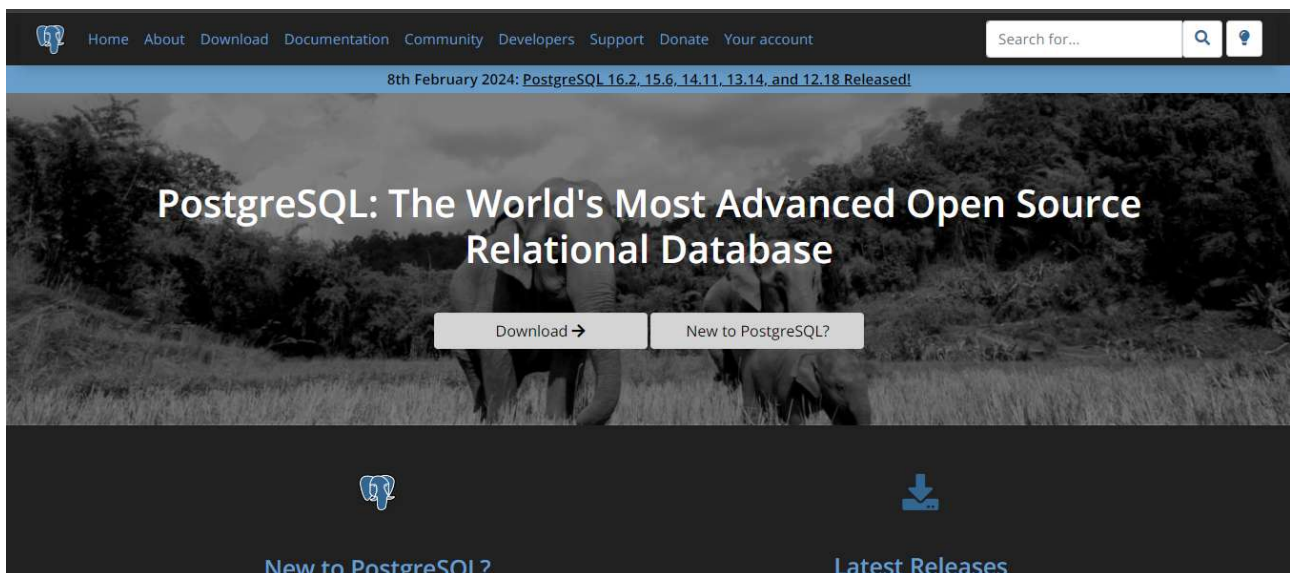


Fig. 3.4. Descarga de PostgreSQL

Creación del proyecto

- En la ubicación de su preferencia es necesario crear una carpeta con el nombre **API-PIZZA-GRAPHQL**. El nombre de la carpeta puede ser distinto, pero es necesario que recuerde cuál es la carpeta con la que va a trabajar.

La Fig. 3.5 muestra la carpeta del proyecto creada.

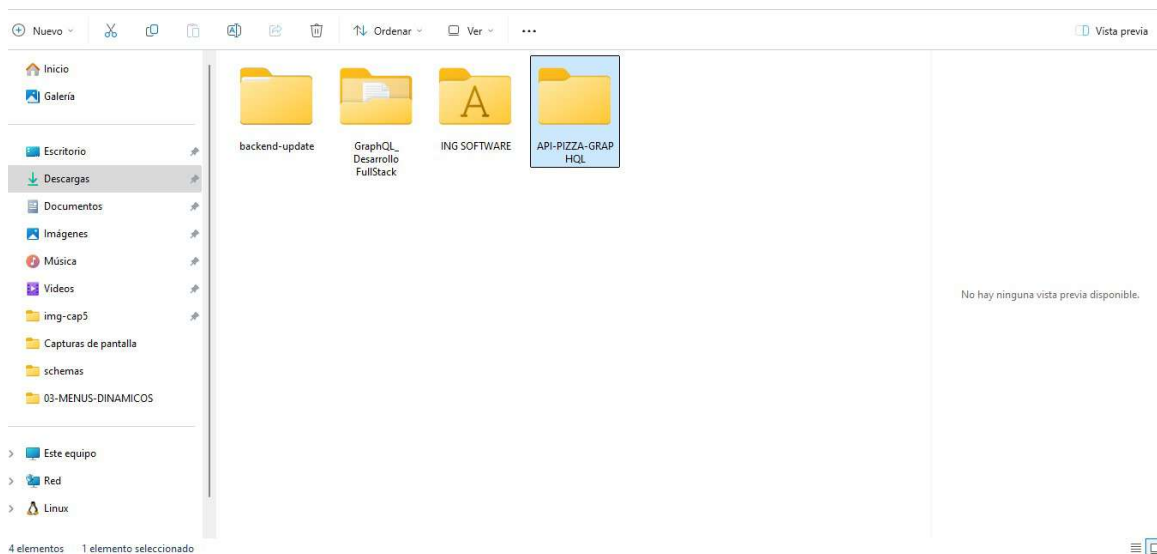


Fig. 3.5. Creación de la carpeta del proyecto

- En la Fig. 3.6 se observa que existen tres opciones; la primera forma permite abrir la carpeta del proyecto que se creó recientemente. La segunda forma es arrastrar la carpeta hacia el editor y esta se abrirá. La tercera forma es mediante el uso del comando **code** de la terminal.

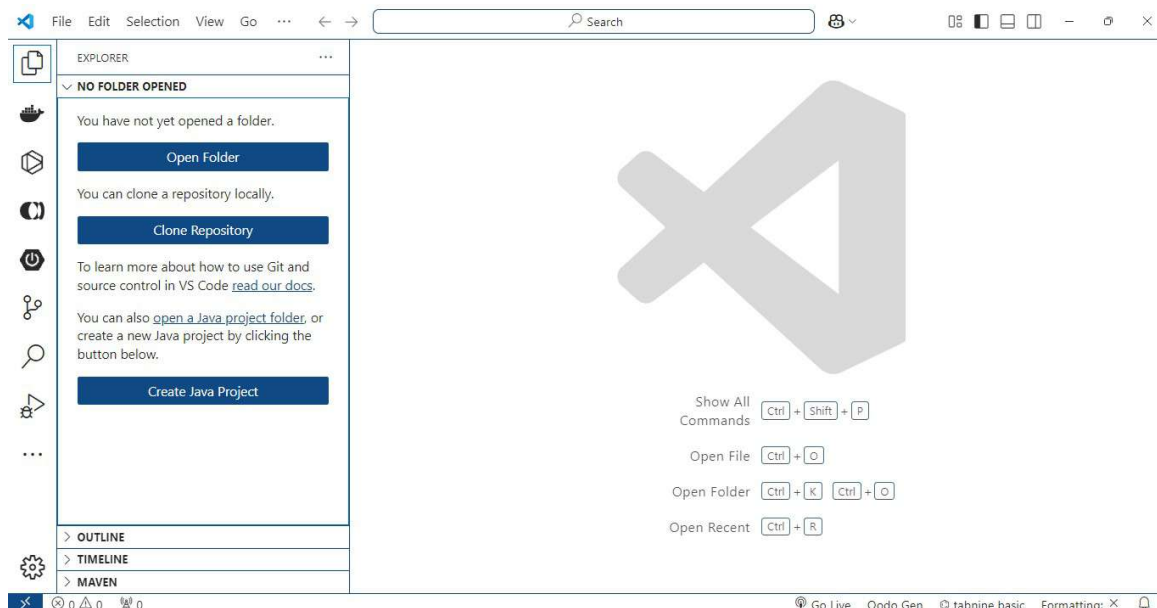


Fig. 3.6. Abrir carpeta en Visual Studio Code

- Una vez que la carpeta del proyecto se encuentra abierta, el siguiente paso es abrir la terminal en Visual Studio Code para instalar las dependencias que el proyecto necesitará posteriormente. La instalación de las dependencias se realizará mediante el Node Package Manager conocido como (NPM). NPM es el sistema de gestión de paquetes por defecto para Node.js, un entorno de ejecución para JavaScript, como se aprecia en la Fig. 3.7.

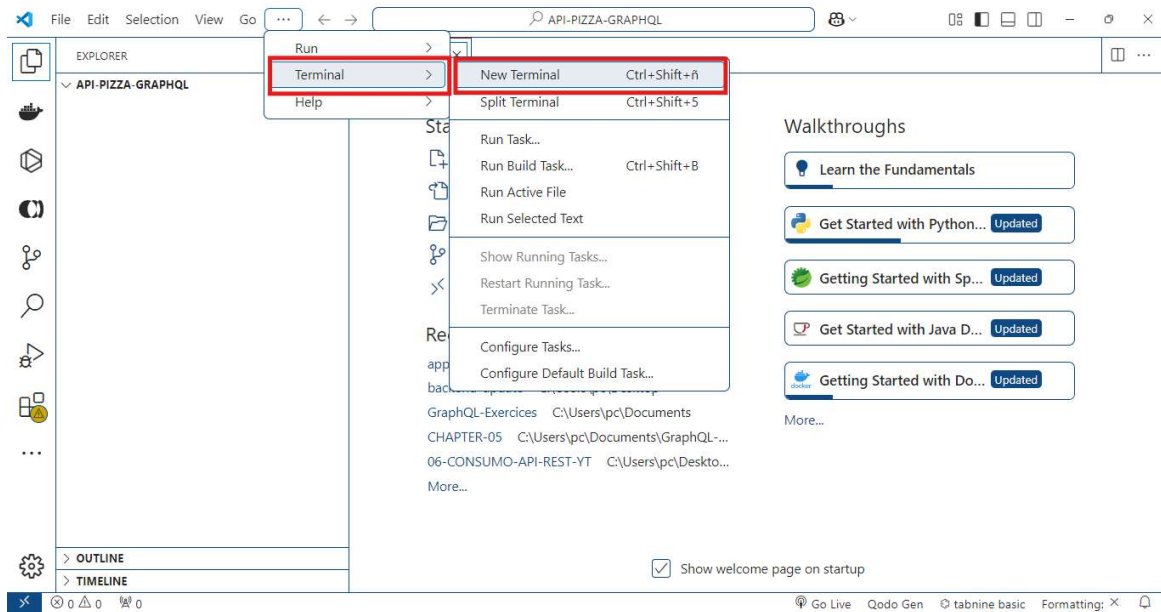


Fig. 3.7. Abrir terminal en VS-Code

- Para comprobar el funcionamiento “Node” se pueden utilizar los siguientes comandos:

node -v permite verificar la versión de Node JS instalada. **npm -v** permite visualizar la versión de NPM instalada, ver Fig. 3.8.

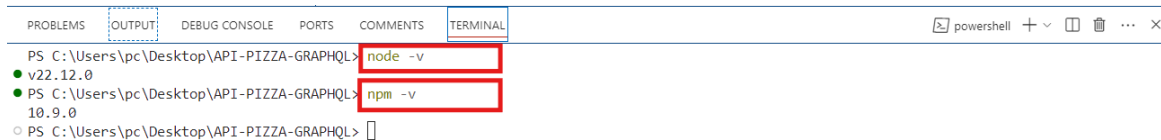
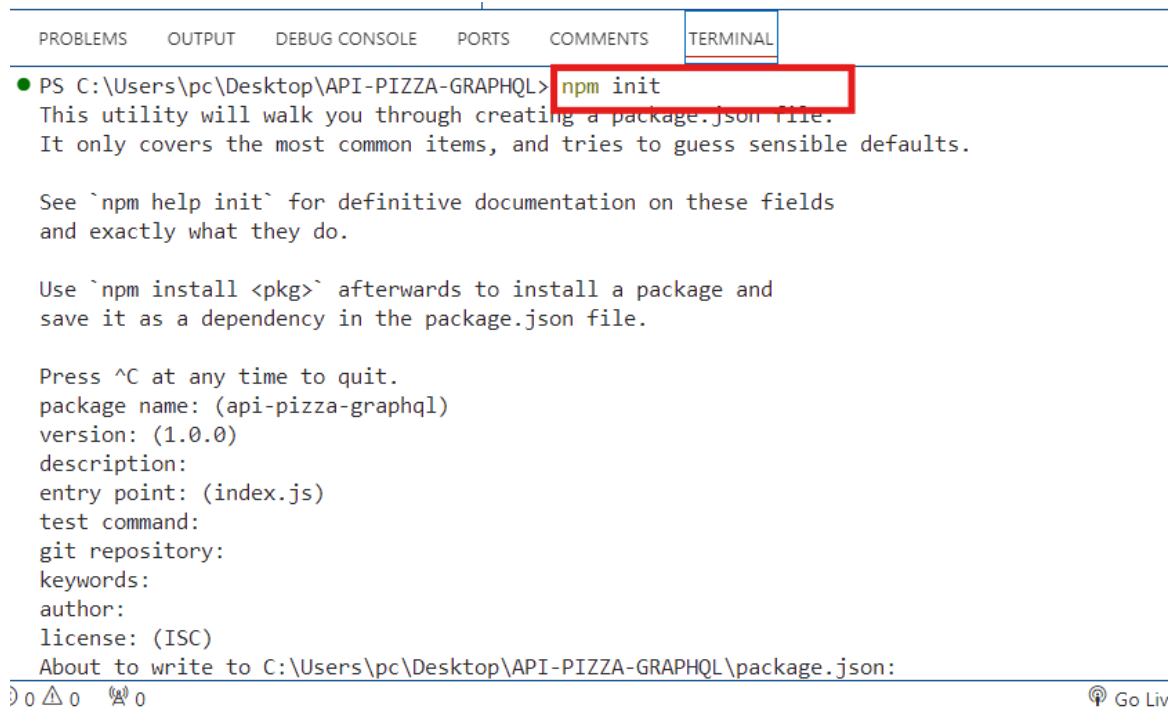


Fig. 3.8. Verificación del funcionamiento de Node

- Al ejecutar el comando “npm init” en su terminal, se inicializará un nuevo proyecto de Node.js. Le pedirá que proporcione información sobre su proyecto, como el nombre, versión, descripción, punto de entrada, comando de prueba, repositorio Git, autor y licencia. Puede completar estos detalles de forma interactiva o proporcionarlos directamente a través de opciones en la línea de comandos.

La Fig. 3.9 muestra el proyecto inicializado mediante el comando “npm init”. Como se observa, cada uno de los campos fue registrado de acuerdo con lo que el autor consideró necesario.



```
PS C:\Users\pc\Desktop\API-PIZZA-GRAPHQL> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

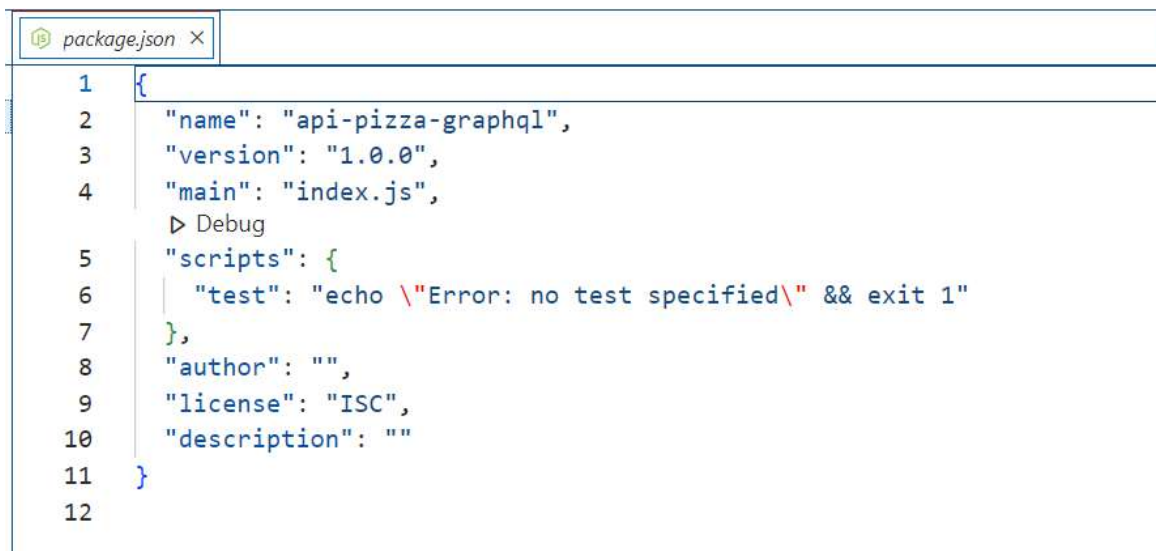
See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (api-pizza-graphql)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\pc\Desktop\API-PIZZA-GRAPHQL\package.json:
```

Fig. 3.9. Inicialización del proyecto

- Mediante la Fig. 3.10 se observa la estructura que tiene el archivo package.json. Es recomendable que la estructura sea similar para evitar posibles problemas e inconvenientes.



```
1 {
2   "name": "api-pizza-graphql",
3   "version": "1.0.0",
4   "main": "index.js",
5   "scripts": {
6     "test": "echo \"Error: no test specified\" && exit 1"
7   },
8   "author": "",
9   "license": "ISC",
10  "description": ""
11 }
12
```

Fig. 3.10. Package.json

- Dentro de la carpeta del proyecto se debe crear el archivo “index.js”, es el archivo principal de la aplicación. Este archivo debe ser verificado cuando se muestra un mensaje en la terminal con el texto “hola mundo”. Para ejecutar el archivo se hace uso del siguiente comando: **node index.js**, ver Fig. 3.11.



Fig. 3.11. Archivo index.js

- La Fig. 3.12 muestra la estructura del proyecto. Tome en cuenta que el archivo “index.js” debe colocarse dentro de la carpeta “src”. Del mismo modo, es necesario que se creen carpetas “config” para la base de datos, “controllers” para los controladores y “type-system” para el sistema de tipos. Cabe mencionar que el archivo “.env” permitirá configurar las variables de entorno que se utilizarán posteriormente.

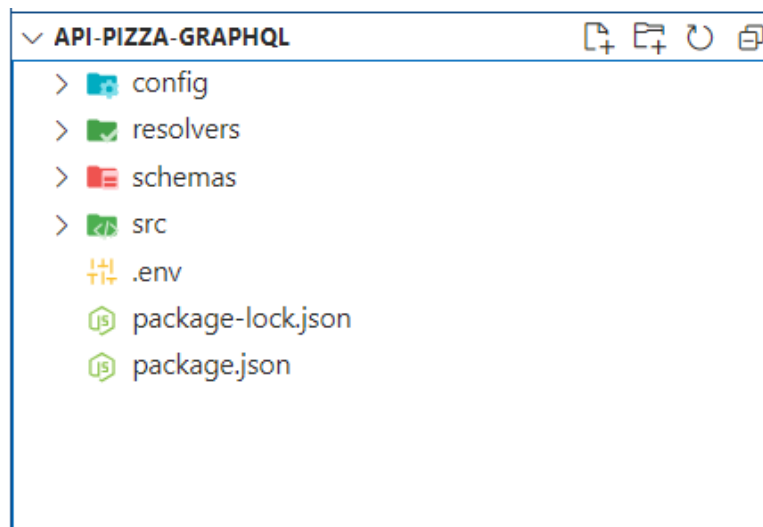


Fig. 3.12. Estructura de proyecto

Conexión a la base de datos

Crear la base de datos pizza.

- Para realizar la creación de la base de datos de pizzas, es necesario abrir la herramienta de pgAdmin; luego es necesario crear una base de datos con el nombre pizza, como se visualiza en la Fig. 3.13. Una vez que la base de datos se encuentre creada, lo siguiente es cargar el script del backup; el script mencionado se encuentra en la parte de anexos.

Antes de crear el archivo para la conexión hacia la base de datos, es necesario colocar los valores de

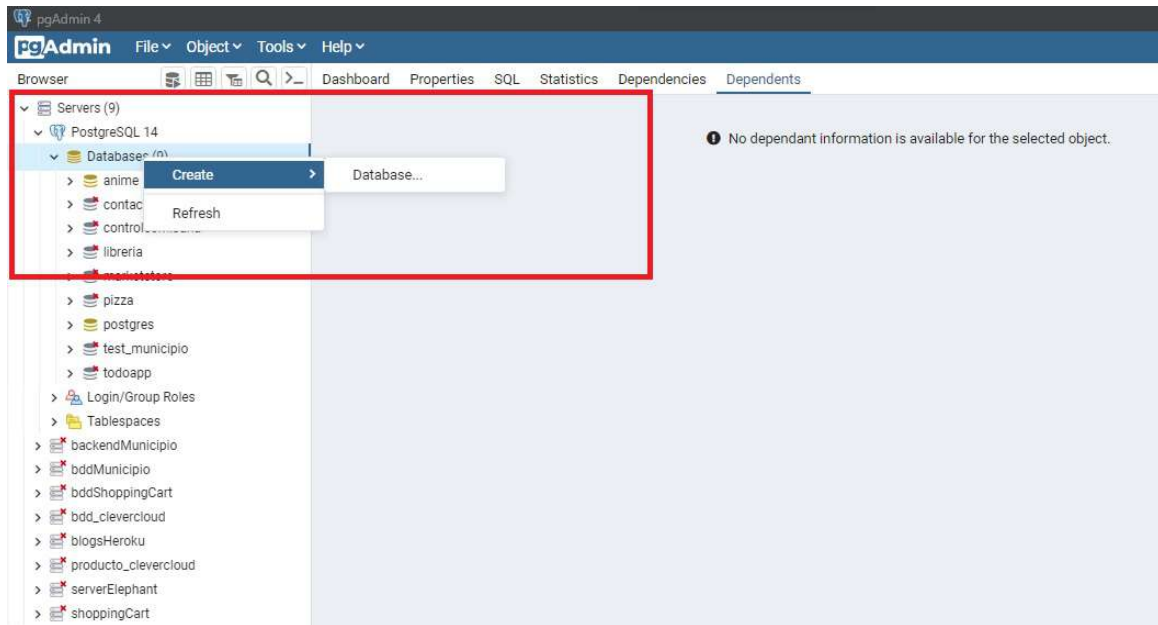


Fig. 3.13. Creación de la base de datos

las variables de entorno en el archivo “.env”, ver el código fuente 3.1.

```
DB_HOST=localhost
DB_PORT=5432
DB_DATABASE=pizza
DB_USER=postgres
DB_PASSWORD=1234admin
```

Código Fuente 3.1: Archivo para las variables de entorno

Creación del archivo de conexión hacia la base de datos

Dentro de la carpeta config se debe crear un archivo con el nombre “cnn.js”. A continuación, es necesario instalar pg-promise, que es una biblioteca para Node.js y proporciona una API para trabajar con bases de datos PostgreSQL de forma sincrónica y asincrónica. Esto permite interactuar con PostgreSQL utilizando promesas. Las promesas en JavaScript representan un valor que puede estar disponible en el presente, en el futuro, o puede que nunca esté disponible. Se utilizan para gestionar operaciones asíncronas de manera estructurada, lo que mejora la claridad del código fuente [27]. Además, se debe instalar la dependencia dotenv para el manejo de las variables de entorno, las cuales permiten almacenar configuraciones sensibles o valores que pueden cambiar entre diferentes entornos, como producción o desarrollo y posteriormente se debe instalar la dependencia dotenv para el manejo de las variables de entorno).

Utilice el siguiente comando para instalar las librerías:

Comando para instalar las librerías

```
npm install --save-dev pg-promise dotenv
```

A continuación se muestra la configuración del archivo “cnn.js” para la conexión con la base de datos. En este caso se ingresan las credenciales para la conexión a la base de datos utilizando las variables de entorno que se establecieron en el archivo “.env”, ver el código fuente 3.2.

```
import * as dotenv from 'dotenv';
import pgPromise from "pg-promise";
```

```
dotenv.config();

const config = {
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  database: process.env.DB_DATABASE,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
};

const pgp = pgPromise({});
const db = pgp(config);

export { db };
```

Código Fuente 3.2: Archivo de configuración de la base de datos

Nota

En GraphQL, así como en el desarrollo moderno con Node.js, se recomienda el uso de “import” en lugar de “require”, ya que “import” forma parte del estándar ECMAScript 6 (ES6) para la gestión de módulos. Para habilitar esta funcionalidad en un entorno de Node.js, es necesario modificar el archivo “package.json”, estableciendo la propiedad “type” con el valor “module”. Esto permite que el código fuente siga las mejores prácticas y aproveche las ventajas de los módulos de ES6, como la carga asíncrona y un mejor soporte para herramientas modernas de desarrollo.

Creación del servicio web

Para la creación del servicio web, es necesario realizar la instalación de las siguientes dependencias:

Dependencias utilizadas

- apollo-server
- graphql-tools

Utilice el comando “npm” seguido del nombre de cada una de las dependencias para realizar la instalación. Posteriormente, dentro de la carpeta “controllers”, es necesario crear un archivo con el nombre de **pizza.controller.js**. En este archivo se van a crear cada una de las consultas y las mutaciones utilizando la conexión de la base de datos. En la carpeta “type-system” se debe crear un archivo con el nombre **schema.graphql**; este archivo establece el sistema de tipos que se va a utilizar.

En el Código Fuente 3.3 se muestran los cambios realizados dentro del archivo principal “index.js”. Se observa que se han importado los paquetes de apollo-server y graphql-tools. Además, se han importado los archivos generados para la creación de los resolutores y del sistema de tipos.

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone";
import path from "path";
import { fileURLToPath } from "url";
import { loadFilesSync } from "@graphql-tools/load-files";
import { mergeTypeDefs, mergeResolvers } from "@graphql-tools/merge";

// Obtener la ruta del directorio actual
```

```
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Cargar y combinar los schemas
const allSchemas = mergeTypeDefs(
  loadFilesSync(path.join(__dirname, "../schemas/**/*.graphql"))
);

// Cargar y combinar resolvers
const allResolvers = mergeResolvers(
  loadFilesSync(path.join(__dirname, "../resolvers/**/*.js"))
);

// Configuración del servidor Apollo
const server = new ApolloServer({
  typeDefs: allSchemas,
  resolvers: allResolvers,
});

// Iniciar el servidor Apollo
startStandaloneServer(server, {
  listen: { port: 4000 },
}).then((server) => {
  console.log(` Server ready at: ${server.url}`);
});
```

Código Fuente 3.3: Archivo de index

SISTEMA DE TIPOS

Anteriormente se mencionó que en GraphQL, el sistema de tipos es una parte fundamental de la especificación que define la estructura y la forma en que los datos se describen y se pueden consultar.

Schemas

El esquema creado en GraphQL define un conjunto de consultas (Query) y mutaciones (Mutation) para interactuar con una entidad de pizzas y sus ingredientes asociados.

En el caso de los tipos de objetos tenemos las siguientes entidades:

pizzas: Este tipo de objeto representa una pizza y contiene campos como "id, name, origin, state, total calories e ingredients". Los campos "id" y "name" son obligatorios. El campo "ingredients" es un array que contiene objetos de tipo "ingredients", lo que representa los ingredientes de la pizza.

ingredients: Este tipo de objeto representa un ingrediente y contiene campos como "id, name, calories, state y portion". Los campos "id" y "name" son obligatorios.

En el código fuente 3.4 se muestra el esquema realizado.

```
type pizzas {
  piz_id: Int!
  piz_name: String!
  piz_origin: String!
  piz_description: String
  piz_state: Boolean!
  total_calories: Float
  ingredients: [ingredients]
}

type ingredients {
  ing_id: Int!
  ing_name: String!
  ing_calories: Float!
  ing_state: Boolean!
  pi_portion: Int
}

type Query {
  pizzas(piz_id: Int): [pizzas]
  ingredients(ing_id: Int): [ingredients]
}

type Mutation {
  createPizza(pizza: inputPizza): pizzas
  updatePizza(pizza: updatePizza): pizzas
  deletePizza(id: ID!): String
  createIngredient(ingredient: inputIngredient): ingredients
  updateIngredient(ingredient: updateIngredient): ingredients
  deleteIngredient(id: ID!): String
  deletePizzaIngredient(pizzaIngredient: pizzaIngredient): String
}

input inputPizza {
  piz_name: String!
  piz_origin: String!
  piz_description: String!
  piz_state: Boolean
  ingredientsPizza: [ingredietPortion]
}

input updatePizza {
  piz_id: Int!
  piz_name: String!
  piz_origin: String!
  piz_description: String!
  piz_state: Boolean
  ingredientsPizza: [ingredietPortion]
}

input inputIngredient {
```

```

    ing_name: String!
    ing_calories: Float!
    ing_state: Boolean
    pi_portion: Int
  }

  input updateIngredient {
    ing_id: Int!
    ing_name: String
    ing_calories: Float
    ing_state: Boolean
    pi_portion: Int
  }

  input ingredietPortion {
    ing_id: Int!
    pi_portion: Int
  }

  input pizzaIngredient {
    ing_id: Int
    piz_id: Int
  }

```

Código Fuente 3.4: Esquema realizado

Consultas (Queries):

pizzas(id: Int): [pizzas]: Cuando se realizan consultas para obtener registros, lo conveniente es utilizar el tipo “Query”. La consulta mediante este tipo permite recuperar información sobre pizzas. El parámetro “id” es opcional y se utiliza para buscar una pizza específica por su identificador. Si se proporciona un “id”, la consulta devuelve una lista que contendrá solo una pizza (ya que el “id” debería ser único). Si no se proporciona ningún “id”, la consulta devuelve una lista de todas las pizzas disponibles en el sistema, ver código fuente 3.5.

```

type Query{
  pizzas(id:Int): [pizzas]
  ingredients:[ingredients]
}

```

Código Fuente 3.5: Consulta definida en el esquema

Además del tipo Query, también se puede utilizar el tipo Mutation para las distintas mutaciones que se vayan a realizar mediante el uso de la API en GraphQL. En el caso de las pizzas, principalmente se tienen dos mutaciones para realizar el registro de nuevas pizzas, que son la mutación. **createPizza(pizza: inputPizza): pizza:** Esta mutación se utiliza para crear una nueva pizza en el sistema. Toma un objeto “inputPizza” como argumento que contiene los detalles de la nueva pizza, incluidos su nombre, origen y la lista de ingredientes con sus respectivas porciones. Después de crear la pizza, devuelve un objeto pizzas que representa la pizza recién creada. Para actualizar los campos de las pizzas ya existentes se utiliza la mutación. **updatePizza(pizza: updatePizza): pizza:** Esta mutación se utiliza para actualizar una pizza existente en el sistema. Toma un objeto “updatePizza”, como argumento que contiene los detalles actualizados de la pizza, incluido su identificador, nombre, origen, estado (activo/inactivo) y la lista

actualizada de ingredientes con sus porciones. Después de actualizar la pizza, devuelve un objeto pizzas que representa la pizza actualizada.

El código fuente 3.6 muestra los tipos de mutaciones que fueron definidos en el esquema.

```
type Mutation{
  createPizza(pizza: inputPizza): pizzas
  updatePizza(pizza: updatePizza): pizzas
  deletePizza(id:ID!):String
}
```

Código Fuente 3.6: Mutaciones definidas en el esquema

Además de los tipos que se mencionaron anteriormente, también se establecieron tipos de datos de entrada para las mutaciones para crear y actualizar pizzas. A continuación, se detallan cada uno de los tipos de entrada creados.

inputPizza: Este tipo de entrada se utiliza para proporcionar los datos necesarios para crear una nueva pizza. Contiene campos como el nombre de la pizza “name”, su origen “origin” y una lista de ingredientes “ingredientsPizza”. La lista de ingredientes se representa como un array de objetos de tipo “ingredietPortion”. Esto permite especificar los ingredientes y sus porciones en la nueva pizza.

updatePizza: Este tipo de entrada se utiliza para proporcionar los datos necesarios para actualizar una pizza existente en el sistema. Además de los campos de inputPizza, también incluye el identificador de la pizza “id” y su estado “state”.

ingredietPortion: Este tipo de entrada se utiliza para representar la relación entre un ingrediente y su porción en una pizza. Contiene el identificador del ingrediente “id” y la porción de ese ingrediente en la pizza “portion”.

El código fuente 3.7 muestra los tipos de entradas que se definieron en el esquema.

```
input inputPizza{
  piz_name: String!,
  piz_origin: String!,
  ingredientsPizza: [ingredietPortion]
}

input updatePizza{
  piz_id: Int!,
  piz_name: String!,
  piz_origin: String!,
  piz_state: Boolean,
  ingredientsPizza: [ingredietPortion]
}

input ingredietPortion{
  ing_id: Int!,
  pi_portion: Float
}
```

Código Fuente 3.7: Mutaciones definidas en el esquema

OPERACIONES

Antes de mencionar cada una de las operaciones realizadas, es necesario crear o definir los resolvers “resolvers” que se utilizarán en la API de GraphQL. Los resolvers se van a crear en el archivo “pizza.controller.js” creado anteriormente.

Lo primero que se debe hacer en el archivo “pizza.controller” es requerir el módulo de configuración de la base de datos, que se obtiene mediante la siguiente línea de comando, ver código fuente 3.8.

```
import { db } from "../config/cnn.js";
```

Código Fuente 3.8: Importar conexión de la base de datos

Posteriormente, dentro de la función “pizzaResolver” se debe establecer cada una de las queries y mutaciones que se vayan a requerir. El código fuente 3.9 muestra un resolver para la consulta pizzas. Dependiendo de si se proporciona un “id” como argumento o no, se ejecuta una consulta SQL para seleccionar todas las pizzas o una pizza específica.

```
Query: {
  pizzas(root, { id }) {
    if (id == undefined) {
      return db.any("SELECT * FROM pizzas WHERE piz_state = true;");
    } else {
      return db.any("SELECT * FROM pizzas WHERE piz_id=$1;", [id]);
    }
  },
  ingredients(root, { id }) {
    return db.any("SELECT * FROM ingredients;");
  },
}
```

Código Fuente 3.9: Consultas definidas en el resolver

El código fuente 3.10 muestra los resolvers para las mutaciones **createPizza** y **updatePizza**, que se utilizan para crear y actualizar pizzas en la base de datos, respectivamente. Se ejecutan consultas SQL para realizar estas operaciones.

```
Mutation: {
  async createPizza(root, { pizza }) {
    try {
      if (pizza === undefined) {
        return null;
      } else {
        const newPizza = await db.one(
          `INSERT INTO pizzas(piz_name, piz_origin, piz_state)
            VALUES ($1, $2, true) returning *`,
          [pizza.piz_name, pizza.piz_origin]
        );
        if (pizza.ingredientsPizza.length > 0) {
          pizza.ingredientsPizza.forEach(async (element) => {
            await db.none(
              `INSERT INTO pizzas_ingredients(piz_id, ing_id, pi_portion)
                VALUES ($1, $2, $3);`,
            );
          });
        }
      }
    } catch (error) {
      console.log(error);
    }
  }
}
```

```

        [newPizza.piz_id, element.ing_id, element.pi_portion]
    );
    });
  }
  return newPizza;
}
} catch (error) {
  return error;
}
},
async updatePizza(root, { pizza }) {
  try {
    if (pizza === undefined) {
      return null;
    } else {
      const newPizza = await db.one(
        `UPDATE pizzas
          SET piz_name=$2, piz_origin=$3, piz_state=$4
          WHERE piz_id=$1 returning *`;
        [pizza.piz_id, pizza.piz_name, pizza.piz_origin, pizza.piz_state]
      );
      if (pizza.ingredientsPizza.length > 0) {
        db.none(`DELETE FROM pizzas_ingredients WHERE piz_id=$1`, [
          newPizza.piz_id,
        ]);
        pizza.ingredientsPizza.forEach(async (element) => {
          await db.none(
            `INSERT INTO pizzas_ingredients(piz_id, ing_id, pi_portion)
              VALUES ($1, $2, $3);`,
            [newPizza.piz_id, element.ing_id, element.pi_portion]
          );
        });
      }
      return newPizza;
    }
  } catch (error) {
    return error;
  }
},

async deletePizza(root, value) {
  try {
    if (value === undefined) {
      return null;
    } else {
      const {id } = value;
      if (id > 0) {
        await db.none(`DELETE FROM pizzas_ingredients WHERE piz_id=$1`, [
          id,
        ]).then(async () => {
          await db.none(`DELETE FROM pizzas WHERE piz_id=$1`, [id]);
        });
      }
    }
  }
}
```

```

        });
    }
    return `Pizza ${id} eliminada correctamente`;
}
} catch (error) {
    return error;
}
},
},
},

```

Código Fuente 3.10: Mutaciones definidas en el resolver

A continuación se muestran los resolvers personalizados para el tipo de objeto pizzas. Proporcionan funcionalidades adicionales para obtener los ingredientes de una pizza y calcular las calorías totales de una pizza, ver código fuente 3.11.

```

pizzas: {
  ingredients(pizza) {
    return db.any(
      `select i.*, pi.pi_portion
        from pizzas p, ingredients i, pizzas_ingredients pi
        where p.piz_id = pi.piz_id AND pi.ing_id = i.ing_id
        AND p.piz_id=${1}`;
      [pizza.piz_id]
    );
  },
  async total_calories(pizza) {
    const total_calories = await db.one(
      `select sum(pi.pi_portion * i.ing_calories) total_calories
        from pizzas p, ingredients i, pizzas_ingredients pi
        where p.piz_id = pi.piz_id AND
        pi.ing_id = i.ing_id AND p.piz_id=${1}`, [pizza.piz_id]);
    return total_calories.total_calories;
  },
},
},

```

Código Fuente 3.11: Consultas personalizadas en el resolver

Antes de enseñar cómo se realizan las consultas y mutaciones mediante el servidor de Apollo, es necesario configurar el paquete "package.json". En el caso de que no haya configurado el paquete "nodemon", el código fuente 3.12 muestra rápidamente cómo añadir un script nuevo que contiene el comando "**start**": "**nodemon ./src/index.js**", lo que permitirá utilizar nodemon y evitar reiniciar la API cada vez que se realicen cambios en la aplicación. Para acceder a la API de GraphQL que se creó, es necesario utilizar la siguiente URL: <http://localhost:4000/>. Es importante tener en cuenta que el puerto puede cambiar; no es obligatorio que se encuentre en el puerto 4000. La Fig. 3.14 muestra la inicialización del servidor de Apollo.

```

{
  "name": "api-pizza-graphql",
  "version": "1.0.0",
  "description": "API PIZZ GRAPHQL",
  "main": "index.js",
  "scripts": {

```

```
"start": "nodemon ./src/index.js",
"test": "echo \"Error: no test specified\" && exit 1"
},
"author": "",
"license": "ISC",
"devDependencies": {
  "pg-promise": "^11.5.4"
},
"dependencies": {
  "apollo-server": "^3.13.0",
  "merge-graphql-schemas": "^1.7.8",
  "nodemon": "^3.1.0"
}
} },
```

Código Fuente 3.12: Configuración del package.json

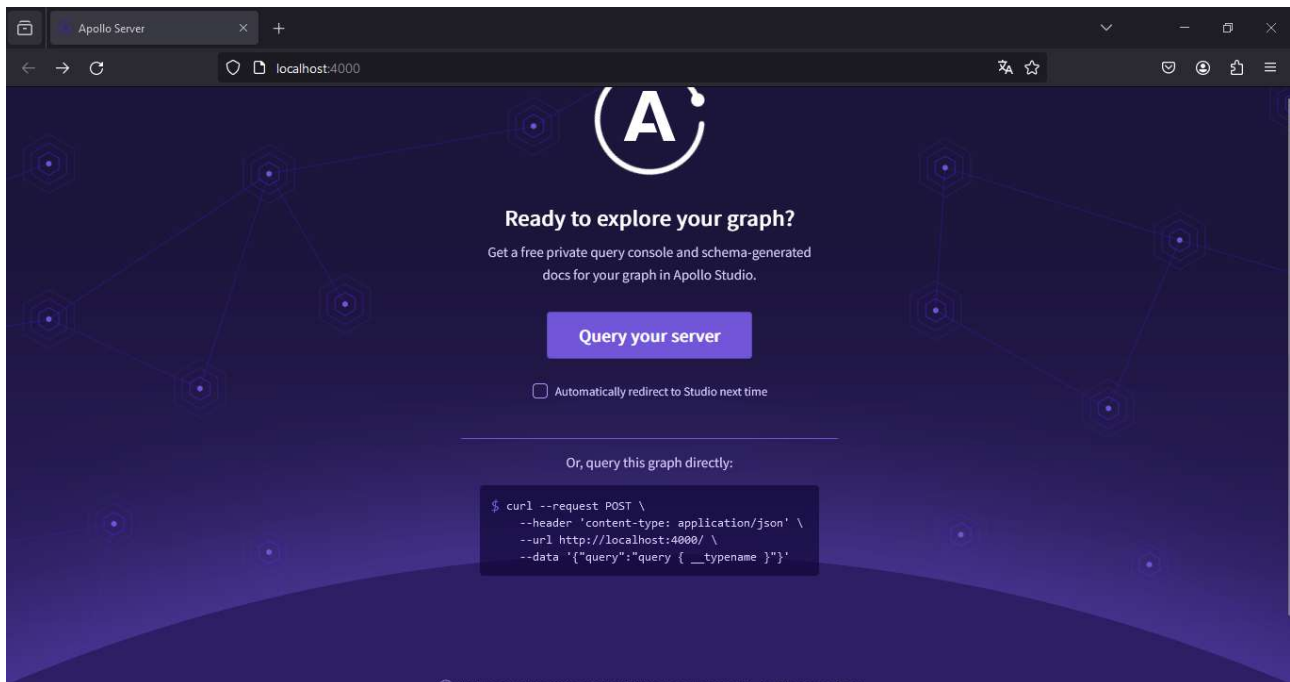


Fig. 3.14. Servidor de Apollo

Consultas

La Fig. 3.15 muestra cómo realizar una consulta simple, mediante Apollo Server. En el caso de este tipo de consultas, se debe especificar el nombre del query; en este caso es pizza. Luego, se especifican los campos que se desean obtener a través de la consulta.

La Fig. 3.16 muestra cómo realizar una consulta con argumentos mediante Apollo Server. En el caso de este tipo de consultas, es necesario establecer cuál será el parámetro y el valor que se le va a asignar. Como resultado de esta consulta se obtiene una pizza específica por el "id".

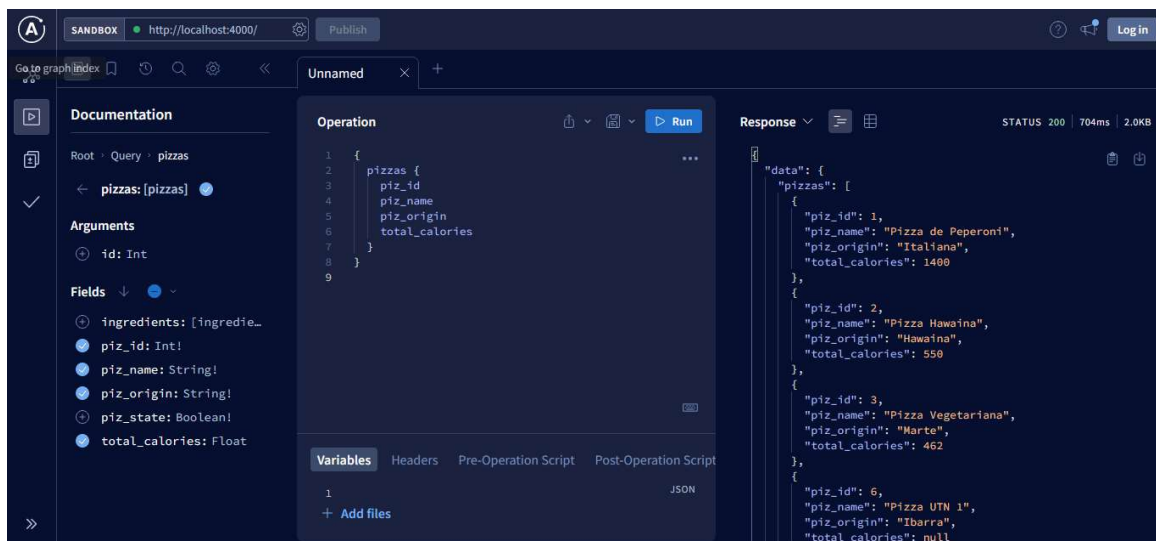


Fig. 3.15. Consulta en Apollo Server

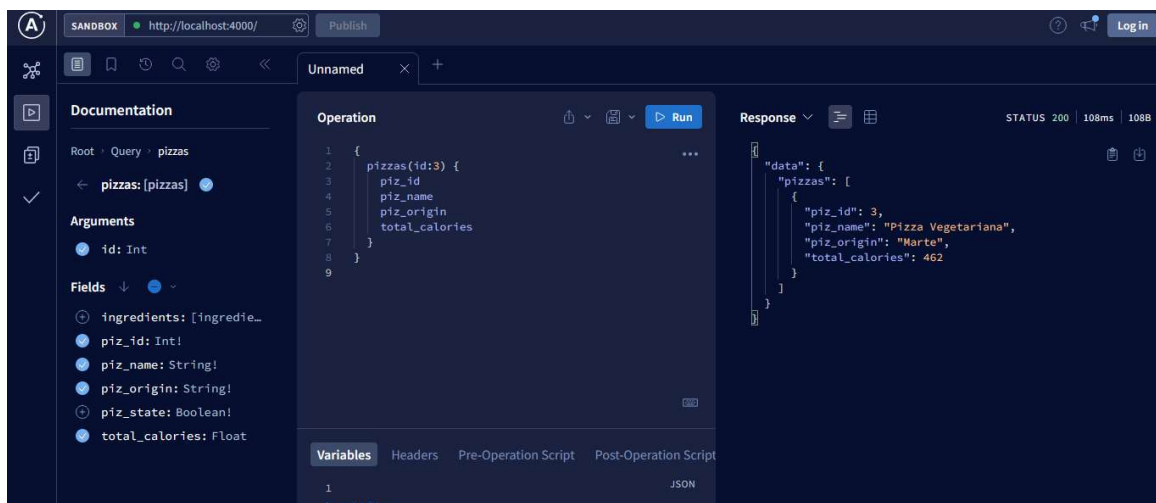


Fig. 3.16. Consulta con argumentos en Apollo Server

Mutaciones

La Fig. 3.17 muestra cómo crear nuevas pizzas a través de la mutación **createPizza**. Esta mutación utiliza el tipo de entrada **Inputpizza** para establecer cada uno de los campos que contiene la entidad "pizza". Existen campos obligatorios como en el caso del nombre y el origen, además de los ingredientes, como se puede ver en el campo de las variables. Donde se detallan los valores de cada uno de los campos que va a tener la nueva pizza y en la pestaña de response se visualiza la respuesta cuando se ejecuta esta mutación.

La Fig. 3.18 muestra cómo actualizar pizzas existentes a través de la mutación **updatePizza**. Esta mutación utiliza el tipo de entrada **updatepizza** para establecer cada uno de los campos que contiene la entidad pizza. Existen campos obligatorios como en el caso del "id". Como se puede ver, en el campo de las variables es donde se detallan los valores de cada uno de los campos que va a tener la pizza actualizada y en la pestaña de response se visualiza la respuesta cuando se ejecuta esta mutación.

A continuación se adjunta el enlace al repositorio de Github del proyecto realizado:

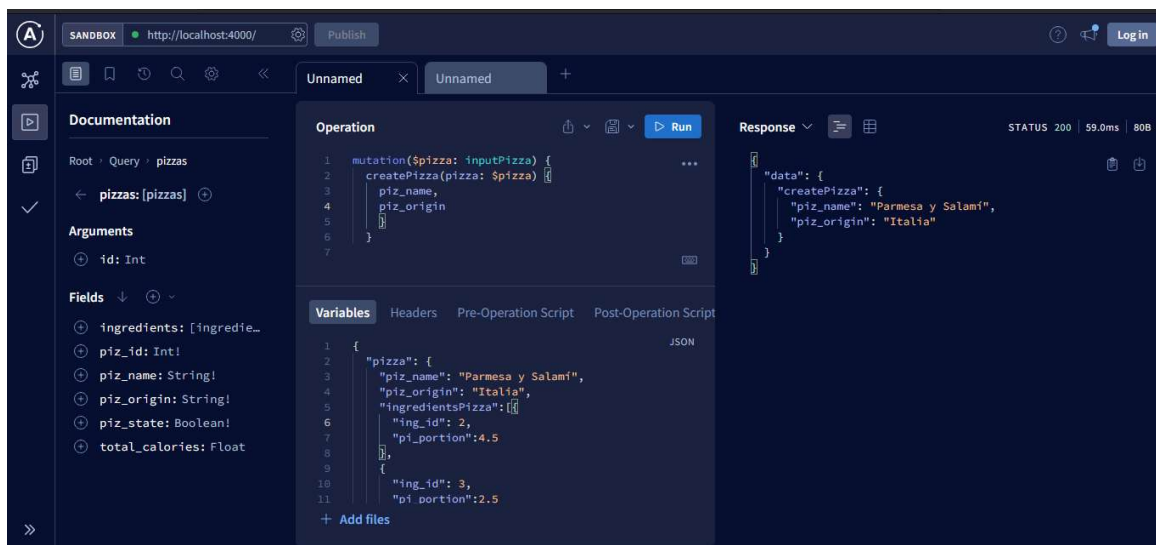


Fig. 3.17. Mutación para crear pizzas en Apollo Server

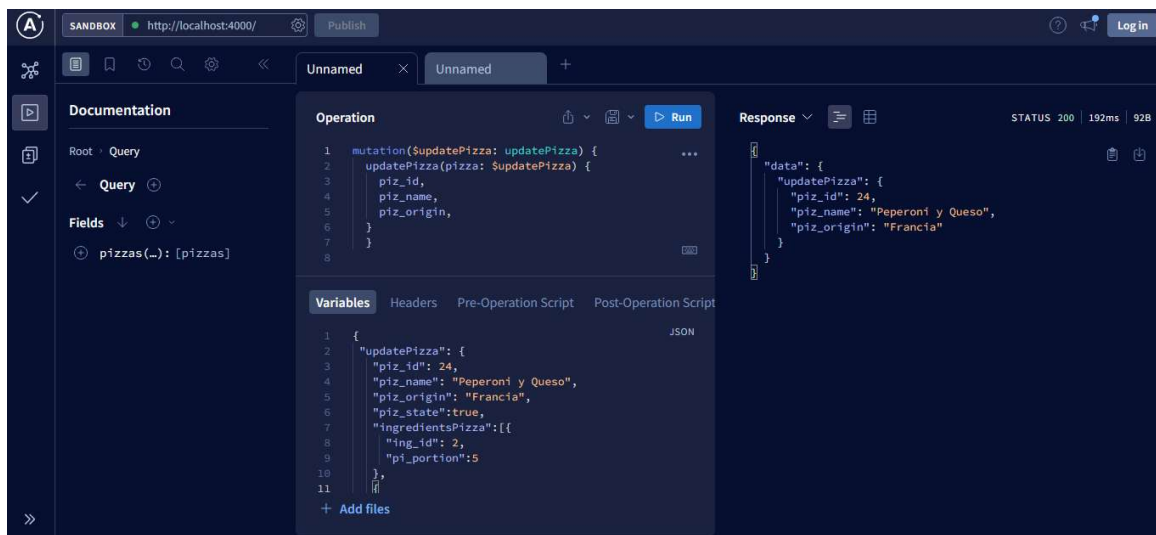


Fig. 3.18. Mutación para actualizar pizzas en Apollo Server

Enlace del repositorio de Github

<https://github.com/antonio-quina/backend-capitulo-03.git>

RESUMEN

EL desarrollo del API con GraphQL evidencia cómo los fundamentos teóricos pueden trasladarse de manera efectiva a aplicaciones concretas. Este capítulo destaca la importancia de contar con un entorno de desarrollo correctamente configurado, donde las herramientas y dependencias estén alineadas con los objetivos del proyecto. En este contexto, la definición clara del esquema de tipos se convierte en un eje central, ya que permite reflejar de forma precisa las entidades del negocio y establecer las bases para un sistema coherente, extensible y mantenible en el tiempo.

A través del caso práctico de una pizzería, se ilustra cómo implementar consultas, mutaciones y suscripciones que materializan operaciones CRUD y facilitan la interacción con bases de datos. Este ejemplo muestra la versatilidad de GraphQL al ofrecer a los clientes exactamente los datos que necesitan, sin redundancias ni sobrecargas. Además, se resalta que la incorporación de suscripciones amplía las posibilidades hacia escenarios en tiempo real, lo cual es fundamental en aplicaciones modernas que requieren inmediatez en la comunicación.

Un aspecto clave abordado en el capítulo es la modularización de los resolvers, que promueve una arquitectura ordenada, flexible y fácil de mantener. Mantener la coherencia en los contratos de datos garantiza que las interacciones entre cliente y servidor se desarrollen sin ambigüedades, reduciendo errores y asegurando la estabilidad de las integraciones. De esta manera, GraphQL no solo se presenta como un lenguaje de consultas, sino también como un marco que impulsa buenas prácticas de diseño y organización del código.

Finalmente, el valor de integrar herramientas colaborativas como GitHub, favorecen el control de versiones, la revisión de código y el trabajo en equipo. La combinación de estos elementos proporciona un marco sólido para establecer las bases de un servidor funcional con GraphQL, al tiempo que prepara el terreno para su escalabilidad y evolución en proyectos de mayor complejidad. Así, este capítulo no solo ofrece una guía técnica, sino también una visión estratégica sobre cómo construir backends modernos, colaborativos y sostenibles.

ACTIVIDADES

Resuelva el siguiente ejercicio

Desarrolle una consulta en GraphQL que permita recuperar todas las pizzas según su origen. La consulta debe ser capaz de filtrar los registros de acuerdo con el origen de procedencia de cada pizza, y mostrar al menos el nombre, origen y lista de ingredientes de cada una.



CAPÍTULO 4

Frontend con GraphQL: Implementación del Cliente y Consumo de APIs

*«El software no solo hace lo que debe hacer,
sino que lo hace de una manera en la que las
personas pueden confiar y disfrutar.»*

Fowler, M., 2018, p. 150.

En este apartado, se explica la integración de GraphQL con uno de los entornos de desarrollo más prominentes, como es “React”. Se estudiará desde la configuración inicial del ambiente hasta la comprensión detallada del sistema de tipos y la estructura del esquema requeridos para habilitar GraphQL en esta plataforma. Al incorporar aspectos esenciales como las operaciones de consultas y mutaciones, este capítulo ofrece una guía completa para aprovechar al máximo las capacidades de GraphQL en el desarrollo de aplicaciones frontend modernas y dinámicas. Además, se examina de cerca la configuración específica necesaria para React, con énfasis en la implementación efectiva de GraphQL. Este apartado proporciona recursos y conocimientos esenciales para los desarrolladores que deseen adoptar esta tecnología en sus proyectos, garantizando una integración fluida y un rendimiento óptimo en sus aplicaciones frontend.

CONFIGURACIÓN DEL AMBIENTE DE DESARROLLO EN REACT

React¹ es una biblioteca de JavaScript de código abierto utilizada principalmente para construir interfaces de usuario interactivas y eficientes, en especial para aplicaciones de una sola página (SPA, por sus siglas en inglés). Desarrollada y mantenida por Facebook (ahora Meta), React permite a los desarrolladores crear componentes reutilizables y gestionar de manera eficiente el estado de la aplicación. Su enfoque basado en componentes y su uso del DOM virtual han revolucionado la forma en que se construyen aplicaciones web modernas, mejorando el rendimiento y la experiencia del usuario [28].

React se integra fácilmente con otras bibliotecas y frameworks, y es ampliamente adoptado en el desarrollo web debido a su flexibilidad y escalabilidad. Además, cuenta con un ecosistema robusto que incluye herramientas como React Router para la gestión de rutas, Redux para la administración del estado

¹ React: <https://es.react.dev/>

global y Next.js para la renderización del lado del servidor (SSR) y la generación de sitios estáticos (SSG) [28].

Características clave de React

DOM virtual: React utiliza un DOM virtual para mejorar el rendimiento. En lugar de manipular directamente el DOM del navegador, React crea una representación virtual ligera del DOM en memoria y lo actualiza de manera eficiente mediante un algoritmo de reconciliación que minimiza las actualizaciones innecesarias [28].

Arquitectura basada en componentes: Las aplicaciones de React se construyen utilizando componentes modulares y reutilizables, lo que facilita la organización y el mantenimiento del código. Cada componente encapsula su propia lógica, estructura y estilos, promoviendo la reutilización del código y la escalabilidad de las aplicaciones [28].

Sintaxis declarativa: React emplea un enfoque declarativo para definir interfaces de usuario. Los desarrolladores describen el estado deseado de la interfaz, y React se encarga de actualizar el DOM de manera eficiente para reflejar los cambios, simplificando la gestión del estado y reduciendo los errores en la aplicación [28].

JSX (JavaScript XML): JSX es una extensión de sintaxis para JavaScript que permite a los desarrolladores escribir estructuras similares a HTML dentro del código JavaScript. JSX mejora la legibilidad del código y facilita la creación de componentes al combinar la lógica y la presentación en un solo archivo. Su sintaxis permite escribir elementos HTML de manera intuitiva y dentro de JavaScript sin necesidad de usar `createElement` o manipular directamente el DOM. Además, JSX puede admitir expresiones de JavaScript dentro de llaves, permitiendo una programación más dinámica y expresiva. Sin embargo, JSX no es comprendido directamente por los navegadores, por lo que debe ser compilado por herramientas como Babel antes de su ejecución [28].

Flujo de datos unidireccional: React sigue un patrón de flujo de datos unidireccional, donde los datos se transmiten de los componentes padres a los componentes hijos a través de *props* (propiedades). Esto garantiza que el estado de la aplicación sea predecible y fácil de depurar, promoviendo una mejor organización y mantenimiento del código [28].

Hooks de React: Introducidos en React 16.8, los Hooks son funciones que permiten a los desarrolladores utilizar estado y otras funcionalidades de React en componentes funcionales sin necesidad de escribir clases. Algunos Hooks comunes incluyen:

- **useState:** Permite gestionar el estado local de un componente de manera sencilla, sin necesidad de definir una clase.
- **useEffect:** Facilita la ejecución de efectos secundarios, como peticiones a APIs o la manipulación del DOM, dentro de los componentes.
- **useContext:** Permite compartir valores entre componentes sin necesidad de pasar *props* manualmente en cada nivel de la jerarquía.
- **useReducer:** Alternativa a `useState` que facilita la gestión de estados complejos mediante la aplicación de una función reductora.
- **useRef:** Proporciona una forma de acceder a elementos del DOM directamente y persistir valores sin causar nuevas renderizaciones.
- **useMemo y useCallback:** Optimizan el rendimiento al evitar ejecuciones innecesarias de funciones y cálculos costosos.

El uso de Hooks ha transformado la forma en que se construyen componentes en React, promoviendo un enfoque funcional y simplificado para la gestión del estado y el ciclo de vida de los componentes [28].

Ecosistema y compatibilidad: React se integra con diversas herramientas y bibliotecas, como Redux para la gestión del estado global, React Router para el manejo de rutas en aplicaciones de una sola página, y frameworks como Next.js para la renderización del lado del servidor y la generación de sitios estáticos. Además, React es compatible con plataformas móviles a través de React Native, lo que permite el desarrollo de aplicaciones nativas con una base de código compartida [28].

Creación del proyecto en React

Vite² es un entorno de desarrollo frontend optimizado que se enfoca en proporcionar una experiencia de desarrollo rápida y eficiente. Fue creado por Evan You, el mismo desarrollador de Vue.js, y está diseñado para reemplazar herramientas más tradicionales como Create React App (CRA) [29]. Su principal objetivo es mejorar el rendimiento durante el desarrollo de aplicaciones web al reducir significativamente los tiempos de compilación y actualización del código.

Una de las principales diferencias entre Vite y otros bundlers como Webpack es su enfoque en el uso de ES Modules (ESM) nativos del navegador. Mientras que Webpack requiere un proceso de compilación completo antes de poder ejecutar la aplicación, Vite aprovecha las capacidades modernas del navegador para cargar módulos de manera eficiente. Esto permite tiempos de arranque casi instantáneos, ya que solo se procesan los módulos realmente necesarios en cada momento, en lugar de compilar toda la aplicación desde el inicio [29].

Además, Vite incorpora un servidor de desarrollo altamente optimizado que ofrece una recarga en caliente (Hot Module Replacement, HMR) extremadamente rápida. Esto significa que los cambios en el código pueden reflejarse en el navegador de inmediato sin necesidad de recargar toda la aplicación, lo que mejora notablemente la productividad del desarrollador [29].

Otra ventaja de Vite es su flexibilidad y compatibilidad con múltiples frameworks frontend. Aunque fue diseñado inicialmente con Vue.js en mente, actualmente ofrece configuraciones optimizadas para React, Svelte y otros entornos modernos. Su ecosistema de plugins basado en la API de Rollup permite extender sus funcionalidades y adaptarlo a diversas necesidades de desarrollo [29].

En términos de rendimiento en producción, Vite utiliza Rollup para la generación de los archivos optimizados, asegurando un código final altamente eficiente y listo para su despliegue. Esto permite que las aplicaciones desarrolladas con Vite no solo sean rápidas en desarrollo, sino también en producción, con tiempos de carga optimizados y una mejor gestión de los recursos estáticos.

Para la creación de proyectos en React se deben seguir los siguientes pasos:

1. Utilizar el comando `npm create vite@latest` para iniciar el proceso de creación del proyecto.

Comando utilizado

```
npm create vite@latest
```

²Vite: <https://vite.dev/>

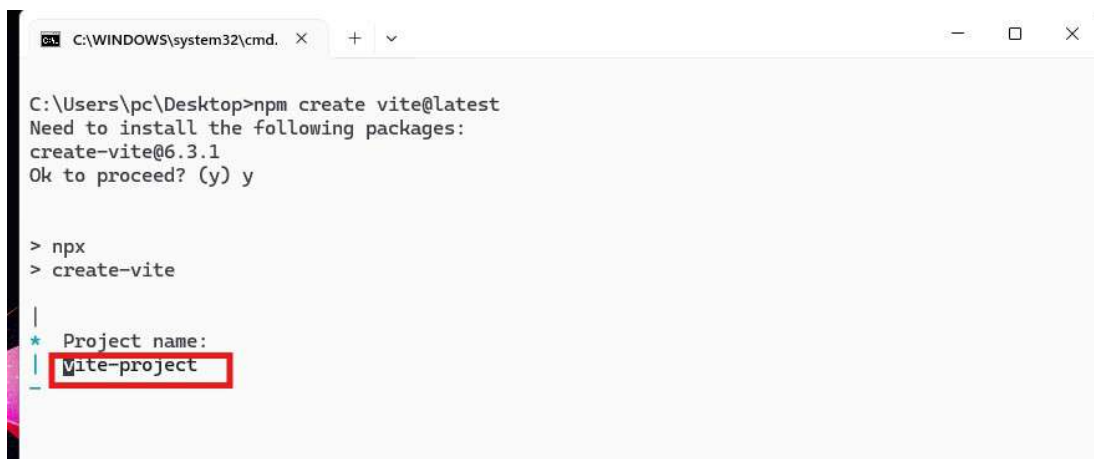
2. En la Fig. 4.1 se puede apreciar la inicialización del proyecto que se utilizará en el frontend.



```
Símbolo del sistema x + - □ x
C:\Users\pc\Desktop>npm create vite@latest
```

Fig. 4.1. Inicialización del proyecto

3. Como siguiente paso se debe dar un nombre al proyecto en la Fig. 4.2 se podrá observar que el nombre utilizado es vite-project pero se debe cambiar por pizza-frontend, se sugiere colocar el mismo nombre para evitar problemas durante la creación del proyecto.



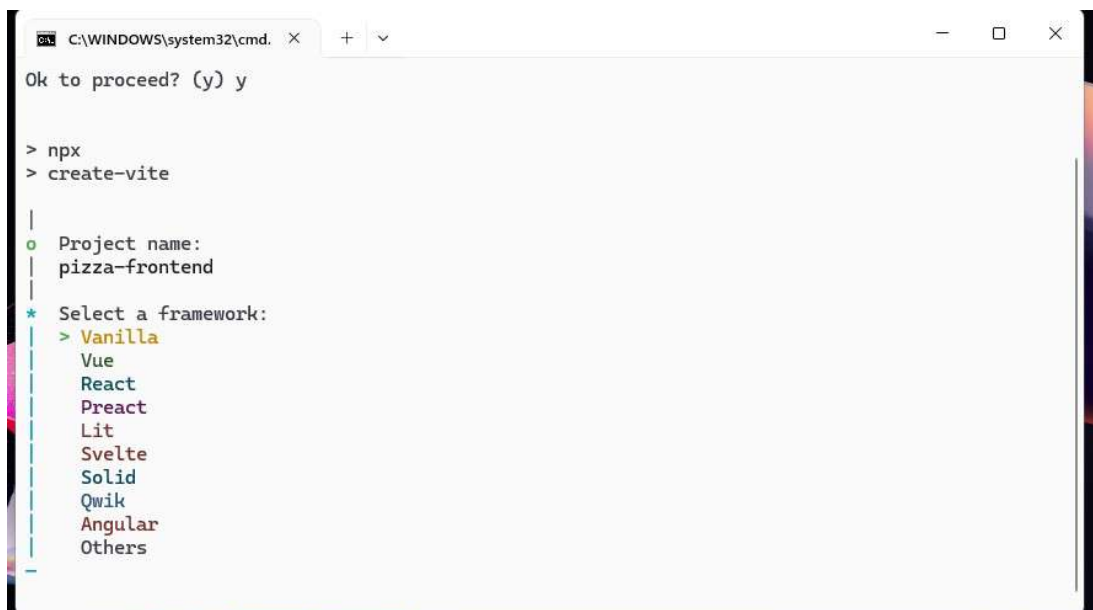
```
C:\WINDOWS\system32\cmd. x + - □ x
C:\Users\pc\Desktop>npm create vite@latest
Need to install the following packages:
create-vite@6.3.1
Ok to proceed? (y) y

> npx
> create-vite

|
| * Project name:
| vite-project
|
| -
```

Fig. 4.2. Configuración de las propiedades del proyecto

4. Luego, es necesario seleccionar el framework. Como se mencionó anteriormente, se utilizará React; ver la Fig. 4.3.



```
C:\WINDOWS\system32\cmd. x + - □ x
Ok to proceed? (y) y

> npx
> create-vite

|
| o Project name:
| pizza-frontend
|
| * Select a framework:
| > Vanilla
|   Vue
|   React
|   Preact
|   Lit
|   Svelte
|   Solid
|   Qwik
|   Angular
|   Others
|
| -
```

Fig. 4.3. Selección del framework del proyecto

- Posteriormente se debe escoger la variante del framework, en este caso se utilizará la opción de javascript debido a que facilita la comprensión y el manejo de React, también es posible utilizarlo con TypeScript; ver la Fig. 4.4.

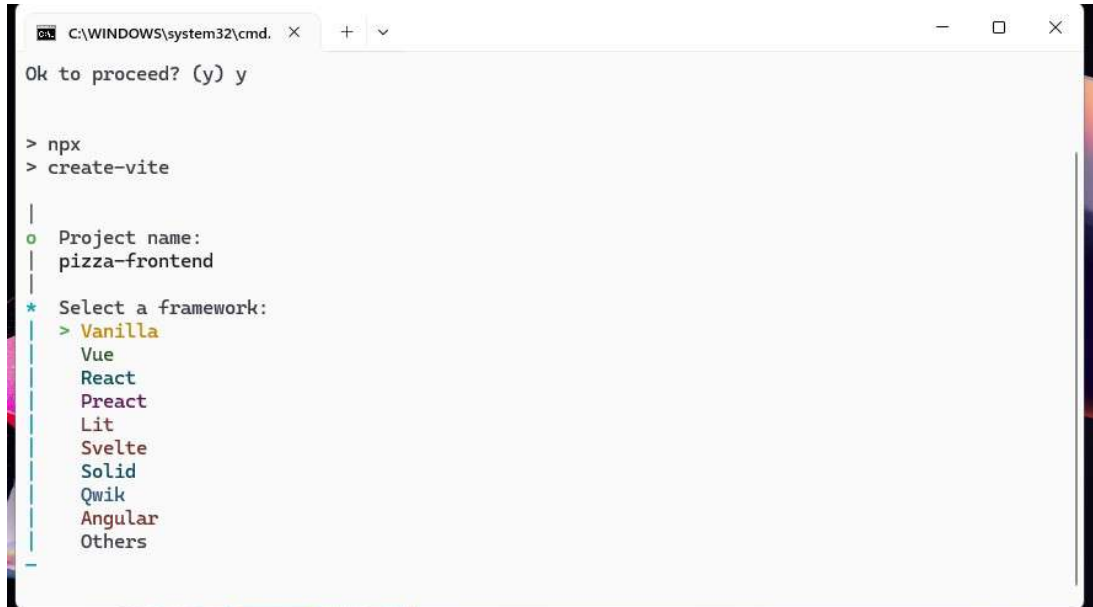


Fig. 4.4. Selección la variante del proyecto

- En la Fig. 4.5 una vez terminada la creación del proyecto se mostrarán los siguientes resultados.

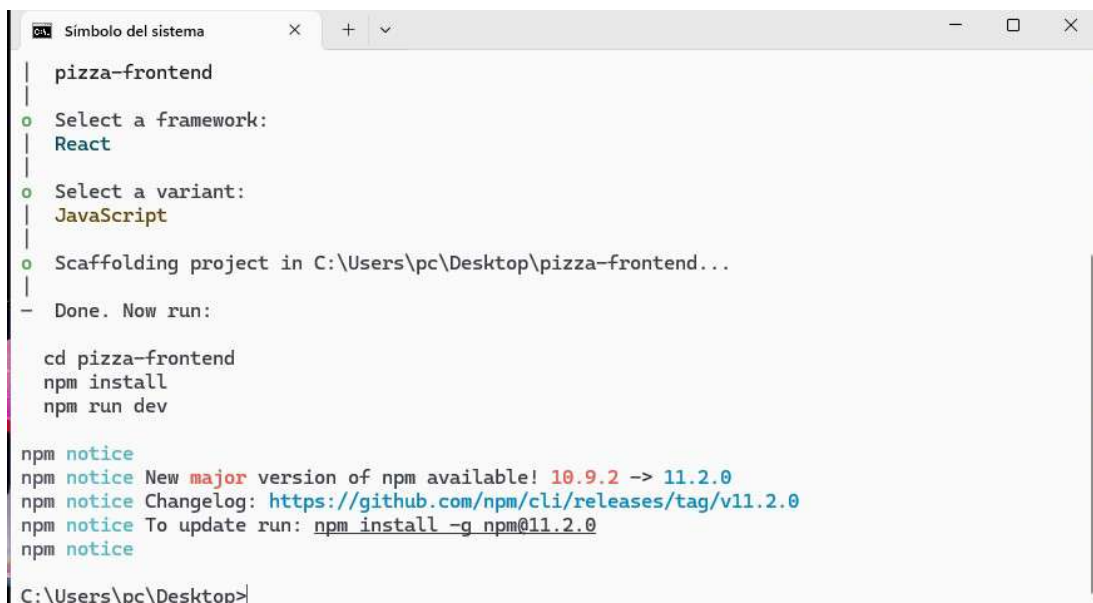


Fig. 4.5. Selección la variante del proyecto

- Para ejecutar el proyecto se debe utilizar un editor de código en este caso será visual studio, una vez que el proyecto esté abierto, primero se debe realizar la instalación de dependencias con el comando **npm i** y posteriormente hay que utilizar el comando **npm run dev** para ejecutar la aplicación, en la siguiente ilustración se puede observar el proyecto ejecutado; ver la Fig. 4.6.

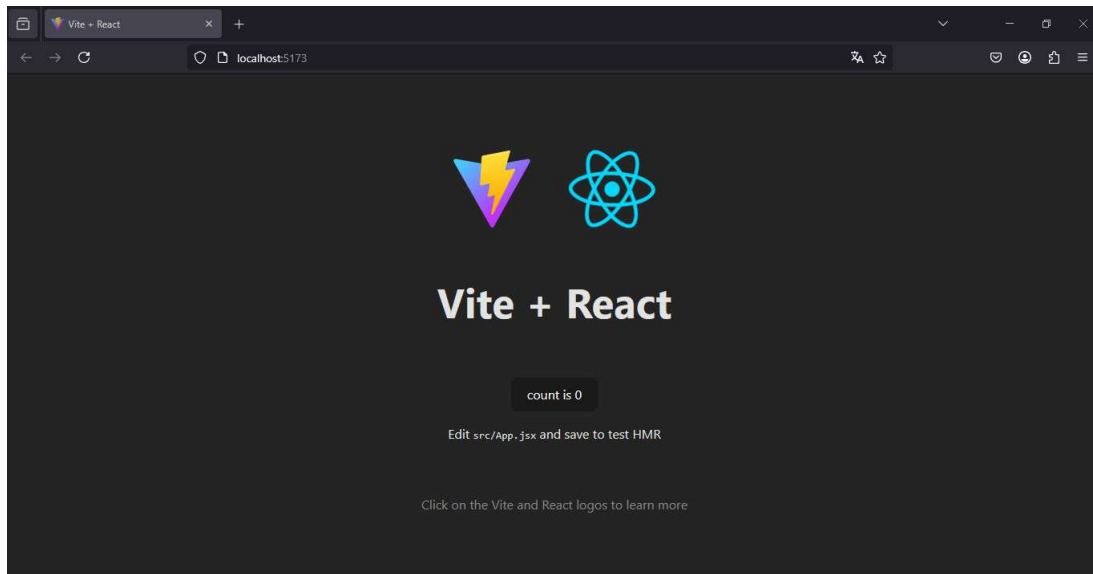


Fig. 4.6. Ejecución del proyecto

8. En la Fig. 4.7 se observa la estructura del proyecto creado el archivo **main.jsx** es el archivo principal del proyecto donde por lo general se agregan la configuraciones necesarias , el archivo **App.jsx** es el componente principal de la aplicación, además también existen archivos css que permiten modificar el estilo de la aplicación.

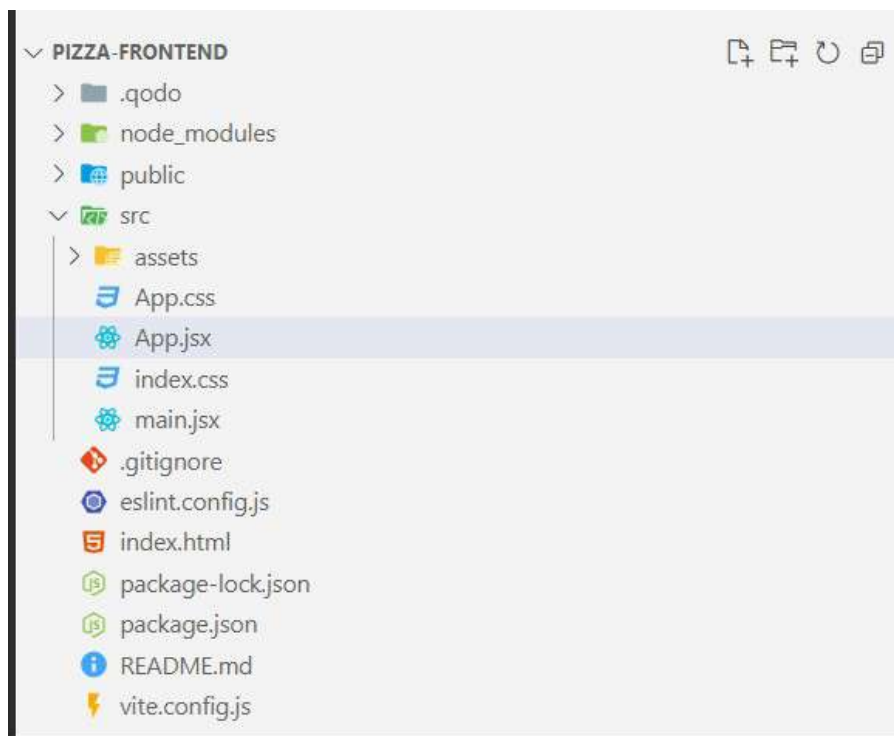


Fig. 4.7. Estructura del proyecto

9. En la Fig. 4.8 se observa cómo debe quedar el archivo **App.jsx** , cabe mencionar que se deben quitar todas las importaciones innecesarias, se puede dejar la importación del archivo App.css para manejar los estilos en la aplicación.

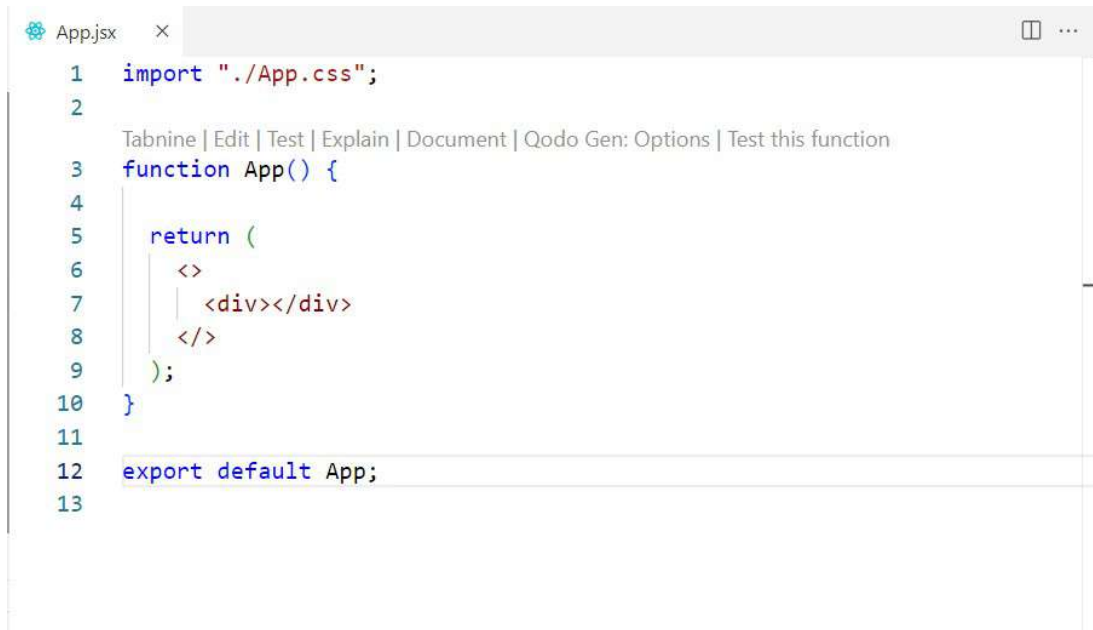


Fig. 4.8. Componente principal del proyecto

10. En la Fig. 4.9 se aprecia que en los archivos css generados se deben borrar los estilos.

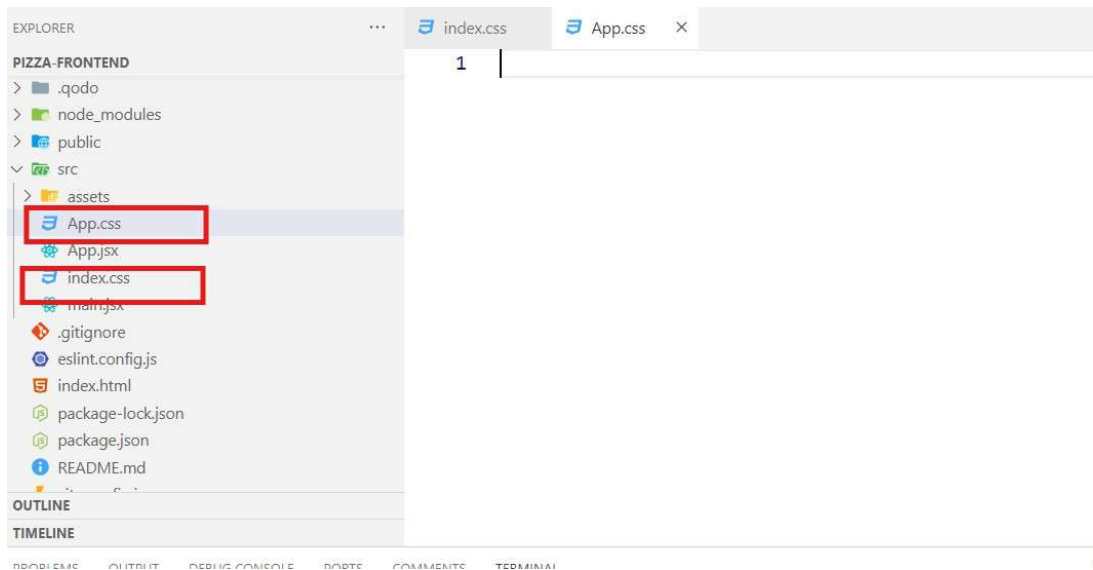


Fig. 4.9. Borrar los estilos del proyecto

Para el uso de los estilos y diseño de interfaces se utilizará **Bootstrap** porque permite crear interfaces de usuario agradables sin necesidad de tener un gran conocimiento en el manejo de estilos y diseño de aplicaciones web. Es necesario dirigirse a la página web de Bootstrap y en la terminal se escribe el siguiente comando:

Comando utilizado

npm i bootstrap

En el mismo contexto, es necesario realizar la instalación de las siguientes dependencias:

react-router-dom es una biblioteca de enrutamiento para aplicaciones React que permite la navegación

entre diferentes vistas o componentes, manteniendo la interfaz sincronizada con la URL. Proporciona una experiencia de usuario fluida al permitir la creación de rutas dinámicas y la gestión del historial de navegación [30].

sweetAlert2 y **sweetalert2-react-content** son bibliotecas utilizadas para crear ventanas modales (pop-ups) personalizadas en aplicaciones web. SweetAlert2 es una versión mejorada de la biblioteca original SweetAlert, que ofrece una experiencia de usuario más atractiva y flexible para mostrar mensajes, alertas, confirmaciones [31]. Sweetalert2-react-content proporciona una integración específica para React [32].

@fortawesome/fontawesome-free es un paquete que contiene iconos de Font Awesome de forma gratuita. Font Awesome es una biblioteca de íconos ampliamente utilizada en desarrollo web que proporciona una amplia gama de iconos para su uso en proyectos web y aplicaciones [33].

Los siguientes comandos permiten instalar las dependencias mencionadas:

Comando utilizado

- `npm i react-router-dom sweetalert2 sweetalert2-react-content`
- `npm i @fortawesome/fontawesome-free`

Es necesario importar las dependencias instaladas en el archivo “main.jsx”. Antes de continuar con la creación de los componentes, se deben crear los servicios que permitan realizar las distintas mutaciones y consultas.

En la Fig. 4.10 se muestra las importaciones realizadas en el archivo main.jsx.



```
main.jsx x
1 import { StrictMode } from 'react'
2 import { createRoot } from 'react-dom/client'
3 import './index.css'
4 import App from './App.jsx'
5
6 import 'bootstrap/dist/css/bootstrap.min.css';
7 import '@fortawesome/fontawesome-free/css/all.min.css';
8 import 'bootstrap/dist/js/bootstrap.bundle';
9
```

Fig. 4.10. Importación de dependencias

OPERACIONES

Creación del servicio message-service

Para la creación de este servicio es necesario seguir los siguientes pasos:

- Dentro de la carpeta “service” se debe crear un archivo con el nombre **message-service.js**.
- Se debe realizar la importación de la librería “sweetalert2” como se observa a continuación.

Importación realizada

```
import Swal from "sweetalert2";
```

- El siguiente paso consiste en definir los métodos necesarios para mostrar los mensajes a los usuarios y realizar su correcta exportación. Estos métodos permitirán gestionar la visualización de alertas y notificaciones de manera eficiente dentro de la aplicación. En el código fuente 4.1, se presenta la implementación completa de este servicio, incluyendo la configuración y el uso de la biblioteca SweetAlert para garantizar una experiencia de usuario más interactiva e intuitiva.

```
import Swal from "sweetalert2";

const showMessage = (mensaje, tipo) => {
  Swal.fire({
    text: mensaje,
    icon: tipo,
  });
};

const showConfirmMessage = (mensaje) => {
  return Swal.fire({
    title: mensaje,
    text: "No podra revertir esta accion !",
    icon: "warning",
    showCancelButton: true,
    confirmButtonText: "Si ",
    cancelButtonText: "No ",
    confirmButtonColor: "green",
    cancelButtonColor: "red"
  });
};

export { showMessage, showConfirmMessage };
```

Código Fuente 4.1: Servicio para mostrar mensajes

Configuración de Apollo Client en React

Para realizar las distintas operaciones como consultas y mutaciones es necesario instalar la librería de GraphQL para React; el siguiente comando permite realizar la instalación:

Comando utilizado

```
npm install @apollo/client graphql
```

Apollo Client es una biblioteca que permite la integración de aplicaciones React con servidores GraphQL. A continuación, se describe la configuración de Apollo Client en un archivo de React.

Importaciones necesarias

Se importan los módulos requeridos de Apollo Client para establecer la conexión con el servidor:

- **ApolloClient**: Clase principal para inicializar Apollo Client.
- **ApolloProvider**: Componente de React que proporciona el cliente de Apollo a toda la aplicación.
- **HttpLink**: Configura la conexión HTTP con el servidor GraphQL.

- **InMemoryCache:** Implementa la caché en memoria para almacenar los resultados de las consultas y mejorar el rendimiento.

Creación del cliente de Apollo

Se instancia un objeto de Apollo Client con los siguientes parámetros:

- **cache:** `new InMemoryCache()`: Usa una caché en memoria para evitar consultas innecesarias y mejorar el rendimiento.
- **link:** `new HttpLink(uri: 'http://localhost:4000')`: Define la URL del servidor GraphQL, que en este caso es `http://localhost:4000`.

En el código fuente 4.2 se observa la creación del cliente de Apollo.

```
cache: new InMemoryCache(),
link: new HttpLink({
  uri: 'http://localhost:4000',
})
})
```

Código Fuente 4.2: Creación del cliente de Apollo

Integración con React

Para que la aplicación pueda acceder a Apollo Client, se usa el componente ApolloProvider para envolver la aplicación tal como se aprecia en el código fuente 4.3, esto garantiza que todos los componentes hijos de App puedan realizar consultas GraphQL.

```
<StrictMode>
<ApolloProvider client={client}>
  <App />
</ApolloProvider>,
</StrictMode>,
)
```

Código Fuente 4.3: Integración con React

En el código fuente 4.4 se observa la configuración que permite que una aplicación React se conecte a un servidor GraphQL, realice consultas y mutaciones, y gestione los datos de manera eficiente mediante Apollo Client.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

import 'bootstrap/dist/css/bootstrap.min.css';
import '@fortawesome/fontawesome-free/css/all.min.css';
import 'bootstrap/dist/js/bootstrap.bundle';

import {
  ApolloClient, ApolloProvider, HttpLink, InMemoryCache
} from '@apollo/client'

const client = new ApolloClient({
```



```
cache: new InMemoryCache(),
link: new HttpLink({
  uri: 'http://localhost:4000',
})
})

createRoot(document.getElementById('root')).render(
  <StrictMode>
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  </StrictMode>,
)
```

Código Fuente 4.4: Configuración del cliente de Apollo

Consultas

Para llevar a cabo operaciones como consultas en GraphQL, es fundamental definir un servicio que gestione y organice las consultas que se emplearán en la aplicación. Este servicio actuará como un punto central donde se establecerán las consultas necesarias para interactuar con el servidor de GraphQL de manera eficiente y estructurada.

Creación del servicio pizza-service

- El primer servicio que se va a crear permitirá que se puedan realizar mutaciones y consultas para el componente de pizzas. Como primer paso, es necesario crear un servicio con el nombre **pizza-service.js** en la carpeta **services**.
- El segundo paso, es importar las siguientes dependencias en el componente pizzas.

Importación realizada

```
import gql, useQuery, useMutation from "@apollo/client";
```

- Luego se deben añadir las consultas que se desean realizar, para lo cual es necesario utilizar el paquete "gql" para describir las consultas y las mutaciones. Cabe mencionar que dentro de este servicio se crearon métodos para listar, agregar y eliminar los ingredientes de las pizzas; ver código fuente 4.5.

```
import { gql } from "@apollo/client";
import { showMessage } from "../message-service";

const GET_PIZZAS = gql`
{
  pizzas {
    piz_id
    piz_name
    piz_origin
    piz_description
    piz_state
  }
}
```

```
      ingredients {
        ing_name
        ing_calories
        ing_state
        ing_id
        pi_portion
      }
    }
  }
};

const CREATE_PIZZA = gql`
  mutation createPizza($pizza: inputPizza) {
    createPizza(pizza: $pizza) {
      piz_name
      piz_origin
      piz_description
      piz_state
      ingredients {
        ing_id
        pi_portion
      }
    }
  }
`;

const UPDATE_PIZZA = gql`
  mutation UpdatePizza($pizza: updatePizza) {
    updatePizza(pizza: $pizza) {
      piz_id
      piz_name
      piz_description
      piz_origin
    }
  }
`;

const DELETE_PIZZA = gql`
  mutation deletePizza($deletePizzaId: ID!) {
    deletePizza(id: $deletePizzaId)
  }
`;

const DELETE_PIZZA_INGREDIENT = gql`
  mutation deletePizzaIngredient($pizzaIngredient: pizzaIngredient) {
    deletePizzaIngredient(pizzaIngredient: $pizzaIngredient)
  }
`;

export {
  DELETE_PIZZA,
```

```
GET_PIZZAS,  
UPDATE_PIZZA,  
CREATE_PIZZA,  
DELETE_PIZZA_INGREDIENT  
};
```

Código Fuente 4.5: Definición de las consultas y mutaciones en el servicio pizza-service

Posteriormente en el mismo servicio se debe agregar y exportar las siguientes funciones adicionales como se aprecia en el código fuente 4.6 las cuales son las siguientes:

deleteIngredient(item, Ingredientlist)

Esta función elimina un ingrediente específico de una lista de ingredientes. Compara el `ing_id` del ingrediente a eliminar con los elementos de la lista y crea una nueva lista sin dicho ingrediente. Finalmente, devuelve la lista actualizada sin modificar la original.

ListSelectedIngredients(Ingreientlist)

Genera una nueva lista que contiene únicamente dos propiedades de cada ingrediente: `ing_id` y `pi_portion`. Se utiliza para simplificar la estructura de datos y enfocarse en los elementos esenciales al procesar los ingredientes seleccionados.

getTotalCalories(ingredients)

Calcula el total de calorías de una lista de ingredientes. Para cada ingrediente, multiplica su valor `ing_calories` por la cantidad especificada en `pi_portion` y suma los resultados. Si la lista está vacía o no es válida, devuelve 0.

addIngredient(IngredientsList, quantity, newIngredient)

Busca un ingrediente en una lista de datos utilizando su `ing_id`. Si la cantidad ingresada es 0, muestra un mensaje de advertencia. Si la cantidad es válida, crea un objeto con los datos del ingrediente encontrado y la cantidad especificada, luego lo devuelve.

showIngredient(ingredients)

Construye una cadena de texto con los nombres de todos los ingredientes en la lista, separados por espacios. Si la lista de ingredientes está vacía o no definida, devuelve la cadena "Sin ingredientes" para indicar la ausencia de elementos.

```
import { gql } from "@apollo/client";  
import { showMessage } from "../message-service";  
  
const deleteIngredient = (item, Ingredientlist) => {  
  let ingredients = [];  
  for (let index = 0; index < Ingredientlist.length; index++) {  
    const element = Ingredientlist[index];  
    if (element.ing_id !== item.ing_id) {  
      ingredients.push(element);  
    }  
  }  
  return ingredients;  
};
```

```

const ListSelectedIngredients = (Ingredientlist) => {
  let result = [];
  for (let index = 0; index < Ingredientlist.length; index++) {
    const element = Ingredientlist[index];
    result.push({
      ing_id: element.ing_id,
      pi_portion: element.pi_portion,
    });
  }
  return result;
};

const getTotalCalories = (ingredients) => {
  let total = 0;
  if (ingredients !== undefined) {
    for (let index = 0; index < ingredients.length; index++) {
      const element = ingredients[index];
      total += element.ing_calories * element.pi_portion;
    }
  }
  return total;
};

const addIngredient = (IngredientsList, quantity, newIngredient) => {
  const ingredient = {
    ing_id: 0,
    ing_name: "",
    ing_calories: 0,
    pi_portion: 0,
    ing_state: false,
  };

  let foundIngredient = IngredientsList.data?.ingredients.find(
    (ingredient) => ingredient.ing_id == parseInt(newIngredient)
  );
  if (quantity == 0) {
    showMessage("Debe ingresar la cantidad", "warning");
  } else {
    ingredient.ing_id = foundIngredient.ing_id;
    ingredient.ing_name = foundIngredient.ing_name;
    ingredient.ing_calories = foundIngredient.ing_calories;
    ingredient.ing_state = foundIngredient.ing_state;
    ingredient.pi_portion = quantity;
  }

  return ingredient;
};

const showIngredient = (ingredients) => {
  let result = "";
  if (ingredients !== undefined) {

```

```

    for (let index = 0; index < ingredients.length; index++) {
      const element = ingredients[index];
      result += " " + element.ing_name + " ";
    }
    if (ingredients.length === 0) {
      result = "Sin ingredientes";
    }
  }
  return result;
};

export {
  DELETE_PIZZA,
  GET_PIZZAS,
  UPDATE_PIZZA,
  CREATE_PIZZA,
  DELETE_PIZZA_INGREDIENT,
  getTotalCalories,
  deleteIngredient,
  ListSelectedIngredients,
  addIngredient,
  showIngredient,
};

```

Código Fuente 4.6: Servicio adicionales

Creación del servicio ingredient-service

- El primer servicio que se va a crear permitirá que se puedan realizar mutaciones y consultas para el componente de ingredientes. Como primer paso, es necesario crear un servicio con el nombre **ingredient-service.js**.
- El segundo paso es importar las siguientes dependencias en el componente ingredientes.

Importación realizada

```
import gql, useQuery, useMutation from "@apollo/client";
```

- Luego se deben añadir las consultas que se desean realizar. Para lo cual es necesario utilizar el paquete "gql" para describir las consultas y las mutaciones; ver código fuente 4.7.

```

import { gql } from "@apollo/client";
const GET_INGREDIENTS = gql`
{
  ingredients {
    ing_id
    ing_name
    ing_calories
    ing_state
  }
}
`

```

```

    }
  `;

const CREATE_INGREDIENTS = gql`
  mutation CreateIngredient($ingredient: inputIngredient) {
    createIngredient(ingredient: $ingredient) {
      ing_name
      ing_state
      ing_calories
    }
  }
  `;

const UPDATE_INGREDIENTS = gql`
  mutation updateIngredient($ingredient: updateIngredient) {
    updateIngredient(ingredient: $ingredient) {
      ing_id
      ing_name
      ing_state
      ing_calories
    }
  }
  `;

const DELETE_INGREDIENTS = gql`
  mutation deleteIngredient($deleteIngredientId: ID!) {
    deleteIngredient(id: $deleteIngredientId)
  }
  `;

const ListActiveIngredients = (list) =>{
  return list.map(item => item.ing_state === true);
}

export {
  GET_INGREDIENTS,
  CREATE_INGREDIENTS,
  UPDATE_INGREDIENTS,
  DELETE_INGREDIENTS ,
  ListActiveIngredients
};

```

Código Fuente 4.7: Definición de las consultas y mutaciones en el servicio ingredient-service

Creación del servicio util-service

En ciertas ocasiones es conveniente crear un servicio que se pueda utilizar en varios componentes con el propósito de reutilizar el código fuente. En el siguiente servicio se van a crear funciones generales.

En el código fuente 4.8 se observa las funciones que se crearon en este servicio.


```
const isAvailable = (estado) => {  
  let opcion = "Inactivo";  
  if (estado) {  
    opcion = "Activo";  
  }  
  return opcion;  
};  
  
export { isAvailable };
```

Código Fuente 4.8: Creación del servicio util-service

CREACIÓN DE COMPONENTES

Con el fin de evitar problemas relacionados con los estilos que se utilizarán dentro de los componentes, se recomienda revisar el código fuente 4.9 del archivo CSS, el cual se debe colocar en el archivo "index.css".

```
.main-container {  
  background-image: url("../assets/images/bg_1.jpg");  
  background-color: #270505 !important;  
}  
  
.img {  
  margin-bottom: 50px;  
  text-align: center;  
}  
  
.title-page {  
  color: #fff !important;  
  font-size: 50px;  
  position: relative;  
  bottom: -150px;  
  background-color: #000;  
  width: 55%;  
  border-radius: 10px;  
  border: solid 5px #fff;  
  padding-top: 6px;  
  padding-bottom: 6px;  
  padding-right: 10px;  
}  
  
.title {  
  color: #fff !important;  
  font-size: 28px;  
  background-color: #000;  
  width: 20%;  
  border-radius: 10px;
```

```
    border: solid 5px #fff;
    padding-top: 6px;
    padding-bottom: 6px;
    padding-left: 20px;
    padding-right: 20px;
}

.card-container {
    background: transparent !important;
    margin-left: 60px;
    margin-top: 20px;
}

.link {
    color: #fff !important;
    font-size: 16px;
    cursor: pointer;
}

.link:hover {
    color: goldenrod !important;
    font-weight: bold;
}

footer {
    background-color: #000;
    width: 100%;
    position: relative;
    bottom: -50px;
}

.footer-title {
    color: #fff;
}

body::-webkit-scrollbar {
    width: 12px;
}

body::-webkit-scrollbar-track {
    background: rgb(188, 188, 188);
}

body::-webkit-scrollbar-thumb {
    background-color: rgb(62, 62, 63);
    border-radius: 20px;
    border: 3px solid rgb(188, 188, 188);
}

body {
```

```
background: linear-gradient(to right, #ACB6E5, #c3fdf3);
/* W3C, IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera 12+, Safari 7+ */

}

input {
  border: solid 1px #777777 !important;
}
```

Código Fuente 4.9: Archivo CSS para los estilos

Creación del componente shared

El código fuente 4.10. es el componente que será la barra de navegación hacia las secciones.

- A continuación se deben realizar las siguientes importaciones:

Importación realizada

```
import React from "react";
import Link from "react-router-dom"
```

El componente "Link" permite crear un enlace para facilitar la navegación hacia una ruta específica.

- Crear la función para el componente shared.

```
const NavBar = () => {
  return (
    <nav
      className="navbar bg-dark border-bottom border-body"
      data-bs-theme="dark"
    >
      <div className="container-fluid">
        <span className="navbar-brand mb-0 h1">Pizza App</span>

        <ul className="nav justify-content-end">
          <li className="nav-item">
            <Link to="/" className="nav-link link">Inicio</Link>
          </li>
          <li className="nav-item">
            <Link to="/pizzas" className="nav-link link">Pizzas</Link>
          </li>
          <li className="nav-item">
            <Link to="/ingredientes" className="nav-link link">Ingredientes</Link>
          </li>
        </ul>
      </div>
    </nav>
  )
}

export default NavBar
```

Código Fuente 4.10: Creación del componente shared

Creación del componente home

Este componente permite que se muestre una página de bienvenida. Para continuar es necesario crear un archivo con el nombre **home.jsx**, luego se deben realizar los siguientes pasos:

- Realizar la importación de las dependencias.

Importación realizada

```
import React from "react".
import NavBar from "../shared/navbar"
```

• Como siguiente paso es necesario crear una función para poder exportar el componente que se va a utilizar. Cabe mencionar que dentro de esta función se debe colocar el código HTML para la vista de la página de inicio; ver código fuente 4.11.

```
const Inicio = () => {
  return (
    <div className="main-container">
      <NavBar></NavBar>
      <div className="card mb-3 card-container">
        <div className="row g-0">
          <div className="col-md-4">
            <img
              src={require("../assets/images/bg_1.png")}
              className="img-fluid rounded-start"
              alt="..."
            />
          </div>
          <div className="col-md-8">
            <div className="card-body">
              <h5 className="card-title title-page">Bienvenidos</h5>
            </div>
          </div>
        </div>
        <div>
          <br></br>
          <footer className="d-flex
flex-wrap justify-content-between
align-items-center py-3 my-4 border-top">
            <p className="col-md-4 mb-0 footer-title">2024 Pizza App </p>
            <a
              href="/"
              className="col-md-4
d-flex align-items-center
justify-content-center
mb-3 mb-md-0 me-md-auto
link-dark text-decoration-none"
            ></a>
          </footer>
        </div>
      </div>
    </div>
  )
}
```

```

    </div>
  );
};

export default Inicio;

```

Código Fuente 4.11: Creación del componente home

Creación del componente pizza-form

Este componente debe ubicarse dentro de la carpeta "pizza", la cual se encuentra dentro del directorio "components". Su principal función es gestionar las operaciones relacionadas con las pizzas, incluyendo la inserción, actualización y eliminación de registros. A través de este componente, se facilitará la interacción con los datos de las pizzas, permitiendo que los usuarios realicen modificaciones de manera sencilla y eficiente dentro de la aplicación.

Para la creación de este componente es importante seguir las siguientes instrucciones:

- Realizar las siguientes importaciones; ver código fuente 4.12.

```

import React, { useEffect, useState } from "react";
import { useQuery, useMutation } from "@apollo/client";
import {
  GET_PIZZAS,
  CREATE_PIZZA,
  UPDATE_PIZZA,
  DELETE_PIZZA_INGREDIENT,
  getTotalCalories,
  deleteIngredient,
  addIngredient,
  ListSelectedIngredients,
} from "../../services/pizza-service";
import { GET_INGREDIENTS } from "../../services/ingredient-service";
import { showMessage } from "../../services/message-service";

```

Código Fuente 4.12: Importaciones del componente pizza-form

- Crear la función para exportar el componente. Es necesario colocar un parámetro para obtener la pizza seleccionada cuando se van a actualizar sus datos; ver código fuente 4.13.

```

const PizzaForm = ({ pizza }) => {

}

export default PizzaForm;

```

Código Fuente 4.13: Función para exportar el componente

- Declarar las variables para cada uno de los campos del formulario de pizzas. En el caso de "React", es necesario utilizar el hook useState para modificar y actualizar el valor de las variables que se crearon; ve código fuente 4.14.

```
const PizzaForm = ({ pizza }) => {
  const IngredientsList = useQuery(GET_INGREDIENTS);
  const [pizzaIngredients, setpizzaIngredients] = useState([]);
  const [newIngredient, setnewIngredient] = useState(0);
  const [name, setName] = useState("");
  const [description, setDescription] = useState("");
  const [origin, setOrigin] = useState("");
  const [id, setId] = useState(0);
  const [quantity, setQuantity] = useState(0);
  const [state, setState] = useState(false);
}

export default PizzaForm;
```

Código Fuente 4.14: Declaración de variables

- Posteriormente hay que utilizar el hook useEffect para actualizar el valor del parámetro pizza cada vez que se actualice; ver código fuente 4.15.

```
useEffect(() => {
  if (pizza !== null) {
    setpizzaIngredients(pizza.ingredients);
    setName(pizza.piz_name);
    setOrigin(pizza.piz_origin);
    setId(pizza.piz_id);
    setState(pizza.piz_state);
    setDescription(pizza.piz_description);
  }
}, [pizza]);
```

Código Fuente 4.15: Uso del hook useEffect

- El método AddIngredients se encarga de agregar un ingrediente a la lista de ingredientes de una pizza, asegurando que la selección sea válida y que el ingrediente esté disponible antes de incorporarlo; ver código fuente 4.16.

```
const AddIngredients = () => {
  const ingredient = addIngredient(
    IngredientsList,
    parseInt(quantity),
    newIngredient
  );
  if (ingredient.ing_id === 0) {
    showMessage("Debe seleccionar un ingrediente", "warning");
    setnewIngredient(0);
    setQuantity(0);
  } else if (!ingredient.ing_state) {
    showMessage("Ingrediente no disponible", "warning");
    setnewIngredient(0);
    setQuantity(0);
  } else {
    showMessage("Ingrediente agregado correctamente", "success");
  }
}
```



```
    setpizzaIngredients([...pizzaIngredients, ingredient]);
    setnewIngredient(0);
    setQuantity(0);
  }
};
```

Código Fuente 4.16: Métodos para agregar ingredientes

- El método DeleteIngredients gestiona la eliminación de ingredientes de una pizza, verificando si la pizza ya está registrada en la base de datos antes de proceder con la eliminación; ver código fuente 4.17.

```
const DeleteIngredients = (item) => {
  if (id > 0) {
    deletePizzaIngredient({
      variables: {
        pizzaIngredient: {
          ing_id: item.ing_id,
          piz_id: id,
        },
      },
    }).then(() => {
      setpizzaIngredients(deleteIngredient(item, pizzaIngredients));
    });
  } else {
    setpizzaIngredients(deleteIngredient(item, pizzaIngredients));
  }
};
```

Código Fuente 4.17: Métodos para eliminar ingredientes

- Es importante crear un método que permita limpiar el formulario una vez que se hayan registrado nuevas pizzas. Para lo cual se utiliza el siguiente método; ver código fuente 4.18.

```
const CleanForm = () => {
  setQuantity(0);
  setnewIngredient(0);
  setpizzaIngredients([]);
  setName("");
  setOrigin("");
  setDescription("");
  setId(0);
  setState(false);
};
```

Código Fuente 4.18: Método para limpiar el formulario

- Es necesario crear un método para modificar el campo del estado, como se aprecia en el código fuente 4.19.

```
const changeState = () => {
  if (state) {
    setState(false);
  } else {
    setState(true);
  }
};
```

Código Fuente 4.19: Método para modificar el estado

- Para realizar las operaciones de inserción y actualización de pizzas en la base de datos, se utiliza el hook useMutation de Apollo Client. Este hook permite definir la mutación que se ejecutará en el servidor, especificando los datos que se desean enviar y la estructura de la operación.

Una de las características clave de useMutation es la posibilidad de ejecutar acciones adicionales tras completar la mutación. En este contexto, la opción refetchQueries resulta fundamental, ya que permite volver a ejecutar una consulta query específica. Esto es especialmente útil cuando se necesita actualizar automáticamente la lista de pizzas en la interfaz de usuario después de haber creado o modificado una pizza; ver código fuente 4.20.

```
const [updatePizza] = useMutation(UPDATE_PIZZA, {
  refetchQueries: [{ query: GET_PIZZAS }, { query: GET_INGREDIENTS }],
});
const [deletePizzaIngredient] = useMutation(DELETE_PIZZA_INGREDIENT, {
  refetchQueries: [{ query: GET_PIZZAS }],
});

const [createPizza] = useMutation(CREATE_PIZZA, {
  refetchQueries: [{ query: GET_PIZZAS }, { query: GET_INGREDIENTS }],
});
```

Código Fuente 4.20: Mutaciones establecidas

- Una vez que se establecieron las mutaciones, se deben crear los correspondientes métodos para actualizar y crear pizzas, como se observa en el código fuente 4.21.

```
const CreatePizza = (event) => {
  event.preventDefault();
  if (name.length > 0 && origin.length > 0 && pizzaIngredients.length > 0) {
    createPizza({
      variables: {
        pizza: {
          piz_name: name,
          piz_origin: origin,
          piz_description: description,
          piz_state: state,
          ingredientsPizza: ListSelectedIngredients(pizzaIngredients),
        },
      },
    })
  }
  .then(() => {
```

```

        showMessage("Pizza creada correctamente", "success");
    })
    .catch(() => {
        showMessage("Error al crear la pizza", "warning");
    });
    document.getElementById("closeBtn").click();
} else {
    showMessage("No se pueden dejar los campos vacios", "warning");
}
};

const UpdatePizza = (event) => {
    event.preventDefault();
    if (name.length > 0 && origin.length > 0 && pizzaIngredients.length > 0) {
        updatePizza({
            variables: {
                pizza: {
                    piz_id: id,
                    piz_name: name,
                    piz_origin: origin,
                    piz_state: state,
                    piz_description: description,
                    ingredientsPizza: ListSelectedIngredients(pizzaIngredients),
                },
            },
        })
        .then((resp) => {
            console.log(resp);
            showMessage("Pizza actualizada correctamente", "success");
        })
        .catch((err) => {
            showMessage("Error al actualizar la pizza", "warning");
        });
        document.getElementById("closeBtn").click();
        CleanForm();
    } else {
        showMessage("No se pueden dejar los campos vacios", "warning");
    }
};

```

Código Fuente 4.21: Métodos para actualizar y crear pizzas

Es importante mencionar que dentro de los métodos mencionados se utilizó el servicio **message-service** para mostrar mensajes de confirmación y error.

- Con los métodos establecidos, lo siguiente es crear la vista para el formulario que se va a utilizar. Es importante recordar que el código HTML para la vista se debe colocar dentro de la palabra **return**, como se observa en el código fuente 4.22.

```

return (
    <div
        className="modal fade"

```

```

id="PizzaModal"
tabIndex={-1}
aria-labelledby="exampleModalLabel"
aria-hidden="true"
>
<div className="modal-dialog modal-xl">
  <div className="modal-content">
    <div className="modal-header">
      <h1 className="modal-title fs-5" id="exampleModalLabel">
        {" Pizza"}
      </h1>
      <button
        type="button"
        className="btn-close"
        data-bs-dismiss="modal"
        aria-label="Close"
        id="closeBtn"
        onClick={CleanForm}
      />
    </div>
    <div className="modal-body">
      <div className="row">
        <div className="col-sm-5">
          <div className="card">
            <div className="card-body">
              <form>
                <div style={{ textAlign: "left" }}>
                  {id > 0 ? (
                    <div className="mb-3">
                      <strong>Id: {id}</strong>
                    </div>
                  ) : (
                    <></>
                  )}
                <div className="mb-3">
                  <strong>Nombre:</strong>
                  <input
                    type="text"
                    value={name}
                    onChange={({ target }) => setName(target.value)}
                    className="form-control"
                    placeholder="Nombre"
                  />
                </div>
                <strong>Descripcion:</strong>
                <div className="mb-3">
                  <input
                    type="text"
                    className="form-control"
                    placeholder="Descripcion"
                    value={description}

```

```

        onChange={({ target }) =>
          setDescription(target.value)
        }
      />
    </div>
    <strong>Origen:</strong>
    <br></br>
    <div className="mb-3">
      <input
        type="text"
        className="form-control"
        placeholder="Origen"
        value={origin}
        onChange={({ target }) => setOrigin(target.value)}
      />
    </div>
    <div class="container">
      <div class="row">
        <div class="col-3">
          <strong style={{ marginLeft: "-8px" }}>
            Estado:
          </strong>
        </div>
        <div class="col-3" style={{ marginLeft: "-45px" }}>
          Activo
        </div>
        <div class="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"
              id="flexSwitchCheckDefault"
              checked={!state}
              onChange={changeState}
            />
          </div>
        </div>
        <div class="col-3" style={{ marginLeft: "-45px" }}>
          Inactivo
        </div>
        <div class="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"

```

```

        id="flexSwitchCheckDefault"
        checked={state}
        onChange={changeState}
      />
    </div>
  </div>
</div>
</div>
</div>
</form>
</div>
</div>
</div>
<div className="col-sm-7">
  <div className="card">
    <div className="card-body">
      <div style={{ textAlign: "left" }}>
        <div className="row">
          <div className="col-4">
            <strong>Ingrediente:</strong>
            <div className="mb-3">
              <select
                style={{ width: "5cm", height: "1cm" }}
                value={newIngredient}
                onChange={(e) => setnewIngredient(e.target.value)}
              >
                <option>Seleccionar</option>
                {IngredientsList.data?.ingredients.map((item) => {
                  if (item.ing_state) {
                    return (
                      <option
                        key={item.ing_id}
                        value={item.ing_id}
                      >
                        {item.ing_name}
                      </option>
                    );
                  }
                })}
              </select>
            </div>
          </div>
          <div className="col-4">
            <div className="mb-3">
              <strong>Porcion:</strong> <br/><br/>
              <input
                type="number"
                className="form-control"
                placeholder="quantity"
                style={{ width: "5cm", height: "1cm" }}
                value={quantity}

```



```

        min={0}
        onChange={(e) => setQuantity(e.target.value)}
      />
    </div>
  </div>
<div className="col-4">
  <button
    type="button"
    style={{ marginTop: "23px", width: "4.8cm" }}
    className="btn btn-success"
    onClick={AddIngredients}
  >
    <i className="fa fa-plus" aria-hidden="true" />{" "}
    Agregar
  </button>
</div>
</div>

<table className="table table-striped table-bordered">
  <thead>
    <tr>
      <th scope="col">Nombre</th>
      <th scope="col">Calorias</th>
      <th scope="col" style={{ textAlign: "center" }}>
        {" "}
        <i
          className="fa fa-trash"
          aria-hidden="true"
        />{" "}
      </th>
    </tr>
  </thead>
  <tbody>
    {pizzaIngredients.map((item, index) => (
      <tr key={index}>
        <td>{item.ing_name}</td>
        <td>{item.ing_calories}</td>
        <td>
          <div style={{ textAlign: "center" }}>
            <button
              type="button"
              className="btn btn-danger"
              onClick={() => {
                DeleteIngredients(item);
              }}
            >
              <i
                className="fa fa-trash"
                aria-hidden="true"
              />
            </button>
          </div>
        </td>
      </tr>
    ))}
  </tbody>
</table>

```

```

        </div>
        </td>
      </tr>
    )})
  </tbody>
  <tfoot>
    <tr>
      <th></th>
      <th>Total de Calorias</th>
      <th style={{ color: "red", textAlign: "center" }}>
        {getTotalCalories(pizzaIngredients)}
      </th>
    </tr>
  </tfoot>
</table>
</div>
</div>
</div>
</div>
</div>
</div>
</div>
<div className="modal-footer justify-content-md-center ">
  <button
    type="button"
    className="btn btn-danger"
    data-bs-dismiss="modal"
    style={{ width: "20%" }}
    onClick={CleanForm}
  >
    Cancelar <i className="fa fa-close" aria-hidden="true" />
  </button>
  <button
    className="btn btn-success"
    style={{ width: "20%" }}
    onClick={id === 0 ? CreatePizza : UpdatePizza}
  >
    Guardar <i className="fa fa-save" aria-hidden="true" />
  </button>
</div>
</div>
</div>
</div>
</div>
);

```

Código Fuente 4.22: Vista del componente pizza-form

Creación del componente pizza-list

Anteriormente ya se creó el archivo **pizza-list** , en esta sección se explicarán los pasos que deben seguirse para la creación de este componente.

- Se debe importar el componente **Navbar** para facilitar la navegación entre los componentes. Además, se debe importar el componente **PizzaForm** para crear y actualizar las pizzas. Finalmente, se deben importar otros componentes, como se indica en el código fuente 4.23.

```
import React, { useState } from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";
import {
  GET_PIZZAS,
  DELETE_PIZZA,
  showIngredient,
} from "../../services/pizza-service";
import {
  showConfirmMessage,
  showMessage,
} from "../../services/message-service";
import {isAvailable} from "../../services/util-service";
import PizzaForm from "../pizza-form";
```

Código Fuente 4.23: Importaciones del componente pizza-list

- Crear la función para exportar el componente; ver código fuente 4.24.

```
const PizzaList = () => {

}

export default PizzaList;
```

Código Fuente 4.24: Función para exportar el componente

- Declarar las variables que se utilizarán dentro de este componente; ver código fuente 4.25.

```
const PizzaList = () => {
  const [pizza, setPizza] = useState(null);
}

export default PizzaList;
```

Código Fuente 4.25: Declaración de variables

- Para realizar las consultas se utiliza el **hook useQuery** que permite hacer las consultas hacia la API de GraphQL, acompañado del uso del componente **hook useMutation**; ver código fuente 4.26.

```
// Queries
const pizzalist = useQuery(GET_PIZZAS, {
  pollInterval: 500,
});

// Mutations
const [deletePizza] = useMutation(DELETE_PIZZA, {
  refetchQueries: [{ query: GET_PIZZAS }],
});
```

Código Fuente 4.26: Consultas y Mutaciones establecidas

- Una vez que se establecieron las mutaciones, se deben crear los correspondientes métodos para seleccionar pizzas y eliminar pizzas; como se observa en el código fuente 4.27.

```
const selectPizza = (pizza) => {
  setPizza(pizza);
};
const DeletePizza = (item) => {
  let id = item.piz_id;
  showConfirmMessage("Esta seguro de eliminar esta Pizza?").then((resp) => {
    if (resp.isConfirmed) {
      deletePizza({
        variables: {
          deletePizzaId: id.toString(),
        },
      })
      .then(() => {
        showMessage("Pizza eliminada correctamente", "success");
      })
      .catch((err) => {
        showMessage("Error al eliminar la pizza", "warning");
      });
    }
  });
};
```

Código Fuente 4.27: Métodos para seleccionar y eliminar pizzas

Es importante mencionar que dentro de los métodos mencionados se utilizó el servicio **message-service** para mostrar mensajes de confirmación y error.

- Con los métodos establecidos, el siguiente paso es crear la vista para el listado de las pizzas que se va a utilizar. Se debe recordar que el código HTML para la vista se debe colocar dentro de la palabra **return**, como se observa en el código fuente 4.28.

```
return (
  <div>
    <>
      <NavBar></NavBar>
      <br />
      <div className="container">
        <h5 className="card-title title">Pizzas</h5>
        <div className="d-grid gap-2 d-md-flex justify-content-md-end">
          <button
            className="btn btn-primary btn-lg"
            type="button"
            data-bs-toggle="modal"
            data-bs-target="#PizzaModal"
          >
            <i className="fa fa-plus" aria-hidden="true" /> Crear
          </button>
        </div>
        <PizzaForm pizza={pizza}></PizzaForm>
      <br />
      <table className="table table-striped table-bordered">
```

```

<thead>
  <tr>
    <th scope="col">Nombre</th>
    <th scope="col">Origen</th>
    <th scope="col">Estado</th>
    <th scope="col">Ingredientes</th>
    <th scope="col" style={{ textAlign: "center" }}>
      Acciones
    </th>
  </tr>
</thead>
<tbody>
  {pizzalist.data?.pizzas.map((item) => (
    <tr key={item.piz_id}>
      <td>{item.piz_name}</td>
      <td>{item.piz_origin}</td>
      <td>{isAvailable(item.piz_state)}</td>
      <td>{showIngredient(item.ingredients)}</td>
      <td style={{ textAlign: "center" }}>
        <button
          type="button"
          className="btn btn-warning"
          data-bs-toggle="modal"
          data-bs-target="#PizzaModal"
          style={{ marginRight: 10 }}
          onClick={() => {
            selectPizza(item);
          }}
        >
          <i className="fa fa-pencil" aria-hidden="true" />
        </button>

        <button
          type="button"
          className="btn btn-danger"
          onClick={() => {
            DeletePizza(item);
          }}
        >
          <i className="fa fa-trash" aria-hidden="true" />
        </button>
        <PizzaForm pizza={pizza}></PizzaForm>
      </td>
    </tr>
  )})
</tbody>
</table>
</div>
</>
</div>
);

```

Código Fuente 4.28: Vista del componente pizza-list

Para verificar el funcionamiento del componente **pizza-list** y ejecutar la aplicación se utiliza el comando **npm start**. A continuación, en la Fig. 4.11, se puede apreciar la vista principal de la aplicación, en donde la barra superior permite navegar hacia los componentes que se crearon recientemente.

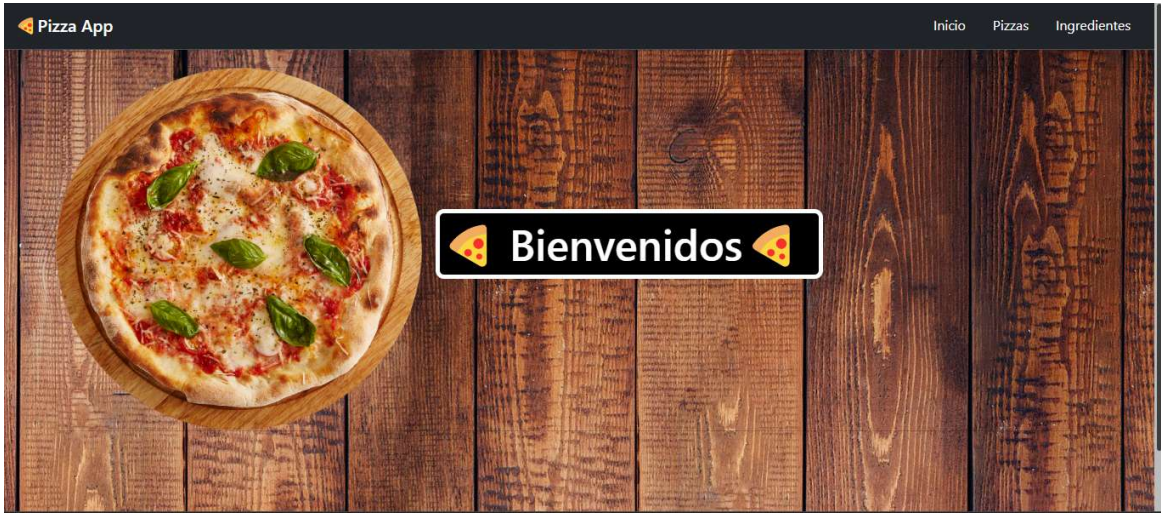


Fig. 4.11. Vista principal

La Fig. 4.12 muestra el funcionamiento de la lista de pizzas creadas hasta el momento.

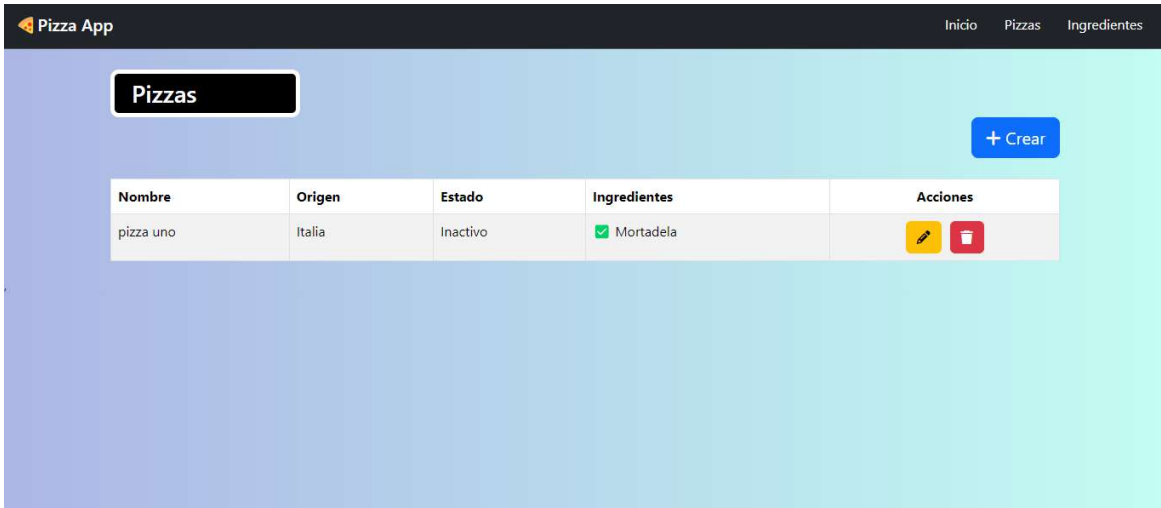


Fig. 4.12. Listado Pizza

La Fig. 4.13 muestra los campos del formulario para crear una nueva pizza.

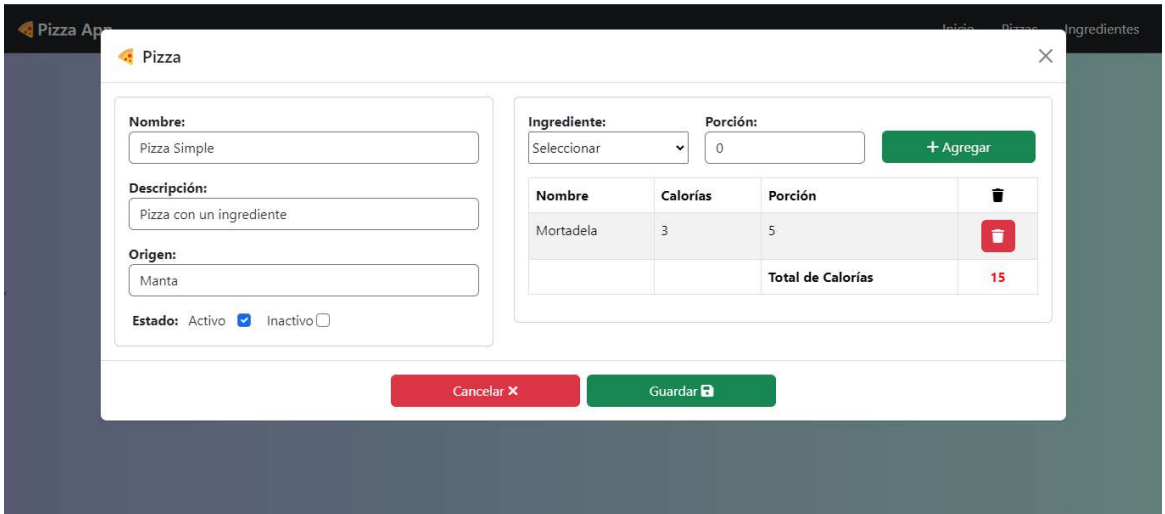


Fig. 4.13. Crear Pizzas

A continuación, la Fig. 4.14 muestra que la pizza ha sido creada correctamente.

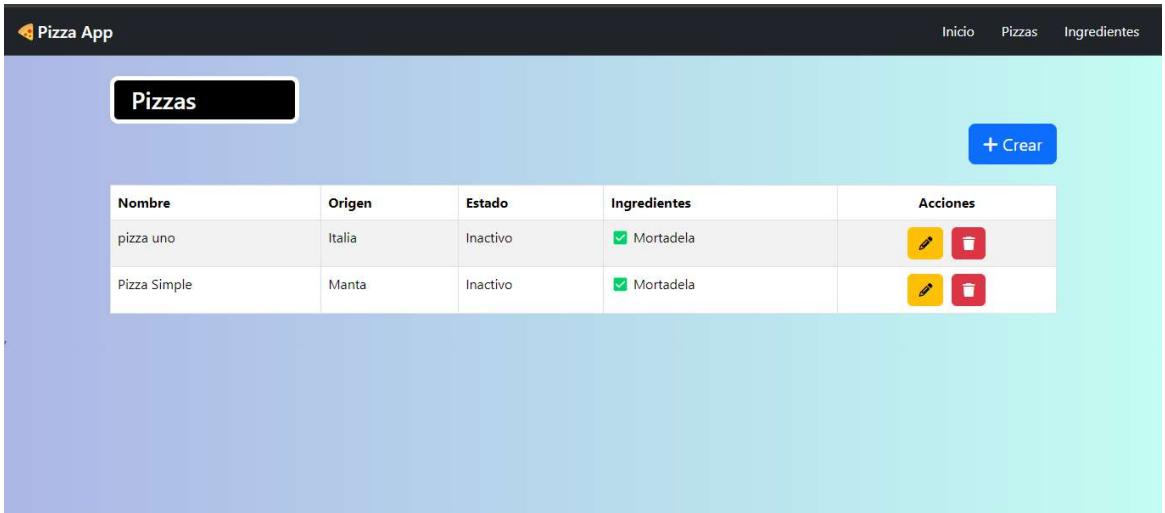


Fig. 4.14. Pizza creada

En la Fig. 4.15 se muestran los campos de una pizza que se pueden actualizar.

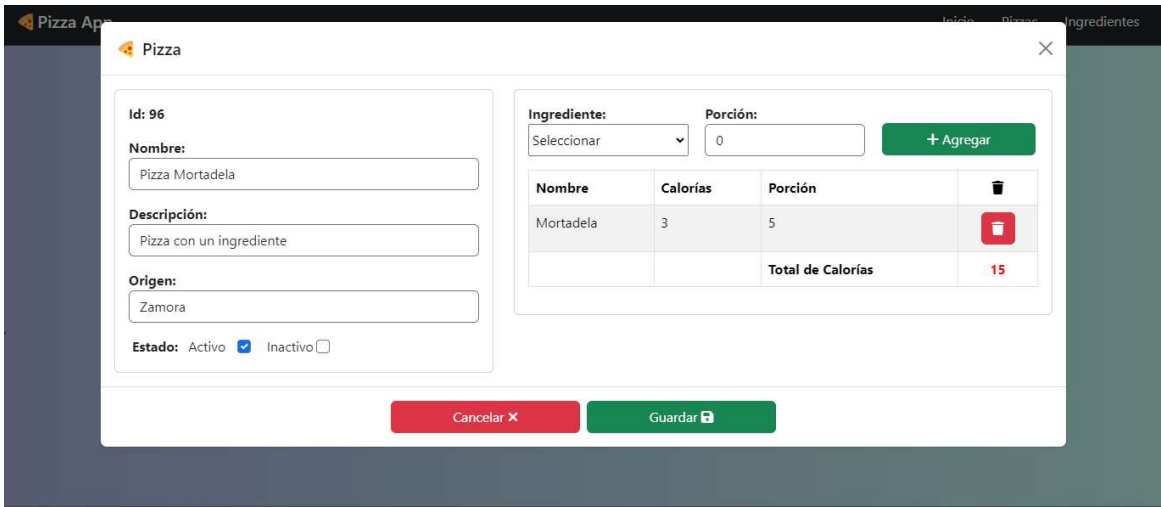


Fig. 4.15. Actualizar Pizzas

La Fig. 4.16 muestra que los campos de la pizza han sido actualizados correctamente.

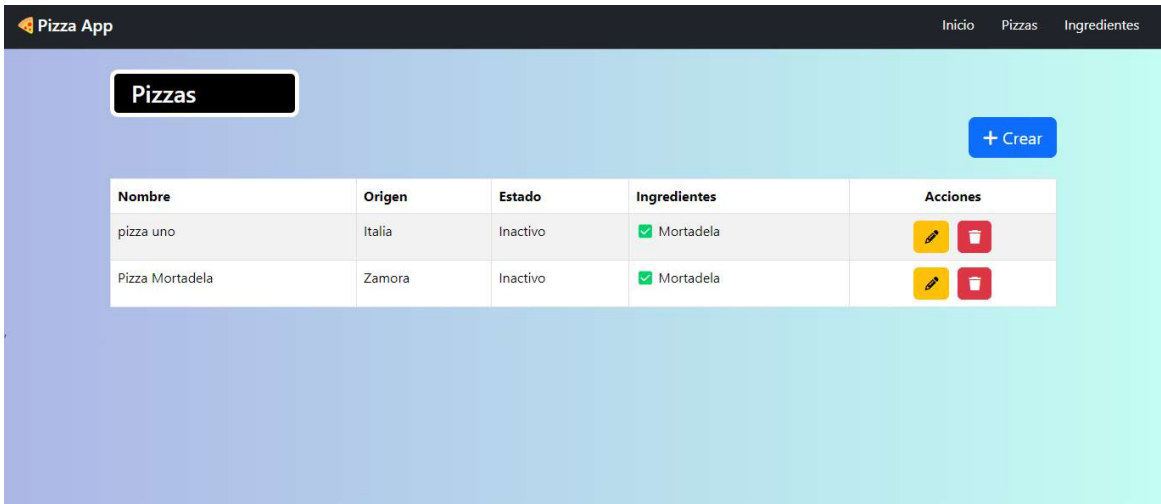


Fig. 4.16. Pizza actualizada

Se puede eliminar una pizza para verificar su funcionamiento, como muestra la Fig. 4.17.

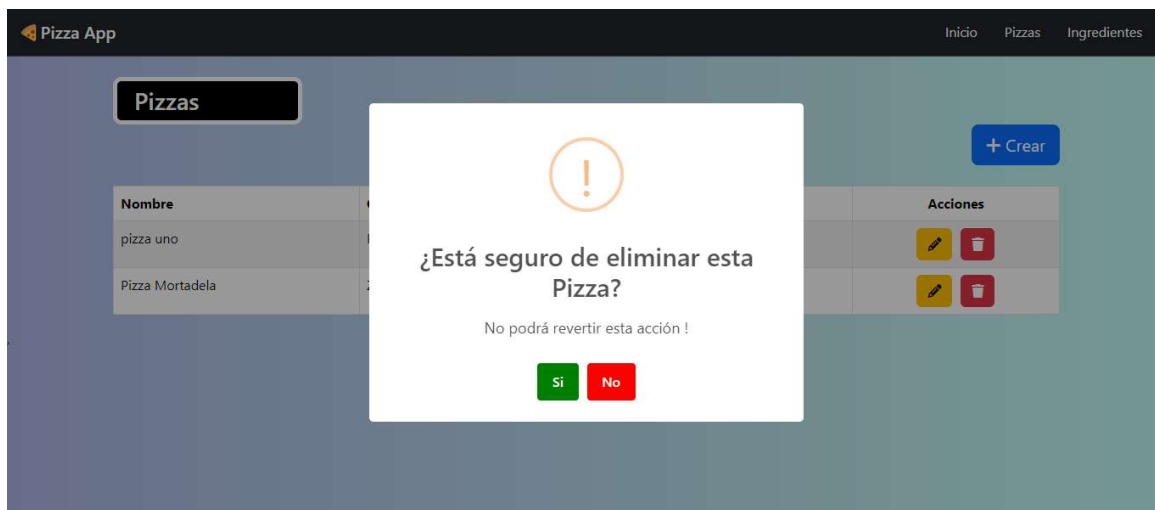


Fig. 4.17. Eliminar Pizzas

En la Fig. 4.18 se observa que la pizza ha sido eliminada correctamente.

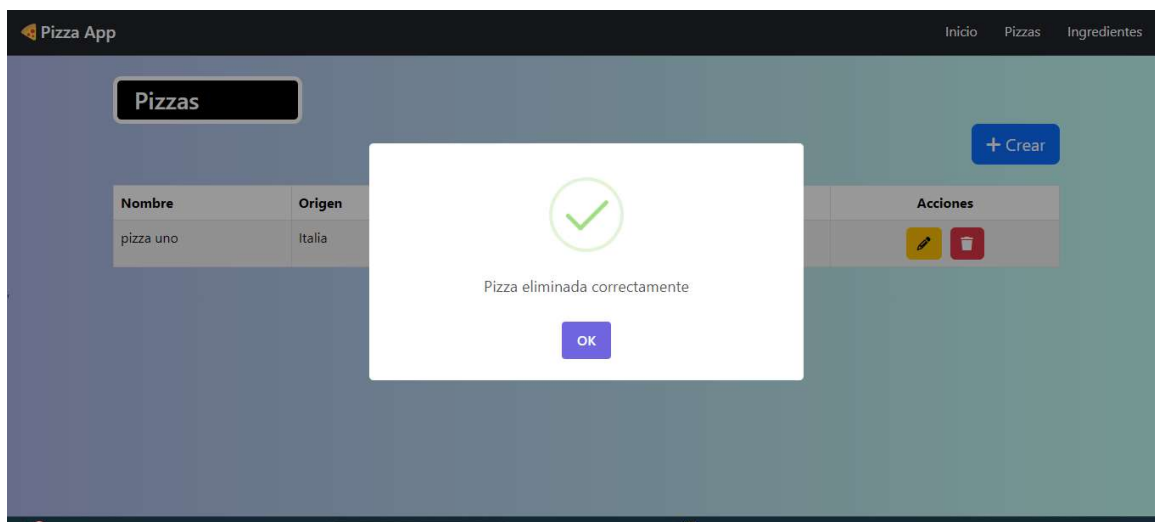


Fig. 4.18. Pizza eliminada

Creación del componente ingredient-form

Este componente debe crearse en la carpeta **ingredient**, la cual está ubicada en la carpeta **components**. Mediante el siguiente componente se van a realizar operaciones de inserción, actualización y eliminación de pizzas.

A continuación, se indican las instrucciones que se deben seguir para la creación de este componente.

- Realizar las siguientes importaciones; ver código fuente 4.29.

```
import React, { useEffect } from "react";
import { useState } from "react";
import { useMutation } from "@apollo/client";
import {
  GET_INGREDIENTS,
  CREATE_INGREDIENTS,
  UPDATE_INGREDIENTS,
} from "../../services/ingredient-service";

import { showMessage } from "../../services/message-service";
```

Código Fuente 4.29: Importaciones del componente ingredient-form

- Crear la función para exportar el componente e incluir un parámetro que permita obtener el ingrediente seleccionado al actualizar sus datos; ver código fuente 4.30.

```
const IngredientForm = ({ ingredient }) => {

}

export default IngredientForm;
```

Código Fuente 4.30: Función para exportar el componente

- Declarar las variables para cada uno de los campos del formulario de ingredientes. En el caso de **React** es necesario utilizar el **hook useState** para modificar y actualizar el valor de las variables que se crearon; ver código fuente 4.31.

```
const IngredientForm = ({ ingredient }) => {
  const [name, setName] = useState("");
  const [calories, setCalories] = useState(0);
  const [state, setState] = useState(false);
  const [id, setId] = useState(0);
}
export default IngredientForm;
```

Código Fuente 4.31: Declaración de variables

- Posteriormente, se debe utilizar el **hook useEffect** para actualizar el valor del parámetro ingrediente cada vez que se actualice; ver código fuente 4.32.

```
useEffect(() => {
  if (ingredient !== null) {
    setName(ingredient.ing_name);
    setCalories(ingredient.ing_calories);
    setId(ingredient.ing_id);
    setState(ingredient.ing_state);
  }
}, [ingredient]);
```

Código Fuente 4.32: Uso del hook useEffect

- Es importante crear un método para limpiar el formulario una vez que se hayan registrado nuevos ingredientes. Este método se muestra en el código fuente 4.33.

```
const CleanForm = () => {
  setName("");
  setCalories(0);
  setId(0);
  setState(false);
};
```

Código Fuente 4.33: Método para limpiar el formulario

- Para realizar las operaciones de insertar y actualizar pizzas, se debe utilizar el **hook useMutation** que permite especificar el tipo de mutación que se desee realizar. Es importante mencionar que la opción **refetchQueries** permite que se pueda llamar a una consulta o query específica con el propósito de actualizar el listado de ingredientes una vez que se ha creado o actualizado un ingrediente; ver código fuente 4.34.

```
// Mutations
const [createIngredient] = useMutation(CREATE_INGREDIENTS, {
  refetchQueries: [{ query: GET_INGREDIENTS }],
});

const [updateIngredient] = useMutation(UPDATE_INGREDIENTS, {
  refetchQueries: [{ query: GET_INGREDIENTS }],
});
```

Código Fuente 4.34: Mutaciones establecidas

- Una vez que se establecieron las mutaciones, se deben crear los correspondientes métodos para actualizar y crear ingredientes. Es necesario crear un método para cambiar el valor de la variable estado; ver código fuente 4.35.

```
const CreateIngredient = (event) => {
  event.preventDefault();
  if (name.length > 0 && calories > 0) {
    createIngredient({
      variables: {
        ingredient: {
          ing_calories: parseFloat(calories),
          ing_name: name,
          ing_state: state,
        },
      },
    })
    .then(() => {
      showMessage("ingrediente creado correctamente", "success");
    })
    .catch(() => {
      showMessage("Error al crear el ingrediente", "warning");
    });
    document.getElementById("closeBtn").click();
  }
};
```

```

    } else {
      showMessage("No se pueden dejar los campos vacios", "warning");
    }
  };

  const UpdateIngredient = (event) => {
    event.preventDefault();
    if (name.length > 0 && calories > 0) {
      updateIngredient({
        variables: {
          ingredient: {
            ing_id: id,
            ing_name: name,
            ing_state: state,
            ing_calories: parseFloat(calories),
          },
        },
      })
        .then(() => {
          showMessage("Ingrediente actualizado correctamente", "success");
        })
        .catch((err) => {
          console.log(err);
          showMessage("Error al actualizar el ingrediente", "warning");
        });
      document.getElementById("closeBtn").click();
    } else {
      showMessage("No se pueden dejar los campos vacios", "warning");
    }
  };

  const changeState = () => {
    if (state) {
      setState(false);
    } else {
      setState(true);
    }
  };
};

```

Código Fuente 4.35: Métodos para cambiar el estado, actualizar y crear ingredientes

Es importante mencionar que dentro de los métodos mencionados se utilizó el servicio **message-service** para mostrar mensajes de confirmación y error.

- Lo siguiente es crear la vista para el formulario que se va a utilizar. Se recuerda que el código HTML para la vista se debe colocar dentro de la palabra **return** como se observa en el código fuente 4.36.

```

return (
  <div
    className="modal fade"
    id="IngredienteModal"
    tabIndex={-1}

```



```

aria-labelledby="exampleModalLabel"
aria-hidden="true"
>
<div className="modal-dialog">
  <div className="modal-content">
    <div className="modal-header">
      <h1 className="modal-title fs-5" id="IngredienteModal">
        Ingredientes
      </h1>
      <button
        type="button"
        className="btn-close"
        data-bs-dismiss="modal"
        aria-label="Close"
        id="closeBtn"
        onClick={CleanForm}
      />
    </div>
    <div className="modal-body">
      <form>
        <div style={{ textAlign: "left" }}>
          {id > 0 ? (
            <div className="mb-3">
              <strong>Id: {id}</strong>
            </div>
          ) : (
            <></>
          )}
          <strong>Nombre</strong> <br></br>
          <div className="mb-3">
            <input
              type="text"
              className="form-control"
              placeholder="Nombre"
              value={name}
              onChange={({ target }) => setName(target.value)}
            />
          </div>
          <strong>Calorias</strong> <br></br>
          <div className="mb-3">
            <input
              type="number"
              className="form-control"
              placeholder="calories"
              style={{ height: "1cm" }}
              value={calories}
              onChange={({ target }) => setCalories(target.value)}
            />
          </div>
          <div class="container">
            <div class="row">

```

```

        <div class="col-3">
          <strong style={{ marginLeft: "-8px" }}>Estado:</strong>
        </div>
        <div class="col-3" style={{ marginLeft: "-45px" }}>
          Activo
        </div>
        <div class="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"
              id="flexSwitchCheckDefault"
              checked={state}
              onChange={changeState}
            />
          </div>
        </div>
        <div class="col-3" style={{ marginLeft: "-45px" }}>
          Inactivo
        </div>
        <div class="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"
              id="flexSwitchCheckDefault"
              checked={!state}
              onChange={changeState}
            />
          </div>
        </div>
      </div>
    </div>
  </form>
</div>
<div className="modal-footer">
  <button
    type="button"
    className="btn btn-danger"
    data-bs-dismiss="modal"
    onClick={CleanForm}
  >
    Cancelar <i className="fa fa-close" aria-hidden="true" />
  </button>

```

```

        <button
          type="button"
          className="btn btn-success"
          onClick={id === 0 ? CreateIngredient : UpdateIngredient}
        >
          Guardar <i className="fa fa-save" aria-hidden="true" />
        </button>
      </div>
    </div>
  </div>
</div>
);

```

Código Fuente 4.36: Vista del componente ingredient-form

Creación del componente ingredient-list

Anteriormente, se había creado el archivo **ingredient-list**. Sin embargo, en esta sección se detallarán los pasos que deben seguirse para la creación de este componente.

- Realizar las siguientes importaciones. Es necesario importar el componente **Navbar** para facilitar la navegación entre los componentes e **IngredientForm** para crear y actualizar los ingredientes; ver código fuente 4.37.

```

import React, { useState } from "react";
import NavBar from "../shared/navbar";
import { useQuery, useMutation } from "@apollo/client";
import {
  GET_INGREDIENTS,
  DELETE_INGREDIENTS,
} from "../services/ingredient-service";

import {
  showMessage,
  showConfirmMessage,
} from "../services/message-service";
import IngredientForm from "../ingredient-form";
import { isAvailable } from "../services/util-service";

```

Código Fuente 4.37: Importaciones del componente ingredient-list

- Crear la función para exportar el componente; ver código fuente 4.38.

```

const IngredientList = () => {

}

export default IngredientList;

```

Código Fuente 4.38: Función para exportar el componente

- Declarar las variables que se utilizarán dentro de este componente; ver código fuente 4.39.

```
const IngredientList = () => {  
  const [ingredient, setIngredient] = useState(null);  
}  
  
export default IngredientList;
```

Código Fuente 4.39: Declaración de variables

- Para realizar las consultas se va a utilizar el **hook useQuery** el cual permite hacer las consultas hacia el API de GraphQL. También se utilizará el componente **hook useMutation**; ver código fuente 4.40.

```
// Queries  
const IngredientsList = useQuery(GET_INGREDIENTS);  
// Mutations  
const [deleteIngredient] = useMutation(DELETE_INGREDIENTS, {  
  refetchQueries: [{ query: GET_INGREDIENTS }],  
});
```

Código Fuente 4.40: Consultas y Mutaciones establecidas

- Una vez que se establecieron las mutaciones, se deben crear los correspondientes métodos para seleccionar ingredientes y eliminar ingredientes, como se observa en el código fuente 4.41.

```
const selectIngredient = (ingredient) => {  
  setIngredient(ingredient);  
};  
  
const DeleteIngredient = (item) => {  
  let id = item.ing_id;  
  showConfirmMessage("Esta seguro de eliminar esta Ingrediente?").then(  
    (resp) => {  
      if (resp.isConfirmed) {  
        deleteIngredient({  
          variables: {  
            deleteIngredientId: id.toString(),  
          },  
        },  
      )  
        .then(() => {  
          showMessage("Ingrediente eliminado correctamente", "success");  
        })  
        .catch((err) => {  
          showMessage("Error al eliminar el ingrediente", "warning");  
        });  
      }  
    })  
  );  
};
```

Código Fuente 4.41: Métodos para seleccionar y eliminar ingredientes

Es importante mencionar que dentro de los métodos mencionados se utilizó el servicio **message-service** para mostrar mensajes de confirmación y error.

- Con los métodos establecidos, lo siguiente es crear la vista para el listado de los ingredientes que se va a utilizar. Es importante recordar que el código HTML para la vista se debe colocar dentro de la palabra **return** como se observa en el código fuente 4.42.

```

return (
<div>
  <>
    <NavBar></NavBar>
    <br />
    <div className="container">
      <h5 className="card-title title">Ingredientes</h5>
      <div className="d-grid gap-2 d-md-flex justify-content-md-end">
        <button
          className="btn btn-primary btn-lg"
          type="button"
          data-bs-toggle="modal"
          data-bs-target="#IngredienteModal"
        >
          <i className="fa fa-plus" aria-hidden="true" /> Crear
        </button>
        <IngredientForm ingredient={ingredient} ></IngredientForm>
      </div>
    <br />
    <table className="table table-striped table-bordered">
      <thead>
        <tr>
          <th scope="col">Nombre</th>
          <th scope="col">Estado</th>
          <th scope="col">Calorias</th>
          <th scope="col" style={{ textAlign: "center" }}>
            Acciones
          </th>
        </tr>
      </thead>
      <tbody>
        {IngredientsList.data?.ingredients.map((item) => (
          <tr key={item.ing_id}>
            <td>{item.ing_name}</td>
            <td>{isAvailable(item.ing_state)}</td>
            <td>{item.ing_calories}</td>
            <td style={{ textAlign: "center" }}>
              <button
                type="button"
                className="btn btn-warning"
                data-bs-toggle="modal"
                data-bs-target="#IngredienteModal"
                style={{ marginRight: 10 }}
                onClick={() => {
                  selectIngredient(item);
                }}
              >

```

```

        <i className="fa fa-pencil" aria-hidden="true" />
      </button>
      <IngredientForm
        key={item.ing_id}
        ingredient={ingredient}
      ></IngredientForm>
      <button
        type="button"
        className="btn btn-danger"
        onClick={() => {
          DeleteIngredient(item);
        }}
      >
        <i className="fa fa-trash" aria-hidden="true" />
      </button>
    </td>
  </tr>
</tbody>
</table>
</div>
</>
</div>
);

```

Código Fuente 4.42: Vista del componente ingredient-list

Para verificar el funcionamiento del componente se van a realizar algunas pruebas. Para ejecutar la aplicación se utiliza el comando **npm start**.

La Fig. 4.19 muestra la vista principal de la aplicación, en donde la barra superior permite navegar hacia los componentes que se crearon recientemente.



Fig. 4.19. Vista principal

Es necesario dirigirse hacia el componente Ingrediente para verificar su funcionamiento; ver Fig. 4.20.

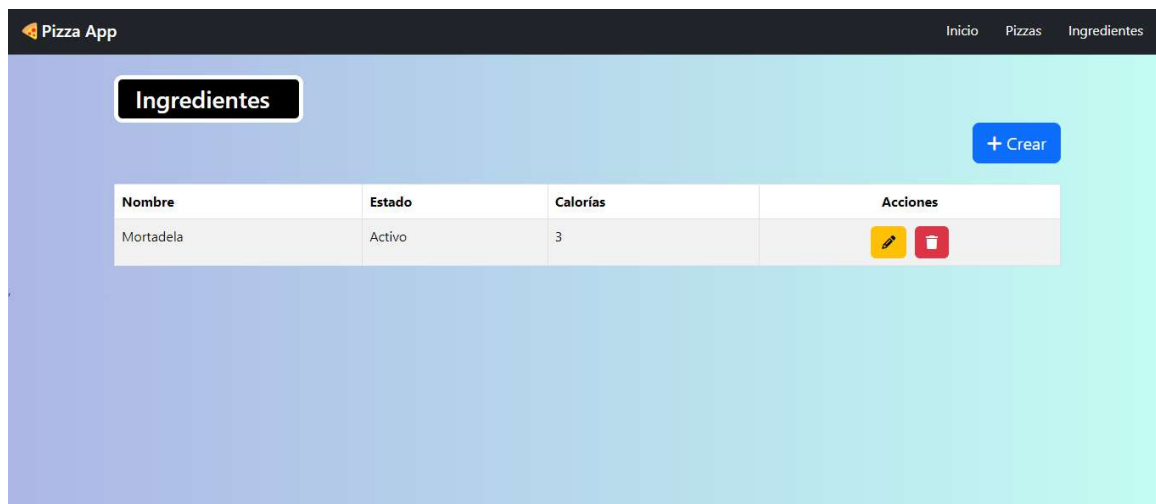


Fig. 4.20. Listado de ingredientes

En la Fig. 4.21 se muestra la creación de un ingrediente para verificar su funcionamiento.

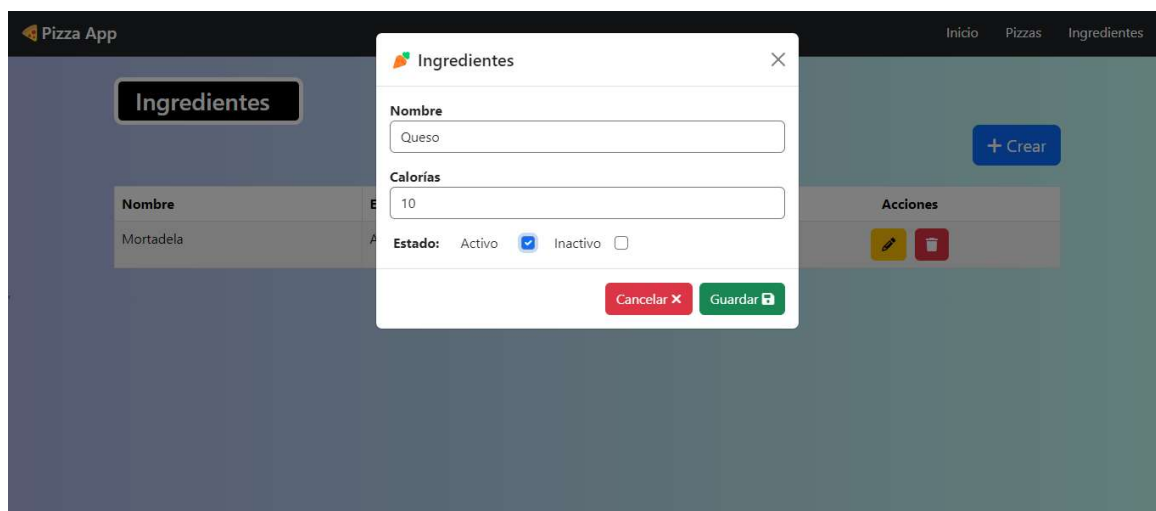


Fig. 4.21. Crear Ingredientes

A continuación, en la Fig. 4.22 se observa que el ingrediente ha sido creado correctamente.

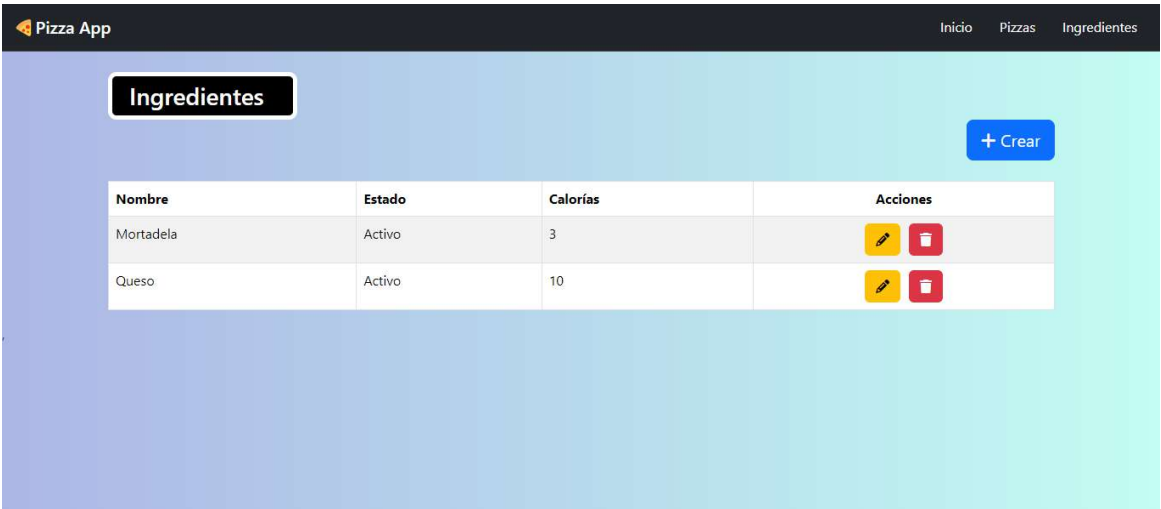


Fig. 4.22. Ingrediente creado

En la Fig. 4.23 se observa el formulario para actualizar un ingrediente y verificar su funcionamiento.

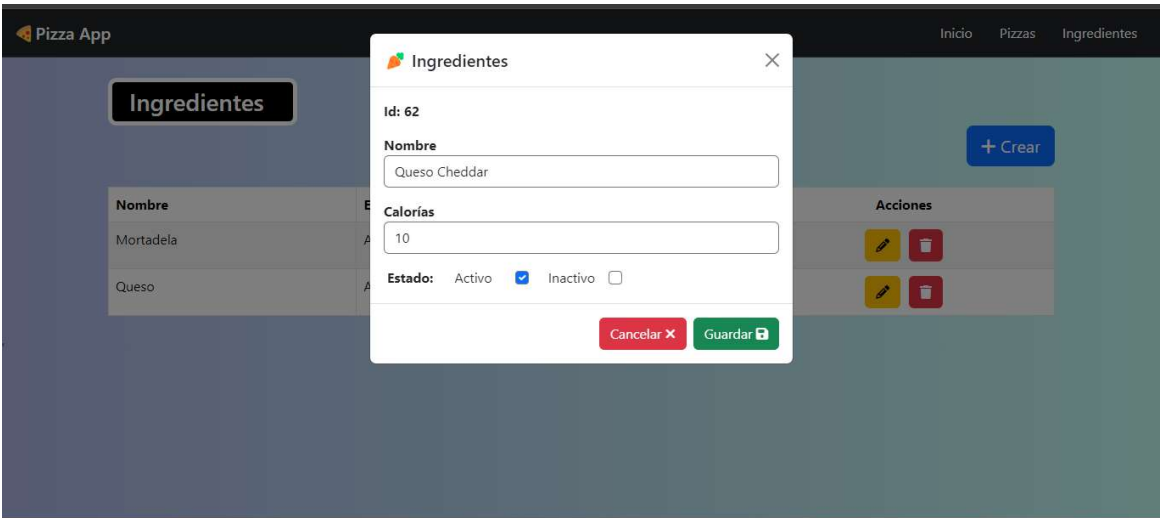


Fig. 4.23. Actualizar ingredientes

En la Fig. 4.24 se observa que el ingrediente ha sido actualizado correctamente.

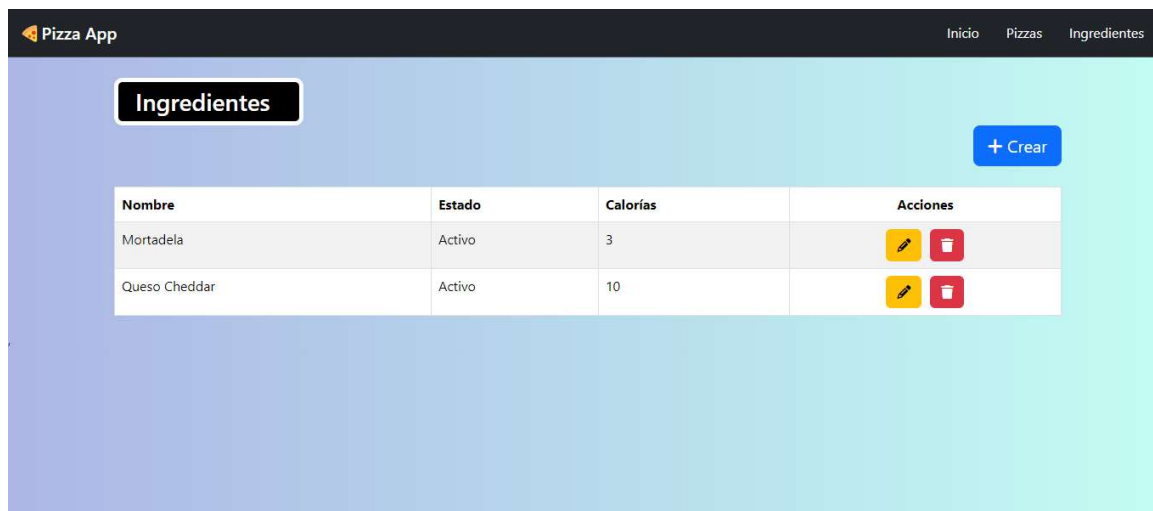


Fig. 4.24. Ingrediente actualizada

En la Fig. 4.25 se observa la eliminación de un ingrediente para verificar su funcionamiento.

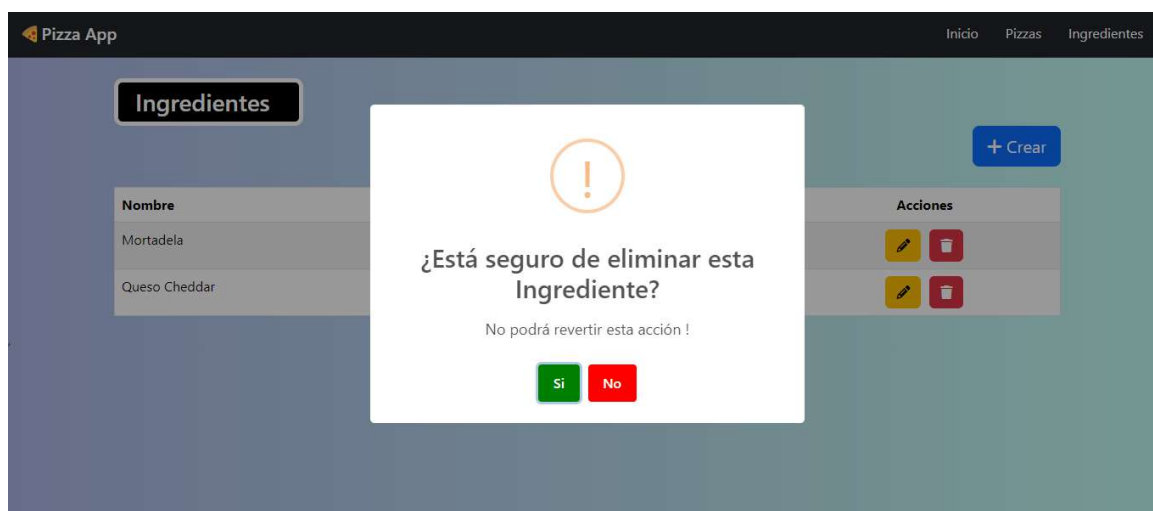


Fig. 4.25. Eliminar Ingredientes

En la Fig. 4.26 se muestra que el ingrediente ha sido eliminado correctamente.

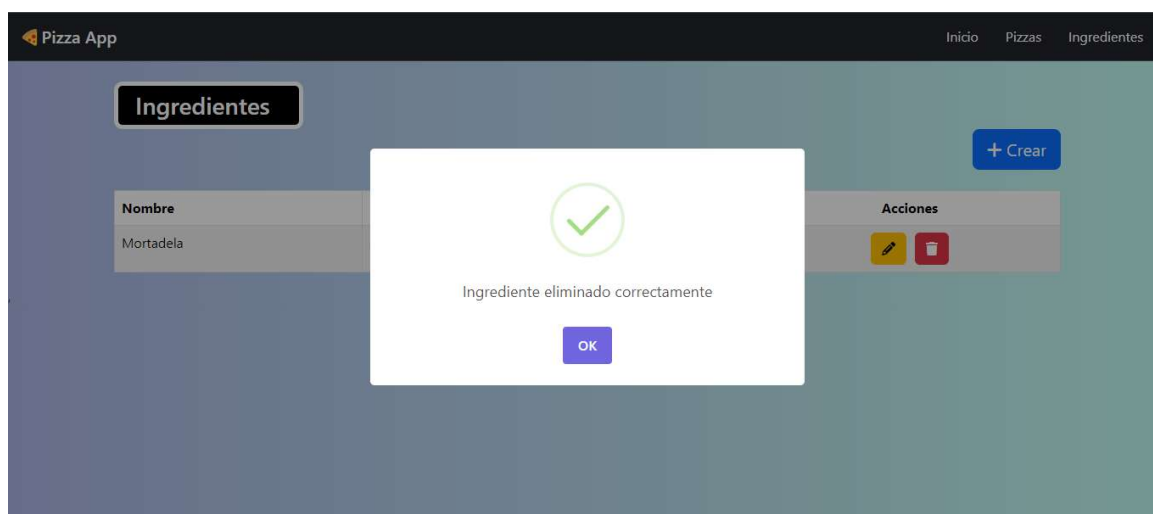


Fig. 4.26. Ingrediente eliminado

A continuación se adjunta el enlace al repositorio de Github del proyecto realizado:

Enlace del repositorio de Github

<https://github.com/antonio-quina/frontend-capitulo-04.git>

RESUMEN

El desarrollo del frontend confirma la relevancia de GraphQL en la construcción de interfaces dinámicas, interactivas y altamente personalizables. Este capítulo resalta cómo la elección de tecnologías modernas, como React y Apollo Client, permite consumir APIs de manera eficiente, optimizando la comunicación con el backend y reduciendo la complejidad en la gestión de datos. La adopción de GraphQL en el frontend proporciona un control preciso sobre la información solicitada, evitando la sobrecarga de datos y mejorando el rendimiento general de la aplicación.

A través de ejemplos prácticos, se ilustra cómo integrar operaciones de consulta y mutación directamente en los componentes de React, mostrando la versatilidad y la eficiencia de GraphQL en escenarios cotidianos de desarrollo. Este enfoque permite que los componentes sean responsables de sus propias solicitudes de datos, promoviendo una arquitectura modular, reutilizable y más fácil de mantener a medida que la aplicación crece. Asimismo, se evidencia cómo la implementación de suscripciones puede mejorar la interactividad en tiempo real, enriqueciendo la experiencia del usuario.

Este capítulo destaca la importancia de gestionar adecuadamente los estados de carga, error y éxito en cada operación, lo que garantiza una experiencia de usuario fluida y confiable. La correcta visualización y manejo de estos estados permite anticipar problemas y ofrecer retroalimentación inmediata, aspectos críticos en aplicaciones modernas donde la percepción de velocidad y estabilidad es fundamental. Estas prácticas no solo mejoran la usabilidad, sino que también facilitan el mantenimiento y la escalabilidad de la interfaz.

Finalmente, la integración efectiva entre frontend y backend potencia el modelo FullStack, consolidando aplicaciones que combinan funcionalidad, rendimiento e intuición. La conexión estrecha entre las capas permite una comunicación coherente y eficiente, aprovechando al máximo las capacidades de GraphQL. De este modo, se logra un entorno de desarrollo que no solo responde a las necesidades actuales de los usuarios, sino que también se adapta a la evolución tecnológica, ofreciendo aplicaciones escalables, intuitivas y listas para entornos de alta complejidad.

ACTIVIDADES

Resuelva el siguiente ejercicio

Desarrolle un nuevo componente que permita filtrar y buscar pizzas e ingredientes por su nombre. Para ello, deberá implementar tanto el frontend como el backend siguiendo estos pasos:

1. Cree nuevas consultas en el API que reciban como parámetro el nombre o parte del nombre de la pizza o del ingrediente.
2. Cree un componente con un campo de búsqueda donde el usuario pueda ingresar el nombre parcial o completo de la pizza o del ingrediente.
3. Muestre los resultados filtrados en una lista o tabla, incluyendo información relevante como nombre, descripción, origen (para pizzas) y nombre, calorías (para ingredientes).
4. Asegúrese de que la búsqueda sea sensible a coincidencias parciales, de modo que, por ejemplo, buscar “moz” retorne “Mozarela”.
5. La consulta debe devolver todos los registros que coincidan con el criterio de búsqueda, incluyendo los campos necesarios para mostrar la información en el componente.



CAPÍTULO 5

GraphQL Hands-On: Retos Prácticos para Desarrolladores

«El desarrollo de software es una combinación de lógica y creatividad, donde cada línea de código fuente puede cambiar el mundo.»

Knuth, D. E., 1997, p. 215.

En este capítulo, se presenta una serie de ejercicios prácticos diseñados para reforzar y aplicar los conceptos fundamentales de GraphQL en el desarrollo de aplicaciones empresariales seguras y de alta calidad. A lo largo de estos ejercicios, los desarrolladores tendrán la oportunidad de implementar componentes esenciales que se utilizan comúnmente en entornos de producción, complementando así la teoría y las aplicaciones prácticas abordadas en los capítulos anteriores. El objetivo principal es proporcionar una experiencia aplicada que permita a los desarrolladores consolidar sus conocimientos y mejorar sus habilidades en el uso de GraphQL. Para lograrlo, se trabajará con casos de uso reales y escenarios típicos en el desarrollo empresarial, con un enfoque especial en seguridad, rendimiento y gestión eficiente de datos. Durante el desarrollo de los ejercicios, se trabajará en la implementación de funcionalidades esenciales, tales como mecanismos de autenticación robusta, manipulación avanzada de datos mediante ORMs, generación de reportes y la integración de GraphQL con APIs REST. Además, se explorarán técnicas avanzadas que abarcan desde la configuración inicial hasta la optimización y despliegue de aplicaciones en la nube.

A continuación, se detallan los principales componentes que se abordarán en este capítulo:

- **ORM (Mapeo Objeto-Relacional):** Configuración y uso de un ORM (Object-Relational Mapping) para interactuar con bases de datos PostgreSQL, facilitando la gestión eficiente de datos de una API GraphQL.
- **Seguridad en GraphQL:** Implementación de componentes de seguridad esenciales como autenticación, autorización y validación de datos, para proteger a la API de vulnerabilidades y accesos no autorizados.
- **Menús Dinámicos:** Creación y gestión de menús dinámicos en la interfaz de usuario, utilizando consultas y mutaciones de GraphQL para personalizar la experiencia de usuario según sus necesidades.
- **Manejo Avanzado de Tablas:** Implementación de funciones avanzadas para manipular y visualizar datos en tablas, incluyendo búsqueda, paginación, control del número de registros mostrados y ordenamiento dinámico, todo a través de GraphQL.

- **Generación de informes:** Creación de informes optimizados usando GraphQL, diseñados para extraer y presentar datos eficientemente, asegurando un rendimiento óptimo en aplicaciones que manejan grandes volúmenes de información.
- **Interacción con APIs REST:** GraphQL se utiliza como una capa de integración para consumir y manipular datos de APIs REST existentes, mejorando la interoperabilidad entre diversas fuentes de datos.
- **GraphQL en Entornos Cloud:** Implementación y despliegue de aplicaciones GraphQL en plataformas cloud, abordando estrategias para optimizar la escalabilidad, mejorar la seguridad y facilitar el mantenimiento en entornos de producción.
- **Generación de Código Fuente:** Automatización del desarrollo utilizando herramientas de generación de código fuente que sean compatibles con GraphQL, mejorando así la estructura y la uniformidad del proyecto.

Con estos ejercicios, los desarrolladores no solo fortalecerán su dominio de GraphQL, sino que también adquirirán habilidades clave para enfrentar desafíos reales en el desarrollo de software empresarial. ¡Es el momento de poner manos a la obra y llevar el aprendizaje a la práctica!

ORM (MAPEO OBJETO-RELACIONAL)

El Mapeo Objeto-Relacional (ORM) es una técnica utilizada en el desarrollo de software para mapear estructuras de datos definidas en código orientado a objetos a tablas en bases de datos relacionales. Su objetivo es facilitar la persistencia de datos al abstraer la complejidad de las operaciones SQL mediante el uso de modelos de objetos. Aunque es un enfoque ampliamente adoptado, puede presentar limitaciones en sistemas donde la trazabilidad y la transparencia son cruciales, especialmente en entornos modernos como la computación en la nube [34].

Las tecnologías ORM resuelven el desajuste entre los modelos de datos relacionales y orientados a objetos, permitiendo a los desarrolladores interactuar con la base de datos a través de objetos en lugar de consultas SQL tradicionales. Esto simplifica el proceso de desarrollo y mejora la mantenibilidad y escalabilidad del software [35]. Además, una característica clave del uso de ORM es que proporciona una abstracción sobre las bases de datos relacionales, permitiendo a los desarrolladores interactuar con las bases de datos mediante una interfaz orientada a objetos en lugar de utilizar SQL directamente. Sin embargo, el estudio muestra que esta abstracción puede introducir errores debido a inconsistencias en la traducción de consultas a SQL, errores en la gestión de datos y diferencias en la compatibilidad con distintos sistemas de gestión de bases de datos (DBMS) [36].

Siguiendo el contexto del desarrollo de APIs abordado en el Capítulo 3, se identificó que los ORM más populares para Node.js son TypeORM¹, Prisma² y Sequelize³. Este último destaca por su madurez, estabilidad y flexibilidad, lo que lo convierte en una opción preferida para muchos desarrolladores [37]. A continuación, se presenta un análisis detallado de Sequelize.

¹TypeORM: <https://typeorm.io>

²Prisma: <https://www.prisma.io>

³Sequelize: <https://sequelize.org>

ORM: Sequelize

Sequelize es un ORM para Node.js que simplifica la interacción con bases de datos relacionales como MySQL⁴, PostgreSQL⁵, SQLite⁶ y Microsoft SQL Server⁷. Permite a los desarrolladores manejar registros de bases de datos como objetos JavaScript, evitando la necesidad de escribir consultas SQL complejas para funciones comunes [37].

Características de Sequelize:

- **Mapeo de Objetos:** Permite mapear las tablas de la base de datos a objetos en JavaScript, donde cada fila se representa como una instancia de dichos objetos. A través de la definición de modelos, es posible gestionar y manipular los datos directamente en el código fuente [37].
- **Construcción de Consultas:** Ofrece un conjunto de métodos que facilitan la creación de consultas SQL, permitiendo su encadenamiento para generar consultas complejas de manera estructurada, legible y fácil de mantener [37].
- **Asociaciones:** Admite la definición de relaciones entre modelos, incluyendo uno-a-uno, uno-a-muchos y muchos-a-muchos, lo que permite estructurar los datos de forma eficiente y modular [37].
- **Migraciones:** Proporciona una herramienta de migración que permite gestionar y aplicar cambios en el esquema de la base de datos de manera consistente y controlada a lo largo del tiempo, asegurando la integridad en diferentes entornos de desarrollo [37].

Implementación del ORM en un proyecto Backend

En este apartado, se detalla paso a paso la implementación de una API GraphQL en **JavaScript** con **Node.js**, utilizando **Sequelize** como ORM para la gestión de datos y **Visual Studio Code (VS Code)** como entorno de desarrollo integrado (IDE, por sus siglas en inglés de *Integrated Development Environment*).

- **Creación del proyecto:** Crear una carpeta que servirá como directorio del proyecto; esto se puede realizar ejecutando el comando `mkdir nombre-del-proyecto` en la terminal del sistema operativo o creando la carpeta manualmente desde el explorador de Windows. Luego, es necesario acceder al directorio mediante el comando `cd nombre-del-proyecto`. Posteriormente, se puede abrir la carpeta en **VS Code** con el comando `code`, lo que permitirá gestionar el proyecto desde la interfaz del editor.
- **Inicializar proyecto:** Dentro de **VS Code**, se debe abrir una terminal de comandos para ejecutar instrucciones en el entorno del proyecto. Esto se puede hacer accediendo al menú **Terminal** y seleccionando la opción **New Terminal**. Este proceso permite interactuar con el sistema mediante la línea de comandos sin necesidad de salir del editor, facilitando la ejecución de comandos, la instalación de dependencias y la administración del entorno de desarrollo.

Inicializar el proyecto ejecutando el comando “npm init”, generará el archivo “package.json”, donde se almacenará la configuración del proyecto, incluyendo *nombre del paquete, versión, descripción, punto de entrada, comandos de prueba, repositorio Git, autor y licencia*.

⁴MySQL: <https://www.mysql.com>

⁵PostgreSQL: <https://www.postgresql.org>

⁶SQLite: <https://www.sqlite.org>

⁷Microsoft SQL Server: <https://www.microsoft.com/sql-server>

Terminal de VS Code

```
npm init
```

La Fig. 5.1 ilustra el proceso de inicialización del proyecto.

```
PS C:\Users\pc\Desktop\API-PIZZA-GRAPHQL> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (api-pizza-graphql)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\pc\Desktop\API-PIZZA-GRAPHQL\package.json:
```

Fig. 5.1. Inicialización del proyecto

La Fig. 5.2 muestra la estructura del archivo `package.json` generado, en donde se puede verificar la configuración del proyecto. Se recomienda mantener una estructura similar para evitar posibles errores durante el desarrollo.

```
1 {
2   "name": "api-pizza-graphql",
3   "version": "1.0.0",
4   "main": "index.js",
5   "scripts": {
6     "test": "echo \\\"Error: no test specified\\\" && exit 1"
7   },
8   "author": "",
9   "license": "ISC",
10  "description": ""
11 }
12
```

Fig. 5.2. Package.json

- **Crear el archivo `index.js`:** La estructura del proyecto sigue un esquema similar al presentado en el Capítulo 3. Es importante ubicar el archivo `index.js` dentro de la carpeta `src`. Además, se deben crear las siguientes carpetas para organizar correctamente el proyecto: `config`, destinada a la configuración de la base de datos; `controllers`, donde se gestionarán los controladores; y `type-system`, para definir el sistema de tipos.
- **Instalación de dependencias:** Es fundamental instalar las dependencias necesarias para el correcto

funcionamiento del proyecto. En este caso, se emplearán las siguientes dependencias:

Comando Utilizado

```
npm i apollo-server dotenv graphql-tools nodemon pg pg-hstore sequelize
```

En el Capítulo 3 se utilizaron las mismas dependencias para crear la API GraphQL. No obstante, en el presente ejercicio se incorporan nuevas dependencias específicas para la implementación del ORM. En particular, se hace uso de *sequelize*, una biblioteca que facilita la interacción con sistemas de gestión de bases de datos (SGBD) como *PostgreSQL*.

- **Configuración de variables de entorno:** Para definir las variables de entorno en el proyecto, se deben seguir los siguientes pasos:

1. Crear un archivo `.env` para gestionar de forma segura información confidencial, como claves de acceso, tokens, usuarios, contraseñas y URLs, evitando su exposición en el código fuente. Además, permite la personalización de variables de entorno según las necesidades del entorno de trabajo. El código fuente 5.1 muestra un ejemplo de su implementación. Se recomienda ubicar este archivo en la raíz del proyecto para facilitar su gestión y acceso.

```
DB_HOST=localhost
DB_PORT=5432
DB_DATABASE=pizza
DB_USER=postgres
DB_PASSWORD=1234admin
```

Código Fuente 5.1: Archivo para las variables de entorno

2. Crear un archivo para gestionar las variables de entorno definidas previamente. Para ello, se debe acceder a la carpeta **config** del proyecto y crear un archivo denominado `env-vars.js`. Una vez creado, es necesario importar la dependencia *dotenv* y exportar las variables de entorno a través de la función *config* para su uso en el proyecto; ver código fuente 5.2.

```
import * as dotenv from 'dotenv';

dotenv.config();

const config = {
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  database: process.env.DB_DATABASE,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  dialect: process.env.DIALECT
};

export { config };
```

Código Fuente 5.2: Archivo de configuración para las variables de entorno

- **Conexión a la base de datos:** En la carpeta **config**, se debe crear un archivo denominado `cnn.js` para gestionar la conexión con la base de datos. Para ello, es necesario importar la dependencia

sequelize, la cual facilita la comunicación con el sistema de gestión de bases de datos. Posteriormente, se debe instanciar el objeto *Sequelize* y configurar sus parámetros utilizando los valores de las variables de entorno definidas previamente, garantizando una conexión segura y flexible con la base de datos; ver código fuente 5.3.

```
import Sequelize from 'sequelize';
import { config } from './env-vars.js';

const sequelize = new Sequelize(
  config.database,
  config.user,
  config.password,
  {
    host: config.host,
    dialect: config.dialect,
    logging: false, // Desactiva los logs
  }
);

export { sequelize };
```

Código Fuente 5.3: Creación de la instancia de Sequelize

- **Creación de los modelos de la base de datos:** Los modelos son abstracciones que representan las tablas de la base de datos como entidades [37]. En **Sequelize**, un modelo es una clase que puede definirse de dos maneras: utilizando el generador de modelos proporcionado por **Sequelize** [38] o mediante una implementación manual, donde se define la estructura y las relaciones de la entidad dentro del código fuente. A continuación, se muestran los códigos fuente para implementar los modelos de la base de datos **Pizza**:

1. **Dependencias de Sequelize para definir modelos:** Importar los tipos de datos de **Sequelize** mediante la dependencia *DataTypes*, la cual permite utilizar diversos tipos de datos, incluyendo valores numéricos como *INTEGER* y *FLOAT*, cadenas de texto con *STRING* y valores booleanos con *BOOLEAN*. Posteriormente, se debe importar la instancia de **Sequelize** para establecer la conexión con la base de datos, como se muestra en el código fuente 5.4.

```
const DataTypes = require("sequelize");
const sequelize = require("../config/cnn");
```

Código Fuente 5.4: Dependencias de Sequelize para definir modelos

Esta porción de código fuente se debe repetir en la definición de cada modelo de las entidades de las tablas de la base de datos.

2. **Modelo de la entidad Pizza:** Crear el archivo *./models/pizza.js*; ver código fuente 5.5.

```
const DataTypes = require("sequelize");
const sequelize = require("../config/cnn");

const Pizza = sequelize.define(
  "Pizza",
  {
    piz_id: {
```

```
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
  },
  piz_name: {
    type: DataTypes.STRING,
  },
  piz_origin: {
    type: DataTypes.STRING,
  },
  piz_state: {
    type: DataTypes.BOOLEAN,
  },
  piz_description: {
    type: DataTypes.STRING,
  },
},
{
  tableName: "pizzas",
  timestamps: false,
}
);

module.exports = Pizza;
```

Código Fuente 5.5: Modelo de la entidad Pizza

3. **Modelo de la entidad Ingredient:** Crear el archivo `./models/ingredient.js`; ver código fuente 5.6.

```
const DataTypes = require("sequelize");
const sequelize = require("../config/cnn");

const Ingredient = sequelize.define(
  "Ingredient",
  {
    ing_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    ing_name: {
      type: DataTypes.STRING,
    },
    ing_calories: {
      type: DataTypes.FLOAT,
    },
    ing_state: {
      type: DataTypes.BOOLEAN,
    },
  },
  {

```

```

    tableName: "ingredients",
    timestamps: false,
  }
);

module.exports = Ingredient;

```

Código Fuente 5.6: Modelo de la entidad Ingredient

4. **Modelo de la entidad Pizza-Ingredient:** Crear el archivo `./models/pizza-ingredient.js`; ver código fuente 5.7.

A diferencia de los modelos anteriores, donde solo se importaron las dependencias y se definieron los campos de cada entidad, en esta entidad es necesario especificar su relación con otros modelos. Cabe destacar que **Sequelize** admite distintos tipos de relaciones entre tablas, lo que permite estructurar y gestionar la base de datos de manera más eficiente. Para más información sobre las relaciones soportadas, se recomienda consultar la **documentación oficial de Sequelize** [37].

```

const DataTypes = require("sequelize");
const sequelize = require("../config/cnn");
const Pizza = require("./pizza");
const Ingredient = require("./ingredient");

const PizzaIngredient = sequelize.define(
  "PizzaIngredient",
  {
    pi_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    piz_id: {
      type: DataTypes.INTEGER,
      references: {
        model: Pizza,
      },
    },
    ing_id: {
      type: DataTypes.INTEGER,
      references: {
        model: Ingredient,
      },
    },
    pi_portion: {
      type: DataTypes.INTEGER,
    },
  },
  {
    tableName: "pizzas_ingredients",
    timestamps: false,
  }
);

```

```
);

Pizza.belongsToMany(Ingredient, {
  through: PizzaIngredient,
  foreignKey: "piz_id",
});

Ingredient.belongsToMany(Pizza, {
  through: PizzaIngredient,
  foreignKey: "ing_id",
});

module.exports = PizzaIngredient;
```

Código Fuente 5.7: Modelo de la entidad Pizza-Ingredient

5. Modelo para Consultas Complejas (complex-queries): Crear el archivo `./models/complex-queries.js`.

En algunos casos, las consultas pueden volverse complejas debido al volumen de información o la naturaleza de los datos que se desean obtener. Para manejar este tipo de operaciones, **Sequelize** proporciona el método `query`, el cual permite ejecutar consultas en lenguaje SQL de manera directa. En el código fuente 5.8, se muestra un ejemplo en el que se utiliza dicho método para obtener el total de calorías de las pizzas y listar los ingredientes de cada una.

```
const sequelize = require("../config/cnn");
const { QueryTypes } = require("sequelize");

const totalCalories = async (piz_id) => {
  const total = await sequelize.query(
    `select sum(pi.pi_portion * i.ing_calories) total_calories
    from pizzas p, ingredients i, pizzas_ingredients pi
    where p.piz_id = pi.piz_id AND pi.ing_id = i.ing_id AND p.piz_id=?`,
    {
      plain: false,
      raw: false,
      replacements: [piz_id],
      type: QueryTypes.SELECT,
    }
  );
  return total;
};

const getIngredientsByPizza = async (piz_id) => {
  const ingredient = await sequelize.query(
    `select i.*, pi.pi_portion
    from pizzas p, ingredients i, pizzas_ingredients pi
    where p.piz_id = pi.piz_id
    AND pi.ing_id = i.ing_id
    AND p.piz_id=?`,
    {
      plain: false,
```



```

        raw: false,
        replacements: [piz_id],
        type: QueryTypes.SELECT,
    }
);
return ingredient;
};

module.exports = {totalCalories, getIngredientsByPizza};

```

Código Fuente 5.8: Modelo complex-queries

- **Creación del Sistema de Tipos (schema):** Crear el archivo `./type-system/schema.js`.

Para definir el esquema (schema) del **sistema de tipos** de la **API GraphQL**, se crea un archivo con extensión `.graphql` dentro de la carpeta **type-system**. En este archivo se especifican las consultas, mutaciones y tipos de datos que serán utilizados en la aplicación. Cabe destacar que en la Sección 3 se realizó un proceso similar. A continuación, el código fuente 5.9 muestra el sistema de tipos de la API GraphQL.

```

type Query {
  pizzas(piz_id: Int!): [pizzas]
  ingredients(ing_id: Int!): [ingredients]
}

type Mutation {
  createPizza(pizza: inputPizza!): pizzas
  updatePizza(pizza: updatePizza!): pizzas
  deletePizza(id: ID!): String
  createIngredient(ingredient: inputIngredient!): ingredients
  updateIngredient(ingredient: updateIngredient!): ingredients
  deleteIngredient(id: ID!): String
  deletePizzaIngredient(pizzaIngredient: pizzaIngredient!): String
}

type pizzas {
  piz_id: Int!
  piz_name: String!
  piz_origin: String!
  piz_description: String
  piz_state: Boolean!
  total_calories: Float
  ingredients: [ingredients]
}

type ingredients {
  ing_id: Int!
  ing_name: String!
  ing_calories: Float!
  ing_state: Boolean!
  pi_portion: Int
}

```

```
input inputPizza {
  piz_name: String!
  piz_origin: String!
  piz_description: String!
  piz_state: Boolean
  ingredientsPizza: [ingredietPortion]
}

input updatePizza {
  piz_id: Int!
  piz_name: String!
  piz_origin: String!
  piz_description: String!
  piz_state: Boolean
  ingredientsPizza: [ingredietPortion]
}

input inputIngredient {
  ing_name: String!
  ing_calories: Float!
  ing_state: Boolean
  pi_portion: Int
}

input updateIngredient {
  ing_id: Int!
  ing_name: String
  ing_calories: Float
  ing_state: Boolean
  pi_portion: Int
}

input ingredietPortion {
  ing_id: Int!
  pi_portion: Int
}

input pizzaIngredient {
  ing_id: Int
  piz_id: Int
}
```

Código Fuente 5.9: Sistema de tipos de la API GraphQL

- **Creación de Funciones de Resolución (Resolvers):** Crear el archivo `./resolvers/pizza.resolver.js`.

Cada campo definido en el esquema de GraphQL debe estar vinculado a una función de resolución, conocida como **resolver** [14], [39]. En este sentido, a diferencia de la Sección 3, donde los resolvers utilizaban la dependencia *pg-promise* para ejecutar consultas SQL en la base de datos, en esta ocasión se implementará el ORM **Sequelize** para gestionar la interacción con los datos. A continuación, se detalla la implementación de los resolvers de la API en el archivo `/pizza.resolver.js`, explicando su estructura y funcionalidad sección por sección:

1. Importar las dependencias, como se muestra en el código fuente 5.10. Se puede observar que las dependencias utilizadas corresponden a los modelos de las entidades de la base de datos definidas en los pasos anteriores.

```
import { Pizza } from "../models/pizza.js";
import { Ingredient } from "../models/ingredient.js";
import { PizzaIngredient } from "../models/pizza-ingredient.js";
import { totalCalories, getIngredientsByPizza }
from "../models/complex-queries.js";
```

Código Fuente 5.10: Importación de las dependencias necesaria

2. Luego se debe crear el resolver para realizar las consultas y las mutaciones como se aprecia en el siguiente código fuente 5.11.

```
const pizzaResolver = {
  Query: {
    // Aquí se definirá las funciones para las consultas
    // (ver Código Fuente 5.12)
  },
  Mutation: {
    // Aquí se definirá las funciones para las mutaciones
    // (ver Código Fuente 5.13)
  },
};

export default pizzaResolver;
```

Código Fuente 5.11: Creación del resolver

3. Crear consultas (*queries*) en **Sequelize** permite acceder a los datos de la base de datos de manera eficiente. Para ello, se pueden utilizar métodos como **findAll**, que recupera todos los registros de una entidad específica, como se muestra en el código fuente 5.12 a continuación. Del mismo modo, es posible emplear **findOne** para obtener la información de un registro específico.

```
Query: {
  async pizzas(root, { piz_id }) {
    if (piz_id == undefined) {
      return await Pizza.findAll();
    } else {
      const pizza = await Pizza.findOne({
        where: { piz_id },
      });
      return [pizza.dataValues];
    }
  },
  async ingredients(root, { ing_id }) {
    if (ing_id == undefined) {
      return await Ingredient.findAll();
    } else {
      const ingredient = await Ingredient.findOne({
        where: { ing_id },
      });
      return [ingredient.dataValues];
    }
  }
}
```

```
});
return [ingredient.dataValues];
}
},
},
```

Código Fuente 5.12: Creación del resolver

4. Crear las mutaciones para insertar, actualizar y eliminar pizzas e ingredientes, en el caso de las operaciones de inserción se puede utilizar el método **create** de **sequelize** que permite insertar registros nuevos, en el caso de la actualización existen los métodos **set** y **save** que se deben utilizar conjuntamente para realizar esta operación, en el caso de la eliminación de registros es posible utilizar el método **destroy**.

En el código fuente 5.13 se observa el método **create** para el registro de nuevas pizzas. Además, para agregar ingredientes a las pizzas, se utiliza el modelo de **PizzaIngredient**.

```
async createPizza(root, { pizza }) {
  try {
    if (pizza === undefined) {
      return null;
    } else {
      const newPizza = await Pizza.create({
        piz_name: pizza.piz_name,
        piz_origin: pizza.piz_origin,
        piz_description: pizza.piz_description,
        piz_state: pizza.piz_state,
      });

      if (pizza.ingredientsPizza.length > 0) {
        pizza.ingredientsPizza.forEach(async (element) => {
          await PizzaIngredient.create({
            piz_id: newPizza.dataValues.piz_id,
            ing_id: element.ing_id,
            pi_portion: element.pi_portion,
          });
        });
      }
      return newPizza;
    }
  } catch (error) {
    return error;
  }
}
```

Código Fuente 5.13: Creación de la mutación para registrar pizzas

El código fuente 5.14 muestra cómo se realiza la actualización del método **updatePizza**. El primer paso es utilizar el método **findOne** para obtener el registro específico y, posteriormente, mediante el método **set** establecer cuáles serán los nuevos valores del registro. Como paso final, se debe utilizar el método **save** para guardar los cambios realizados.

```
async updatePizza(root, { pizza }) {
```

```

try {
  if (pizza === undefined) {
    return null;
  } else {
    const Updatepizza = await Pizza.findOne({
      attributes: [
        "piz_name",
        "piz_origin",
        "piz_state",
        "piz_description",
        "piz_id",
      ],
      where: { piz_id: pizza.piz_id },
    });

    Updatepizza.set({
      piz_id: pizza.piz_id,
      piz_name: pizza.piz_name,
      piz_origin: pizza.piz_origin,
      piz_state: pizza.piz_state,
      piz_description: pizza.piz_description,
    });

    await Updatepizza.save();

    if (pizza.ingredientsPizza.length > 0) {
      await PizzaIngredient.destroy({
        where: { piz_id: pizza.piz_id },
      });

      pizza.ingredientsPizza.forEach(async (element) => {
        console.log("Pizza Ingredients")
        console.log({
          piz_id: pizza.piz_id,
          ing_id: element.ing_id,
          pi_portion: element.pi_portion,
        })
        await PizzaIngredient.create({
          piz_id: pizza.piz_id,
          ing_id: element.ing_id,
          pi_portion: element.pi_portion,
        });
      });
    }

    return Updatepizza;
  }
} catch (error) {
  return error;
}
}

```

Código Fuente 5.14: Creación de la mutación para actualizar pizzas

En el caso de la eliminación de registros, es conveniente utilizar el método **destroy** como se aprecia en el código fuente 5.15.

```
async deletePizza(root, value) {
  try {
    if (value === undefined) {
      return null;
    } else {
      const { id } = value;
      if (id > 0) {
        await PizzaIngredient.destroy({
          where: { piz_id: id },
        });
        await Pizza.destroy({
          where: { piz_id: id },
        });
      }
      return `Pizza ${id} eliminada correctamente`;
    }
  } catch (error) {
    return error;
  }
}
```

Código Fuente 5.15: Creación de la mutación para eliminar pizzas

Cabe indicar que el proceso es similar en el caso de los **ingredientes**. Se deben utilizar los métodos mencionados anteriormente para de esa manera crear las distintas mutaciones que permitan realizar las operaciones de insertar, actualizar y eliminar pizzas como se observa en el código fuente 5.16.

```
async createIngredient(root, { ingredient }) {
  try {
    if (ingredient === undefined) {
      return null;
    } else {
      console.log({ ingredient });
      const createdIngredient = await Ingredient.create({
        ing_name: ingredient.ing_name,
        ing_calories: ingredient.ing_calories,
        ing_state: true,
      });
      return createdIngredient;
    }
  } catch (error) {
    return error;
  }
}
```

```

    },
    async updateIngredient(root, { ingredient }) {
      try {
        if (ingredient === undefined) {
          return null;
        } else {
          const updateIngredient = await Ingredient.findOne({
            attributes: ["ing_name", "ing_calories", "ing_state", "ing_id"],
            where: { ing_id: ingredient.ing_id },
          });

          updateIngredient.set({
            ing_id: ingredient.ing_id,
            ing_name: ingredient.ing_name,
            ing_calories: ingredient.ing_calories,
            ing_state: ingredient.ing_state,
          });

          await updateIngredient.save();

          return updateIngredient;
        }
      } catch (error) {
        return error;
      }
    },
    async deleteIngredient(root, value) {
      try {
        if (value === undefined) {
          return null;
        } else {
          const { id } = value;
          if (id > 0) {
            await PizzaIngredient.destroy({
              where: { ing_id: id },
            });
            await Ingredient.destroy({
              where: { ing_id: id },
            });
          }
          return `Ingrediente ${id} eliminado correctamente`;
        }
      } catch (error) {
        return error;
      }
    },
  },

```

Código Fuente 5.16: Mutaciones para los ingredientes

Antes se mencionó que en ciertos casos es conveniente utilizar el método **query** de **sequelize** que permite crear consultas SQL directamente utilizando comandos propios como **SELECT**. En

el siguiente ejemplo se observa la creación de una consulta compleja; ver código fuente 5.17.

```
pizzas: {
  async ingredients(pizzas) {
    const ingredients = getIngredientsByPizza(pizzas.piz_id);
    return ingredients;
  },

  async total_calories(pizza) {
    const total = await totalCalories(pizza.piz_id);
    return total[0].total_calories;
  },
},
```

Código Fuente 5.17: Consultas utilizando el modelo complex-queries

- **Configuración del archivo index.js.** Una vez que se han establecido los modelos, los tipos de datos y el controlador, es necesario dirigirse hacia el archivo principal para realizar ciertos cambios que permitan ejecutar nuestra aplicación.

En el código fuente 5.18 se observa toda la configuración que se ha realizado en la modificación del archivo index, el cual se ejecuta para utilizar el API. Es necesario mencionar que tanto para las consultas y mutaciones se puede utilizar el cliente que viene por defecto, tal como se muestra en el Capítulo 3.

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';
import { sequelize } from '../config/cnn.js';
import path from "path";
import { fileURLToPath } from "url";
import { loadFilesSync } from "@graphql-tools/load-files";
import { mergeTypeDefs, mergeResolvers } from "@graphql-tools/merge";

// Obtener la ruta del directorio actual
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Cargar y combinar los schemas
const allSchemas = mergeTypeDefs(
  loadFilesSync(path.join(__dirname, "../schemas/**/*.graphql"))
);

// Cargar y combinar resolvers
const allResolvers = mergeResolvers(
  loadFilesSync(path.join(__dirname, "../resolvers/**/*.js"))
);

// Configuración del servidor Apollo
const server = new ApolloServer({
  typeDefs: allSchemas,
  resolvers: allResolvers
});
```

```
// Conexión hacia la base de datos con Sequelize
sequelize.authenticate().then(() => {
  console.log('Connected to PostgreSQL..');
  return startStandaloneServer(server, {
    listen: { port: 4000 }
  });
}).then(({ url }) => {
  console.log(`Server ready at: ${url}`);
}).catch((error) => {
  console.error('Error connecting to the database:', error);
});
```

Código Fuente 5.18: Configuración del archivo index.js

Creación del proyecto frontend en React

Es necesario recordar que en el Capítulo 4 se creó el **frontend** para realizar la administración de pizzas e ingredientes, en esta sección se utilizará dicho **frontend** para realizar algunas modificaciones y generar las consultas y utilizar mutaciones. En el caso del frontend se realiza los siguientes pasos:

1. Modificar la URL del cliente de Apollo para que se conecte hacia la nueva API que se encuentra en el puerto 5000; ver código fuente 5.19.

```
const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: new HttpLink({
    uri: 'http://localhost:5000',
  })
})
```

Código Fuente 5.19: Modificación del archivo App.js

2. Posteriormente utilizar el comando **npm start** para ejecutar la aplicación del frontend, una vez que la ejecución ha concluido se puede observar el componente principal de la aplicación tal como se visualiza en la Fig. 5.3.

A continuación, la Fig. 5.4 muestra la carga correcta de los registros en el caso del componente de



Fig. 5.3. Componente principal del aplicación

las pizzas. Por lo que se puede entender que la integración con el API está funcionando de forma correcta sin ningun inconveniente.

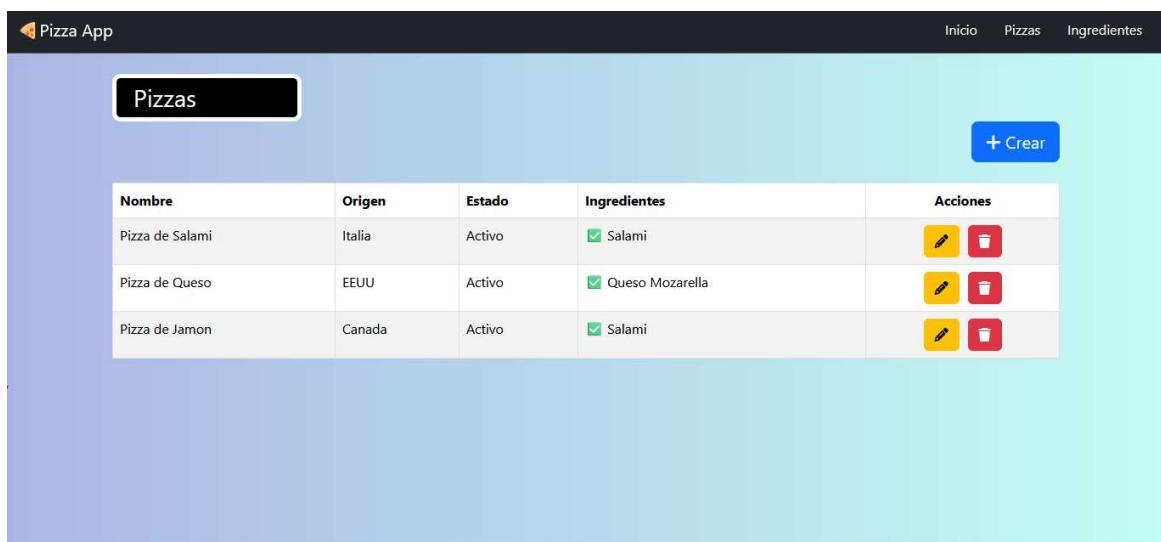


Fig. 5.4. Componente Pizzas

De igual forma el componente para visualizar los registros de los ingredientes se encuentra funcionando correctamente después de los cambios realizados en la API; ver la Fig. 5.5.

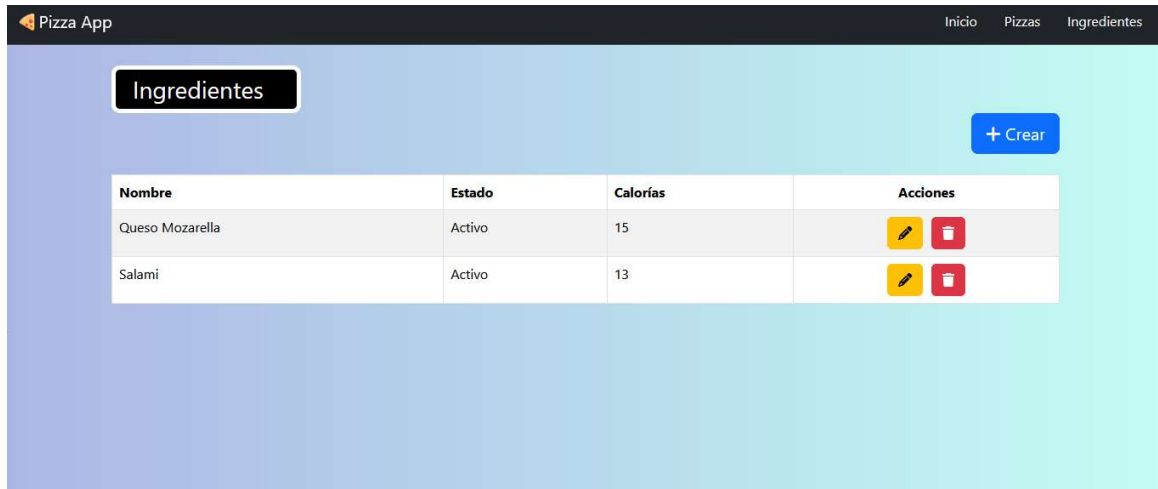


Fig. 5.5. Componente Ingredientes

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-01.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-01.git>

SEGURIDAD EN GRAPHQL

JWT y Autenticación

En el desarrollo de software, JSON Web Tokens (JWT) es una solución popular y eficiente para la autenticación y autorización de usuarios en aplicaciones web modernas.

Durante el proceso de autenticación, el servidor verifica las credenciales del usuario (como el nombre de usuario y contraseña) y, si son correctas, genera un JWT que se envía al cliente. Este token se almacena típicamente en el lado del cliente, ya sea en cookies seguras o en almacenamiento local, y se envía al servidor en cada solicitud posterior.

Este enfoque tiene varias ventajas en el desarrollo de software. Primero, reduce la complejidad en el manejo de sesiones, ya que el JWT contiene toda la información necesaria para la autenticación y autorización. Segundo, mejora la escalabilidad, ya que el servidor no necesita mantener el estado de cada usuario conectado. Finalmente, al ser un estándar ampliamente soportado, JWT se integra fácilmente con una variedad de frameworks y tecnologías, facilitando la construcción de aplicaciones web seguras y eficientes.

JSON Web Tokens

JSON Web Token (JWT), especifica un formato compacto y seguro para representar declaraciones (claims) que pueden ser transferidas entre dos partes. Estas declaraciones se codifican en formato JSON y pueden firmarse digitalmente para asegurar su integridad o encriptarse para proteger su confidencialidad. El propósito de JWT es facilitar la transmisión segura de información en entornos restringidos, como cabeceras de autorización HTTP o parámetros de consulta de la URL [40].

También se conoce a JWT como un método eficiente para la gestión de sesiones en entornos sin estado (stateless), lo que significa que el servidor no almacena información de sesión entre peticiones [41].

Un JWT consta de tres partes separadas por puntos:

- **Encabezado (Header):** Contiene la información del tipo de token y el algoritmo criptográfico utilizado.
- **Payload (Carga):** Contiene las declaraciones o información sobre el usuario y otros datos, como la fecha de emisión o expiración.
- **Firma (Signature):** Se genera utilizando el encabezado, la carga y una clave secreta, o una clave privada en caso de firmarse con algoritmos asimétricos.

Funcionamiento del JWT

Autenticación básica con JWT : La autenticación con JWT se describe como un proceso basado en la emisión de tokens por un servidor de autenticación tras la validación de credenciales. Estos tokens permiten a los usuarios acceder a recursos protegidos sin necesidad de reenviar sus credenciales en cada solicitud [42].

Autorización basada en JWT : Además de servir como método de autenticación, el JWT es útil para la autorización, que implica determinar a qué recursos o funcionalidades puede acceder un usuario. Dentro del payload del token, se incluyen claims o declaraciones que pueden especificar roles, permisos o niveles de acceso [43]. El JWT puede también incluir información sobre el nivel de confianza del usuario, basado en su comportamiento. Por ejemplo, si un usuario intenta acceder desde un dispositivo o una IP diferente, el sistema puede requerir pasos adicionales para autenticarlo nuevamente. Este tipo de autorización dinámica puede mejorar la seguridad sin comprometer la experiencia del usuario.

Creación del proyecto backend en NodeJS

Antes de crear el proyecto para desarrollar el API con autenticación es necesario realizar ciertos cambios en la base de datos que utilizamos , anteriormente debido a que se debe crear tablas para almacenar la información y los roles de los usuarios como los cambios mencionados la estructura de nuestra base datos se mantendrá de la siguiente forma como se aprecia en la Fig. 5.6.

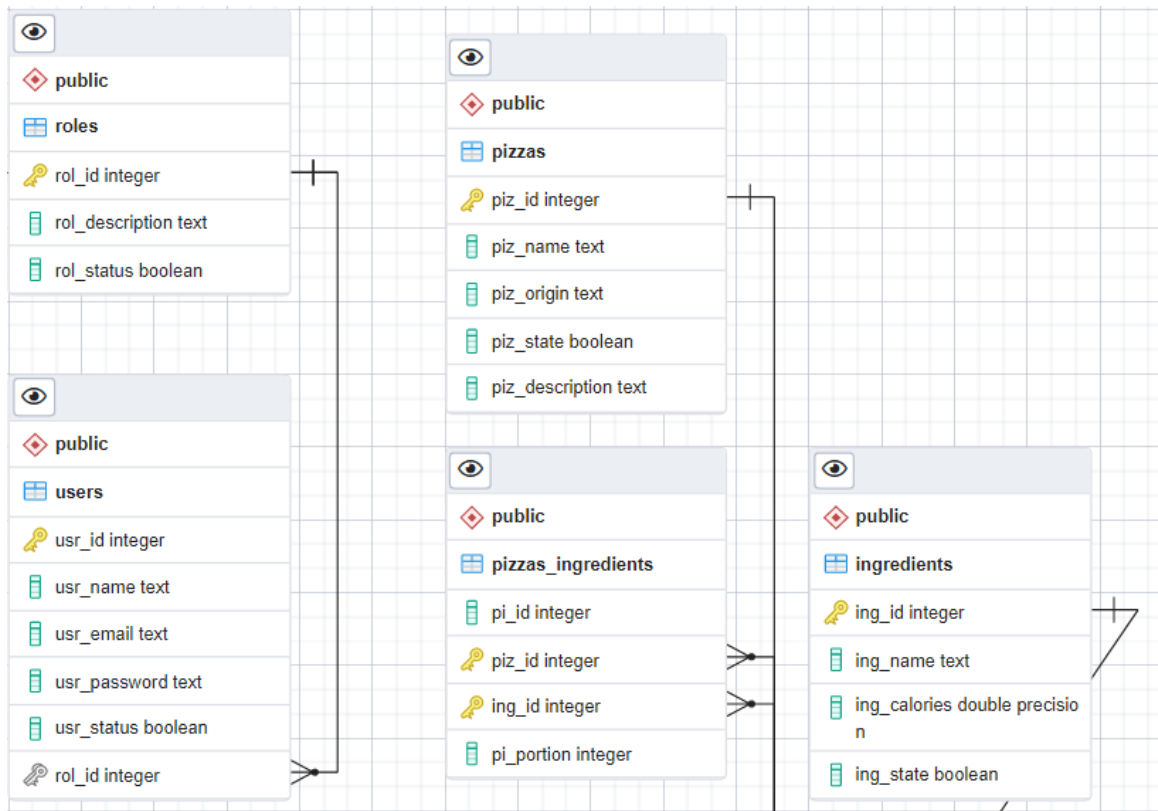


Fig. 5.6. Base de datos modificada

Actualmente, existen dos tablas. La tabla de usuarios que cuenta con los siguientes campos: **id**, **name**, **email**, **password**, **status** cabe mencionar que en el caso del campo **password** esta se encriptará utilizando una librería de js conocida como **bcryptjs**. Por otro lado en la tabla rol tiene los siguientes campos: **id**, **description** y **status**.

Durante esta práctica y a lo largo del resto de ciertas secciones se utilizará lo que se realizó en la primera sección del Capítulo 5. Como recomendación puede copiar lo que se realizó anteriormente en una carpeta distinta y continuar con el ejercicio siguiente. Para construir el backend con las rutas autenticadas es necesario seguir los siguientes pasos:

1. Como primer paso copiar el proyecto del backend existente a una ruta nueva puede escoger la ruta o lugar que usted desee.
2. Posteriormente se debe instalar las siguientes dependencias:

bcryptjs: Es una biblioteca JavaScript que implementa el algoritmo de **hash bcrypt** para la creación y validación de contraseñas seguras. A diferencia de **bcrypt**, que requiere dependencias nativas, **bcryptjs** está escrito completamente en JavaScript, lo que lo hace más fácil de instalar y utilizar en diferentes plataformas sin depender de compilaciones nativas [44].

jsonwebtoken: Es una librería para trabajar con JSON Web Tokens (JWT). Un JWT es un estándar abierto para transmitir información de manera segura entre dos partes como un objeto JSON. Es comúnmente usado para autenticación y autorización de usuarios en aplicaciones web [45].

3. Utilizar el siguiente comando:

Comando Utilizado

```
npm i jsonwebtoken bcryptjs
```

- Posteriormente en el archivo .env hay que agregar una nueva credencial para guardar el la palabra clave que servirá para encriptar y desencriptar el token, como se aprecia en el código fuente 5.20.

```
DB_HOST=localhost
DB_PORT=5432
DB_DATABASE=pizza
DB_USER=postgres
DB_PASSWORD=1234admin
DIALECT=postgres
JWTSECRET=api-graphql-secret99
```

Código Fuente 5.20: Variables de entorno

Creación de los modelos

En el ejercicio anterior se utilizó el ORM de sequelize para crear la conexión hacia la base de datos y definir cuáles son los modelos que se utilizarán, de igual forma dentro de esta sección se van a crear dos modelos nuevos que permitirán representar las entidades de los usuarios y de los roles que se utilizarán para la autenticación de los usuarios.

- Crear el modelo de la entidad usuario dentro de la carpeta **models** con el nombre user. El código fuente 5.21 muestra el modelo de la entidad usuario.

```
import { DataTypes } from "sequelize";
import { sequelize } from "../config/cnn.js";

const User = sequelize.define(
  "User",
  {
    usr_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    usr_name: {
      type: DataTypes.STRING,
    },
    usr_email: {
      type: DataTypes.STRING,
    },
    usr_password: {
      type: DataTypes.STRING,
    },
    usr_status: {
      type: DataTypes.BOOLEAN,
    },
    rol_id: {
      type: DataTypes.INTEGER,
    },
  },
  {
  },
);
```



```
{
  tableName: "users",
  timestamps: false,
}
);

export {User}
```

Código Fuente 5.21: Creación del modelo del usuario

6. Crear el modelo de la entidad rol dentro de la carpeta **models** con el nombre rol. El código fuente 5.22 muestra el modelo de la entidad rol.

```
import { DataTypes } from "sequelize";
import { sequelize } from "../config/cnn.js";

const Role = sequelize.define(
  "Rol",
  {
    rol_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    rol_description: {
      type: DataTypes.STRING,
    },
    rol_status: {
      type: DataTypes.BOOLEAN,
    },
  },
  {
    tableName: "roles",
    timestamps: false,
  }
);

export {Role}
```

Código Fuente 5.22: Creación del modelo del rol

7. Crear el modelo de la entidad usuario dentro de la carpeta **schemas** con el nombre usuario. El código fuente 5.23 muestra el esquema de la entidad usuario.

```
type users {
  usr_id: Int
  usr_name: String!
  usr_email: String!
  usr_password: String!
  usr_status: Boolean!
  rol_id: Int
  user_token: String
}

type Query {
  users(usr_id: Int!): [users]
}

type Mutation {
  loginUser(usr_email: String!, usr_password: String!): users
  createUser(user: addUser!): users
  updateUser(user: modifyUser!): users
  deleteUser(id: ID!): String
}

input addUser {
  usr_name: String!
  usr_email: String!
  usr_password: String!
  usr_status: Boolean!
  rol_id: Int,
  user_token: String
}

input modifyUser {
  usr_id: Int!
  usr_name: String
  usr_email: String
  usr_status: Boolean
  rol_id: Int,
  user_token: String
}
```

Código Fuente 5.23: Esquema para la entidad del usuario

8. Crear el modelo de la entidad rol dentro de la carpeta **schemas** con el nombre rol. El código fuente 5.24 muestra el esquema de la entidad rol.

```

type roles {
  rol_id: Int!
  rol_description: String
  rol_status: Boolean
}

type Query {
  roles(rol_id: Int): [roles]
}

input CreateRoleInput {
  rol_description: String
  rol_status: Boolean
}

input UpdateRoleInput {
  rol_id: Int!
  rol_description: String
  rol_status: Boolean
}

type Mutation {
  createRole(input: CreateRoleInput): roles
  updateRole(input: UpdateRoleInput): roles
  deleteRole(rol_id: Int!): Boolean
}

```

Código Fuente 5.24: Esquema para la entidad del rol

9. Posteriormente se debe crear una nueva carpeta con el nombre **services**. En esta carpeta se crea el servicio para obtener el token del usuario que ha iniciado sesión.
10. Dentro de la carpeta **services** se crea un nuevo archivo con el nombre de **auth.service**, luego es necesario importar la dependencia de JWT para poder verificar y validar el token. Además debe importar las variables de entorno que se establecieron anteriormente.

En el código fuente 5.25 se aprecia la función que se utilizará para obtener el token del usuario.

```

import jwt from 'jsonwebtoken'
import { config } from '../config/env-vars.js';

const getUserFromToken = (token) => {
  if (token) {
    return jwt.verify(token, config.JWTSECRET);
  }
  return "";
};

export {getUserFromToken}

```

Código Fuente 5.25: Función par obtener el token del usuario

11. El siguiente paso que se debe seguir es modificar el archivo index.js de la aplicación para que las rutas se mantengan protegidas mediante el uso de la autenticación. El código fuente 5.26 muestra la instancia del servidor de Apollo donde se colocó una propiedad nueva que permite establecer el contexto del mismo y agregar la correspondiente autorización y autenticación.

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';
import { sequelize } from '../config/cnn.js';
import { getUserFromToken } from '../services/auth.service.js';
import path from "path";
import { fileURLToPath } from "url";
import { loadFilesSync } from "@graphql-tools/load-files";
import { mergeTypeDefs, mergeResolvers } from "@graphql-tools/merge";

// Obtener la ruta del directorio actual
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Cargar y combinar los schemas
const allSchemas = mergeTypeDefs(
  loadFilesSync(path.join(__dirname, "../schemas/**/*.graphql"))
);

// Cargar y combinar resolvers
const allResolvers = mergeResolvers(
  loadFilesSync(path.join(__dirname, "../resolvers/**/*.js"))
);

// Configuración del servidor Apollo
const server = new ApolloServer({
  typeDefs: allSchemas,
  resolvers: allResolvers
});

// Conexión hacia la base de datos con Sequelize
sequelize.authenticate().then(() => {
  console.log('Connected to PostgreSQL..');
  return startStandaloneServer(server, {
    listen: { port: 4000 },
    context: ({ req }) => {
      const token = req.headers.authorization || "";
      const { userId } = getUserFromToken(token);
      return { userId };
    },
  });
}).then(({ url }) => {
  console.log(`Server ready at: ${url}`);
}).catch((error) => {
  console.error('Error connecting to the database:', error);
});
```

Código Fuente 5.26: Modificación del archivo index.js

Una vez realizados los cambios en el archivo `index.js` hay que modificar los controladores.

12. Crear el resolver para el usuario. El proceso es similar a lo que se hizo en el ejercicio anterior, sin embargo, existe una diferencia: se crearon las mutaciones para realizar el login, en donde se utiliza la dependencia **bcrypt** para comparar las claves y verificar que sean las mismas. En el caso de que no sean iguales, se mostrará un mensaje de error.

Cabe mencionar que existe un parámetro tanto en las mutaciones como en las consultas para validar que el usuario se encuentre autenticado. El parámetro mencionado se obtiene a través de la función del servicio de autenticación que se creó dentro de la carpeta de servicios, como se muestra en el código fuente 5.27.

```
import bcrypt from 'bcryptjs'
import jwt from 'jsonwebtoken'
import { config } from '../config/env-vars.js';
import { User } from '../models/users.js'
const userResolver = {
  Query: {
    async users(root, { usr_id }, { userId }) {
      if (!userId) {
        throw new Error("Not authenticated");
      } else {
        if (usr_id == undefined) {
          return await User.findAll();
        } else {
          const user = await User.findOne({
            where: { usr_id },
          });
          return [user.dataValues];
        }
      }
    },
  },
  Mutation: {
    createUser: async (root, { user }) => {
      const hashedPassword = await bcrypt.hash(user.usr_password, 10);
      const newUser = await User.create({
        usr_name: user.usr_name,
        usr_email: user.usr_email,
        usr_password: hashedPassword,
        usr_status: user.usr_status,
        rol_id: user.rol_id,
      });
      const { usr_id, usr_name, usr_email, usr_password, usr_status, rol_id } =
        newUser;
      const token = jwt.sign({ userId: usr_id }, config.JWTSECRET, {
        expiresIn: "1h",
      });
      return {
        usr_id: usr_id,
        usr_name: usr_name,
```

```

        usr_email: usr_email,
        usr_password: usr_password,
        usr_status: usr_status,
        rol_id: rol_id,
        user_token: token,
    };
},
loginUser: async (root, { usr_email, usr_password }) => {
    const user = await User.findOne({
        where: { usr_email: usr_email },
    });
    let token = "";

    if (!user) {
        throw new Error("User not found");
    } else {
        const valid = await bcrypt.compare(
            usr_password,
            user.dataValues.usr_password
        );
        if (!valid) {
            throw new Error("Invalid password");
        } else {
            token = jwt.sign({ userId: user.dataValues.usr_id },
                config.JWTSECRET, {
                    expiresIn: "1h",
                });
        }
    }
    return {
        usr_id: user.dataValues.usr_id,
        usr_name: user.dataValues.usr_name,
        usr_email: user.dataValues.usr_email,
        usr_password: user.dataValues.usr_password,
        usr_status: user.dataValues.usr_status,
        rol_id: user.dataValues.rol_id,
        user_token: token,
    };
},
async updateUser(root, { user }, { userId }) {
    if (!userId) {
        throw new Error("Not authenticated");
    } else {
        try {
            if (user === undefined) {
                return null;
            } else {
                const updateUser = await User.findOne({
                    attributes: [
                        "usr_name",
                        "usr_email",

```

```

        "usr_password",
        "usr_status",
        "usr_id",
    ],
    where: { usr_id: user.usr_id },
  });

  updateUser.set({
    usr_id: user.usr_id,
    usr_name: user.usr_name,
    usr_email: user.usr_email,
    usr_status: user.usr_status,
  });

  await updateUser.save();

  return updateUser;
}
} catch (error) {
  return error;
}
}
},
async deleteUser(root, value, { userId }) {
  if (!userId) {
    throw new Error("Not authenticated");
  } else {
    try {
      if (value === undefined) {
        return null;
      } else {
        const { id } = value;
        if (id > 0) {
          await User.destroy({
            where: { usr_id: id },
          });
        }
        return `Usuario ${id} eliminado correctamente`;
      }
    } catch (error) {
      return error;
    }
  }
},
},
};

export default userResolver;

```

Código Fuente 5.27: Creación del resolver para la entidad de los usuarios

En el caso del resto de controladores el proceso es similar. El único cambio que se debe tomar en cuenta es que se coloca el parámetro **userId** como se muestra en el código fuente 5.28.

```
import { DataTypes } from "sequelize";
import { sequelize } from "../config/cnn.js";

const Role = sequelize.define(
  "Rol",
  {
    rol_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    rol_description: {
      type: DataTypes.STRING,
    },
    rol_status: {
      type: DataTypes.BOOLEAN,
    },
  },
  {
    tableName: "roles",
    timestamps: false,
  }
);

export { Role }
```

Código Fuente 5.28: Creación del resolver para la entidad de los roles

- El código fuente 5.29 muestra los cambios realizados para hacer uso de estas consultas y mutaciones solo si el usuario se encuentra autenticado.

```
import { Pizza } from "../models/pizza.js";
import { Ingredient } from "../models/ingredient.js";
import { PizzaIngredient } from "../models/pizza-ingredient.js";
import { totalCalories,
  getIngredientsByPizza }
from "../models/complex-queries.js";

const pizzaResolver = {
  Query: {
    async pizzas(root, { piz_id }, { userId }) {
      if (!userId) {
        throw new Error("Not authenticated");
      } else {
        if (piz_id == undefined) {
          return await Pizza.findAll();
        } else {
          const pizza = await Pizza.findOne({
            where: { piz_id },

```

```

    });
    return [pizza.dataValues];
  }
}
},

async ingredients(root, { ing_id }, { userId }) {
  if (!userId) {
    throw new Error("Not authenticated");
  } else {
    if (ing_id == undefined) {
      return await Ingredient.findAll();
    } else {
      const ingredient = await Ingredient.findOne({
        where: { ing_id },
      });
      return [ingredient.dataValues];
    }
  }
},

Mutation: {
  async createPizza(root, { pizza }, { userId }) {
    if (!userId) {
      throw new Error("Not authenticated");
    } else {
      try {
        if (pizza === undefined) {
          return null;
        } else {
          const newPizza = await Pizza.create({
            piz_name: pizza.piz_name,
            piz_origin: pizza.piz_origin,
            piz_description: pizza.piz_description,
            piz_state: pizza.piz_state,
          });

          if (pizza.ingredientsPizza.length > 0) {
            pizza.ingredientsPizza.forEach(async (element) => {
              await PizzaIngredient.create({
                piz_id: newPizza.dataValues.piz_id,
                ing_id: element.ing_id,
                pi_portion: element.pi_portion,
              });
            });
            return newPizza;
          }
        }
      } catch (error) {
        return error;
      }
    }
  }
}
}

```

```

    }
  },
  async updatePizza(root, { pizza }, { userId }) {
    if (!userId) {
      throw new Error("Not authenticated");
    } else {
      try {
        if (pizza === undefined) {
          return null;
        } else {
          const Updatepizza = await Pizza.findOne({
            attributes: [
              "piz_name",
              "piz_origin",
              "piz_state",
              "piz_description",
              "piz_id",
            ],
            where: { piz_id: pizza.piz_id },
          });

          Updatepizza.set({
            piz_id: pizza.piz_id,
            piz_name: pizza.piz_name,
            piz_origin: pizza.piz_origin,
            piz_state: pizza.piz_state,
            piz_description: pizza.piz_description,
          });

          await Updatepizza.save();

          if (pizza.ingredientsPizza.length > 0) {
            await PizzaIngredient.destroy({
              where: { piz_id: pizza.piz_id },
            });

            pizza.ingredientsPizza.forEach(async (element) => {
              await PizzaIngredient.create({
                piz_id: pizza.piz_id,
                ing_id: element.ing_id,
                pi_portion: element.pi_portion,
              });
            });
            return Updatepizza;
          }
        }
      } catch (error) {
        return error;
      }
    }
  },

```

```

async deletePizza(root, value, { userId }) {
  if (!userId) {
    throw new Error("Not authenticated");
  } else {
    try {
      if (value === undefined) {
        return null;
      } else {
        const { id } = value;
        if (id > 0) {
          await PizzaIngredient.destroy({
            where: { piz_id: id },
          });
          await Pizza.destroy({
            where: { piz_id: id },
          });
        }
        return `Pizza ${id} eliminada correctamente`;
      }
    } catch (error) {
      return error;
    }
  }
},

async createIngredient(root, { ingredient }, { userId }) {
  if (!userId) {
    throw new Error("Not authenticated");
  } else {
    try {
      if (ingredient === undefined) {
        return null;
      } else {
        console.log({ ingredient });
        const createdIngredient = await Ingredient.create({
          ing_name: ingredient.ing_name,
          ing_calories: ingredient.ing_calories,
          ing_state: true,
        });

        return createdIngredient;
      }
    } catch (error) {
      return error;
    }
  }
},

async updateIngredient(root, { ingredient }, { userId }) {
  if (!userId) {
    throw new Error("Not authenticated");
  }
}

```

```

    } else {
      try {
        if (ingredient === undefined) {
          return null;
        } else {
          const updateIngredient = await Ingredient.findOne({
            attributes: ["ing_name", "ing_calories", "ing_state", "ing_id"],
            where: { ing_id: ingredient.ing_id },
          });

          updateIngredient.set({
            ing_id: ingredient.ing_id,
            ing_name: ingredient.ing_name,
            ing_calories: ingredient.ing_calories,
            ing_state: ingredient.ing_state,
          });

          await updateIngredient.save();

          return updateIngredient;
        }
      } catch (error) {
        return error;
      }
    }
  },
  async deleteIngredient(root, value, { userId }) {
    if (!userId) {
      throw new Error("Not authenticated");
    } else {
      try {
        if (value === undefined) {
          return null;
        } else {
          const { id } = value;
          if (id > 0) {
            await PizzaIngredient.destroy({
              where: { ing_id: id },
            });
            await Ingredient.destroy({
              where: { ing_id: id },
            });
          }
          return `Ingrediente ${id} eliminado correctamente`;
        }
      } catch (error) {
        return error;
      }
    }
  },
},
},

```

```

pizzas: {
  async ingredients(pizzas) {
    return getIngredientsByPizza(pizzas.piz_id);
  },

  async total_calories(pizza) {
    const total = await totalCalories(pizza.piz_id);
    return total[0].total_calories;
  },
},
};

export default pizzaResolver;

```

Código Fuente 5.29: Cambios en el resolver para la entidad pizzas

Para probar la aplicación se puede utilizar el cliente de **postman** o el propio cliente del servidor de Apollo tal como se realizó en el Capítulo 3. Es necesario que se agregue al token el prefijo **Bearer** seguido del token obtenido después de realizar el inicio de sesión.

Creación del proyecto frontend en React

Es necesario mencionar que se hará uso del proyecto frontend del Capítulo 4 dentro del cual se realizará los siguientes pasos:

Instalación de Dependencias

PrimeReact⁸, es una biblioteca de componentes de interfaz de usuario para React que ofrece una amplia gama de más de 80 componentes personalizables y ricos en funcionalidades [46]. Está diseñada para facilitar la creación de aplicaciones web modernas y receptivas, proporcionando herramientas que permiten a los desarrolladores implementar sus ideas de manera eficiente.

1. El primer paso es instalar la biblioteca de **PrimeReact** usando el siguiente comando de **npm**.

Comando utilizado

```
npm install primereact
```

Además de **PrimeReact**, es necesario instalar las siguientes dependencias adicionales: **primeicons** y **primelflex**, que proporcionan íconos y un sistema de diseño basado en **Flexbox**, respectivamente.

PrimeIcons es una biblioteca de íconos desarrollada por **PrimeTek** para complementar las bibliotecas de interfaz de usuario de **Prime**, como **PrimeNG**, **PrimeReact** y **PrimeFaces** [47]. Ofrece una amplia gama de íconos que pueden integrarse fácilmente en aplicaciones web.

PrimeFlex es una biblioteca ligera de utilidades CSS que proporciona un sistema de diseño basado en **Flexbox**, así como otras ayudas para el desarrollo de interfaces de usuario responsivas [48]. Está diseñada para acompañar a las bibliotecas de **Prime UI** y también puede utilizarse en páginas web estáticas.

⁸Primereact: <https://primereact.org/>

Comando utilizado

```
npm install primeicons primeflex
```

Importar Estilos

Existen distintos tipos de estilos que esta librería ofrece. En la Fig. 5.7 se visualiza algunos de los estilos que existen; por el momento cabe mencionar que para el desarrollo de los ejercicios que se realizará en este capítulo se utilizará el tema **lara-light-indigo**.

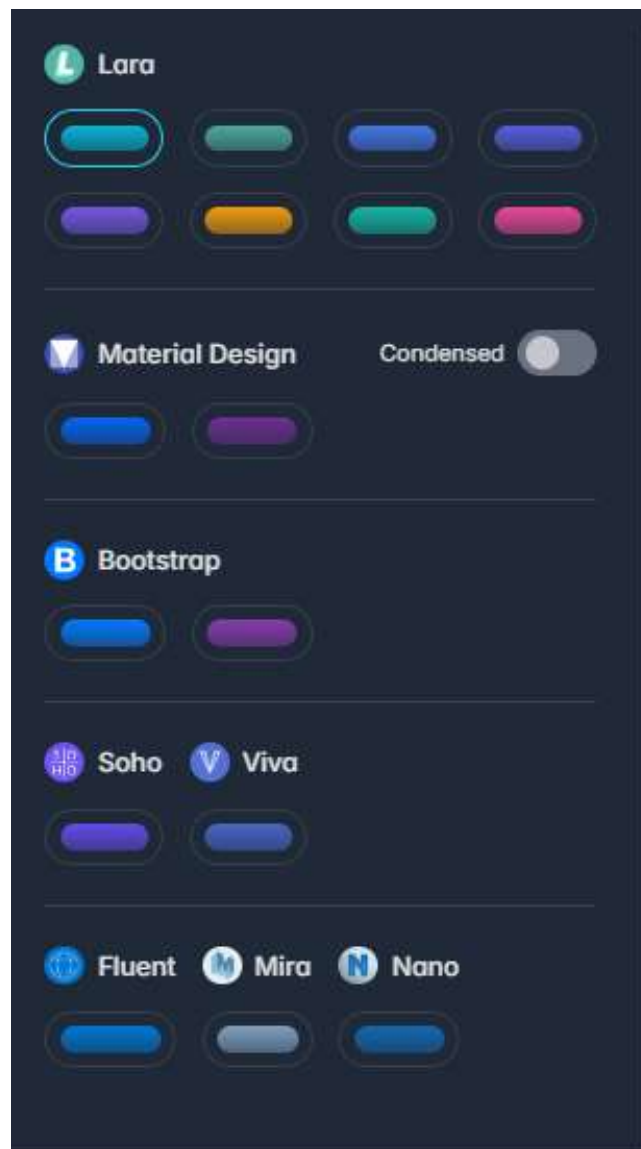


Fig. 5.7. Estilos Primeract

2. Se deben importar los estilos de PrimeReact y PrimeIcons en el archivo principal del proyecto. Esto garantiza que los componentes de **PrimeReact** tengan el diseño adecuado como se muestra en el código fuente 5.30.

```
import 'primereact/resources/themes/lara-light-indigo/theme.css';  
import 'primereact/resources/primereact.min.css';
```



```
import 'primeicons/primeicons.css';
import 'primeflex/primeflex.css';
```

Código Fuente 5.30: Importación de estilos

3. Es necesario realizar la configuración de Primereact como se aprecia en el código fuente 5.31 dentro del archivo main.jsx.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import "bootstrap/dist/css/bootstrap.min.css";
import "@fortawesome/fontawesome-free/css/all.min.css";
import "bootstrap/dist/js/bootstrap.bundle";
import { PrimeReactProvider } from "primereact/api";
import "primeicons/primeicons.css";
import "primereact/resources/themes/lara-light-indigo/theme.css";
import {
  ApolloClient,
  ApolloProvider,
  createHttpLink,
  InMemoryCache,
} from "@apollo/client";
import { setContext } from "@apollo/client/link/context";
import { BrowserRouter } from "react-router-dom";

const httpLink = createHttpLink({
  uri: "http://localhost:5000",
});

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem("token");
  return {
    headers: {
      ...headers,
      authorization: token ? `${token}` : "",
    },
  };
});

const client = new ApolloClient({
  link: authLink.concat(httpLink),
  cache: new InMemoryCache(),
});

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <ApolloProvider client={client}>
      <PrimeReactProvider>
        <BrowserRouter>
          <App />
```

```

    </BrowserRouter>
    </PrimeReactProvider>
    </ApolloProvider>
  ,
</StrictMode>,
)

```

Código Fuente 5.31: Configuración del archivo principal

4. Es importante mencionar la siguiente parte del código 5.31 descrito anteriormente, permite obtener el token que se obtiene del almacenamiento local, este token se guardará en el instante cuando el usuario realice el inicio de sesión; ver código fuente 5.32.

```

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem("token");
  return {
    headers: {
      ...headers,
      authorization: token ? `${token}` : "",
    },
  };
});

const client = new ApolloClient({
  link: authLink.concat(httpLink),
  cache: new InMemoryCache(),
});

```

Código Fuente 5.32: Configuración del archivo principal

Una vez que se ha instalado la biblioteca que se utilizarán para los estilos y componentes, se debe crear el componente que permitirá realizar el inicio de sesión, por lo que se debe seguir las siguientes indicaciones:

■ Creación del servicio para el manejo de los usuarios

Antes de crear los componentes que se utilizarán dentro de la aplicación es necesario crear el servicio que permitirá interactuar con las mutaciones y las consultas que se definieron en nuestra API, para crear el servicio el proceso es similar a lo que se ha visto anteriormente lo único que se toma en cuenta son los parámetros y los respectivos nombres de las operaciones, el código fuente 5.33 permite entender como se creó el servicio para el manejo de los usuarios.

```

import { gql } from "@apollo/client";

const GET_USERS = gql`
  query GetUsers($usr_id: Int!) {
    users(usr_id: $usr_id) {
      usr_id
      usr_name
      usr_email
      usr_status
      rol_id
    }
  }
`

```

```
        user_token
      }
    }
  }
`;

const LOGIN_USER = gql`
  mutation LoginUser($usr_email: String!, $usr_password: String!) {
    loginUser(usr_email: $usr_email, usr_password: $usr_password) {
      usr_id
      usr_name
      usr_email
      usr_status
      rol_id
      user_token
    }
  }
`;

const CREATE_USER = gql`
  mutation CreateUser($user: addUser!) {
    createUser(user: $user) {
      usr_id
      usr_name
      usr_email
      usr_status
      rol_id
      user_token
    }
  }
`;

const UPDATE_USER = gql`
  mutation UpdateUser($user: modifyUser!) {
    updateUser(user: $user) {
      usr_id
      usr_name
      usr_email
      usr_status
      rol_id
      user_token
    }
  }
`;

const DELETE_USER = gql`
  mutation deleteUser($deleteUserId: ID!){
    deleteUser(id: $deleteUserId)
  }
`;

export { GET_USERS, LOGIN_USER, CREATE_USER, UPDATE_USER, DELETE_USER };
```

Código Fuente 5.33: Creación del servicio user-service.js

■ Creación del componente Header.jsx

Dentro de la carpeta **components** en la carpeta shared es necesario crear un componente denominado **Header.jsx** que permitirá crear una barra superior como se muestra en el código fuente 5.34.

```
import React from "react";
const Header = () => {
  return (
    <nav
      className="navbar bg-dark border-bottom border-body"
      data-bs-theme="dark"
    >
      <div className="container-fluid">
        <span className="navbar-brand mb-0 h1">Pizza App</span>
      </div>
    </nav>
  )
}

export default Header
```

Código Fuente 5.34: Creación del componente Header.jsx

■ Creación del componente Home.jsx

El código fuente 5.35 muestra la creación del nuevo con el nombre de **home.jsx**. Después hay que realizar las importaciones y crear la función que permitirá exportar el componente.

```
import React, { useState } from "react";
import { useMutation } from "@apollo/client";
import { Button } from "primereact/button";
import { Card } from "primereact/card";
import { InputText } from "primereact/inputtext";
import { jwtDecode } from "jwt-decode";
import { useNavigate } from "react-router-dom";
import { showMessage } from "../services/message-service";
import { Password } from "primereact/password";
```

Código Fuente 5.35: Importación de las dependencias

El código fuente 5.36 muestra la creación de la función para exportar el componente y asignar un parámetro para identificar si el usuario está autenticado.

```
const Inicio = ({ setAuth }) => {
  return(
    );
  };

export default Inicio;
```

Código Fuente 5.36: Creación del componente home.jsx

Como siguiente paso el código fuente 5.37 muestra la creación de las variables en **React**. Las variables se manejan utilizando el **hook useState** que permite controlar el estado de las variables, en el caso de la variable **navigate** permite utilizar el método **useNavigate()**; para posteriormente definir enlaces o rutas de navegación hacia otros componentes.

```
const Inicio = ({ setAuth }) => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const navigate = useNavigate();
  return(

);
};

export default Inicio;
```

Código Fuente 5.37: Declaración de las variables del componente home.jsx

Ahora se debe utilizar la mutación **LOGINUSER** que se creó en el servicio anteriormente. El método **onCompleted** que se observa dentro de la mutación permite almacenar los datos del usuario y del token en el almacenamiento local una vez que se haya realizado el inicio de sesión como se muestra en el código fuente 5.38.

```
const Inicio = ({ setAuth }) => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const navigate = useNavigate();

  const [login, { error }] = useMutation(LOGIN_USER, {
    onCompleted: (data) => {
      localStorage.setItem("token", data.loginUser.user_token);
      localStorage.setItem("rol", data.loginUser.rol_id);
      localStorage.setItem("username", data.loginUser.usr_name);
      localStorage.setItem("status", data.loginUser.usr_status);
      setAuth(jwtDecode(data.loginUser.user_token));
    },
  });

  return(

);
};

export default Inicio;
```

Código Fuente 5.38: Mutación para el inicio de sesión

Después, es necesario crear un método **submit** que va a permitir realizar el inicio de sesión en caso de que sea exitoso se dirigirá hacia el componente de pizzas, en caso de que exista un error este se visualizará en un mensaje como se muestra en el código fuente 5.39.

```
const Inicio = ({ setAuth }) => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const navigate = useNavigate();

  const [login, { error }] = useMutation(LOGIN_USER, {
    onCompleted: (data) => {
      localStorage.setItem("token", data.loginUser.user_token);
      localStorage.setItem("rol", data.loginUser.rol_id);
      localStorage.setItem("username", data.loginUser.usr_name);
      localStorage.setItem("status", data.loginUser.usr_status);
      setAuth(jwtDecode(data.loginUser.user_token));
    },
  });

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      await login({
        variables: { usr_email: username, usr_password: password },
      });

      if (localStorage.getItem("status") === "true") {
        navigate("pizzas");
      } else {
        showMessage("Usuario no activo", "error");
      }
    } catch (error) {
      showMessage(error.message, "error");
    }
  };

  return(
    <div>
      <input type="text" value={username} onChange={setUsername} />
      <input type="password" value={password} onChange={setPassword} />
      <button onClick={handleSubmit}>Iniciar Sesión</button>
    </div>
  );
};

export default Inicio;
```

Código Fuente 5.39: Mutación para el inicio de sesión

Ahora es necesario crear la estructura del formulario para inicio de sesión como se aprecia en el código fuente 5.40 cabe mencionar el código **HTML** se debe colocar dentro del método **return**.

```
const Inicio = ({ setAuth }) => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const navigate = useNavigate();

  const [login, { error }] = useMutation(LOGIN_USER, {
    onCompleted: (data) => {
      localStorage.setItem("token", data.loginUser.user_token);
      localStorage.setItem("rol", data.loginUser.rol_id);
      localStorage.setItem("username", data.loginUser.usr_name);
      localStorage.setItem("status", data.loginUser.usr_status);
      setAuth(jwtDecode(data.loginUser.user_token));
    },
  });

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      await login({
        variables: { usr_email: username, usr_password: password },
      });

      if (localStorage.getItem("status") === "true") {
        navigate("pizzas");
      } else {
        showMessage("Usuario no activo", "error");
      }
    } catch (error) {
      showMessage(error.message, "error");
    }
  };

  return(
    <div className="card-login flex justify-content-center">
      <Header></Header>
      <Card className="md:w-25rem">
        <h2> {"Login"}</h2>
        <img
          src={require("../assets/images/pizza.jpg")}
          className="img-login-main"
          alt="..."
        />
        <br></br>
        <InputText
          placeholder="Correo"
          className="input-main-login"
          value={username}>
```



```

      onChange={(e) => setUsername(e.target.value)}
    />

    <Password
      value={password}
      onChange={(e) => setPassword(e.target.value)}
      feedback={true}
      toggleMask={true}

      placeholder="Clave"

    />
    <br />
    <br />
    <Button
      onClick={handleSubmit}
      label="Login"
      rounded
      className="input-main-login"
    />
  </Card>
</div>>
);
};

export default Inicio;

```

Código Fuente 5.40: Mutación para el inicio de sesión

Al ejecutar la aplicación se debe ver de la siguiente forma como se observa en la Fig. 5.8.

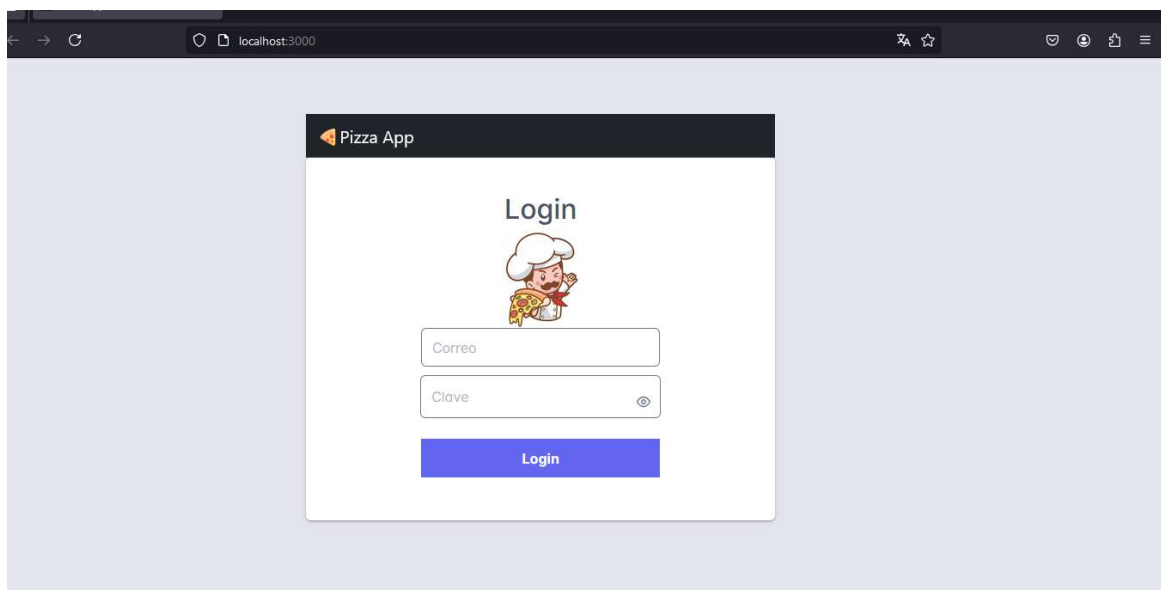


Fig. 5.8. Login

En el caso de que los componentes de formulario no se vean similares se recomienda revisar el

archivo index.css para verificar los estilos como se muestra en el código fuente 5.41.

```
.main-container {
  background-image: url("../assets/images/bg_1.jpg");
  background-color: #270505 !important;
}

.img {
  margin-bottom: 50px;
  text-align: center;
}

.title-page {
  color: #fff !important;
  font-size: 50px;
  position: relative;
  bottom: -150px;
  background-color: #000;
  width: 55%;
  border-radius: 10px;
  border: solid 5px #fff;
  padding-top: 6px;
  padding-bottom: 6px;
  padding-right: 10px;
}

.title {
  color: #fff !important;
  font-size: 28px;
  background-color: #000;
  width: 20%;
  border-radius: 10px;
  border: solid 5px #fff;
  padding-top: 6px;
  padding-bottom: 6px;
  padding-left: 20px;
  padding-right: 20px;
}

.card-container {
  background: transparent !important;
  margin-left: 60px;
  margin-top: 20px;
}

.link {
  color: #fff !important;
  font-size: 16px;
  cursor: pointer;
}
```

```
.link:hover {
  color: goldenrod !important;
  font-weight: bold;
}

footer {
  background-color: #000;
  width: 100%;
  position: relative;
  bottom: -50px;
}

.footer-title {
  color: #fff;
}

body::-webkit-scrollbar {
  width: 12px;
}

body::-webkit-scrollbar-track {
  background: rgb(188, 188, 188);
}

body::-webkit-scrollbar-thumb {
  background-color: rgb(62, 62, 63);
  border-radius: 20px;
  border: 3px solid rgb(188, 188, 188);
}

body {
  background: linear-gradient(to right, #e2e3eb, #e2e3eb);
  /* W3C, IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera 12+, Safari 7+ */
}

input {
  border: solid 1px #777777 !important;
}

.custom-card-container {
  width: 80%;
  text-align: center;
}

.header-welcome-login {
  text-align: center;
  margin-top: 15px;
  font-weight: 500;
  height: 33px;
}
```

```
}

.title-login {
  font-weight: bolder;
  margin-left: 50px;
  margin-top: -10px;
}

.input-login {
  margin-left: 35px;
}

.button-login {
  width: 100%;
  margin-left: 35px;
  font-size: 25px;
}

.img-login {
  width: 150%;
  height: 100%;
  margin-left: 323px;
  margin-top: -23px;
}

.img-login-main {
  max-width: 18%;
}

.card-login {
  width: 40%;
  height: 40%;
  margin-left: 365px;
  margin-top: 65px;
  text-align: center;
}

.input-main-login {
  width: 55%;
  margin-bottom: 10px;
  height: 45px;
}

.side-bar-button {
  height: 30px;
  background-color: transparent;
  border: 0;
  margin-left: -15px;
}
```

```
.side-bar-button:active {
  border: 0;
}

.side-bar-opt-button {
  margin-bottom: 5px;
  text-decoration: none;
  width: 100%;
  text-align: start;
}

.side-bar-opt-button:hover {
  background-color: rgb(158, 185, 208);
  color: #fff;
  text-decoration: none;
  width: 100%;
  text-align: start;
}

.btn-forgot-password {
  background-color: transparent;
  font-weight: normal !important;
}

.btn-forgot-password:hover {
  background-color: transparent;
  font-weight: normal !important;
}

.btn-sidebar {
  color: #fff;
  cursor: pointer;
}

/* App.css */

.password-input .p-inputtext {
  width: 100%;
  padding-left: 140px;
  text-align: start;
}

.password-panel {
  width: 100%;
  max-width: 300px;
}

.p-password input {
  text-align: start;
}
```

```
.p-password .p-password-toggle {
  right: 0.75rem;
}

.p-datatable-table {
  border: solid 1px rgb(159, 159, 159);
}

.p-datatable-thead{
  border: solid 1px rgb(159, 159, 159);
}

th {
  border: 1px rgb(159, 159, 159);
}

tr {
  border: 1px rgb(159, 159, 159);
}

.p-accordion-header > a {
  text-decoration: none;
  color: rgb(125, 125, 125);
  font-weight: normal ;
  font-weight: 8px;
}

.p-accordion{
  width: 200px;
  height: 50px;
}
```

Código Fuente 5.41: Estilos utilizados

Posteriormente es necesario crear un componente para proteger las rutas y solo dar acceso cuando el usuario se encuentre autenticado. Para eso es necesario crear una nueva carpeta llamada **auth**, dentro de esta carpeta hay que crear un nuevo componente con el nombre de **privateroute.jsx** como se aprecia en el código fuente 5.42 en donde se pasa como parámetro el token del usuario en caso de no tener un token mediante el componente **Navigate** se le redirige a la ruta inicial.

```
import { Navigate, useLocation } from 'react-router-dom';

export const PrivateRoute = ({auth, children }) => {

  return auth ? children : <Navigate to='/' />;
};

export default PrivateRoute
```

Código Fuente 5.42: Creación componente para proteger las rutas

Después, se debe modificar el archivo **App.jsx** para obtener el token del almacenamiento local como se observa en el método **useEffect** y luego agregar el componente **privateroute** en cada una de las rutas de la aplicación para que se mantengan protegidas, como se muestra en el código fuente 5.43.

```
import { Routes, Route } from "react-router-dom";
import PizzaList from "../components/pizza/pizza-list";
import Home from "../components/home";
import IngredientList from "../components/ingredient/ingredient-list";
import React, { useState, useEffect } from "react";
import { jwtDecode } from "jwt-decode";
import PrivateRoute from "../components/auth/privateroute";
function App() {
  const [auth, setAuth] = useState(null);

  useEffect(() => {
    const token = localStorage.getItem("token");
    if (token) {
      setAuth(jwtDecode(token));
    }
  }, []);

;

  return (
    <Routes>
      <Route path="/" element={<Home setAuth={setAuth}></Home>}></Route>
      <Route path="pizzas"
        element={ <PrivateRoute auth={auth}><PizzaList/></PrivateRoute>}
      ></Route>
      <Route
        path="ingredientes"
        element={<PrivateRoute auth={auth}><IngredientList/>
          </PrivateRoute>}
      ></Route>

      <Route
        path="usuarios"
        element={<PrivateRoute auth={auth}></PrivateRoute>}
      ></Route>
    </Routes>

  );
}
export default App;
```

Código Fuente 5.43: Modificación del archivo App.jsx

Además, es necesario crear componente que tenga un menú lateral denominado **Sidebar** para visualizar los opciones del usuario y poder salir de la sesión del usuario.

En este componente se debe crear una variable con el nombre visible que permitirá mostrar el menú lateral, además hay que utilizar el **navigate** para dirigir al usuario a las distintas rutas de la aplicación cabe mencionar que este componente aparecerá en la barra superior cuando el usuario haya iniciado sesión, como se muestra en el código fuente 5.44.

```
import React, { useState } from "react";
import { Sidebar } from "primereact/sidebar";
import { Button } from "primereact/button";
import { useNavigate } from "react-router-dom";
const SideBar = () => {
  const [visible, setVisible] = useState(false);
  const navigate = useNavigate();

  const navigateTo = (route) => {
    navigate(route);
  };

  const logOut = () => {
    localStorage.removeItem("token");
    localStorage.removeItem("rol");
    localStorage.removeItem("username");
    navigate("/");
  };

  return (
    <div className=" flex justify-content-start">
      <Sidebar visible={visible} onHide={() => setVisible(false)}>
        <div style={{ display: "inline-flex" }}>
          <img
            src={require("../assets/images/pizza.jpg")}
            className="img-login-main"
            alt="..."
            style={{ marginLeft: "5px" }}
          />
          <h3 style={{ marginLeft: "5px" }}>Menu</h3>
        </div>

        {localStorage.getItem("rol") == 1 && (
          <>
            <Button
              label="Dashboard"
              icon="pi pi-book"
              text
              className="side-bar-opt-button"
            />
            <br />
            <Button
              label="Usuarios"
              text
              icon="pi pi-users"
              className="side-bar-opt-button"
              onClick={() => {
```

```

        navigateTo("/usuarios");
      }}
    />
    <br />
  </>
)}

<Button
  label="Ingredientes"
  icon="pi pi-briefcase"
  className="side-bar-opt-button"
  text
  onClick={() => {
    navigateTo("/ingredientes");
  }}
/>
<br />
<Button
  label="Pizzas"
  text
  icon="pi pi-cart-minus"
  className="side-bar-opt-button"
  onClick={() => {
    navigateTo("/pizzas");
  }}
/>
<br />
<Button
  label="Salir"
  icon="pi pi-sign-out"
  text
  className="side-bar-opt-button"
  onClick={logout}
/>
</Sidebar>
<i
  className="pi pi-bars btn-sidebar"
  onClick={() => setVisible(true)}
></i>
</div>
);
};

export default SideBar;

```

Código Fuente 5.44: Creación del componente Sidebar.jsx

Dentro del componente **Navbar** es necesario crear una función para obtener el rol del usuario en este caso se llamará **getUserRol** es importante mencionar que por el momento se utilizarán datos estáticos para identificar el rol del usuario. En el código fuente 5.45 se muestra dicha función con el uso de menús dinámicos.

```
import React from "react";
import { Link } from "react-router-dom";
import SideBar from "../sidebar";

const NavBar = () => {
  const getUserRol = () => {
    let userName= "Operador: "
    if (parseInt(localStorage.getItem("rol") )== 1) {
      userName = "Admin: "
    }
    return userName
  };
  return (
    <nav
      className="navbar bg-dark border-bottom border-body"
      data-bs-theme="dark"
    >
      <div className="container-fluid">
        <SideBar></SideBar>
        <span className="navbar-brand mb-0 h1">Pizza App</span>

        <ul className="nav justify-content-end">
          <li className="nav-item">
            <Link to="/" className="nav-link link">
              <i className="pi pi-user"></i> {getUserRol()}
              + " " +
              localStorage.getItem('username')}
            </Link>
          </li>
        </ul>
      </div>
    </nav>
  );
};

export default NavBar;
```

Código Fuente 5.45: Modificación del archivo NavBar.jsx

Para probar la aplicación como primer paso se va a realizar el inicio de sesión; ver Fig. 5.9.

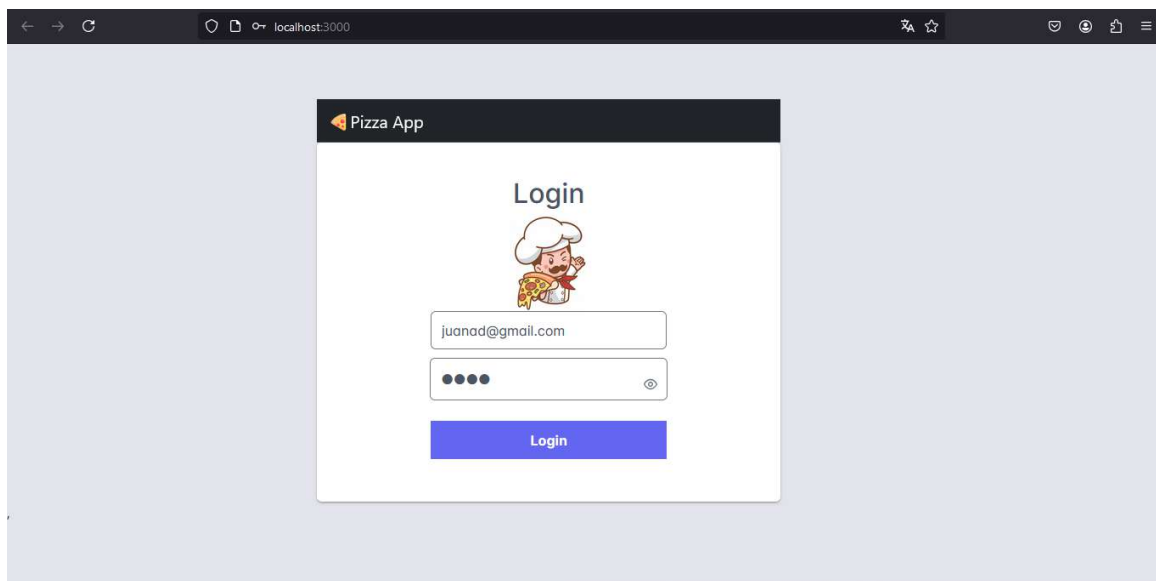


Fig. 5.9. Inicio de Sesión

Cuando se ha iniciado sesión de forma correcta se podrá visualizar la Fig. 5.10 en donde se observa como página principal las pizzas creadas, además se puede ver el nombre del usuario que ha iniciado sesión en la barra superior.

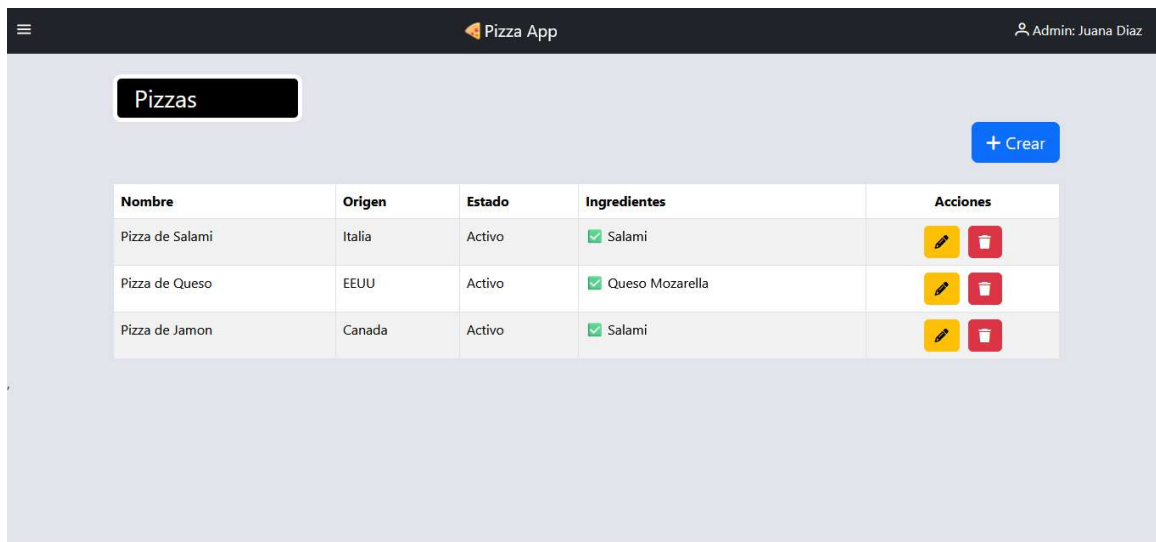


Fig. 5.10. Página principal

En la Fig. 5.11 se puede ver el menú lateral como se muestra existen distintas opciones en el caso de la opción usuarios y dashboard está se realizará posteriormente en los siguientes capítulos.

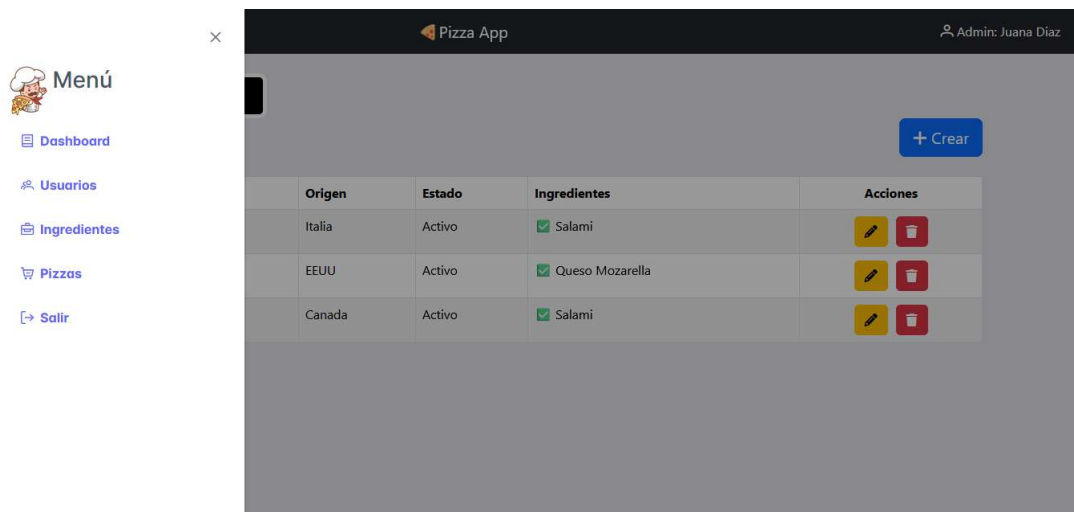


Fig. 5.11. Menú lateral

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-02.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-02.git>

MENÚS DINÁMICOS

A diferencia del ejercicio anterior en donde solo se mostraba el menú lateral con las opciones de forma estática, en este ejercicio el menú se mostrará de forma dinámica, es decir que solo se muestren las opciones a las que tiene acceso el usuario. Para continuar con este nuevo ejercicio es necesario seguir las siguientes instrucciones:

1. Modificar el modelo de la base de datos para agregar dos tablas más que permitirán almacenar la información de los menús en el caso de la entidad **menu** y para obtener los menús por usuarios se utilizará la entidad **rolmenu** tal como se aprecia en la Fig. 5.12.

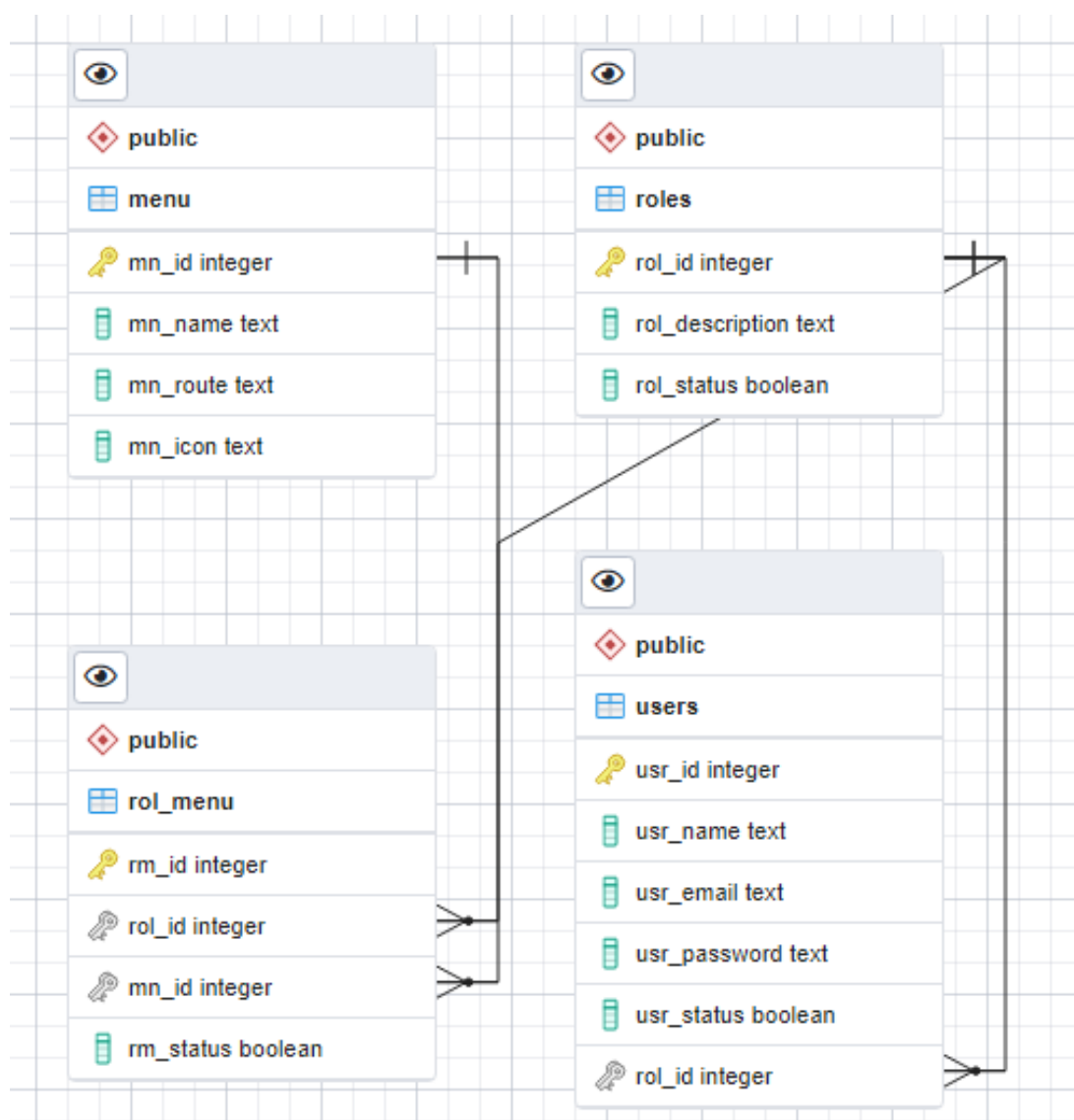


Fig. 5.12. Modelo modificado de la base de datos

- Es importante mencionar que en la tabla menu se agregaron los siguientes registros como se observa en la Fig. 5.13.

		mn_id [PK] integer 	mn_name text 	mn_route text 	mn_icon text 
1		1	Dashboard	/main	pi pi-book
2		2	Roles	/roles	pi pi-user
3		5	Ingredientes	/ingredientes	pi pi-briefcase
4		3	Accesos	/accesos	pi pi-unlock
5		4	Usuarios	/usuarios	pi pi-users
6		6	Pizzas	/pizzas	pi pi-cart-minus

Fig. 5.13. Tabla Menu

3. En la Fig. 5.14 se muestra la tabla rol-menu, donde se agregaron los siguientes registros:

		rm_id [PK] integer 	rol_id integer 	mn_id integer 	rm_status boolean 
1		7	1	1	true
2		8	1	2	true
3		9	1	3	true
4		10	1	4	true
5		11	1	5	true
6		12	2	1	true
7		13	1	6	true

Fig. 5.14. Tabla Rol Menu

Creación del proyecto backend en NodeJS

Cabe mencionar que se va a utilizar el proyecto backend del ejercicio anterior y dentro del proyecto mencionado es importante realizar los siguientes cambios:

1. En la carpeta de **models** se debe crear el modelo para la entidad menu; ver código fuente 5.46.


```
import { DataTypes } from "sequelize";
import { sequelize } from "../config/cnn.js";

const Menu = sequelize.define('Menu', {
  mn_id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  mn_name: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  mn_route: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  mn_icon: {
    type: DataTypes.TEXT,
    allowNull: false
  }
}, {
  tableName: 'menu',
  timestamps: false
});

export {Menu}
```

Código Fuente 5.46: Creación del modelo para la entidad menu

2. En la carpeta de **models** se debe crear el modelo para la entidad rolmenu; ver código fuente 5.47.

```
import { DataTypes } from "sequelize";
import { sequelize } from "../config/cnn.js";

const RoleMenu = sequelize.define('RoleMenu', {
  rm_id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  rol_id: {
    type: DataTypes.INTEGER,
    allowNull: false
  },
  mn_id: {
    type: DataTypes.INTEGER,
    allowNull: false
  },
  rm_status: {
    type: DataTypes.BOOLEAN,

```

```
    allowNull: false,  
    defaultValue: true  
  },  
{  
  tableName: 'rol_menu',  
  timestamps: false  
});  
export {RoleMenu}
```

Código Fuente 5.47: Creación del modelo para la entidad rolmenu

3. En la carpeta de **schemas** se debe crear el esquema de datos para la entidad menu; ver código fuente 5.48.

```
type Menu {  
  mn_id: Int!  
  mn_name: String!  
  mn_route: String!  
  mn_icon: String!  
}  
  
type Query {  
  menus: [Menu]  
  menu(mn_id: Int!): Menu  
}  
  
input MenuInput {  
  mn_name: String!  
  mn_route: String!  
  mn_icon: String!  
}  
  
input MenuUpdateInput {  
  mn_name: String  
  mn_route: String  
  mn_icon: String  
}
```

Código Fuente 5.48: Creación del esquema de datos para la entidad menu

4. En la carpeta de **schemas** se debe crear el esquema de datos para la entidad rolmenu; ver código fuente 5.49.

```
type RoleMenu {  
  rm_id: Int!  
  rol_id: Int!  
  mn_id: Int!  
  rm_status: Boolean!  
  menus: [Menu]  
}
```

```

type Menu {
  mn_id: Int!
  mn_name: String!
  mn_route: String!
  mn_icon: String!
}

input RoleMenuInput {
  rol_id: Int!
  mn_id: Int!
  rm_status: Boolean!
}

input RoleMenuUpdateInput {
  rm_id: Int!
  rol_id: Int
  mn_id: Int
  rm_status: Boolean
}

type Query {
  rolemenus(rol_id: Int!): [RoleMenu]
}

type Mutation {
  addRoleMenu(input: RoleMenuInput!): RoleMenu
  updateRoleMenu(input: RoleMenuUpdateInput!): RoleMenu
  deleteRoleMenu(rm_id: Int!): String
}

```

Código Fuente 5.49: Creación del esquema de datos para la entidad rolmenu

5. Para la creación de los **resolvers** y las **entidades** se debe seguir el mismo proceso realizado con el **ejercicio 01** en donde se hizo uso del **ORM** de **sequelize**. Antes de crear el **resolver** para esta entidad es necesario crear una consulta que permita obtener los menús para el rol asignado. Para lo cual se debe agregar el siguiente método dentro del modelo de **complex-queries**; ver código fuente 5.50.

```
const getMenuByRol = async (rol_id) => {
  const Menu = await sequelize.query(
    `SELECT mn.mn_id ,mn.mn_name, mn.mn_route, mn.mn_icon from menu mn ,
      rol_menu rm
    where rm.mn_id = mn.mn_id and rm.rol_id=? order by mn.mn_id`,
    {
      plain: false,
      raw: false,
      replacements: [rol_id],
      type: QueryTypes.SELECT,
    }
  );
  return Menu;
};
```

Código Fuente 5.50: Consulta para obtener los menus por rol

6. En la carpeta de **resolvers** se debe crear el resolver para la entidad rolmenu; ver código fuente 5.51.

```
import { RoleMenu } from "../models/role-menu.js";
import { getMenuByRol } from "../models/complex-queries.js";

const roleMenuResolver = {
  Query: {
    async rolemenus(root, { rol_id }, { userId }) {
      if (!userId) {
        throw new Error("Not authenticated");
      } else {
        if (rol_id === undefined) {
          return await RoleMenu.findAll();
        } else {
          const role_menu = await RoleMenu.findOne({
            where: { rol_id: rol_id },
          });
          return [role_menu.dataValues];
        }
      }
    },
  },
  Mutation: {
    addRoleMenu: async (_, { input }, { userId }) => {
      if (!userId) {
        throw new Error("Not authenticated");
      } else {
        const roleMenu = await RoleMenu.create(input);
        return roleMenu;
      }
    },
    updateRoleMenu: async (_, { input }, { userId }) => {
      if (!userId) {
```

```

        throw new Error("Not authenticated");
    } else {
        await RoleMenu.update(input, {
            where: { rm_id: input.rm_id },
        });
        return await RoleMenu.findByPk(input.rm_id);
    }
},
deleteRoleMenu: async (_, { rm_id }, {userId }) => {
    console.log("rm_id ", rm_id)
    if (!userId) {
        throw new Error("Not authenticated");
    } else {
        await RoleMenu.destroy({
            where: { rm_id:rm_id },
        });
        return `Acceso Rol Menu ${rm_id} eliminado correctamente`;
    }
},
},
RoleMenu: {
    async menus(rolemenu) {
        return getMenuByRol(rolemenu.rol_id);
    },
},
};

export default roleMenuResolver;

```

Código Fuente 5.51: Creación del resolver para la entidad rolmenu

7. En la carpeta de **resolvers** se debe crear el resolver para la entidad menu; ver código fuente 5.52.

```

import { Menu } from "../models/menu.js";

const menuResolver = {
    Query: {
        async menus(_, { mn_id }, { userId }) {
            if (!userId) {
                throw new Error("Not authenticated");
            }

            if (mn_id === undefined) {
                return await Menu.findAll();
            } else {
                const menu = await Menu.findOne({ where: { mn_id } });
                if (!menu) {
                    throw new Error(`Menu with id ${mn_id} not found`);
                }
                return [menu.dataValues];
            }
        },
    },
};

```

```
    },  
  };  
  
export default menuResolver;
```

Código Fuente 5.52: Creación del resolver para la entidad menu

Una vez que se realizaron todos esos cambios, se puede utilizar clientes como **Postman** o el mismo cliente del servidor de Apollo para verificar el funcionamiento.

Creación del proyecto frontend en React

En el proyecto del frontend anterior se deben realizar los siguientes cambios:

1. Crear un servicio en la carpeta **service**, que permitirá visualizar los menús que se encuentran asociados a cierto rol, por ejemplo en el caso del rol de administrador tendrá acceso a todos los menús que estén disponibles en la tabla menú de la base de datos, luego hay que especificar las mutaciones y las consultas como se aprecia en el código fuente 5.53.

```
import { gql } from "@apollo/client";  
  
const GET_ROLE_MENU = gql`  
  {  
    rolemenus {  
      rm_id  
      mn_id  
      rm_status  
      rol_id  
    }  
  }  
`;  
  
const GET_USER_MENU = gql`  
query ($roleId: Int) {  
  rolemenus(rol_id: $roleId) {  
    menus {  
      mn_id  
      mn_name  
      mn_route  
      mn_icon  
    }  
  }  
}  
`;  
  
const CREATE_ROLE_MENU = gql`  
mutation addRoleMenu($input: RoleMenuInput!) {  
  addRoleMenu(input: $input) {  
    mn_id  
  }  
}  
`;
```

```

        rm_status
        rol_id
      }
    }
  }
};

const UPDATE_ROLE_MENU = gql`
  mutation updateRoleMenu($input: RoleMenuUpdateInput!) {
    updateRoleMenu(input: $input) {
      mn_id
      rm_status
      rol_id
    }
  }
`;

const DELETE_ROLE_MENU = gql`
  mutation deleteRoleMenu($rmId: Int!) {
    deleteRoleMenu(rm_id: $rmId)
  }
`;

export {
  GET_ROLE_MENU,
  CREATE_ROLE_MENU,
  UPDATE_ROLE_MENU,
  DELETE_ROLE_MENU,
  GET_USER_MENU
};

```

Código Fuente 5.53: Creación del servicio para obtener los menús por rol

2. Luego es necesario crear un servicio para interactuar con las operaciones y mutaciones de la tabla menú como se aprecia en el código fuente 5.54.

```

import { gql } from "@apollo/client";
const GET_MENUS = gql`
  {
    menus {
      mn_id
      mn_name
      mn_route
    }
  }
`;

export { GET_MENUS };

```

Código Fuente 5.54: Creación del servicio para la tabla menu

Creación del componente Header

3. A continuación, es necesario crear un componente que funcione como encabezado para la página principal. En el caso de este tipo de componentes se debe crear la carpeta **shared** debido a que en ésta se encuentran los componentes genéricos que se van a utilizar en cada parte de la aplicación; ver código fuente 5.55.

```
import React from "react";
import { Link } from "react-router-dom"
const Header = () => {
  return (
    <nav
      className="navbar bg-dark border-bottom border-body"
      data-bs-theme="dark"
    >
      <div className="container-fluid">
        <span className="navbar-brand mb-0 h1"> Pizza App</span>
        <ul className="nav justify-content-end">
          <li className="nav-item">
            <Link to="/login" className="nav-link link">
              <i className="pi pi-user">Login</i>
            </Link>
          </li>
        </ul>
      </div>
    </nav>
  )
}
```

Código Fuente 5.55: Creación del componente header

Creación del componente Home

4. Es importante un componente principal con el nombre **home.jsx** que sea la página de bienvenida como se observa en el código fuente 5.56. Es similar al que se usó, en los capítulos anteriores, sin embargo hay que realizar un cambio al utilizar el componente **header** para crear un encabezado y dirigirse hacia el inicio de sesión.

```
import React from "react";
import Header from "../shared/header";
import image from "../assets/images/bg_1.png"

const MainPage = () => {
  return (
    <div className="main-container">
      <Header></Header>
      <div className="card mb-3 card-container">
        <div className="row g-0">
          <div className="col-md-4">
            <img
              src={image}
            >
          </div>
        </div>
      </div>
    </div>
  )
}
```

```

        className="img-fluid rounded-start"
        alt="..."
    />
</div>
<div className="col-md-8">
    <div className="card-body">
        <h5 className="card-title title-page">Bienvenidos</h5>
    </div>
</div>
</div>
<div>
    <br></br>
    <footer className="d-flex flex-wrap
justify-content-between align-items-center py-3 my-4 border-top">
    <p className="col-md-4 mb-0 footer-title">2024 Pizza App</p>
    <a
        href="/"
        className="col-md-4 d-flex align-items-center
justify-content-center mb-3
mb-md-0 me-md-auto link-dark text-decoration-none"
    ></a>
    </footer>
</div>
</div>
);
};

export default MainPage;

```

Código Fuente 5.56: Creación del componente home

Modificación del componente Login

- El código fuente 5.57 hace referencia al componente que se creó en el ejercicio anterior y genera modificaciones. Para el inicio de sesión es importante mencionar que se debe crear un componente con el nombre **Login** debido a que durante este ejercicio se realizarán cambios.

```

import React, { useState } from "react";
import { useMutation } from "@apollo/client";
import { Button } from "primereact/button";
import { Card } from "primereact/card";
import { InputText } from "primereact/inputtext";
import { LOGIN_USER } from "../services/user-service";
import { jwtDecode } from "jwt-decode";
import { useNavigate } from "react-router-dom";
import { showMessage } from "../services/message-service";
import { Password } from "primereact/password";
import {
    setUserStatus,
    setRolUser,
    setTokenUser,

```

```

    setUsername,
    getUserStatus,
  } from "../services/auth-service";
import HeaderLogin from "../shared/header-login";

const Login = ({ setAuth }) => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const navigate = useNavigate();

  const [login, { error }] = useMutation(LOGIN_USER, {
    onCompleted: (data) => {
      setTokenUser(data.loginUser.user_token);
      setRolUser(data.loginUser.rol_id);
      setUsername(data.loginUser.usr_name);
      setUserStatus(data.loginUser.usr_status);
      setAuth(jwtDecode(data.loginUser.user_token));
    },
  });

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      await login({
        variables: { usr_email: username, usr_password: password },
      });

      if (getUserStatus() === "true") {
        navigate("/welcome");
      } else {
        showMessage("Usuario no activo", "error");
      }
    } catch (error) {
      showMessage(error.message, "error");
    }
  };

  return (
    <div className="card-login flex justify-content-center">
      <HeaderLogin></HeaderLogin>
      <Card className="md:w-25rem">
        <h2> {" Iniciar Sesión "}</h2>
        <img
          src={require("../assets/images/pizza.jpg")}
          className="img-login-main"
          alt="..."
        />
        <br></br>
        <InputText
          placeholder="Correo"
          className="input-main-login"

```

```

        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />

      <Password
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        feedback={true}
        toggleMask={true}
        inputClassName="p-inputtext"
        placeholder="Contraseña"
        weakLabel="Débil"
        mediumLabel="Media"
        strongLabel="Fuerte"
        panelClassName="password-panel"
        className="password-input"
      />
    <br />
    <br />
    <Button
      onClick={handleSubmit}
      label="Iniciar Sesión"
      rounded
      className="input-main-login"
    />
  </Card>
</div>
);
};

export default Login;

```

Código Fuente 5.57: Modificación del componente Login

Creación del componente NotFound

- En ciertos casos hay rutas desconocidas o que no pertenecen a la aplicación, en esos casos es necesario crear un componente que muestre información para que el usuario entienda que la dirección ingresada es inválida; el código fuente 5.58 permite realizar esa tarea.

```

import React from "react";
import Header from "../shared/header";
import image from "../assets/images/Error-404.png";

const NotFoundPage = () => {
  return (
    <div className="main-container">
      <Header></Header>
      <div className="card mb-3 card-container">
        <div style={{textAlign:'center'}}>
          <img src={image} className="img-notfound" alt="..." />
        </div>
      </div>
    </div>
  );
};

```

```

    </div>
    <div>
      <br></br>
      <footer className="d-flex flex-wrap justify-content-between
align-items-center py-3 my-4 border-top">
        <p className="col-md-4 mb-0 footer-title">2024 Pizza App</p>
        <a
          href="/"
          className="col-md-4 d-flex align-items-center
justify-content-center
mb-3 mb-md-0
me-md-auto
link-dark
text-decoration-none"
        ></a>
      </footer>
    </div>
  </div>
);
};

export default NotFoundPage;

```

Código Fuente 5.58: Componente NotFound

Creación del componente RolMenuForm

- Este componente se debe crear dentro de la carpeta **components** con el nombre **access-menu.jsx**; ver código fuente 5.59.

```

import React, { useEffect } from "react";
import { useState } from "react";
import { useMutation, useQuery } from "@apollo/client";
import {
  GET_ROLE_MENU,
  CREATE_ROLE_MENU,
  UPDATE_ROLE_MENU
} from "../services/access-service";
import {
  GET_MENUS
} from "../services/menu-service";
import { GET_ROLES } from "../services/role-service";
import { showMessage } from "../services/message-service";

```

Código Fuente 5.59: Importación de las dependencias

8. Posteriormente, es necesario crear la función para exportar el componente y además declarar las variables que se van a utilizar; ver código fuente 5.60.

```
const RoleMenuForm = ({ rolemenu }) => {
  const rolelist = useQuery(GET_ROLES);
  const menulist = useQuery(GET_MENUS);
  const [state, setState] = useState(false);
  const [rolId, setrolId] = useState(0);
  const [mnId, setmnId] = useState(0);
  const [id, setId] = useState(0); };

export default RoleMenuForm;
```

Código Fuente 5.60: Función del componente RoleMenuForm

9. Se debe crear los distintos métodos que permitirán actualizar y registrar nuevos menús; ver código fuente 5.61.

```
const RoleMenuForm = ({ rolemenu }) => {
  const rolelist = useQuery(GET_ROLES);
  const menulist = useQuery(GET_MENUS);
  const [state, setState] = useState(false);
  const [rolId, setrolId] = useState(0);
  const [mnId, setmnId] = useState(0);
  const [id, setId] = useState(0);

  useEffect(() => {
    if (rolemenu !== null) {
      setId(rolemenu.rm_id);
      console.log(rolemenu.rm_id)
      setState(rolemenu.rm_status);
      setrolId(rolemenu.rol_id);
      setmnId(rolemenu.mn_id);
    }
  }, [rolemenu]);

  // Mutations
  const [createRoleMenu] = useMutation(CREATE_ROLE_MENU, {
    refetchQueries: [{ query: GET_ROLE_MENU }],
  });

  const [updateRoleMenu] = useMutation(UPDATE_ROLE_MENU, {
    refetchQueries: [{ query: GET_ROLE_MENU }],
  });

  const CleanForm = () => {
    setmnId(0);
    setrolId(0);
    setId(0);
    setState(false);
  };
};
```

```

const CreateRoleMenu = (event) => {
  event.preventDefault();
  if (parseInt(rolId) > 0 && parseInt(mnId) > 0) {
    createRoleMenu({
      variables: {
        input: {
          mn_id: parseInt(mnId),
          rm_status: state,
          rol_id: parseInt(rolId)
        },
      },
    })
    .then(() => {
      showMessage("Acceso creado correctamente", "success");
    })
    .catch(() => {
      showMessage("Error al crear el acceso", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar sin seleccionar", "warning");
  }
};

const UpdateRoleMenu = (event) => {
  event.preventDefault();
  if (parseInt(rolId) > 0 && parseInt(mnId) > 0) {
    updateRoleMenu({
      variables: {
        input: {
          rm_id: parseInt(id),
          mn_id: parseInt(mnId),
          rm_status: state,
          rol_id: parseInt(rolId)
        },
      },
    })
    .then(() => {
      showMessage("Acceso actualizado correctamente", "success");
    })
    .catch((err) => {
      console.log(err);
      showMessage("Error al actualizar el acceso", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar sin seleccionar", "warning");
  }
};

const changeState = () => {

```



```

    if (state) {
      setState(false);
    } else {
      setState(true);
    }
  };

export default RoleMenuForm;

```

Código Fuente 5.61: Funcion del componente RoleMenuForm

- Para definir la estructura HTML del formulario dentro del componente, es importante destacar que ésta se encuentra dentro de la palabra clave **return**. Es en este bloque donde se inserta el código HTML que React utiliza para renderizar los componentes creados. A continuación, se muestra el código completo del componente RoleMenuForm, el cual incluye la estructura HTML del formulario; ver código fuente 5.62.

```

import React, { useEffect } from "react";
import { useState } from "react";
import { useMutation, useQuery } from "@apollo/client";
import {
  GET_ROLE_MENU,
  CREATE_ROLE_MENU,
  UPDATE_ROLE_MENU
} from "../../services/access-service";
import {
  GET_MENUS
} from "../../services/menu-service";
import { GET_ROLES } from "../../services/role-service";

import { showMessage } from "../../services/message-service";

const RoleMenuForm = ({ rolemenu }) => {
  const rolelist = useQuery(GET_ROLES);
  const menulist = useQuery(GET_MENUS);
  const [state, setState] = useState(false);
  const [rolId, setrolId] = useState(0);
  const [mnId, setmnId] = useState(0);
  const [id, setId] = useState(0);

  useEffect(() => {
    if (rolemenu !== null) {
      setId(rolemenu.rm_id);
      console.log(rolemenu.rm_id)
      setState(rolemenu.rm_status);
      setrolId(rolemenu.rol_id);
      setmnId(rolemenu.mn_id);
    }
  }, [rolemenu]);

```

```
// Mutations
const [createRoleMenu] = useMutation(CREATE_ROLE_MENU, {
  refetchQueries: [{ query: GET_ROLE_MENU }],
});

const [updateRoleMenu] = useMutation(UPDATE_ROLE_MENU, {
  refetchQueries: [{ query: GET_ROLE_MENU }],
});

const CleanForm = () => {
  setmnId(0);
  setrolId(0);
  setId(0);
  setState(false);
};

const CreateRoleMenu = (event) => {
  event.preventDefault();
  if (parseInt(rolId) > 0 && parseInt(mnId) > 0) {
    createRoleMenu({
      variables: {
        input: {
          mn_id: parseInt(mnId),
          rm_status: state,
          rol_id: parseInt(rolId)
        },
      },
    })
    .then(() => {
      showMessage("Acceso creado correctamente", "success");
    })
    .catch(() => {
      showMessage("Error al crear el acceso", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar sin seleccionar", "warning");
  }
};

const UpdateRoleMenu = (event) => {
  event.preventDefault();
  if (parseInt(rolId) > 0 && parseInt(mnId) > 0) {
    updateRoleMenu({
      variables: {
        input: {
          rm_id: parseInt(id),
          mn_id: parseInt(mnId),
          rm_status: state,
          rol_id: parseInt(rolId)
        },
      },
    })
    .then(() => {
      showMessage("Acceso actualizado correctamente", "success");
    })
    .catch(() => {
      showMessage("Error al actualizar el acceso", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar sin seleccionar", "warning");
  }
};
```

```

    },
  })
  .then(() => {
    showMessage("Acceso actualizado correctamente", "success");
  })
  .catch((err) => {
    console.log(err);
    showMessage("Error al actualizar el acceso", "warning");
  });
document.getElementById("closeBtn").click();
} else {
  showMessage("No se pueden dejar sin seleccionar", "warning");
}
};

const changeState = () => {
  if (state) {
    setState(false);
  } else {
    setState(true);
  }
};

return (
  <div
    className="modal fade"
    id="RoleMenuModal"
    tabIndex={-1}
    aria-labelledby="exampleModallLabel"
    aria-hidden="true"
  >
    <div className="modal-dialog">
      <div className="modal-content">
        <div className="modal-header">
          <h1 className="modal-title fs-5" id="UserModallLabel">
            Accesos
          </h1>
          <button
            type="button"
            className="btn-close"
            data-bs-dismiss="modal"
            aria-label="Close"
            id="closeBtn"
            onClick={CleanForm}
          />
        </div>
        <div className="modal-body">
          <form>
            <div style={{ textAlign: "left" }}>
              {id > 0 ? (
                <div className="mb-3">

```

```

        <strong>Id: {id}</strong>
      </div>
    ) : (
      <></>
    )}

    <strong>Rol</strong> <br></br>
    <div className="mb-3">
      <select
        style={{ width: "5cm", height: "1cm" }}
        value={rolId}
        onChange={(e) => setrolId(e.target.value)}
      >
        <option>Seleccionar</option>
        {rolelist.data?.roles.map((item) => {
          return (
            <option key={item.rol_id} value={item.rol_id}>
              {item.rol_description}
            </option>
          );
        })}
      </select>
    </div>
    <br></br>

    <strong>Menu</strong> <br></br>
    <div className="mb-3">
      <select
        style={{ width: "5cm", height: "1cm" }}
        value={mnId}
        onChange={(e) => setmnId(e.target.value)}
      >
        <option>Seleccionar</option>
        {menulist.data?.menus.map((item) => {
          return (
            <option key={item.mn_id} value={item.mn_id}>
              {item.mn_name}
            </option>
          );
        })}
      </select>
    </div>
    <br></br>
    <div className="container">
      <div className="row">
        <div className="col-3">
          <strong style={{ marginLeft: "-8px" }}>Estado:</strong>
        </div>
        <div className="col-3" style={{ marginLeft: "-45px" }}>
          Activo
        </div>
      </div>
    </div>
  </pre>

```

```

        <div className="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"
              id="flexSwitchCheckDefault"
              checked={state}
              onChange={changeState}
            />
          </div>
        </div>
        <div className="col-3" style={{ marginLeft: "-45px" }}>
          Inactivo
        </div>
        <div className="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"
              id="flexSwitchCheckDefault"
              checked={!state}
              onChange={changeState}
            />
          </div>
        </div>
      </div>
    </form>
  </div>
  <div className="modal-footer">
    <button
      type="button"
      className="btn btn-danger"
      data-bs-dismiss="modal"
      onClick={CleanForm}
    >
      Cancelar <i className="fa fa-close" aria-hidden="true" />
    </button>
    <button
      type="button"
      className="btn btn-success"
      onClick={id === 0 ? CreateRoleMenu : UpdateRoleMenu}
    >
      Guardar <i className="fa fa-save" aria-hidden="true" />
    </button>
  </div>

```

```

        </button>
      </div>
    </div>
  </div>
</div>
);
};

export default RoleMenuForm;

```

Código Fuente 5.62: Estructura HTML para el componente RoleMenuForm

Creación del componente RolMenuList

El siguiente componente permite listar los menús de los roles, además mediante un formulario se podrá agregar, actualizar o eliminar menús a los roles creados.

Como primer paso hay que realizar las correspondientes importaciones; ver código fuente 5.63.

```

import React, { useState } from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";
import { GET_ROLES } from "../../services/role-service";
import { DELETE_ROLE_MENU, GET_ROLE_MENU } from "../../services/access-service";
import {
  showConfirmMessage,
  showMessage,
} from "../../services/message-service";
import { GET_MENUS } from "../../services/menu-service";
import RoleMenuForm from "../rolmenu-form";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";

```

Código Fuente 5.63: Importación de las dependencias

Declarar las variables y los métodos que se utilizarán dentro de este componente; ver código fuente 5.64.

```

import React, { useState } from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";
import { GET_ROLES } from "../../services/role-service";
import { DELETE_ROLE_MENU, GET_ROLE_MENU } from "../../services/access-service";
import {
  showConfirmMessage,
  showMessage,
} from "../../services/message-service";
import { GET_MENUS } from "../../services/menu-service";
import RoleMenuForm from "../rolmenu-form";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";
const RoleMenuList = () => {

```

```

const [rolemenu, setroleMenu] = useState(null);
const rolelist = useQuery(GET_ROLES);
const menulist = useQuery(GET_MENUS);
// Queries
const roleMenuList = useQuery(GET_ROLE_MENU, {
  pollInterval: 500,
});

// Mutations
const [deleteRoleMenu] = useMutation(DELETE_ROLE_MENU, {
  refetchQueries: [{ query: GET_ROLE_MENU }],
});

const selectRoleMenu = (rolemenu) => {
  setroleMenu(rolemenu);
};

const DeleteRoleMenu = (item) => {
  let id = item.rm_id;
  showConfirmMessage("¿Está seguro de eliminar este acceso?").then((resp) => {
    if (resp.isConfirmed) {
      deleteRoleMenu({
        variables: {
          rmId: parseInt(id.toString()),
        },
      })
        .then(() => {
          showMessage("Acceso eliminado correctamente", "success");
        })
        .catch((err) => {
          console.log(err);
          showMessage("Error al eliminar el acceso", "warning");
        });
    }
  });
};

const actionsButtons = (item) => {
  return (
    <>
    <button
      type="button"
      className="btn btn-warning"
      data-bs-toggle="modal"
      data-bs-target="#RoleMenuModal"
      style={{ marginRight: 10 }}
      onClick={() => {
        selectRoleMenu(item);
      }}
    >
    <i className="fa fa-pencil" aria-hidden="true" />
  )

```

```

        </button>
        <button
          type="button"
          className="btn btn-danger"
          onClick={() => {
            DeleteRoleMenu(item);
          }}
        >
          <i className="fa fa-trash" aria-hidden="true" />
        </button>
        <RoleMenuForm key={item.rm_id} rolemenu={rolemenu}></RoleMenuForm>
      </>
    );
  };

  const roleMenuStatus = (item) => {
    return (
      <Tag
        value={item.rm_status ? "Activo" : "Inactivo"}
        severity={getSeverity(item)}
      ></Tag>
    );
  };

  const userRol = (item) => {
    return (
      <Tag
        key={item.rm_id}
        value={getRolebyId(item)}
        severity={getRolLabel(item)}
      ></Tag>
    );
  };

  const getMenubyId = (item) => {
    let menuname = "";
    for (let index = 0; index < menulist?.data.menus.length; index++) {
      const element = menulist?.data.menus[index];
      if (item.mn_id == element.mn_id) {
        menuname = element.mn_name;
      }
    }
    return <Tag key={item.rm_id} value={menuname} severity="warning"></Tag>;
  };

  const getRolebyId = (item) => {
    let name = "";
    for (let index = 0; index < rolelist?.data.roles.length; index++) {
      const element = rolelist.data.roles[index];
      if (item.rol_id == element.rol_id) {
        name = element.rol_description;
      }
    }
  };

```



```

    }
  }
  return name;
};

const getRollLabel = (item) => {
  switch (item.rol_id) {
    case 1:
      return "secondary";

    case 2:
      return "success";

    default:
      return "danger";
  }
};

const getSeverity = (item) => {
  switch (item.rm_status) {
    case true:
      return "info";

    case false:
      return "danger";

    default:
      return null;
  }
};
};

export default RoleMenuList;

```

Código Fuente 5.64: Declaración de las variables y métodos

Para definir la estructura HTML del formulario dentro del componente, es importante destacar que ésta se encuentra dentro de la palabra clave **return**. En este bloque se inserta el código HTML que React utiliza para renderizar los componentes creados.

A continuación, se muestra el código fuente 5.65 donde se encuentra el código completo del componente **RoleMenuList**, el cual incluye la estructura HTML del formulario.

```

import React, { useState } from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";
import { GET_ROLES } from "../../services/role-service";
import { DELETE_ROLE_MENU, GET_ROLE_MENU } from "../../services/access-service";
import {
  showConfirmMessage,
  showMessage,
} from "../../services/message-service";

```

```
import { GET_MENUS } from "../../services/menu-service";
import RoleMenuForm from "../rolmenu-form";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";
const RoleMenuList = () => {
  const [rolemenu, setroleMenu] = useState(null);
  const rolelist = useQuery(GET_ROLES);
  const menulist = useQuery(GET_MENUS);
  // Queries
  const roleMenuList = useQuery(GET_ROLE_MENU, {
    pollInterval: 500,
  });

  // Mutations
  const [deleteRoleMenu] = useMutation(DELETE_ROLE_MENU, {
    refetchQueries: [{ query: GET_ROLE_MENU }],
  });

  const selectRoleMenu = (rolemenu) => {
    setroleMenu(rolemenu);
  };

  const DeleteRoleMenu = (item) => {
    let id = item.rm_id;
    showConfirmMessage("¿Está seguro de eliminar este acceso?").then((resp) => {
      if (resp.isConfirmed) {
        deleteRoleMenu({
          variables: {
            rmId: parseInt(id.toString()),
          },
        })
          .then(() => {
            showMessage("Acceso eliminado correctamente", "success");
          })
          .catch((err) => {
            console.log(err);
            showMessage("Error al eliminar el acceso", "warning");
          });
      }
    });
  };

  const actionsButtons = (item) => {
    return (
      <>
      <button
        type="button"
        className="btn btn-warning"
        data-bs-toggle="modal"
        data-bs-target="#RoleMenuModal"
      ></button>
    )
  };
}
```

```

        style={{ marginRight: 10 }}
        onClick={() => {
          selectRoleMenu(item);
        }}
      >
      <i className="fa fa-pencil" aria-hidden="true" />
    </button>
    <button
      type="button"
      className="btn btn-danger"
      onClick={() => {
        DeleteRoleMenu(item);
      }}
    >
      <i className="fa fa-trash" aria-hidden="true" />
    </button>
    <RoleMenuForm key={item.rm_id} rolemenu={rolemenu}></RoleMenuForm>
  </>
);
};

const roleMenuStatus = (item) => {
  return (
    <Tag
      value={item.rm_status ? "Activo" : "Inactivo"}
      severity={getSeverity(item)}
    ></Tag>
  );
};

const userRol = (item) => {
  return (
    <Tag
      key={item.rm_id}
      value={getRolebyId(item)}
      severity={getRolLabel(item)}
    ></Tag>
  );
};

const getMenubyId = (item) => {
  let menuname = "";
  for (let index = 0; index < menulist?.data.menus.length; index++) {
    const element = menulist?.data.menus[index];
    if (item.mn_id == element.mn_id) {
      menuname = element.mn_name;
    }
  }
  return <Tag key={item.rm_id} value={menuname} severity="warning"></Tag>;
};

```

```
const getRolebyId = (item) => {
  let name = "";
  for (let index = 0; index < rolelist?.data.roles.length; index++) {
    const element = rolelist.data.roles[index];
    if (item.rol_id == element.rol_id) {
      name = element.rol_description;
    }
  }
  return name;
};

const getRolLabel = (item) => {
  switch (item.rol_id) {
    case 1:
      return "secondary";

    case 2:
      return "success";

    default:
      return "danger";
  }
};

const getSeverity = (item) => {
  switch (item.rm_status) {
    case true:
      return "info";

    case false:
      return "danger";

    default:
      return null;
  }
};

return (
  <div>
    <>
    <NavBar></NavBar>
    <br />
    <div className="container">
      <h5 className="card-title title">Accesos</h5>
      <div className="d-grid gap-2 d-md-flex justify-content-md-end">
        <button
          className="btn btn-primary btn-lg"
          type="button"
          data-bs-toggle="modal"
          data-bs-target="#RoleMenuModal"
        >
```

```

        <i className="fa fa-plus" aria-hidden="true" /> Crear
      </button>
    </div>
    <RoleMenuForm rolemenu={rolemenu}></RoleMenuForm>
    <br />
    <DataTable
      value={roleMenuList.data?.rolemenus}
      showGridlines
      stripedRows
      paginator
      rows={5}
      filterLocale="ES"
      rowsPerPageOptions={[5, 10, 25, 50]}
      tableStyle={{ minWidth: "60rem" }}
      emptyMessage="No existen registros."
    >
      <Column header="Acciones" body={actionsButtons}></Column>
      <Column field="mn_id" header="Menú" body={getMenuById}></Column>
      <Column
        field="rol_id"
        sortable
        filter
        filterPlaceholder="Buscar por rol"
        header="Rol"
        filterMatchMode="contains"
        showFilterMenuOptions={false}
        body={userRol}
      ></Column>
      <Column
        sortable
        field="rm_status"
        header="Estado"
        body={roleMenuStatus}
      ></Column>
    </DataTable>
  </div>
</>
</div>
);
};

export default RoleMenuList;

```

Código Fuente 5.65: Estructura HTML del componente RoleMenuList

Creación del componente UserMenu

11. Es necesario construir un componente que permita visualizar los menús del usuario registrado. Para lo cual, dentro de la carpeta de **components** en la carpeta **users** se crea un componente llamado **user-menu.js**.
12. Una vez que se ha creado el componente, lo siguiente es realizar algunas importaciones para obtener

acceso a las operaciones de los servicios. Cabe mencionar que aparte de las dependencias para los servicios se está utilizando dependencias para Apollo Client y para el manejo de los hooks de React; ver código fuente 5.66.

```
import React, { useEffect } from "react";
import { useState } from "react";
import { useQuery } from "@apollo/client";
import {
  GET_USER_MENU,
} from "../../services/access-service";
```

Código Fuente 5.66: Importación de las dependencias

13. Lo siguiente es crear la función que permitirá exportar el componente, como se puede observar el código fuente 5.67 muestra el componente que recibe como parámetro el usuario del cual se obtendrá su rol para mostrar los menús a los que tiene acceso.

```
const UserMenu = ({ user }) => {};

export default UserMenu;
```

Código Fuente 5.67: Función del componente UserMenu

14. A continuación, se crea una variable para obtener y asignar el valor del rol del usuario para eso es necesario utilizar el **hook useEffect** que permitirá modificar el valor cada vez que el componente se renderize; ver código fuente 5.68.

```
const UserMenu = ({ user }) => {};
const [id, setId] = useState(0);

useEffect(() => {
  if (user !== null) {
    setId(user.rol_id);
  }
}, [user]);

export default UserMenu;
```

Código Fuente 5.68: Función del componente UserMenu

15. Mediante el método **useQuery** se puede obtener todos los menús a los que el usuario ingresado tiene acceso tomando cuenta que esta consulta está haciendo un filtro por el tipo de rol; ver código fuente 5.69.

```
const UserMenu = ({ user }) => {};
const [id, setId] = useState(0);

useEffect(() => {
  if (user !== null) {
    setId(user.rol_id);
  }
}, [user]);

const menulist = useQuery(GET_USER_MENU, {
```

```

    variables : {
      rolId: id == 0 ? 1: id
    }
  });

```

```
export default UserMenu;
```

Código Fuente 5.69: Método useQuery para obtener los menús del usuario

16. Ahora hay que crear el componente para que se visualice la información dentro del bloque return. Se agrega el código fuente 5.70 para listar los menús de los usuarios. Se debe utilizar un elemento `ul` de HTML para listar la información.

```

import React, { useEffect } from "react";
import { useState } from "react";
import { useQuery } from "@apollo/client";
import {
  GET_USER_MENU,
} from "../../services/access-service";

const UserMenu = ({ user }) => {
  const [id, setId] = useState(0);

  useEffect(() => {
    if (user !== null) {
      setId(user.rol_id);
    }
  }, [user]);

  const menulist = useQuery(GET_USER_MENU, {
    variables : {
      rolId: id == 0 ? 1: id
    }
  });

  return (
    <div
      className="modal fade"
      id="UserMenuModal"
      tabIndex={-1}
      aria-labelledby="exampleModalLabel"
      aria-hidden="true"
    >
      <div className="modal-dialog">
        <div className="modal-content">
          <div className="modal-header">
            <h1 className="modal-title fs-5" id="UserModalLabel">
              Accesos
            </h1>
            <button
              type="button"

```

```

        className="btn-close"
        data-bs-dismiss="modal"
        aria-label="Close"
      />
    </div>
    <div className="modal-body">
      <form>
        <div style={{ textAlign: "left" }}>

          <div className="mb-3">
            <ul>
              {menulist.data?.rolemenus[0].menus.map((item) => {
                return (
                  <li key={item.mn_id}>
                    {'Menu ' + item.mn_name}
                  </li>
                );
              })}
            </ul>

          </div>

        </div>
      </form>
    </div>
    <div className="modal-footer">
      <button
        type="button"
        className="btn btn-danger"
        data-bs-dismiss="modal"
      >
        Cancelar
      </button>

    </div>
  </div>
</div>
</div>
);
};

export default UserMenu;

```

Código Fuente 5.70: Componente para visualizar los menús del usuario

Creación del componente UserForm

17. Primero se debe realizar las siguientes importaciones; ver código fuente 5.71.

```

import React, { useEffect } from "react";
import { useState } from "react";
import { useMutation, useQuery } from "@apollo/client";

```



```
import {
  GET_USERS,
  CREATE_USER,
  UPDATE_USER,
} from "../../services/user-service";
import { GET_ROLES } from "../../services/role-service";

import { showMessage } from "../../services/message-service";
import { Password } from "primereact/password";
```

Código Fuente 5.71: Importaciones para el componente UserForm

18. Luego es necesario crear la función que permitirá exportar el componente. Es necesario pasar como parámetro el usuario, debido a que se va a utilizar este formulario para realizar la actualización de la información; ver código fuente 5.72.

```
const UserForm = ({ user }) => {};

export default UserForm;
```

Código Fuente 5.72: Función del componente UserForm

19. A continuación, es necesario crear las variables y métodos asociados a las mutaciones que permitirán ingresar y actualizar la información; ver código fuente 5.73.

```
const UserForm = ({ user }) => {};
const rolelist = useQuery(GET_ROLES);
const [name, setName] = useState("");
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
const [state, setState] = useState(false);
const [rolId, setrolId] = useState(0);
const [id, setId] = useState(0);

useEffect(() => {
  if (user !== null) {
    setName(user.usr_name);
    setEmail(user.usr_email);
    setPassword(user.usr_password);
    setId(user.usr_id);
    setState(user.usr_status);
    setrolId(user.rol_id);
  }
}, [user]);

// Mutations
const [createUser] = useMutation(CREATE_USER, {
  refetchQueries: [{ query: GET_USERS }],
});

const [updateUser] = useMutation(UPDATE_USER, {
  refetchQueries: [{ query: GET_USERS }],
});
```

```

const CleanForm = () => {
  setName("");
  setEmail("");
  setrolId(0);
  setId(0);
  setPassword("");
  setState(false);
};

const CreateUser = (event) => {
  event.preventDefault();
  if (name.length > 0 && email.length > 0) {
    createUser({
      variables: {
        user: {
          usr_name: name,
          usr_email: email,
          usr_status: state,
          usr_password: password,
          rol_id: parseInt(rolId),
          user_token: "",
        },
      },
    })
    .then(() => {
      showMessage("Usuario creado correctamente", "success");
    })
    .catch(() => {
      showMessage("Error al crear el usuario", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar los campos en blanco", "warning");
  }
};

const UpdateUser = (event) => {
  event.preventDefault();
  if (name.length > 0 && email.length > 0) {
    updateUser({
      variables: {
        user: {
          usr_id: id,
          usr_name: name,
          usr_email: email,
          usr_status: state,
          rol_id: parseInt(rolId),
          user_token: "",
        },
      },
    })
  },
};

```

```

    })
    .then(() => {
      showMessage("Usuario actualizado correctamente", "success");
    })
    .catch((err) => {
      console.log(err);
      showMessage("Error al actualizar el usuario", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar los campos en blanco", "warning");
  }
};

const changeState = () => {
  if (state) {
    setState(false);
  } else {
    setState(true);
  }
};
export default UserForm;

```

Código Fuente 5.73: Función del componente UserForm

20. Una vez establecidos los métodos, se crea la estructura del formulario como se observa en el código fuente 5.74.

```

import React, { useEffect } from "react";
import { useState } from "react";
import { useMutation, useQuery } from "@apollo/client";
import {
  GET_USERS,
  CREATE_USER,
  UPDATE_USER,
} from "../services/user-service";
import { GET_ROLES } from "../services/role-service";

import { showMessage } from "../services/message-service";
import { Password } from "primereact/password";

const UserForm = ({ user }) => {
  const rolelist = useQuery(GET_ROLES);
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [state, setState] = useState(false);
  const [rolId, setrolId] = useState(0);
  const [id, setId] = useState(0);

  useEffect(() => {

```

```

    if (user !== null) {
      setName(user.usr_name);
      setEmail(user.usr_email);
      setPassword(user.usr_password);
      setId(user.usr_id);
      setState(user.usr_status);
      setrolId(user.rol_id);
    }
  }, [user]);

// Mutations
const [createUser] = useMutation(CREATE_USER, {
  refetchQueries: [{ query: GET_USERS }],
});

const [updateUser] = useMutation(UPDATE_USER, {
  refetchQueries: [{ query: GET_USERS }],
});

const CleanForm = () => {
  setName("");
  setEmail("");
  setrolId(0);
  setId(0);
  setPassword("");
  setState(false);
};

const CreateUser = (event) => {
  event.preventDefault();
  if (name.length > 0 && email.length > 0) {
    createUser({
      variables: {
        user: {
          usr_name: name,
          usr_email: email,
          usr_status: state,
          usr_password: password,
          rol_id: parseInt(rolId),
          user_token: "",
        },
      },
    })
      .then(() => {
        showMessage("Usuario creado correctamente", "success");
      })
      .catch(() => {
        showMessage("Error al crear el usuario", "warning");
      });
    document.getElementById("closeBtn").click();
  } else {

```

```

    showMessage("No se pueden dejar los campos en blanco", "warning");
  }
};

const UpdateUser = (event) => {
  event.preventDefault();
  if (name.length > 0 && email.length > 0) {
    updateUser({
      variables: {
        user: {
          usr_id: id,
          usr_name: name,
          usr_email: email,
          usr_status: state,
          rol_id: parseInt(rolId),
          user_token: "",
        },
      },
    })
    .then(() => {
      showMessage("Usuario actualizado correctamente", "success");
    })
    .catch((err) => {
      console.log(err);
      showMessage("Error al actualizar el usuario", "warning");
    });
    document.getElementById("closeBtn").click();
  } else {
    showMessage("No se pueden dejar los campos en blanco", "warning");
  }
};

const changeState = () => {
  if (state) {
    setState(false);
  } else {
    setState(true);
  }
};

return (
  <div
    className="modal fade"
    id="UserModal"
    tabIndex={-1}
    aria-labelledby="exampleModalLabel"
    aria-hidden="true"
  >
    <div className="modal-dialog">
      <div className="modal-content">
        <div className="modal-header">

```

```

<h1 className="modal-title fs-5" id="UserModalLabel">
  Usuarios
</h1>
<button
  type="button"
  className="btn-close"
  data-bs-dismiss="modal"
  aria-label="Close"
  id="closeBtn"
  onClick={CleanForm}
/>
</div>
<div className="modal-body">
  <form>
    <div style={{ textAlign: "left" }}>
      {id > 0 ? (
        <div className="mb-3">
          <strong>Id: {id}</strong>
        </div>
      ) : (
        <></>
      )}
      <strong>Nombre</strong> <br></br>
      <div className="mb-3">
        <input
          type="text"
          className="form-control"
          placeholder="Nombre"
          value={name}
          onChange={({ target }) => setName(target.value)}
        />
      </div>
      <strong>Correo</strong> <br></br>
      <div className="mb-3">
        <input
          type="text"
          className="form-control"
          placeholder="Correo"
          value={email}
          onChange={({ target }) => setEmail(target.value)}
        />
      </div>
      {id == 0 && (
        <>
          <strong>Clave</strong>
          <br></br>
          <div className="mb-3">
            <Password
              value={password}
              onChange={(e) => setPassword(e.target.value)}
              feedback={true}
            />
          </div>
        </>
      )}
    </div>
  </form>
</div>

```

```

        toggleMask={true}
        inputClassName="form-control"
        weakLabel="Debil"
        mediumLabel="Media"
        strongLabel="Fuerte"
        panelClassName="password-panel"
        className="password-input"
      />
    </div>
  </>
)}
<strong>Rol</strong> <br></br>
<div className="mb-3">
  <select
    style={{ width: "5cm", height: "1cm" }}
    value={rolId}
    onChange={(e) => setrolId(e.target.value)}
  >
    <option>Seleccionar</option>
    {rolelist.data?.roles.map((item) => {
      return (
        <option key={item.rol_id} value={item.rol_id}>
          {item.rol_description}
        </option>
      );
    })}
  </select>
</div>
<br></br>
<div className="container">
  <div className="row">
    <div className="col-3">
      <strong style={{ marginLeft: "-8px" }}>Estado:</strong>
    </div>
    <div className="col-3" style={{ marginLeft: "-45px" }}>
      Activo
    </div>
    <div className="col-1">
      <div
        className="form-check"
        style={{ marginLeft: "-45px" }}
      >
        <input
          className="form-check-input"
          type="checkbox"
          id="flexSwitchCheckDefault"
          checked={state}
          onChange={changeState}
        />
      </div>
    </div>
  </div>

```

```

        <div className="col-3" style={{ marginLeft: "-45px" }}>
          Inactivo
        </div>
        <div className="col-1">
          <div
            className="form-check"
            style={{ marginLeft: "-45px" }}
          >
            <input
              className="form-check-input"
              type="checkbox"
              id="flexSwitchCheckDefault"
              checked={!state}
              onChange={changeState}
            />
          </div>
        </div>
      </div>
    </div>
    <div className="modal-footer">
      <button
        type="button"
        className="btn btn-danger"
        data-bs-dismiss="modal"
        onClick={CleanForm}
      >
        Cancelar <i className="fa fa-close" aria-hidden="true" />
      </button>
      <button
        type="button"
        className="btn btn-success"
        onClick={id === 0 ? CreateUser : UpdateUser}
      >
        Guardar <i className="fa fa-save" aria-hidden="true" />
      </button>
    </div>
  </div>
</div>
);
};

export default UserForm;

```

Código Fuente 5.74: Componente UserForm

Creación del componente UserList

21. Luego se necesita crear un componente para listar la información de los usuarios. Primero se realizan

las importaciones necesarias como se observa en el código fuente 5.75.

```
import React, { useState , useRef} from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";
import { GET_USERS, DELETE_USER } from "../../services/user-service";
import {
  showConfirmMessage,
  showMessage,
} from "../../services/message-service";
import { GET_ROLES } from "../../services/role-service";
```

Código Fuente 5.75: Importación de las dependencias

22. Anteriormente solo se importaron las dependencias que permiten interactuar con los servicios y los **hooks de React**. En esta ocasión se utilizará un componente de primereact que permitirá utilizar tablas con un mejor estilo. Para realizar esto es necesario importar las siguientes dependencias; ver código fuente 5.76.

```
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";
```

Código Fuente 5.76: Importación de las dependencias

23. Es necesario importar los componentes para el formulario y el menú de usuario; ver código fuente 5.77.

```
import UserMenu from "../user-menu";
import UserForm from "../user-form";
```

Código Fuente 5.77: Importación de los componentes

24. Posteriormente hay que crear una función que permita exportar el componente para visualizar la información de los usuarios como se aprecia en el código fuente 5.78.

```
const UserList = () => {};
```

Código Fuente 5.78: Función del componente UserList

25. Luego se debe crear las variables y los métodos que se utilizarán para eliminar el usuario seleccionado. Cabe mencionar que también se crearon componentes como **actionButtons** que permite visualizar los botones actualizar y eliminar los datos del usuario. Además se crearon los componentes como **userStatus** que permite observar el estado del usuario el cual puede ser activo o inactivo; ver código fuente 5.79.

```
const UserList = () => {
  const [user, setUser] = useState(null);
  const dt = useRef(null);
  // Queries
  const userlist = useQuery(GET_USERS, {
    pollInterval: 500,
  });
  const rolelist = useQuery(GET_ROLES);
```

```
// Mutations
const [deleteUser] = useMutation(DELETE_USER, {
  refetchQueries: [{ query: GET_USERS }],
});

const selectUser = (user) => {
  setUser(user);
};

const DeleteUser = (item) => {
  let id = item.usr_id;
  console.log("Delete ", id);
  showConfirmMessage("Esta seguro de eliminar este Usuario?").then(
    (resp) => {
      if (resp.isConfirmed) {
        deleteUser({
          variables: {
            deleteUserId: id.toString(),
          },
        })
          .then(() => {
            showMessage("Usuario eliminado correctamente", "success");
          })
          .catch((err) => {
            showMessage("Error al eliminar el usuario", "warning");
          });
      }
    }
  );
};

const actionsButtons = (item) => {
  return (
    <>
    <button
      type="button"
      className="btn btn-info"
      data-bs-toggle="modal"
      data-bs-target="#UserMenuModal"
      style={{ marginRight: 10 }}
      onClick={() => {
        selectUser(item);
      }}
    >
    <i className="fa fa-eye" aria-hidden="true" />
    </button>
    <button
      type="button"
      className="btn btn-warning"
      data-bs-toggle="modal"
      data-bs-target="#UserModal"
    >
```

```

        style={{ marginRight: 10 }}
        onClick={() => {
          selectUser(item);
        }}
      >
      <i className="fa fa-pencil" aria-hidden="true" />
    </button>
    <button
      type="button"
      className="btn btn-danger"
      onClick={() => {
        DeleteUser(item);
      }}
    >
      <i className="fa fa-trash" aria-hidden="true" />
    </button>
    <UserMenu user={user}></UserMenu>
    <UserForm user={user}></UserForm>
  </>
);
};

const userStatus = (item) => {
  return (
    <Tag
      value={item.usr_status ? "Activo" : "Inactivo"}
      severity={getSeverity(item)}
    ></Tag>
  );
};

const userRol = (item) => {
  return (
    <Tag
      key={item.rm_id}
      value={getRolebyId(item)}
      severity={getRolLabel(item)}
    ></Tag>
  );
};

const getRolebyId = (item) => {
  let name = "";
  for (let index = 0; index < rolelist?.data.roles.length; index++) {
    const element = rolelist.data.roles[index];
    if (item.rol_id == element.rol_id) {
      name = element.rol_description;
    }
  }
  return name;
};

```

```
const getRollLabel = (item) => {
  switch (item.rol_id) {
    case 1:
      return "secondary";

    case 2:
      return "success";

    default:
      return "warning";
  }
};

const getSeverity = (item) => {
  switch (item.usr_status) {
    case true:
      return "success";

    case false:
      return "danger";

    default:
      return null;
  }
};
};
```

Código Fuente 5.79: Función del componente UserList

26. Finalmente, es importante crear la estructura del componente para listar la información de los usuarios. Es necesario destacar que en los componentes para listar información de las pizzas e ingredientes se seguirá un proceso similar para modificar el componente que se utilizó para las tablas; ver código fuente 5.80.

```
import React, { useState , useRef} from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";

import { GET_USERS, DELETE_USER } from "../services/user-service";
import {
  showConfirmMessage,
  showMessage,
} from "../services/message-service";
import UserForm from "../user-form";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";
import UserMenu from "../user-menu";
import { GET_ROLES } from "../services/role-service";
const UserList = () => {
```

```

const [user, setUser] = useState(null);
const dt = useRef(null);
// Queries
const userlist = useQuery(GET_USERS, {
  pollInterval: 500,
});
const rolelist = useQuery(GET_ROLES);

// Mutations
const [deleteUser] = useMutation(DELETE_USER, {
  refetchQueries: [{ query: GET_USERS }],
});

const selectUser = (user) => {
  setUser(user);
};

const DeleteUser = (item) => {
  let id = item.usr_id;
  console.log("Delete ", id);
  showConfirmMessage("¿Está seguro de eliminar este Usuario?").then(
    (resp) => {
      if (resp.isConfirmed) {
        deleteUser({
          variables: {
            deleteUserId: id.toString(),
          },
        })
          .then(() => {
            showMessage("Usuario eliminado correctamente", "success");
          })
          .catch((err) => {
            showMessage("Error al eliminar el usuario", "warning");
          });
      }
    }
  );
};

const actionsButtons = (item) => {
  return (
    <>
    <button
      type="button"
      className="btn btn-info"
      data-bs-toggle="modal"
      data-bs-target="#UserMenuModal"
      style={{ marginRight: 10 }}
      onClick={() => {
        selectUser(item);
      }}
    >
  
```

```

    >
    <i className="fa fa-eye" aria-hidden="true" />
  </button>
  <button
    type="button"
    className="btn btn-warning"
    data-bs-toggle="modal"
    data-bs-target="#UserModal"
    style={{ marginRight: 10 }}
    onClick={() => {
      selectUser(item);
    }}
  >
    <i className="fa fa-pencil" aria-hidden="true" />
  </button>
  <button
    type="button"
    className="btn btn-danger"
    onClick={() => {
      DeleteUser(item);
    }}
  >
    <i className="fa fa-trash" aria-hidden="true" />
  </button>
  <UserMenu user={user}></UserMenu>
  <UserForm user={user}></UserForm>
</>
);
};

const userStatus = (item) => {
  return (
    <Tag
      value={item.usr_status ? "Activo" : "Inactivo"}
      severity={getSeverity(item)}
    ></Tag>
  );
};

const userRol = (item) => {
  return (
    <Tag
      key={item.rm_id}
      value={getRolebyId(item)}
      severity={getRolLabel(item)}
    ></Tag>
  );
};

const getRolebyId = (item) => {
  let name = "";

```

```

    for (let index = 0; index < rolelist?.data.roles.length; index++) {
      const element = rolelist.data.roles[index];
      if (item.rol_id == element.rol_id) {
        name = element.rol_description;
      }
    }
    return name;
  };

  const getRolLabel = (item) => {
    switch (item.rol_id) {
      case 1:
        return "secondary";

      case 2:
        return "success";

      default:
        return "warning";
    }
  };

  const getSeverity = (item) => {
    switch (item.usr_status) {
      case true:
        return "success";

      case false:
        return "danger";

      default:
        return null;
    }
  };

  return (
    <div>
      <>
        <NavBar></NavBar>
        <br />
        <div className="container">
          <h5 className="card-title title">Usuarios</h5>
          <div className="d-grid gap-2 d-md-flex justify-content-md-end">
            <button
              className="btn btn-primary btn-lg"
              type="button"
              data-bs-toggle="modal"
              data-bs-target="#UserModal"
            >
              <i className="fa fa-plus" aria-hidden="true" /> Crear

```

```

        </button>
      </div>
      <UserForm user={user}></UserForm>
      <br />
      <DataTable
        value={userlist.data?.users}
        ref={dt}
        showGridlines
        stripedRows
        paginator
        rows={5}
        filterLocale="ES"
        rowsPerPageOptions={[5, 10, 25, 50]}
        tableStyle={{ minWidth: "60rem" }}
        emptyMessage="No existen registros."
      >
        <Column header="Acciones" body={actionsButtons}></Column>
        <Column
          field="usr_name"
          header="Nombre"
        ></Column>
        <Column
          field="usr_email"
          header="Correo"
        ></Column>
        <Column
          sortable
          field="rol_id"
          header="Rol"
          body={userRol}
        ></Column>
        <Column
          sortable
          field="usr_status"
          header="Estado"
          body={userStatus}
        ></Column>

      </DataTable>
    </div>
  </>
</div>
);
};

export default UserList;

```

Código Fuente 5.80: Componente UserList

Modificación de los componentes para listar información

Como se mencionó anteriormente se utilizaron componentes nuevos para listar la información de los listados.

Modificación del componente **PizzaList**

A continuación se observan los cambios realizados al componente **PizzaList** los cuales se aprecian en el código fuente 5.81.

```
import React, { useState, useRef } from "react";
import { useQuery, useMutation } from "@apollo/client";
import NavBar from "../shared/navbar";

import { GET_PIZZAS, DELETE_PIZZA } from "../../services/pizza-service";
import {
  showConfirmMessage,
  showMessage,
} from "../../services/message-service";
import PizzaForm from "../pizza-form";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";
import ReportHeader from "../shared/report-header";
import { formatPizzaFields, exportReportPdf } from
  from
  "../../services/export-file-service";
const PizzaList = () => {
  const [pizza, setPizza] = useState(null);
  const dt = useRef(null);
  // Queries
  const pizzalist = useQuery(GET_PIZZAS, {
    pollInterval: 500,
  });

  // Mutations
  const [deletePizza] = useMutation(DELETE_PIZZA, {
    refetchQueries: [{ query: GET_PIZZAS }],
  });

  const selectPizza = (pizza) => {
    setPizza(pizza);
  };
  const DeletePizza = (item) => {
    let id = item.piz_id;
    showConfirmMessage("¿Está seguro de eliminar esta Pizza?").then((resp) => {
      if (resp.isConfirmed) {
        deletePizza({
          variables: {
            deletePizzaId: id.toString(),
          },
        },
      )
      .then(() => {
        showMessage("Pizza eliminada correctamente", "success");
      });
    });
  };
}
```

```

    })
    .catch((err) => {
      showMessage("Error al eliminar la pizza", "warning");
    });
  }
});
};

const actionsButtons = (item) => {
  return (
    <>
    <button
      type="button"
      className="btn btn-warning"
      data-bs-toggle="modal"
      data-bs-target="#PizzaModal"
      style={{ marginRight: 10 }}
      onClick={() => {
        selectPizza(item);
      }}
    >
    <i className="fa fa-pencil" aria-hidden="true" />
    </button>

    <button
      type="button"
      className="btn btn-danger"
      onClick={() => {
        DeletePizza(item);
      }}
    >
    <i className="fa fa-trash" aria-hidden="true" />
    </button>
    <PizzaForm pizza={pizza}></PizzaForm>
  </>
  );
};

const statusPizza = (item) => {
  return (
    <Tag
      value={item.piz_state ? "Activa" : "Inactiva"}
      severity={getSeverity(item)}
    ></Tag>
  );
};

const getSeverity = (item) => {
  switch (item.piz_state) {
    case true:
      return "success";
  }
};

```

```

    case false:
      return "warning";

    default:
      return null;
  }
};

const IngredientsPizza = (item) => {
  return (
    <ul>
      {item.ingredients.map((ingredient, index) => (
        <li key={index}>{ingredient.ing_name}</li>
      ))}
    </ul>
  );
};

return (
  <div>
    <>
      <NavBar></NavBar>
      <br />
      <div className="container">
        <h5 className="card-title title">Pizzas</h5>
        <div className="d-grid gap-2 d-md-flex justify-content-md-end">
          <button
            className="btn btn-primary btn-lg"
            type="button"
            data-bs-toggle="modal"
            data-bs-target="#PizzaModal"
          >
            <i className="fa fa-plus" aria-hidden="true" /> Crear
          </button>
        </div>
        <PizzaForm pizza={pizza}></PizzaForm>
        <br />
        <DataTable
          ref={dt}
          value={pizzalist.data?.pizzas}
          showGridlines
          stripedRows
          paginator
          rows={5}
          filterLocale="ES"
          rowsPerPageOptions={[5, 10, 25, 50]}
          tableStyle={{ minWidth: "60rem" }}
          emptyMessage="No existen registros."
        >
          <Column header="Acciones" body={actionsButtons}></Column>

```

```

        <Column
          field="piz_name"
          sortable
          filter
          filterPlaceholder="Buscar por nombre"
          header="Nombre"
          filterMatchMode="contains"
          showFilterMenuOptions={false}
        ></Column>
        <Column
          field="piz_origin"
          sortable
          filter
          filterPlaceholder="Buscar por origen"
          filterMatchMode="contains"
          showFilterMenuOptions={false}
          header="Origen"
        ></Column>
        <Column
          sortable
          field="piz_state"
          header="Estado"
          body={statusPizza}
        ></Column>
        <Column
          field="ingredients"
          header="Ingredientes"
          body={IngredientsPizza}
        ></Column>
      </DataTable>
    </div>
  </>
</div>
);
};

export default PizzaList;

```

Código Fuente 5.81: Modificación del componente PizzaList

Modificación del componente IngredientList

A continuación, se observan los cambios realizados al componente **IngredientList** los cuales se aprecian en el código fuente 5.82.

```

import React, { useState , useRef} from "react";
import NavBar from "../shared/navbar";
import { useQuery, useMutation } from "@apollo/client";
import {
  GET_INGREDIENTS,
  DELETE_INGREDIENTS,
} from "../..services/ingredient-service";

```

```
import {
  showMessage,
  showConfirmMessage,
} from "../../services/message-service";
import IngredientForm from "../ingredient-form";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Tag } from "primereact/tag";
import { formatIngredientsFields } from "../../services/export-file-service";
import ReportHeader from "../shared/report-header";
const IngredientList = () => {

  const [ingredient, setIngredient] = useState(null);
  const dt = useRef(null);
  // Queries
  const IngredientsList = useQuery(GET_INGREDIENTS);
  // Mutations
  const [deleteIngredient] = useMutation(DELETE_INGREDIENTS, {
    refetchQueries: [{ query: GET_INGREDIENTS }],
  });

  const selectIngredient = (ingredient) => {
    setIngredient(ingredient);
  };

  const DeleteIngredient = (item) => {
    let id = item.ing_id;
    showConfirmMessage("¿Está seguro de eliminar esta Ingrediente?").then(
      (resp) => {
        if (resp.isConfirmed) {
          deleteIngredient({
            variables: {
              deleteIngredientId: id.toString(),
            },
          })
            .then(() => {
              showMessage("Ingrediente eliminado correctamente", "success");
            })
            .catch((err) => {
              showMessage("Error al eliminar el ingrediente", "warning");
            });
        }
      }
    );
  };

  const actionsButtons = (item) => {
    return (
      <>
```

```

        <button
          type="button"
          className="btn btn-warning"
          data-bs-toggle="modal"
          data-bs-target="#IngredienteModal"
          style={{ marginRight: 10 }}
          onClick={() => {
            selectIngredient(item);
          }}
        >
        <i className="fa fa-pencil" aria-hidden="true" />
      </button>
      <button
        type="button"
        className="btn btn-danger"
        onClick={() => {
          DeleteIngredient(item);
        }}
      >
      <i className="fa fa-trash" aria-hidden="true" />
    </button>
    <IngredientForm ingredient={ingredient}></IngredientForm>
  </>
);
};

const statusIngredient = (item) => {
  return (
    <Tag
      value={item.ing_state ? "Activo" : "Inactivo"}
      severity={getSeverity(item)}
    ></Tag>
  );
};

const getSeverity = (item) => {
  switch (item.ing_state) {
    case true:
      return "success";

    case false:
      return "warning";

    default:
      return null;
  }
};

return (

```

```

<div>
  <>
  <NavBar></NavBar>
  <br />
  <div className="container">
    <h5 className="card-title title">Ingredientes</h5>
    <div className="d-grid gap-2 d-md-flex justify-content-md-end">
      <button
        className="btn btn-primary btn-lg"
        type="button"
        data-bs-toggle="modal"
        data-bs-target="#IngredienteModal"
      >
        <i className="fa fa-plus" aria-hidden="true" /> Crear
      </button>
      <IngredientForm ingredient={ingredient} ></IngredientForm>
    </div>
    <br />
    <DataTable
      value={IngredientsList.data?.ingredients}
      ref={dt}
      showGridlines
      stripedRows
      paginator
      rows={5}
      rowsPerPageOptions={[5, 10, 25, 50]}
      tableStyle={{ minWidth: "60rem" }}
      emptyMessage="No existen registros."
    >
      <Column header="Acciones" body={actionsButtons}></Column>
      <Column
        field="ing_name"
        sortable
        filter
        filterPlaceholder="Buscar por nombre"
        header="Nombre"
        filterMatchMode="contains"
        showFilterMenuOptions={false}

      ></Column>
      <Column
        field="ing_calories"
        sortable
        header="Calorias"
      ></Column>
      <Column
        sortable
        field="ing_state"
        header="Estado"
        body={statusIngredient}
      ></Column>

```

```
        </DataTable>
      </div>
    </>
  </div>
);
};

export default IngredientList;
```

Código Fuente 5.82: Modificación del componente IngredientList

A continuación, la Fig. 5.15 muestra la pantalla principal que permite ver los menús a los que el usuario tiene acceso. La Fig. 5.16 permite visualizar las opciones de los menús del rol administrador.

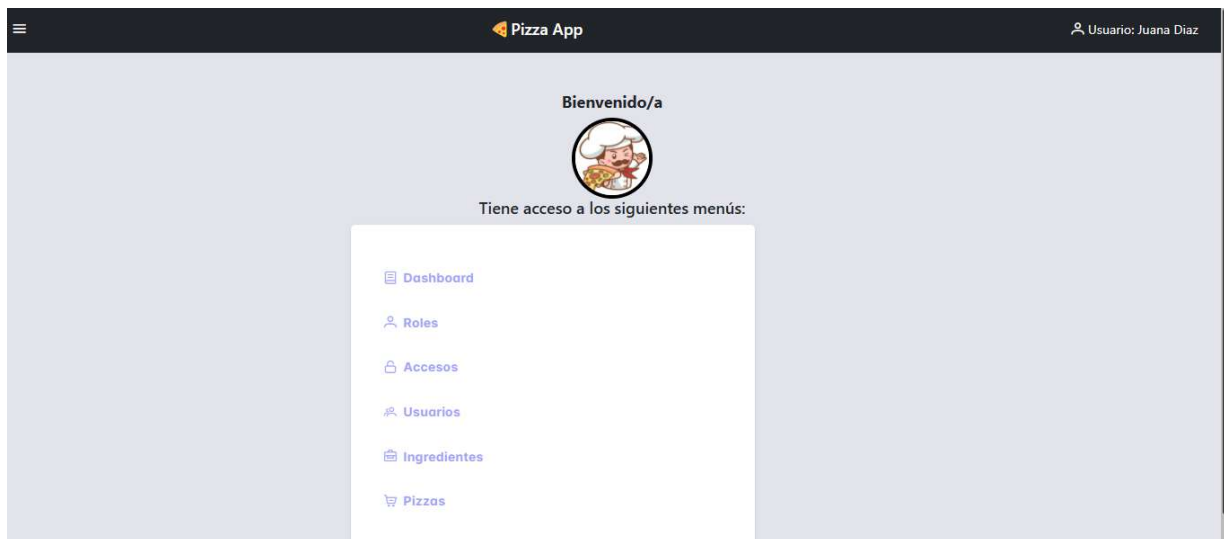


Fig. 5.15. Menús del Usuario

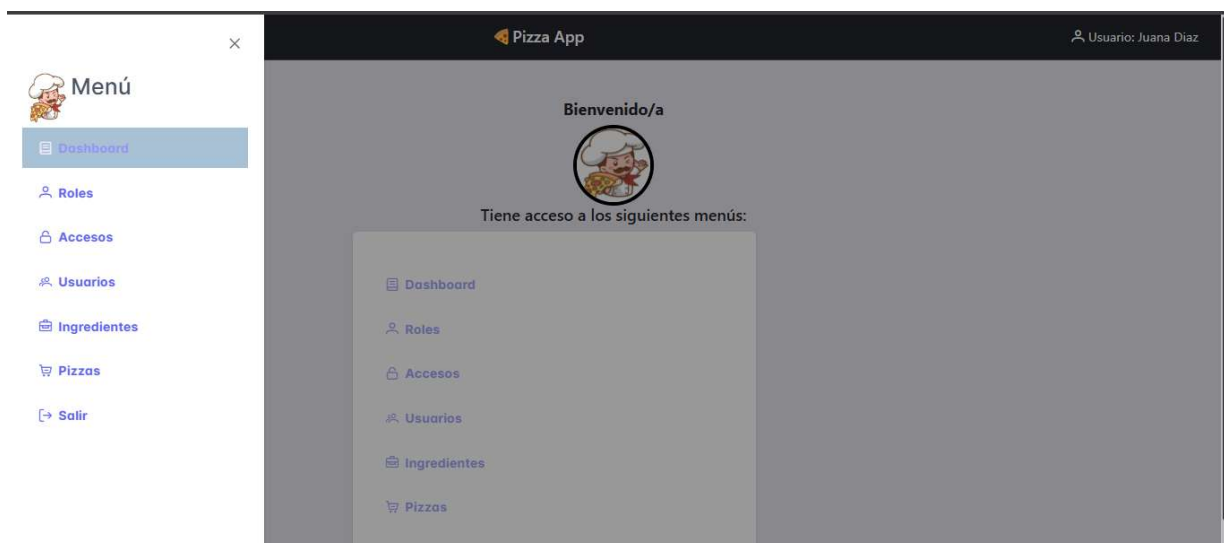


Fig. 5.16. Menús del usuario administrador

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-03.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-03.git>

MANEJO AVANZADO DE TABLAS

Modificación del proyecto frontend creado

Para el manejo avanzado de tablas en **React** se utilizará **PrimeReact** y su componente **DataTable**. Se puede realizar la configuración de paginación, ordenación y filtrado de datos de manera controlada y dinámica, sin necesidad de implementar funcionalidades externas. A continuación se mencionarán los aspectos clave de cada una de estas funciones en **DataTable**.

Paginación

La paginación es una técnica utilizada en el desarrollo web para dividir grandes conjuntos de datos en partes más manejables, mejorando el rendimiento y la experiencia del usuario. Se implementa comúnmente en aplicaciones web con bases de datos grandes, paneles administrativos y dashboards [46] .

Propiedades de Paginación

paginator: Activa la paginación. Sin esta propiedad, todos los datos se mostrarán en una sola página.

rows: Determina el número de filas visibles por página. Este valor puede personalizarse según las necesidades de la aplicación.

rowsPerPageOptions: Ofrece un menú desplegable donde el usuario puede seleccionar cuántas filas mostrar en cada página.

El código fuente 5.83 muestra un ejemplo básico sobre la paginación.

```
<DataTable
  value={data} // Datos mostrados en la tabla
  paginator // Activa la paginación en el DataTable
  rows={10} // Número de filas visibles por página
  rowsPerPageOptions={[5, 10, 20]} // Opciones para cambiar el número de filas por
  página
  onPage={(e) => handlePageChange(e)} // Evento para manejar el cambio de página
/>
```

Código Fuente 5.83: Ejemplo Básico de Paginación

Ordenación

La ordenación permite al usuario organizar los datos en función de los valores de una o varias columnas. **DataTable** proporciona opciones de ordenación tanto simple como múltiple [46] .

Propiedades de Ordenación

sortable: Cuando esta propiedad se coloca en una columna, habilita la ordenación en esa columna específica [46] .

sortMode: Puede configurarse como “single” para permitir que solo una columna se ordene a la vez o “múltiple” para ordenar en múltiples columnas al mismo tiempo [46].

onSort: Evento que se activa cada vez que el usuario aplica una ordenación. Esto es útil para actualizar el estado de ordenación o para realizar una solicitud de datos ordenados desde una API externa [46].

El código fuente 5.84 muestra un ejemplo básico sobre la ordenación.

```
<DataTable value={data} sortMode="multiple" onSort={(e) => handleSort(e)}>
  <Column field="name" header="Nombre" sortable />
  <Column field="age" header="Edad" sortable />
</DataTable>
```

Código Fuente 5.84: Ejemplo Básico de Ordenación

Filtrado

El filtrado permite a los usuarios refinar los datos en función de criterios específicos en cada columna. PrimeReact proporciona un sistema de filtrado avanzado que permite personalizar el modo de coincidencia y el diseño de los filtros [46].

Propiedades de Filtrado

filter: Habilita el filtro en una columna específica. Esto añadirá una entrada de búsqueda en la parte superior de la columna [46].

filterPlaceholder: Define un texto de ayuda que se muestra en el campo de filtro. En este ejemplo, “Buscar por nombre” y “Edad mínima” ayudan al usuario a entender el propósito del filtro [46].

filterMatchMode: Especifica el tipo de coincidencia que se aplicará al filtro, como contains (contiene), startsWith (empieza con), equals (igual a), gte (mayor o igual que), etc. Esto proporciona un filtrado más preciso, especialmente en campos numéricos o de fecha [46].

Modos de Coincidencia del Filtro

Algunos de los modos de coincidencia más comunes son:

contains: Coincide con cualquier parte del contenido en la celda [46].

startsWith: Coincide si el valor de la celda empieza con el texto en el filtro [46].

equals: Coincide si el valor de la celda es exactamente igual al texto en el filtro [46].

gte y lte: Útil para valores numéricos o de fecha, para verificar si un valor es mayor o igual o menor o igual que el filtro [46].

El código fuente 5.85 muestra un ejemplo básico sobre el filtrado.

```
<DataTable value={data} paginator rows={10}>
  <Column field="name" header="Nombre" filter filterPlaceholder
    ="Buscar por nombre" />
  <Column field="age"
    header="Edad" filter filterMatchMode="gte" filterPlaceholder="Edad mínima" />
</DataTable>
```

Código Fuente 5.85: Ejemplo Básico de Filtrado

Una vez que se han establecido las operaciones que se pueden realizar con el componente **datatable**, es necesario continuar con la modificación de los componentes. En es sentido, para listar información y realizar la paginación se utiliza **paginator** y **rowsPerPageOptions=[5, 10, 25, 50]** para establecer el número de filas por páginas, en el caso de la ordenación se utilizará **sortable** que permite ordenar los campos de nuestro **datatable**. Cabe mencionar que hay que colocar el nombre del campo que se desea ordenar en la propiedad **field**.

En el caso del filtro se utiliza la propiedad **filter** para indicar que la columna va a ser filtrada. **filterPlaceholder** permite colocar el nombre del **datatable** y las propiedades **filterMatchMode** y **showFilterMenuOptions** permiten escoger el método de filtrado y las opciones del menú del filtro.

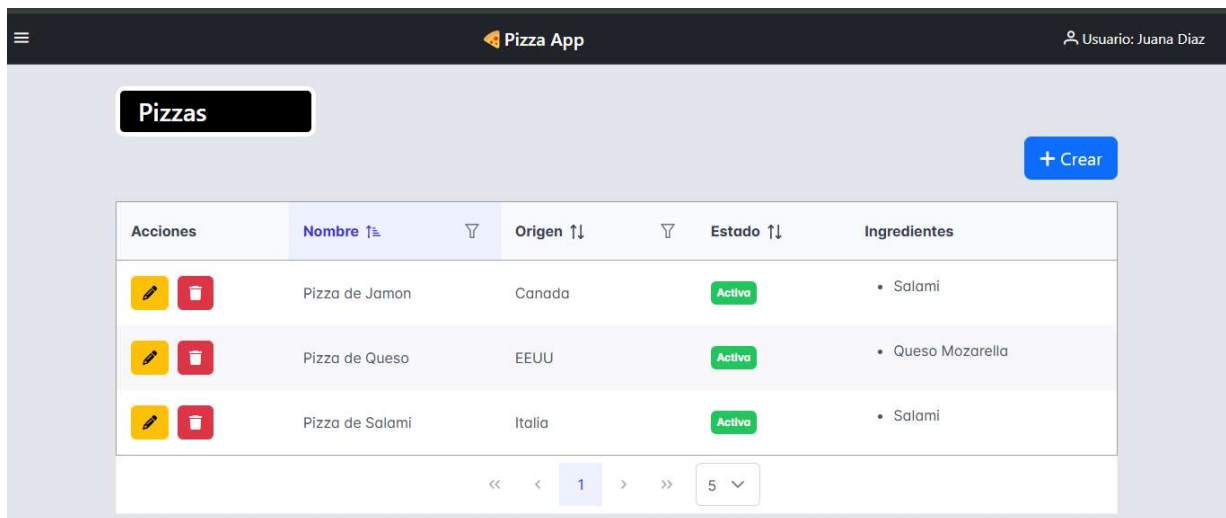
Es importante mencionar que este cambio se debe realizar en cada uno de los componentes en donde se desee realizar estas operaciones con tablas; ver código fuente 5.86.

```
<DataTable
  ref={dt}
  value={pizzalist.data?.pizzas}
  showGridlines
  stripedRows
  paginator
  rows={5}
  filterLocale="ES"
  rowsPerPageOptions={[5, 10, 25, 50]}
  tableStyle={{ minWidth: "60rem" }}
  emptyMessage="No existen registros."
>
  <Column header="Acciones" body={actionsButtons}></Column>
  <Column
    field="piz_name"
    sortable
    filter
    filterPlaceholder="Buscar por nombre"
    header="Nombre"
    filterMatchMode="contains"
    showFilterMenuOptions={false}
  ></Column>
  <Column
    field="piz_origin"
    sortable
    filter
    filterPlaceholder="Buscar por origen"
    filterMatchMode="contains"
    showFilterMenuOptions={false}
    header="Origen"
  ></Column>
  <Column
    sortable
    field="piz_state"
    header="Estado"
    body={statusPizza}
  ></Column>
  <Column
```

```
field="ingredients"  
header="Ingredientes"  
body={IngredientsPizza}  
></Column>  
</DataTable>
```

Código Fuente 5.86: Modificación del componente PizzaList

En la Fig. 5.17 se observa la tabla para listar la información con los respectivos cambios.



Pizza App

Usuario: Juana Diaz

Pizzas

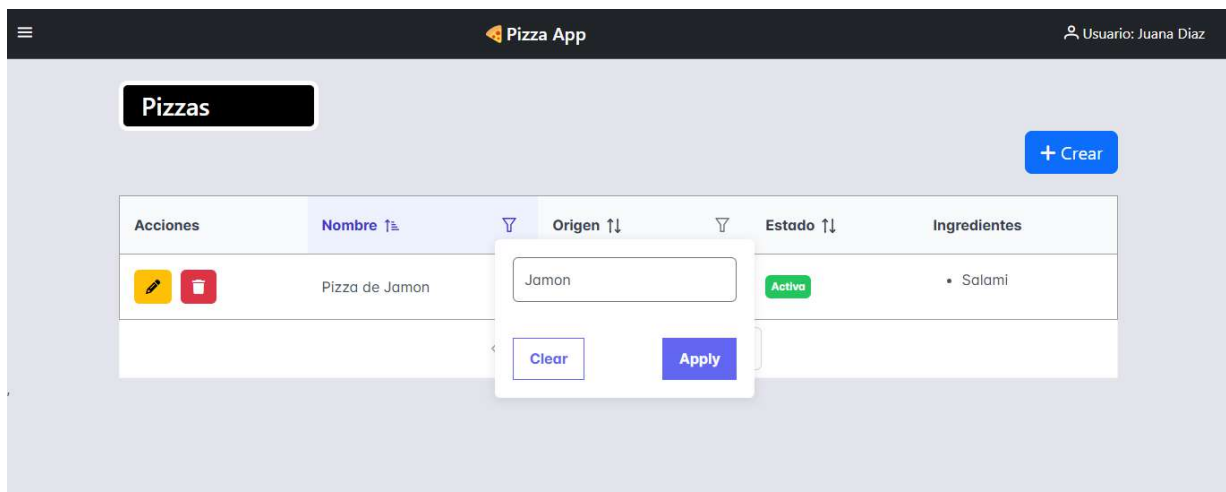
+ Crear

Acciones	Nombre	Origen	Estado	Ingredientes
 	Pizza de Jamon	Canada	Activo	• Salami
 	Pizza de Queso	EEUU	Activo	• Queso Mozzarella
 	Pizza de Salami	Italia	Activo	• Salami

<< < 1 > >> 5

Fig. 5.17. Manejo de Tablas

En la Fig. 5.18 muestra un ejemplo para el filtrado de información de tabla.



Pizza App

Usuario: Juana Diaz

Pizzas

+ Crear

Acciones	Nombre	Origen	Estado	Ingredientes
 	Pizza de Jamon	Jamon	Activo	• Salami

Clear Apply

Fig. 5.18. Manejo de Tablas Filtrado

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-04.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-04.git>

REPORTERÍA

Reportes en PDF

jsPDF⁹ es una librería que proporciona una interfaz fácil de usar para crear y manipular archivos PDF a través de JavaScript [49]. Permite generar documentos PDF con contenido dinámico, tales como texto, imágenes, gráficos, tablas, y más, sin necesidad de interacción con un servidor. Esto significa que los usuarios pueden generar, personalizar y descargar archivos PDF de manera local en sus navegadores.

Reportes en Excel

Para la creación de reportes en Excel utilizaremos la librería `xlsx`¹⁰, es una herramienta de JavaScript diseñada para trabajar con hojas de cálculo de Excel y otros formatos tabulares de manera eficiente [50]. Permite crear, leer, modificar y escribir archivos en formatos como `.xlsx`, `.xlsm`, `.csv`, `.xlsb`, entre otros, tanto en aplicaciones del lado del cliente como en servidores con Node.js. Es ampliamente utilizada en aplicaciones web y de escritorio que requieren manipulación de datos estructurados.

Características principales:

Creación de archivos Excel: Permite generar hojas de cálculo desde cero, creando hojas, celdas, filas y columnas con contenido personalizado.

Lectura de archivos: Facilita la extracción de datos de archivos Excel existentes para procesarlos, analizarlos o visualizarlos. Modificación de archivos: permite editar contenido de hojas de cálculo existentes, como agregar o actualizar valores en celdas específicas.

Conversión de datos: Soporta la transformación de datos entre diferentes formatos, como convertir JSON a Excel, CSV a XLSX, o viceversa.

Compatibilidad multiplataforma: Puede usarse en navegadores, aplicaciones web y entornos de servidor con Node.js.

Soporte avanzado: Incluye funcionalidades como formateo de celdas, fusión de celdas, definición de estilos básicos y más.

Dashboard

Es una herramienta visual que consolida y presenta de manera interactiva información clave y métricas relevantes para el monitoreo, análisis y toma de decisiones en diversas áreas, como negocios, educación o gestión gubernamental. Estos paneles permiten a los usuarios obtener en tiempo real una visión clara de los procesos y resultados, facilitando la identificación de tendencias, patrones y áreas que requieren atención [51].

Se caracteriza por su capacidad de visualizar datos mediante gráficos y tablas, ofreciendo interactividad para explorar información y filtrar detalles según las necesidades del usuario. Además, permite la actualización en tiempo real, asegurando datos precisos para la toma de decisiones. Su diseño es altamente personalizable, adaptándose a distintos roles y necesidades, y puede integrar múltiples fuentes de datos para proporcionar una visión holística. También destaca por incluir indicadores clave de rendimiento (KPI), lo que facilita el monitoreo de objetivos estratégicos y el análisis de tendencias [52].

⁹jsPDF: <https://artsykydj.github.io/jsPDF/docs/jsPDF.html>

¹⁰xlsx: <https://www.npmjs.com/package/xlsx>

La implementación de dashboards ha demostrado ser efectiva en diversos contextos. Por ejemplo, en el ámbito educativo, se ha desarrollado un dashboard para el control y seguimiento académico de estudiantes en educación superior, integrando y analizando información relevante para mejorar la toma de decisiones y reducir la deserción estudiantil [53].

1. Primer paso. Es necesario crear un servicio que va a permitir exportar los reportes en distintos formatos, una vez que creado el servicio con el nombre **export-file-service.js** después hay que importar las dependencias necesarias como se aprecia en el código fuente 5.87.

```
import jsPDF from "jspdf";
import autoTable from "jspdf-autotable";
import { isAvailable } from "../util-service";
import image from "../assets/images/pizza.jpg";
import logo from "../assets/images/pizza-logo.jpg";
import Pizzalogo from "../assets/images/bg_1.png";
```

Código Fuente 5.87: Importación de las dependencias

2. Segundo paso. Se debe crear métodos que permitan formatear los datos de los listados. Es importante mencionar que existe un campo booleano el cual muestra si el registro se encuentra activo o no, por esa razón se debe hacer ese cambio antes de generar el reporte; ver código fuente 5.88.

```
const formatPizzaFields = (list) => {
  console.log(list);
  let formattedList = [];
  let piz_state = "";

  if (list !== undefined) {
    for (let index = 0; index < list.length; index++) {
      const element = list[index];
      if (element.piz_state) {
        piz_state = "Activa";
      } else {
        piz_state = "Inactiva";
      }
      formattedList.push([element.piz_name, element.piz_origin, piz_state]);
    }
  }

  return formattedList;
};

const formatIngredientsFields = (list) => {
  let formattedList = [];
  let state = "";

  if (list !== undefined) {
    for (let index = 0; index < list.length; index++) {
      const element = list[index];
      if (element.ing_state) {
        state = "Activo";
      } else {
        state = "Inactivo";
      }
    }
  }
}
```

```

    }
    formattedList.push(
      [element.ing_name, element.ing_calories,element.pi_portion, state]);
  }
}

return formattedList;
};

const formatUserFields = (list) => {
  let formattedList = [];
  let state = "";
  if (list !== undefined) {
    for (let index = 0; index < list.length; index++) {
      const element = list[index];
      if (element.usr_status) {
        state = "Activo";
      } else {
        state = "Inactivo";
      }
      formattedList.push([element.usr_name, element.usr_email, state]);
    }
  }

  return formattedList;
};

```

Código Fuente 5.88: Métodos para modificar los valores de los listados

3. Tercer paso. Se debe crear el método para exportar el reporte en CSV. Es importante mencionar que no se utilizará ninguna dependencia para generar este tipo de reporte ya que se maneja de forma automática mediante la librería de **primereact**; ver código fuente 5.89.

```

const exportCSV = (dt, selectionOnly) => {
  dt.current.exportCSV({ selectionOnly });
};

```

Código Fuente 5.89: Método para exportar el reporte en CSV

4. Cuarto paso. Crear el método para exportar el reporte en PDF. Para esto se hará uso de la librería **Jspdf**, cabe mencionar que para mayor información se debe acceder a la librería, en este método se observa que existen funciones como **autotable** que permite establecer columnas y filas para colocar el contenido del documento; ver código fuente 5.90.

```

const exportPdf = async (data, name, columns) => {
  const doc = new jsPDF({ filters: ["ASCIIToText"] });
  let date = new Date();
  let fecha = date.toLocaleDateString();

  autoTable(doc, {
    body: [
      [
        {

```

```

        content: name,
        styles: {
          halign: "left",
          fontSize: 9,
          fontStyle: "bold",
          textColor: "#ffffff",
        },
      },
    },
    {
      content: fecha,
      styles: {
        halign: "right",
        fontStyle: "bold",
        fontSize: 9,
        textColor: "#ffffff",
      },
    },
  ],
  theme: "plain",
  styles: {
    fillColor: "#0939B0",
  },
});
autoTable(doc, {
  head: [columns],
  body: data,
  theme: "striped",
});
doc.save(name + ".pdf");
};

const exportReportPdf = async (item, name, columns) => {
  const doc = new jsPDF({ filters: ["ASCIIToText"] });
  let date = new Date();
  let fecha = date.toLocaleDateString();
  doc.drawImage(image, 10, 5, 20, 20);
  doc.drawImage(logo, 180, 5, 20, 20);
  doc.setFont("helvetica", "bold");
  doc.text("Pizza GraphQL", 80, 15);
  doc.setFont("helvetica", "normal");
  doc.setFontSize(8);
  doc.text("Av. República de El Salvador", 80, 20);
  doc.text("099 250 3272", 90, 24);
  doc.text("pizzagraphql@gmail.com", 83, 28);
  doc.setFont("helvetica", "bold");
  doc.text(
    "-----"
    "-----"
    "-----"
    "-----",
  );
};

```



```

    10,
    30
);
doc.setFont("helvetica", "bold");
doc.setFontSize(10);
doc.text("Información", 15, 40);
doc.setFontSize(8);
doc.setFont("helvetica", "normal");
doc.text("Fecha: " + fecha, 15, 45);
doc.text(
    "Hora: " + new Date().getHours() + ":" +
    new Date().getMinutes() +
    ":" +
    new Date().getSeconds(),
    15,
    50
);
doc.text("Nombre: " + item.piz_name, 15, 55);
doc.text("Origen: " + item.piz_origin, 15, 60);
doc.text("Estado: " + isAvailable(item.piz_state), 15, 65);
doc.setFont("helvetica", "bold");
doc.setFontSize(10);
doc.text("Detalle", 15, 75);
doc.addImage(Pizzalogo, 80, 35, 35, 35);
doc.setFontSize(8);
doc.setFont("helvetica", "normal");

autoTable(doc, {
    head: [columns],
    body: formatIngredientsFields(item.ingredients),
    theme: "striped",
    startY: 78,
    halign: "LEFT",
    styles: {
        fontStyle: "normal",
        fontSize: 8,
    },
});

autoTable(doc, {
    body: [["Total de Calorías: " + item.total_calories]],
    theme: "plain",
    styles: {
        fontStyle: "normal",
        fontSize: 8,
    },
});

doc.text("Escanea el código:", 20, 258)
doc.text("Página " + doc.getNumberOfPages(), 170, 290)
doc.addImage(codeQR, 15, 260, 35, 35);

```

```
doc.save(name + ".pdf");
};
```

Código Fuente 5.90: Método para exportar el reporte en PDF

- Quinto paso. Crear el método para exportar el reporte en Excel. Se utilizó la librería `xlsx` ya mencionada anteriormente, mediante el método `saveExcelFile` se puede generar el archivo excel a partir del buffer que se obtiene en el método `exportExcel`; ver código fuente 5.91.

```
const exportExcel = (data, name) => {
  import("xlsx").then((xlsx) => {
    const worksheet = xlsx.utils.json_to_sheet(data);
    const workbook = { Sheets: { data: worksheet }, SheetNames: ["data"] };
    const excelBuffer = xlsx.write(workbook, {
      bookType: "xlsx",
      type: "array",
    });

    saveAsExcelFile(excelBuffer, name);
  });
};

const saveAsExcelFile = (buffer, fileName) => {
  import("file-saver").then((module) => {
    if (module && module.default) {
      let EXCEL_TYPE =
        "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet;charset=UTF-8";
      let EXCEL_EXTENSION = ".xlsx";
      const data = new Blob([buffer], {
        type: EXCEL_TYPE,
      });

      module.default.saveAs(
        data,
        fileName + "_export_" + new Date().getTime() + EXCEL_EXTENSION
      );
    }
  });
};
```

Código Fuente 5.91: Método para exportar el reporte en Excel

- Sexto paso. Para mostrar las distintas opciones y exportar reportes es necesario crear un nuevo componente llamado `reportHeader`. El código fuente 5.92 muestra el siguiente componente, así como también el uso de los servicios que se crearon anteriormente y los distintos botones para exportar la información.

```
import React from "react";
import { Button } from "primereact/button";
import {
  exportCSV,
  exportExcel,
  exportPdf,
} from "../../services/export-file-service";

const ReportHeader = ({ dt, formatdata, data, columns, name }) => {
  return (
    <div className="flex align-items-center justify-content-end gap-2">
      <Button
        type="button"
        icon="pi pi-file"
        tooltip="Exportar a CSV"
        title="CSV"
        rounded
        style={{ borderRadius: "10px", marginRight: "10px" }}
        onClick={() => exportCSV(dt, false)}
      />
      <Button
        type="button"
        icon="pi pi-file-excel"
        title="XLSX"
        tooltip="Exportar a Excel"
        severity="success"
        style={{ borderRadius: "10px", marginRight: "10px" }}
        rounded
        onClick={() => {
          exportExcel(data, name);
        }}
      />
      <Button
        type="button"
        icon="pi pi-file-pdf"
        tooltip="Exportar a PDF"
        severity="warning"
        title="PDF"
        rounded
        style={{ borderRadius: "10px" }}
        onClick={() => {
          exportPdf(formatdata, name, columns);
        }}
      />
    </div>
  );
};

export default ReportHeader;
```

Código Fuente 5.92: Método para exportar el reporte en Excel

Colocar el componente **ReportHeader** a los componentes que sean necesarios, en este caso se tomará como guía al componente **PizzaList**. Cabe mencionar que el proceso descrito a continuación se puede realizar en cada uno de los componentes para visualizar los listados.

7. Séptimo paso. Realizar la importación del componente **ReportHeader**. El código fuente 5.93 muestra como importar el método **formatPizzaFields** para modificar los valores del listado.

```
import ReportHeader from "../shared/report-header";
import { formatPizzaFields, exportReportPdf } from "../../services/
export-file-service";
```

Código Fuente 5.93: Importación de dependencias y servicios

8. Octavo paso. Dentro de la propiedad **header** del componente **PizzaList** se debe llamar al componente **ReportHeader** para visualizar las distintas opciones y exportar los reportes; ver código fuente 5.94.

```
<DataTable
  ref={dt}
  header={
    <ReportHeader
      formatdata={formatPizzaFields(pizzalist.data?.pizzas)}
      data={pizzalist.data?.pizzas}
      dt={dt}
      columns={["Nombre", "Origen", "Estado"]}
      name={"Pizzas"}
    ></ReportHeader>
  }
  value={pizzalist.data?.pizzas}
  showGridlines
  stripedRows
  paginator
  rows={5}
  filterLocale="ES"
  rowsPerPageOptions={[5, 10, 25, 50]}
  tableStyle={{ minWidth: "60rem" }}
  emptyMessage="No existen registros."
>
  <Column header="Acciones" body={actionsButtons}></Column>
  <Column
    field="piz_name"
    sortable
    filter
    filterPlaceholder="Buscar por nombre"
    header="Nombre"
    filterMatchMode="contains"
    showFilterMenuOptions={false}
  ></Column>
  <Column
```

```

        field="piz_origin"
        sortable
        filter
        filterPlaceholder="Buscar por origen"
        filterMatchMode="contains"
        showFilterMenuOptions={false}
        header="Origen"
    ></Column>
    <Column
        sortable
        field="piz_state"
        header="Estado"
        body={statusPizza}
    ></Column>
    <Column
        field="ingredients"
        header="Ingredientes"
        body={IngredientsPizza}
    ></Column>
</DataTable>

```

Código Fuente 5.94: Modificación DaTatable

9. Noveno paso. Ahora se va a colocar un botón que va a permitir exportar en pdf un registro seleccionado de la tabla para ello se debe agregar un nuevo botón cerca de las acciones de actualizar y eliminar tal como se observa en el código fuente 5.95.

```

<button
    type="button"
    className="btn btn-success"
    style={{marginRight:"10px"}}
    onClick={() => {
        exportReportPdf(item,"Reporte Pizzas",["Nombre", "Calorías",
        "Porción","Estado"]);
    }}
>
    <i className="fa fa-download" aria-hidden="true" />
</button>

```

Código Fuente 5.95: Botón para exportar reporte en PDF

La Fig. 5.19, muestra la visualización de las opciones para generar el reporte PDF.

La Fig. 5.20 muestra cómo se visualiza el reporte pdf.

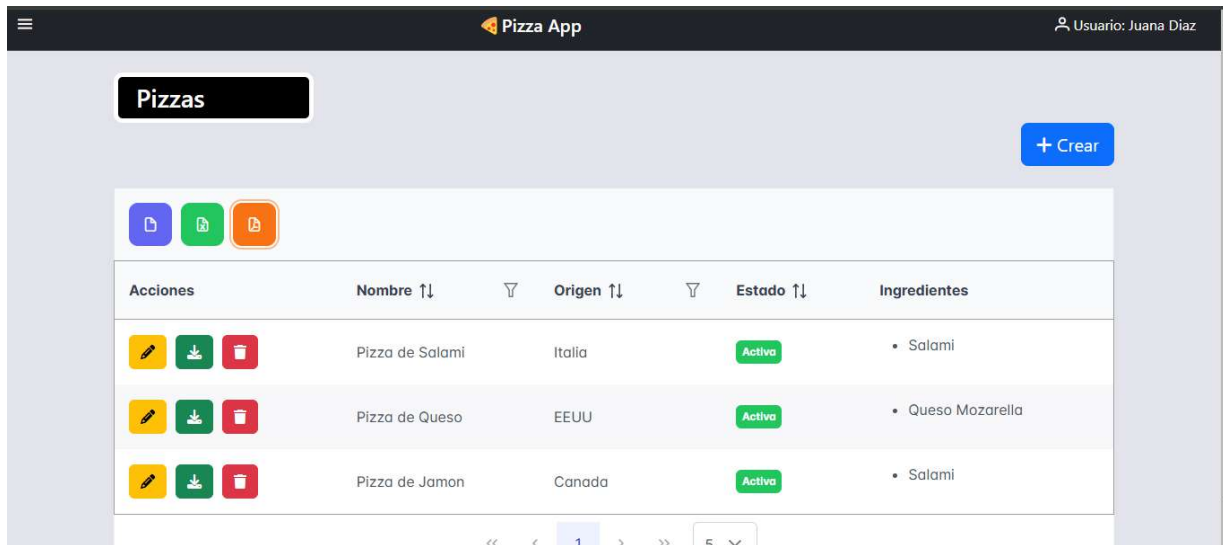


Fig. 5.19. Opciones para generar el reporte PDF

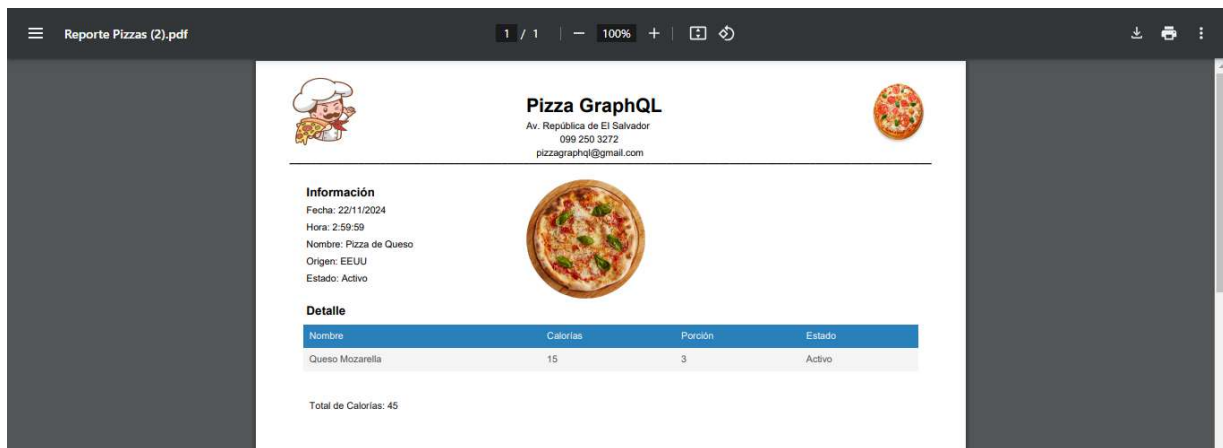


Fig. 5.20. Reporte PDF

10. Décimo paso. Para la creación del dashboard principal es necesario crear un servicio para establecer los tipos de gráficos que se visualizarán en el dashboard; ver código fuente 5.96.

```
const documentStyle = getComputedStyle(document.documentElement);
const textColor = documentStyle.getPropertyValue("--text-color");
const textColorSecondary = documentStyle.getPropertyValue(
  "--text-color-secondary"
);
const surfaceBorder = documentStyle.getPropertyValue("--surface-border");

const colors = [
  "#FF5733", // Rojo anaranjado
  "#33FF57", // Verde
  "#3357FF", // Azul
  "#F39C12", // Naranja
  "#8E44AD", // Púrpura
  "#3498DB", // Azul claro
  "#E74C3C", // Rojo
  "#2ECC71", // Verde esmeralda
];
```

```

"#1ABC9C", // Turquesa
"#9B59B6", // Lila
"#34495E", // Azul oscuro
"#F1C40F", // Amarillo
"#E67E22", // Naranja oscuro
"#F1C407", // Blanco humo
"#95A5A6", // Gris claro
];

const getColors = (data) => {
  let respColors = [];
  if (data !== undefined) {
    if (data.length === colors.length) {
      respColors = colors;
    } else if (data.length < colors.length) {
      for (let i = 0; i < data.length; i++) {
        respColors.push(colors[i]);
      }
    } else {
      let numberMissingColors = data.length - colors.length;
      let missingColors = [];
      for (let i = 0; i < numberMissingColors; i++) {
        missingColors.push(colors[i]);
      }
      respColors = [...colors, ...missingColors];
    }
  }

  return respColors;
};

const getNamesPizzas = (data) => {
  let names = [];
  if (data !== undefined) {
    data.forEach((element) => {
      names.push(element.piz_name);
    });
  }
  return names;
};

const getCaloriesPizzas = (data) => {
  let calories = [];
  if (data !== undefined) {
    data.forEach((element) => {
      calories.push(element.total_calories);
    });
  }
  return calories;
};

```

```
const getNamesIngredients = (data) => {
  let names = [];
  if (data !== undefined) {
    data.forEach((element) => {
      names.push(element.ing_name);
    });
  }
  return names;
};

const getCaloriesIngredients = (data) => {
  let calories = [];
  if (data !== undefined) {
    data.forEach((element) => {
      calories.push(element.ing_calories);
    });
  }
  return calories;
};

const getPizzasData = (data) => {
  return {
    labels: getNamesPizzas(data),
    datasets: [
      {
        label: "Numero de calorías",
        data: getCaloriesPizzas(data),
        backgroundColor: getColors(data),
        borderColor: getColors(data),
        borderWidth: 1,
      },
    ],
  };
};

const getIngredientsData = (data) => {
  return {
    labels: getNamesIngredients(data),
    datasets: [
      {
        label: "Numero de calorías",
        data: getCaloriesIngredients(data),
        backgroundColor: getColors(data),
        borderColor: getColors(data),
        borderWidth: 1,
      },
    ],
  };
};
```



```
const CharPizzaLine = (data) => {
  return {
    labels: getNamesPizzas(data),
    datasets: [
      {
        label: "Calorías por Pizzas",
        data: getCaloriesPizzas(data),
        fill: true,
        borderColor: documentStyle.getPropertyValue("--blue-500"),
        tension: 0.4,
        backgroundColor: documentStyle.getPropertyValue("--blue-100"),
      },
    ],
  };
};

const CharIngredientLine = (data) => {
  return {
    labels: getNamesIngredients(data),
    datasets: [
      {
        label: "Calorías por Ingredientes",
        data: getCaloriesIngredients(data),
        fill: true,
        borderColor: documentStyle.getPropertyValue("--orange-500"),
        tension: 0.4,
        backgroundColor: documentStyle.getPropertyValue("--orange-100"),
      },
    ],
  };
};

const RadarChart = (data, ingredients) => {
  const documentStyle = getComputedStyle(document.documentElement);
  const textColor = documentStyle.getPropertyValue("--text-color");

  return {
    labels: getNamesPizzas(data),
    datasets: [
      {
        label: "Pizzas",
        borderColor: documentStyle.getPropertyValue("--blue-400"),
        pointBackgroundColor: documentStyle.getPropertyValue("--blue-400"),
        pointBorderColor: documentStyle.getPropertyValue("--blue-400"),
        pointHoverBackgroundColor: textColor,
        pointHoverBorderColor: documentStyle.getPropertyValue("--blue-400"),
        data: getCaloriesPizzas(data),
      },
      {
        label: "Ingredientes",
        borderColor: documentStyle.getPropertyValue("--pink-400"),

```

```

        pointBackgroundColor: documentStyle.getPropertyValue("--pink-400"),
        pointBorderColor: documentStyle.getPropertyValue("--pink-400"),
        pointHoverBackgroundColor: textColor,
        pointHoverBorderColor: documentStyle.getPropertyValue("--pink-400"),
        data: getCaloriesIngredients(ingredients),
      },
    ],
  };
};

const PolarCharPizza = (data)=>{
  return {
    datasets: [
      {
        data: getCaloriesPizzas(data),
        backgroundColor: getColors(data),
        label: 'Pizzas'
      }
    ],
    labels: getNamesPizzas(data)
  };
}

const PolarOptions = ()=> {
  return {
    plugins: {
      legend: {
        labels: {
          color: textColor
        }
      }
    },
    scales: {
      r: {
        grid: {
          color: surfaceBorder
        }
      }
    }
  };
};

const configRadarChart = () => {
  return {
    plugins: {
      legend: {
        labels: {
          color: textColor,
        },
      },
    },
  },
};

```

```
scales: {
  r: {
    grid: {
      color: textColorSecondary,
    },
  },
};

const configLine = () => {
  return {
    maintainAspectRatio: false,
    aspectRatio: 0.6,
    plugins: {
      legend: {
        labels: {
          color: textColor,
        },
      },
    },
    scales: {
      x: {
        ticks: {
          color: textColorSecondary,
        },
        grid: {
          color: surfaceBorder,
        },
      },
      y: {
        ticks: {
          color: textColorSecondary,
        },
        grid: {
          color: surfaceBorder,
        },
      },
    },
  };
};

const optionsBar = () => {
  return {
    scales: {
      y: {
        beginAtZero: true,
      },
    },
  };
};
```

```
const optionsPieChart = () => {
  return {
    plugins: {
      legend: {
        labels: {
          usePointStyle: false,
        },
      },
    },
  };
};

const optionsDonutChart = () => {
  return {
    cutout: "60%",
  };
};

export {
  getPizzasData,
  getIngredientsData,
  optionsBar,
  optionsPieChart,
  optionsDonutChart,
  configLine,
  CharIngredientLine,
  CharPizzaLine,
  RadarChart,
  configRadarChart,
  PolarCharPizza,
  PolarOptions,
};
```

Código Fuente 5.96: Creación del servicio chart-service

11. Décimo primer paso. Se debe crear el componente para visualizar los gráficos; ver código fuente 5.97.

```
import React, { useState } from "react";
import { Card } from "primereact/card";
import NavBar from "../shared/navbar";
import { Knob } from "primereact/knob";
import { Chart } from "primereact/chart";
import { GET_PIZZAS } from "../../services/pizza-service";
import { GET_INGREDIENTS } from "../../services/ingredient-service";
import { useQuery } from "@apollo/client";
import {
  getIngredientsData,
  getPizzasData,
  optionsBar,
  optionsPieChart,
```

```
optionsDonutChart,
configLine,
CharPizzaLine,
CharIngredientLine,
configRadarChart,
RadarChart,
PolarCharPizza,
PolarOptions,
} from "../../services/chart-service";
```

Código Fuente 5.97: Importacion de dependencias

12. Décimo segundo paso. Se debe utilizar los servicios para listar la información de las pizzas e ingredientes; ver código fuente 5.98.

```
const pizzalist = useQuery(GET_PIZZAS, {
  pollInterval: 500,
});
const IngredientsList = useQuery(GET_INGREDIENTS);
```

Código Fuente 5.98: Servicios para obtener información

13. Décimo tercer paso. Se debe establecer los valores para cada uno de los gráficos que se van a mostrar cabe mencionar que en casi todos los gráficos se utiliza como parámetro un listado de información; ver código fuente 5.99.

```
const [numPizzas, setnumPizzas] = useState(pizzalist.data?.pizzas.length);
const [numIngredients, setnumIngredients] = useState(
  IngredientsList.data?.ingredients.length
);
const [chartPizzaData, setChartPizzaData] = useState(
  getPizzasData(pizzalist.data?.pizzas)
);
const [chartIngredientData, setChartIngredientData] = useState(
  getIngredientsData(IngredientsList.data?.ingredients)
);

const [chartBarOptions, setChartBarOptions] = useState(optionsBar());
const [chartPieOptions, setChartPieOptions] = useState(optionsPieChart());
const [chartRadar, setChartRadar] = useState(
  RadarChart(pizzalist.data?.pizzas, IngredientsList.data?.ingredients)
);
const [charRadarOptions, setChartCharRadarOptions] = useState(
  configRadarChart()
);
const [chartDonutOptions, setChartDonutOptions] = useState(
  optionsDonutChart()
);
```

Código Fuente 5.99: Servicios para obtener información

14. Décimo cuarto paso. Se debe establecer la estructura HTML del componente; ver código fuente 5.100.

```

<NavBar></NavBar>
<div>
  <Card style={{ width: "80%", marginLeft: "120px", marginTop: "30px" }}>
    <div className="row">
      <div className="col-sm-6">
        <Card
          title="Pizzas"
          style={{ textAlign: "center", fontSize: "12px" }}
        >
          <Knob
            value={numPizzas == undefined ? 0 : numPizzas }
            strokeWidth={10}
            size={100}
            style={{ marginTop: "-25px" }}
          />
        </Card>
      </div>
      <div className="col-sm-6">
        <Card
          title="Ingredientes"
          style={{ textAlign: "center", fontSize: "12px" }}
        >
          <Knob
            value={numIngredients == undefined ? 0 : numIngredients}
            strokeWidth={10}
            size={100}
            style={{ marginTop: "-25px" }}
            valueColor="#19d160"
          />
        </Card>
      </div>
    </div>
  </Card>

  <Card
    title=" Calorías por Pizzas"
    style={{ width: "80%", marginLeft: "120px", marginTop: "30px" }}
  >
    <div className="row">
      <div className="col-sm-6">
        <Card style={{ textAlign: "center" }}>
          <Chart
            type="bar"
            data={chartPizzaData}
            options={chartBarOptions}
          />
        </Card>
      </div>
    </div>

    <div className="col-sm-6">

```

```

        <Card style={{ textAlign: "center" }}>
          <Chart
            type="pie"
            data={chartPizzaData}
            options={chartPieOptions}
            height="240px"
          />
        </Card>
      </div>
    </div>
  </Card>

  <Card
    title="Calorías por Ingredientes"
    style={{ width: "80%", marginLeft: "120px", marginTop: "30px" }}
  >
    <div className="row">
      <div className="col-sm-6">
        <Card style={{ textAlign: "center" }}>
          <Chart
            type="bar"
            data={chartIngredientData}
            options={chartBarOptions}
          />
        </Card>
      </div>
      <div className="col-sm-6">
        <Card style={{ textAlign: "center" }}>
          <Chart
            type="pie"
            data={chartIngredientData}
            options={chartPieOptions}
            height="240px"
          />
        </Card>
      </div>
    </div>
  </Card>

  <Card
    title="Calorías Pizzas e Ingredientes"
    style={{ width: "80%", marginLeft: "120px", marginTop: "30px" }}
  >
    <div className="row">
      <div className="col-sm-6">
        <Card style={{ textAlign: "center" }}>
          <Chart
            type="doughnut"
            data={chartPizzaData}
            options={chartDonutOptions}
            height="250px"
          />
        </Card>
      </div>
    </div>
  </Card>

```

```

        />
      </Card>
    </div>

    <div className="col-sm-6">
      <Card style={{ textAlign: "center" }}>
        <Chart
          type="doughnut"
          data={chartIngredientData}
          options={chartDonutOptions}
          height="250px"
        />
      </Card>
    </div>
  </div>
</Card>

<Card
  title="Calorías Pizzas e Ingredientes"
  style={{ width: "80%", marginLeft: "120px", marginTop: "30px" }}
>
  <div className="row">
    <div className="col-sm-6">
      <Card style={{ textAlign: "center" }}>
        <Chart
          type="line"
          data={CharPizzaLine(pizzalist.data?.pizzas)}
          options={configLine}
          height="250px"
          width="400px"
        />
      </Card>
    </div>

    <div className="col-sm-6">
      <Card style={{ textAlign: "center" }}>
        <Chart
          type="line"
          data={CharIngredientLine(IngredientsList.data?.ingredients)}
          options={configLine}
          height="250px"
          width="400px"
        />
      </Card>
    </div>
  </div>
</div>

<Card
  title="Calorías Pizzas e Ingredientes"
  style={{ width: "80%", marginLeft: "120px", marginTop: "30px" }}

```



```
>
<div className="row">
  <Card style={{ textAlign: "center" }}>
    <Chart
      type="radar"
      data={chartRadar}
      options={charRadarOptions}
      height="250px"
      width="600px"
    />
  </Card>
</div>

</Card>
</div>
</>
```

Código Fuente 5.100: Estructura HTML componente Dashboard

Por último, las Fig. 5.21, Fig. 5.22 y Fig. 5.23 muestran los dashboards.

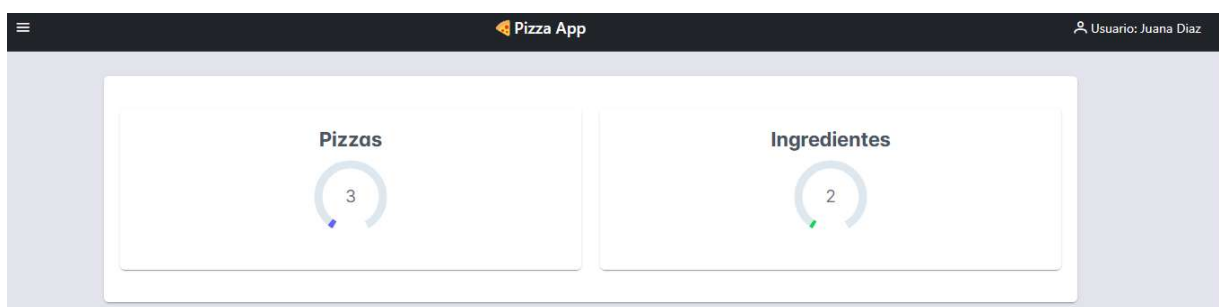


Fig. 5.21. Gráficos Dashboard

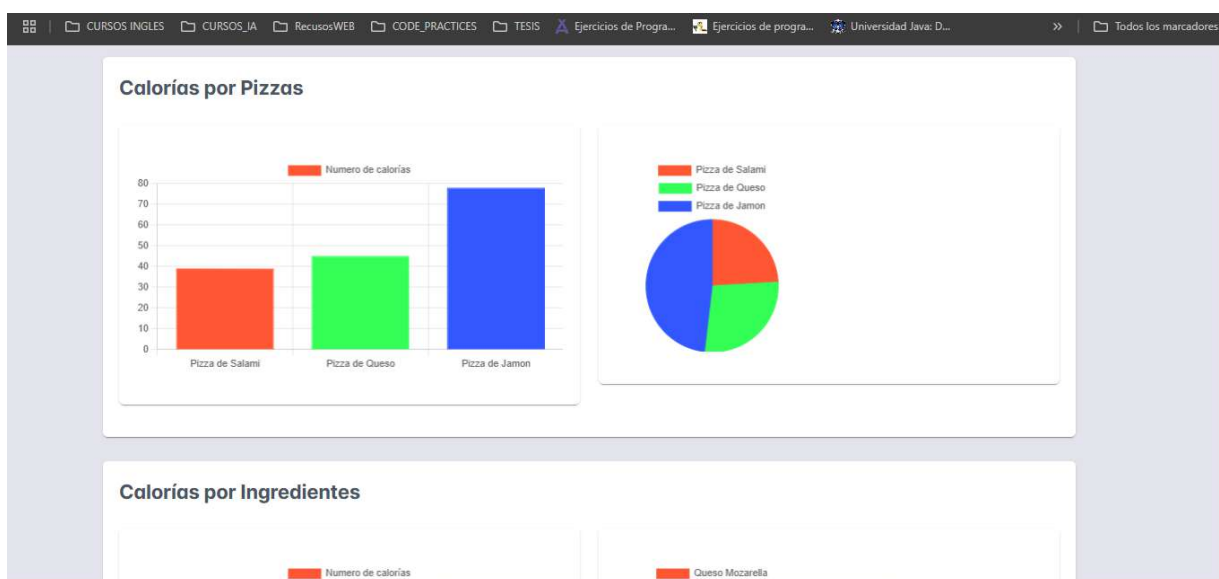


Fig. 5.22. Gráficos Dashboard



Fig. 5.23. Gráficos Dashboard

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-05.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-05.git>

CONSUMO E INTEGRACIÓN CON APIS REST

El consumo de APIs REST (Representational State Transfer) implica la interacción de aplicaciones cliente con servicios web a través de peticiones HTTP, permitiendo la obtención y manipulación de datos de manera eficiente y estructurada [54]. Este proceso se basa en la utilización de métodos estándar como GET, POST, PUT y DELETE, los cuales permiten realizar operaciones sobre recursos identificados por URIs. Al ser stateless, cada solicitud HTTP incluye toda la información necesaria para su procesamiento, lo que facilita la escalabilidad y la independencia entre el cliente y el servidor [55]. En los últimos años, el consumo de APIs REST se ha convertido en una práctica fundamental en el desarrollo de software, posibilitando la integración con servicios de terceros, el acceso a bases de datos remotas y la comunicación entre microservicios en arquitecturas distribuidas.

Se implementan tanto en el backend como en el frontend, empleando herramientas como **fetch** o bibliotecas como **Axio** en el caso de frameworks de JavaScript. Su uso permite integrar servicios externos (redes sociales, pasarelas de pago), consumir datos dinámicos y facilitar arquitecturas modulares. Además, tecnologías como autenticación mediante tokens y manejo de errores son cruciales para garantizar seguridad y confiabilidad.

Consumo del API REST de GEMINI

El consumo de APIs en inteligencia artificial (IA) permite integrar modelos y servicios de IA en aplicaciones. Estas APIs suelen ofrecer funcionalidades como procesamiento de lenguaje natural, visión por computadora, generación de texto o análisis de datos. Por ejemplo, las APIs de OpenAI se incluyen para

chatbots y generación de contenido, las de Google Cloud AI para reconocimiento de imágenes o traducción, y las de AWS para aprendizaje automático.

A través de métodos HTTP estándar, las aplicaciones pueden enviar datos (como texto o imágenes) y recibir resultados procesados por modelos avanzados. Esto reduce la necesidad de infraestructura propia, facilita la escalabilidad y acelera la implementación de soluciones basadas en IA. Además, garantiza actualizaciones constantes de los modelos y el acceso a tecnologías de punta.

GEMINI es el modelo de IA más avanzado de Google, diseñado para manejar texto, imágenes y audio en múltiples aplicaciones [56]. Su API REST permite a desarrolladores y empresas integrar funciones como generación de texto, búsqueda semántica, embeddings y procesamiento multimodal, facilitando casos de uso como análisis de datos, chatbots y personalización en tiempo real. Ofrece herramientas para gestionar documentos, permisos y modelos, con capacidades avanzadas para creación de contenido y codificación en lenguajes populares. Además, su enfoque en seguridad y eficiencia lo hace ideal para aplicaciones empresariales y científicas.

La API de Gemini es una interfaz diseñada para que los desarrolladores incorporen las funcionalidades de los modelos de IA de Gemini en sus aplicaciones [56]. Esta API permite interactuar con modelos generativos multimodales, útiles para tareas como la creación de contenido, el análisis de datos y otras actividades. El acceso a la API de Gemini se gestiona mediante solicitudes HTTP, utilizando métodos comunes como POST para enviar información y recibir respuestas del modelo [56]. Es compatible con múltiples formatos de entrada, como texto, imágenes y audio, lo que permite desarrollar aplicaciones flexibles y robustas.

Antes de la creación del proyecto para el backend, se debe obtener las credenciales para hacer uso del API REST de GEMINI por lo cual es necesario los siguientes pasos:

1. Dirigirse a la pagina oficial de GEMINI ¹¹ como se observa en la Fig. 5.24.

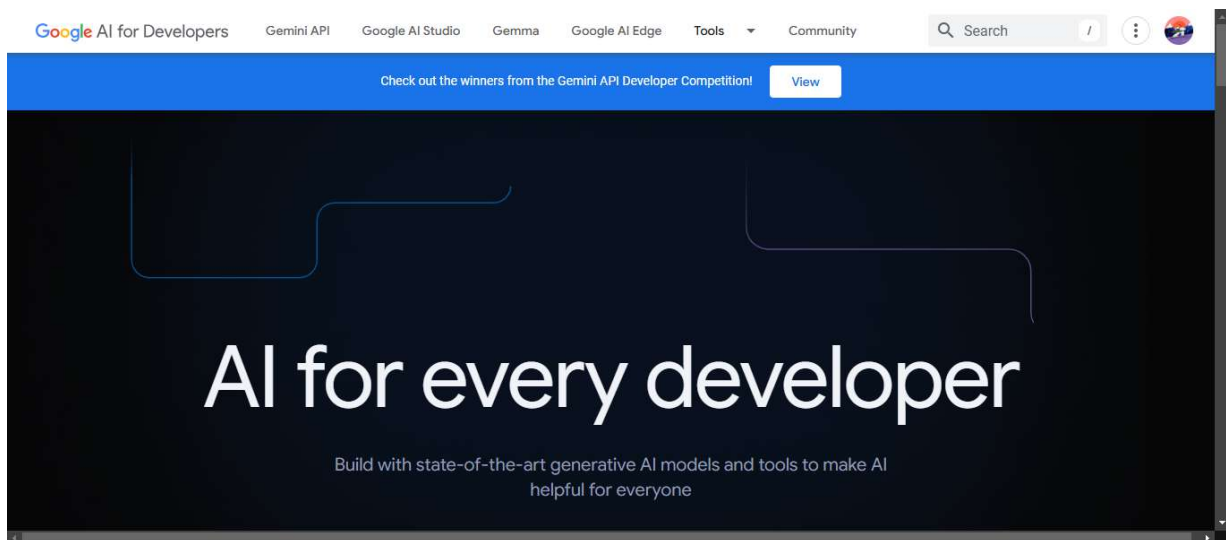


Fig. 5.24. Página Oficial de Gemini

2. Seleccionar la opción Google AI Studio; ver la Fig. 5.25.

¹¹GEMINI: <https://ai.google.dev/>

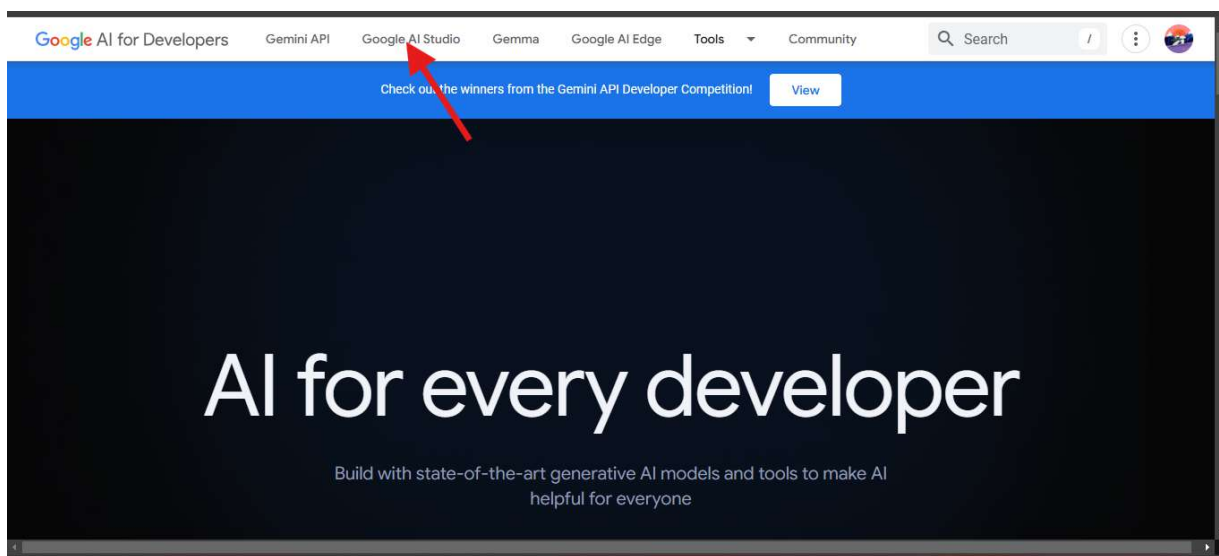


Fig. 5.25. Ingreso a Google AI Studio

3. Escoger la opción **get api key**; ver la Fig. 5.26.

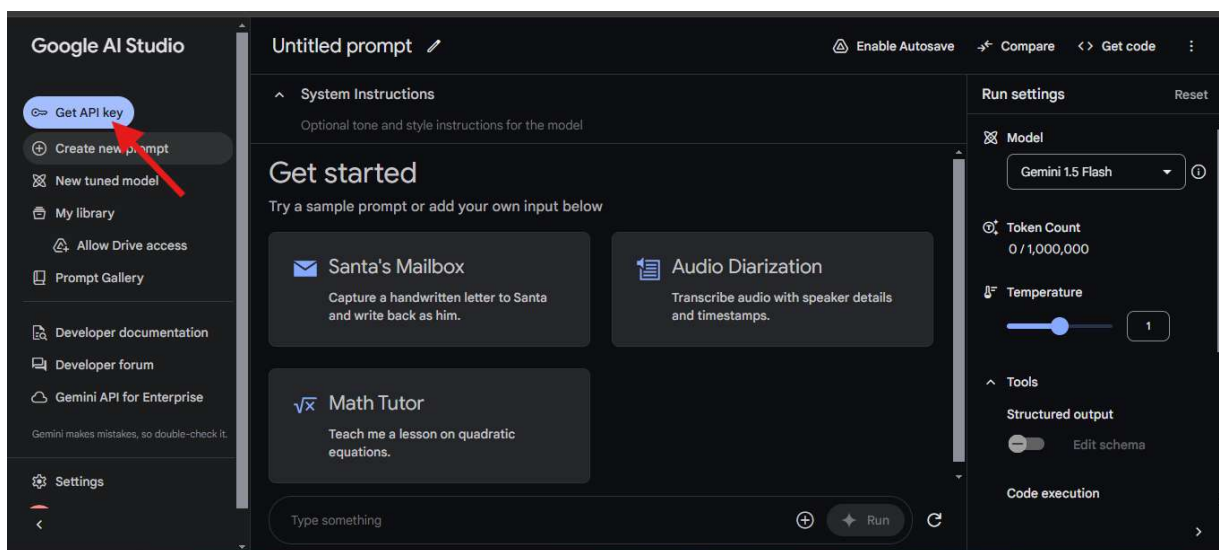


Fig. 5.26. Crear clave del API

4. Seleccionar un proyecto; ver la Fig. 5.27.

5. Como último paso copiar la clave de nuestra API; ver la Fig. 5.28.

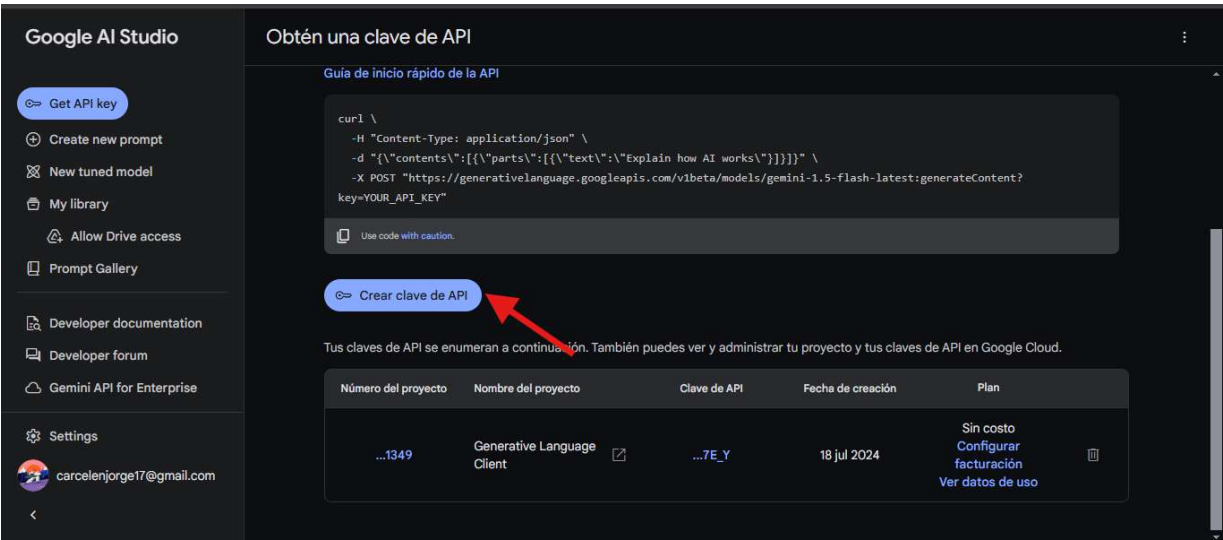


Fig. 5.27. Clave del API

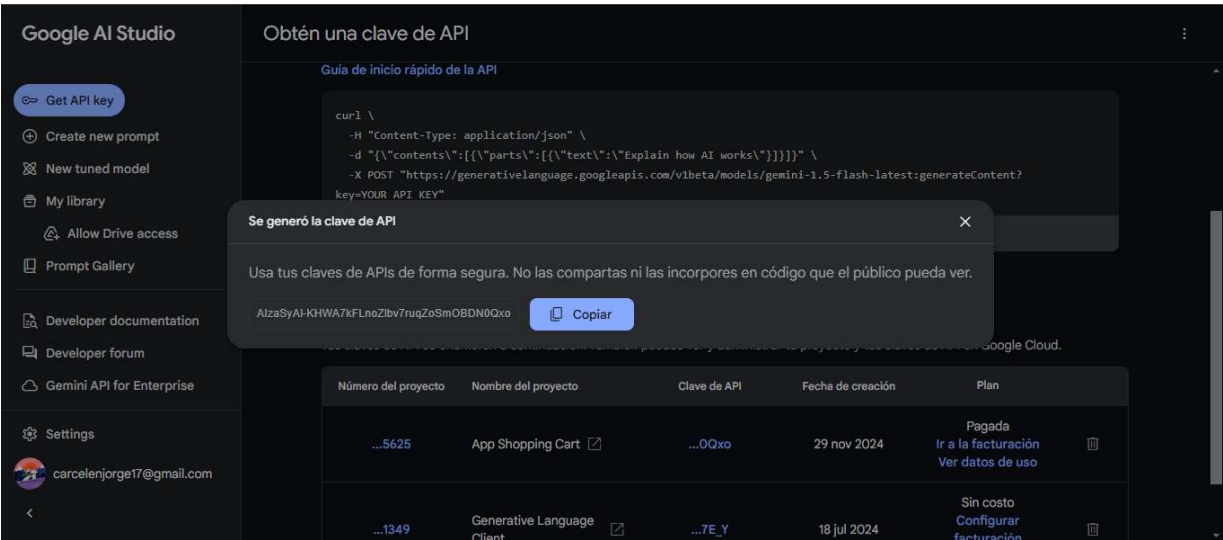


Fig. 5.28. Clave del API

Creación del proyecto backend en NodeJS

- Crear un proyecto, para ello utilizar npm init –y con la opción –y para evitar las preguntas al crear un nuevo proyecto.
- Realizar la instalación de las siguientes dependencias.

Comando utilizado

npm i nodemon apollo-server dotenv merge-graphql-schemas node-fetch

- Crear la siguiente estructura de carpetas o directorios que se aprecia en la Fig. 5.29.

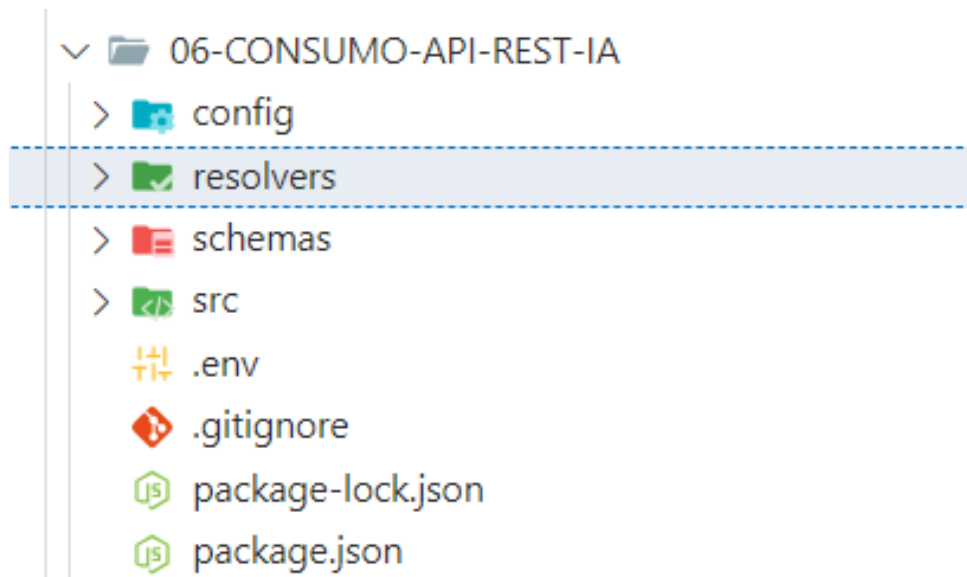


Fig. 5.29. Estructura de carpetas

- Dentro del archivo `.env` es necesario agregar las siguientes variables de entorno para configurar las credenciales de acceso hacia la API de GEMINI; ver código fuente 5.101.

```
API_TOKEN='AIzaSyCdMQ1X9bFea80E8MQC9-xxxxxxx'
API_URL="https://generativelanguage.googleapis.com/v1beta/models
/gemini-1.5-flash:generateContent?key="
```

Código Fuente 5.101: Variables de entorno

- En la carpeta `config` se debe crear un archivo llamando **`env.vars.js`** que va a permitir exportar las variables de entorno; ver código fuente 5.102.

```
require("dotenv/config");

module.exports = {
  API_URL: process.env.API_URL,
  API_TOKEN: process.env.API_TOKEN,
};
```

Código Fuente 5.102: Exportar variables de entorno

- En la carpeta **`type-system`** se debe crear un archivo **`schema.graphql`**. Se debe agregar los tipos **`Query`** y **`Mutation`**. Cabe mencionar que se crearon dos formas para interactuar con la API de GEMINI, en el servicio de la aplicación frontend se puede utilizar la consulta o la mutación para consumir la API que se va a crear en el backend; ver código fuente 5.103.

```
type Query {
  chatbot(prompt: String): String
}

type Mutation {
  SendPrompt(prompt: String): String
}
```

Código Fuente 5.103: Esquema de datos

- En la carpeta **controllers** es necesario crear un **resolver** llamado **controller.js** para utilizar tanto la consulta como la mutación que se estableció en el esquema de datos. A diferencia de los capítulos anteriores se hace uso de la librería **fetch** para consumir el API REST de GEMINI como el origen de datos (En lugar de la base de datos), y posteriormente obtener los resultados de la consulta y retornar los datos como resultado del método; ver código fuente 5.104.

```
const fetch = (...args) =>
  import("node-fetch").then(({ default: fetch }) => fetch(...args));
const vars = require("../config/env-vars.js");
const resolver = {
  Query: {
    async chatbot(root, { prompt }) {
      if (prompt == undefined) {
        throw new Error("Debe ingresar un prompt ");
      } else {
        const url = vars.API_URL + vars.API_TOKEN;
        const headers = {
          "Content-Type": "application/json",
        };
        const data = {
          contents: [
            {
              parts: [{ text: prompt }],
            },
          ],
        };
        const response = await fetch(url, {
          method: "POST",
          headers: headers,
          body: JSON.stringify(data),
        })
          .then((response) => response.json())
          .then(async (data) => {
            return data.candidates[0].content.parts[0].text;
          })
          .catch((error) => {
            return error;
          });

        return response
      }
    },
  },
  Mutation: {
    async SendPrompt(root, { prompt }) {
      if (prompt == undefined) {
        throw new Error("Debe ingresar un prompt ");
      } else {
        const url = vars.API_URL + vars.API_TOKEN;
        const headers = {
          "Content-Type": "application/json",

```

```

    };

    const data = {
      contents: [
        {
          parts: [{ text: prompt }],
        },
      ],
    };

    const response = await fetch(url, {
      method: "POST",
      headers: headers,
      body: JSON.stringify(data),
    })
      .then((response) => response.json())
      .then(async (data) => {
        return data.candidates[0].content.parts[0].text;
      })
      .catch((error) => {
        return error;
      });

    return response
  }
}
};

module.exports = resolver;

```

Código Fuente 5.104: Creación del resolver

- Se debe modificar el archivo **index.js** para crear la instancia del servidor de Apollo; ver código fuente 5.105.

```

const { ApolloServer } = require("apollo-server");
const { fileLoader, mergeTypes } = require("merge-graphql-schemas");

const typeDefs = mergeTypes(fileLoader("./type-system/schema.graphql"));
const resolvers = require("../controllers/controller");

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

server.listen(4000).then(({ url }) => {
  console.log(`Run server in the URL: ${url}`);
});

```

Código Fuente 5.105: Modificación del archivo index.js

Creación del proyecto frontend en React

1. Crear un proyecto nuevo con el siguiente comando.

Comando Utilizado

```
npx create-react-app chatbot
```

2. Posteriormente instalar las dependencias que se van a utilizar.

Comando Utilizado

```
npm i @apollo/client @chatscope/chat-ui-kit-react
```

Para este ejercicio, se utilizó el paquete de React `@chatscope/chat-ui-kit-react`, el cual contiene componentes especializados para construir interfaces de chat modernas y personalizables. Incluye elementos como listas de mensajes, entradas de texto y avatares. Es ideal para crear aplicaciones de mensajería o chatbots con diseño profesional y responsivo [57].

3. Crear el servicio para llamar las mutaciones y consultas dentro de la carpeta `services`; ver código fuente 5.106.

```
import { gql } from "@apollo/client";

const SEND_PROMPT = gql`
mutation SendPrompt($prompt: String){
  SendPrompt(prompt: $prompt)
}
`;

export { SEND_PROMPT };
```

Código Fuente 5.106: Creación del servicio

Cabe mencionar que antes de realizar este paso se debe eliminar varios archivos y estilos que vienen por defecto al momento de crear una aplicación en React, tal como se explicó en el Capítulo 4.

4. Una vez definido el servicio, se debe crear un componente que permita ingresar prompts para interactuar con la API.
5. Importar las dependencias necesarias; ver código fuente 5.107.

```
import { useState } from "react";
import "./App.css";
import { useMutation } from "@apollo/client";
import "@chatscope/chat-ui-kit-styles/dist/default/styles.min.css";
import {
  MainContainer,
  ChatContainer,
  MessageList,
  Message,
  MessageInput,
  TypingIndicator,
} from "@chatscope/chat-ui-kit-react";
```

```
import { SEND_PROMPT } from "../services/chatbot-service";
```

Código Fuente 5.107: Importación de las dependencias

6. Crear una variable para identificar cuando el chat vaya a generar la respuesta del prompt realizado; ver código fuente 5.108.

```
const [isTyping, setIsTyping] = useState(false);
```

Código Fuente 5.108: Declaración de variables

7. Posteriormente, utilizar la mutación **SENDPROMPT**. Cabe mencionar que esta mutación se va a llamar desde el método **processMessageToChat**. Una vez procesado el mensaje, se podrá ver la respuesta de la API; ver código fuente 5.109.

```
const [sendPrompt] = useMutation(SEND_PROMPT);

const handleSend = async (message) => {
  const newMessage = {
    message,
    direction: "outgoing",
    sender: "user",
  };

  const newMessages = [...messages, newMessage];

  setMessages(newMessages);
  setIsTyping(true);
  await processMessageToChat(message);
};

const processMessageToChat = (message) => {
  sendPrompt({
    variables: {
      prompt: message,
    },
  })
  .then((resp) => {
    const value = resp.data.SendPrompt;

    const newMessage = {
      message: value,
      sentTime: "justo ahora",
      sender: "ChatAI",
    };

    const newMessages = [...messages, newMessage];
    setMessages(newMessages);
    setIsTyping(false);
  })
  .catch((error) => {
    console.log(error);
  });
};
```

```
});
```

Código Fuente 5.109: Uso del Servicio

8. Por último, se debe utilizar los componentes **chatcontainer** y **maincontainer** para crear la estructura del componente. Como se había mencionado, estos componentes vienen de la librería **chatscope/chat-ui-kit-react**; ver código fuente 5.110.

```
<div className="App">
  <MainContainer>
    <ChatContainer>
      <MessageList
        scrollBehavior="smooth"
        typingIndicator={
          isTyping ? (
            <TypingIndicator content="ChatAI está escribiendo" />
          ) : null
        }
      >
        {messages.map((message, i) => {
          console.log(message);
          return <Message key={i} model={message} />;
        })}
      </MessageList>
      <MessageInput
        placeholder="Escribe tu mensaje aquí"
        onSend={handleSend}
      />
    </ChatContainer>
  </MainContainer>
</div>
```

Código Fuente 5.110: Creación de la estructura del componente

Una vez terminado se puede visualizar la aplicación; ver Fig. 5.30.

A continuación se observa un ejemplo. Se realiza una pregunta para obtener una lista de las cinco canciones famosas actuales; ver Fig. 5.31.

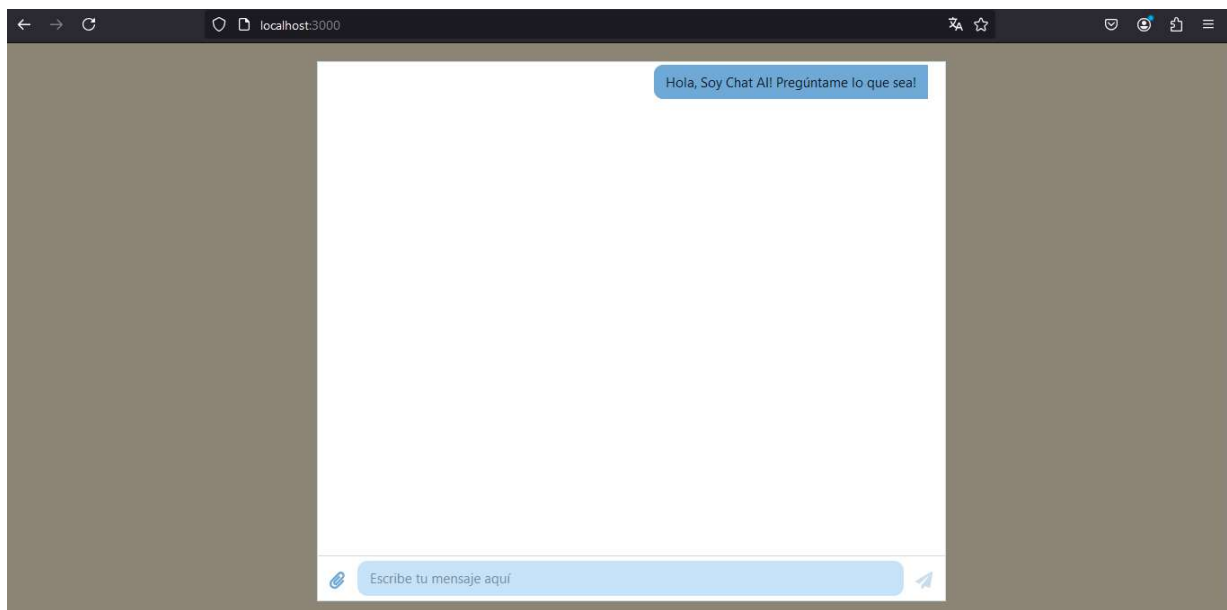


Fig. 5.30. Aplicación Frontend

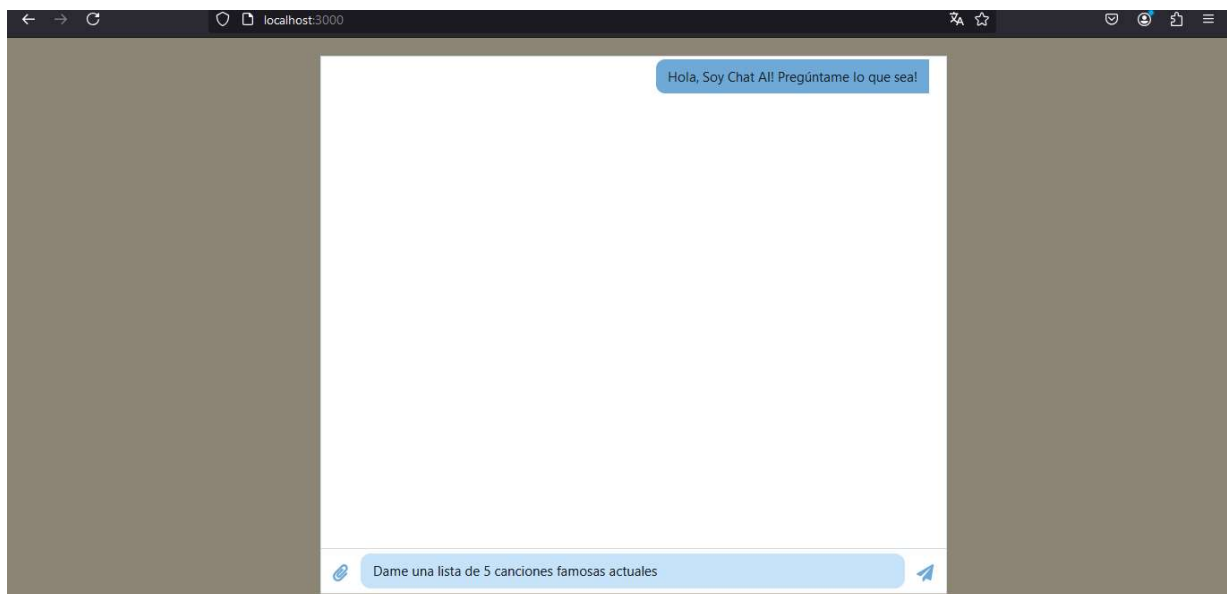


Fig. 5.31. Aplicación Frontend

Una vez realizada la pregunta se obtiene el siguiente resultado; ver la Fig. 5.32.

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-06.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-06.git>

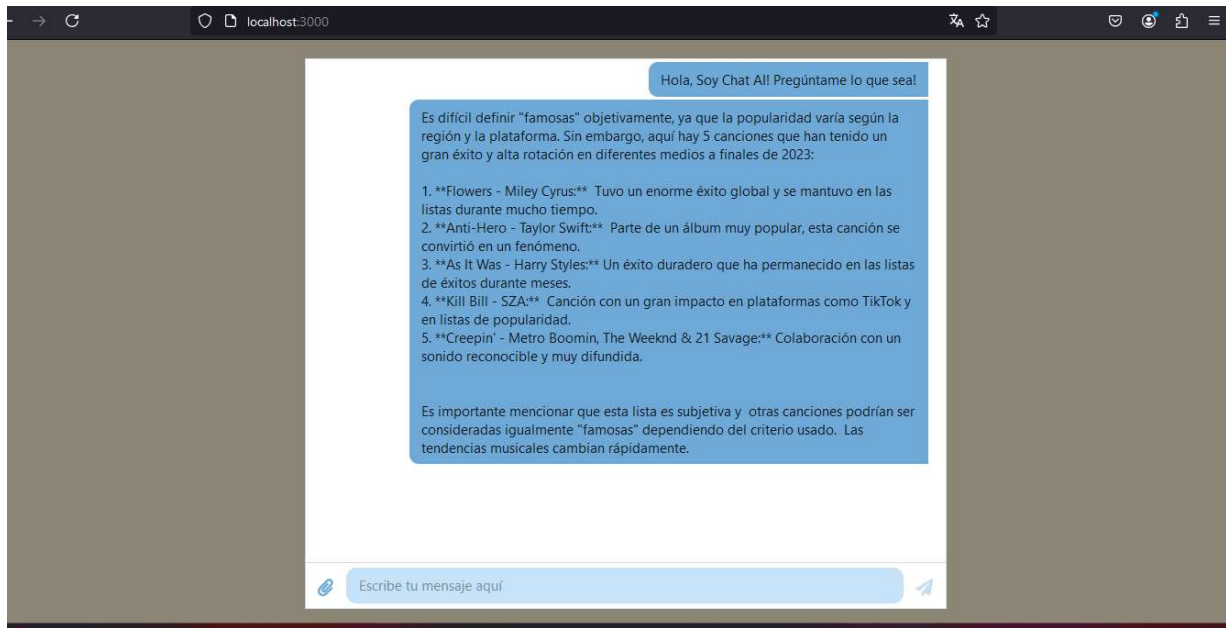


Fig. 5.32. Aplicación Frontend

Consumo del API REST de YouTube

YouTube es un servicio en línea que se ha convertido en una plataforma fundamental para compartir, consultar y conversar sobre historias audiovisuales en un entorno digital. Con una oferta inmensa de contenido que abarca desde entretenimiento, educación, música, tutoriales, noticias, deportes y mucho más, YouTube ha transformado la forma en que las personas consumen y producen contenido en el mundo moderno [58].

Al ser un servicio basado en Internet, es natural que la plataforma esté diseñada para operar de manera continua, garantizando que su biblioteca de videos esté disponible las 24 horas del día, los siete días de la semana. Esta accesibilidad permanente permite a los usuarios disfrutar de contenido en cualquier momento que lo deseen, sin restricciones horarias ni interrupciones [58]. Además, YouTube se destaca por ser una herramienta inclusiva y global, a la que se puede acceder desde cualquier dispositivo con conexión a Internet, ya sea una computadora, un teléfono inteligente, una tableta o incluso un televisor inteligente. Este enfoque multiplataforma asegura que los usuarios puedan interactuar con la plataforma desde cualquier lugar del mundo, sin importar su ubicación geográfica.

Creación del proyecto backend en NodeJS

Para la creación del proyecto backend se deben seguir los siguientes pasos:

1. Establecer la estructura del proyecto tal y como se observa en la Fig. 5.33.

```
> config
> resolvers
> schema
> services
> src
.env
.gitignore
package-lock.json
package.json
```

Fig. 5.33. Estructura proyecto

2. Realizar la instalación de las siguientes dependencias.

Comando Utilizado

```
npm i @apollo/server @graphql-tools/graphql-file-loader @graphql-tools/load @graphql-tools/load-files dotenv graphql graphql-tag nodemon
```

3. Declarar variables de entorno para obtener acceso a las credenciales del API de YouTube; ver código fuente 5.111.

```
API_TOKEN='AIzaSyAxbHqygEsqfgh8MhkhaKOP-7M3Ksu1P7k0r0r0'
API_URL="https://www.googleapis.com/youtube/v3/"
```

Código Fuente 5.111: Credenciales API de Youtube

4. En la carpeta **config** crear un archivo que permita exportar las variables de entorno que se definieron en el archivo **.env**; ver código fuente 5.112.

```
import * as dotenv from 'dotenv';

dotenv.config();

const config = {
  API_TOKEN: process.env.API_TOKEN,
  API_URL: process.env.API_URL
};

export { config };
```

Código Fuente 5.112: Archivo env-vars.js

5. En la carpeta **schema** se debe crear un archivo **schema.graphql** para definir los tipos de datos y consultas que se utilizarán; ver código fuente 5.113.

```
type Video {
  id: String
```

```
    title: String
    description: String
    channelTitle: String
    player: String
  }

  type Playlist {
    id: String
    title: String
    description: String
    channelTitle: String
    player: String
  }

  type SearchBody {
    id: String
    title: String
    description: String
    thumbnail: Thumbnail
  }

  type SearchVideo {
    id: String
    title: String
    description: String
    thumbnail: Thumbnail
    videos: [Video]
  }

  type Channel {
    id: String
    title: String
    description: String
    thumbnail: Thumbnail
    playlists: [Playlist]
  }

  type Thumbnail {
    url: String
    width: Int
    height: Int
  }

  type Query {
    video(maxresults: Int): [Video]
    SearchVideos(filter: String!): [SearchVideo]
    SearchChannels(filter: String!): [Channel]
    listVideo(id: String): [Video]
    SearchPlaylists(filter: String!): [SearchBody]
  }
```

Código Fuente 5.113: Archivo schema.graphql

6. Es importante destacar que el API de YouTube permite obtener el enlace de un archivo embebido. En el caso de los videos, es necesario crear un servicio que permita obtener la URL de ese archivo, como se observa en el código fuente 5.114.

```
const getFrameUrl = (item) => {
  return "https://" + item.player.embedHtml.substring(40, 73);
};

export {getFrameUrl}
```

Código Fuente 5.114: Archivo query-service.js

7. Como siguiente paso, se debe crear el resolver para interactuar con la API de YouTube y consultar información relacionada con videos, canales y listas de reproducción. Es importante mencionar que, al igual que el ejercicio anterior, se utilizará el método **fetch** para obtener los valores del API de YouTube y retornar los valores mediante consultas que se van a definir en el resolver; ver código fuente 5.115.

```
import { config } from "../config/env-vars.js";
const API_KEY = config.API_TOKEN;
const API_URL = config.API_URL;
import { getFrameUrl } from "../services/query-service.js";
const fetch = (...args) =>
  import("node-fetch").then(({ default: fetch }) => fetch(...args));

const resolver = {
  Query: {
    video: async (_root, { maxresults }) => {
      let numResults = maxresults || 20;
      const response = await fetch(
        `${API_URL}videos?&key=${API_KEY}&part=player,` +
        `contentDetails,` +
        `topicDetails` +
        `,snippet&chart=mostPopular&maxResults=${numResults}`,
        {
          method: "GET",
        }
      )
      .then((res) => res.json())
      .catch((error) => error);

      return response.items?.map((item) => ({
        id: item.id,
        title: item.snippet.title,
        description: item.snippet.description,
        channelTitle: item.snippet.channelTitle,
        player: getFrameUrl(item),
      }));
    }
  }
}
```



```

    ))) || [];
  },
  SearchChannels: async (_root, { filter }) => {
    if (!filter) throw new Error("Debe proporcionar un filtro para buscar");

    const response = await fetch(
      `${API_URL}search?key=${API_KEY}&part=snippet&q=${filter}&type=channel`,
      {
        method: "GET",
      }
    )
    .then((res) => res.json())
    .catch((error) => error);

    return response.items?.map((item) => ({
      id: item.id.channelId,
      title: item.snippet.title,
      description: item.snippet.description,
      thumbnail: item.snippet.thumbnails.default,
    }))) || [];
  },
  SearchVideos: async (_root, { filter }) => {
    if (!filter) throw new Error("Debe proporcionar un filtro para buscar");

    const response = await fetch(
      `${API_URL}search?key=${API_KEY}&part=snippet&q=${filter}&type=video`,
      {
        method: "GET",
      }
    )
    .then((res) => res.json())
    .catch((error) => error);

    return response.items?.map((item) => ({
      id: item.id.videoId,
      title: item.snippet.title,
      description: item.snippet.description,
      thumbnail: item.snippet.thumbnails.default,
    }))) || [];
  },
  SearchPlaylists: async (_root, { filter }) => {
    if (!filter) throw new Error("Debe proporcionar un filtro para buscar");

    const response = await fetch(
      `${API_URL}search?key=${API_KEY}&part=snippet&q=${filter}&type=playlist`,
      {
        method: "GET",
      }
    )
    .then((res) => res.json())
    .catch((error) => error);
  }
}

```

```

    return response.items?.map((item) => ({
      id: item.id.playlistId,
      title: item.snippet.title,
      description: item.snippet.description,
      thumbnail: item.snippet.thumbnails.default,
    })) || [];
  },
  listVideo: async (_root, { id }) => {
    if (!id) throw new Error("Debe proporcionar un ID válido");

    const response = await fetch(
      `${API_URL}videos?&key=${API_KEY}&part=snippet&id=${id}`,
      {
        method: "GET",
      }
    )
      .then((res) => res.json())
      .catch((error) => error);

    return response.items?.map((item) => ({
      id: item.id,
      title: item.snippet.title,
      description: item.snippet.description,
      channelTitle: item.snippet.channelTitle,
      player: getFrameUrl(item),
    })) || [];
  },
},
Channel: {
  playlists: async (channel) => {
    if (!channel) return [];

    const response = await fetch(
      `${API_URL}playlists?&key=${API_KEY}&part=player,snippet&channelId=${channel.id}`,
      {
        method: "GET",
      }
    )
      .then((res) => res.json())
      .catch((error) => error);

    return response.items?.map((item) => ({
      id: item.id,
      title: item.snippet.title,
      description: item.snippet.description,
      channelTitle: item.snippet.channelTitle,
      player: item.player.embedHtml,
    })) || [];
  },
},

```

```

    },
  },

  SearchVideo: {
    videos: async (video) => {
      if (!video) return [];

      const response = await fetch(
        `${API_URL}videos?&key=${API_KEY}&part=player,
        contentDetails,topicDetails,snippet&id=${video.id}`,
        {
          method: "GET",
        }
      )
      .then((res) => res.json())
      .catch((error) => error);

      return response.items?.map((item) => ({
        id: item.id,
        title: item.snippet.title,
        description: item.snippet.description,
        channelTitle: item.snippet.channelTitle,
        player: getFrameUrl(item),
      })) || [];
    },
  },
};

export default resolver;

```

Código Fuente 5.115: Archivo main-resolver.js

8. Como último paso crear el archivo index de la aplicación; ver código fuente 5.116.

```

import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone";
import path from "path";
import { fileURLToPath } from "url";
import { loadFilesSync } from "@graphql-tools/load-files";
import { mergeTypeDefs, mergeResolvers } from "@graphql-tools/merge";

// Obtener la ruta del directorio actual
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Cargar y combinar los schemas
const allSchemas = mergeTypeDefs(
  loadFilesSync(path.join(__dirname, "../schema/**/*.graphql"))
);

// Cargar y combinar resolvers
const allResolvers = mergeResolvers(

```

```
loadFilesSync(path.join(__dirname, "../resolvers/**/*.js"))
);

// Configuración del servidor Apollo
const server = new ApolloServer({
  typeDefs: allSchemas,
  resolvers: allResolvers,
});

// Iniciar el servidor Apollo
startStandaloneServer(server, {
  listen: { port: 4000 },
}).then((server) => {
  console.log(`Server ready at: ${server.url}`);
});
```

Código Fuente 5.116: Archivo index.js

Creación del proyecto frontend en React

1. Crear un proyecto nuevo con el siguiente comando.

Comando Utilizado

```
npx create-react-app chatbot
```

2. Posteriormente instalar las dependencias que se van a utilizar.

Comando Utilizado

```
npm i @apollo/client primeicons primereact react-router-dom
```

3. Realizar la configuración del cliente de Apollo y **primereact** como se ha realizado anteriormente en el Capítulo 4; ver código fuente 5.117.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css';
import App from './App';
import { PrimeReactProvider } from "primereact/api";
import { BrowserRouter } from "react-router-dom";
import "primeicons/primeicons.css";
import "primereact/resources/themes/lara-light-indigo/theme.css";
import {
  ApolloClient, ApolloProvider, HttpLink, InMemoryCache
} from '@apollo/client'

const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: new HttpLink({
    uri: 'http://localhost:4000',
  })
});
```

```
  })

  createRoot(document.getElementById('root')).render(
    <StrictMode>
      <ApolloProvider client={client}>
        <PrimeReactProvider>
          <BrowserRouter>
            <App />
          </BrowserRouter>
        </PrimeReactProvider>
      </ApolloProvider>
    </StrictMode>,
  )
```

Código Fuente 5.117: Archivo main.jsx

4. Como siguiente paso, es crear un servicio dentro de la carpeta **service** para interactuar con las consultas que se definieron en el esquema; ver código fuente 5.118.

```
import { gql } from "@apollo/client";
const GET_POPULAR_VIDEOS = gql`
  {
    video {
      title
      channelTitle
      player
    }
  }
`;

const GET_SEARCH_CHANNELS = gql`
  query ($filter: String!) {
    SearchChannels(filter: $filter) {
      id
      title
      description
      thumbnail {
        url
      }
      playlists {
        title
        channelTitle
        player
        id
      }
    }
  }
`;
```

```
const GET_SEARCH_PLAYLISTS = gql`
  query ($filter: String!) {
    SearchPlaylists(filter: $filter) {
      id
      title
      description
      thumbnail {
        url
      }
    }
  }
`;

const GET_VIDEOS_BY_ID = gql`
  query ($listVideoId: String) {
    listVideo(id: $listVideoId) {
      id
      title
      channelTitle
      player
    }
  }
`;

const GET_SEARCH_VIDEOS = gql`
  query ($filter: String!) {
    SearchVideos(filter: $filter) {
      id
      title
      description
      thumbnail {
        url
      }
      videos {
        player
      }
    }
  }
`;

export {
  GET_POPULAR_VIDEOS,
  GET_SEARCH_CHANNELS,
  GET_SEARCH_PLAYLISTS,
  GET_SEARCH_VIDEOS,
  GET_VIDEOS_BY_ID
};
```

Código Fuente 5.118: Archivo api-service.js

5. En este momento se va a crear un componente dentro de la carpeta **components** para visualizar los

videos más populares. Dentro de este servicio se debe utilizar la consulta **GET-POPULAR-VIDEOS** para mostrar la información en un componente **datatable**; ver código fuente 5.119.

```
import React from "react";
import { useQuery } from "@apollo/client";
import { GET_POPULAR_VIDEOS } from "../../services/api-service";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";

const PopularVideoComponent = () => {
  const popularVideoList = useQuery(GET_POPULAR_VIDEOS);

  const HtmlContent = (video) => {

    return (
      <iframe
        width="300"
        height="150"
        src={video.player}
      ></iframe>
    );
  };

  return (
    <div>
      <>
        <div className="container">
          <DataTable
            value={popularVideoList.data?.video}
            showGridlines
            stripedRows
            paginator
            rows={5}
            filterLocale="ES"
            rowsPerPageOptions={[5, 10, 25, 50]}
            tableStyle={{ minWidth: "60rem" }}
            emptyMessage="No existen registros."
          >
            <Column sortable field="channelTitle" header="Canal"></Column>

            <Column sortable field="title" header="Título"></Column>

            <Column
              field="player"
              header="Video"
              body={HtmlContent}
            ></Column>
          </DataTable>
        </div>
      </>
    </div>
  );
};
```

```
};

export default PopularVideoComponent;
```

Código Fuente 5.119: Componente PopularVideo

6. Cuando se llame al componente desde la aplicación principal, es decir, desde el archivo **App.jsx** se debe visualizar de la siguiente forma; ver Fig. 5.34.

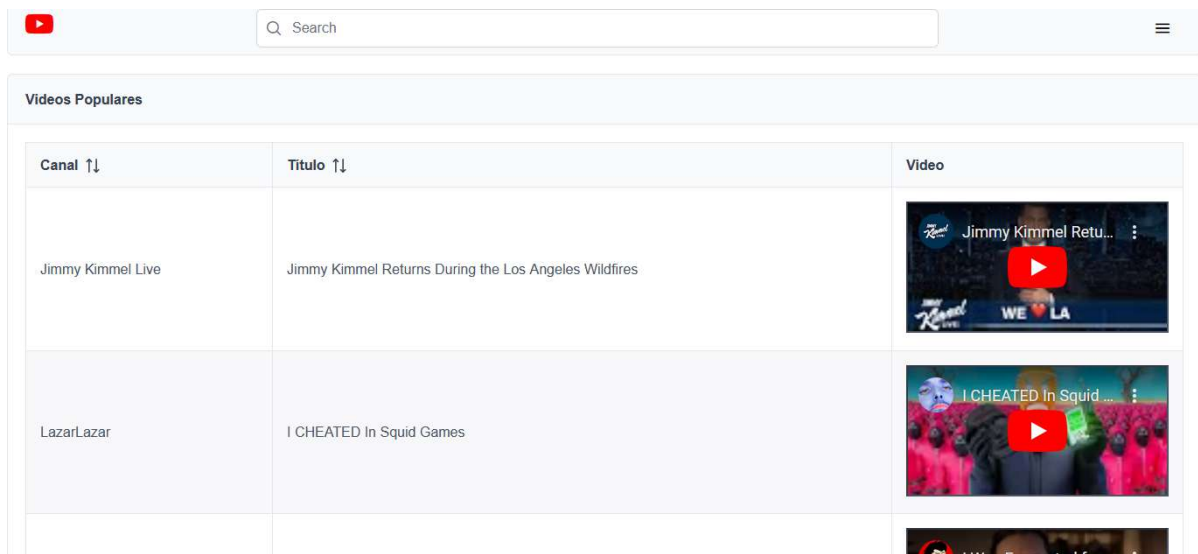


Fig. 5.34. Componente Popular Videos

7. Se debe crear un componente para visualizar el contenido de los canales que se van a buscar. Dentro de la carpeta **components** es necesario crear un componente llamado **SearchPlaylistComponent**; ver código fuente 5.120.

```
import React, { useState } from "react";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";
import { Button } from "primereact/button";
import { Dialog } from "primereact/dialog";
import { Panel } from "primereact/panel";

const SearchPlaylistComponent = ({ data }) => {
  const [visible, setVisible] = useState(false);
  const [channel, setChannel] = useState([]);

  const setData = (value) => {
    setVisible(true);
    setChannel(value);
  };

  const HtmlContent = (value) => {
    console.log(value);
    return (
```



```

        <>
        <Button icon="pi pi-external-link" onClick={() => setData(value)} />
    </>
    );
};

const VideoContent = (value) => {
    return (
        <a
            href={"https" + value.player.substring(42, 118)}
            target="_blank"
            rel="noopener noreferrer"
            className="p-button"
            style={{ textDecoration: "none" }}
        >
            <i className="pi pi-check" style={{ fontSize: "1rem" }}></i>
        </a>
    );
};
return (
    <div>
        <>
            <div className="container">
                <DataTable
                    value={data}
                    showGridlines
                    stripedRows
                    paginator
                    rows={5}
                    filterLocale="ES"
                    rowsPerPageOptions={[5, 10, 25, 50]}
                    tableStyle={{ minWidth: "60rem" }}
                    emptyMessage="No existen registros."
                >
                    <Column sortable field="title" header="Título"></Column>

                    <Column sortable field="description" header="Descripción"></Column>

                    <Column
                        field="thumbnail"
                        header="Detalle"
                        body={HtmlContent}
                    ></Column>
                </DataTable>
            </div>
        </>

        <div className="card flex justify-content-center">
            <Dialog
                header="Canal"
                visible={visible}
            >

```

```

        style={{ width: "60vw" }}
        onHide={() => {
          if (!visible) return;
          setVisible(false);
        }}
      >
      <p className="m-0">{channel.title}</p>
      <img
        src={channel.thumbnail?.url}
        alt="Image"
        width="150"
        style={{ borderRadius: "100%", border: "5px solid black" }}
      />
      <br />
      <br />
      <Panel header="Descripción">
        <p className="m-0">{channel.description}</p>
      </Panel>

      <Panel header="Lista de reproducción">
        <div className="container">
          <DataTable
            value={channel.playlists}
            showGridlines
            stripedRows
            paginator
            rows={5}
            filterLocale="ES"
            rowsPerPageOptions={[5, 10, 25, 50]}
            tableStyle={{ minWidth: "10rem" }}
            emptyMessage="No existen registros."
          >
            <Column sortable field="title" header="Título"></Column>

            <Column
              field="player"
              style={{ textAlign: "center" }}
              header="Lista de reproducción"
              body={VideoContent}
            ></Column>
          </DataTable>
        </div>
      </Panel>
    </Dialog>
  </div>
</div>
);
};

export default SearchPlaylistComponent;

```

Código Fuente 5.120: Componente SearchPlaylist

8. Nuevamente se debe crear un componente que permita visualizar el contenido de los videos que se van a buscar. Dentro de la carpeta **components** se debe crear un componente llamado **SearchVideoComponent**; ver código fuente 5.121.

```
import React from "react";
import { DataTable } from "primereact/datatable";
import { Column } from "primereact/column";

const SearchVideoComponent = ({ data }) => {
  const HtmlContent = (value) => {
    return (
      <a
        href={value.videos[0].player}
        target="_blank"
        rel="noopener noreferrer"
        className="p-button"
        style={{textDecoration: 'none'}}
      >
        <i className="pi pi-check" style={{ fontSize: '1rem' }}></i>
      </a>
    );
  };

  return (
    <div>
      <>
        <div className="container">
          <DataTable
            value={data}
            showGridlines
            stripedRows
            paginator
            rows={5}
            filterLocale="ES"
            rowsPerPageOptions={[5, 10, 25, 50]}
            tableStyle={{ minWidth: "60rem" }}
            emptyMessage="No existen registros."
          >
            <Column sortable field="title" header="Título"></Column>

            <Column sortable field="description" header="Descripción"></Column>

            <Column
              field="thumbnail"
              header="Video"
              body={HtmlContent}
            ></Column>
          </DataTable>
        </div>
      </>
    </div>
  );
};
```

```
        </div>
      </>
    </div>
  );
};

export default SearchVideoComponent;
```

Código Fuente 5.121: Componente SearchVideo

9. Dentro del archivo **App.js** hay que importar los servicios para llamar a las respectivas consultas que se crearon. Se deben importar las dependencias de **useState** y **useLazyQuery**; ver código fuente 5.122.

```
const [visible, setVisible] = useState(false);
const [searchTerm, setSearchTerm] = useState("");
const [searchOption, setSearchOption] = useState("Videos Populares");
const [getVideos, result] = useLazyQuery(GET_SEARCH_VIDEOS);
const [getChannels, channelList] = useLazyQuery(GET_SEARCH_CHANNELS);
```

Código Fuente 5.122: Creación de servicios para el componente principal

10. Para mostrar el contenido de las consultas se deben utilizar los siguientes métodos; ver código fuente 5.123.

```
const showVideos = () => {
  getVideos({ variables: { filter: searchTerm } })
};

const showChannels = () => {
  getChannels({ variables: { filter: searchTerm } })
};
```

Código Fuente 5.123: Métodos para mostrar información

11. Se debe crear un método principal para mostrar la información según el tipo de contenido; ver código fuente 5.124.

```
const searchResources = () => {
  if (searchOption === "Videos") {
    showVideos();
    setSearchTerm("");
  } else if (searchOption === "Canales") {
    showChannels();
    setSearchTerm("");
  } else {
    setSearchOption("Videos Populares")
    setSearchTerm("")
  }
};
```

Código Fuente 5.124: Método principal para mostrar información

- Para definir la estructura HTML del formulario dentro del componente, es importante destacar que esta se encuentra dentro de la palabra clave `return`. Es en este bloque donde se inserta el código HTML que React utiliza para renderizar los componentes creados.

A continuación, se muestra el código completo del componente principal, el cual incluye la estructura HTML del formulario; ver código fuente 5.125.

```
const startContent = (
  <React.Fragment>
    <i
      className="pi pi-youtube"
      style={{ fontSize: "2rem", color: "red", marginTop: "-24px" }}
    ></i>
  </React.Fragment>
);

const centerContent = (
  <IconField iconPosition="left" style={{ marginTop: "-15px" }}>
    <InputIcon className="pi pi-search" onClick={searchResources}/>
    <InputText
      placeholder="Search"
      size={80}
      value={searchTerm}
      onChange={(e) => setSearchTerm(e.target.value)}
    />
  </IconField>
);

const endContent = (
  <React.Fragment>
    <Button
      text
      icon="pi pi-bars"
      onClick={() => setVisible(true)}
      style={{ color: "black", marginTop: "-14px" }}
    ></Button>
  </React.Fragment>
);

return (
  <div className="App">
    <div className="card">
      <Toolbar
        start={startContent}
        center={centerContent}
        end={endContent}
        style={{ height: 60 }}
      />
    </div>
  <br />
);
```

```

<Panel header={searchOption}>

  {searchOption === "Videos Populares"
  && <PopularVideoComponent></PopularVideoComponent>}
  {searchOption === "Videos"
  && <SearchVideoComponent
  data={result.data?.SearchVideos} ></SearchVideoComponent>}
  {searchOption === "Canales"
  && <SearchPlaylistComponent
  data={channelList.data?.SearchChannels} ></SearchPlaylistComponent>}

</Panel>

<Sidebar
  visible={visible}
  onHide={() => setVisible(false)}
  position="right"
>
  <h2>
    {" "}
    <i
      className="pi pi-youtube"
      style={{ fontSize: "1.5rem", color: "red", marginRight: "5px" }}
    ></i>
    Youtube
  </h2>
  <Button
    icon="pi pi-youtube"
    label="Videos Populares"
    severity="secondary"
    text
    onClick={()=>{setSearchOption("Videos Populares")}}
  />
  <Button
    icon="pi pi-youtube"
    label="Videos"
    severity="secondary"
    text
    onClick={()=>{setSearchOption("Videos")}}/>
  <br />
  <Button
    icon="pi pi-youtube"
    label="Canales"
    severity="secondary"
    text
    onClick={()=>{setSearchOption("Canales")}}
  />
</Sidebar>
</div>
);

```

```
}
```

Código Fuente 5.125: Estructura HTML del componente principal

- Definir los estilos que se usarán para el componente principal dentro del archivo **index.css**; ver código fuente 5.126.

```
body {
  margin: 0;
  font-family: -apple-system
  , BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
  'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
  sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
  monospace;
}
```

Código Fuente 5.126: Aplicar estilos al componente principal

En la Fig. 5.35 se puede observar la vista principal de la aplicación.

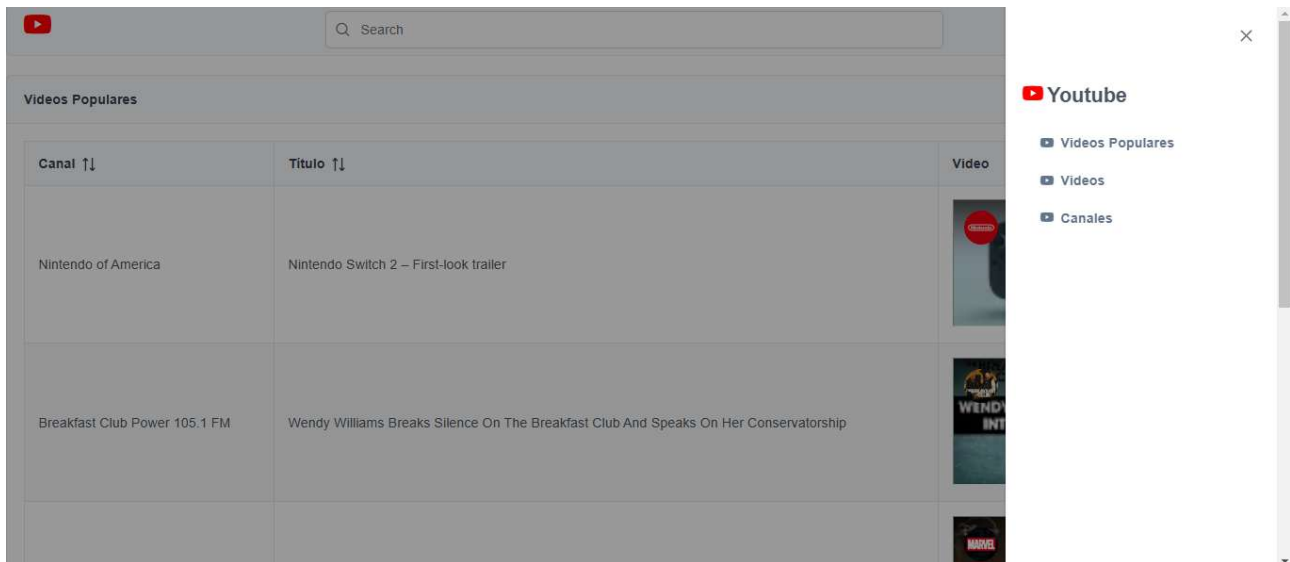


Fig. 5.35. Vista Principal

A continuación, se realiza la búsqueda de videos relacionados con la palabra **java** como se aprecia en Fig. 5.36.

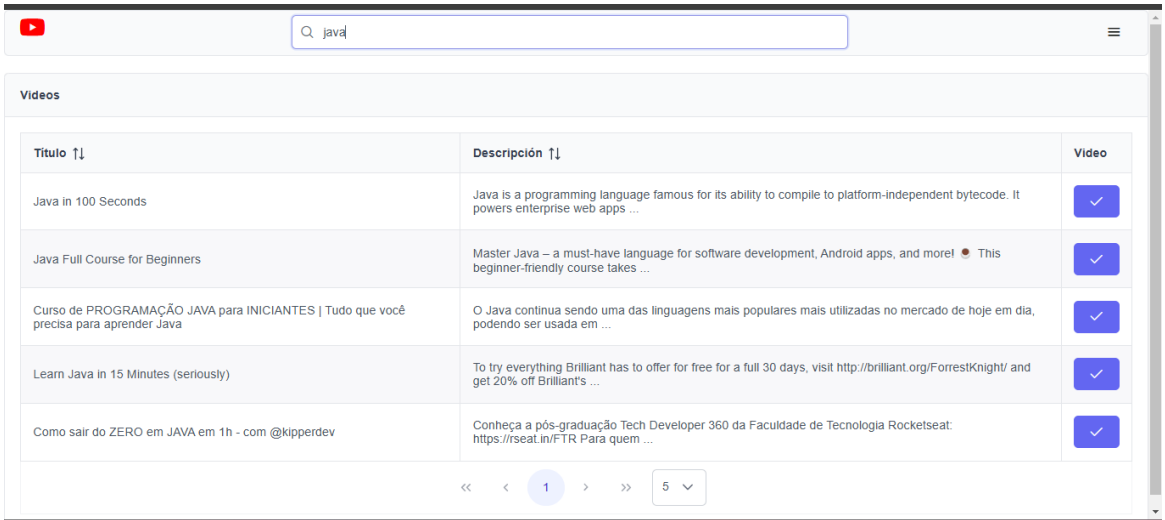


Fig. 5.36. Búsqueda de videos

A continuación, se realizará la búsqueda de canales relacionados con la palabra **mr beast** como se aprecia en la Fig. 5.37.

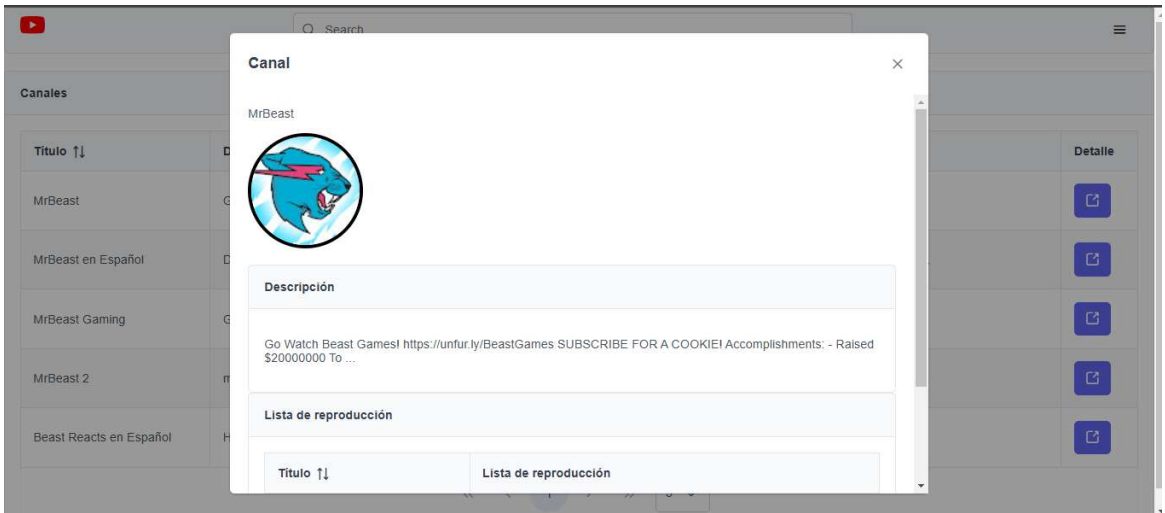


Fig. 5.37. Búsqueda de canales

A continuación se adjunta los enlaces de los repositorios de Github del ejercicio realizado:

Enlace del repositorio backend

<https://github.com/antonio-quina/backend-cap-05-ejercicio-07.git>

Enlace del repositorio frontend

<https://github.com/antonio-quina/frontend-cap-05-ejercicio-07.git>

GRAPHQL EN LA NUBE

La computación en la nube ha transformado significativamente el desarrollo de software, ofreciendo a las organizaciones acceso a recursos informáticos bajo demanda, como servidores, almacenamiento y bases de datos, sin la necesidad de gestionar infraestructura física propia [59]. Esta modalidad permite a los equipos de desarrollo trabajar con mayor agilidad y velocidad, propiciando la innovación y mejorando la eficiencia [60].

Además, la nube facilita la colaboración entre equipos distribuidos geográficamente, ya que proporciona plataformas que permiten compartir código fuente en tiempo real, haciendo que la codificación colaborativa sea más fluida y eficiente.

La escalabilidad es otra ventaja clave, ya que las plataformas en la nube permiten ajustar los recursos según las necesidades del proyecto, garantizando un rendimiento óptimo y una rentabilidad al asignar recursos de manera precisa.

Hasura

Hasura es una plataforma en la nube que permite a los desarrolladores construir, desplegar y gestionar aplicaciones modernas basadas en APIs GraphQL y REST de manera rápida y eficiente. La plataforma se centra en simplificar el desarrollo backend, permitiendo a los equipos concentrarse en la lógica de negocio y la experiencia del usuario, mientras automatiza tareas repetitivas como la gestión de bases de datos, la autenticación y la autorización [61].

Características principales

GraphQL como servicio principal. Hasura¹² proporciona una capa de GraphQL sobre bases de datos existentes, como PostgreSQL, MySQL, SQL Server y BigQuery. Esto permite a los desarrolladores consultar y manipular datos de manera eficiente sin necesidad de escribir código backend adicional. Ofrece capacidades de suscripción en tiempo real, lo que permite a las aplicaciones recibir actualizaciones instantáneas cuando los datos cambian.

Integración con bases de datos. Hasura se conecta directamente a bases de datos relacionales y no relacionales, generando automáticamente un esquema GraphQL basado en la estructura de la base de datos. Soporta operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y permite la personalización de consultas y mutaciones.

Autenticación y autorización. La plataforma integra sistemas de autenticación externos (como Auth0, Firebase o soluciones personalizadas) y proporciona un sistema de control de acceso basado en roles

¹²Hasura: <https://hasura.io/>

(RBAC). Los permisos se pueden configurar a nivel de tabla, columna o fila, lo que permite un control granular sobre quién puede acceder a qué datos.

Generación de APIs GraphQL mediante Hasura

Para realizar la generación de APIs con Hasura se deben seguir los siguientes pasos:

1. Ingresar a la plataforma de Hasura como se observa en la Fig. 5.38, es necesario crear una cuenta.

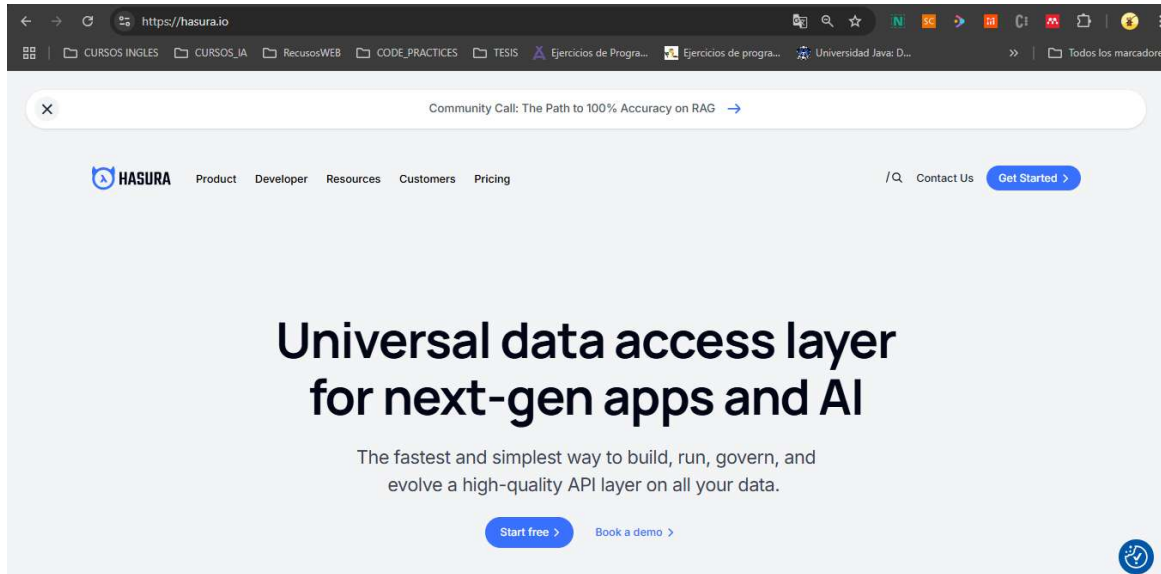


Fig. 5.38. Plataforma Hasura

2. En el panel de configuración se debe seleccionar la opción que aparece en la Fig. 5.39.

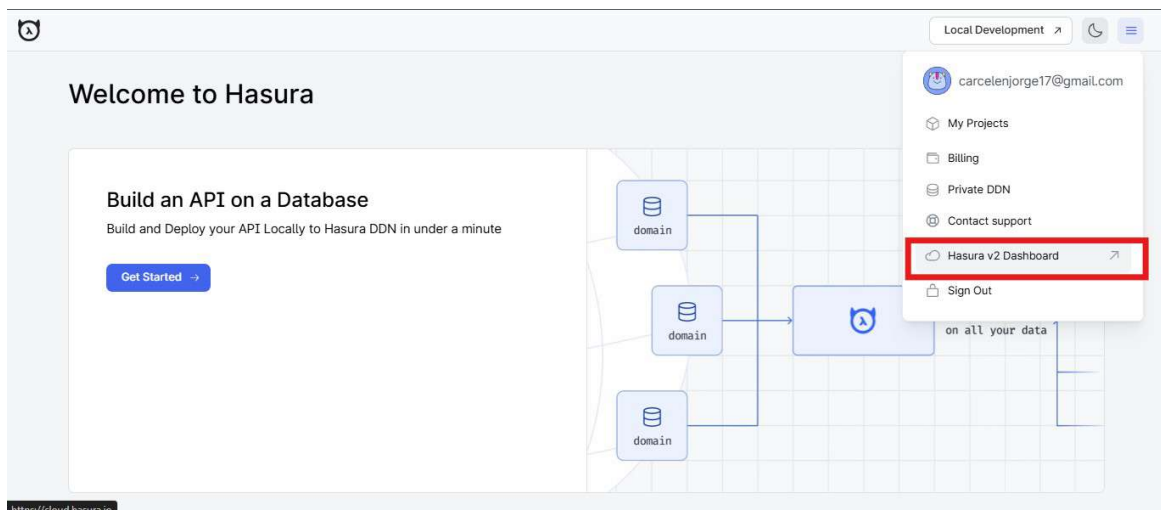


Fig. 5.39. Vista Plataforma Hasura

3. Para crear un nuevo proyecto hay que seleccionar la opción proyecto nuevo; ver la Fig. 5.40.

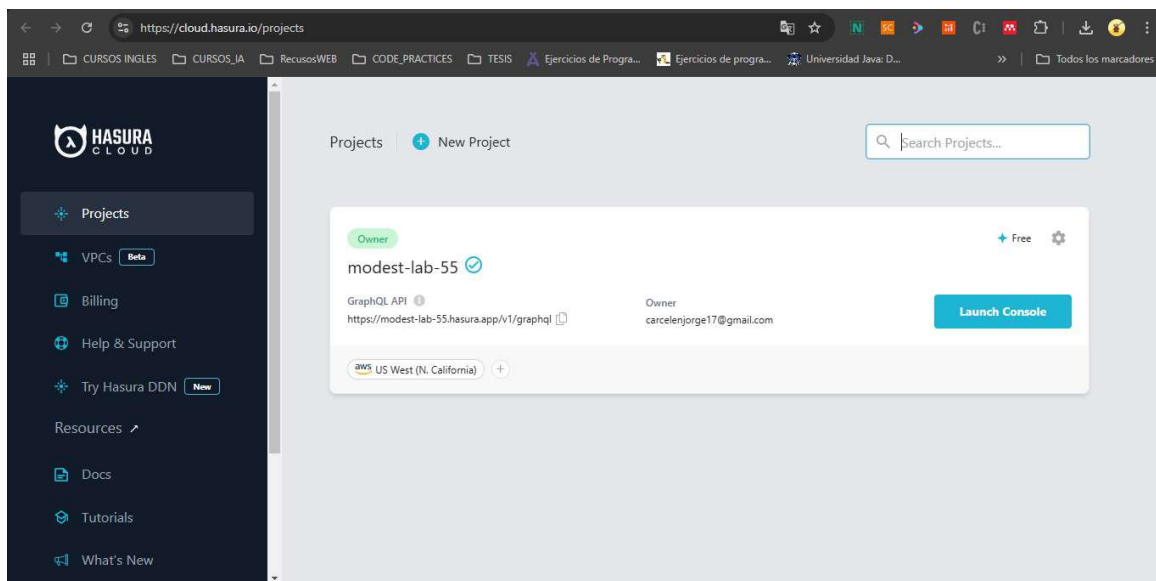


Fig. 5.40. Consola Hasura

4. En la Fig. 5.41 se visualiza el proyecto creado.

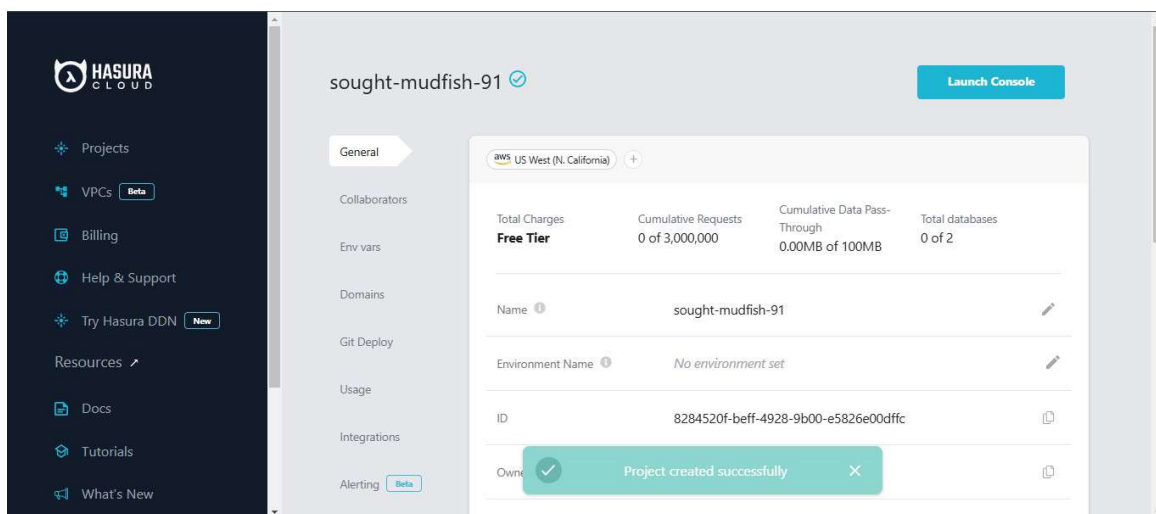


Fig. 5.41. Proyecto Creado

5. Antes de continuar, es necesario desplegar la base de datos creada en la nube para hacer la conexión con el proyecto creado. Para crear bases de datos en la nube existen distintas alternativas como Amazon Web Services, HeroKu, RailWay, Azure entre otros. Por el momento se utilizará **ElephantSQL** debido a que es un servicio fácil sin embargo se puede utilizar cualquiera de las mencionadas anteriormente; ver la Fig. 5.42.

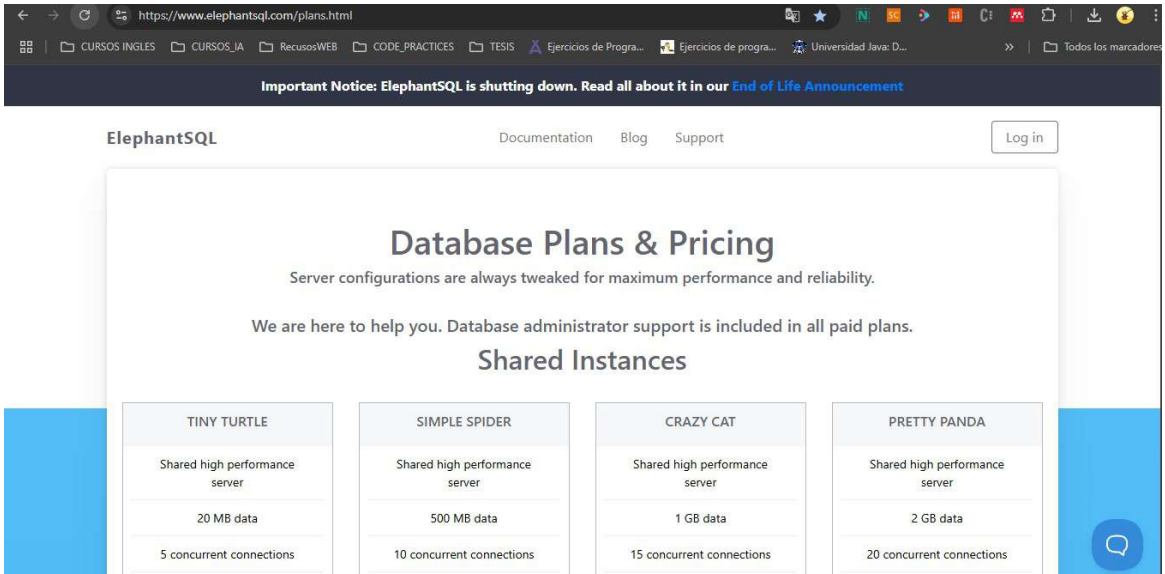


Fig. 5.42. ElephantSQL

6. Posteriormente es necesario iniciar sesión y crear una nueva instancia como se aprecia en la Fig. 5.43. El proceso de creación es sencillo, solo es importante que se sigan cada una de las indicaciones y que al final se guarden las credenciales de la base de datos en un lugar seguro.

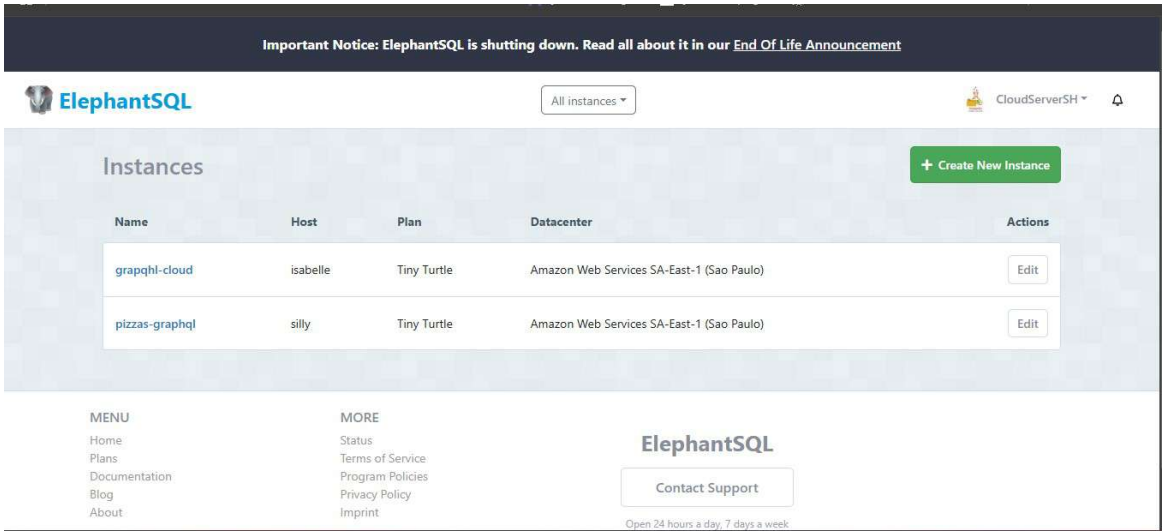


Fig. 5.43. Creación base de datos

7. Como se puede observar en la Fig. 5.44 se destaca toda la información de la base de datos creada.

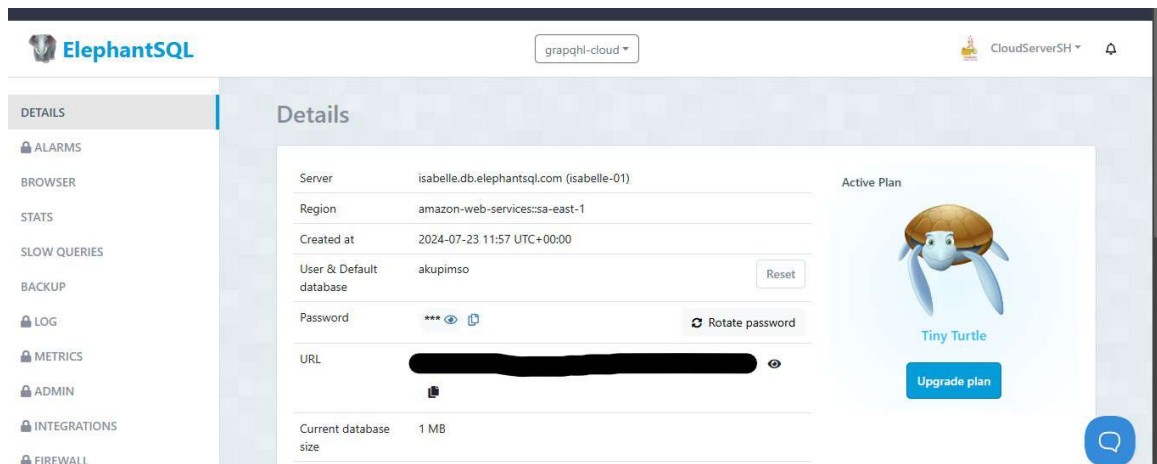


Fig. 5.44. Información base de datos

8. Para la conexión se debe seleccionar la base de datos, en este caso postgresSQL y seleccionar la opción conectarse a una base de datos existente; ver la Fig. 5.45.

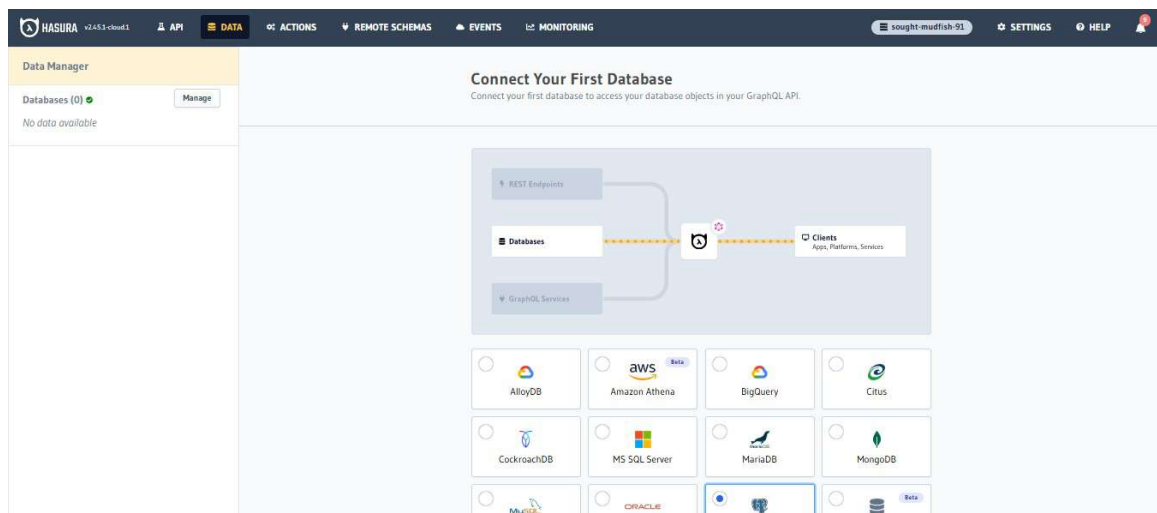


Fig. 5.45. Conexión a la base de datos

9. Posteriormente se escoge el método de conexión y se ingresan los valores de las credenciales como se aprecia en la Fig. 5.46.

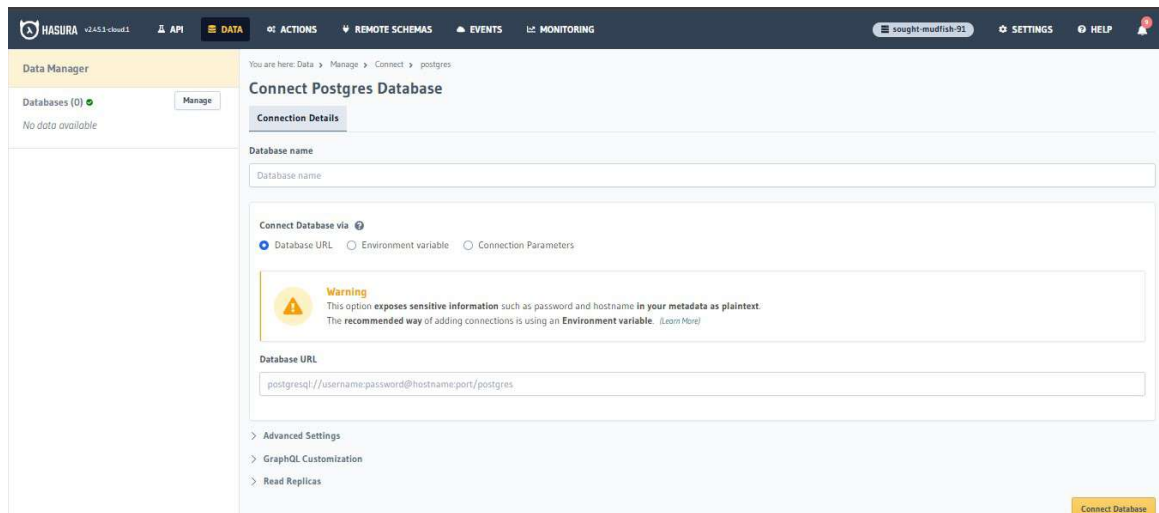


Fig. 5.46. Cambio de credenciales en la base de datos

10. Una vez creada la conexión, se deben escoger todas entidades y relaciones de la base de datos con la opción **track all** para realizar las consultas y mutaciones con la API que se va generar; ver la Fig. 5.47.

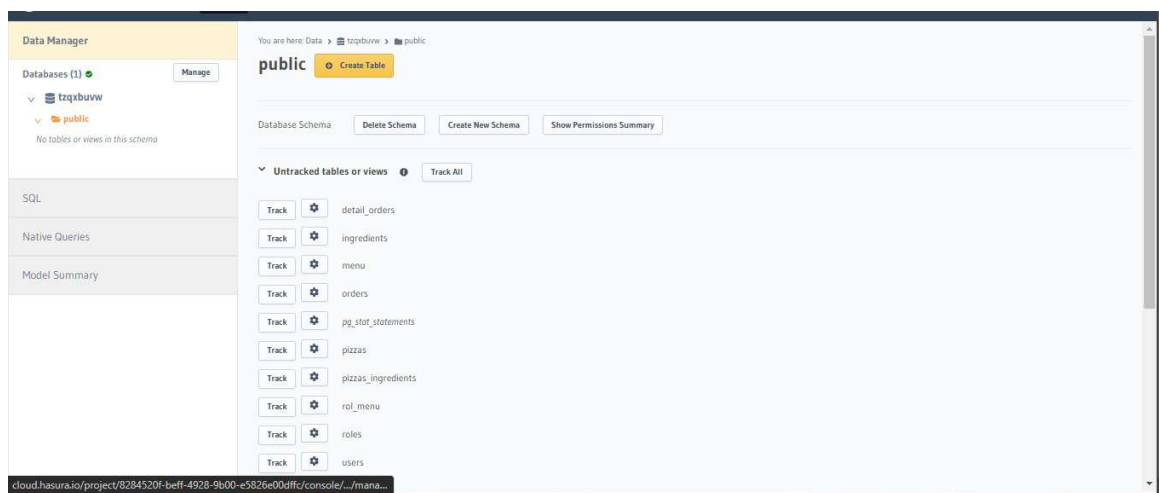


Fig. 5.47. Seleccionar entidades de la base de datos

11. Para probar el API con sus respectivas mutaciones y consultas hay que dirigirse hacia la pestaña API, en este caso en la ilustración se realiza una consulta para mostrar las pizzas registradas; ver la Fig. 5.48.

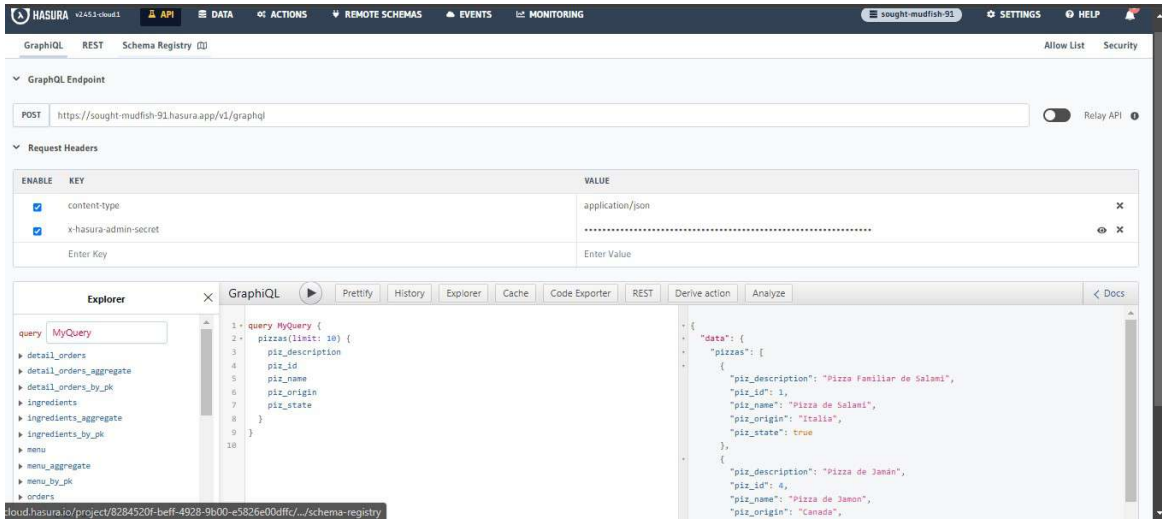


Fig. 5.48. Prueba de la API creada en Hasura

Cabe mencionar que una vez que la API ha sido creada en Hasura, se puede probar con otro tipo de clientes como Postman para lo cual se deben seguir los siguientes pasos.

12. Para probar el API con sus respectivas mutaciones y consultas, hay colocar los valores de los **headers** como se observa en la Fig. 5.49. En el caso de la URL se debe utilizar la que Hasura genera por defecto.

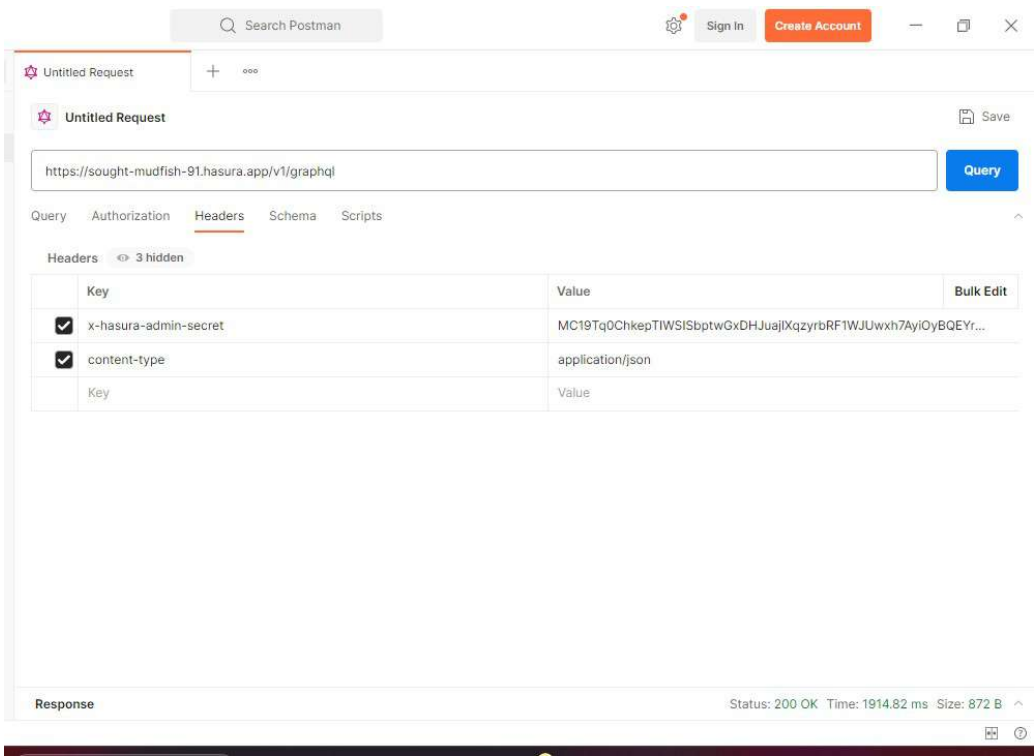


Fig. 5.49. Prueba de la API en Postman

13. Una vez ejecutada la consulta, se pueden observar los siguientes resultados; ver la Fig. 5.50.

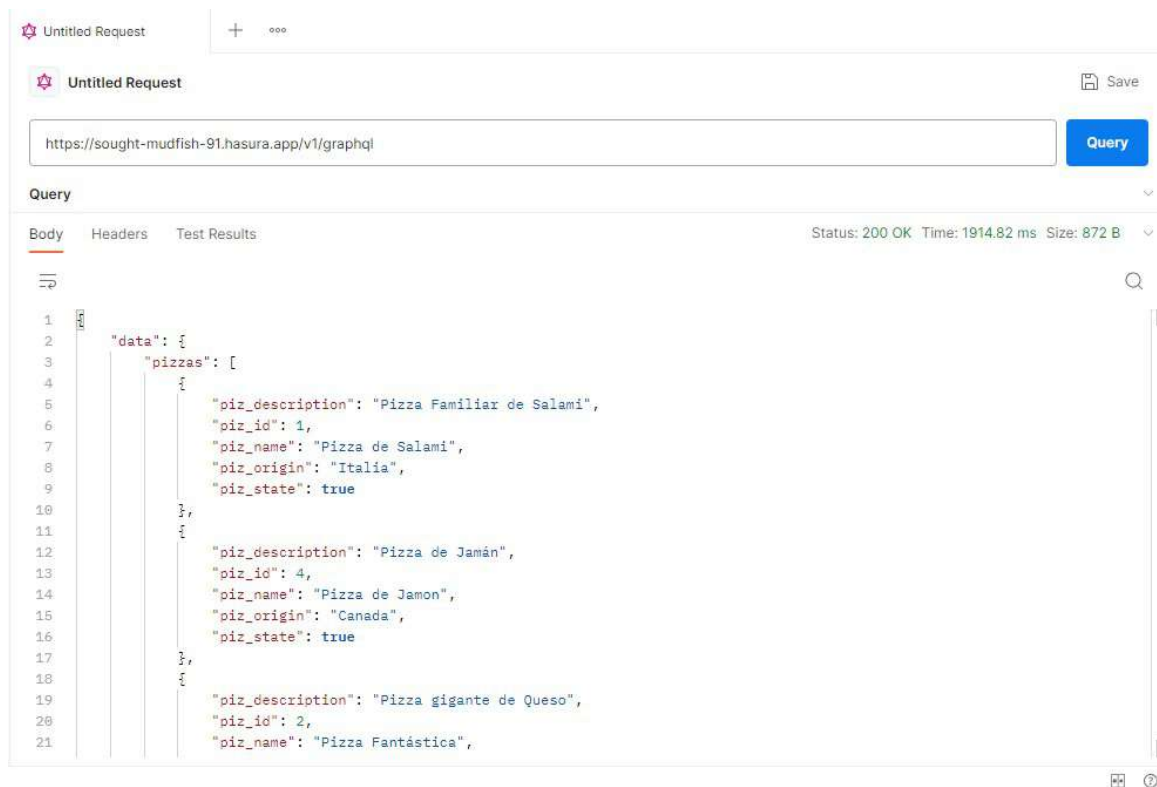


Fig. 5.50. Resultados de la consulta de la API en Postman

RESUMEN

Las actividades realizadas a lo largo de este capítulo permiten replicar escenarios reales de desarrollo, permitiendo que el aprendizaje trascienda la teoría y se enfoque en la experiencia directa con tecnologías y patrones utilizados en entornos profesionales.

Entre las actividades propuestas se incluye la integración de ORMs y la configuración de autenticación mediante JWT, lo que permite al lector comprender cómo manejar la persistencia de datos y la seguridad en aplicaciones modernas. Asimismo, se abordan la implementación de menús dinámicos, la generación de reportes y el consumo de APIs REST dentro de un entorno GraphQL, demostrando la versatilidad de este enfoque en distintos contextos operativos.

Este capítulo introduce herramientas en la nube, como Hasura, que facilitan la creación automática de servicios a partir de bases de datos existentes, acelerando el desarrollo y permitiendo centrarse en la lógica de negocio y la experiencia del usuario. Esta incorporación refuerza la idea de que el aprendizaje de GraphQL no se limita a conceptos locales, sino que puede aplicarse en arquitecturas modernas y escalables que combinan servicios en la nube y prácticas de integración continua.

Finalmente, se reafirma que el dominio de GraphQL se fortalece mediante la práctica constante y la experimentación en proyectos que emulan entornos profesionales. La combinación de retos teóricos y prácticos prepara al lector para enfrentar escenarios reales de desarrollo, consolidando competencias en diseño, implementación y optimización de APIs, así como en la construcción de aplicaciones FullStack eficientes, seguras y escalables.

ACTIVIDADES

Resuelva el siguiente ejercicio

Utilizando el proyecto anterior construya un pequeño módulo para gestionar los **pedidos de pizzas**, incluyendo creación de entidades, componentes, roles de usuario y funcionalidades de CRUD, siguiendo estas indicaciones:

1. Backend – Entidades

- Cree una **entidad Pedido** con los siguientes campos:
 - **id**: Identificador único del pedido.
 - **usr_id**: Identificador único del usuario.
 - **descripcion**: Descripción del pedido.
 - **fecha_pedido**: Fecha en la que se realiza el pedido.
 - **estado**: Estado del pedido, con los valores posibles:
 - **P** – Pendiente
 - **C** – Cancelado
 - **R** – Recibido

Nota: Si un pedido está en estado R, el cliente **no podrá cancelarlo**. Solo un usuario administrador puede cambiar el estado a R.
- Cree una **entidad DetallePedido** para registrar las pizzas asociadas a cada pedido, con los siguientes campos:
 - **id_detalle**: Identificador único del detalle.
 - **id_pedido**: Relación con el pedido correspondiente.
 - **piz_id**: Identificador de la pizza solicitada.
 - **cantidad**: Cantidad de pizzas de ese tipo.
 - **fecha**: Fecha en la que se agregó el detalle al pedido.

2. Frontend – Componente de pedidos

- Cree un **nuevo componente** para mostrar los pedidos del cliente, incluyendo sus detalles.
- Permita realizar operaciones de **CRUD** sobre los pedidos y sus detalles: crear, leer, actualizar y eliminar.
- Incluya la funcionalidad para que el **cliente pueda cancelar sus pedidos**, respetando la restricción de que un pedido **recibido no puede ser cancelado**.
- Agregue una opción para que **el usuario administrador** pueda cambiar el estado de un pedido a R (Recibido).
- Incluya un **botón para generar el reporte del pedido**, mostrando la información del pedido, su detalle y el total.

3. Roles y seguridad

- Cree un **nuevo rol de usuario** que tenga acceso exclusivo a las funcionalidades descritas anteriormente.
- Asegúrese de que solo los usuarios con el rol adecuado puedan:
 - Crear, modificar o cancelar pedidos.
 - Consultar los pedidos y generar reportes.
 - Cambiar el estado a R si son administradores.

Bibliografía

- [1] H. Muccini y K. Vaidhyanathan, «Software Architecture for ML-based Systems: What Exists and What Lies Ahead,» en *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, 2021, págs. 121-128. DOI: 10.1109/WAIN52551.2021.00026.
- [2] H. Cervantes y R. Kazman, *Designing Software Architectures, A Practical Approach* (SEI Series in Software Engineering). Addison-Wesley, 2024, ISBN: 978-0138108021. dirección: <https://books.google.com.ec/books?id=ERnLzwEACAAJ>.
- [3] R. Raymond y S. M. A. Savarimuthu, «Software Design Patterns and Architecture Patterns –A Study Explored,» en *2022 5th International Conference on Contemporary Computing and Informatics (IC3I)*, 2022, págs. 1998-2006. DOI: 10.1109/IC3I56241.2022.10073279.
- [4] D. Sobhy, R. Bahsoon, L. Minku y R. Kazman, *Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review*, jul. de 2021. DOI: 10.1145/3464305.
- [5] L. Bass, P. Clements y R. Kazman, *Software Architecture in Practice*, 4th. Addison-Wesley, 2021, ISBN: 978-0-321-81573-6.
- [6] O. I. D. G. Alvarez, N. I. Layedra y V. R. III, «Análisis comparativo de Patrones de Diseño de Software,» vol. 7, págs. 2145-2565, 7 2022. DOI: 10.23857/pc.v7i7. dirección: <http://polodelconocimiento.com/ojs/index.php/eshttps://orcid.org/0000-0002-7245-5640>.
- [7] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., 2022, ISBN: 9781491925409.
- [8] Z. M. Rodríguez, L. D. P. Rodríguez y J. C. G. Suarez, «Arquitectura basada en Microservicios y DevOps para una ingeniería de software continua,» *Industrial Data*, vol. 23, n.º 2, págs. 141-149, 2020. DOI: 10.15381/idata.v23i2.17278. dirección: <https://www.redalyc.org/articulo.oa?id=81665362014>.
- [9] Amazon Web Services, *Microservicios en AWS*, 2024. dirección: <https://aws.amazon.com/es/microservices/>.
- [10] A. Taivalsaari, T. Mikkonen, C. Pautasso y K. Systä, «Full Stack Is Not What It Used to Be,» en *Web Engineering*, M. Brambilla, R. Chbeir, F. Frasincar e I. Manolescu, eds., Cham: Springer International Publishing, 2021, págs. 363-371, ISBN: 978-3-030-74296-6.
- [11] Amazon Web Services, *La diferencia entre Frontend y Backend*, 2024. dirección: <https://aws.amazon.com/es/compare/the-difference-between-frontend-and-backend/>.
- [12] RedHat, *¿Qué es una API y cómo funciona?* 2023. dirección: <https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces#historia-de-las-api>.
- [13] IBM, «API,» 2024. dirección: <https://www.ibm.com/mx-es/topics/api>.
- [14] A. Quiña-Mera, P. Fernandez, J. M. García y A. Ruiz-Cortés, «GraphQL: A Systematic Mapping Study,» *ACM Comput. Surv.*, vol. 55, n.º 10, feb. de 2023, ISSN: 0360-0300. DOI: 10.1145/3561818. dirección: <https://doi.org/10.1145/3561818>.
- [15] IBM, *¿Qué es una API REST?* 2023. dirección: <https://www.ibm.com/mx-es/topics/rest-apis>.
- [16] AWS, *¿Qué es una API REST?* 2023. dirección: <https://aws.amazon.com/es/what-is/restful-api/#:~:text=La%20API%20RESTful%20es%20una,llevar%20a%20cabo%20varias%20tareass..>

- [17] AWS, *¿Cuál es la diferencia entre GraphQL y REST?* 2023. dirección: <https://aws.amazon.com/es/compare/the-difference-between-graphql-and-rest/#:~:text=respuestas%20de%20API-,Diferencias%20clave%3A%20comparaci%C3%B3n%20entre%20GraphQL%20y%20REST,y%20un%20conjunto%20de%20herramientas..>
- [18] A. Quiña-Mera, J. M. García, P. Fernández, P. Vega-Molina y A. Ruiz-Cortés, «GraphQL or REST for Mobile Applications?» En *Advanced Research in Technologies, Information, Innovation and Sustainability*, T. Guarda, F. Portela y M. F. Augusto, eds., Cham: Springer Nature Switzerland, 2022, págs. 16-30, ISBN: 978-3-031-20319-0. DOI: 10.1007/978-3-031-20319-0_2. dirección: https://doi.org/10.1007/978-3-031-20319-0_2.
- [19] A. Quiña-Mera, C. Guevara-Vega, J. Caiza, J. Mise y P. Landeta, «REST, GraphQL, and GraphQL Wrapper APIs Evaluation. A Computational Laboratory Experiment,» en *Proceedings of International Conference on Information Technology and Applications*, Springer, 2023, págs. 397-407. DOI: 10.1007/978-981-19-9331-2_34. dirección: https://link.springer.com/chapter/10.1007/978-981-19-9331-2_34.
- [20] A. Quiña-Mera, Z. M. G. de la Cruz y C. Guevara-Vega, «Efficiency study of GraphQL and REST Microservices in Docker containers: A computational experiment,» *Data and Metadata*, vol. 4, pág. 199, 2025. DOI: 10.56294/dm2025199. dirección: <https://doi.org/10.56294/dm2025199>.
- [21] Meta, *The GraphQL Foundation. GraphQL*, 2025. dirección: <https://spec.graphql.org/>.
- [22] T. G. Foundation, *GraphQL*, 2024. dirección: <https://graphql.org/>.
- [23] X. Huang, «Research and Application of Node.js Core Technology,» *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, 2020. DOI: 10.1109/IC. dirección: <https://doi.org/10.1109/IC>.
- [24] B. Basumatary y N. Agnihotri, «Benefits and Challenges of Using Node.js,» *International Journal of Innovative Research in Computer Science & Technology*, vol. 10, n.º 3, págs. 67-70, mayo de 2022. DOI: 10.55524/ijircst.2022.10.3.13.
- [25] R. Jin, R. Cordingly, D. Zhao y W. Lloyd, «GraphQL vs. REST: A Performance and Cost Investigation for Serverless Applications,» *Proceedings of the ACM Symposium on Cloud Computing*, 2024. DOI: 10.1145/3702634. dirección: <https://doi.org/10.1145/3702634>. 3702956.
- [26] A. Makris, K. Tserpes, G. Spiliopoulos, D. Zissis y D. Anagnostopoulos, «MongoDB Vs PostgreSQL: A comparative study on performance aspects,» *Geoinformatica*, vol. 25, págs. 243-268, 2021. DOI: 10.1007/s10707-020-00407-w.
- [27] D. Flanagan, *JavaScript: The Definitive Guide*, 7th. O'Reilly Media, 2020, ISBN: 978-1491952023.
- [28] Meta Open Source, *React - Biblioteca para interfaces de usuario*, 2025. dirección: <https://es.react.dev/>.
- [29] E. Y. et al., *Vite - Next Generation Frontend Tooling*, Consultado el 18 de marzo de 2025, 2025. dirección: <https://vitejs.dev/>.
- [30] Meta Open Source, *React Router Documentation*, 2025. dirección: <https://reactrouter.com/>.
- [31] SweetAlert2, *SweetAlert2 Documentation*, <https://sweetalert2.github.io/>, 2025.
- [32] SweetAlert2, *sweetalert2-react-content - NPM*, 2025. dirección: <https://www.npmjs.com/package/sweetalert2-react-content>.
- [33] FontAwesome, *Font Awesome Free*, 2025. dirección: <https://fontawesome.com>.
- [34] P. Michail y C. Kalloniatis, «Object Relational Mapping vs. Event-Sourcing: A Systematic Review,» en *Electronic Government and the Information Systems Perspective*, A. Kö, E. Francesconi, G. Kotsis, A. M. Tjoa e I. Khalil, eds., Cham: Springer International Publishing, 2022, págs. 18-31, ISBN: 978-3-031-12673-4. DOI: 10.1007/978-3-031-12673-4_2.
- [35] C. Xia, G. Yu y M. Tang, «Efficient Implement of ORM (Object/Relational Mapping) Use in J2EE Framework: Hibernate,» en *2009 International Conference on Computational Intelligence and Software Engineering*, 2009, págs. 1-3. DOI: 10.1109/CISE.2009.5365905.
- [36] T. Sotiropoulos, S. Chaliasos, V. Atlidakis, D. Mitropoulos y D. Spinellis, «Data-Oriented Differential Testing of Object-Relational Mapping Systems,» en *2021 IEEE/ACM 43rd International Conference*

on *Software Engineering (ICSE)*, IEEE, 2021, págs. 1535-1547. DOI: 10.1109/ICSE43902.2021.00137.

- [37] Sequelize, *Sequelize*, <https://sequelize.org/>, 2025.
- [38] Sequelize Contributors, *Sequelize CLI — Sequelize*, 2025. dirección: <https://sequelize.org/docs/v7/cli/> (visitado 20-02-2025).
- [39] S. Buna, *GraphQL in Action*. Shelter Island, NY: Manning Publications, 2021, ISBN: 9781617295683. dirección: <https://www.manning.com/books/graphql-in-action>.
- [40] M. Jones, J. Bradley y N. Sakimura, «RFC 7519: JSON Web Token (JWT)», USA: RFC Editor, 2015.
- [41] BBVA Labs, «JSON Web Tokens (JWT): claves para usarlos de manera segura», BBVA, 2020, Disponible en: <https://www.bbva.com/es/innovacion/json-web-tokens-jwt-claves-para-usarlos-de-manera-segura/>.
- [42] G. Chesterfield, *Securing APIs with JSON Web Tokens (JWT) and OAuth 2.0*. Amazon, 2024, Disponible en: <https://www.amazon.com/-/es/Greyson-Chesterfield-ebook/dp/B0D7N5W6SX>.
- [43] Caktus Group, «JWT Authentication: How Does it Work?» *Caktus Group Blog*, 2020, Disponible en: <https://www.caktusgroup.com/blog/2020/08/21/jwt-authentication-how-does-it-work/>.
- [44] dcodeIO, *bcryptjs*, Disponible en: <https://www.npmjs.com/package/bcryptjs>, 2015.
- [45] auth0, *jsonwebtoken*, Disponible en: <https://www.npmjs.com/package/jsonwebtoken>, 2023.
- [46] P. Informatics, *PrimeReact*, Disponible en: <https://primereact.org/>, 2023.
- [47] P. Informatics, *PrimeIcons*, Disponible en: <https://www.npmjs.com/package/primeicons>, 2023.
- [48] P. Informatics, *PrimeFlex*, Disponible en: <https://www.npmjs.com/package/primeflex>, 2023.
- [49] Parallax, *jsPDF Documentation*, Available online, Accessed: 2025, 2025. dirección: <https://parallax.com/products/jspdf>.
- [50] SheetJS, *SheetJS Community Edition Documentation*, <https://docs.sheetjs.com/>, Accessed: 2025, 2025.
- [51] O. Aldama López, M. Delgado Fernández y M. Díaz-Canel Bermúdez, «Metodología de los tableros y cuadro de mando integral en la gestión de gobierno orientada a la innovación», *Revista Cubana de Transformación Digital*, vol. 6, n.º 3, e236, 2022. dirección: https://www.researchgate.net/publication/363408710_METODOLOGIA_DE_LOS_TABLEROS_Y_CUADRO_DE_MANDO_INTEGRAL_EN_LA_GESTION_DE_GOBIERNO_ORIENTADA_A_LA_INNOVACION.
- [52] N. Autor, «Diseño de un Dashboard basado en un sistema de incidencias técnicas reportadas en una caja de ahorro y crédito», *Revista Nombre*, vol. X, n.º Y, Z, 2024. dirección: https://www.researchgate.net/publication/384232566_Disenio_de_un_Dashboard_basado_en_un_sistema_de_incidencias_tecnicas_reportadas_en_una_caja_de_ahorro_y_credito.
- [53] F. B. Pincay Ponce, «Dashboard para el control y seguimiento académico de los estudiantes en la Educación Superior», *Revista Científica de Innovación Educativa y Sociedad Actual .ALCON*, vol. 4, n.º 5, págs. 1-13, 2024. dirección: https://www.researchgate.net/publication/384646307_Dashboard_para_el_control_y_seguimiento_academico_de_los_estudiantes_en_la_Educacion_Superior.
- [54] «Smart Cities», en *Proceedings of the 3rd IberoAmerican Congress on Smart Cities (ICSC-CITIES 2020)*, Springer, 2020. dirección: <https://link.springer.com/content/pdf/10.1007/978-3-030-69136-3.pdf>.
- [55] «Applications and Usability of Interactive TV», en *Proceedings of the 17th European Conference on Interactive TV and Video*, Springer, 2019. dirección: <https://link.springer.com/content/pdf/10.1007/978-3-030-81996-5.pdf>.
- [56] G. Cloud, *API de modelos para Gemini en Vertex AI*, 2024. dirección: <https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/gemini?hl=es-419>.
- [57] Chatscope, *Chat UI Kit React*, 2025. dirección: <https://github.com/chatscope/chat-ui-kit-react>.

- [58] Google Help, *¿Qué es YouTube? Primera pregunta hacia el éxito*, Consultado el 23 de diciembre de 2024, 2024. dirección: <https://support.google.com/youtube/community-guide/278093747/%C2%BFqu%C3%A9-es-youtube-primera-pregunta-hacia-el-%C3%A9xito?hl=es>.
- [59] G. Cloud. «Ventajas de la computación en la nube.» Accedido: 15 de enero de 2025. (2025), dirección: <https://cloud.google.com/learn/advantages-of-cloud-computing?hl=es-419>.
- [60] B. Network. «Desarrollo de software en la nube.» Accedido: 15 de enero de 2025. (), dirección: <https://www.bernanetwork.com/desarrollo-de-software-en-la-nube>.
- [61] Hasura, *Hasura Cloud: Plataforma de desarrollo de APIs GraphQL y REST*, Plataforma en la nube para construir, desplegar y gestionar aplicaciones modernas basadas en APIs GraphQL y REST., 2023. dirección: <https://hasura.io>.



GISIA
Grupo de Investigación
**Ingeniería de Software
e Inteligencia Artificial**

**Este libro está elaborado por investigadores pertenecientes a los siguientes
Grupos de Investigación de la Universidad Técnica del Norte:**

Grupo de Investigación de Ciencias en Red (e-CIER)

Grupo de Investigación de Ingeniería de Software e Inteligencia Artificial (GISIA)



COLECCIÓN Computación e Innovación Ingeniería, Software y Transformación Digital

En la actualidad, la innovación y la tecnología se han convertido en pilares fundamentales del desarrollo humano, económico y social. En el ámbito educativo, han transformado las metodologías de enseñanza-aprendizaje, permitiendo el acceso a contenidos interactivos, personalizados y globales. En sectores como la salud, la industria, las finanzas, la agricultura y la gestión pública, las soluciones digitales han mejorado la eficiencia, la sostenibilidad y la calidad de vida. Este contexto exige una formación continua y rigurosa en competencias digitales, pensamiento computacional e innovación tecnológica, de modo que las personas y las organizaciones puedan adaptarse con éxito a los desafíos de la era digital.

Esta colección de libros cumple un rol estratégico en la consolidación del conocimiento científico, técnico y aplicado en las ciencias computacionales y su interacción con otras áreas clave del desarrollo industrial y social. Su objetivo central es proporcionar un cuerpo de conocimiento actualizado, riguroso y multidisciplinario que permita a estudiantes, investigadores, profesionales y tomadores de decisiones comprender y aplicar las últimas tendencias, metodologías y herramientas en el contexto de la cuarta revolución industrial.

Los libros que la integran abordan temas de frontera como inteligencia artificial, computación en la nube, ciencia de datos, ciberseguridad, desarrollo de software ágil, sistemas embebidos, blockchain y transformación digital, entre otros. Asimismo, promueven una visión crítica, ética y sostenible del uso de las tecnologías digitales, destacando su impacto sobre los modelos de negocio, la educación, la gobernanza y la vida cotidiana.

Entre sus principales características se destacan el enfoque transversal e interdisciplinario de los contenidos, la incorporación de estudios de caso reales, la contextualización regional y la articulación entre fundamentos teóricos y aplicaciones prácticas. Los textos están escritos por autores expertos en sus campos, con amplia experiencia en investigación, docencia o práctica profesional, y han sido diseñados para fomentar el aprendizaje autónomo, la actualización continua y el pensamiento crítico. Además, se promueve la inclusión de recursos digitales complementarios, como simuladores, repositorios de código, plataformas interactivas y vínculos a datasets abiertos, que enriquecen la experiencia lectora y formativa.

De tal forma que esta colección de libros constituye una herramienta de gran valor para el fortalecimiento de capacidades técnicas, la generación de soluciones disruptivas y la transformación sostenible de organizaciones, instituciones y territorios. Su enfoque integrador y orientado a la acción la convierte en un recurso indispensable para enfrentar los desafíos contemporáneos y construir futuros más resilientes, eficientes e inclusivos desde la ciencia, la tecnología y la innovación.

VOLÚMENES PUBLICADOS

1

Quiña-Mera, A., Guevara-Vega, C., & Carcelén, J. (2025). *GraphQL: Desarrollo FullStack. Desarrollo web orientado a microservicios con GraphQL*.
Universidad Técnica del Norte.



GraphQL

Construcción eficiente de APIs y aplicaciones modernas

En un mundo digital en constante evolución, GraphQL ha transformado la comunicación entre el frontend y el backend, proporcionando un intercambio de datos más flexible y eficiente. Este libro ofrece una guía completa para el desarrollo FullStack, combinando fundamentos teóricos con implementaciones prácticas, ejemplos aplicados, buenas prácticas y estrategias para una integración efectiva de GraphQL. Desde los conceptos básicos hasta su implementación en arquitecturas basadas en microservicios, los lectores aprenderán a diseñar APIs con Node.js, gestionar bases de datos con Sequelize y fortalecer la seguridad con JWT. Además, se explora la creación de clientes con React, incluyendo el desarrollo de componentes dinámicos, manejo avanzado de tablas, generación de reportes, integración con APIs REST y generación de código mediante herramientas en la nube. Todo ello con el objetivo de optimizar la interacción con los datos y mejorar el rendimiento de las aplicaciones, convirtiéndose en una lectura esencial para quienes buscan construir soluciones modernas, escalables y de alto rendimiento.

