

1ST EDITION

Modern Full-Stack React Projects

Build, maintain, and deploy modern web apps using
MongoDB, Express, React, and Node.js



DANIEL BUGL

Foreword by Matthias Zronek, Senior Software Engineer and Technical Advisor

Proyectos modernos de React de pila completa

Cree, mantenga e implemente aplicaciones web modernas
utilizando MongoDB, Express, React y Node.js

Daniel Bugl



Proyectos modernos de React de pila completa

Derechos de autor © 2024 Packt Publishing

Todos los derechos reservados. Ninguna parte de este libro podrá reproducirse, almacenarse en un sistema de recuperación de datos ni transmitirse en ninguna forma ni por ningún medio sin la autorización previa por escrito del editor, excepto en el caso de citas breves incluidas en artículos críticos o reseñas.

En la preparación de este libro se ha hecho todo lo posible para garantizar la precisión de la información presentada. Sin embargo, la información contenida en este libro se vende sin garantía, ni expresa ni implícita. Ni el autor, ni Packt Publishing ni sus distribuidores serán responsables de ningún daño causado o presuntamente causado, directa o indirectamente, por este libro.

Packt Publishing se ha esforzado por proporcionar información sobre las marcas comerciales de todas las empresas y productos mencionados en este libro mediante el uso adecuado de mayúsculas. Sin embargo, Packt Publishing no puede garantizar la exactitud de esta información.

Gerente de producto del grupo: Rohit Rajkumar

Gerente de productos editoriales: Kushal Dave

Gerente del proyecto del libro: Aishwarya Mohan

Editora senior: Rakhi Patel

Editor técnico: K Bimala Singha

Editor de textos: Sas Editing

Indexador: Pratik Shirodkar

Diseñador de producción: Prashant Ghare

Coordinadores de marketing de DevRel: Anamika Singh y Nivedita Pandey

Primera publicación: junio de 2024

Referencia de producción: 1090524

Publicado por Packt Publishing Ltd.

Casa Grosvenor

11 Plaza de San Pablo

Birmingham

B3 1RB, Reino Unido

ISBN 978-1-83763-795-9

www.packtpub.com

A mi familia y amigos, por apoyarme durante la creación de este libro.

A mis padres, quienes me han apoyado durante toda mi vida.

A mi cofundador, Georg Schelkshorn, y a mi socio comercial, Matthias Zronek, quienes siempre me desafían y continuamente inspiran mi crecimiento. Gracias por ocuparse del negocio mientras escribía este libro.

A mi increíble novia, Junxian Wang, por mejorar mi vida de muchas maneras, por hacerme más productivo y por cuidarme siempre. Te quiero muchísimo.

Sin todos vosotros este libro no hubiera sido posible.

– Daniel Bugl

Colaboradores

Acerca del autor

Daniel Bugl es un desarrollador full-stack, diseñador de productos y emprendedor centrado en tecnologías web. Es licenciado en Ciencias en Informática Empresarial y Sistemas de Información y tiene un Máster en Ciencias de Datos por la Universidad Tecnológica de Viena (TU Wien). Colabora en numerosos proyectos de código abierto y es miembro de la comunidad React. Además, fundó y dirige su propia startup de hardware y software, TouchLay, que ayuda a otras empresas a presentar sus productos y servicios. En su empresa, trabaja constantemente con tecnologías web, especialmente con React y Next.js. En los últimos dos años, junto con Georg Schelkshorn y Matthias Zronek, ha trabajado como asesor técnico y desarrollador full-stack para grandes empresas y el sector público, entre otras tareas, en servicios al ciudadano para el gobierno austriaco.

Quiero agradecer a las personas involucradas en la producción de este libro: a mi cofundador, Georg Schelkshorn; a mi socio comercial, Matthias Zronek; a mi familia y amigos; y a mi novia, Junxian Wang.

Acerca de los revisores

Matthias Zronek es ingeniero de software senior y asesor tecnico con mas de 20 años de experiencia profesional. Le apasiona programar y cada nuevo lenguaje que aprende. Actualmente, se centra en el desarrollo de frameworks web integrales para que los equipos frontend puedan crear aplicaciones web escalables y sostenibles con React y TypeScript. En su tiempo libre, escribe aplicaciones altamente optimizadas para benchmarking de PC en C++ o intenta aplicar ingeniería inversa a binarios interesantes. Si no está frente a un ordenador (desmontado), pasa tiempo con su querida esposa, su hijo recién nacido y su fascinante y peculiar gato.

Georg Schelkshorn es un entusiasta de React full-stack con gran entusiasmo por DevOps y la creación de interfaces intuitivas . Junto con Daniel Bugl, codirige TouchLay, una empresa especializada en soluciones interactivas de hardware y software, además de asesorar a empresas en la implementación exitosa de proyectos web modernos. A través de numerosos proyectos desafiantes, ha adquirido un profundo conocimiento de lo que hace que un proyecto full-stack de React funcione a escala. A Georg le apasiona explorar nuevas áreas de aprendizaje y espera que disfrutes explorando la experiencia de aprendizaje de este libro.

Kirill Ezhemenskii es un director de tecnología visionario que dirige una empresa de salud con soluciones de software de vanguardia . Es un maestro de la programación funcional y un experto en desarrollo web y móvil moderno. Aprovecha el poder de React, Next.js, GraphQL y TypeScript para crear aplicaciones impresionantes y de alto rendimiento que funcionan en cualquier plataforma. Además, es un mentor generoso que comparte su experiencia y pasión por React Native con los futuros desarrolladores.

Prefacio

Conocí a Daniel hace 14 años. Es muy amigo de la familia de mi esposa, así que lo invitaban a menudo a las reuniones familiares. Por aquel entonces, Daniel ya era un joven brillante con una curiosidad inagotable. Un día, me invitó con entusiasmo a visitar su "laboratorio" (que por aquel entonces era simplemente una habitación en el sótano de la familia) para mostrarme su último proyecto. Junto con el primo de mi esposa, había construido una mesa táctil con una caja de madera. Era un prototipo inicial, pero ya completamente funcional. Daniel tenía solo 13 años por aquel entonces. Pasaron los años y lo vi sumergirse en el desarrollo web, crear un negocio exitoso y adoptar React en 2015.

Hoy, Daniel tiene la misma edad que yo en aquel entonces y es un honor para mí reseñar su tercer libro. También he tenido la gran fortuna de trabajar estrechamente con Daniel durante los últimos cinco años en proyectos corporativos de React. Este libro refleja su enfoque bien estructurado ante los desafíos de las aplicaciones web empresariales. Extrae con cuidado la esencia de conceptos complejos y los presenta con tanta claridad que incluso los capítulos más difíciles resultarán manejables. El código fuente incluido evita que te distraigas con detalles innecesarios.

Daniel comienza tu experiencia brindándote las herramientas básicas necesarias para crear un proyecto React moderno y completo desde cero. Cada paso está guiado y explicado detalladamente. Más adelante, a medida que los capítulos se vuelven más desafiantes, el enfoque se alejará de lo básico, dando a tus nuevas habilidades la oportunidad de completar los aspectos básicos. Por eso es importante trabajar también en lo básico para que puedas concentrarte en el tema avanzado en cuestión.

Hace cinco años, me encontraba en la misma situación que tú, el lector. Tenía muchas ganas de aprender React full-stack, pero no sabía por dónde empezar. Tuve la oportunidad de aprender directamente de Daniel. Con este libro, tú por fin también tienes esa oportunidad.

Matías Zronek

Ingeniero de software sénior y asesor técnico

Tabla de contenido

Prefacio

xvii

Parte 1: Introducción a Full-Stack Desarrollo

1

Preparación para el desarrollo full-stack	3
Requisitos técnicos	3
Motivación para convertirse en un desarrollador	4
full-stack ¿Qué novedades hay en esta versión de Full-Stack React Projects?	4
Cómo sacar el máximo provecho de este libro	4
Configuración del desarrollo ambiente	5
Instalación de VS Code y extensiones	5
Configurar un proyecto con Vite	8
Configuración de ESLint y Prettier para aplicar las mejores prácticas y el estilo del código	11
Configurar Husky para asegurarnos de que enviamos el código correcto	16
Resumen	20

2

Conociendo Node.js y MongoDB	21
Requisitos técnicos Escribir y ejecutar scripts con Node.js	21
Similitudes y diferencias entre JavaScript en el navegador y en Node.js	22
JavaScript en el navegador y en Node.js Creando nuestro primer script en Node.js Manejo de archivos en Node.js	23
Concurrencia con JavaScript en el navegador y Node.js Creando nuestro primer servidor web	25
Extendiendo el servidor web para servir nuestro JSON	28
Presentamos Docker, una plataforma para contenedores	30
Plataforma Docker electrónica	31

Instalación de Docker	32	Ejecutar comandos directamente en la base de datos 38	
Creando un contenedor	32	Accediendo a la base de datos a través de VS Code	42
Acceder a Docker a través de VS Code	34	Acceso a la base de datos MongoDB a través de Node.js Resumen	45
Presentamos MongoDB, una base de datos de documentos	34		47
Configuración de un servidor MongoDB	37		

Parte 2: Construcción e implementación de nuestro primer Aplicación de pila completa con una API REST

3

Implementación de un backend con Express, Mongoose ODM y Jest 51

Requisitos técnicos	51	Definición de casos de prueba para la función de servicio createPost	64
Diseño de un servicio backend	52	Definiendo una función para listar publicaciones	66
Creando la estructura de carpetas para nuestro servicio backend	53	Definición de casos de prueba para publicaciones de listas	67
Creación de esquemas de bases de datos utilizando Mangosta	55	Definición de las funciones de obtener publicaciones individuales, actualizar y eliminar publicaciones	71
	56	Usando la extensión Jest VS Code	74
Definición de un modelo para publicaciones de blog	57		
Usando el modelo de publicación de blog	57		
Definiendo las fechas de creación y última actualización en la entrada del blog	59		
Desarrollo y prueba de funciones de servicio	60	Proporcionar una API REST mediante Express 75	
Configuración del entorno de prueba	60	Definiendo nuestras rutas API	77
Escribiendo nuestra primera función de servicio: createPost	63	Configuración de Express	77
		Uso de dotenv para configurar variables de entorno	79
		Usando nodemon para un desarrollo más sencillo	81
		Creando nuestras rutas API con Express	83
		Resumen	90

4

Integración de un frontend con React y TanStack Query 91

Requisitos técnicos	91	Creando la interfaz de usuario para nuestra aplicación	94
Principios de React	92	Estructura del componente	95
Configuración de un proyecto React full-stack	93	Implementación de componentes estáticos de React	97

Integración del servicio backend mediante TanStack Query	106	Implementación de filtros y ordenación	110
Configuración de la consulta TanStack para React	107	Creando nuevas publicaciones	114
Obteniendo publicaciones del blog	108	Resumen	118

5

Implementación de la aplicación con Docker y CI/CD **119**

Requisitos técnicos Creación de imágenes de Docker	119	Crear una cuenta en Google Cloud	131
Creación del Dockerfile de backend	120	Implementando nuestras imágenes Docker en un registro Docker	132
Creación de un archivo .dockerignore	120	Implementar la imagen de Docker de backend en Carrera en la nube	134
Construcción de la imagen de Docker	122	Implementar la imagen de Docker del frontend en Carrera en la nube	135
Creación y ejecución de un contenedor desde nuestra imagen	122		
	123		
Creación del Dockerfile de frontend	124	Configuración de CI para automatizar pruebas	136
Creación del archivo .dockerignore para el frontend	126	Añadiendo CI para el frontend	137
Construyendo la imagen de Docker del frontend	126	Agregar CI para el backend	138
Creación y ejecución del contenedor frontend	127	Configuración de CD para automatizar la implementación	141
Administrando múltiples imágenes usando Docker Componer	127	Obtener credenciales de Docker Hub	141
Limpieza de contenedores no utilizados	129	Obtener credenciales de Google Cloud	141
Definiendo el flujo de trabajo de implementación			142
Implementando nuestra aplicación full-stack en la nube	130	Resumen	146
Creación de una base de datos MongoDB Atlas	130		

Parte 3: Práctica del desarrollo de pila completa

Aplicaciones web

6

Agregar autenticación con JWT **149**

Requisitos técnicos ¿Qué es JWT?	149	Carga útil JWT	151
Encabezado JWT	150	Firma JWT	152
	150	Creando un JWT	152

[Tabla de contenido](#)

Uso de JWT	153	Integración del inicio de sesión y el registro en el frontend usando React Router y JWT	166
Almacenamiento de JWT	155	Uso de React Router para implementar múltiples rutas	
Implementación de inicio de sesión, registro y rutas autenticadas en el backend mediante JWT	155	Creando la página de registro	168
Creando el modelo de usuario	155	Vinculación a otras rutas mediante el componente Enlace	170
Creación del servicio de registro	156	Creación de la página de inicio de sesión y almacenamiento del JWT	172
Creando la ruta de registro	157	Usando el JWT almacenado e implementando un cierre de sesión simple	174
Creando el servicio de inicio de sesión	158	Obteniendo los nombres de usuario	175
Creando la ruta de inicio de sesión	160	Envío del encabezado JWT al crear publicaciones	178
Definiendo rutas autenticadas	162	Manejo avanzado de tokens	180
Acceder al usuario actualmente conectado	164	Resumen	181

7

Mejorar el tiempo de carga mediante la renderización del lado del servidor	183		
Requisitos técnicos	183	Actualización de index.html y package.json Cómo hacer que React Router funcione con la renderización del lado del servidor	197
Evaluación comparativa del tiempo de carga de nuestra aplicación	184	Obtención de datos del lado del servidor	204
Representación de componentes de React en el servidor	191	Utilizando datos iniciales	204
Configuración del servidor	193	Usando la hidratación	205
Definiendo el punto de entrada del lado del servidor	196	Renderizado avanzado del lado del servidor	209
Definiendo el punto de entrada del lado del cliente	196	Resumen	212

8

Cómo asegurarse de que los clientes lo encuentren con los motores de búsqueda			
Mejoramiento	213		
Requisitos técnicos Optimizar una aplicación para motores de búsqueda Crear un archivo robots.txt	213	Creación de URL significativas (slugs)	221
Crear páginas separadas para publicaciones	214	Agregar títulos dinámicos	222
	215	Agregar otras metaetiquetas	223
	216	Creación de un mapa del sitio	226

Mejorar las incrustaciones en redes sociales	229	Uso de las metaetiquetas del artículo OG	230
Metaetiquetas de Open Graph	229	Resumen	232

9

Implementación de pruebas de extremo a extremo con Playwright	233		
Requisitos técnicos	233	Configuraciones de prueba reutilizables utilizando accesorios	244
Configuración de Playwright para pruebas de extremo a extremo	234	Descripción general de los accesorios incorporados	245
Instalación de Playwright	234	Escribiendo nuestro propio futuro	245
Preparación del backend para pruebas de extremo a extremo	235	Uso de accesorios personalizados	247
Escritura y ejecución de pruebas de extremo a extremo	237	Visualización de informes de pruebas y ejecución en CI	248
Usando la extensión VS Code	238	Visualización de un informe HTML	248
		Ejecución de pruebas de Playwright en CI	250
		Resumen	251

10

Agregación y visualización de estadísticas con MongoDB y Victory	253		
Requisitos técnicos	254	Implementación de la agregación de datos en el backend	268
Recopilación y simulación de eventos	254	Definición de funciones de servicio de agregación	268
Creando el modelo de evento	254	Definiendo las rutas	271
Definición de una función de servicio y ruta a seguir eventos	255	Integración y visualización de datos en el frontend usando Victory	272
Recopilación de eventos en el frontend	257	Integración de la API de agregación	273
Simulación de eventos	258	Visualización de datos con Victory	275
Agregación de datos con MongoDB	262	Resumen	279
Obtener el número total de vistas por publicación	262		
Obtener el número de visualizaciones diarias por publicación	264		
Cálculo de la duración media de la sesión	266		

11

Construyendo un backend con una API GraphQL	281
--	------------

Requisitos técnicos ¿Qué es GraphQL?	281	Implementación de autenticación y mutaciones GraphQL	296
Mutaciones	282	Agregar autenticación a GraphQL	296
		Implementando mutaciones	297
Implementación de una API GraphQL en un backend	285	Uso de mutaciones	299
Implementar campos que consultan publicaciones	287	Descripción general de los conceptos avanzados de GraphQL	301
Definiendo el tipo de publicación	289	Fragmentos	302
Definiendo el tipo de usuario	290	Introspección	303
Probando consultas profundamente anidadas	291		
Implementación de tipos de entrada	294	Resumen	303

12

Interfaz con GraphQL en el frontend mediante el cliente Apollo	305
---	------------

Requisitos técnicos	305	Uso de fragmentos para reutilizar partes de consultas	312
Configuración del cliente Apollo y realización de nuestra primera consulta	306	Uso de mutaciones en el frontend	314
Consultar publicaciones desde el frontend usando GraphQL	307	Migración del inicio de sesión a GraphQL	316
Resolución de nombres de usuario de autores en una sola consulta	310	Migración de la publicación creada a GraphQL	317
		Resumen	320

Uso de variables en consultas GraphQL 312

Parte 4: Exploración de un modelo full-stack basado en eventos Arquitectura

13

Creación de un backend basado en eventos con Express y Socket.IO	323
---	------------

Requisitos técnicos	323	¿Qué son las aplicaciones basadas en eventos?	324
		¿Qué son los WebSockets?	325

¿Qué es Socket.IO?	326	Transmitir mensajes de chat desde el servidor a todos los clientes	332
Conexión a Socket.IO Emisión y recepción de eventos	326	Unirse a salas para enviar mensajes en	333
Configuración de Socket.IO	328	Uso de reconocimientos para obtener información sobre un usuario	334
Configuración de un cliente Socket.IO simple	329		
Creación de un backend para una aplicación de chat usando Socket.IO	331	Agregar autenticación mediante la integración de JWT con Socket.IO	336
Emisión de eventos para enviar mensajes de chat desde el cliente al servidor	331	Resumen	340

14

Creación de una interfaz para consumir y enviar eventos	341
--	------------

Requisitos técnicos	341	Implementación de la funcionalidad de chat	350
Integración del cliente Socket.IO con React	342	Implementando los componentes de chat	351
Limpieza del proyecto Creación de un contexto Socket.IO Conectar el contexto y mostrar el estado	342	Implementando un gancho useChat	352
	343	Implementación del componente ChatRoom	354
	346	Implementar comandos de chat con reconocimientos	356
Desconectar el socket al cerrar sesión	350	Resumen	358

15

Cómo añadir persistencia a Socket.IO con MongoDB	359
---	------------

Requisitos técnicos	359	Refactorizar la aplicación para que sea más extensible	365
Almacenamiento y reproducción de mensajes mediante MongoDB	360	Definición de funciones de servicio	365
Creación del esquema Mongoose	360	Refactorización del servidor Socket.IO para utilizar las funciones de servicio	367
Creación de las funciones de servicio	361	Refactorización del código del lado del cliente	368
Almacenamiento y reproducción de mensajes	361		
Distinguir visualmente los mensajes reproducidos	363	Implementar comandos para unirse y cambiar de sala	370
		Resumen	373

Parte 5: Avanzando hacia una solución full-stack preparada para la empresa

Aplicaciones

16

Introducción a Next.js 377

Requisitos técnicos ¿Qué es	378	Creación de componentes y	
Next.js?	378	páginas estáticas	386
Configuración de Next.js	379	Componentes de Denoing	386
Introducción al enrutador de aplicaciones	382	Páginas de denoing	390
Definición de la estructura de carpetas	384	Agregar enlaces entre páginas	392
		Resumen	395

17

Presentación de los componentes del servidor React 397

Requisitos técnicos ¿Qué	397	Obteniendo una sola publicación	410
son los RSC?	398	Uso de acciones del servidor para registrarse,	
		iniciar sesión y crear nuevas publicaciones	412
Agregar una capa de datos a nuestra		Implementando la página de registro	413
aplicación Next.js	402	Implementación de la página de inicio de sesión y	
Configurar la conexión a la base de datos Crear	403	manejo de JWT	416
los modelos de base de datos Definir las	404	Comprobación de si el usuario ha iniciado sesión	418
funciones de la capa de datos	405	Implementación del cierre de sesión	419
Uso de RSC para obtener datos de la		Implementando la creación de publicaciones	421
base de datos	409	Resumen	422
Obtener una lista de publicaciones	409		

18

Conceptos y optimizaciones avanzadas de Next.js 423

Requisitos técnicos Definición de rutas	423	Almacenamiento en caché en	426
API en Next.js Creación de una ruta API para listar	424	Next.js Exploración de la representación estática en rutas	428
publicaciones de blog	424	API Cómo hacer que la ruta sea dinámica	430

Funciones de almacenamiento en caché en la capa de datos	430	Creación de URL significativas (slugs)	438
Revalidación de la caché mediante acciones del servidor	433	Creación de un mapa del sitio	439
Revalidar la caché a través de un webhook	434		
Revalidar la caché periódicamente	435	Carga optimizada de imágenes y fuentes en	
	436	Next.js	440
Cómo optar por no usar el almacenamiento en caché		e Componente de fuente	440
SEO con Next.js Agregar	436	e Componente de imagen	442
títulos dinámicos y metaetiquetas Crear un	436		
archivo robots.txt	437	Resumen	444

19

Implementación de una aplicación Next.js	445
---	------------

Requisitos técnicos	445	Creación de una configuración de implementación	
Implementación de una aplicación Next.js con Vercel	446	personalizada para	451
Configuración de variables de entorno en Vercel	447	aplicaciones Next.js Resumen	454

20

Profundizando en el desarrollo full-stack	455
--	------------

Descripción general de otros marcos de trabajo de pila completa	455	Jotai	459
		Redux	459
Next.js	456	MobX	459
Remezcla	456	xstate	460
Gatsby	457	Estado actual	460
Descripción general de las bibliotecas de UI	457	Consejos para el mantenimiento de proyectos a gran escala	
Interfaz de usuario de materiales (MUI)	458		460
Viento de cola CSS	458	Usando TypeScript	460
Reaccionar Aria	458	Configurar un Monorepo	460
NextUI	458	Optimización del tamaño del paquete	461
Descripción general de las soluciones avanzadas de gestión de estados	459		
Retroceso	459	Resumen	461

Índice	463
---------------	------------

Otros libros que te pueden gustar	476
--	------------

Prefacio

Hola, soy Daniel, emprendedor, asesor técnico y desarrollador full-stack con foco en tecnologías del ecosistema React.

En mi tiempo como asesor técnico y desarrollador para empresas y el sector público, he notado que cada vez más compañías buscan reducir la brecha entre los desarrolladores frontend y backend. Sus requisitos comerciales a menudo resultan en la necesidad de un llamado "backend para frontend", donde los datos se obtienen de diferentes sistemas backend y se preparan de una manera que se pueda mostrar fácilmente en el frontend.

Como emprendedor, también tengo experiencia en iniciar nuevos proyectos con equipos más pequeños, donde es esencial que cada desarrollador de su equipo pueda hacer todo, no solo el frontend o el backend. En tales casos, a menudo tiene sentido desarrollar el backend y el frontend en el mismo lenguaje, que suele ser JavaScript (o TypeScript), porque hay un gran ecosistema y una gran cantidad de desarrolladores disponibles.

En ambos casos, convertirse en un desarrollador full-stack es cada vez más importante. Llevo mucho tiempo asesorando a desarrolladores para que aprendan más sobre desarrollo full-stack y he observado que existen problemas y malentendidos comunes que la mayoría de los desarrolladores encuentran al aprender desarrollo full-stack. En este libro, quiero resumir todos mis aprendizajes sobre desarrollo full-stack, ofreciéndote consejos sobre dónde y cómo aprender más sobre el ecosistema en constante crecimiento del desarrollo full-stack en JavaScript.

Hoy en día, muchas empresas utilizan una pila compuesta por MongoDB, Express, React y Node.js, denominada pila MERN. En este libro, te enseñaré a crear aplicaciones React modernas y completas utilizando estas tecnologías. Las enseñaré desde cero, utilizando la menor cantidad de bibliotecas posible, para que puedas aprender los conceptos esenciales. Esto te permitirá adaptarte a las nuevas tecnologías durante años, incluso cuando las herramientas específicas que se utilizan en este libro se vuelvan obsoletas. Además, te enseñaré sobre la implementación de aplicaciones y DevOps, ya que he descubierto que este sector suele estar descuidado y no hay suficientes desarrolladores que lo conozcan. En la última parte del libro, presentaré Next.js como un framework completo y ofreceré una perspectiva sobre los nuevos desarrollos en este sector, como los componentes y las acciones de servidor de React.

Espero que disfrutes de este libro. Si tienes alguna pregunta o comentario, ¡no dudes en contactarme!

Para quién es este libro

Este libro está dirigido a desarrolladores con experiencia en React que desean aprender a crear, integrar e implementar diversos sistemas backend para convertirse en desarrolladores full-stack. Deberías tener un buen conocimiento de JavaScript y React, pero no necesitas conocimientos previos de desarrollo, creación, integración e implementación de sistemas backend. Si te enfrentas a alguno de los siguientes desafíos, este libro es perfecto para ti:

- Sabes cómo hacer un frontend con React pero no tienes idea de cómo integrarlo correctamente con un backend
- Quieres crear un proyecto full-stack desde cero pero no sabes cómo
- Desea obtener más información sobre la implementación de aplicaciones y DevOps
- Desea obtener más información sobre el desarrollo moderno de React, como React Server Components, Server Actions y Next.js

Este libro le proporcionará proyectos del mundo real e incluye todos los pasos necesarios para convertirse en un desarrollador full-stack, incluidos, entre otros, el desarrollo backend, el desarrollo frontend, las pruebas (pruebas unitarias y pruebas de extremo a extremo) y la implementación.

Qué cubre este libro

El Capítulo 1, Preparación para el desarrollo full-stack, ofrece una breve descripción general del contenido del libro y le enseña cómo configurar un proyecto que se utilizará como base para el desarrollo de sus proyectos full-stack.

El Capítulo 2, "Conociendo Node.js y MongoDB", proporciona información sobre cómo escribir y ejecutar scripts con Node.js. \$en, explica cómo usar Docker para configurar un servicio de base de datos. También presenta MongoDB, una base de datos de documentos, y cómo acceder a ella mediante Node.js.

El capítulo 3, Implementación de un backend con Express, pone en práctica lo aprendido en el capítulo 2. Mediante la creación de un servicio de backend. Express se utiliza para proporcionar una API REST, Mongoose ODM para interactuar con MongoDB y Jest para escribir pruebas unitarias para el código de backend.

El capítulo 4, "Integración de un frontend con React y TanStack Query", proporciona instrucciones para crear un frontend que interactúe con el servicio backend creado previamente. Utiliza Vite para configurar un proyecto React, en el que creamos una interfaz de usuario básica. Además, enseña a usar TanStack Query, una biblioteca de obtención de datos, para gestionar el estado del backend e integrar la API del backend con el frontend.

El capítulo 5, Implementación de la aplicación con Docker y CI/CD, profundiza en DevOps al enseñarle sobre Docker y cómo empaquetar una aplicación con él. Luego, proporciona instrucciones sobre cómo implementar una aplicación en un proveedor de nube y cómo configurar CI/CD para automatizar la implementación.

El capítulo 6, "Añadir autenticación con JWT", explica los tokens web JSON, una forma de añadir autenticación a las aplicaciones web. También proporciona instrucciones sobre cómo configurar múltiples rutas con React Router.

El capítulo 7, Cómo mejorar el tiempo de carga mediante la renderización del lado del servidor, cubre la evaluación comparativa de una aplicación y le enseña sobre Web Vitals. Además, brinda instrucciones sobre cómo implementar una forma de renderizar componentes de React en un servidor desde cero y cómo recuperar datos previamente en el servidor.

El capítulo 8, "Cómo asegurar que los clientes te encuentren con la optimización para motores de búsqueda", se centra en cómo optimizar una aplicación para que sea encontrada por motores de búsqueda como Google o Bing. Además, proporciona información sobre cómo crear metaetiquetas para facilitar la integración con diversas redes sociales.

El capítulo 9, "Implementación de pruebas integrales con Playwright", presenta Playwright como una herramienta para escribir pruebas integrales que ejecuta automáticamente acciones en una aplicación para determinar si el código sigue funcionando correctamente después de realizar cambios. También explica cómo ejecutar Playwright en CI mediante GitHub Actions.

El capítulo 10, Agregación y visualización de estadísticas con MongoDB y Victory, proporciona instrucciones sobre cómo recopilar eventos en una aplicación. En \$en, se enseña cómo agregar datos con MongoDB para generar estadísticas resumidas, como el número de visualizaciones o la duración de la sesión. Finalmente, se explica la creación de gráficos para visualizar estas estadísticas agregadas con la biblioteca Victory.

El capítulo 11, Creación de un backend con una API GraphQL, presenta GraphQL como una alternativa a las API REST y aprenderá cuándo es útil usarlo y cómo implementarlo en un backend.

El capítulo 12, Interfaz con GraphQL en el frontend usando Apollo Client, le enseña cómo usar Apollo Client en el frontend para interactuar con el backend GraphQL implementado previamente.

El capítulo 13, "Desarrollo de un backend basado en eventos con Express y Socket.IO", presenta una arquitectura basada en eventos, útil para aplicaciones que gestionan datos en tiempo real, como aplicaciones colaborativas (Google Docs o una pizarra digital) o aplicaciones financieras (la plataforma de intercambio de criptomonedas Kraken). Enseña sobre WebSockets y cómo usar Socket.IO para implementar un backend basado en eventos.

El capítulo 14, Creación de un frontend para consumir y enviar eventos, implementa un frontend para el backend basado en eventos creado previamente e interactúa con él mediante Socket.IO.

El capítulo 15, Cómo agregar persistencia a Socket.IO usando MongoDB, le enseña cómo integrar correctamente una base de datos en una aplicación basada en eventos para persistir (y luego reproducir) eventos.

El capítulo 16, Introducción a Next.js, presenta Next.js como un framework de aplicaciones web integral y listo para empresas para React. Destaca las diferencias entre usar un framework y un empaquetador simple como Vite. También te enseña sobre Next.js App Router, un nuevo paradigma para definir rutas y páginas.

El capítulo 17, Introducción a los componentes de servidor de React, presenta un nuevo concepto en React: los componentes de servidor. Estos componentes permiten integrar directamente aplicaciones de React con una base de datos sin necesidad de una API REST o GraphQL. Además, se explican las acciones de servidor, que permiten invocar funciones en el servidor a través del frontend.

xx Prefacio

El capítulo 18, Conceptos y optimizaciones avanzados de Next.js, profundiza en el marco de Next.js y proporciona información sobre cómo funciona el almacenamiento en caché en Next.js y cómo se puede utilizar para optimizar las aplicaciones. También te enseña a definir rutas de API en Next.js y a añadir metadatos para la optimización en motores de búsqueda. Por último, te enseña a cargar imágenes y fuentes de forma óptima en Next.js.

El capítulo 19, "Implementación de una aplicación Next.js", enseña dos maneras de implementar una aplicación Next.js. La forma más sencilla es usar la plataforma Vercel, con la que podemos poner en marcha nuestra aplicación rápidamente. Sin embargo, también enseña cómo crear una configuración de implementación personalizada con Docker.

El capítulo 20, "Profundizando en el desarrollo full-stack", aborda brevemente varios temas avanzados que aún no se han abordado en este libro. Comienza con una descripción general de otros frameworks full-stack y luego resume conceptos como el mantenimiento de proyectos a gran escala, la optimización del tamaño del paquete, una descripción general de las bibliotecas de interfaz de usuario y soluciones avanzadas de gestión de estado.

Para aprovechar al máximo este libro

Software/hardware tratado en el libro	Requisitos del sistema operativo
Node.js v20.10.0	Windows, macOS o Linux
Git v2.43.0	
Visual Studio Code v1.84.2	
Docker v24.0.6	
Docker Desktop v4.25.2	
MongoDB Shell v2.1.0	

Si utiliza la versión digital de este libro, le recomendamos que escriba el código usted mismo o acceda a él desde el repositorio de GitHub (enlace disponible en la siguiente sección). Esto le ayudará a evitar posibles errores al copiar y pegar código.

Descargue los archivos de código de ejemplo

Puede descargar los archivos de código de ejemplo para este libro desde GitHub en <https://github.com/PacktPublishing/Modern-Full-Stack-React-Projects>. Si hay alguna actualización del código, se publicará en el repositorio de GitHub.

También tenemos otros paquetes de códigos de nuestro rico catálogo de libros y videos disponibles en <https://github.com/PacktPublishing/>. ¡Échales un vistazo!

Código en acción

Los videos de Código en acción para este libro se pueden ver en <https://packt.link/VINfo>.

Convenciones utilizadas

A lo largo de este libro se utilizan varias convenciones de texto.

Código en texto: Indica palabras clave en texto, nombres de tablas de bases de datos, nombres de carpetas, nombres de archivos, extensiones de archivos, rutas de acceso, URL ficticias, entradas del usuario y perfiles de Twitter. Ejemplo: "Primero, necesitas crear un archivo robots.txt para que los motores de búsqueda puedan rastrear partes de tu sitio web y qué partes pueden rastrear".

Un bloque de código se establece de la siguiente manera:

```
exportar const getPostById = async (postId) => {
    const res = await fetch(`$import.meta.env.VITE_BACKEND_URL}/
publicaciones/${postId}`)
    devolver esperar res.json()
}
```

Cuando deseamos llamar su atención sobre una parte particular de un bloque de código, las líneas o elementos relevantes se muestran en negrita:

```
{Publicación completa ?
    <h3>{título}</h3>
) : (
    <Enlace a={`/posts/${_id}}>
        <h3>{título}</h3>
    </Enlace>
)}
```

Cualquier entrada o salida de la línea de comandos se escribe de la siguiente manera:

```
$ npm install node-emoji
```

Negrita: Indica un término nuevo, una palabra importante o palabras que aparecen en pantalla. Por ejemplo, las palabras en menús o cuadros de diálogo aparecen en negrita. Ejemplo: "Conéctese a la base de datos, expanda la sección Zonas de juego (si aún no está expandida) y haga clic en el botón Crear nueva zona de juego".

Consejos o notas importantes

Aparece así.

Ponte en contacto con nosotros

Los comentarios de nuestros lectores siempre son bienvenidos.

Comentarios generales: Si tiene preguntas sobre cualquier aspecto de este libro, envíenos un correo electrónico a customercare@packtpub.com y menciona el título del libro en el asunto de tu mensaje.

Erratas: Aunque hemos tomado todas las precauciones para garantizar la precisión de nuestro contenido, pueden ocurrir errores. Si encuentra algún error en este libro, le agradeceríamos que nos lo informara. Visite www.packtpub.com/support/errata y complete el formulario.

Piratería: Si encuentra copias ilegales de nuestras obras en internet, en cualquier formato, le agradeceríamos que nos proporcionara la dirección o el nombre del sitio web. Por favor, contáctenos en copyright@packt.com con el enlace al material.

Si está interesado en convertirse en autor: si hay un tema en el que es experto y está interesado en escribir o contribuir a un libro, visite authors.packtpub.com.

Comparte tus pensamientos

Una vez que hayas leído Proyectos Full-Stack React Modernos, ¡nos encantaría conocer tu opinión! Haz clic aquí para ir directamente a la página de revisión de Amazon de este libro y compartir sus comentarios.

Su reseña es importante para nosotros y para la comunidad tecnológica y nos ayudará a garantizar que ofrecemos contenido de excelente calidad.

Descargue una copia gratuita en PDF de este libro

¡Gracias por comprar este libro!

¿Te gusta leer mientras viajas pero no puedes llevar tus libros impresos a todas partes?

¿Tu compra de libro electrónico no es compatible con el dispositivo que eliges?

No te preocunes, ahora con cada libro de Packt obtendrás una versión PDF de ese libro sin DRM sin costo.

Lee en cualquier lugar, en cualquier dispositivo. Busca, copia y pega código de tus libros técnicos favoritos directamente en tu aplicación.

Los beneficios no terminan ahí, puedes obtener acceso exclusivo a descuentos, boletines informativos y excelente contenido gratuito en tu bandeja de entrada todos los días.

Siga estos sencillos pasos para obtener los beneficios:

1. Escanea el código QR o visita el enlace a continuación



<https://packt.link/libro-egrafico/978-1-83763-795-9>

2. Envíe su comprobante de compra

3. ¡Listo! Te enviaremos tu PDF gratuito y otros beneficios directamente a tu correo electrónico.

Parte 1:

Introducción a Desarrollo de pila completa

En esta parte, aprenderá a configurar un proyecto y las herramientas para el desarrollo full-stack. También conocerá y dará los primeros pasos con Node.js, Docker y MongoDB. Después de esta parte, tendrá una configuración básica de proyecto que podrá utilizar para los proyectos posteriores que se desarrollan a lo largo de este libro.

Esta parte incluye los siguientes capítulos:

- Capítulo 1, Preparación para el desarrollo full-stack
- Capítulo 2, Conociendo Node.js y MongoDB

1

Preparación para la pila completa Desarrollo

En este capítulo, primero daré una breve descripción general del contenido del libro y explicaré por qué las habilidades que se enseñan en él son importantes en un entorno de desarrollo moderno. A continuación, nos pondremos manos a la obra y configuraremos un proyecto que servirá de base para el desarrollo de nuestros proyectos full-stack. Al final de este capítulo, tendrá un entorno de desarrollo integrado (IDE) y un proyecto configurados y listos para el desarrollo full-stack, y comprenderá qué herramientas se pueden utilizar para configurar dichos proyectos.

En este capítulo cubriremos los siguientes temas principales:

- Motivación para convertirse en un desarrollador full-stack
- ¿Qué novedades hay en la tercera edición?
- Cómo sacar el máximo provecho de este libro
- Configuración del entorno de desarrollo

Requisitos técnicos

Este capítulo le guiará en la configuración de todas las tecnologías necesarias para desarrollar aplicaciones web integrales a lo largo de este libro. Antes de comenzar, instale lo siguiente, si aún no lo tiene instalado:

- Node.js v20.10.0
- Git v2.43.0
- Visual Studio Code v1.84.2

Estas versiones son las que se usan en el libro. Aunque instalar una versión más reciente no debería ser un problema, tenga en cuenta que algunos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/cap1>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/dyf3nECvKAE>.

Importante

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

Motivación para convertirse en un desarrollador full-stack

Comprender el desarrollo full-stack es cada vez más importante, ya que las empresas buscan aumentar la cooperación (y reducir la brecha) entre el frontend y el back-end. El front-end se integra cada vez más profundamente con el back-end, utilizando tecnologías como la renderización del lado del servidor. A lo largo de este libro, aprenderemos sobre el desarrollo, la integración, las pruebas y la implementación de proyectos full-stack.

¿Qué novedades hay en esta versión de Proyectos Full-Stack React?

A diferencia de las versiones anteriores de Full-Stack React Projects, esta nueva versión se centra más en la integración del frontend con el backend que las dos ediciones anteriores y, por lo tanto, intencionalmente no se centra tanto en la creación de una interfaz de usuario (UI) o el uso de bibliotecas de UI, como Material UI (MUI), en el frontend. Esta edición proporciona el conocimiento esencial para integrar e implementar aplicaciones web full-stack. La implementación de aplicaciones faltaba por completo en las ediciones anteriores, y las pruebas solo se introdujeron brevemente. Esta edición se centra más en estas partes esenciales del desarrollo full-stack de modo que, después de leer este libro, podrá desarrollar, integrar, probar e implementar una aplicación web full-stack.

Cómo sacar el máximo provecho de este libro

Para que el libro sea breve y conciso, utilizaremos tecnologías y herramientas específicas. Sin embargo, estos conceptos también se aplican a otras tecnologías. Intentaremos presentar brevemente alternativas para que, si algo no se adapta a su proyecto, pueda elegir otras herramientas. Recomiendo probar primero las tecnologías presentadas en este libro para poder seguir las instrucciones, pero no dude en probar las alternativas por su cuenta más adelante.

Se recomienda encarecidamente que escriba el código usted mismo. No se limite a ejecutar los ejemplos de código proporcionados. Es importante escribir el código usted mismo para aprenderlo y comprenderlo correctamente. Sin embargo, si surge algún problema, siempre puedes consultar los ejemplos de código.

Dicho esto, comencemos a configurar nuestro entorno de desarrollo en la siguiente sección.

Configuración del entorno de desarrollo

En este libro, usaremos Visual Studio Code (VS Code) como editor de código. Puedes usar el editor que prefieras, pero ten en cuenta que las extensiones y la configuración pueden variar ligeramente según el editor que elijas.

Ahora instalaremos VS Code y extensiones útiles, y luego continuemos configurando todas las herramientas necesarias para nuestro entorno de desarrollo.

Instalación de VS Code y extensiones

Antes de que podamos comenzar a desarrollar y configurar las otras herramientas, necesitamos configurar nuestro editor de código siguiendo estos pasos:

1. Descarga VS Code para tu sistema operativo desde la página web de redes sociales (al momento de escribir esto, la URL es <https://code.visualstudio.com/>). Usaremos la versión 1.84.2. en este libro.
2. Despues de descargar e instalar la aplicación, ábrala y deberia ver la siguiente ventana:

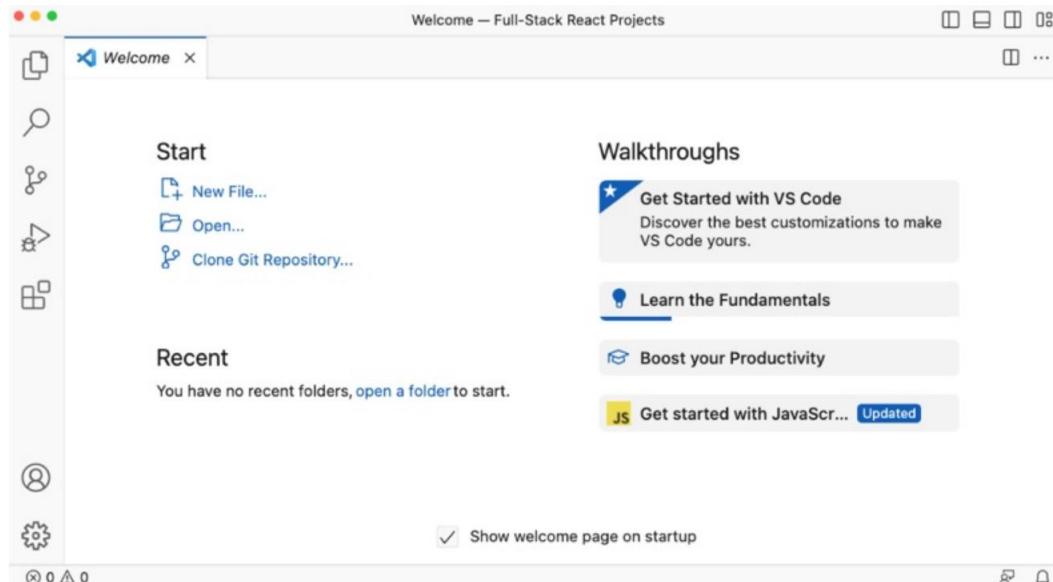


Figura 1.1 – Una nueva instalación de VS Code (en macOS)

3. Para facilitar las cosas más adelante, vamos a instalar algunas extensiones, así que haga clic en Extensiones ícono, que es el ícono "h" desde arriba a la izquierda en la captura de pantalla. Se abrirá una barra lateral donde verá "Extensiones de búsqueda en Marketplace" en la parte superior. Ingrese el nombre de una extensión y haga clic en "Instalar". Empecemos por instalar la extensión de Docker :

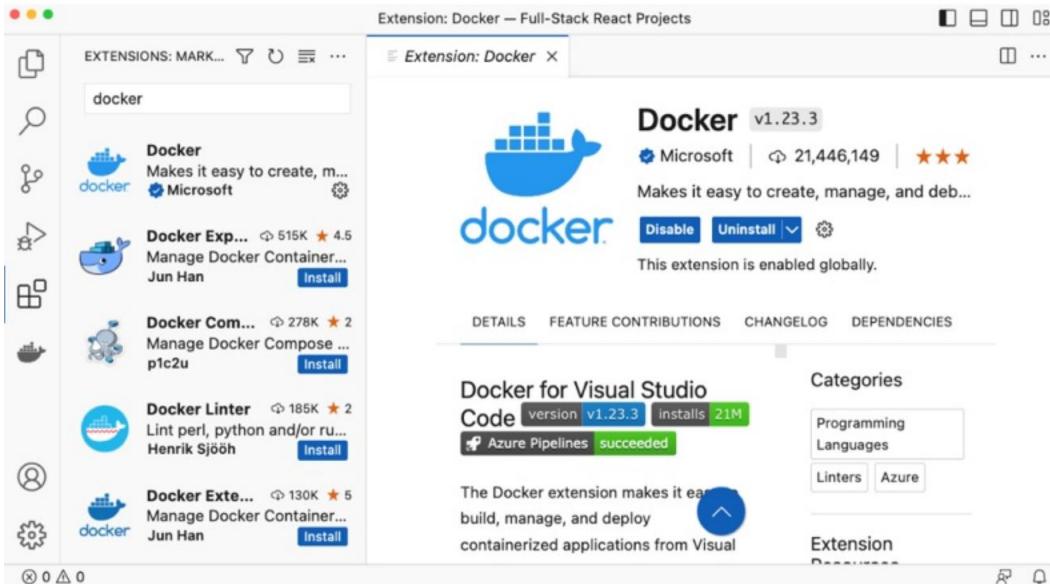


Figura 1.2 – Instalación de la extensión Docker en VS Code

4. Instale las siguientes extensiones:

Docker (de Microsoft)

ESLint (de Microsoft)

Prettier – Formateador de código (de Prettier)

MongoDB para VS Code (por MongoDB)

El soporte para JavaScript y Node.js ya viene integrado en VS Code.

5. Crea una carpeta para los proyectos creados en este libro (por ejemplo, puedes llamarla Full-Stack- React-Projects).

Dentro de esta carpeta, crea una nueva carpeta llamada ch1.

6. Vaya a la pestaña Archivos (primer ícono desde arriba) y haga clic en el botón Abrir carpeta para abrir la carpeta vacía. carpeta ch1.

7. Si aparece un cuadro de diálogo que pregunta ¿Confía en los autores de los archivos en esta carpeta?, marque Confiar en los autores de todos los archivos en la carpeta principal 'Full-Stack-React-Projects' y luego haga clic en el botón Sí, confío en los autores .

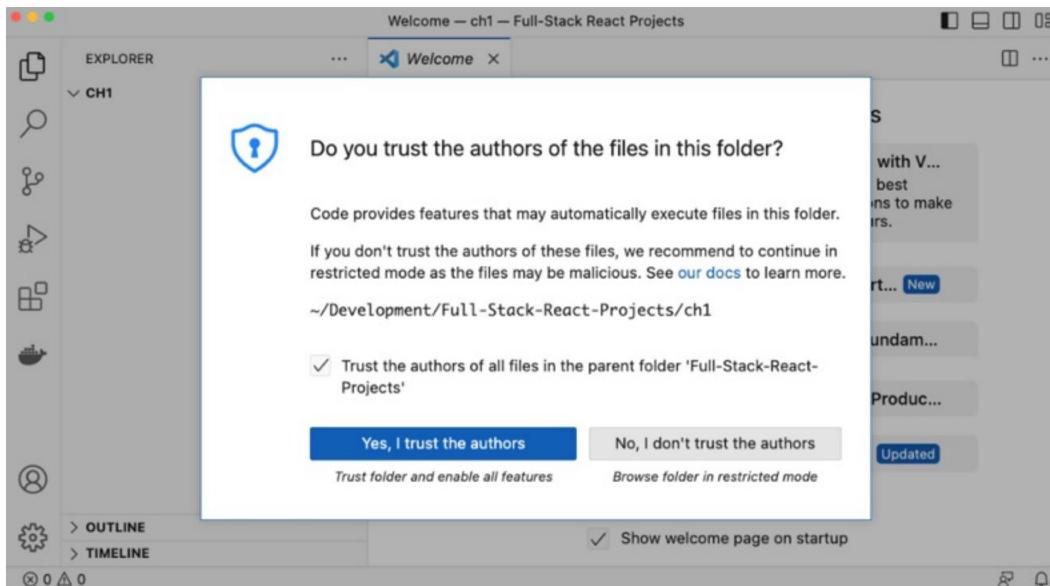


Figura 1.3 – Permitir que VS Code ejecute archivos en nuestra carpeta de proyecto

Consejo

Puede ignorar esta advertencia en sus proyectos, ya que puede estar seguro de que no contienen código malicioso. Al abrir carpetas de fuentes no confiables, puede presionar "No, no confío en los autores" y aun así explorar el código. Sin embargo, al hacerlo, se deshabilitarán algunas funciones de VS Code.

Hemos configurado VS Code correctamente y estamos listos para empezar a configurar nuestro proyecto. Si clonaste la carpeta de los ejemplos de código de GitHub proporcionados, también aparecerá una notificación indicando que se encontró un repositorio de Git. Puedes cerrarla, ya que solo queremos abrir la carpeta ch1.

Ahora que VS Code está listo, continuemos configurando un nuevo proyecto con Vite.

Configurar un proyecto con Vite

En este libro, usaremos Vite para configurar nuestro proyecto, ya que es el más popular y apreciado según la encuesta "State of JS 2022" (<https://2022.stateofjs.com/>). Vite también facilita la configuración de un proyecto frontend moderno, permitiendo ampliar la configuración posteriormente si es necesario. Sigue estos pasos para configurar tu proyecto con Vite:

1. En la barra de menú de VS Code, vaya a Terminal | Nueva terminal para abrir una nueva terminal.
2. Dentro de la Terminal, ejecute el siguiente comando:

```
$npm create vite@5.0.0 .
```

Asegúrese de que haya un punto al final del comando para crear el proyecto en la carpeta actual en lugar de crear una nueva carpeta.

Nota

Para que las instrucciones de este libro sigan funcionando incluso con nuevas versiones, fijamos todos los paquetes a una versión fija. Siga las instrucciones con las versiones indicadas. Después de terminar este libro, al iniciar nuevos proyectos por su cuenta, intente siempre usar las versiones más recientes, pero tenga en cuenta que podrían ser necesarios cambios para que funcionen. Consulte la documentación de los paquetes correspondientes y siga la ruta de migración desde la versión del libro a la más reciente.

3. Cuando se le pregunte si debe instalarse create-vite, simplemente escriba y y presione Retorno/Intro Tecla para continuar.
4. Cuando se le pregunte sobre el framework, use las flechas para seleccionar React y presione Enter. Si se le solicita el nombre del proyecto, presione Ctrl + C para cancelar y luego ejecute el comando de nuevo, asegurándose de que haya un punto al final para seleccionar la carpeta actual.
5. Cuando se le pregunte sobre la variante, seleccione JavaScript.
6. Ahora, nuestro proyecto está configurado y podemos ejecutar npm install para instalar las dependencias.

7. Posteriormente, ejecute npm run dev para iniciar el servidor de desarrollo, como se muestra en la siguiente captura de pantalla:



The screenshot shows the VS Code interface with the title bar "ch1 — Full-Stack React Projects". The left sidebar (EXPLORER) shows a project structure with files like node_modules, public, src, .gitignore, index.html, package-lock.json, package.json, and vite.config.js. The right side has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, displaying the following command history and output:

```
npm install
npm run dev
added 83 packages, and audited 84 packages in 7s
8 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities
npm run dev
```

Figura 1.4 – La terminal después de configurar un proyecto con Vite y antes de iniciar el servidor de desarrollo

Nota

Para simplificar la configuración, usamos npm directamente. Si prefieres yarn o pnpm, puedes ejecutar yarn create vite o pnpm create vite, respectivamente.

8. En la Terminal, verás una URL que indica dónde se ejecuta tu aplicación. Puedes mantener presionada la tecla Ctrl (Cmd en macOS) y hacer clic en el enlace para abrirlo en tu navegador, o introducir la URL manualmente en un navegador.

9. Para probar si su aplicación es interactiva, haga clic en el botón con el texto “el conteo es 0” y el conteo debería aumentar cada vez que lo presione.

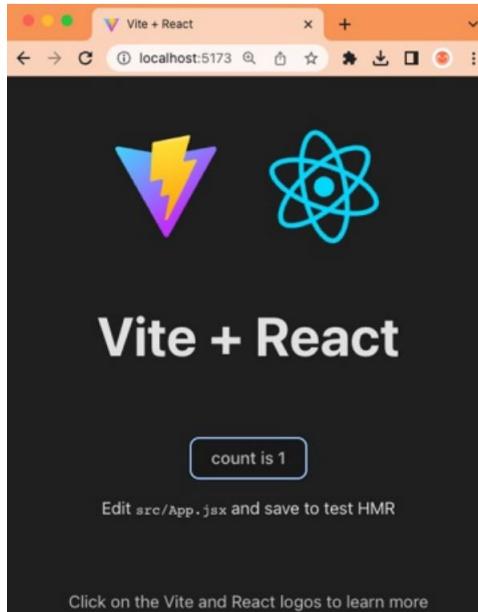


Figura 1.5 – Nuestra primera aplicación React ejecutándose con Vite

Alternativas a Vite

Las alternativas a Vite son los empaquetadores, como webpack, Rollup y Parcel. Estos son altamente configurables, pero a menudo no ofrecen una gran experiencia para los servidores de desarrollo. Primero deben empaquetar todo nuestro código antes de servirlo al navegador. En cambio, Vite es compatible de forma nativa con el estándar de módulos ECMAScript (ESM). Además, Vite requiere muy poca configuración para comenzar. Una desventaja de Vite es que puede ser difícil configurar ciertos escenarios más complejos con él. Un empaquetador prometedor es Turbopack; sin embargo, todavía es muy nuevo al momento de escribir este artículo. Para el desarrollo integral con renderizado del lado del servidor, más adelante conoceremos Next.js, que es un framework de React que también proporciona un servidor de desarrollo listo para usar.

Ahora que nuestro proyecto repetitivo está en funcionamiento, dediquemos algún tiempo a configurar herramientas que aplicarán las mejores prácticas y un estilo de código consistente.

Configuración de ESLint y Prettier para aplicar las mejores prácticas y el estilo del código

Ahora que nuestra aplicación React está configurada, vamos a configurar ESLint para aplicar las mejores prácticas de codificación con JavaScript y React. También vamos a configurar Prettier para aplicar un estilo de código y formatear nuestro código automáticamente.

Instalación de las dependencias necesarias

Primero, vamos a instalar todas las dependencias necesarias:

1. En la Terminal, haga clic en el ícono Dividir Terminal en la parte superior derecha del panel de Terminal para crear un nuevo panel de Terminal . \$is mantendrá nuestra aplicación ejecutándose mientras ejecutamos otros comandos.
2. Haga clic en este panel recién abierto para enfocarlo. \$en, ingrese el siguiente comando para instalar ESLint, Prettier y los complementos relevantes:

```
$ npm install --save-dev prettier@3.1.0 \
  eslint@8.54.0 \
  eslint-plugin-react@7.33.2 \
  eslint-config-prettier@9.0.0 \
  eslint-plugin-jsx-a11y@6.8.0
```

Los paquetes instalados son los siguientes:

Prettier: formatea nuestro código automáticamente según un estilo de código definido

eslint: analiza nuestro código y aplica las mejores prácticas

eslint-config-react: Habilita reglas en ESLint relevantes para proyectos React

eslint-config-prettier: deshabilita las reglas relacionadas con el estilo del código en ESLint para que Las más bonitas pueden manejarlos en su lugar

eslint-plugin-jsx-a11y: permite que ESLint verifique problemas de accesibilidad (a11y) en nuestro código JSX

Nota

\$e --save-dev %ag en npm guarda esas dependencias como dependencias de desarrollo, lo que significa que solo se instalarán para el desarrollo. No se instalarán ni se incluirán en una aplicación implementada. Esto es importante para mantener el tamaño de nuestros contenedores lo más pequeño posible más adelante.

Tras instalar las dependencias, debemos configurar Prettier y ESLint. Comenzaremos configurando Prettier.

Configurando Prettier

Prettier formateará el código para nosotros y reemplazará el formateador de código predeterminado para JavaScript en VS Code. Nos permitirá dedicar más tiempo a escribir código, ya que se formatea automáticamente al guardar el archivo. Sigue estos pasos para configurar Prettier:

1. Haga clic derecho debajo de la lista de archivos en la barra lateral izquierda de VS Code (si no está abierto, haga clic en Archivos icono) y pulsa "Nuevo archivo..." para crear un nuevo archivo. Llámalo `.prettierrc.json` (¡no olvides el punto al principio del nombre del archivo!).
2. El archivo recién creado debería abrirse automáticamente, para que podamos empezar a escribir la siguiente configuración . Primero, creamos un nuevo objeto y configuramos la opción `trailingComma` en `'all'` para asegurarnos de que los objetos y arrays que abarcan varias líneas siempre tengan una coma al final, incluso en el último elemento. Esto reduce el número de líneas que se tocan al confirmar un cambio mediante Git:

```
{  
  "trailingComma": "todos",
```

3. \$en, establecemos la opción `tabWidth` a 2 espacios:

```
  "Ancho de pestaña": 2,
```

4. Establezca `printWidth` en 80 caracteres por línea para evitar líneas largas en nuestro código:

```
  "Ancho de impresión": 80,
```

5. Establezca la opción `semi` en falso para evitar puntos y comas donde no sea necesario:

```
  "semi": falso,
```

6. Por último, reforzamos el uso de comillas simples en lugar de comillas dobles:

```
  "jsxSingleQuote": verdadero,  
  "singleQuote": verdadero  
}
```

Nota

Estas configuraciones para Prettier son solo un ejemplo de una convención de estilo de codificación. Por supuesto, puedes ajustarlas a tus preferencias. Hay muchas más opciones, todas ellas disponibles en la documentación de Prettier (<https://prettier.io/docs/en/options.html>).

Configurando la extensión Prettier

Ahora que tenemos un archivo de configuración para Prettier, debemos asegurarnos de que la extensión VS Code esté configurada correctamente para formatear el código para nosotros:

1. Abra la configuración de VS Code yendo a Archivo | Preferencias... | Configuración en Windows/Linux, o Código | Configuración... | Configuración en macOS.
2. En el editor de configuración recién abierto, haga clic en la pestaña Espacio de trabajo . \$is garantiza que guardemos toda nuestra configuración en un archivo .vscode/settings.json en la carpeta de nuestro proyecto. Cuando otros desarrolladores abran nuestro proyecto, también usarán automáticamente esa configuración.
3. Busque el formato del editor al guardar y marque la casilla de verificación para habilitar el código de formato. al guardar.
4. Busque el formateador predeterminado del editor y seleccione Prettier - Formateador de código de la lista.
5. Para verificar el funcionamiento de Prettier, abra el archivo .prettierrc.json, añada algunos espacios al principio de una línea y guarde el archivo. Observe que Prettier ha reformateado el código para ajustarse al estilo definido. Esto reducirá el número de espacios para la sangría a dos.

Ahora que Prettier está configurado correctamente, ya no tenemos que preocuparnos por formatear el código manualmente . ¡Escribe el código sobre la marcha y guarda el archivo para formatearlo!

Creando un archivo de ignorados de Prettier

Para mejorar el rendimiento y evitar ejecutar Prettier en archivos que no deben formatearse automáticamente, podemos ignorar ciertos archivos y carpetas creando un archivo de Prettier que se ignore. Siga estos pasos:

1. Cree un nuevo archivo llamado .prettierignore en la raíz de nuestro proyecto, similar a cómo creamos el archivo .prettierrc.json.
2. Agregue el siguiente contenido para ignorar el código fuente transpilado:

```
dist/
```

La carpeta \$e node_modules/ es ignorada automáticamente por Prettier.

Ahora que hemos configurado exitosamente Prettier, vamos a configurar ESLint para aplicar las mejores prácticas de codificación.

Configuración de ESLint

Mientras que Prettier se centra en el estilo y el formato de nuestro código, ESLint se centra en el código en sí, evitando errores comunes o código innecesario. Vamos a configurarlo ahora:

1. Elimine el archivo `.eslintrc.cjs` creado automáticamente.
2. Cree un nuevo archivo `.eslintrc.json` y comience a escribir la siguiente configuración. Primero, configure la raíz como verdadera para asegurar que ESLint no busque más configuración en las carpetas principales:

```
{  
  "raíz": verdadero,
```

3. Definir un objeto `env`, en el que establecemos el entorno del navegador como verdadero para que ESLint entienda variables globales específicas del navegador, como documento y ventana:

```
"env": {  
  "navegador": verdadero  
},
```

4. Definimos un objeto `parserOptions`, donde especificamos que estamos usando el último ECMAScript Versión y ESM:

```
"parserOptions": {  
  "ecmaVersion": "última versión",  
  "sourceType": "módulo"  
},
```

5. Defina una matriz de extensiones para extender las configuraciones recomendadas. En concreto, extendemos las reglas recomendadas de ESLint y las de los plugins instalados:

```
"se extiende": [  
  "eslint:recomendado",  
  "plugin:react/recomendado",  
  "complemento:react/jsx-runtime",  
  "plugin:jsx-a11y/recomendado",
```

6. Como último elemento de la matriz, usamos prettier para deshabilitar todas las reglas relacionadas con el estilo de código en ESLint y deja que Prettier se encargue:

```
"más bonita"  
],
```

7. Ahora, definimos la configuración de los plugins. Primero, le indicamos al plugin de React que detecte automáticamente la versión de React instalada:

```
"ajustes": {  
  "reaccionar": {
```

```
        "versión": "detectar"  
    }  
},
```

8. Finalmente, fuera de la sección de configuración, definimos una matriz de anulaciones, en la que especificamos que ESLint solo debería analizar archivos .js y .jsx:

```
"anulaciones": [  
    {  
        "archivos": ["*.js", "*.jsx"]  
    }  
]
```

9. Cree un nuevo archivo `.eslintignore`, con el siguiente contenido:

```
dist/  
vite.config.js
```

ESLint ignora automáticamente la carpeta `$e node_modules/`.

10. Guarde los archivos y ejecute `npx eslint src` en la terminal para ejecutar el linter. Verá que ya hay algunos errores debido a que nuestras reglas configuradas no coinciden con el código fuente proporcionado por el proyecto predeterminado en Vite:

```
① ➔ ~/D/F/ch1 ↵ main: > npx eslint src  
/Users/dan/Development/Full-Stack-React-Projects/ch1/src/App.jsx  
 12:9  error  Using target="_blank" without rel="noopener" (which implies rel="noopener") is a security risk in  
  older browsers: see https://mathiasbynens.github.io/rel-noopener/#recommendations  react/jsx-no-target-blank  
 15:9  error  Using target="_blank" without rel="noopener" (which implies rel="noopener") is a security risk in  
  older browsers: see https://mathiasbynens.github.io/rel-noopener/#recommendations  react/jsx-no-target-blank  
  
✖ 2 problems (2 errors, 0 warnings)  
 2 errors and 0 warnings potentially fixable with the `--fix` option.
```

Figura 1.6 – Al ejecutar ESLint por primera vez, obtenemos algunos errores sobre violaciones de reglas

11. Afortunadamente, ESLint corrige automáticamente todos estos problemas. Ejecute `npx eslint src --fix` para corregirlos automáticamente. Ahora, al ejecutar `npx eslint src` de nuevo, no obtendrá ningún resultado. \$is significa que no hubo errores de linter.

Consejo

El comando `$e npx` permite ejecutar comandos proporcionados por paquetes npm, de forma similar a como se ejecutan en scripts `package.json`. También permite ejecutar paquetes remotos sin instalarlos permanentemente. Si el paquete aún no está instalado, se le preguntará si desea hacerlo.

Agregar un nuevo script para ejecutar nuestro linter

En la sección anterior, hemos estado llamando al linter ejecutando npx eslint src manualmente.

Ahora vamos a agregar un script de pelusa a package.json:

1. En la Terminal, ejecute el siguiente comando para definir un script de lint en el archivo package.json:

```
$ npm pkg set scripts.lint="eslint src"
```

2. Ahora, ejecute npm run lint en la terminal. \$is debería ejecutar eslint src con éxito.

Tal como lo hizo npx eslint src:

```
● → ~/D/F/ch1 ↵ main± > npm run lint
> ch1@0.0.0 lint
> eslint src
○ → ~/D/F/ch1 ↵ main± > █
```

Figura 1.7 – El linter se ejecuta correctamente, sin errores

Después de configurar ESLint y Prettier, aún debemos asegurarnos de que se ejecuten antes de confirmar el código.

Configuremos Husky para asegurarnos de confirmar el código correcto ahora.

Configurar Husky para asegurarnos de que enviamos el código correcto

Después de configurar Prettier y ESLint, ahora nuestro código se formateará automáticamente al guardarlo mediante Prettier y veremos errores de ESLint en VS Code cuando cometamos errores o ignoremos las mejores prácticas. Sin embargo, podríamos pasar por alto algunos de estos errores y enviar accidentalmente código inválido. Para evitarlo, podemos configurar Husky y lint-staged, que se ejecutan antes de enviar nuestro código a Git y garantizan que Prettier y ESLint se ejecuten correctamente en el código fuente antes de enviarlo.

Importante

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

Configuremos Husky y lint-staged siguiendo estos pasos:

1. Ejecute el siguiente comando para instalar Husky y lint-staged como dependencias de desarrollo:

```
$ npm install --save-dev husky@8.0.3 \
pelusa-escenificada@15.1.0
```

2. Abra el archivo package.json y agregue la siguiente configuración preconfigurada en un nuevo objeto después de devDependencies, luego guarde el archivo. \$Is ejecutará Prettier y ESLint en todos los archivos .js y .jsx confirmados e intentará corregir automáticamente el estilo del código y los errores de linter, si es posible:

```
"preparado para pelusa": {  
  "**/*.{js.jsx)": [  
    "npx más bonito --write",  
    "npx eslint --fix"  
  ]  
}
```

3. Inicialice un repositorio Git en la carpeta ch1 y realice una confirmación inicial solo con el paquete.

json le, ya que lint-staged no se ejecuta en la confirmación inicial:

```
$ git init  
$ git add paquete.json  
$ git commit -m "tarea: confirmación inicial"
```

4. Agregue el script de instalación de Husky a un script de preparación en package.json, para que Husky se instale automáticamente cuando se clone el proyecto y se ejecute npm install:

```
$ npm pkg set scripts.prepare="instalación husky"
```

5. Dado que no necesitamos ejecutar npm install nuevamente en este momento, debemos ejecutar manualmente el script de preparación esta vez:

```
$ npm run prepare
```

6. Agregue un gancho de pre-confirmación para lint-staged, de modo que ESLint y Prettier se ejecuten cada vez que lo hagamos. confirmación de git:

```
$ npx husky agregar .husky/pre-commit "npx lint-staged"
```

7. Ahora, agrega todos los archivos a Git e intenta realizar una confirmación:

```
$git add .  
$ git commit -m "tarea: configuración básica del proyecto"
```

Si todo funcionó correctamente, deberías ver a husky ejecutando lint-staged, que, a su vez, ejecuta prettier y eslint después de ejecutar git commit. Si recibes un error de configuración, asegúrate de que todos los archivos se hayan guardado correctamente y vuelve a ejecutar git commit.

```
● ➔ ~/D/F/ch1 ↵ main± > git commit -m "chore: basic project setup"
✓ Preparing lint-staged...
✓ Running tasks for staged files...
✓ Applying modifications from tasks...
✓ Cleaning up temporary files...
[main f226bcd] chore: basic project setup
```

Figura 1.8 – Husky y lint-staged aplican con éxito el estilo del código y las mejores prácticas antes de confirmar

Configuración de commitlint para aplicar un estándar para nuestros mensajes de confirmación

Además de revisar nuestro código, también podemos revisar nuestros mensajes de confirmación. Quizás hayas notado que ya prefijamos un tipo a nuestros mensajes de confirmación (el tipo de tarea). Los tipos facilitan el seguimiento de los cambios en una confirmación. Para forzar el uso de tipos, podemos configurar commitlint. Sigue estos pasos para configurarlo:

1. Instale commitlint y una configuración convencional para commitlint:

```
$ npm install --save-dev @commitlint/cli@18.4.3 \
@commitlint/config-conventional@18.4.3
```

2. Cree un nuevo archivo `.commitlintrc.json` en la raíz de nuestro proyecto y agregue el siguiente contenido:

```
{
  "extiende": ["@commitlint/config-conventional"]
}
```

3. Agregue un gancho `commit-msg` a Husky:

```
$ npx husky agregar .husky/commit-msg \
'npx commitlint --edit ${1}'
```

4. Ahora, si intentamos agregar nuestros archivos modificados y confirmar sin un tipo o con un tipo incorrecto, recibiremos un error de commitlint y no podremos realizar dicha confirmación. Si agregamos el tipo correcto, se realizará correctamente:

```
$git add .
$ git commit -m "sin tipo"
$ git commit -m "incorrecto: tipo"
$ git commit -m "tarea: configurar commitlint"
```

La siguiente figura muestra a Husky en acción. Si escribimos un mensaje de confirmación incorrecto, lo rechazará y no nos permitirá confirmar el código. Solo si ingresamos un mensaje de confirmación con el formato correcto, la confirmación se procesará:

```

④ → ~/D/F/ch1 ↵ main± > git commit -m "no type"
→ No staged files match any configured task.
✗  input: no type
✗  subject may not be empty [subject-empty]
✗  type may not be empty [type-empty]

✗  found 2 problems, 0 warnings
① Get help: https://github.com/conventional-changelog/commitlint/#what-is-commitlint

husky - commit-msg hook exited with code 1 (error)
④ → ~/D/F/ch1 ↵ main± > git commit -m "wrong: type"
→ No staged files match any configured task.
✗  input: wrong: type
✗  type must be one of [build, chore, ci, docs, feat, fix, perf, refactor, revert, style, test] [type-enum]

✗  found 1 problems, 0 warnings
① Get help: https://github.com/conventional-changelog/commitlint/#what-is-commitlint

husky - commit-msg hook exited with code 1 (error)
● → ~/D/F/ch1 ↵ main± > git commit -m "chore: configure commitlint"
→ No staged files match any configured task.
[main e5e8cd0] chore: configure commitlint
 4 files changed, 2019 insertions(+), 57 deletions(-)
 create mode 100644 .commitlintrc.json
 create mode 100755 .husky/commit-msg

```

Figura 1.9 – commitlint funcionando correctamente y previniendo confirmaciones sin un tipo y con tipos incorrectos

Mensajes de confirmación en la configuración convencional de commitlint (<https://www.conventionalcommits.org/>) están estructurados de tal manera que primero se debe listar un tipo, luego un ámbito opcional y finalmente la descripción, como type(scope): description. Los tipos posibles son los siguientes:

- Corrección: Para corregir errores
- hazaña: Para nuevas funciones
- refactorizar: para reestructurar el código sin agregar características ni corregir errores
- compilación: para cambios en el sistema de compilación o dependencias
- ci: Para cambios en la configuración de CI/CD
- docs: Solo para cambios en la documentación
- perf: Para optimizaciones de rendimiento
- estilo: para formatear el código xing
- prueba: Para agregar o ajustar pruebas

El alcance es opcional y se utiliza mejor en un monorepositorio para especificar que se realizaron cambios en una determinada aplicación o biblioteca dentro de él.

Resumen

Ahora que hemos configurado nuestro proyecto correctamente y hemos comenzado a aplicar los estándares, podemos seguir trabajando en él sin preocuparnos por un estilo de código consistente, mensajes de confirmación consistentes ni por cometer pequeños errores. ESLint, Prettier, Husky y commitlint nos cubren las espaldas.

En el próximo capítulo, Capítulo 2, Conociendo Node.js y MongoDB, aprenderemos cómo escribir y ejecutar pequeños scripts de Node.js y cómo funciona MongoDB, un sistema de base de datos.

2

Conociendo Node.js y MongoDB

En el capítulo anterior, configuramos nuestro IDE y un proyecto básico para el desarrollo frontend. En este capítulo, primero aprenderemos a escribir y ejecutar scripts con Node.js. Luego presentaremos Docker para configurar un servicio de base de datos. Una vez configurado Docker y un contenedor para nuestra base de datos, accederemos a él para aprender más sobre MongoDB, la base de datos de documentos que usaremos de ahora en adelante. Finalmente, conectaremos todo lo aprendido en este capítulo accediendo a MongoDB mediante scripts de Node.js.

Al final de este capítulo, comprenderá las herramientas y los conceptos más importantes en el desarrollo de backend con JavaScript. Este capítulo nos brinda una buena base para crear un servicio de backend para nuestra primera aplicación full-stack en los próximos capítulos.

En este capítulo cubriremos los siguientes temas principales:

- Escribir y ejecutar scripts con Node.js
- Presentamos Docker, una plataforma para contenedores
- Presentamos MongoDB, una base de datos de documentos
- Acceso a la base de datos MongoDB a través de Node.js

Requisitos técnicos

Antes de comenzar, instale lo siguiente (además de todos los requisitos técnicos del Capítulo 1, Preparación para el desarrollo full-stack), si aún no los tiene instalados:

- Docker v24.0.6
- Docker Desktop v4.25.2
- MongoDB Shell v2.1.0

Las versiones mencionadas son las que se usan en el libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que algunos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/capítulo2>.

El video de \$e CiA para este capítulo se puede encontrar en: https://youtu.be/q_LHsdJEaPo.

Importante

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

Escritura y ejecución de scripts con Node.js

Para convertirnos en desarrolladores full-stack, es importante familiarizarnos con las tecnologías backend. Como ya conocemos JavaScript por haber escrito aplicaciones frontend, podemos usar Node.js para desarrollar servicios backend con JavaScript. En esta sección, crearemos nuestro primer script sencillo en Node.js para familiarizarnos con las diferencias entre los scripts backend y el código frontend.

Similitudes y diferencias entre JavaScript en el navegador y en Node.js

Node.js se basa en V8, el motor de JavaScript utilizado por los navegadores basados en Chromium (Google Chrome, Brave, Opera, Vivaldi y Microsoft Edge). Por lo tanto, el código JavaScript se ejecuta de la misma manera en el navegador y en Node.js. Sin embargo, existen algunas diferencias, especialmente en el entorno. El entorno se basa en el motor y permite renderizar contenido de un sitio web en el navegador (utilizando los objetos de documento y ventana). En Node.js, se proporcionan ciertos módulos para interactuar con el sistema operativo, para tareas como la creación de archivos y la gestión de solicitudes de red. Estos módulos permiten crear un servicio backend utilizando Node.js.

Echemos un vistazo a la arquitectura de Node.js versus JavaScript en el navegador:

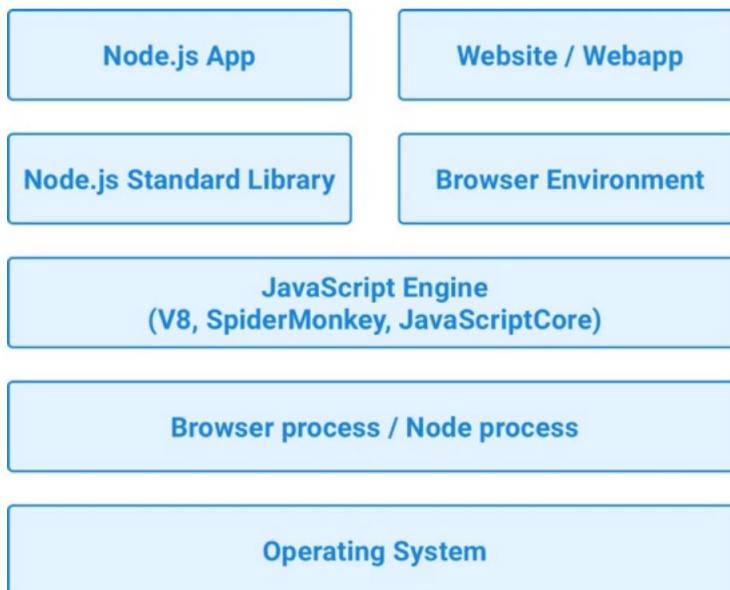


Figura 2.1 – La arquitectura de Node.js versus JavaScript en el navegador

Como podemos ver en la visualización, tanto Node.js como el JavaScript del navegador se ejecutan en un motor JavaScript, que siempre es V8 en Node.js, y puede ser V8 para navegadores basados en Chromium, SpiderMonkey para Firefox o JavaScriptCore para Safari.

Ahora que sabemos que podemos ejecutar código JavaScript en Node.js, ¡probémoslo!

Creando nuestro primer script de Node.js

Antes de que podamos comenzar a escribir servicios backend, necesitamos familiarizarnos con el entorno Node.js.

Entonces, comencemos escribiendo un ejemplo simple de "hola mundo":

1. Copie la carpeta ch1 del capítulo anterior a una nueva carpeta ch2, de la siguiente manera:

```
$ cp -R ch1 ch2
```

Nota

En macOS, es importante ejecutar el comando con `-R %ag` en mayúscula, no `-r`. `$e -r %ag` trata de manera diferente los enlaces simbólicos y provoca que la carpeta `node_modules/` se rompa. `$e -r %ag` solo existe por razones históricas y no debe usarse en macOS. Siempre es preferible usar `-R %ag`.

2. Abra la nueva carpeta ch2 en VS Code.

3. Cree una nueva carpeta de backend en la carpeta ch2. \$is contendrá nuestro código de backend.

4. En la carpeta backend, cree un archivo helloworld.js e ingrese el siguiente código:

```
console.log('¡Hola mundo node.js!')
```

5. Abra una terminal en la carpeta ch2 y ejecute el siguiente comando para ejecutar el script Node.js:

```
$ nodo backend/helloworld.js
```

Verás que la salida de la consola muestra ¡Hola mundo Node.js!. Al escribir código Node.js, podemos usar funciones conocidas del entorno JavaScript frontend y ejecutar el mismo código. ¡Código JavaScript en el backend!

Nota

Si bien la mayoría del código JavaScript del frontend se ejecutará correctamente en Node.js, no todo el código del frontend funcionará automáticamente en un entorno Node.js. Hay ciertos objetos, como documentos y ventana, que son específicas de un entorno de navegador. Es importante tener esto en cuenta, especialmente cuando introducimos la representación del lado del servidor más adelante.

Ahora que tenemos una comprensión básica de cómo funciona Node.js, comenzemos a manejar archivos con Node.js.

Manejo de archivos en Node.js

A diferencia del entorno de navegador, Node.js ofrece funciones para gestionar archivos en nuestro ordenador mediante el módulo node:fs (sistema de archivos). Por ejemplo, podríamos usar esta funcionalidad para leer y escribir varios archivos o incluso usarlos como una simple base de datos.

Siga estos pasos para crear su primer script Node.js que maneja archivos:

1. Cree un nuevo archivo backend/files.js.
2. Importe las funciones writeFileSync y readFileSync desde el módulo interno Node.js node:fs. No es necesario instalar este módulo a través de npm, ya que lo proporciona el entorno de ejecución de Node.js.

```
importar { writeFileSync, readFileSync } desde 'node:fs'
```

3. Cree una matriz simple que contenga usuarios, con un nombre y una dirección de correo electrónico:

```
const users = [{ nombre: 'Adam Ondra', correo electrónico: 'adam.ondra@climb.ing' }]
```

4. Antes de poder guardar esta matriz en un archivo, primero debemos convertirla en una cadena usando JSON. convertir en cadena:

```
const usersJson = JSON.stringify(users)
```

5. Ahora podemos guardar nuestra cadena JSON en un archivo usando la función `writeFileSync`. Esta función toma dos argumentos: primero el nombre del archivo y luego la cadena que se escribirá en el archivo:

```
writeFileSync('backend/usuarios.json', usuariosJson)
```

6. Después de escribir en el archivo, podemos intentar leerlo nuevamente usando `readFileSync` y analizando la cadena JSON usando `JSON.parse`:

```
constante readUsersJson = readFileSync('backend/usuarios.json')
const leerUsuarios = JSON.parse(leerUsuariosJson)
```

7. Finalmente, registramos la matriz analizada:

```
console.log(leerUsuarios)
```

8. Ahora podemos ejecutar nuestro script. Verá que se registra la matriz y se crea un archivo `users.json` en nuestra carpeta `backend/`:

```
$ nodo backend/files.js
```

Es posible que hayas notado que hemos estado usando `writeFileSync` y no `writeFile`. El comportamiento predeterminado en Node.js es ejecutar todo de forma asíncrona, lo que significa que si usamos `writeFile`, es posible que el archivo no se haya creado aún en el momento en que llamamos a `readFile`, ya que el código asíncrono no se ejecuta en orden.

Este comportamiento puede resultar molesto cuando se escriben scripts simples como hicimos nosotros, pero es muy útil cuando se trata, por ejemplo, con solicitudes de red, donde no queremos bloquear a otros usuarios para que no accedan a nuestro servicio mientras tratamos con otra solicitud.

Después de aprender a manejar archivos con Node.js, aprendamos más sobre cómo se ejecuta el código asíncrono en el navegador y Node.js.

Concurrencia con JavaScript en el navegador y Node.js

Una característica esencial y especial de JavaScript es que la mayoría de las funciones de la API son asíncronas por defecto. `$is` significa que el código no se ejecuta simplemente en la secuencia definida. En concreto, JavaScript está basado en eventos. En el navegador, esto significa que el código JavaScript se ejecutará debido a las interacciones del usuario. Por ejemplo, cuando se hace clic en un botón, definimos un controlador `onClick` para ejecutar algún código.

En el lado del servidor, las operaciones de entrada/salida, como la lectura y escritura de archivos, y las solicitudes de red, se gestionan de forma asíncrona. `$is` significa que podemos gestionar múltiples solicitudes de red simultáneamente, sin tener que gestionar subprocesos ni multiprocesamiento. Específicamente, en Node.js, libuv se encarga de asignar subprocesos para las operaciones de E/S, a la vez que nos da, como programadores, acceso a un único subproceso en tiempo de ejecución para escribir nuestro código. Sin embargo, esto no significa que cada conexión a nuestro backend cree un nuevo subproceso. Las `$reads` se crean en el %y cuando resulta ventajoso. Como desarrolladores, no tenemos que lidiar con multiprocesos y podemos centrarnos en desarrollar con código asíncrono y devoluciones de llamadas.

Si el código es síncrono, se ejecuta directamente colocándolo en la pila de llamadas. Si es asíncrono, se inicia la operación y la instancia de dicha operación se almacena en una cola, junto con una función de devolución de llamada. El entorno de ejecución de Node.js ejecutará primero todo el código en la pila. El bucle de eventos entrará en acción y comprobará si hay tareas completadas en la cola. En ese caso, la función de devolución de llamada se ejecuta colocándola en la pila. Una función de devolución de llamada puede ejecutar código síncrono o asíncrono. Al añadir un detector de eventos, por ejemplo, un onClick

Oyente en el navegador: cuando el usuario hace clic en el elemento relacionado, la devolución de llamada también se coloca en la cola de tareas, lo que significa que se ejecutará cuando no haya nada más en la pila. De forma similar, en Node.js, podemos agregar oyentes para eventos de red y ejecutar una devolución de llamada cuando llega una solicitud.

A diferencia de los servidores multihilo, un servidor Node.js acepta todas las solicitudes en un solo hilo, que contiene el bucle de eventos. Los servidores multihilo tienen la desventaja de que los hilos pueden bloquear la E/S por completo y ralentizar el servidor. Sin embargo, Node.js delega las operaciones de forma precisa en los % y a los hilos. Esto reduce el bloqueo de las operaciones de E/S por defecto. La desventaja de Node.js es que tenemos menos control sobre el funcionamiento del multihilo y, por lo tanto, debemos evitar el uso de funciones síncronas siempre que sea posible. De lo contrario, bloquearemos el hilo principal de Node.js y ralentizaremos el servidor. Para simplificar, en este capítulo seguiremos utilizando funciones síncronas. En los próximos capítulos, evitaremos su uso y nos basaremos únicamente en funciones asíncronas (cuando sea posible) para obtener el mejor rendimiento.

El siguiente diagrama visualiza la diferencia entre servidores multiproceso y un servidor Node.js:

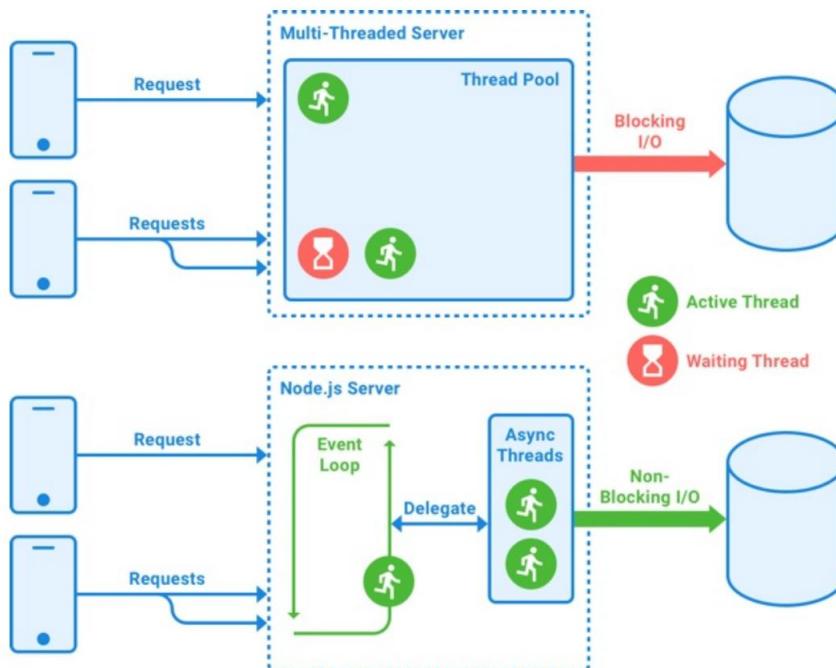


Figura 2.2 – La diferencia entre servidores multiproceso y un servidor Node.js

Podemos ver esta asincronía en acción usando `setTimeout`, una función que quizás conozcas del código frontend. Espera un número específico de milisegundos y luego ejecuta el código especificado en la función de devolución de llamada. Por ejemplo, si ejecutamos el siguiente código (con un script de Node.js o en el navegador, el resultado es el mismo para ambos):

```
console.log('primero')
establecerTiempo de espera(() => {
  console.log('segundo')
}, 1000)
console.log('tercero')
```

Podemos ver que se imprimen en el siguiente orden:

```
primero
tercero
segundo
```

\$is tiene sentido, ya que estamos retrasando el "segundo" `console.log` un segundo. Sin embargo, se obtendrá el mismo resultado si ejecutamos el siguiente código:

```
console.log('primero')
establecerTiempo de espera(() => {
  console.log('segundo')
}, 0)
console.log('tercero')
```

Ahora que esperamos cero milisegundos antes de ejecutar el código, se podría pensar que "segundo" se imprime después de "primero". Sin embargo, no es así. En cambio, obtenemos el mismo resultado que antes:

```
primero
tercero
segundo
```

La razón es que, al usar `setTimeout`, el motor de JavaScript llama a una API web (en el navegador) o a una API nativa (en Node.js). Esta API se ejecuta en código nativo en el motor, registra internamente el tiempo de espera y envía la devolución de llamada a la cola de tareas, ya que el temporizador se completa inmediatamente. Mientras tanto, el motor de JavaScript continúa procesando el resto del código, insertándolo en la pila y ejecutándolo. Cuando la pila está vacía (no hay más código que ejecutar), el bucle de eventos avanza. Detecta que hay algo en la cola de tareas, por lo que ejecuta ese código, lo que resulta en que "second" se imprima al final.

Consejo

Puede utilizar la herramienta Lupa para visualizar el funcionamiento interno de la pila de llamadas, las API web, el bucle de eventos y la cola de tareas/devolución de llamadas: <http://latentflip.com/loupe/>

Ahora que hemos aprendido cómo se maneja el código asíncrono en el navegador y Node.js, ¡creemos nuestro primer servidor web con Node.js!

Creando nuestro primer servidor web

Ahora que conocemos los fundamentos de Node.js, podemos usar la biblioteca node:http para crear un servidor web sencillo. Para nuestro primer servidor sencillo, simplemente devolveremos una respuesta 200 OK y texto plano en cualquier solicitud. Comencemos con los pasos:

1. Cree un nuevo archivo backend/simpleweb.js, ábralo e importe createServer

función del módulo node:http:

```
importar { createServer } desde 'node:http'
```

2. La función `\$e createServer` es asíncrona, por lo que requiere que le pasemos una función de devolución de llamada .

Esta función se ejecutará cuando llegue una solicitud del servidor. Tiene dos argumentos: un objeto de solicitud (req) y un objeto de respuesta (res). Utilice la función `createServer` para definir un nuevo servidor:

```
const servidor = crearServidor((req, res) => {
```

3. Por ahora, ignoraremos el objeto de solicitud y solo devolveremos una respuesta estática. Primero, establecemos el código de estado en 200:

```
res.código de estado = 200
```

4. \$en, establecemos el encabezado Content-Type en text/plain, de modo que el navegador sepa con qué tipo de datos de respuesta está tratando:

```
res.setHeader('Tipo de contenido', 'texto sin formato')
```

5. Por último, finalizamos la solicitud devolviendo una cadena HTTP "¡Hola mundo!" en la respuesta:

```
res.end('¡Hola mundo HTTP!')  
})
```

6. Despues de definir el servidor, debemos asegurarnos de escuchar en un host y puerto específicos. Esto definirá dónde estará disponible el servidor. Por ahora, usamos localhost en el puerto 3000 para asegurarnos de que nuestro servidor esté disponible a través de http://localhost:3000/.

```
constante host = 'localhost'  
puerto constante = 3000
```

7. La función `$e server.listen` también es asíncrona y requiere que pasemos una función de devolución de llamada, que se ejecutará en cuanto el servidor esté en funcionamiento. Podemos registrar algo aquí por ahora:

```
servidor.listen(puerto, host, () => {
    console.log('Servidor escuchando en http://${host}:${port}')
})
```

8. Ejecute el script Node.js de la siguiente manera:

```
$ nodo backend/simpleweb.js
```

9. Observará que recibimos el mensaje de registro " Servidor escuchando en http://localhost:3000" , lo que indica que el servidor se inició correctamente. En este momento, la terminal no nos devuelve el control; el script sigue ejecutándose. Ahora podemos abrir http://localhost:3000 en un navegador:



Hello HTTP world!

Figura 2.3 – ¡Una respuesta en texto simple de nuestro primer servidor web!

Ahora que hemos configurado un servidor web simple, podemos ampliarlo para que sirva un archivo JSON en lugar de simplemente devolver texto sin formato.

Extendiendo el servidor web para servir nuestro archivo JSON

Ahora podemos intentar combinar nuestros conocimientos del módulo `node:fs` con el servidor HTTP para crear un servidor que sirva el archivo `users.json` creado previamente. Comencemos con los pasos:

1. Copie el archivo `backend/simpleweb.js` a un nuevo archivo `backend/webfiles.js`.
2. Al comienzo del archivo, agregue una importación de `readFileSync`:

```
importar { readFileSync } desde 'node:fs'
```

3. Cambie el encabezado `Content-Type` a `application/json`:

```
res.setHeader('Tipo-de-contenido', 'application/json')
```

4. Reemplace la cadena en `res.end()` con la cadena JSON de nuestro archivo. En este caso, no...

Es necesario analizar el JSON, ya que `res.end()` espera una cadena de todos modos:

```
res.end(readFileSync('backend/users.json'))
```

5. Si aún se está ejecutando, detenga el script del servidor anterior mediante Ctrl + C. Necesitamos hacer esto porque no podemos escuchar en el mismo puerto dos veces.

6. Ejecute el servidor y actualice la página para ver el JSON del archivo que se está imprimiendo. Intente cambiar el archivo `users.json` y compruebe cómo se lee de nuevo en la siguiente solicitud (al actualizar el sitio web).

```
$ nodo backend/webfiles.js
```

Aunque son útiles como ejercicio, los archivos no son una base de datos adecuada para su uso en producción. Por ello, más adelante presentaremos MongoDB como base de datos. Ejecutaremos el servidor MongoDB en Docker, así que primero echemos un vistazo a Docker.

Presentamos Docker, una plataforma para contenedores

Docker es una plataforma que nos permite empaquetar, administrar y ejecutar aplicaciones en entornos poco aislados, llamados contenedores. Los contenedores son ligeros, están aislados entre sí e incluyen todas las dependencias necesarias para ejecutar una aplicación. Por lo tanto, podemos usarlos para configurar fácilmente diversos servicios y aplicaciones sin tener que gestionar dependencias ni conflictos entre ellos.

Nota

También existen otras herramientas, como Podman (que incluso tiene una capa de compatibilidad para los comandos Docker CLI) y Rancher Desktop, que también admite comandos Docker CLI.

Podemos usar Docker localmente para configurar y ejecutar servicios en un entorno aislado. Esto evita contaminar nuestro entorno host y garantiza un estado consistente sobre el cual construir. Esta consistencia es especialmente importante al trabajar en equipos de desarrollo grandes, ya que garantiza que todos trabajen con el mismo estado.

Además, Docker facilita la implementación de contenedores en varios servicios en la nube y su ejecución en un flujo de trabajo de integración continua/entrega continua (CI/CD) .

En esta sección, primero presentaremos una descripción general de la plataforma Docker. Luego, aprenderemos a crear un contenedor y a acceder a Docker desde VS Code. Finalmente, comprenderemos cómo funciona Docker y cómo se puede usar para administrar servicios.

La plataforma Docker

La plataforma Docker consta esencialmente de tres partes:

- Cliente Docker: puede ejecutar comandos enviándolos al demonio Docker, que es ejecutándose en la máquina local o en un entorno remoto.
- Host Docker: contiene el demonio Docker, las imágenes y los contenedores.

Registro de Docker: Aloja y almacena imágenes, extensiones y complementos de Docker. De forma predeterminada, se usará el registro público Docker Hub para buscar imágenes.

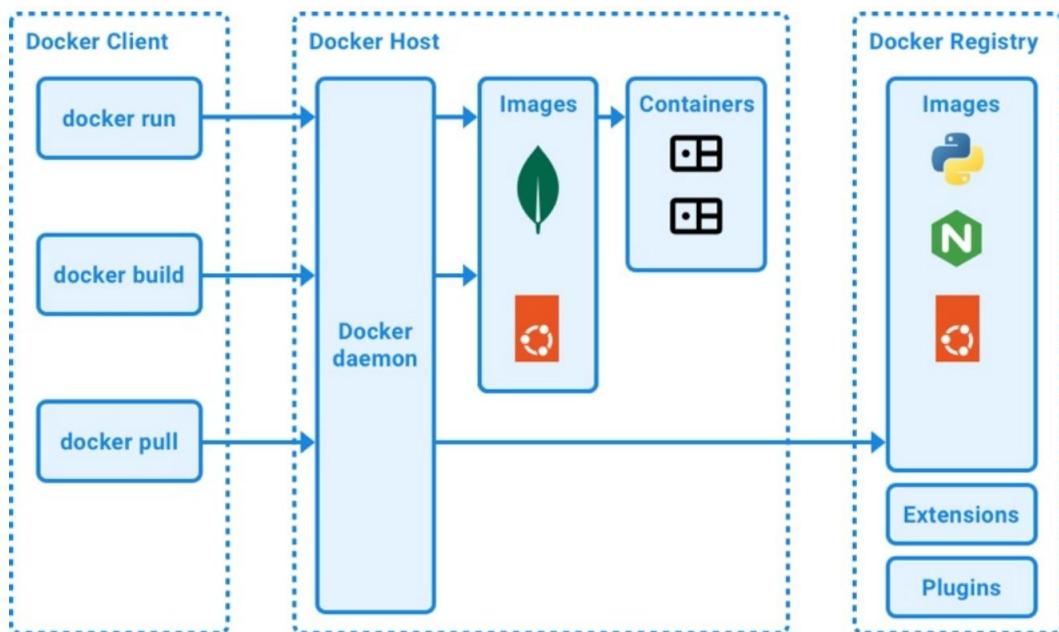


Figura 2.4 – Descripción general de la plataforma Docker

Las imágenes de Docker pueden considerarse plantillas de solo lectura y se utilizan para crear contenedores. Las imágenes pueden basarse en otras imágenes. Por ejemplo, la imagen de Mongo, que contiene un servidor MongoDB, se basa en la imagen de Ubuntu.

Los contenedores Docker son instancias de imágenes. Ejecutan un sistema operativo con un servicio configurado (como un servidor MongoDB en Ubuntu). Además, se pueden configurar, por ejemplo, para redireccionar algunos puertos desde el contenedor al host o para montar un volumen de almacenamiento en el contenedor que almacena datos en el host. Por defecto, un contenedor está aislado del host, por lo que si queremos acceder a los puertos o al almacenamiento desde él en el host, debemos indicarle a Docker que lo permita.

Instalación de Docker

La forma más sencilla de configurar la plataforma Docker para el desarrollo local es usar Docker Desktop. Puede descargarse desde el sitio web oficial de Docker (<https://www.docker.com/products/docker-desktop/>). Sigue las instrucciones para instalarlo e iniciar el motor Docker. Tras la instalación, deberías tener un comando docker disponible en tu terminal. Ejecuta el siguiente comando para verificar su correcto funcionamiento:

```
$ docker -v
```

El comando \$is debe mostrar la versión de Docker, como en el siguiente ejemplo:

```
Versión 24.0.6 de Docker, compilación ed223bc
```

Después de instalar e iniciar Docker, podemos pasar a la creación de un contenedor.

Creando un contenedor

El cliente Docker puede instanciar un contenedor a partir de una imagen mediante el comando docker run. Ahora, creamos un contenedor de Ubuntu y ejecutemos un shell (/bin/bash) en él:

```
$ docker run -i -t ubuntu:24.04 /bin/bash
```

Nota

La cadena \$e :24.04 después del nombre de la imagen se denomina etiqueta y permite fijar imágenes a ciertas versiones. En este libro, usamos etiquetas para extraer versiones específicas de las imágenes, de modo que los pasos sean reproducibles incluso cuando se publiquen nuevas versiones. Por defecto, si no se especifica ninguna etiqueta, Docker intentará usar la etiqueta más reciente.

Se abrirá un nuevo shell. Podemos verificar que este shell se esté ejecutando en el contenedor ejecutando el siguiente comando para ver qué sistema operativo se está ejecutando:

```
$ uname -a
```

Si obtiene un número de versión que termina con -linuxkit, habrá ejecutado exitosamente un comando en el contenedor, ¡porque LinuxKit es un kit de herramientas para crear pequeñas máquinas virtuales Linux!

Ahora puede escribir el siguiente comando para salir del shell y del contenedor:

```
$ salida
```

La siguiente figura muestra el resultado de ejecutar estos comandos:

```
● → ~/D/F/ch2 ↵ main± > docker run -i -t ubuntu /bin/bash
  Unable to find image 'ubuntu:latest' locally
  latest: Pulling from library/ubuntu
  d0a4bfa485d1: Pull complete
  Digest: sha256:2adf22367284330af9f832ffefb717c78239f6251d9d0f58de50b86229ed1427
  Status: Downloaded newer image for ubuntu:latest
  root@d05b654cc753:/# uname -a
  Linux d05b654cc753 5.15.49-linuxkit #1 SMP PREEMPT Tue Sep 13 07:51:32 UTC 2022
    aarch64 aarch64 aarch64 GNU/Linux
  root@d05b654cc753:/# exit
  exit
○ → ~/D/F/ch2 ↵ main± > █
```

Figura 2.5 – Ejecución de nuestro primer contenedor Docker

El comando \$e docker run hace lo siguiente:

- Si nunca antes ha ejecutado un contenedor basado en la imagen de Ubuntu, Docker comenzará extrayendo la imagen del registro de Docker (esto es equivalente a ejecutar docker pull ubuntu).
- Después de descargar la imagen, Docker crea un nuevo contenedor (el equivalente a ejecutar contenedor docker crear).
- \$en, Docker configura un sistema de archivos de lectura y escritura para el contenedor y crea un valor predeterminado. interfaz de red.

Finalmente , Docker inicia el contenedor y ejecuta el comando especificado. En nuestro caso, especificamos el comando /bin/bash. Al pasar las opciones -i (mantiene la entrada estándar abierta) y -t (asigna una pseudo-tty), Docker conecta el shell del contenedor a nuestra terminal en ejecución, lo que nos permite usar el contenedor como si accediéramos directamente a una terminal en nuestro host.

Como podemos ver, Docker es muy útil para crear entornos autónomos donde se ejecutan nuestras aplicaciones y servicios. Más adelante en este libro, aprenderemos a empaquetar nuestras propias aplicaciones en contenedores Docker. Por ahora, solo usaremos Docker para ejecutar servicios sin tener que instalarlos en nuestro sistema host.

Acceder a Docker a través de VS Code

También podemos acceder a Docker mediante la extensión de VS Code que instalamos en el Capítulo 1, Preparación para el desarrollo full-stack. Para ello, haga clic en el ícono de Docker en la barra lateral izquierda de VS Code. Se abrirá la barra lateral de Docker , mostrando una lista de contenedores, imágenes, registros, redes, volúmenes, contextos y recursos relevantes:

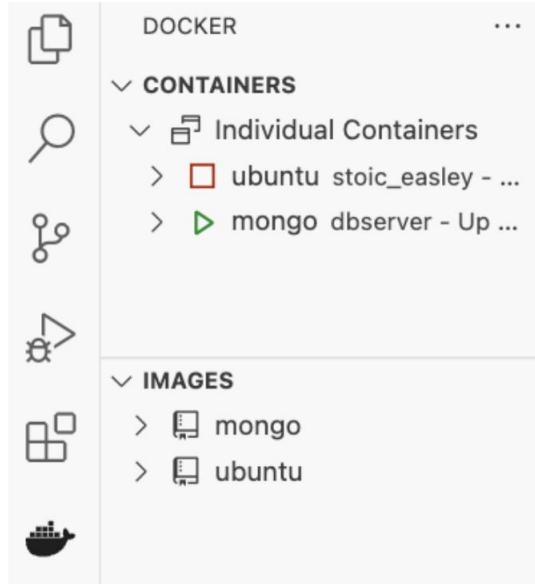


Figura 2.6 – La barra lateral de Docker en VS Code

Aquí puede ver qué contenedores están detenidos y cuáles están en ejecución. Puede hacer clic derecho en un contenedor para iniciararlo, detenerlo, reiniciarlo o eliminarlo. También puede consultar sus registros para depurar lo que ocurre dentro del contenedor. Además, puede conectar un shell al contenedor para acceder a su sistema operativo.

Ahora que conocemos los conceptos básicos de Docker, podemos crear un contenedor para nuestro servidor de base de datos MongoDB.

Presentamos MongoDB, una base de datos de documentos

MongoDB, al momento de escribir este artículo, es la base de datos NoSQL más popular. A diferencia de las bases de datos de lenguaje de consulta estructurado (SQL) (como MySQL o PostgreSQL), NoSQL significa que la base de datos no utiliza SQL para consultarla. En cambio, las bases de datos NoSQL tienen otras formas de consultar la base de datos y, a menudo, presentan una estructura muy diferente de almacenamiento y consulta de datos.

Existen los siguientes tipos principales de bases de datos NoSQL:

- Almacenes de clave-valor (por ejemplo, Valkey/Redis)
- Bases de datos orientadas a columnas (por ejemplo, Amazon Redshift)
- Bases de datos basadas en gráficos (por ejemplo, Neo4j)
- Bases de datos basadas en documentos (por ejemplo, MongoDB)

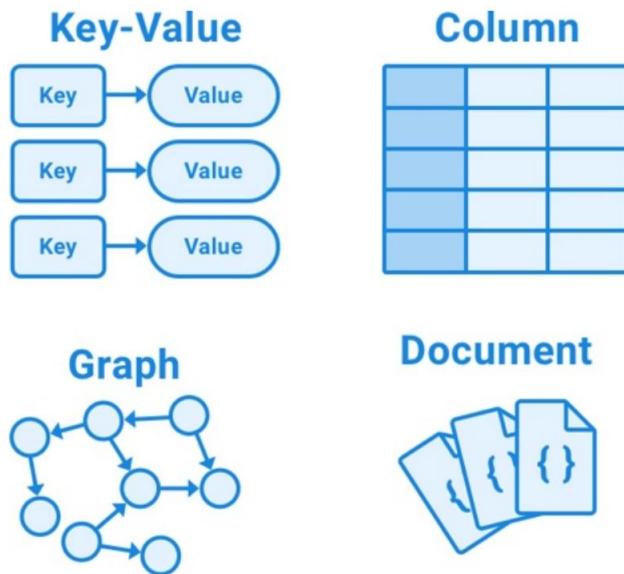


Figura 2.7 – Descripción general de las bases de datos NoSQL

MongoDB es una base de datos basada en documentos, lo que significa que cada entrada se almacena como un documento. En MongoDB, estos documentos son básicamente objetos JSON (internamente, se almacenan como BSON, un formato JSON binario para ahorrar espacio y mejorar el rendimiento, entre otras ventajas).

En cambio, las bases de datos SQL almacenan los datos como filas en tablas. Por lo tanto, MongoDB ofrece mucha más flexibilidad. Se pueden añadir o eliminar campos libremente en los documentos. La desventaja de esta estructura es que no se cuenta con un esquema consistente para los documentos. Sin embargo, esto se puede solucionar mediante bibliotecas como Mongoose, que aprenderemos en el Capítulo 3, "Implementación de un backend con Express, Mongoose ODM y Jest".

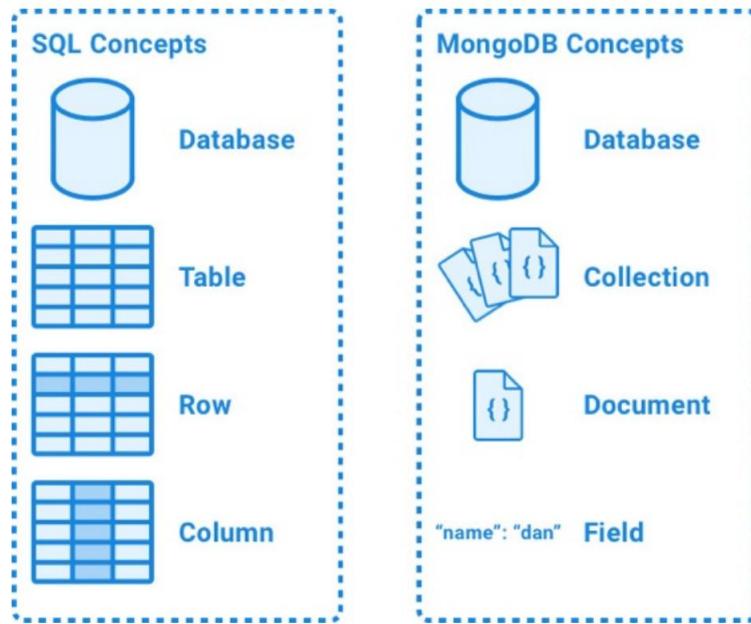


Figura 2.8 – Comparación entre bases de datos MongoDB y SQL

MongoDB también se basa en un motor JavaScript. Desde la versión 3.2, utiliza SpiderMonkey (el motor JavaScript que usa Firefox) en lugar de la versión 8. Sin embargo, esto aún permite ejecutar código JavaScript en MongoDB. Por ejemplo, podemos usar JavaScript en MongoDB Shell para facilitar las tareas administrativas. Sin embargo, debemos tener cuidado con esto, ya que el entorno de MongoDB es muy diferente al de un navegador o Node.js.

En esta sección, aprenderemos primero a configurar un servidor MongoDB con Docker. A continuación, aprenderemos más sobre MongoDB y cómo acceder a él directamente mediante MongoDB Shell para administrar nuestra base de datos y los datos. También aprenderemos a usar VS Code para acceder a MongoDB. Al final de esta sección, comprenderá cómo funcionan las operaciones CRUD en MongoDB.

Nota

CRUD es un acrónimo de crear, leer, actualizar y eliminar, que son las operaciones comunes que suelen proporcionar los servicios de backend.

Configuración de un servidor MongoDB

Antes de empezar a usar MongoDB, necesitamos configurar un servidor. Como ya tenemos Docker instalado, podemos simplificar las cosas ejecutando MongoDB en un contenedor Docker. Esto también nos permite tener instancias de MongoDB independientes y limpias para nuestras aplicaciones mediante la creación de contenedores independientes. Comencemos con los pasos:

1. Asegúrese de que Docker Desktop esté en ejecución y de que Docker esté iniciado. Puede comprobarlo ejecutando el siguiente comando, que lista todos los contenedores en ejecución:

```
$ docker ps
```

Si Docker no se inicia correctamente, obtendrá un error "No se puede conectar al demonio de Docker". En ese caso, asegúrese de que Docker Desktop esté ejecutándose y que Docker Engine no esté en pausa.

Si Docker se inicia correctamente, verá el siguiente resultado:

RECIPIENTE	IMAGEN DE IDENTIFICACIÓN	ESTADO DE CREACIÓN DEL COMANDO	PUERTOS	NOMBRES
------------	--------------------------	--------------------------------	---------	---------

Si ya tiene algunos contenedores en ejecución, aparecerá una lista de contenedores iniciados.

2. Ejecute el siguiente comando de Docker para crear un nuevo contenedor con un servidor MongoDB:

```
$ docker run -d --name dbserver -p 27017:27017 --restart a menos que se detenga mongo:6.0.4
```

El comando \$e docker run crea y ejecuta un nuevo contenedor. Los argumentos \$e son los siguientes:

-d: ejecuta el contenedor en segundo plano (modo demonio).

--name: Indica el nombre del contenedor. En nuestro caso, lo llamamos dbserver.

-p: Asigna un puerto del contenedor al host. En nuestro caso, asignamos el puerto predeterminado del servidor MongoDB, 27017, en el contenedor al mismo puerto en nuestro host. \$is nos permite acceder al servidor MongoDB que se ejecuta dentro de nuestro contenedor desde fuera. Si ya tiene un servidor MongoDB ejecutándose en ese puerto, puede cambiar el primer número a otro puerto, pero asegúrese de ajustar también el número de puerto de 27017 al puerto especificado en las siguientes guías.

--restart unless-stopped: se asegura de iniciar (y reiniciar) automáticamente el contenedor a menos que lo detengamos manualmente. \$is asegura que cada vez que iniciamos Docker, nuestro servidor MongoDB ya estará ejecutándose.

mongo: \$is es el nombre de la imagen. \$e La imagen mongo contiene un servidor MongoDB.

3. Instale MongoDB Shell en su sistema host (no dentro del contenedor) siguiendo las instrucciones del sitio web de MongoDB (<https://www.mongodb.com/docs/mongodb-shell/install/>).

4. En su sistema host, ejecute el siguiente comando para conectarse al servidor MongoDB mediante MongoDB Shell (mongosh).

Después del nombre de host y el puerto, especifique el nombre de la base de datos.

Vamos a llamar a nuestra base de datos ch2:

```
$ mongosh mongodb://localhost:27017/ch2
```

Verá la salida del servidor de bases de datos y, al final, obtenemos un shell ejecutándose en la base de datos seleccionada, como se puede ver en el prompt ch2>. Aquí, podemos ingresar comandos para su ejecución en nuestra base de datos.

Curiosamente, MongoDB, al igual que Node.js, también expone un motor JavaScript, pero con un entorno diferente. Así, podemos ejecutar código JavaScript, como el siguiente:

```
ch2> console.log("prueba")
```

La siguiente figura muestra el código JavaScript que se ejecuta en MongoDB Shell:

```
○ ~/D/F/ch2 ↵ main: > mongosh mongodb://localhost:27017/ch2
Current Mongosh Log ID: 6564926d8342fc83e3545366
Connecting to:      mongod://localhost:27017/ch2?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.1.0
(node:51213) [DEP0040] DeprecationWarning: The punycode module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
Using MongoDB:     6.0.4
Using Mongosh:     2.1.0

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-11-27T12:58:10.497+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2023-11-27T12:58:11.204+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-11-27T12:58:11.204+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
2023-11-27T12:58:11.204+00:00: vm.max_map_count is too low
-----

ch2> console.log("test")
test
ch2>
```

Figura 2.9 – Conexión a nuestro servidor de base de datos MongoDB ejecutándose en un contenedor Docker

Ahora que tenemos un shell conectado a nuestro servidor de base de datos MongoDB, podemos comenzar a practicar la ejecución de comandos directamente en la base de datos.

Ejecutar comandos directamente en la base de datos

Antes de comenzar a crear un servicio de backend que interactúe con MongoDB, dediquemos un tiempo a familiarizarnos con MongoDB a través de MongoDB Shell. \$e MongoDB Shell es muy importante para depurar y realizar tareas de mantenimiento en la base de datos, por lo que es una buena idea conocerlo bien.

Crear una colección e insertar y listar documentos

Las colecciones en MongoDB son el equivalente a las tablas en bases de datos relacionales. Almacenan documentos, que son como objetos JSON. Para facilitar su comprensión, una colección puede considerarse como un array JSON muy grande que contiene objetos JSON. A diferencia de los arrays simples, las colecciones admiten la creación de índices, lo que agiliza la búsqueda de ciertos campos en los documentos. En MongoDB, una colección se crea automáticamente al intentar insertar un documento en ella o crear un índice para ella.

Utilicemos MongoDB Shell para insertar un documento en nuestra base de datos en la colección de usuarios:

1. Para insertar un nuevo documento de usuario en la colección de usuarios, ejecute el siguiente comando en el Shell de MongoDB:

```
> db.users.insertOne({ nombre de usuario: 'dan', nombre completo: 'Daniel Bugl', edad: 26 })
```

Los comandos que acceden a la base de datos tienen como prefijo db, luego sigue el nombre de la colección y finalmente viene la operación, todos separados por puntos.

Nota

Si bien insertOne() nos permite insertar un solo documento en la colección, también hay un método insertMany(), donde podemos pasar un array de documentos para agregar a la colección.

2. Ahora podemos enumerar todos los documentos de la colección de usuarios ejecutando el siguiente comando:

```
> db.usuarios.find()
```

Al hacerlo, se devolverá una matriz con nuestro documento insertado previamente:

```
[  
  {  
    _id: Id. del objeto("6405f062b0d06adeaeefc3bc"),  
    nombre de usuario: 'dan',  
    Nombre completo: 'Daniel Bugl',  
    edad: 26  
  }  
]
```

Como podemos ver, MongoDB creó automáticamente un ID único (ObjectId) para nuestro documento. Este ID consta de 12 bytes en formato hexadecimal (por lo que cada byte se muestra como dos caracteres). Los bytes se definen de la siguiente manera:

- Los primeros 4 bytes son una marca de tiempo, que representa la creación del ID medida en segundos desde la época de Unix.
- Los siguientes 5 bytes son un valor aleatorio exclusivo de la máquina y del proceso de base de datos que se está ejecutando actualmente.

- Los últimos 3 bytes son un contador incremental inicializado aleatoriamente

Nota

La forma en que se generan los identificadores ObjectId en MongoDB garantiza que los ID sean únicos, evitando colisiones de ID incluso cuando se generan dos ID al mismo tiempo desde diferentes instancias, sin requerir una forma de comunicación entre las instancias, lo que ralentizaría la creación de documentos al escalar la base de datos.

Consulta y ordenación de documentos

Ahora que hemos insertado algunos documentos, podemos consultarlos accediendo a diferentes campos del objeto. También podemos ordenar la lista de documentos devueltos por MongoDB. Siga estos pasos:

1. Antes de comenzar a realizar consultas, insertemos dos documentos más en nuestra colección de usuarios:

```
> db.usuarios.insertarMuchos([
    { nombre de usuario: 'jane', nombre completo: 'Jane Doe', edad: 32 },
    { nombre de usuario: 'john', nombre completo: 'John Doe', edad: 30 }
])
```

2. Ahora podemos empezar a consultar un nombre de usuario específico usando findOne y pasando un objeto con el campo "nombre de usuario". Al usar findOne, MongoDB devolverá el primer objeto coincidente:

```
> db.users.findOne({ nombre de usuario: 'jane' })
```

3. También podemos consultar nombres completos o cualquier otro campo en los documentos de la colección.

Al usar find, MongoDB devolverá una matriz de todas las coincidencias:

```
> db.users.find({ nombrecompleto: 'Daniel Bugl' })
```

4. Una cosa importante a tener en cuenta es que al consultar un ObjectId, debemos envolver la cadena de ID con un constructor ObjectId(), de la siguiente manera:

```
> db.users.findOne({ _id: ObjectId('6405f062b0d06adeaeefc3bc') })
```

Asegúrese de cambiar la cadena pasada al constructor ObjectId() a un ObjectId válido regresó de los comandos anteriores.

5. MongoDB también proporciona ciertos operadores de consulta, precedidos por \$. Por ejemplo, podemos encontrar a todos los mayores de 30 años en nuestra colección usando el operador \$gt, como se muestra a continuación:

```
> db.users.find({ edad: { $gt: 30 } })
```

Notarás que John Doe no se devuelve, porque su edad es exactamente 30. Si queremos hacer coincidir edades mayores o iguales a 30, necesitamos usar el operador \$gte.

6. Si queremos ordenar nuestros resultados, podemos usar el método `.sort()` después de `.find()`. Por ejemplo, podemos devolver todos los elementos de la colección de usuarios, ordenados por edad en orden ascendente (1 para ascendente, -1 para descendente):

```
> db.users.find().sort({ edad: 1 })
```

Actualización de documentos

Para actualizar un documento en MongoDB, combinamos los argumentos de las operaciones de consulta e inserción en una sola. Podemos usar los mismos criterios para filtrar documentos que para `find()`. Para actualizar un solo campo del documento, usamos el operador `$set`:

1. Podemos actualizar el campo de edad para el usuario con el nombre de usuario `dan` de la siguiente manera:

```
> db.users.updateOne({ nombre de usuario: 'dan' }, { $set: { edad: 27 } })
```

Nota

Al igual que `findOne`, `updateOne` solo actualiza el primer documento coincidente. Si queremos actualizar todos los documentos coincidentes, podemos usar `updateMany`.

MongoDB devolverá un objeto con información sobre cuántos documentos coincidieron (`matchedCount`), cuántos fueron modificados (`modifiedCount`) y cuántos fueron insertados nuevamente (`upsertedCount`).

2. El método `$e` `updateOne` acepta un tercer argumento, que es un objeto de opciones. Una opción útil es la opción `upsert`, que, si se establece en verdadero, insertará un documento si aún no existe y lo actualizará si ya existe. Primero, intentemos actualizar un usuario inexistente con `upsert: false`.

```
> db.users.updateOne({ nombre de usuario: 'nuevo' }, { $set: { nombre_completo: 'Nuevo usuario' } })
```

3. Ahora configuraremos `upsert` como verdadero, lo que inserta al usuario:

```
> db.users.updateOne({ nombre de usuario: 'nuevo' }, { $set: { nombre_completo: 'Nuevo usuario' } }, { upsert: verdadero })
```

Nota

Si desea eliminar un campo de un documento, utilice el operador `$unset`. Si desea reemplazar todo el documento con uno nuevo, puede usar el método `replaceOne` y pasarle un nuevo documento como segundo argumento.

Eliminar documentos

Para eliminar documentos de la base de datos, MongoDB proporciona `deleteOne` y `deleteMany` métodos, que tienen una API similar a los métodos `updateOne` y `updateMany`. El primer argumento se utiliza, nuevamente, para hacer coincidir documentos.

Supongamos que el usuario con el nombre de usuario "new" desea eliminar su cuenta. Para ello, debemos eliminarlo de la colección de usuarios. Podemos hacerlo de la siguiente manera:

```
> db.users.deleteOne({ nombre de usuario: 'nuevo' })
```

¡Eso es todo! Como puedes ver, es muy sencillo realizar operaciones CRUD en MongoDB si ya sabes trabajar con objetos JSON y JavaScript, lo que la convierte en la base de datos perfecta para un backend de Node.js.

Ahora que hemos aprendido cómo acceder a MongoDB usando MongoDB Shell, aprendamos cómo acceder a él desde VS Code.

Accediendo a la base de datos a través de VS Code

Hasta ahora, hemos usado la Terminal para acceder a la base de datos. Si recuerdan, en el Capítulo 1, Preparación para el Desarrollo Full-stack, instalamos una extensión de MongoDB para VS Code. Ahora podemos usar esta extensión para acceder a nuestra base de datos de forma más visual:

1. Haga clic en el ícono de MongoDB en la barra lateral izquierda (debería ser un ícono de hoja) y haga clic en el botón Agregar conexión :

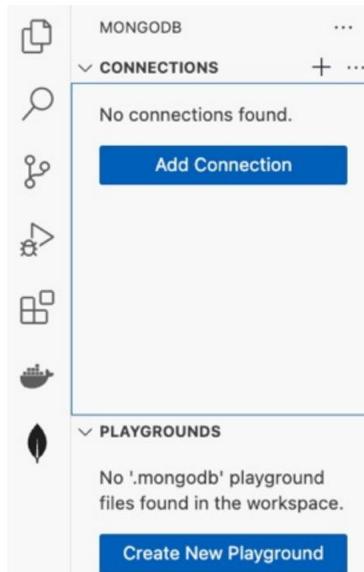


Figura 2.10 – La barra lateral de MongoDB en VS Code

2. Se abrirá una nueva pestaña de MongoDB . En esta pestaña, haga clic en Conectar en el menú Conectar con.

Cuadro de cadena de conexión :

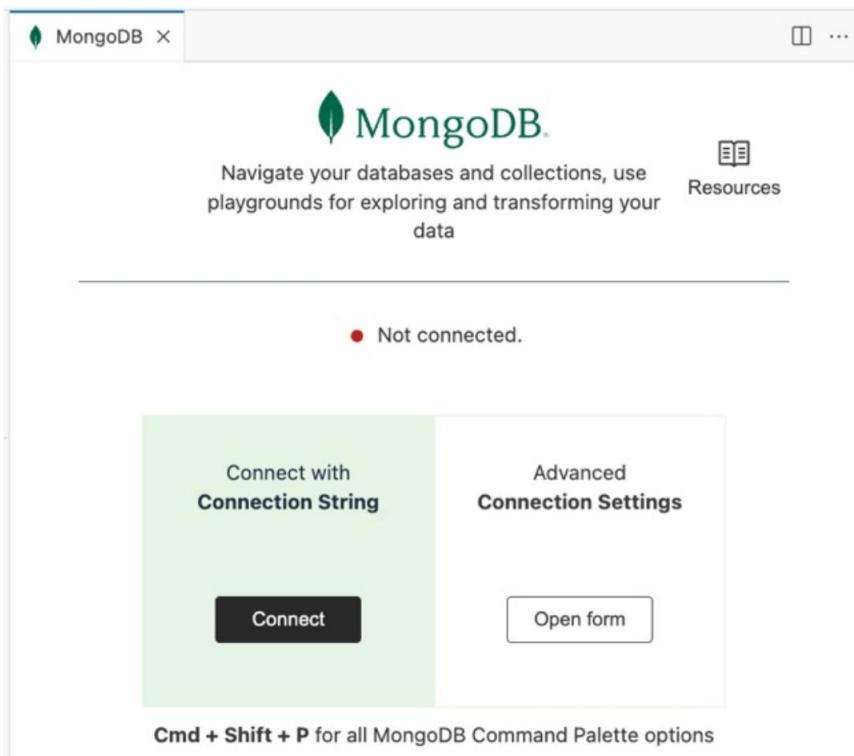


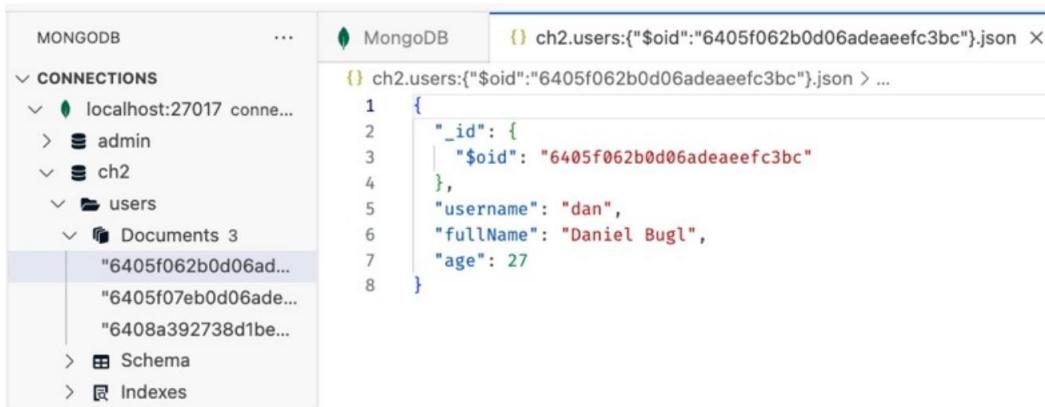
Figura 2.11 – Agregar una nueva conexión MongoDB en VS Code

3. Se abrirá una ventana emergente en la parte superior. En ella, introduzca la siguiente cadena de conexión para conectarse.

a su base de datos local:

```
mongodb://localhost:27017/
```

4. Pulse Intro para confirmar. Se mostrará una nueva conexión en la barra lateral de MongoDB. Puede explorar el árbol para ver bases de datos, colecciones y documentos. Por ejemplo, haga clic en el primer documento para verlo:



The screenshot shows the MongoDB extension interface in VS Code. On the left, there's a tree view of database connections, showing 'localhost:27017' with collections 'admin', 'ch2', 'users', 'Documents 3', and 'Indexes'. A specific document '_id: "6405f062b0d06ad...' is selected and highlighted with a light blue background. On the right, a code editor window titled 'MongoDB' displays the JSON representation of this document:

```

1  {
2   "_id": {
3     "$oid": "6405f062b0d06ad..."
4   },
5   "username": "dan",
6   "fullName": "Daniel Bugl",
7   "age": 27
8 }

```

Figura 2.12 – Visualización de un documento en la extensión MongoDB en VS Code

Consejo

También puede editar directamente un documento editando un campo en VS Code y guardando el archivo. El documento actualizado se guardará automáticamente en la base de datos.

La extensión \$e de MongoDB es muy útil para depurar nuestra base de datos, ya que nos permite detectar problemas visualmente y editar documentos rápidamente. Además, podemos hacer clic derecho en "Documentos" y "Buscar documentos..." para abrir una nueva ventana donde podemos ejecutar consultas MongoDB, tal como lo hacíamos en la Terminal. Las consultas \$e se pueden ejecutar en la base de datos haciendo clic en el botón "Reproducir" en la esquina superior derecha. Es posible que deba confirmar el diálogo con "Sí" y los resultados se mostrarán en un nuevo panel, como se muestra en la siguiente captura de pantalla:

```

// MongoDB Playground Untitled-1
// Use Ctrl+Space inside a snippet or ...
// The current database to use.
use('ch2');

// Search for documents in the current database
db.getCollection('users')
  .find(
    {
      /*
       * Filter
       * fieldA: value or expression
      */
    },
    {
      /*
       * Projection
       * _id: 0, // exclude _id
       * fieldA: 1 // include field
      */
    }
  )
  .sort({
    /*
     * fieldA: 1 // ascending
     * fieldB: -1 // descending
    */
  });

```

```

[{"_id": {"$oid": "6405f062b0d06adeaeefc3b"}, "username": "dan", "fullName": "Daniel Bugl", "age": 27}, {"_id": {"$oid": "6405f07eb0d06adeaeefc3b"}, "username": "jane", "fullName": "Jane Doe", "age": 32}, {"_id": {"$oid": "6408a392738d1be75915796"}, "username": "john", "fullName": "John Doe", "age": 30}]

```

Figura 2.13 – Consulta de MongoDB en VS Code

Ahora que hemos aprendido los conceptos básicos del uso y la depuración de bases de datos MongoDB, podemos comenzar a integrar nuestra base de datos en un servicio backend Node.js, en lugar de simplemente almacenar y leer información de archivos.

Acceder a la base de datos MongoDB a través de Node.js

Ahora vamos a crear un nuevo servidor web que, en lugar de devolver usuarios desde un archivo JSON, devuelva la lista de usuarios de nuestra colección de usuarios creada previamente:

1. En la carpeta ch2, abra una terminal. Instale el paquete mongodb, que contiene el paquete social.

Controlador MongoDB para Node.js:

```
$ npm install mongodb@6.3.0
```

2. Cree un nuevo archivo backend/mongodbweb.js y ábralo. Importe lo siguiente:

```
importar { createServer } desde 'node:http'
importar { MongoClient } desde 'mongodb'
```

3. Defina la URL de conexión y el nombre de la base de datos y luego cree un nuevo cliente MongoDB:

```
constante url = 'mongodb://localhost:27017/'  
constante nombre_base = 'ch2'  
const cliente = nuevo MongoClient(url)
```

4. Conéctese a la base de datos y registre un mensaje después de que nos hayamos conectado exitosamente o cuando haya

Hay un error con la conexión:

```
intentar {  
    esperar cliente.conectar()  
    console.log(`Conectado exitosamente a la base de datos!`) } catch (err)  
{ console.error('Error al  
    conectarse a la base de datos:', err)  
}
```

5. A continuación, crea un servidor HTTP, como hicimos antes:

```
const servidor = crearServidor(async (req, res) => {
```

6. \$en, seleccione la base de datos del cliente y la colección de usuarios de la base de datos:

```
const db = cliente.db(dbName) const usuarios  
= db.collection('usuarios')
```

7. Ahora, ejecute el método find() en la colección de usuarios. En el controlador de MongoDB Node.js, también necesitamos llamar al método toArray() para resolver el iterador en un array:

```
const usersList = await usuarios.find().toArray()
```

8. Finalmente, configure el código de estado y el encabezado de respuesta, y devuelva la lista de usuarios:

```
res.código de estado = 200  
res.setHeader('Tipo-de-contenido', 'application/json') res.end(JSON.stringify(usersList)) }
```

9. Ahora que hemos definido nuestro servidor, copie el código anterior para escuchar al host local.

en el puerto 3000:

```
constante host = 'localhost'  
const port = 3000  
server.listen(port, host, () => { console.log(`Servidor  
    escuchando en http://${host}:${port}`) })
```

10. Ejecute el servidor ejecutando el script:

```
$ nodo backend/mongodbweb.js
```

11. Abra <http://localhost:3000> en su navegador y debería ver la lista de usuarios de nuestra base de datos:



Figura 2.14 – ¡Nuestro primer servicio Node.js que recupera datos de una base de datos MongoDB!

Como hemos visto, también podemos usar métodos similares a los que usamos en MongoDB Shell en Node.js. Sin embargo, las API del módulo node:http y del paquete mongodb son de muy bajo nivel, lo que requiere mucho código para crear una API HTTP y comunicarse con la base de datos.

En el próximo capítulo, aprenderemos sobre las bibliotecas que abstraen estos procesos para permitir una creación más sencilla de API HTTP y el manejo de la base de datos. Estas bibliotecas son Express y Mongoose. Express es un framework web que nos permite definir fácilmente rutas de API y gestionar solicitudes. Mongoose nos permite crear esquemas para documentos en nuestra base de datos para facilitar la creación, lectura, actualización y eliminación de objetos.

Resumen

En este capítulo, aprendimos a usar Node.js para desarrollar scripts que se ejecutan en un servidor. También aprendimos a crear contenedores con Docker y cómo funciona MongoDB y cómo se puede interactuar con él. Al final de este capítulo, incluso creamos con éxito nuestro primer servicio backend simple usando Node.js y MongoDB!

En el próximo capítulo, Capítulo 3, Implementación de un backend usando Express, Mongoose ODM y Jest, aprenderemos cómo combinar lo que aprendimos en este capítulo para extender nuestro simple servicio de backend a un backend listo para producción para una aplicación de blog.

Machine Translated by Google

Parte 2:

Construyendo e implementando nuestro Primera aplicación full-stack con una API REST

En esta parte, construiremos e implementaremos nuestra primera aplicación full-stack con una API REST . Comenzaremos implementando un servicio backend con Express y Mongoose ODM. Luego, crearemos pruebas unitarias con Jest. Posteriormente, crearemos un frontend con React y lo integraremos con nuestro servicio backend mediante TanStack Query. Finalmente, implementaremos la aplicación con Docker. y aprenda cómo configurar una canalización CI/CD.

Esta parte incluye los siguientes capítulos:

- Capítulo 3, Implementación de un backend usando Express, Mongoose ODM y Jest
- Capítulo 4, Integración de un frontend usando React y TanStack Query
- Capítulo 5, Implementación de la aplicación con Docker y CI/CD

3

Implementando un backend usando Express, Mongoose ODM y Jest

Tras aprender los fundamentos de Node.js y MongoDB, los pondremos en práctica creando nuestro primer servicio backend con Express para proporcionar una API REST, el modelado de datos de objetos (ODM) de Mongoose para interactuar con MongoDB y Jest para probar nuestro código. Primero, aprenderemos a estructurar un proyecto backend utilizando un patrón arquitectónico. A continuación, crearemos esquemas de base de datos con Mongoose. A continuación, crearemos funciones de servicio para interactuar con los esquemas de base de datos y escribiremos pruebas para ellas con Jest. A continuación, aprenderemos qué es REST y cuándo es útil. Finalmente, proporcionaremos una API REST y la serviremos con Express. Al final de este capítulo, tendremos un servicio backend funcional que será utilizado por un frontend desarrollado en el siguiente capítulo.

En este capítulo cubriremos los siguientes temas principales:

- Diseño de un servicio backend
- Creación de esquemas de bases de datos utilizando Mongoose
- Desarrollar y probar funciones de servicio
- Proporcionar una API REST mediante Express

Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones que aparecen en esos capítulos son las que se utilizan en este libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que algunos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas en los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub en <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/capítulo3>.

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

El video de \$e CiA para el capítulo se puede encontrar en: <https://youtu.be/fFHVVVn03rc>.

Diseño de un servicio backend

Para diseñar nuestro servicio backend, vamos a utilizar una variación de un patrón arquitectónico existente llamado patrón modelo-vista-controlador (MVC). El patrón MVC consta de las siguientes partes:

- Modelo: maneja datos y lógica básica de datos.
- Controlador: controla cómo se procesan y se muestran los datos
- Ver: muestra el estado actual

En las aplicaciones tradicionales de pila completa, el backend renderizaba y mostraba el frontend por completo, y una interacción solía requerir una actualización completa de la página. La arquitectura MVC se diseñó principalmente para este tipo de aplicaciones. Sin embargo, en las aplicaciones modernas, el frontend suele ser interactivo y se renderiza en el backend únicamente mediante renderizado del lado del servidor. Por lo tanto, en las aplicaciones modernas, a menudo distinguimos entre los servicios del backend y el backend para el frontend (que gestiona la generación de sitios web estáticos y el renderizado del lado del servidor).

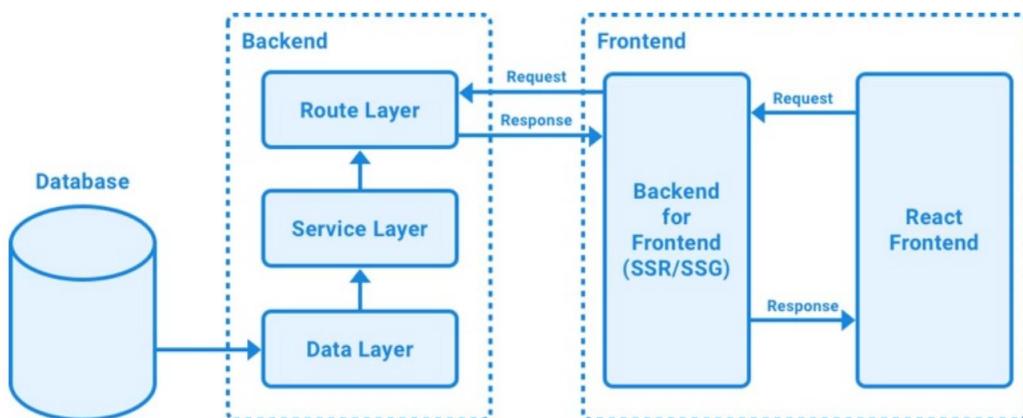


Figura 3.1 – Una arquitectura moderna de pila completa, con un único servicio de backend y un frontend con renderizado del lado del servidor (SSR) y generación de sitios estáticos (SSG)

Para las aplicaciones modernas, la idea es que el servicio backend solo se encargue del procesamiento y la entrega de solicitudes y datos, y ya no renderice la interfaz de usuario. En su lugar, tenemos una aplicación independiente que gestiona específicamente la renderización de las interfaces de usuario en el frontend y el servidor. Para adaptarnos a este cambio, ajustamos el patrón arquitectónico MVC a un patrón de datos-servicio-ruta para el servicio backend, como se indica a continuación:

- Capa de ruta: define las rutas a las que los consumidores pueden acceder y maneja la entrada del usuario mediante el procesamiento Los parámetros y el cuerpo de la solicitud y luego llamar a las funciones del servicio
- Capa de servicio: proporciona funciones de servicio, como funciones de creación, lectura, actualización y eliminación (CRUD) . que acceden a la base de datos a través de la capa de datos
- Capa de datos: solo se ocupa del acceso a la base de datos y realiza una validación básica para garantizar que La base de datos es consistente

Esta separación de preocupaciones funciona mejor para servicios que solo exponen rutas y no se encargan de la representación de interfaces de usuario. Cada capa de este patrón solo se encarga de un paso del procesamiento de la solicitud.

Después de aprender sobre el diseño de servicios backend, comenzemos a crear una estructura de carpetas que refleje lo que hemos aprendido.

Creando la estructura de carpetas para nuestro servicio backend

Ahora crearemos una estructura de carpetas para nuestro servicio backend basada en este patrón. Siga estos pasos:

1. Primero, copie la carpeta ch2 a una nueva carpeta ch3 para crear una nueva carpeta para nuestro servicio backend, como sigue:

```
$ cp -R ch2 ch3
```

2. Abra la nueva carpeta ch3 en VS Code.

3. Edite el archivo .eslintrc.json y reemplace el entorno del navegador con el nodo y el entorno es6. como sigue:

```
"env": {  
  "nodo": verdadero,  
  "es6": verdadero  
},
```

4. Además, elimine los complementos de React y jsx-a11y del archivo .eslintrc.json. También podemos eliminar las configuraciones y anulaciones relacionadas con React eliminando las líneas resaltadas:

```
"se extiende": [  
  "eslint:recomendado",  
  "plugin:react/recomendado",
```

```

    "plugin:react/jsx-runtime", "plugin:jsx-a11y/
recommended", "más bonito"

],
"configuración":
{
  "reaccionar":
  {
    "versión": "detectar"
  }
}

}], "anula": [
{
  "archivos": ["*.js", "*jsx"]
}
]
]
```

5. Elimine los archivos index.html y vite.config.js.

6. Ahora también podemos eliminar el archivo vite.config.js del archivo .eslintignore:

```

dist/
vite.config.js

```

7. Elimine las carpetas pública, backend y src.

8. Abra la carpeta ch3 en VS Code, abra una terminal y ejecute los siguientes comandos para eliminar vite y react:

```

$ npm uninstall --save react react-dom $ npm uninstall --save-dev
vite @types/react \ @types/react-dom @vitejs/plugin-react \ eslint-plugin-jsx-a11y
eslint-plugin-react

```

9. Edite el archivo package.json y elimine de él los scripts dev, build y preview:

```

"scripts": { "dev":
  "vite",
  "construir": "invitar a construir",
  "lint": "src de eslint", "preview":
  "vista previa de vite", "prepare": "instalación
de husky"
},

```

10. Ahora, cree una nueva carpeta src/ y, dentro de ella, cree las carpetas src/db/ (para la capa de datos), src/services/ (para la capa de servicios) y src/routes/ (para la capa de rutas).

Nuestra primera aplicación será un blog. Para ello, necesitaremos la API para realizar lo siguiente:

- Obtener una lista de publicaciones
- Obtener una sola publicación
- Crear una nueva publicación
- Actualizar una publicación existente
- Eliminar una publicación existente

Para proporcionar estas funciones, primero necesitamos crear un esquema de base de datos para definir la apariencia de un objeto de entrada de blog en nuestra base de datos. Necesitamos funciones de servicio para gestionar la funcionalidad CRUD de las entradas de blog. Finalmente, definiremos nuestra API REST para consultar, crear, actualizar y eliminar entradas de blog.

Creación de esquemas de bases de datos con Mongoose

Antes de que podamos comenzar a definir los esquemas de la base de datos, primero debemos configurar Mongoose. Mongoose es una biblioteca que simplifica el modelado de objetos de MongoDB al reducir el código repetitivo necesario para interactuar con MongoDB. También incluye lógica de negocio común, como la configuración de `createdAt` y `updatedAt` marcas de tiempo automáticamente y validación y conversión de tipos para mantener el estado de la base de datos consistente.

Siga estos pasos para configurar la biblioteca mongoose:

1. Primero, instale la biblioteca mongoose:

```
$ npm install mongoose@8.0.2
```

2. Cree un nuevo archivo `src/db/init.js` e importe mongoose allí:

```
importar mangosta de 'mangosta'
```

3. Defina y exporte una función que inicializará la conexión a la base de datos:

```
función de exportación initDatabase() {
```

4. Primero, definimos `DATABASE_URL` para apuntar a nuestra instancia local de MongoDB que se ejecuta a través de Docker y especificamos `blog` como el nombre de la base de datos:

```
constante URL_BASE_DE_DATOS = 'mongodb://localhost:27017/blog'
```

La cadena de conexión \$e es similar a la que usamos en el capítulo anterior al acceder directamente a la base de datos a través de Node.js.

5. \$en, agrega un oyente al evento abierto en la conexión Mongoose para que podamos mostrar un registro mensaje una vez que estemos conectados a la base de datos:

```
mangosta.conexión.on('abierto', () => {
    console.info('conectado exitosamente a la base de datos:', DATABASE_URL)
})
```

6. Ahora, use la función mongoose.connect() para conectarse a nuestra base de datos MongoDB y devuelve el objeto de conexión:

```
constante conexión = mongoose.connect(URL_BASE_DE_DATOS)
conexión de retorno
}
```

7. Cree un nuevo archivo src/example.js e importe y ejecute allí la función initDatabase:

```
importar { initDatabase } desde './db/init.js'
initDatabase()
```

8. Ejecute el archivo src/example.js usando Node.js para ver que Mongoose se conecta correctamente a nuestra base de datos:

```
$ nodo src/ejemplo.js
```

Como siempre, puedes detener el servidor presionando Ctrl + C en la Terminal.

Podemos ver nuestro mensaje de registro impreso en la terminal, lo que nos permite saber que Mongoose se conectó correctamente a nuestra base de datos. Si se produce un error, por ejemplo, porque Docker (o el contenedor) no se está ejecutando, se bloqueará un rato y luego mostrará un error indicando que la conexión se ha rechazado (ECONNREFUSED). En ese caso, asegúrese de que el contenedor Docker MongoDB se esté ejecutando correctamente y se pueda conectar.

Definición de un modelo para publicaciones de blog

Después de inicializar la base de datos, lo primero que debemos hacer es definir la estructura de datos para las publicaciones del blog. Las entradas de blog en nuestro sistema deben tener un título, un autor, contenido y algunas etiquetas asociadas . Sigue estos pasos para definir la estructura de datos de las entradas de blog:

1. Cree una nueva carpeta src/db/models/.
2. Dentro de esa carpeta, crea un nuevo archivo src/db/models/post.js, importa el archivo mongoose y las clases de esquema:

```
importar mangosta, { Esquema } desde 'mangosta'
```

3. Definir un nuevo esquema para las publicaciones:

```
const postSchema = nuevo Esquema({
```

4. Ahora especifica todas las propiedades de una entrada de blog y sus tipos correspondientes. Tenemos un título, un autor y un contenido obligatorios, que son cadenas:

```
título: { tipo: Cadena, requerido: verdadero },  
autor: String,  
Contenido: Cadena,
```

5. Por último, tenemos las etiquetas, que son una matriz de cadenas:

```
etiquetas: [Cadena],  
})
```

6. Ahora que hemos definido el esquema, podemos crear un modelo Mongoose a partir de él utilizando el Función mongoose.model():

```
exportar const Post = mongoose.model('post', postSchema)
```

Nota

El primer argumento de mongoose.model() especifica el nombre de la colección. En nuestro caso, la colección se llamará posts porque especificamos "post" como nombre. En los modelos de Mongoose, el nombre del documento debe especificarse en singular.

Ahora que hemos definido la estructura y el modelo de datos para las publicaciones del blog, podemos comenzar a usarlos para crear y consultar publicaciones.

Usando el modelo de publicación de blog

Después de crear nuestro modelo, ¡intentemos usarlo! Por ahora, simplemente accederemos a él en src/example.js le porque aún no hemos definido ninguna función de servicio o ruta:

1. Importe el modelo Post en el archivo src/example.js:

```
importar { initDatabase } desde './db/init.js'  
importar { Post } desde './db/models/post.js'
```

2. La función \$e initDatabase() que definimos anteriormente es una función asíncrona, por lo que debemos esperarla; de lo contrario, estaríamos intentando acceder a la base de datos antes de conectarnos a ella:

```
esperar initDatabase()
```

3. Crea una nueva entrada de blog llamando a new Post() y definiendo algunos datos de ejemplo:

```
const post = nueva publicación({  
    Título: '¡Hola Mongoose!', Autor: 'Daniel  
Bugl', Contenido: 'Esta publicación  
se almacena en una base de datos MongoDB usando Mongoose.', Etiquetas: ['mongoose',  
'mongodb'], })
```

4. Llama a .save() en la publicación del blog para guardarla en la base de datos:

```
esperar post.save()
```

5. Ahora podemos usar la función .find() para enumerar todas las publicaciones y registrar el resultado:

```
const posts = await Post.find() console.log(posts)
```

6. Ejecute el script de ejemplo para ver nuestra publicación insertada y listada:

```
$ nodo src/ejemplo.js
```

Obtendrá el siguiente resultado después de ejecutar el script anterior:

```
○ → ~/D/F/ch3 ↵ main± > node src/test.js  
successfully connected to database: mongodb://localhost:27017/ch3  
[  
  {  
    _id: new ObjectId("64281a742bd96957bf8bde51"),  
    title: 'Hello Mongoose!',  
    author: 'Daniel Bugl',  
    contents: 'This post is stored in a MongoDB database using Mongoose.',  
    tags: [ 'mongoose', 'mongodb' ],  
    __v: 0  
  }  
]
```

Figura 3.2 – ¡Nuestro primer documento insertado a través de Mongoose!

Como puede ver, usar Mongoose es muy similar a usar MongoDB directamente. Sin embargo, ofrece algunas envolturas alrededor de los modelos para mayor comodidad, lo que facilita el manejo de documentos.

Definir fechas de creación y última actualización en la entrada del blog

Quizás hayas notado que no hemos añadido ninguna fecha a nuestra entrada de blog. Por lo tanto, no sabemos cuándo se creó una entrada ni cuándo se actualizó por última vez. Mongoose simplifica la implementación de esta funcionalidad; probémosla ahora:

1. Edite el archivo `src/db/models/post.js` y agregue un segundo argumento al constructor de `Schema()`. `$e`, segundo argumento, opciones de especies para el esquema. Aquí, establecemos la configuración `timestamps: true`.

```
const postSchema = nuevo Esquema(  
  {  
    título: Cadena, autor:  
    Cadena, contenido:  
    Cadena, etiquetas:  
    [Cadena], }, { marcas  
  
      de tiempo: verdadero },  
)
```

2. Ahora solo necesitamos crear una nueva entrada de blog ejecutando el script de ejemplo. Veremos que la última entrada insertada ahora tiene las marcas de tiempo `createdAt` y `updatedAt`.

```
$ nodo src/ejemplo.js
```

3. Para comprobar si la marca de tiempo "updateAt" funciona, intentemos actualizar la entrada de blog creada con el método "`findByIdAndUpdate`". Guardemos el resultado de "`await post.save()`" en una constante "`createdPost`" y, a continuación, agreguemos el siguiente código cerca del final del archivo `src/example.js`, antes de llamar a "`Post.find()`":

```
const createdPost = await post.save()  
  
await Post.findByIdAndUpdate(createdPost._id, { $set: { title: '¡Hola de  
nuevo, Mangosta!' }, })
```

4. Ejecute el servidor nuevamente para ver las publicaciones del blog actualizadas:

```
$ nodo src/ejemplo.js
```

Recibirás tres publicaciones, y la última de ellas ahora se ve así:

```
{
  _id: new ObjectId("64281e304e7d65f85ce07e1f"),
  title: 'Hello again, Mongoose!',
  author: 'Daniel Bugl',
  contents: 'This post is stored in a MongoDB database using Mongoose.',
  tags: [ 'mongoose', 'mongodb' ],
  createdAt: 2023-04-01T12:06:08.303Z,
  updatedAt: 2023-04-01T12:06:08.312Z,
  __v: 0
}
```

Figura 3.3 – Nuestro documento actualizado con las marcas de tiempo actualizadas automáticamente

Como podemos ver, usar Mongoose facilita enormemente la gestión de documentos MongoDB. Ahora que hemos definido nuestro modelo de base de datos, ¡comencemos a desarrollar (y escribir pruebas) para las funciones de servicio!

Desarrollo y prueba de funciones de servicio

Hasta ahora, siempre hemos probado el código colocándolo en el archivo src/example.js. Ahora, escribiremos algunas funciones de servicio y aprenderemos a escribir pruebas reales para ellas con Jest.

Configuración del entorno de prueba

Primero, vamos a configurar nuestro entorno de prueba siguiendo estos pasos:

1. Instale jest y mongodb-memory-server como dependencias de desarrollo:

```
$ npm install --save-dev jest@29.7.0 \
servidor de memoria mongodb@9.1.1
```

Jest es un ejecutor de pruebas utilizado para definir y ejecutar pruebas unitarias. \$e mongodb-memory-server La biblioteca nos permite crear una nueva instancia de una base de datos MongoDB, almacenando datos solo en la memoria, para que podamos ejecutar nuestras pruebas en una nueva instancia de base de datos.

2. Cree una carpeta src/test/ para colocar la configuración de nuestras pruebas.

3. En esta carpeta, creamos un archivo src/test/globalSetup.js, donde importaremos MongoMemoryServer de la biblioteca previamente instalada:

```
importar { MongoMemoryServer } desde 'mongodb-memory-server'
```

4. Ahora defina una función globalSetup, que crea un servidor de memoria para MongoDB:

```
exportar función asíncrona predeterminada globalSetup() {
  constante instancia = await MongoMemoryServer.create({
```

5. Al crear MongoMemoryServer, configure la versión binaria en 6.0.4, que es la misma versión que instalamos para nuestro contenedor Docker:

```
binario:  
{ versión: '6.0.4', }, })
```

6. Almacenaremos la instancia de MongoDB como una variable global para poder acceder a ella más adelante.

Función globalTeardown:

```
global.__MONGOINSTANCE = instancia
```

7. También almacenaremos la URL para conectarnos a nuestra instancia de prueba en DATABASE_URL variable de entorno:

```
proceso.env.DATABASE_URL = instancia.getUri()  
}
```

8. Edite src/db/init.js y ajuste DATABASE_URL para que provenga de la variable de entorno para que nuestras pruebas utilicen la base de datos correcta:

```
función de exportación initDatabase() {  
    const DATABASE_URL = proceso.env.DATABASE_URL
```

9. Además, cree un archivo src/test/globalTeardown.js para detener la instancia de MongoDB

Cuando nuestras pruebas hayan terminado, agreguemos el siguiente código dentro de ellas:

```
exportar función asíncrona predeterminada globalTeardown() { await  
    global.__MONGOINSTANCE.stop()  
}
```

10. Ahora, cree un archivo src/test/setupFileAfterEnv.js. Aquí, definiremos una función beforeEach para inicializar la conexión a la base de datos en Mongoose antes de que se ejecuten todas las pruebas y una función afterEach para desconectarse de la base de datos después de que todas las pruebas terminen de ejecutarse:

```
importar mangosta desde 'mangosta' importar  
{ beforeEach, afterEach } desde '@jest/globals'
```

```
importar { initDatabase } desde './db/init.js'
```

```
beforeEach(async () => { await  
    initDatabase() })
```

```
después de todo(async () => {
```

```
    esperar mangosta.desconectar()
})
```

11. \$en, crea un nuevo archivo `jest.config.json` en la raíz de nuestro proyecto, donde definiremos la configuración para nuestras pruebas.

En el archivo `jest.config.json`, primero configuramos el entorno de pruebas en node:

```
{
  "testEnvironment": "nodo",
```

12. A continuación, indique a Jest que utilice `globalSetup`, `globalTeardown` y `setupFileAfterEnv`

los que creamos anteriormente:

```
"globalSetup": "<directorio raíz>/src/test/globalSetup.js",
"globalTeardown": "<directorio raíz>/src/test/globalTeardown.js",
"setupFilesAfterEnv": ["<directorio raíz>/src/test/setupFileAfterEnv.
js"]
}
```

Nota

En este caso, `<rootDir>` es una cadena especial que Jest resuelve automáticamente al directorio raíz . No es necesario rellenar manualmente un directorio raíz.

13. Finalmente, edite el archivo `package.json` y agregue un script de prueba, que ejecutará Jest:

```
"guiones": {
  "prueba": "OPCIONES_DE_NODO==módulos-vm-experimentales bromo",
  "pelusa": "eslint src",
  "preparar": "instalación de Husky"
},
```

Nota

Al momento de escribir este artículo, el ecosistema JavaScript aún está en proceso de migración al estándar de módulos ECMAScript (ESM) . En este libro, ya utilizamos este nuevo estándar. Sin embargo, Jest aún no lo soporta por defecto, por lo que debemos pasar la instrucción `--experimental-vm-modules`. opción al ejecutar Jest.

14. Si intentamos ejecutar este script ahora, veremos que no se encuentran pruebas, pero aún podemos ver que Jest está configurado y funcionando correctamente:

```
$ npm prueba
```

```
⑥ ➔ ~/D/F/ch3 ↵ main± > npm run test

> ch3@0.0.0 test
> NODE_OPTIONS=--experimental-vm-modules jest

No tests found, exiting with code 1
Run with `--passWithNoTests` to exit with code 0
In /Users/dan/Development/Full-Stack-React-Projects/ch3
  14 files checked.
  testMatch: **/_tests_/**/*.[jt]s?(x), **/?(*).+(spec|test).[tj]s?(x) - 0 matches
  testPathIgnorePatterns: /node_modules/ - 14 matches
  testRegex: - 0 matches
  Pattern: - 0 matches
```

Figura 3.4 – Jest se configuró correctamente, pero aún no hemos definido ninguna prueba

Ahora que nuestro entorno de pruebas está configurado, podemos empezar a escribir nuestras funciones de servicio y pruebas unitarias. Siempre es recomendable escribir las pruebas unitarias inmediatamente después de escribir las funciones de servicio, ya que así podremos depurarlas inmediatamente y conservar su comportamiento previsto en la memoria.

Escribiendo nuestra primera función de servicio: createPost

Para nuestra primera función de servicio, crearemos una función para crear una nueva publicación. Luego, podemos escribir pruebas para ella verificando que la función de creación cree una nueva publicación con las propiedades especificadas. Siga estos pasos:

1. Cree un nuevo archivo src/services/posts.js.
2. En el archivo src/services/posts.js, primero importe el modelo Post:

```
importar { Post } desde '../db/models/post.js'
```

3. Defina una nueva función createPost, que toma un objeto con título, autor, contenido y etiquetas como argumentos y crea y devuelve una nueva publicación:

```
exportar función asíncrona createPost({ título, autor, contenidos, etiquetas }) {

  const post = new Post({ título, autor, contenidos, etiquetas })
  devolver esperar post.save()
}
```

Aquí enumeramos específicamente todas las propiedades que queremos que el usuario pueda proporcionar, en lugar de simplemente pasar el objeto completo al constructor de la nueva instancia de Post(). Si bien esto requiere más código, nos permite controlar qué propiedades puede configurar un usuario. Por ejemplo, si posteriormente añadimos permisos a los modelos de base de datos, podríamos permitir que los usuarios los configuren aquí accidentalmente si olvidamos excluir dichas propiedades. Por razones de seguridad, siempre es recomendable tener una lista de propiedades permitidas, en lugar de simplemente pasar el objeto completo.

Después de escribir nuestra primera función de servicio, continuemos escribiendo casos de prueba para ella.

Definición de casos de prueba para la función de servicio `createPost`

Para probar si la función `createPost` funciona como se espera, vamos a definir casos de prueba unitaria para ella usando Jest:

1. Cree una nueva carpeta `src/__tests__/`, que contendrá todas las definiciones de pruebas.

Nota

Como alternativa, los archivos de prueba también pueden ubicarse junto con los archivos relacionados que están probando. Sin embargo, en este libro, utilizamos el directorio `__tests__` para facilitar la distinción entre las pruebas y otros archivos.

2. Crea un nuevo archivo `src/__tests__/posts.test.js` para nuestras pruebas relacionadas con las publicaciones. En este archivo, empieza importando mongoose y las funciones describe, expect y test de `@jest/globals`:

```
importar mangosta de 'mangosta'  
importar {describe, espera, prueba} desde '@jest/globals'
```

3. También importa la función `createPost` desde nuestros servicios y el modelo Post:

```
importar { createPost } desde '../services/posts.js'  
importar { Post } desde '../db/models/post.js'
```

4. \$en, use la función `describe()` para definir una nueva prueba. \$esta función describe un grupo de Pruebas. Podemos llamar a nuestro grupo que crea publicaciones:

```
describe('creando publicaciones', () => {
```

5. Dentro del grupo, definiremos una prueba mediante la función `test()`. Podemos pasar un `async` Función aquí para poder usar la sintaxis `async/await`. Llamamos a la primera prueba; la creación de publicaciones con todos los parámetros debería ser correcta:

```
test('con todos los parámetros debería tener éxito', async () => {
```

6. Dentro de esta prueba, usaremos la función `createPost` para crear una nueva publicación con algunos parámetros:

```
constante post = {  
    Título: '¡Hola Mangosta!',  
    autor: 'Daniel Bugl',  
    Contenido: 'Esta publicación se almacena en una base de datos MongoDB usando  
    Mangosta.',  
    etiquetas: ['mangosta', 'mongodb'],
```

```
    } const createdPost = await createPost(publicación)
```

7. \$en, verifique que devuelva una publicación con un ID utilizando la función expect() de Jest y el comparador toBeInstanceOf para verificar que sea un ObjectId:

```
    esperar(createdPost._id).toBeInstanceOf(mangosta.Types.  
        Id. de objeto)
```

8. Ahora usa Mongoose directamente para encontrar la publicación con el ID dado:

```
constante encontradoPost = await Post.findById(createdPost._id)
```

9. Esperamos que foundPost sea igual a un objeto que contenga al menos las propiedades del objeto de publicación original que definimos. Además, esperamos que la publicación creada tenga las marcas de tiempo createdAt y updatedAt:

```
    esperar(postencontrado).toEqual(esperar.objetoConteniendo(post))  
    esperar(postencontrado.creadoEn).toBeInstanceOf(Fecha)  
    esperar(postencontrado.actualizadoEn).toBeInstanceOf(Fecha) })
```

10. Además, se define una segunda prueba, llamada "crear publicaciones sin título", que debería fallar. Dado que se definió que el título es obligatorio, no debería ser posible crear una publicación sin él.

```
test('sin título debería fallar', async () => { const post = { author: 'Daniel Bugl',  
    contents: 'Publicación  
    sin título', tags: ['empty'],  
}
```

11. Utilice una construcción try/catch para capturar el error y espere() que el error sea un Mongoose ValidationError, que nos indica que el título es obligatorio:

```
    intenta  
    { await createPost(publicación) }  
    catch (err) {  
        esperar(err).toBeInstanceOf(mongoose.Error.ValidationError) esperar(err.message).toContain(`title`  
            es obligatorio')  
    }})
```

12. Finalmente, realice una prueba llamada "Crear publicaciones con parámetros mínimos" que debería tener éxito y solo ingresar el título:

```
test('con parámetros mínimos debería tener éxito', async () => {
  constante post = {
    título: 'Sólo un título',
  }
  const createdPost = await createPost(publicación)
  esperar(createdPost._id).toBeInstanceOf(mangosta.Types.
    Id. de objeto)
  })
})
```

13. Ahora que hemos definido nuestras pruebas, ejecute el script que definimos anteriormente:

```
$ npm prueba
```

Como podemos ver, al usar pruebas unitarias podemos hacer pruebas aisladas en nuestras funciones de servicio sin tener que definir y acceder manualmente a las rutas o escribir algunas configuraciones de prueba manuales. Estas pruebas también tienen la ventaja adicional de que cuando cambiamos el código más tarde, podemos asegurarnos de que el comportamiento definido previamente no haya cambiado al volver a ejecutar las pruebas.

Definición de una función para listar publicaciones

Después de definir una función para crear publicaciones, ahora vamos a definir una función interna `listPosts`, que nos permite consultar publicaciones y definir un orden de clasificación. Luego, vamos a utilizar esta función para definir las funciones `listAllPosts`, `listPostsByAuthor` y `listPostsByTag`:

1. Edite el archivo `src/services/posts.js` y defina una nueva función al final del archivo.

La función \$e acepta una consulta y un argumento de opciones (con `sortBy` y `sortOrder` propiedades). Con `sortBy`, podemos definir por qué campo queremos ordenar y el `sortOrder`. El argumento nos permite especificar si las publicaciones deben ordenarse en orden ascendente o descendente. De forma predeterminada, enumeramos todas las publicaciones (objeto vacío como consulta) y mostramos primero las publicaciones más nuevas (ordenadas por `createdAt`, en orden descendente):

```
función asíncrona listPosts(
  consulta = {},
  { sortBy = 'createdAt', sortOrder = 'descending' } = {},
)
```

2. Podemos usar el método `.find()` de nuestro modelo Mongoose para listar todas las publicaciones, pasando un argumento para ordenarlos:

```
devolver esperar Post.find(consulta).sort({ [sortBy]: sortOrder })
```

3. Ahora podemos definir una función para listar todas las publicaciones, que simplemente pasa un objeto vacío como consulta:

```
exportar función asíncrona listAllPosts(opciones) {  
    devolver lista de esperaPosts({}, opciones)  
}
```

4. De manera similar, podemos crear una función para enumerar todas las publicaciones de un determinado autor pasando autor a el objeto de consulta:

```
exportar función asíncrona listPostsByAuthor(autor, opciones) {  
    devolver lista de esperaPosts({ autor }, opciones)  
}
```

5. Por último, define una función para listar publicaciones por etiqueta:

```
exportar función asíncrona listPostsByTag(etiquetas, opciones) {  
    devolver lista de esperaPosts({etiquetas}, opciones)  
}
```

En MongoDB, podemos simplemente hacer coincidir cadenas en un array, haciendo coincidir la cadena como si fuera un solo valor. Solo necesitamos agregar una consulta para las etiquetas: 'nodejs'. MongoDB devolverá todos los documentos que contengan la cadena 'nodejs' en su array de etiquetas.

Nota

El operador \$e { [variable]: ... } resuelve la cadena almacenada en la variable en un nombre clave para el objeto creado. Por lo tanto, si nuestra variable contiene 'createdAt', el objeto resultante será { createdAt: ... }.

Después de definir la función de lista de publicaciones, también escribimos casos de prueba para ella.

Definición de casos de prueba para publicaciones de listas

Definir casos de prueba para las entradas de lista es similar a crear entradas. Sin embargo, ahora necesitamos crear un estado inicial con algunas entradas en la base de datos para poder probar las funciones de lista. Podemos hacerlo usando la función beforeEach(), que ejecuta código antes de cada caso de prueba . Podemos usar la función beforeEach() para todo el archivo de prueba o solo para cada prueba dentro de un grupo describe(). En nuestro caso, la definiremos para todo el archivo, ya que las entradas de ejemplo serán útiles cuando probemos la función eliminar entrada más adelante:

1. Edite el archivo src/_tests__/posts.js, ajuste la declaración de importación para importar la función beforeEach desde @jest/globals e importe las distintas funciones para enumerar las publicaciones de nuestros servicios:

```
importar {describe, espera, prueba, beforeEach } de '@jest/  
globales'
```

```
importar {crearPost,
          listaTodasLasPublicaciones,
          listaPublicacionesPorAutor,
          listaPublicacionesPorEtiqueta,
      } de './servicios/posts.js'
```

2. Al final del archivo, defina una serie de publicaciones de muestra:

```
constante samplePosts = [
    { título: 'Aprendiendo Redux', autor: 'Daniel Bugl', etiquetas: ['redux'] },
    { título: 'Aprenda los ganchos de React', autor: 'Daniel Bugl', etiquetas: ['react'] },
    {
        Título: 'Proyectos React Full-Stack',
        autor: 'Daniel Bugl',
        etiquetas: ['react', 'nodejs'],
    },
    { título: 'Guía de TypeScript' },
]
```

3. Ahora, defina una matriz vacía que se llenará con las publicaciones creadas. \$en, defina una función beforeEach que primero borra todas las publicaciones de la base de datos y la matriz de publicaciones de muestra creadas. Luego, crea las publicaciones de muestra en la base de datos nuevamente para cada una de las publicaciones definidas anteriormente en la matriz. \$is garantiza que tengamos un estado consistente de la base de datos antes de ejecutar cada caso de prueba y que tengamos una matriz con la que comparar al probar las funciones de lista de publicaciones:

```
deje que createdSamplePosts = []

antes de cada(async) => {
    esperar Post.deleteMany({})
    createdSamplePosts = []
    para (const post de samplePosts) {
        const createdPost = nueva publicación(publicación)
        createdSamplePosts.push(await createdPost.save())
    }
})
```

Para garantizar que nuestras pruebas unitarias sean modulares e independientes entre sí, insertamos publicaciones en la base de datos directamente utilizando funciones de Mongoose (en lugar de la función createPost).

4. Ahora que tenemos algunas publicaciones de ejemplo listas, escribamos nuestro primer caso de prueba, que simplemente listará todas las publicaciones. Definiremos un nuevo grupo de prueba para listar publicaciones y una prueba para verificar que todas las publicaciones de ejemplo estén listadas por la función `listAllPosts()`:

```
describe('listado de publicaciones', () => {
  test('debería devolver todas las publicaciones', async () => {
    const posts = await listaTodosLosPosts()
    esperar(publicaciones.longitud).igual(publicacionesDeMuestracreadas.longitud)
  })
})
```

5. A continuación, realice una prueba para verificar que el orden predeterminado muestre primero las publicaciones más recientes. Ordenamos manualmente la matriz `createdSamplePosts` por `createdAt` (en orden descendente) y luego comparamos las fechas ordenadas con las devueltas por la función `listAllPosts()`:

```
test('debería devolver publicaciones ordenadas por fecha de creación descendente por defecto', async() => {
  const posts = await listaTodosLosPosts()
  const sortedSamplePosts = createdSamplePosts.sort(
    (a, b) => b.creadoEn - a.creadoEn,
  )
  esperar(publicaciones.map((publicación) => publicación.creadaEn)).toEqual(
    sortedSamplePosts.map((publicación) => publicación.creadaEn),
  )
})
```

Nota

La función `$e .map()` aplica una función a cada elemento de una matriz y devuelve el resultado. En nuestro caso, seleccionamos la propiedad `createdAt` de todos los elementos del array. No podemos comparar directamente los arrays entre sí, ya que Mongoose devuelve documentos con mucha información adicional en metadatos ocultos, que Jest intentará comparar.

6. Además, defina un caso de prueba donde el valor de `sortBy` se cambie a `updatedAt` y el valor de `sortOrder` se cambie a ascendente (mostrando primero las publicaciones actualizadas más antiguas):

```
test('debería tener en cuenta las opciones de clasificación proporcionadas',
async() => {
  const posts = await listaAllPosts({
    ordenar por: 'actualizado en',
    sortOrder: 'ascendente',
  })
  const sortedSamplePosts = createdSamplePosts.sort(
    (a, b) => a.actualizadoEn - b.actualizadoEn,
  )
  esperar(publicaciones.map((publicación) => publicación.actualizadoEn)).toEqual(
```

```

        sortedSamplePosts.map((publicación) => publicación.actualizadoEn),
    )
})

```

7. Sí, agrega una prueba para garantizar que el listado de publicaciones por autor funcione:

```

test('debería poder filtrar publicaciones por autor', async () => {
  const posts = await
  listPostsByAuthor('Daniel Bugl') expect(posts.length).toBe(3)})

```

Nota

Controlamos el entorno de pruebas mediante la creación de un conjunto específico de publicaciones de muestra antes de ejecutar cada caso de prueba. Podemos usar este entorno controlado para simplificar nuestras pruebas. Como ya sabemos que solo hay tres publicaciones con ese autor, podemos simplemente comprobar si la función devolvió exactamente tres publicaciones. Esto simplifica nuestras pruebas y las mantiene seguras, ya que controlamos completamente el entorno.

8. Por último, añade una prueba para verificar que el listado de publicaciones por etiqueta funciona:

```

test('debería poder filtrar publicaciones por etiqueta', async () => {
  const posts = await listPostsByTag('nodejs')
  esperar(posts.length.toBe(1))})

```

9. Ejecute las pruebas nuevamente y observe cómo pasan todas:

```

$ npm prueba
PASS  src/_tests__/posts.test.js
  creating posts
    ✓ with all parameters should succeed (16 ms)
    ✓ without title should fail (5 ms)
    ✓ with minimal parameters should succeed (6 ms)
  listing posts
    ✓ should return all posts (4 ms)
    ✓ should return posts sorted by creation date descending by default (4 ms)
    ✓ should take into account provided sorting options (4 ms)
    ✓ should be able to filter posts by author (3 ms)
    ✓ should be able to filter posts by tag (4 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        0.349 s, estimated 1 s
Ran all test suites.

```

Figura 3.5 – ¡Todas nuestras pruebas pasaron exitosamente!

Como podemos ver, para algunas pruebas, necesitamos preparar un estado inicial. En nuestro caso, solo tuvimos que crear algunas entradas, pero este estado inicial puede volverse más sofisticado. Por ejemplo, en una plataforma de blogs más avanzada, podría ser necesario crear primero una cuenta de usuario, luego un blog en la plataforma y, finalmente, crear entradas para dicho blog. En ese caso, podríamos crear funciones de utilidad de prueba, como `createTestUser`, `createTestBlog` y `createTestPost`, e importarlas en nuestras pruebas. Podemos usar estas funciones en `beforeEach()` en varios archivos de prueba en lugar de hacerlo manualmente cada vez. Dependiendo de la estructura de su aplicación, podrían necesitarse diferentes funciones de utilidad de prueba, así que no dude en definirlas como considere oportuno.

Después de definir los casos de prueba para la función de lista de publicaciones, continuemos definiendo las funciones de obtener publicación individual, actualizar publicación y eliminar publicación.

Definición de las funciones obtener publicación única, actualizar y eliminar publicación

Las funciones de obtener, actualizar y eliminar publicaciones individuales se pueden definir de forma muy similar a la función de lista de publicaciones. Veamos esto rápidamente:

1. Edite el archivo `src/services/posts.js` y defina una función `getPostById` de la siguiente manera:

```
exportar función asíncrona getPostById(postId) {  
    regreso en espera Post.findById(postId)  
}
```

Puede parecer trivial definir una función de servicio que solo llame a `Post.findById`, pero es recomendable hacerlo de todos modos. Más adelante, podríamos querer añadir restricciones adicionales, como el control de acceso. Tener la función de servicio nos permite modificarla solo en un punto y no tener que preocuparnos por olvidarla. Otra ventaja es que, si, por ejemplo, queremos cambiar el proveedor de la base de datos posteriormente, el desarrollador solo tiene que preocuparse de que las funciones de servicio vuelvan a funcionar, y pueden verificarse con los casos de prueba.

2. En el mismo archivo, defina la función `updatePost`. Esta tomará el ID de una publicación existente y un objeto de parámetros a actualizar. Usaremos la función `findOneAndUpdate` de Mongoose, junto con el operador `$set`, para cambiar los parámetros especificados. Como tercer argumento, proporcionamos un objeto `options` con `new: true` para que la función devuelva el objeto modificado en lugar del original:

```
exportar función asíncrona updatePost(postId, { título, autor, contenidos, etiquetas }) {  
  
    devolver esperar Post.findOneAndUpdate(  
        { _id: ID de publicación },  
        { $set: { título, autor, contenido, etiquetas } },  
        { nuevo: verdadero },  
    )  
}
```

3. En el mismo archivo, defina también una función `deletePost`, que simplemente toma el ID de una publicación existente y la elimina de la base de datos:

```
exportar función asíncrona deletePost(postId) {  
    regreso aguarda Post.deleteOne({ _id: postId })  
}
```

Consejo

Dependiendo de tu aplicación, podrías querer establecer una marca de tiempo "deleteOn" en lugar de eliminarlo inmediatamente. Si en, configura una función que obtenga todos los documentos eliminados durante más de 30 días y los elimine. Por supuesto, esto significa que siempre debemos filtrar las publicaciones ya eliminadas en la función "listPosts" y que debemos crear casos de prueba para este comportamiento.

4. Edite el archivo `src/_tests_/posts.js` e importe la función `getPostById`:

```
obtenerPostById,  
} de '../servicios/posts.js'
```

5. Agregue pruebas para obtener una publicación por ID y no obtener una publicación porque el ID no existía en la base de datos:

```
describe('obteniendo una publicación', () => {  
    test('debería devolver la publicación completa', async () => {  
        constante post = await getPostById(createdSamplePosts[0]._id)  
        esperar(post.toObject()).toEqual(createdSamplePosts[0].  
            aObjeto())  
    })  
  
    test('debería fallar si el id no existe', async () => {  
        const post = await getPostById('00000000000000000000000000000000')  
        esperar(publicar).toEqual(nulo)  
    })  
})
```

En la primera prueba, usamos `.toObject()` para convertir el objeto Mongoose con todas sus propiedades internas y metadatos en un simple objeto JavaScript (POJO) para que podamos compararlo con el objeto de publicación de muestra comparando todas las propiedades.

6. A continuación, importe la función `updatePost`:

```
actualizarPublicar,  
} de '../servicios/posts.js'
```

7. `$en`, agregamos pruebas para actualizar una publicación correctamente. Agregamos una prueba para verificar que la propiedad especificada se modificó y otra para verificar que no interfiera con otras propiedades:

```
describe('actualizando publicaciones', () => { test('debería
    actualizar la propiedad especificada', async () => { await updatePost(createdSamplePosts[0]._id, {
        autor: 'Autor de la prueba',
    })
    const updatedPost = await
    Post.findById(createdSamplePosts[0]._id)
    expect(updatedPost.author).toEqual('Autor de la prueba') })

    test('no debe actualizar otras propiedades', async () => { await
        updatePost(createdSamplePosts[0]._id, {
            autor: 'Autor de la prueba',
        })
        const updatedPost = await Post.
        findById(createdSamplePosts[0]._id)
        esperar(updatedPost.title).toEqual('Aprendiendo Redux') })
```

8. Además, agregue una prueba para garantizar que la marca de tiempo "updateAt" se haya actualizado. Para ello, primero convierta los objetos de fecha a números usando `.getTime()` y luego compárelos con el comparador `expect(...).toBeGreaterThan(...)`:

```
test('debería actualizar la marca de tiempo updateAt', async () => {
    esperar actualizaciónPost(createdSamplePosts[0]._id, {
        autor: 'Autor de la prueba', }) const

    updatedPost = await Post.
    findById(createdSamplePosts[0]._id)
    esperar(updatedPost.updateAt.getTime()).toBeGreaterThan(
        createdSamplePosts[0].updateAt.getTime(),
    )
})
```

9. Agregue también una prueba fallida para ver si la función `updatePost` devuelve nulo cuando no se encuentra ninguna publicación con un ID coincidente:

```
test('debería fallar si el id no existe', async () => {
    const post = await updatePost('00000000000000000000000000000000', {
        autor: 'Autor de la prueba',
    })
```

```

        esperar(publicar).toEqual(nulo)
    })
})

```

10. Por último, importe la función deletePost:

```

eliminarPublicación,
} de '../servicios/posts.js'

```

11. \$en, agrega pruebas para eliminaciones exitosas y fallidas verificando si la publicación fue eliminada y verificando el removedCount devuelto:

```

describe('eliminando publicaciones', () => {
  test('debería eliminar la publicación de la base de datos', async () => {
    constante resultado = await deletePost(createdSamplePosts[0]._id)
    esperar(resultado.deletedCount).toEqual(1)
    const removedPost = aguardar publicación.
      findById(publicacionesdemuestracreadas[0]._id)
    esperar(publicacióneliminada).toEqual(nulo)
  })

  test('debería fallar si el id no existe', async () => {
    const resultado = await deletePost('00000000000000000000000000000000')
    esperar(resultado.deletedCount).toEqual(0)
  })
})

```

12. Finalmente, vuela a ejecutar todas las pruebas; todas deberían pasar:

```
$ npm prueba
```

Escribir pruebas para funciones de servicio puede ser tedioso, pero nos ahorrará mucho tiempo a largo plazo. Añadir funcionalidades adicionales posteriormente, como el control de acceso, puede cambiar el comportamiento básico de las funciones del servicio. Al realizar pruebas unitarias, podemos garantizar que no se altere el comportamiento existente al añadir nuevas funcionalidades.

Usando la extensión Jest VS Code

Hasta ahora, hemos ejecutado nuestras pruebas usando Jest desde la Terminal. \$ere también es una extensión de Jest para VS Code, que podemos usar para que la ejecución de pruebas sea más visual. La extensión \$e es especialmente útil para proyectos grandes con muchas pruebas en varios archivos. Además, la extensión puede supervisar y volver a ejecutar las pruebas automáticamente si cambiamos las definiciones. Podemos instalar la extensión de la siguiente manera:

1. Vaya a la pestaña Extensiones en la barra lateral de VS Code.
2. Ingrese Orta.vscode-jest en el cuadro de búsqueda para encontrar la extensión Jest.

3. Instale la extensión presionando el botón Instalar .

4. Ahora ve al ícono de prueba recién agregado en la barra lateral (debería ser un ícono de química %ask):

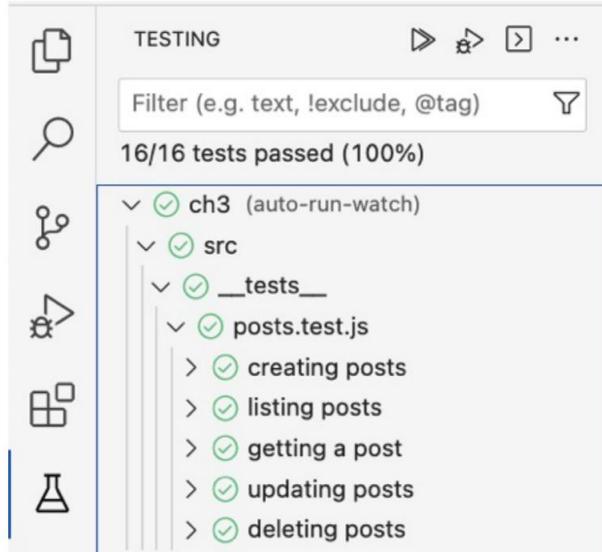


Figura 3.6 – La pestaña Pruebas en VS Code proporcionada por la extensión Jest

La extensión Jest nos proporciona una vista general de todas las pruebas definidas. Podemos pasar el cursor sobre ellas y pulsar el ícono de reproducción para volver a ejecutar una prueba específica. Por defecto, la extensión Jest habilita la función de supervisión automática (como se puede ver en la Figura 3.6). Si está habilitada , la extensión volverá a ejecutar las pruebas automáticamente al guardar los archivos de definición de prueba. ¡Eso es muy práctico!

Ahora que hemos definido y probado nuestras funciones de servicio, ¡podemos comenzar a usarlas al definir rutas, lo que haremos a continuación!

Proporcionar una API REST mediante Express

Con nuestras capas de datos y servicios configuradas, contamos con un buen marco para escribir nuestro backend. Sin embargo, aún necesitamos una interfaz que permita a los usuarios acceder a nuestro backend. Esta interfaz será...

Una API REST (Transferencia de Estado Representacional) . Una API REST proporciona una forma de acceder a nuestro servidor mediante solicitudes HTTP, lo cual podemos usar al desarrollar nuestro frontend.

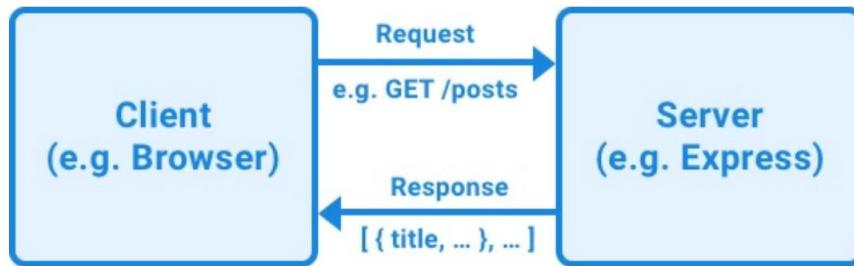


Figura 3.7 – La interacción entre el cliente y el servidor mediante solicitudes HTTP

Como podemos ver, los clientes pueden enviar solicitudes a nuestro servidor backend y el servidor responderá a ellas. Hay cinco métodos comúnmente utilizados en una arquitectura basada en REST:

- GET: \$is se utiliza para leer recursos. Generalmente, no debería influir en el estado de la base de datos y, dada la misma entrada, debería devolver la misma salida (a menos que el estado de la base de datos se haya modificado mediante otras solicitudes). Este comportamiento se denomina idempotencia. En respuesta a una solicitud GET exitosa, el servidor suele devolver el/los recurso(s) con un código de estado 200 OK.
- POST: \$is se utiliza para crear nuevos recursos, a partir de la información proporcionada en el cuerpo de la solicitud. En respuesta a una solicitud POST exitosa, un servidor generalmente devuelve el objeto recién creado con un código de estado 201 Creado o devuelve una respuesta vacía (con el código de estado 201 Creado) con una URL en el encabezado Ubicación que apunta al recurso recién creado.
- PUT: \$is se utiliza para actualizar un recurso existente con un ID determinado, reemplazándolo completamente con los nuevos datos proporcionados en el cuerpo de la solicitud. En algunos casos, también se puede utilizar para crear un nuevo recurso con un ID especificado por el cliente. En respuesta a una solicitud PUT exitosa, el servidor devuelve el recurso actualizado con un código de estado 200 OK, 204 Sin contenido si no devuelve el recurso actualizado o 201 Creado si creó un nuevo recurso.
- PATCH: \$is se utiliza para modificar un recurso existente con un ID determinado, actualizando únicamente los campos especificados en el cuerpo de la solicitud en lugar de reemplazar el recurso completo. En respuesta a una solicitud PATCH exitosa, el servidor devuelve el recurso actualizado con 200 OK o 204 Sin contenido si no lo devuelve.
- ELIMINAR: \$is se utiliza para eliminar un recurso con un ID determinado. En respuesta a una ELIMINACIÓN exitosa solicitud, un servidor devuelve el recurso eliminado con 200 OK o 204 Sin contenido si no devuelve el recurso eliminado.

Las rutas de la API HTTP REST suelen definirse en una estructura similar a una carpeta. Siempre es recomendable anteponer /api/v1/ a todas las rutas (v1 es la versión de la definición de la API, comenzando por 1). Si queremos cambiar la definición de la API más adelante, podemos ejecutar fácilmente /api/v1/ y /api/v2/ en paralelo durante un tiempo hasta que se haya migrado todo.

Definiendo nuestras rutas API

Ahora que hemos aprendido cómo funcionan las API REST HTTP, comenzemos por definir rutas para nuestro backend, cubriendo toda la funcionalidad que ya hemos implementado en las funciones de servicio:

- GET /api/v1/posts: Obtener una lista de todas las publicaciones
- GET /api/v1/posts?sortBy=updatedAt&sortOrder=ascending: Obtener una lista de todas las publicaciones ordenadas por updatedAt (ascendente)

Nota

Todo lo que aparece después del símbolo ? se denomina cadena de consulta y sigue el formato clave1=valor1&clave2=valor2&.... La cadena de consulta se puede utilizar para proporcionar parámetros opcionales adicionales a una ruta.

- GET /api/v1/posts?author=daniel: Obtener una lista de publicaciones del autor "daniel"
- GET /api/v1/posts?tag=react: Obtener una lista de publicaciones con la etiqueta react
- GET /api/v1/posts/:id: Obtener una sola publicación por ID
- POST /api/v1/posts: Crear una nueva publicación
- PATCH /api/v1/posts/:id: Actualizar una publicación existente por ID
- ELIMINAR /api/v1/posts/:id: Eliminar una publicación existente por ID

Como podemos ver, al combinar nuestras funciones de servicio ya desarrolladas con lo aprendido sobre las API REST, podemos definir fácilmente rutas para nuestro backend. Ahora que hemos definido nuestras rutas, configaremos Express y nuestro servidor backend para poder exponerlas.

Nota

\$is es solo un ejemplo de cómo se puede diseñar una API REST. Su propósito es servir de ejemplo para iniciarte en el desarrollo full-stack. Más adelante, si tienes tiempo libre, puedes consultar otros recursos, como <https://standards.rest>, para profundizar tus conocimientos sobre el diseño de API REST.

Configuración de Express

Express es un framework de aplicaciones web para Node.js. Ofrece funciones de utilidad para definir fácilmente rutas para API REST y servir a servidores HTTP. Express también es muy extensible y existen numerosos plugins para él en el ecosistema JavaScript.

Nota

Si bien Express es el framework más conocido al momento de escribir este artículo, también existen frameworks más recientes, como Koa (<https://koajs.com>) o Fastify (<https://fastify.dev>). Koa fue diseñado por el equipo detrás de Express, pero busca ser más compacto, expresivo y robusto. Fastify se centra en la eficiencia y la reducción de gastos generales. No dude en consultarlos por su cuenta para ver si se ajustan mejor a sus necesidades.

Antes de poder configurar las rutas, tomémonos un tiempo para configurar nuestra aplicación Express y el servidor backend siguiendo estos pasos:

1. Primero, instale la dependencia expressa:

```
$ npm install express@4.18.2
```

2. Cree un nuevo archivo src/app.js. Este archivo contendrá todo lo necesario para configurar nuestra aplicación Express. En este archivo, primero importe express:

```
importar express desde 'express'
```

3. \$en crea una nueva aplicación Express, de la siguiente manera:

```
aplicación constante = express()
```

4. Ahora podemos definir rutas en la aplicación Express. Por ejemplo, para definir una ruta GET, podemos escribir el siguiente código:

```
aplicación.get('/', (req, res) => {
    res.send('¡Hola desde Express!')
})
```

5. Exportamos la app para poder usarla en otros archivos:

```
exportar { aplicación }
```

6. A continuación, necesitamos crear un servidor y especificar un puerto, de forma similar a como hicimos antes al crear un servidor HTTP. Para ello, creamos un nuevo archivo src/index.js. En este archivo, importamos la aplicación Express:

```
importar { app } desde './app.js'
```

7. \$en, definimos un puerto, hacemos que la aplicación Express lo escuche y registramos un mensaje que nos dice dónde se está ejecutando el servidor:

```
constante PUERTO = 3000
aplicación.listen(PUERTO)
console.info('servidor express ejecutándose en http://localhost:${PUERTO}')
```

8. Edite package.json y agregue un script de inicio para ejecutar nuestro servidor:

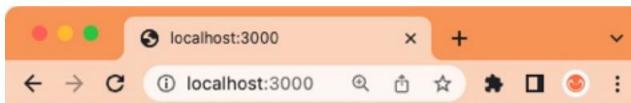
```
"guiones": {  
  "inicio": "nodo src/index.js",
```

9. Ejecute el servidor backend ejecutando el siguiente comando:

```
$ npm start
```

10. Ahora, navegue a <http://localhost:3000/> en su navegador y verá Hola desde

¡Exprés! Se imprime, igual que antes con el servidor HTTP simple:



Hello from Express!

Figura 3.8 – ¡Accediendo a nuestra primera aplicación Express desde el navegador!

¡Eso es todo lo que hay que hacer para configurar una aplicación Express sencilla! Ahora podemos seguir definiendo rutas usando la aplicación.

get() para rutas GET, app.post() para rutas POST, etc. Sin embargo, antes de empezar a desarrollar nuestras

rutas, dediquemos un tiempo a mejorar nuestro entorno de desarrollo. Primero, debemos crear PORT.

y DATABASE_URL configurables para que podamos cambiarlos sin tener que cambiar el código.

Para ello, vamos a utilizar variables de entorno.

Uso de dotenv para configurar variables de entorno

Una buena forma de cargar variables de entorno es usar dotenv, que carga variables de entorno desde archivos .env a nuestro process.env. Esto hace que sea fácil definir variables de entorno para el desarrollo local mientras mantiene la posibilidad de configurarlas de manera diferente en, por ejemplo, un entorno de prueba. Siga estos pasos para configurar dotenv:

1. Instale la dependencia dotenv:

```
$ npm install dotenv@16.3.1
```

2. Edite src/index.js, importe dotenv y llame a dotenv.config() para inicializar las variables de entorno. Debemos hacer esto antes de llamar a cualquier otro código en nuestra aplicación:

```
importar dotenv desde 'dotenv'  
dotenv.config()
```

3. Ahora podemos empezar a reemplazar nuestras variables estáticas con variables de entorno. Edite src/index.js y reemplace el puerto estático 3000 con process.env.PORT:

```
const PUERTO = proceso.env.PUERTO
```

4. Ya hemos migrado la función initDatabase para utilizar process.env.DATABASE_URL
Anteriormente, al configurar Jest, podemos editar src/index.js e importar allí initDatabase:

```
importar { initDatabase } desde './db/init.js'
```

5. Ajuste el código existente para que primero llame a initDatabase y, solo cuando se inicialice la base de datos, inicie la aplicación Express. Ahora también podemos gestionar errores al conectarnos a la base de datos añadiendo un bloque try/catch:

```
intentar {
    esperar initDatabase()

    const PUERTO = proceso.env.PUERTO
    aplicación.listen(PUERTO)

    console.info('servidor express ejecutándose en http://
localhost:${PUERTO}')
} atrapar (err) {
    console.error('error al conectar a la base de datos:', err)
}
```

6. Finalmente, crea un archivo .env en la raíz del proyecto y define allí las dos variables de entorno:

```
PUERTO=3000
URL_DE_BASE_DE_DATOS=mongodb://localhost:27017/blog
```

7. Deberíamos excluir el archivo .env del repositorio Git, ya que solo se utiliza para el desarrollo local.
Edite .gitignore y agréguele .env en una nueva línea:

```
.env
```

Actualmente, no disponemos de información relevante en nuestras variables de entorno, pero es recomendable hacerlo ya. Más adelante, podríamos tener credenciales en las variables de entorno que no queremos que se envíen accidentalmente a un repositorio de Git.

8. Para que sea más fácil para alguien comenzar con nuestro proyecto, podemos crear una copia de nuestro .env y duplíquelo en .env.template, asegurándonos de que no contenga credenciales confidenciales. Estas podrían almacenarse, por ejemplo, en un gestor de contraseñas compartido.

9. Si sigue ejecutándose desde antes, detenga el servidor (presionando Ctrl + C en la Terminal) y

Inicie de nuevo de la siguiente manera:

```
$ npm start
```

Obtendrás el siguiente resultado:

```
○ → ~/D/F/ch3 ↵ main± > npm start

> ch3@0.0.0 start
> node src/index.js

successfully connected to database: mongodb://localhost:27017/ch3
express server running on http://localhost:3000
```

Figura 3.9 – Inicialización de la conexión a la base de datos y el servidor Express con variables de entorno

Como podemos ver, dotenv facilita el mantenimiento de variables de entorno para el desarrollo y al mismo tiempo nos permite la posibilidad de cambiarlas en un entorno de integración continua, prueba o producción.

Quizás hayas notado que necesitamos reiniciar el servidor manualmente después de realizar algunos cambios. Esto contrasta marcadamente con la recarga en caliente que traímos de fábrica con Vite, donde cualquier cambio que hagamos se aplica al frontend del navegador al instante. Ahora, dediquemos tiempo a mejorar la experiencia de desarrollo haciendo que el servidor se reinicie automáticamente al realizar cambios.

Usando nodemon para un desarrollo más sencillo

Para que nuestro servidor se reinicie automáticamente al realizar cambios, podemos usar la herramienta nodemon. Esta herramienta nos permite ejecutar nuestro servidor, de forma similar al comando CLI de node. Sin embargo, ofrece la posibilidad de reiniciar el servidor automáticamente al realizar cambios en los archivos fuente.

1. Instale la herramienta nodemon como una dependencia de desarrollo:

```
$ npm install --save-dev nodemon@3.0.2
```

2. Cree un nuevo archivo nodemon.json en la raíz de su proyecto y agréguele el siguiente contenido:

```
{
  "watch": ["./src", ".env", "package-lock.json"]
}
```

Si garantiza que todo el código de la carpeta src/ se monitorice para detectar cambios y se actualizara si se modifica algún archivo dentro de ella. Además, especificamos el archivo .env para cuando se modifican las variables de entorno y el archivo package-lock.json para cuando se agregan o actualizan paquetes.

3. Ahora, edite package.json y defina un nuevo script "dev" que ejecute nodemon:

```
"guiones": {  
  "dev": "nodemon src/index.js",
```

4. Detenga el servidor (si está ejecutándose actualmente) y vuelva a iniciar el ejecutando el siguiente comando:

```
$ npm ejecuta dev
```

5. Como podemos ver, ¡nuestro servidor ahora se ejecuta a través de nodemon! Podemos probarlo cambiando el puerto en el archivo .env:

```
PUERTO=3001  
URL_DE_BASE_DE_DATOS=mongodb://localhost:27017/blog
```

6. Edite también .env.template para cambiar el puerto a 3001:

```
PUERTO=3001
```

7. Mantenga el servidor en funcionamiento.

```
[nodemon] 2.0.22  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): src/**/* .env package.json  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node src/index.js`  
successfully connected to database: mongodb://localhost:27017/ch3  
express server running on http://localhost:3000  
[nodemon] restarting due to changes...  
[nodemon] starting `node src/index.js`  
successfully connected to database: mongodb://localhost:27017/ch3  
express server running on http://localhost:3001
```

Figura 3.10 – Nodemon reinicia automáticamente el servidor después de que cambiamos el puerto

Tras realizar el cambio, nodemon reinició automáticamente el servidor con el nuevo puerto. Ahora tenemos algo similar a la recarga en caliente, pero para el desarrollo backend: ¡genial! Ahora que hemos mejorado la experiencia del desarrollador en el backend, comenzemos a escribir nuestras rutas de API con Express.

¡Mantenga el servidor en funcionamiento (a través de nodemon) para verlo reiniciar y actualizarse en vivo mientras codifica!

Creando nuestras rutas API con Express

Ahora podemos empezar a crear nuestras rutas de API previamente definidas con Express. Empezamos definiendo las rutas GET:

1. Cree un nuevo archivo src/routes/posts.js e importe allí las funciones de servicio:

```
importar {  
    listaTodasLasPublicaciones,  
    listaPublicacionesPorAutor,  
    listaPublicacionesPorEtiqueta,  
    obtenerPostById,  
} de './servicios/posts.js'
```

2. Ahora cree y exporte una nueva función llamada postsRoutes, que toma la aplicación Express como argumento:

```
función de exportación postsRoutes(app) {
```

3. En esta función, define las rutas. Comienza con la ruta GET /api/v1/posts:

```
aplicación.get('/api/v1/posts', async (req, res) => {
```

4. En esta ruta, necesitamos usar los parámetros de consulta (req.query en Express) para asignarlos a los argumentos de nuestras funciones. Queremos poder agregar parámetros de consulta para sortBy, sortOrder, autor y etiqueta:

```
const { sortBy, sortOrder, autor, etiqueta } = req.query  
opciones constantes = { sortBy, sortOrder }
```

5. Antes de llamar a nuestras funciones de servicio, que podrían generar un error si pasamos datos no válidos a las funciones de base de datos, deberíamos agregar un bloque try-catch para manejar los posibles errores adecuadamente:

```
intentar {
```

6. Ahora debemos comprobar si se proporcionó el autor o la etiqueta. Si se proporcionaron ambos, devolvemos un código de estado 400 "Solicitud incorrecta" y un objeto JSON con un mensaje de error mediante la llamada a res.json():

```
si (autor && etiqueta) {  
    devolver res  
        .estado(400)  
        .json({ error: 'consulta por autor o etiqueta, no  
ambos' })
```

7. De lo contrario, llamamos a la función de servicio correspondiente y devolvemos una respuesta JSON en Express mediante `res.json()`.

En caso de error, lo detectamos, lo registramos y devolvemos un código de estado 500:

```

} } de lo contrario si (autor) {
    devolver res.json(await listPostsByAuthor(autor, opciones)) } de lo contrario si
(etiqueta) {

    devolver res.json(await listPostsByTag(etiqueta, opciones)) } de lo contrario { devolver
res.json(await
    listAllPosts(opciones))

} } atrapar (err) {
    console.error('error al listar publicaciones', err) return
    res.status(500).end()

}
}

```

8. A continuación, definimos una ruta de API para obtener una sola publicación. Usamos el parámetro `:id` para poder...

para acceder a él como parámetro dinámico en la función:

```
aplicación.get('/api/v1/posts/:id', async (req, res) => {
```

9. Ahora, podemos acceder a `req.params.id` para obtener la parte `:id` de nuestra ruta y pasarlala a nuestro función de servicio:

```

const { id } = req.params try { const post =
await
    getPostById(id)
}

```

10. Si el resultado de la función es nulo, devolvemos una respuesta 404 porque no se encontró la publicación.

De lo contrario, devolvemos la publicación como una respuesta JSON:

```

si (publicación === nulo) devuelve res.status(404).end() devuelve
res.json(publicación) } catch (err) {

    console.error('error al obtener la publicación', err) return
    res.status(500).end()

}
}

```

De forma predeterminada, Express devolverá la respuesta JSON con el estado 200 OK.

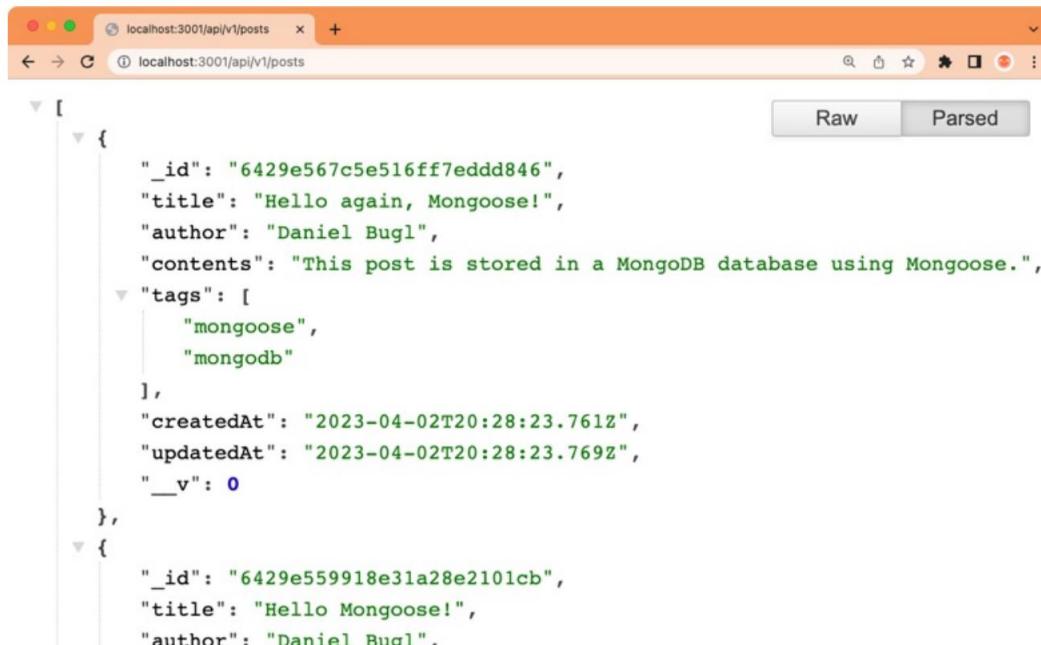
11. Después de definir nuestras rutas GET, aún necesitamos montarlas en nuestra aplicación. Editar src/app.js e importe la función postsRoutes allí:

```
importar { postsRoutes } desde './routes/posts.js'
```

12. \$en, llama a la función postsRoutes(app) después de inicializar nuestra aplicación Express:

```
const app = express()
postsRoutes(app)
```

13. ¡Vaya a <http://localhost:3001/api/v1/posts> para ver la ruta en acción!



The screenshot shows a browser window with the address bar set to `localhost:3001/api/v1/posts`. The page content displays a JSON object representing a post document. The JSON is formatted with syntax highlighting and collapsible sections. The document includes fields like `_id`, `title`, `author`, `contents`, `tags`, `createdAt`, `updatedAt`, and `v`. There are two posts listed.

```
{
  "_id": "6429e567c5e516ff7eddd846",
  "title": "Hello again, Mongoose!",
  "author": "Daniel Bugl",
  "contents": "This post is stored in a MongoDB database using Mongoose.",
  "tags": [
    "mongoose",
    "mongodb"
  ],
  "createdAt": "2023-04-02T20:28:23.761Z",
  "updatedAt": "2023-04-02T20:28:23.769Z",
  "__v": 0
},
{
  "_id": "6429e559918e31a28e2101cb",
  "title": "Hello Mongoose!",
  "author": "Daniel Bugl",
}
```

Figura 3.11 – ¡Nuestra primera ruta API real en acción!

Consejo: Puede instalar una extensión JSON Formatter en su navegador para formatear correctamente la respuesta JSON , como en la Figura 3.11.

Tras definir las rutas GET, necesitamos definir las rutas POST. Sin embargo, estas aceptan un cuerpo, que se formateará como objetos JSON. Por lo tanto, necesitamos una forma de analizar este cuerpo JSON en Express.

Definición de rutas con un cuerpo de solicitud JSON

Para definir rutas con un cuerpo de solicitud JSON en Express, necesitamos usar el módulo body-parser. Este módulo detecta si un cliente envió una solicitud JSON (mirando el encabezado Content-Type) y luego lo analiza automáticamente para que podamos acceder al objeto en req.body.

1. Instale la dependencia body-parser:

```
$ npm install body-parser@1.20.2
```

2. Edite src/app.js e importe el analizador corporal allí:

```
importar bodyParser desde 'body-parser'
```

3. Ahora agregue el siguiente código después de que nuestra aplicación se inicialice para cargar el complemento body-parser como middleware en nuestra aplicación Express:

```
aplicación constante = express()  
aplicación.use(bodyParser.json())
```

Nota

El middleware en Express nos permite realizar acciones antes y después de cada solicitud. En este caso, body-parser lee el cuerpo JSON, lo analiza como JSON y nos proporciona un objeto JavaScript al que podemos acceder fácilmente desde nuestras definiciones de ruta. Cabe destacar que solo las rutas definidas después del middleware tienen acceso a él, por lo que el orden de definición del middleware y las rutas es importante.

4. Despues de cargar el body-parser, editamos src/routes/posts.js e importamos el servicio Funciones necesarias para realizar el resto de nuestras rutas:

```
crearPost,  
actualizarPublicar,  
eliminarPublicación,  
} de '../servicios/posts.js'
```

5. Ahora, definimos la ruta POST /api/v1/posts usando app.post y req.body, dentro de la función postsRoutes:

```
aplicación.post('/api/v1/posts', async (req, res) => {  
    intentar {  
        const post = await createPost(req.body)  
        devolver res.json(publicación)  
    } atrapar (err) {  
        console.error('error al crear publicación', err)  
        devolver res.status(500).end()  
    }  
}
```

```
    }  
})
```

6. De manera similar, podemos definir la ruta de actualización, donde necesitamos hacer uso del parámetro id y el cuerpo de la solicitud:

```
aplicación.patch('/api/v1/posts/:id', async (req, res) => {  
    intentar {  
        const post = await updatePost(req.params.id, req.body)  
        devolver res.json(publicación)  
    } atrapar (err) {  
        console.error('error al actualizar la publicación', err)  
        devolver res.status(500).end()  
    }  
})
```

7. Finalmente, definimos la ruta de eliminación, que no requiere el analizador de cuerpo; solo necesitamos obtener el parámetro id. Devolvemos 404 si no se encontró la publicación y 204 Sin contenido si se eliminó correctamente.

```
aplicación.delete('/api/v1/posts/:id', async (req, res) => {  
    intentar {  
        const { removedCount } = await deletePost(req.params.id)  
        si (removedCount === 0) devuelve res.sendStatus(404)  
        devolver res.status(204).end()  
    } atrapar (err) {  
        console.error('error al eliminar publicación', err)  
        devolver res.status(500).end()  
    }  
})
```

Como podemos ver, Express simplifica enormemente la definición y gestión de rutas, solicitudes y respuestas. Detecta y configura los encabezados automáticamente, lo que le permite leer y enviar respuestas JSON correctamente. También permite cambiar fácilmente el código de estado HTTP.

Ahora que terminamos de definir las rutas con un cuerpo de solicitud JSON, permitamos el acceso a nuestras rutas desde otras URL mediante el uso compartido de recursos de origen cruzado (CORS).

Permitir el acceso desde otras URL mediante CORS

Los navegadores cuentan con una función de seguridad que solo nos permite acceder a las API en la misma URL que la página en la que nos encontramos . Para permitir el acceso a nuestro backend desde otras URL distintas a la propia (por ejemplo, cuando ejecutamos el frontend en un puerto diferente en el siguiente capítulo), necesitamos permitir las solicitudes CORS. Configurémoslo ahora usando la biblioteca cors con Express:

1. Instale la dependencia cors:

```
$ npm install cors@2.8.5
```

2. Edite src/app.js e importe cors allí:

```
import cors from 'cors'
```

3. Ahora agregue el siguiente código después de que nuestra aplicación se inicialice para cargar el complemento cors como middleware en nuestra aplicación Express:

```
const app = express()
app.use(cors())
app.use(bodyParser.json())
```

Ahora que se permiten las solicitudes CORS, ¡podemos comenzar a probar las rutas en un navegador!

Probando las rutas

Después de definir nuestras rutas, podemos probarlas utilizando la función fetch() en el navegador:

1. En su navegador, vaya a <http://localhost:3001/>, abra la consola haciendo clic derecho en una página y haga clic en Inspeccionar, luego vaya a la pestaña Consola .
2. En la consola, ingrese el siguiente código para realizar una solicitud GET para obtener todas las publicaciones:

```
buscar('http://localhost:3001/api/v1/posts')
.then(res => res.json()) .then(console.log)
```

3. Ahora podemos modificar este código para realizar una solicitud POST especificando el encabezado Content-Type para indicarle al servidor que enviaremos JSON y luego enviaremos un cuerpo con JSON. stringify (ya que el cuerpo debe ser una cadena):

```
buscar('http://localhost:3001/api/v1/posts', {
  encabezados: { 'Content-Type': 'application/json' },
  método: 'POST',
  cuerpo: JSON.stringify({ título: 'Publicación de prueba' })
})
.then(res => res.json()) .then(console.log)
```

4. De manera similar, también podemos enviar una solicitud PATCH, de la siguiente manera:

```
fetch('http://localhost:3001/api/v1/posts/  
642a8b15950196ee8b3437b2', { headers: { 'Tipo de  
contenido': 'application/json' },  
método: 'PARCHÉ',  
cuerpo: JSON.stringify({ título: 'Publicación de prueba modificada' })  
})  
.then(res => res.json()) .then(consola.log)
```

¡Asegúrese de reemplazar los ID de MongoDB en la URL con los devueltos por la solicitud POST realizada anteriormente!

5. Finalmente, podemos enviar una solicitud DELETE:

```
obtener('http://localhost:3001/api/v1/posts/  
642a8b15950196ee8b3437b2', {  
método: 'BORRAR',  
})  
.then(res => res.estado)  
.then(consola.log)
```

6. Al realizar una solicitud GET, podemos ver que nuestra publicación ha sido eliminada nuevamente:

```
fetch('http://localhost:3001/api/v1/posts/  
642a8b15950196ee8b3437b2') .then(res =>  
res.status) .then(console.log)
```

Esta solicitud ahora debería devolver un 404.

Consejo: En lugar de la consola del navegador, también puedes usar herramientas de línea de comandos como curl o aplicaciones como Postman para realizar las solicitudes. Si ya estás familiarizado con ellas, puedes usar diferentes herramientas para probarlas.

¡Ahora hemos definido con éxito todas las rutas necesarias para manejar una API de publicación de blog simple!

Resumen

La primera versión de nuestro servicio backend ya está completa, lo que nos permite crear, leer, actualizar y eliminar entradas de blog mediante una API REST (con Express), que luego se almacenan en una base de datos MongoDB (con Mongoose). Además, hemos creado funciones de servicio con pruebas unitarias, definidas con la suite de pruebas Jest. En resumen, en este capítulo hemos logrado sentar una base sólida para nuestro backend.

En el próximo capítulo, Capítulo 4, Integrando un Frontend usando React y TanStack Query, vamos a integrar nuestro backend en un frontend de React usando TanStack Query, una biblioteca para manejar el estado asíncrono y, por lo tanto, los datos obtenidos de nuestro servidor. Esto significa que, después del próximo capítulo, habremos desarrollado nuestra primera aplicación full-stack.

Integración de un frontend usando Consulta de React y TanStack

Tras diseñar, implementar y probar nuestro servicio backend, es hora de crear un frontend para interactuar con el backend. Primero, comenzaremos configurando un proyecto React full-stack basado en el código fuente de Vite y el servicio backend creado en los capítulos anteriores. A continuación, crearemos una interfaz de usuario básica para nuestra aplicación de blog. Finalmente, usaremos TanStack Query, una biblioteca de obtención de datos para gestionar el estado del backend, para integrar la API del backend en el frontend. Al final de este capítulo, ¡habremos desarrollado con éxito nuestra primera aplicación full-stack!

En este capítulo cubriremos los siguientes temas principales:

- Principios de React
- Configuración de un proyecto React de pila completa
- Creación de la interfaz de usuario para nuestra aplicación
- Integración del servicio backend mediante TanStack Query

Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones que aparecen en esos capítulos son las que se utilizan en este libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que ciertos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas en los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-modernos-de-React-de-pila-completa/árbol/principal/capítulo 4>

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/WXqJu2Ut7Hs>

Principios de React

Antes de comenzar a aprender cómo configurar un proyecto React full-stack, repasemos los tres principios fundamentales de React. Estos principios nos permiten escribir fácilmente aplicaciones web escalables:

- Declarativo: En lugar de indicarle a React cómo hacer las cosas, le indicamos qué queremos que haga.

Como resultado, podemos diseñar fácilmente nuestras aplicaciones y React actualizará y renderizará eficientemente solo los componentes correctos cuando cambien los datos. Por ejemplo, el siguiente código, que duplica cadenas en un array, es imperativo, lo cual es lo opuesto a declarativo:

```
entrada constante = ['a', 'b', 'c']
deje resultado = []
para (sea i = 0; i < longitud de entrada; i++) {
    resultado.push(entrada[i] + entrada[i])
}
console.log(resultado) // imprime: [ 'aa', 'bb', 'cc' ]
```

Como podemos ver, en el código imperativo, necesitamos decirle a JavaScript exactamente qué hacer, paso a paso.

Sin embargo, con código declarativo, podemos simplemente decirle a la computadora lo que queremos, de la siguiente manera:

```
entrada constante = ['a', 'b', 'c']
const resultado = entrada.map(str => str + str)
console.log(resultado) // imprime: [ 'aa', 'bb', 'cc' ]
```

En este código declarativo, le decimos a la computadora que queremos mapear cada elemento de la entrada Matriz de str a str + str. Como puede ver, el código declarativo es mucho más conciso.

- Basado en componentes: React encapsula componentes que gestionan su propio estado y vistas y luego nos permite componerlos para crear interfaces de usuario complejas.
- Aprende una vez, escribe en cualquier lugar: React no hace suposiciones sobre tu pila de tecnología e intenta garantizar que puedas desarrollar aplicaciones sin reescribir el código existente tanto como sea posible.

Los tres principios fundamentales de React facilitan la escritura de código, la encapsulación de componentes y su intercambio entre múltiples plataformas. En lugar de reinventar la rueda, React intenta aprovechar al máximo las funcionalidades existentes de JavaScript. Como resultado, aprenderemos patrones de diseño de software aplicables en muchos más casos que el diseño de interfaces de usuario.

Ahora que hemos aprendido los principios fundamentales de React, ¡comencemos a configurar un proyecto React completo!

Configuración de un proyecto React full-stack

Antes de comenzar a desarrollar nuestra aplicación frontend, necesitamos fusionar el framework frontend creado previamente, basado en Vite, con el servicio backend creado en el Capítulo 3, "Implementación de un backend con Express, Mongoose ODM y Jest". Para fusionarlos, sigamos estos pasos:

1. Copie la carpeta ch1 a una nueva carpeta ch4, de la siguiente manera:

```
$ cp -R ch1 ch4
```

2. Copie la carpeta ch3 a una nueva carpeta ch4/backend, de la siguiente manera:

```
$ cp -R ch3 ch4/backend
```

3. Elimine la carpeta .git en la carpeta ch4/backend copiada, de la siguiente manera:

```
$ rm -rf ch4/backend/.git
```

4. Abra la nueva carpeta ch4 en VS Code.

5. Elimine el script de preparación de Husky (la línea está resaltada en el fragmento de código) del archivo backend/package.json, ya que tenemos Husky configurado en el directorio raíz:

```
"scripts": { "dev":  
    "nodemon src/index.js", "start": "node src/index.js",  
    "test": "NODE_OPTIONS=--experimental-vm-  
modules jest", "lint": "eslint src", "prepare": "husky install" },
```

6. Elimine también la siguiente configuración preparada por lint del archivo backend/package.json:

```
"lint-staged": { "**/*.{js,jsx)":  
    [ "npx más bonito --write", "npx  
    eslint --fix"  
  
    ]  
}
```

7. \$en, elimine las carpetas backend/.husky, backend/.vscode y backend/.git.

8. Para asegurarse de que todas las dependencias estén instaladas correctamente, ejecute el siguiente comando en la raíz de la carpeta ch4:

```
$ npm install
```

9. También vaya al directorio backend/ e instale todas las dependencias allí:

```
$ cd backend/  
$ npm install
```

10. Ahora también podemos eliminar los paquetes husky, lint-staged y @commitlint del proyecto backend, ya que los tenemos configurados en la carpeta principal del proyecto:

```
$ npm uninstall husky lint-staged \  
@commitlint/cli @commitlint/config-convencional
```

Consejo

Siempre es recomendable revisar periódicamente qué paquetes aún necesitas y cuáles puedes descartar para mantener tu proyecto limpio. En este caso, copiamos código de otro proyecto, pero no necesitamos la configuración de Husky, lint-staged y commitlint, ya que la tenemos configurada en la raíz de nuestro proyecto.

11. Ahora regrese a la raíz de la carpeta ch4 y ejecute el siguiente comando para iniciar el servidor frontend:

```
$ cd ../  
$ npm ejecuta dev
```

12. Abra la interfaz en su navegador yendo a la URL mostrada por Vite: http://
host local:5173/

13. Abra src/App.jsx, cambie el título de la siguiente manera y guarde el archivo:

Vite + React + Node.js

14. ¡Verás que el cambio se refleja instantáneamente en el navegador!

Después de configurar con éxito nuestro proyecto full-stack combinando nuestros proyectos de capítulos anteriores, ahora comenzemos a diseñar y crear la interfaz de usuario para nuestra aplicación de blog.

Creando la interfaz de usuario para nuestra aplicación

Al diseñar la estructura de un frontend, también debemos considerar la estructura de carpetas para que nuestra aplicación pueda crecer fácilmente en el futuro. Al igual que hicimos con el backend, guardaremos todo nuestro código fuente en la carpeta src/. Luego, podemos agrupar los archivos en carpetas separadas para las diferentes funciones.

Otra forma popular de estructurar proyectos frontend es agrupar el código por rutas. Por supuesto, también es posible combinarlas; por ejemplo, en proyectos de Next.js, podemos agrupar nuestros componentes por características y luego crear otra carpeta y estructura de archivos para las rutas, donde se utilizan los componentes.

Para proyectos full-stack, también es recomendable separar primero el código creando carpetas separadas para la integración de la API y los componentes de la interfaz de usuario.

Ahora, definamos la estructura de carpetas para nuestro proyecto:

1. Cree una nueva carpeta `src/api/`.
2. Cree una nueva carpeta `src/components/`.

Consejo

Es recomendable empezar con una estructura simple y anidar con mayor profundidad solo cuando realmente sea necesario. No dediques demasiado tiempo a pensar en la estructura de los archivos al iniciar un proyecto, ya que, por lo general, no sabes de antemano cómo agruparlos y, de todos modos, esto podría cambiar más adelante.

Después de definir la estructura de carpetas de alto nivel para nuestros proyectos, tomémonos ahora un tiempo para considerar la estructura de componentes.

Estructura del componente

Basándonos en lo que hemos definido en el backend, nuestra aplicación de blog va a tener las siguientes características:

- Visualización de una sola publicación
- Crear una nueva publicación
- Publicaciones de listados
- Filtrar publicaciones
- Ordenar publicaciones

La idea de los componentes en React es que cada componente se ocupe de una sola tarea o elemento de UI.

Deberíamos intentar que los componentes sean lo más precisos posible para poder reutilizar el código. Si nos vemos obligados a copiar y pegar código de un componente a otro, sería buena idea crear un nuevo componente y reutilizarlo en varios otros.

Normalmente, al desarrollar una interfaz, empezamos con una maqueta de la interfaz de usuario. Para nuestra aplicación de blog, una maqueta podría tener el siguiente aspecto:

Title:

Author:

author:

Sort By: / Sort Order:

Full-Stack React Projects

Let's become full-stack developers!

Written by Daniel Bugl

Hello React!

Figura 4.1 – Una maqueta inicial de nuestra aplicación de blog

Nota

En este libro, no abordaremos los frameworks de interfaz de usuario (UI) ni de CSS. Por lo tanto, los componentes se diseñan y desarrollan sin aplicar estilos. En su lugar, el libro se centra en la integración completa de backends con frontends. Siéntete libre de usar un framework de interfaz de usuario (como MUI) o de CSS (como Tailwind) para aplicar estilos a la aplicación del blog.

Al dividir la interfaz de usuario en componentes, utilizamos el principio de responsabilidad única, que establece que cada módulo debe tener responsabilidad sobre una única parte encapsulada de la funcionalidad.

En nuestra maqueta, podemos dibujar cuadros alrededor de cada componente y subcomponente, y asignarles nombres. Tenga en cuenta que cada componente debe tener exactamente una responsabilidad. Empecemos con los componentes fundamentales que conforman la aplicación:

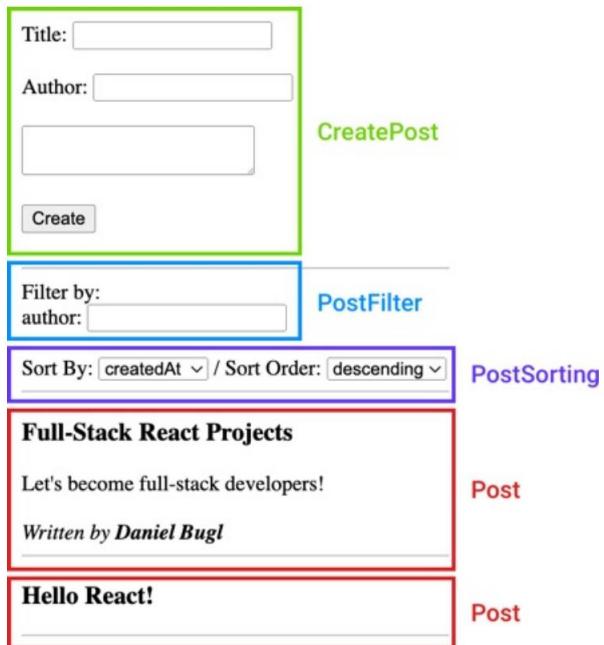


Figura 4.2 – Definición de los componentes fundamentales de nuestra maqueta

Definimos un componente `CreatePost`, con un formulario para crear una nueva publicación, un componente `PostFilter` para filtrar la lista de publicaciones, un componente `PostSorting` para ordenar publicaciones y un componente `Post` para mostrar una sola publicación.

Ahora que hemos definido nuestros componentes fundamentales, vamos a ver qué componentes pertenecen lógicamente juntos, formando así un grupo: podemos agrupar los componentes `Post` en `PostList`, luego crear un componente `App` para agrupar todo junto y definir la estructura de nuestra aplicación.

Ahora que hemos terminado de estructurar nuestros componentes React, podemos pasar a implementar los componentes React estáticos.

Implementación de componentes estáticos de React

Antes de integrar con el backend, modelaremos las características básicas de nuestra aplicación como componentes estáticos de React. Es lógico trabajar primero con la estructura de la vista estática de nuestra aplicación, ya que podemos experimentar y reestructurar la interfaz de usuario si es necesario, antes de añadir la integración a los componentes, lo que dificultaría y haría más tedioso su desplazamiento. También es más fácil trabajar primero solo con la interfaz de usuario, lo que nos permite empezar rápidamente con proyectos y características. Después, podemos pasar a implementar integraciones y gestionar el estado.

Comencemos ahora a implementar los componentes estáticos.

El componente Post

Ya hemos considerado los elementos que debe tener una publicación durante la creación de la maqueta y el diseño del backend. Una publicación debe tener título, contenido y autor.

Implementemos ahora el componente Post:

1. Primero, cree un nuevo archivo src/components/Post.jsx.
2. En ese archivo, importe PropTypes:

```
importar PropTypes desde 'prop-types'
```

3. Defina un componente de función, aceptando propiedades de título, contenido y autor:

```
función de exportación Post({ título, contenido, autor }) {
```

4. A continuación, renderiza todos los accesorios de forma que se asemejen a la maqueta:

```
devolver (
  <artículo>
    <h3>{título}</h3>
    <div>{contenido}</div> {autor &&
      <em>
        <br />
        Escrito por <strong>{author}</strong> </em> )} </article>

  )
}
```

Consejo: Tenga en cuenta que siempre debe preferir el espaciado mediante CSS, en lugar de usar la etiqueta HTML `
`. Sin embargo, en este libro nos centramos en la estructura de la interfaz de usuario y la integración con el backend , por lo que simplemente usamos HTML siempre que sea posible.

5. Ahora, defina propTypes, asegurándose de que solo se requiera el título:

```
Post.propTypes = {
  título: PropTypes.string.isRequired, contenido:
  PropTypes.string, autor: PropTypes.string,
}
```

Información

Los PropTypes se utilizan para validar las propiedades pasadas a los componentes de React y garantizar que se pasen las correctas al usar JavaScript. Al usar un lenguaje con seguridad de tipos, como TypeScript, podemos hacerlo escribiendo directamente las propiedades pasadas al componente.

6. Probemos nuestro componente reemplazando el archivo src/App.jsx con el siguiente contenido:

```
importar { Post } desde './components/Post.jsx'

función de exportación App() {
    devolver (
        <Publicación
            título='Proyectos React de pila completa'
            contents="¡Convertámonos en desarrolladores full-stack!"
            autor='Daniel Bugl'
        />
    )
}
```

7. Edite src/main.jsx y actualice la importación del componente App, porque ya no usamos la exportación predeterminada:

```
importar { App } desde './App.jsx'
```

Información

Personalmente, prefiero no usar las exportaciones predeterminadas, ya que dificultan la reagrupación y reexportación de componentes y funciones desde otros archivos. Además, permiten cambiar los nombres de los componentes, lo cual podría resultar confuso. Por ejemplo, si cambiamos el nombre de un componente, el nombre al importarlo no se cambia automáticamente.

8. Además, elimine la siguiente línea de src/main.jsx:

```
importar './index.css'
```

9. Finalmente, podemos eliminar los archivos index.css y App.css, ya que ya no son necesarios.

Ahora que nuestro componente Post estático se ha implementado, podemos pasar al componente CreatePost.

El componente CreatePost

Ahora implementaremos un formulario para permitir la creación de nuevas publicaciones. Aquí, proporcionamos campos para el autor, y un título y un elemento <textarea> para el contenido de la publicación del blog.

Implementemos ahora el componente CreatePost:

1. Cree un nuevo archivo src/components/CreatePost.jsx.
2. Defina el siguiente componente, que contiene un formulario para ingresar el título, autor y contenido.

de una entrada de blog:

```
función de exportación CreatePost() { return ( <form  
    onSubmit={(e)  
        => e.preventDefault()}>  
        <div>  
            <label htmlFor='create-title'>Título: </label> <input type='text' name='create-  
            title' id='create-title'  
        />  
        </div>  
        <br />  
        <div>  
            <label htmlFor='create-author'>Autor: </label> <input type='text' name='create-  
            author' id='create-author' /> </div> <br /> <textarea /> <br /> <br /> <input type='submit'  
            value='Crear' />  
        </form>  
  
    )  
}
```

En el bloque de código anterior, definimos un controlador onSubmit y llamamos a e.preventDefault() en el objeto de evento para evitar una actualización de la página cuando se envía el formulario.

3. Probemos el componente reemplazando el archivo src/App.jsx con el siguiente contenido:

```
importar { CreatePost } desde './components/CreatePost.jsx'  
  
función de exportación App() { return  
    <CreatePost />  
}
```

Como puede ver, el componente CreatePost se renderiza correctamente. Ahora podemos pasar a los componentes PostFilter y PostSorting.

Consejo: si desea probar varios componentes a la vez y conservar las pruebas para más adelante, o crear una guía de estilo para su propia biblioteca de componentes, debe consultar Storybook (<https://storybook.js.org>), que es una herramienta útil para crear, probar y documentar componentes de UI de forma aislada.

Los componentes PostFilter y PostSorting

Al igual que con el componente CreatePost, crearemos dos componentes que proporcionan campos de entrada para filtrar y ordenar publicaciones. Empecemos con PostFilter:

1. Cree un nuevo archivo src/components/PostFilter.jsx.

2. En este archivo, importamos PropTypes:

```
importar PropTypes desde 'prop-types'
```

3. Ahora, definimos el componente PostFilter y hacemos uso de la propiedad de campo:

```
función de exportación PostFilter({ campo }) {
    devolver (
        <div>
            <label htmlFor={'filtro-${campo}'}>{campo}: </label> <entrada
                tipo='texto'
                nombre={'filtro-${campo}' id={'filtro-$
                {campo}'} /> </div>

    )
}

PostFilter.propTypes = {
    campo: PropTypes.string.isRequired,
}
```

A continuación, vamos a definir el componente PostSorting.

4. Cree un nuevo archivo src/components/PostSorting.jsx.

5. En este archivo, creamos una entrada de selección para seleccionar el campo por el que se ordenará. También creamos otra entrada de selección para seleccionar el orden de clasificación:

```
importar PropTypes desde 'prop-types'

función de exportación PostSorting({ campos = [] }) {
    devolver (
```

```
<div>
  <label htmlFor='sortBy'>Ordenar por: </label> <select
    name='sortBy' id='sortBy'> {fields.map((campo) =>
      ( <option key={campo} value={campo}
        > {campo} </option> ))} </select> {' / '} <label
      htmlFor='sortOrder'>Orden de clasificación: </label>
      <seleccionar nombre='sortOrder' id='sortOrder'>
        <opción valor='ascendente'>ascendente</opción> <opción
          valor='descendente'>descendente</opción> </seleccionar> </div>

    )
}

PostSorting.propTypes = {
  campos: PropTypes.arrayOf(PropTypes.string).isRequired,
}
```

Hemos definido correctamente los componentes de la interfaz de usuario para filtrar y ordenar las publicaciones. En el siguiente paso, crearemos un componente PostList para combinar el filtrado y la ordenación con una lista de publicaciones.

El componente PostList

Tras implementar los demás componentes relacionados con las publicaciones, podemos implementar la parte más importante de nuestra aplicación de blog: el feed de entradas. Por ahora, el feed simplemente mostrará una lista de entradas.

Comencemos a implementar el componente PostList ahora:

1. Cree un nuevo archivo src/components/PostList.jsx.
2. Primero, importamos Fragment, PropTypes y el componente Post:

```
importar { Fragmento } de 'react' importar
PropTypes de 'prop-types' importar { Publicación }
de './Post.jsx'
```

3. \$en, definimos el componente de función PostList, aceptando un array de publicaciones como propiedad. Si no se define la propiedad publicaciones, se establece como un array vacío, por defecto:

```
función de exportación PostList({ posts = [] }) {
```

4. A continuación, renderizamos todas las publicaciones utilizando la función .map y la sintaxis spread:

```
devolver (
  <div>
    {posts.map((post) => ( <Post {...post}
      clave={post._id} /> ))}
  </div>
)
```

Devolvemos el componente <Post> para cada publicación y pasamos todas las claves del objeto de publicación al componente como propiedades. Esto se logra mediante la sintaxis spread, que tiene el mismo efecto que listar manualmente todas las claves del objeto como propiedades, de la siguiente manera:

```
<Publicación
  título={post.title}
  autor={post.author}
  contenido={post.content} />
```

Nota

Si renderizamos una lista de elementos, debemos asignar a cada elemento una propiedad clave única. React utiliza esta propiedad clave para calcular eficientemente la diferencia entre dos listas cuando los datos cambian.

Usamos la función map, que aplica una función a todos los elementos de un array. \$is es similar a usar un bucle for y almacenar todos los resultados, pero es más conciso, declarativo y fácil de leer. Como alternativa, podríamos hacer lo siguiente en lugar de usar la función map:

```
dejar renderedPosts = []
sea índice = 0
para (dejar publicación de publicaciones) {
  renderedPosts.push(<Post {...post} clave={post._id} />) índice++
}

devolver (
  <div>
    {publicaciones
    renderizadas} </div>
)
```

Sin embargo, no se recomienda utilizar este estilo con React.

5. También necesitamos definir los tipos de propiedad. Aquí, podemos usar los tipos de propiedad del componente Post, envolviéndolo dentro de la función `PropTypes.shape()`, que define un tipo de propiedad de objeto:

```
PostList.propTypes = { publicaciones:  
  PropTypes.arrayOf(PropTypes.shape(Post.propTypes)).es obligatorio, }
```

6. En la maqueta, tenemos una línea horizontal después de cada entrada del blog. Podemos implementar esto sin un elemento contenedor `<div>` adicional, usando Fragment, como se indica a continuación:

```
{posts.map((post) =>  
  <Fragmento clave={post._id}> <Publicación  
  {...publicación} /> <hr /> </  
  
Fragmento> ))}
```

Nota

La propiedad clave `$e` siempre debe agregarse al elemento principal superior que se renderiza dentro de la función `map`. En este caso, tuvimos que mover la propiedad clave del componente `Post` a `Fragment`.

7. Nuevamente, probamos nuestro componente editando el archivo `src/App.jsx`:

```
importar { PostList } desde './components/PostList.jsx'  
  
constantes publicaciones =  
[ {  
  Título: 'Proyectos React Full-Stack', Contenido:  
  "¡Convirtámonos en desarrolladores full-stack!", Autor: 'Daniel Bug!', }, { Título:  
  '¡Hola React!', },  
  
]  
  
función de exportación App() {  
  devolver <PostList posts={posts} />  
}
```

Ahora podemos ver que nuestra aplicación enumera todas las publicaciones que definimos en la matriz de publicaciones.

Como puedes ver, listar varias publicaciones mediante el componente PostList funciona correctamente. Ahora podemos pasar a configurar la aplicación.

Armando la aplicación

Luego de implementar todos los componentes, ahora tenemos que juntar todo en el componente App. ¡Entonces habremos reproducido exitosamente la maqueta!

Comencemos modificando el componente App y armando nuestra aplicación de blog:

1. Abra src/App.jsx y agregue importaciones para CreatePost, PostFilter y

Componentes de PostSorting:

```
importar { PostList } desde './components/PostList.jsx' importar { CreatePost } desde './
components/CreatePost.jsx' importar { PostFilter } desde './components/PostFilter.jsx' importar
{ PostSorting } desde './components/PostSorting.jsx'
```

2. Ajuste el componente de la aplicación para que contenga todos los componentes:

```
función de exportación App() { return
  ( <div
    style={{ padding: 8 }}>
      <CreatePost /> <br />
      <hr />

      Filtrar por:
      <Campo PostFilter='autor' /> <br />

      <Campos de clasificación posterior={['createdAt', 'updatedAt']} /> <hr />

      <PostList publicaciones={publicaciones} /> <
      div>
    )
  }
}
```

3. Después de guardar el archivo, el navegador debería actualizarse automáticamente y ahora podremos ver la interfaz de usuario completa:



Figura 4.3 – Implementación completa de nuestra aplicación de blog estático, según la maqueta

Como podemos ver, todos los componentes estáticos que definimos anteriormente se renderizan juntos en un solo componente de la aplicación. Nuestra aplicación ahora luce como una maqueta. A continuación, podemos integrar nuestros componentes con el servicio backend.

Integración del servicio backend mediante TanStack Query

Tras crear todos los componentes de la interfaz de usuario, podemos integrarlos con el backend que creamos en el capítulo anterior. Para ello, usaremos TanStack Query (anteriormente React Query), una biblioteca de obtención de datos que también nos ayuda con el almacenamiento en caché, la sincronización y la actualización de datos desde un backend.

TanStack Query se centra específicamente en la gestión del estado de los datos obtenidos (estado del servidor). Aunque otras bibliotecas de gestión de estado también pueden gestionar el estado del servidor, se especializan en la gestión del estado del cliente . El estado del servidor presenta diferencias significativas con el estado del cliente, como las siguientes:

- Permanecer de forma remota en una ubicación que el cliente no controla directamente
- Requerir API asíncronas para obtener y actualizar el estado

- Tener que lidiar con la propiedad compartida, lo que significa que otras personas pueden cambiar el estado sin tu conocimiento
- El estado se vuelve obsoleto ("desactualizado") en algún momento cuando el servidor u otras personas lo cambian

Estos desafíos con el estado del servidor resultan en problemas como tener que almacenar en caché, deduplicar múltiples solicitudes, actualizar el estado "desactualizado" en segundo plano, etc.

TanStack Query ofrece soluciones a estos problemas de forma inmediata, simplificando así la gestión del estado del servidor. Siempre puede combinarse con otras bibliotecas de gestión de estado que también se centran en el estado del cliente. Sin embargo, para casos donde el estado del cliente básicamente refleja el estado del servidor, TanStack Query por sí sola puede ser una solución de gestión de estado suficiente.

Nota

¡La razón por la que React Query pasó a llamarse TanStack Query es que la biblioteca ahora también admite otros marcos, como Solid, Vue y Svelte!

Ahora que sabes por qué y cómo TanStack Query puede ayudarnos a integrar nuestro frontend con el backend, ¡comencemos a usarlo!

Configuración de la consulta TanStack para React

Para configurar TanStack Query, primero tenemos que instalar la dependencia y configurar un cliente de consulta. El cliente de consulta se proporciona a React a través de un contexto y almacenará información sobre solicitudes activas, resultados en caché, cuándo volver a buscar datos periódicamente y todo lo necesario para que TanStack Query funcione.

Comencemos a configurarlo ahora:

1. Abra una nueva terminal (¡no salga de Vite!) e instale @tanstack/react-query dependencia ejecutando el siguiente comando en la raíz de nuestro proyecto:

```
$ npm install @tanstack/react-query@5.12.2
```

Ahora vamos a mover nuestro componente de aplicación actual a un nuevo componente de blog, ya que vamos a utilizar el componente de aplicación para configurar bibliotecas y contextos.

2. Cambie el nombre del archivo src/App.jsx a src/Blog.jsx.

No actualice las importaciones todavía. Si VS Code le solicita que actualice las importaciones, haga clic en No.

3. Ahora, en src/Blog.jsx, cambie el nombre de la función de Aplicación a Blog:

```
función de exportación Blog() {
```

4. Cree un nuevo archivo src/App.jsx. En este archivo, importe QueryClient y QueryClientProvider desde TanStack React Query:

```
importar { QueryClient, QueryClientProvider } desde '@tanstack/react-query'
```

5. Además, importe el componente Blog:

```
importar { Blog } desde './Blog.jsx'
```

6. Ahora, crea un nuevo cliente de consulta:

```
constante queryClient = nuevo QueryClient()
```

7. Defina el componente App y renderice el componente Blog envuelto dentro de QueryClientProvider:

```
función de exportación App() { return (  
    <QueryClientProvider cliente={queryClient}>  
        <Blog />  
    </QueryClientProvider>  
)  
}
```

¡Eso es todo lo que hay que hacer para configurar TanStack Query! Ahora podemos usarlo dentro de nuestro componente Blog (y sus componentes secundarios).

Obteniendo publicaciones del blog

Lo primero que debemos hacer es obtener la lista de entradas del blog desde nuestro backend. Implementémoslo ahora:

1. En primer lugar, en la segunda ventana de Terminal abierta (no donde se está ejecutando Vite), ejecute el backend servidor (¡no abandones Vite!), de la siguiente manera:

```
$ cd backend/  
$ npm start
```

Si recibe un error, asegúrese de que Docker y MongoDB estén ejecutándose correctamente.

Consejo: si desea desarrollar el backend y el frontend al mismo tiempo, puede iniciar el backend usando npm run dev para asegurarse de que se recargue en caliente cuando cambie el código.

2. Cree un archivo .env en la raíz del proyecto e ingrese en él el siguiente contenido:

```
VITE_BACKEND_URL="http://localhost:3001/api/v1"
```

Vite es compatible con dotenv de fábrica. Todas las variables de entorno accesibles desde el frontend deben tener el prefijo VITE_. Aquí, configuramos una variable de entorno para que apunte a nuestro servidor backend.

3. Cree un nuevo archivo `src/api/posts.js` . En este archivo, definiremos una función para recuperar publicaciones, que acepta los parámetros de consulta del punto final `/posts` como argumento. Estos parámetros de consulta se utilizan para filtrar por autor y etiqueta, y para definir la ordenación mediante `sortBy` y `sortOrder` .

```
exportar const getPosts = async (queryParams) => {
```

4. Recuerda que podemos usar la función fetch para realizar una solicitud a un servidor. Necesitamos pasarle la variable de entorno y agregar el punto final /posts. Después de la ruta, agregamos los parámetros de consulta, precedidos por el símbolo ?.

```
const res = await fetch(  
  `${import.meta.env.VITE_BACKEND_URL}/publicaciones?` +
```

5. Ahora necesitamos usar la clase URLSearchParams para convertir un objeto en parámetros de consulta. La clase \$at escapará automáticamente la entrada y la convertirá en parámetros de consulta válidos:

```
nuevo URLSearchParams(queryParams),
```

6. Como hicimos antes en el navegador, necesitamos analizar la respuesta como JSON:

```
)  
devolver esperar res.json()  
}
```

7. Edite src/Blog.jsx y elimine la matriz de publicaciones de muestra:

```
constantes publicaciones = [  
  {  
    Título: 'Proyectos React Full-Stack'  
    Contenido: "¡Convirtámonos en desarrolladores full-stack!"  
    autor: 'Daniel Bug!',  
  },  
  { título: '¡Hola React!' },  
]
```

8. Además, importe la función useQuery desde @tanstack/react-query y getPosts función de nuestra carpeta api en el archivo src/Blog.jsx:

```
importar { useQuery } desde '@tanstack/react-query'  
importar { PostList } desde './components/PostList.jsx'
```

```
importar { CreatePost } desde './components/CreatePost.jsx' importar { PostFilter } desde './components/PostFilter.jsx' importar { PostSorting } desde './components/PostSorting.jsx'  
importar { getPosts } desde './api/posts.js'
```

9. Dentro del componente Blog, defina un gancho useQuery:

```
función de exportación Blog() { const  
  postsQuery = useQuery({ queryKey: ['posts'],  
    queryFn: () => getPosts(), })
```

La clave de consulta \$e es muy importante en TanStack Query, ya que se utiliza para identificar de forma única una solicitud, entre otras cosas, para fines de almacenamiento en caché. Asegúrese siempre de usar claves de consulta únicas. De lo contrario, es posible que las solicitudes no se activen correctamente.

Para la opción queryFn, simplemente llamamos a la función getPosts, sin parámetros de consulta por ahora.

10. Después del gancho useQuery, obtenemos las publicaciones de nuestra consulta y volvemos a una matriz vacía

Si las publicaciones aún no están cargadas:

```
constante posts = postsQuery.data ?? []
```

11. ¡Revisa tu navegador y verás que las publicaciones ahora se cargan desde nuestro backend!

¡Ahora que hemos obtenido con éxito las publicaciones del blog, pongamos en funcionamiento los filtros y la clasificación!

Implementación de filtros y ordenamiento

Para implementar filtros y ordenamiento, necesitamos gestionar un estado local y pasarlo como parámetro de consulta a postsQuery. Hagámolo ahora:

1. Comenzamos editando el archivo src/Blog.jsx e importando el gancho useState desde React:

```
importar { useState } desde 'react'
```

2. Luego agregamos ganchos de estado para el filtro de autor y las opciones de clasificación dentro del componente Blog, antes del gancho useQuery:

```
const [autor, establecerAutor] = useState("") const [sortBy,  
establecerSortBy] = useState('createdAt') const [sortOrder, establecerSortOrder]  
= useState('descending')
```

3. \$en, ajustamos queryKey para que contenga los parámetros de consulta (de modo que, al cambiar un parámetro de consulta, TanStack Query vuelve a obtener la consulta a menos que la solicitud ya esté en caché). También ajustamos queryFn para llamar a getPosts con los parámetros de consulta relevantes:

```
const postsQuery = useQuery({ claveDeConsulta:  
  ['publicaciones', { autor, sortBy, sortOrder }], funciónDeConsulta: () => obtenerPublicaciones({ autor,  
  sortBy, sortOrder }), })
```

4. Ahora pase los valores y los controladores onChange relevantes a los componentes de filtrado y clasificación:

```
<Filtro de publicación  
campo='autor'  
valor={autor}  
onChange={(valor) => setAuthor(valor)} /> <br />
```

```
<Ordenación de publicaciones  
campos={['createdAt', 'updatedAt']} valor={sortBy}  
onChange={(valor) =>  
  setSortBy(valor)} orderValue={sortOrder} onOrderChange={(orderValue)  
=> setSortOrder(orderValue)} />
```

Nota

Para simplificar, por ahora solo utilizamos ganchos de estado. Una solución o contexto de gestión de estados podría simplificar considerablemente la gestión de filtros y la ordenación, especialmente en aplicaciones de mayor tamaño. Sin embargo, para nuestra pequeña aplicación de blog, no es necesario utilizar ganchos de estado, ya que nos centramos principalmente en la integración del backend y el frontend.

5. Ahora, edite src/components/PostFilter.jsx y agregue el valor y las propiedades onChange:

```
función de exportación PostFilter({ campo, valor, onChange }) {  
  devolver (  
    <div>  
      <label htmlFor={'filtro-${campo}'}>{campo}: </label> <input type='text' name={`filtro-${campo}`}  
      id={`filtro-$  
        {campo}'}  
      value={valor}
```

```
        onChange={(e) => onChange(e.objetivo.valor)} /> </div>

    )
}

PostFilter.propTypes = {
    campo: PropTypes.string.isRequired, valor:
    PropTypes.string.isRequired, onChange:
    PropTypes.func.isRequired,
}
```

6. También hacemos lo mismo para src/components/PostSorting.jsx:

```
función de exportación PostSorting({ campos = [],

    valor,
    onChange,
    valor del pedido,
    onOrderChange, })

{ devolver (
    <div>
        <label htmlFor='sortBy'>Ordenar por: </label>
        <seleccionar
            nombre='sortBy'
            id='sortBy'
            valor={valor}
            onChange={(e) => onChange(e.target.value)}
        >
            {campos.map((campo) => ( <opción
                clave={campo} valor={campo}> {campo} </opción> ))) </
            select> ' /
        } <label

        htmlFor='sortOrder'>Orden de clasificación: </label>
        <seleccionar
            nombre='ordenamiento'
            id='orden de clasificación'
            valor={valorDeOrden}
        }
```

```
        onChange={(e) => onOrderChange(e.objetivo.valor)}
      >
        <option value={'ascending'}>ascendente</option>
        <option value={'descending'}>descendente</option>
      </seleccionar>
    </div>
  )
}

PostSorting.propTypes = {
  campos: PropTypes.arrayOf(PropTypes.string).isRequired,
  valor: PropTypes.string.isRequired,
  onChange: PropTypes.func.isRequired,
  orderValue: PropTypes.string.isRequired,
  onOrderChange: PropTypes.func.isRequired,
}
```

7. En tu navegador, introduce a Daniel Bugl como autor. Deberías ver que TanStack Query recupera las publicaciones del backend mientras escribes. Una vez que encuentre una coincidencia, el backend mostrará todas las publicaciones de ese autor.
8. Después de probarlo, asegúrese de limpiar el filtro nuevamente, para que las publicaciones recién creadas no se filtren por el autor más adelante.

Consejo

Si no desea realizar tantas solicitudes al backend, asegúrese de utilizar un gancho de estado antirrebote, como `useDebounce`, y luego pase solo el valor antirrebote al parámetro de consulta. Si estás interesado en obtener más conocimientos sobre el gancho `useDebounce` y otros ganchos útiles, te recomiendo que consultes mi libro titulado *Learn React Hooks*.

La aplicación se ahora debería verse de la siguiente manera, con las publicaciones filtradas por el autor ingresado en el campo y ordenadas por el campo seleccionado, en el orden seleccionado:

The screenshot shows a web application interface. At the top, there's a browser header with the title 'Vite + React' and the URL 'localhost:5173'. Below the header is a form with fields for 'Title' and 'Author', both represented by input boxes. There's also a large empty box for content and a 'Create' button. Underneath the form are filtering and sorting options: 'Filter by: author: Daniel Bugl' and 'Sort By: updatedAt / Sort Order: ascending'. The main content area displays two posts. The first post is 'Hello Mongoose!' and the second is 'Hello again, Mongoose!'. Each post has a footer line 'Written by Daniel Bugl'.

Figura 4.4 – Nuestra primera aplicación full-stack: ¡un frontend que obtiene publicaciones de un backend!

Ahora que la clasificación y el filtrado funcionan correctamente, aprendamos sobre las mutaciones, que nos permiten realizar solicitudes al servidor que cambian el estado del backend (por ejemplo, insertar o actualizar entradas en la base de datos).

Creando nuevas publicaciones

Ahora vamos a implementar una función para crear publicaciones. Para ello, necesitamos usar la función `useMutation` Hook de TanStack Query. Si bien las consultas son idempotentes (es decir, invocarlas varias veces no debería afectar el resultado), las mutaciones se utilizan para crear, actualizar o eliminar datos o realizar operaciones en el servidor. Comencemos a usar mutaciones para crear nuevas entradas:

1. Edite `src/api/posts.js` y defina una nueva función `createPost`, que acepte una publicación.
objeto como argumento:

```
export const createPost = async (publicación) => {
```

2. También realizamos una solicitud al punto final /posts, como hicimos para getPosts:

```
const res = await fetch(`${import.meta.env.VITE_BACKEND_URL}/publicaciones`, {
```

3. Sin embargo, ahora también establecemos un método en una solicitud POST, pasamos un encabezado para indicarle al backend que enviaremos un cuerpo JSON y luego enviamos nuestro objeto de publicación como una cadena JSON:

```
método: 'POST',
encabezados: { 'Content-Type': 'application/json' }, cuerpo: JSON.stringify(post),
```

4. Al igual que con getPosts, también necesitamos analizar la respuesta como JSON:

```
})
devolver esperar res.json()
}
```

Después de definir la función API createPost, usémosla en el componente CreatePost creando un nuevo gancho de mutación allí.

5. Edite src/components/CreatePost.jsx e importe el gancho useMutation de @tanstack/react-query, el gancho useState de React y nuestra función API createPost:

```
importar { useMutation } desde '@tanstack/react-query' importar { useState } desde
'react' importar { createPost } desde './api/posts.js'
```

6. Dentro del componente CreatePost, defina los ganchos de estado para el título, el autor y el contenido:

```
const [título, establecerTítulo] = useState("") const [autor,
establecerAutor] = useState("") const [contenido, establecerContenido]
= useState("")
```

7. Ahora, definamos un gancho de mutación. Aquí, llamaremos a nuestra función createPost:

```
const createPostMutation = useMutation({ mutationFn: () =>
createPost({ título, autor, contenido }), })
```

8. A continuación, vamos a definir una función handleSubmit, que evitará la acción de envío predeterminada (que actualiza la página) y, en su lugar, llamará a .mutate() para ejecutar la mutación:

```
constante handleSubmit = (e) =>
{ e.preventDefault()
createPostMutation.mutate()
}
```

9. Agregamos el controlador onSubmit a nuestro formulario:

```
<formulario onSubmit={manejarEnvío}>
```

10. También agregamos las propiedades value y onChange a nuestros campos, como hicimos antes para la clasificación y filtros:

```
<div>
  <label htmlFor='create-title'>Título: </label> <input
    tipo='texto'
    nombre='crear-título'
    id='crear-título'
    valor={título}
    onChange={(e) => setTitle(e.objetivo.valor)} /> </div> <br />

<div>
  <label htmlFor='create-author'>Autor: </label> <input type='text'
    nombre='crear-autor'
    id='crear-autor'
    valor={autor}
    onChange={(e) => setAuthor(e.target.value)} /> </div> <br />

<área de texto
  valor={contenido}
  onChange={(e) => setContents(e.target.value)} />
```

11. Para el botón de envío, nos aseguramos de que diga "Creando..." en lugar de "Crear" mientras esperamos a que finalice la mutación. También desactivamos el botón si no se ha definido ningún título (ya que es obligatorio) o si la mutación está pendiente.

```
<br />
<br />
<entrada
  tipo='enviar'
  valor={createPostMutation.isPending ? 'Creando...' : 'Crear'} deshabilitado={!título ||

createPostMutation.isPending} />
```

12. Por último, agregamos un mensaje debajo del botón de envío, que se mostrará si la mutación es exitosa:

```
{createPostMutation.isSuccess ? (
  <>
  <br />
  ¡Publicación creada exitosamente!
</>
) : nulo}
</form>
```

Nota

Además de isPending e isSuccess, las mutaciones también devuelven estados isIdle (cuando la mutación está inactiva o en un estado nuevo/reiniciado) e isError. También se puede acceder a los mismos estados desde las consultas, por ejemplo, para mostrar una animación de carga mientras se obtienen las publicaciones.

13. Ahora podemos intentar agregar una nueva publicación, y parece funcionar bien, pero la lista de publicaciones no se actualiza.

¡Automáticamente, solo después de una actualización!

El problema es que la clave de consulta no cambió, por lo que TanStack Query no actualiza la lista de publicaciones. Sin embargo, también queremos actualizar la lista al crear una nueva publicación. Corrijamos eso ahora.

Consultas invalidantes

Para garantizar que la lista de publicaciones se actualice después de crear una nueva, debemos invalidar la consulta. Podemos usar el cliente de consultas para ello. Hagámoslo ahora:

1. Edite src/components/CreatePost.jsx e importe el gancho useQueryClient:

```
importar { useMutation, useQueryClient } desde '@tanstack/react-query'
```

2. Utilice el cliente de consultas para invalidar todas las consultas que comienzan con la clave de consulta "posts". \$is funcionará con cualquier parámetro de consulta para la solicitud getPosts, ya que coincide con todas las consultas que comienzan con "posts" en la matriz:

```
constante queryClient = useQueryClient()
constante createPostMutation = usarMutation({
  mutationFn: () => createPost({ título, autor, contenidos }),
  onSuccess: () => queryClient.invalidateQueries(['posts']),
})
```

Intenta crear una nueva publicación y verás que ahora funciona, ¡incluso con filtros y ordenamiento activos! Como podemos ver, TanStack Query es ideal para gestionar el estado del servidor con facilidad.

Resumen

En este capítulo, aprendimos a crear un frontend de React e integrarlo con nuestro backend mediante TanStack Query. Hemos cubierto las principales funciones de nuestro backend: listar entradas con ordenación, crear entradas y filtrar por autor. El manejo de etiquetas, así como la eliminación y edición de entradas, es similar a las funciones ya explicadas y se presenta a modo de ejercicio.

En el próximo capítulo, Capítulo 5, Implementación de la aplicación con Docker y CI/CD, implementaremos nuestra aplicación con Docker y configuraremos pipelines de CI/CD para automatizar la implementación de nuestra aplicación.

5

Implementación de la aplicación con Docker y CI/CD

Ahora que hemos desarrollado con éxito nuestra primera aplicación full-stack con un servicio backend y un frontend, vamos a empaquetar nuestra aplicación en imágenes Docker y aprender a implementarlas utilizando los principios de integración continua (CI) y entrega continua (CD). Ya aprendimos a iniciar contenedores Docker en el Capítulo 2, "Conociendo Node.js y MongoDB". En este capítulo, aprenderemos a crear nuestras propias imágenes Docker para instanciar contenedores. Desplegaremos manualmente nuestra aplicación en un proveedor de nube. Finalmente, configuraremos CI/CD para automatizar el despliegue de nuestra aplicación. Al final de este capítulo, habremos desplegado con éxito nuestra primera aplicación full-stack MongoDB Express React Node.js (MERN) y la habremos preparado para futuras implementaciones automatizadas.

En este capítulo cubriremos los siguientes temas principales:

- Creación de imágenes de Docker
- Implementación de nuestra aplicación full-stack en la nube
- Configuración de CI para automatizar las pruebas
- Configuración del CD para automatizar la implementación

Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones que aparecen en esos capítulos son las que se utilizan en este libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que ciertos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas en los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/capítulo5>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/aQplfCQGWew>

Creación de imágenes de Docker

En el Capítulo 2, "Conociendo Node.js y MongoDB", aprendimos que en la plataforma Docker usamos imágenes de Docker para crear contenedores que luego pueden ejecutar servicios. Ya aprendimos a usar la imagen de Mongo existente para crear un contenedor para nuestro servicio de base de datos. En esta sección, aprenderemos a crear nuestra propia imagen para instanciar un contenedor. Para ello, primero necesitamos crear un Dockerfile, que contiene todas las instrucciones necesarias para construir la imagen de Docker. Primero, crearemos una imagen de Docker para nuestro servicio backend y ejecutaremos un contenedor desde ella. \$en, haremos lo mismo para nuestro frontend. Finalmente, crearemos un Docker Compose para iniciar nuestros servicios de base de datos y backend junto con nuestro frontend.

Creando el Dockerfile de backend

Un Dockerfile le indica a Docker paso a paso cómo construir la imagen. Cada línea del archivo es una instrucción que le indica a Docker qué hacer. El formato de un Dockerfile es el siguiente:

```
# comentario  
argumentos de INSTRUCCIÓN
```

Cada Dockerfile debe comenzar con una instrucción FROM, que especifica en qué imagen se basará la nueva imagen. Puedes extender tu imagen a partir de imágenes existentes, como Ubuntu o Node.

Comencemos creando el Dockerfile para nuestro servicio backend:

1. Copie la carpeta ch4 a una nueva carpeta ch5, de la siguiente manera:

```
$ cp -R ch4 ch5
```

2. Cree un nuevo archivo backend/Dockerfile dentro de la carpeta ch5.
3. En este archivo, primero definimos una imagen base para nuestra imagen, que será la versión 20 de la imagen del nodo:

```
DESDE el nodo:20
```

Esta imagen la proporciona Docker Hub, similar a las imágenes de Ubuntu y Mongo a partir de las cuales creamos contenedores anteriormente.

Nota

Tenga cuidado de utilizar únicamente imágenes sociales e imágenes creadas por autores confiables. La imagen de nodo \$e, por ejemplo, es mantenida socialmente por el equipo Node.js.

4. \$en, establecemos el directorio de trabajo, que es donde se colocarán todos los archivos de nuestro servicio dentro la imagen:

```
WORKDIR /aplicación
```

La instrucción \$e WORKDIR es similar a usar cd en la terminal. Cambia el directorio de trabajo para que no tengamos que anteponer la ruta completa a todos los comandos siguientes. Docker crea la carpeta automáticamente si aún no existe.

5. A continuación, copiamos los archivos package.json y package-lock.json de nuestro proyecto a el directorio de trabajo:

```
COPIA paquete.json paquete-lock.json ./
```

La instrucción \$e COPY copia archivos desde el sistema de archivos local a la imagen de Docker (en relación con el directorio de trabajo local). Se pueden especificar varios archivos, y el último argumento de la instrucción es el destino (en este caso, el directorio de trabajo actual de la imagen).

Se necesita el archivo \$e package-lock.json para garantizar que la imagen de Docker contenga las mismas versiones de los paquetes npm que nuestra compilación local.

6. Ahora, ejecutamos npm install para instalar todas las dependencias en la imagen:

```
EJECUTAR npm install
```

La instrucción \$e RUN ejecuta un comando en el directorio de trabajo de la imagen.

7. \$en, copiamos el resto de nuestra aplicación desde el sistema de archivos local a la imagen de Docker:

```
COPIAR . .
```

Nota

¿Te preguntas por qué inicialmente solo copiamos package.json y package-lock? JSON? Las imágenes de Docker se construyen capa por capa. Cada instrucción forma una capa de la imagen. Si algo cambia, solo se reconstruyen las capas posteriores al cambio. Por lo tanto, en nuestro caso, si cambia algún código, solo se vuelve a ejecutar esta última instrucción COPY al reconstruir la imagen de Docker. Sólo si cambian las dependencias se vuelven a ejecutar la otra instrucción COPY y npm install. El uso de este orden de instrucciones reduce enormemente el tiempo necesario para reconstruir la imagen.

8. Finalmente, ejecutamos nuestra aplicación:

```
CMD ["npm", "inicio"]
```

La instrucción '\$e CMD' no se ejecuta durante la creación de la imagen. En su lugar, almacena información en los metadatos de la imagen, indicando a Docker qué comando ejecutar al instanciar un contenedor a partir de ella. En nuestro caso, el contenedor ejecutará 'npm start' al usar nuestra imagen.

Nota

Quizás haya notado que pasamos una matriz JSON a la instrucción CMD en lugar de simplemente escribir CMD npm start. La versión de la matriz JSON \$e se llama formato exec y, si el primer argumento es un ejecutable, ejecutará el comando directamente sin invocar un shell. La versión \$e sin la matriz JSON se llama formato shell y ejecutará el comando con un shell, prefijando /bin/sh -c. Ejecutar un comando sin un shell tiene la ventaja de permitir que la aplicación reciba correctamente señales, como SIGTERM o SIGKILL, al finalizar la aplicación. Como alternativa, se puede usar la instrucción ENTRYPOINT para especificar qué ejecutable se debe usar para ejecutar un comando (el valor predeterminado es /bin/sh -c).

En algunos casos, es posible que incluso desees ejecutar el script directamente usando CMD ["node", "src/index.js"], para que el script pueda recibir correctamente todas las señales. Sin embargo, esto requeriría implementar la señal SIGINT en nuestro servidor backend para permitir cerrar el contenedor mediante Ctrl + C. Por lo tanto, para simplificar, simplemente usamos npm start.

Después de crear nuestro Dockerfile, también debemos crear un archivo .dockerignore para asegurarnos de que no se copien archivos innecesarios en nuestra imagen.

Creando un archivo .dockerignore

El comando '\$e COPY', que copia todos los archivos, también copia la carpeta 'node_modules' y otros archivos, como el archivo '.env', que no queremos incluir en nuestra imagen. Para evitar que ciertos archivos se copien en nuestra imagen de Docker, necesitamos crear un archivo '.dockerignore'. Hagámoslo ahora:

1. Cree un nuevo archivo backend/.dockerignore.
2. Ábralo e ingrese el siguiente contenido para ignorar la carpeta node_modules y todos los archivos .env:

```
módulos_de_nodo  
.env*
```

Ahora que hemos definido un archivo .dockerignore, las instrucciones COPY ignorarán estas carpetas y archivos.

Construyamos la imagen de Docker.

Construyendo la imagen de Docker

Después de crear con éxito el Dockerfile de backend y un archivo .dockerignore para evitar que ciertos archivos y carpetas se agreguen a nuestra imagen de Docker, ahora podemos comenzar a construir nuestra imagen de Docker:

1. Abra una terminal.
2. Ejecute el siguiente comando para crear la imagen de Docker:

```
$ docker image build -t blog-backend backend/
```

Especificamos blog-backend como el nombre de nuestra imagen y backend/ como el directorio de trabajo.

Tras ejecutar el comando, Docker comenzará a leer los archivos Dockerfile y .dockerignore. \$en, descargará la imagen del nodo y ejecutará nuestras instrucciones una por una. Finalmente, exportará todas las capas y metadatos a nuestra imagen de Docker.

La siguiente captura de pantalla muestra el resultado de la creación de una imagen de Docker:

```
→ ~/D/F/ch5 ↵ main± > docker image build -t blog-backend backend/
[+] Building 120.8s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 159B
=> [internal] load .dockerignore
=> => transferring context: 59B
=> [internal] load metadata for docker.io/library/node:latest
=> [auth] library/node:pull token for https://index.docker.io/v1/repositories/library/node/tags/latest
=> [1/5] FROM docker.io/library/node:latest@sha256:db2672e3c200b85e0b813cd294fac16764711d7a66b41315e6261f2231f2331
=> => resolve docker.io/library/node:latest@sha256:db2672e3c200b85e0b813cd294fac16764711d7a66b41315e6261f2231f2331
=> => sha256:05962e2d2a19f217fc55ed6b61aae16070c7c220c690aee1b60564e03bf377 7.54kB / 7.54kB
=> => sha256:df2021ddb7d686bdbb125598b2a6163d63035f080356b3014595f354ea0b40d6 49.61MB / 49.61MB
=> => sha256:8d647f1dd7e741209a8a75083ccc889e39cb3e94c17f45441ea96e1a679d971 23.58MB / 23.58MB
=> => sha256:db2672e3c200b85e0b813cd294fac16764711d7a66b41315e6261f2231f2331 1.21kB / 1.21kB
=> => sha256:63d8ba1a9663e2daa5fa8f49bce6b048cc5e31565f9786be1ca7d8be3f911ec5 2.00kB / 2.00kB
=> => sha256:5cd9a70365f741a6b9f7a4e32cd7d4a29ac73da0b78ca0a83e937f285fd5 63.99MB / 63.99MB
=> => sha256:95089c600b361807380090316c250b0b8ea4fa2175b11ac8f49bb7581c61125 202.45MB / 202.45MB
=> => sha256:00e0658e345c7433de5051c897a98b802ae91e6375952104fa0fbf7d969ab581 3.37kB / 3.37kB
=> => sha256:95f3e8ec3833596867c7796e2860de73800b5599a63bec578722cd6ebe857b3f 49.00MB / 49.00MB
=> => extracting sha256:df2021ddb7d686bdbb125598b2a6163d63035f080356b3014595f354ea0b40d6
=> => sha256:335e1806199ff1916aaecfe3a5571fc0012f56e3abfdbfb0df6e8a859eef 2.23MB / 2.23MB
=> => sha256:764c4860e9c44928a589ad3a023ddd3bcff4b1bddff7c8377393e57fd47db9 452B / 452B
=> => extracting sha256:8d647f1dd7e741209a8a75083ccc889e39cb3e94c17f45441ea96e1a679d971
=> => extracting sha256:5cd9a70365f741a6b9f7a4e32cd7d4aa29ac73da0b78ca0a83e937f285fd5
=> => extracting sha256:95089c600b361807380090316c250b0b8ea4fa2175b11ac8f49bb7581c61125
=> => extracting sha256:00e0658e345c7433de5051c897a98b802ae91e6375952104fa0fbf7d969ab581
=> => extracting sha256:95f3e8ec3833596867c7796e2860de73800b5599a63bec578722cd6ebe857b3f
=> => extracting sha256:335e1806199ff1916aaecfe3a5571fc0012f56e3abfdbfb0df6e8a859eef
=> => extracting sha256:764c4860e9c44928a589ad3af023ddd3bcff4b1bddff7c8377393e57fd47db9
=> [internal] load build context
=> => transferring context: 297.81kB
=> [2/5] WORKDIR /app
=> [3/5] COPY package.json package-lock.json .
=> [4/5] RUN npm install
=> [5/5] COPY .
=> => exporting to image
=> => exporting layers
=> => writing image sha256:df44d3116a1f3adf1445f2aed7fd220e45083da7388ce6ffa8699a7e80babaf0
=> => naming to docker.io/library/blog-backend
```

Figura 5.1 – La salida al crear una imagen de Docker

¡Ahora que hemos creado con éxito nuestra propia imagen, creemos y ejecutemos un contenedor basado en ella!

Creando y ejecutando un contenedor desde nuestra imagen

Ya creamos contenedores Docker basados en las imágenes de Ubuntu y Mongo en el Capítulo 2, "Conociendo Node.js y MongoDB".

Ahora, vamos a crear y ejecutar un contenedor desde nuestra propia imagen. Comencemos:

- Ejecute el siguiente comando para enumerar todas las imágenes disponibles:

```
$ imágenes docker
```

El comando \$is debe devolver la imagen del backend del blog que acabamos de crear y el mongo y las imágenes de Ubuntu que usamos anteriormente.

2. Asegúrese de que el contenedor dbserver con nuestra base de datos ya esté ejecutándose.

3. \$en, inicia un nuevo contenedor, de la siguiente manera:

```
$ docker run -it -e PUERTO=3001 -e URL_DE_BASE_DE_DATOS=mongodb://host.docker.internal:27017/blog -p 3001:3001 blog-backend
```

Analicemos los argumentos del comando docker run:

-ejecuta el contenedor en modo interactivo (-t para asignar una pseudo-terminal y -i para mantener abierto el flujo de entrada).

-e PORT=3001 establece la variable de entorno PORT dentro del contenedor en 3001.

-e DATABASE_URL=mongodb://host.docker.internal:27017/blog establece la variable de entorno DATABASE_URL. Aquí, reemplazamos localhost por host.

docker.internal, ya que el servicio MongoDB se ejecuta en un contenedor diferente en el host Docker (nuestra máquina).

-p 3001:3001 reenvía el puerto 3001 desde el interior del contenedor al puerto 3001 en el host (nuestra máquina).

blog-backend es el nombre de nuestra imagen.

4. El contenedor \$e blog-backend ya está en ejecución, lo cual es muy similar a ejecutar el backend directamente en nuestro host desde la Terminal. Vaya a <http://localhost:3001/api/v1/posts>

para verificar que esté funcionando correctamente como antes y devolviendo todas las publicaciones.

5. Mantenga el contenedor en funcionamiento por ahora.

Hemos empaquetado correctamente nuestro backend como una imagen Docker y hemos creado un contenedor a partir de ella. Ahora, hagamos lo mismo con nuestro frontend.

Creando el Dockerfile del frontend

Tras crear una imagen de Docker para el servicio backend, repetiremos el mismo proceso para crear una imagen para el frontend. Para ello, crearemos primero un Dockerfile, luego el archivo .dockerignore, compilaremos la imagen y, finalmente, ejecutaremos un contenedor. Ahora, comenzaremos con la creación del Dockerfile del frontend.

En el Dockerfile para nuestro frontend, vamos a utilizar dos imágenes:

- Una imagen de compilación para construir nuestro proyecto usando Vite (que será descartada, quedando solo la imagen de compilación) salida mantenida)
- Una imagen final, que servirá a nuestro sitio estático usando nginx

Hagamos el Dockerfile ahora:

1. Crea un nuevo Dockerfile en la raíz de nuestro proyecto.
2. En este archivo recién creado, primero usamos nuevamente la imagen del nodo, pero esta vez la etiquetamos como compilación.

Al hacerlo, se permiten compilaciones de varias etapas en Docker, lo que significa que podemos usar otra imagen base más adelante para nuestra imagen final:

```
DESDE el nodo:20 COMO compilación
```

3. Durante la compilación, también configuraremos la variable de entorno VITE_BACKEND_URL. En Docker, podemos usar la instrucción ARG para definir variables de entorno que solo son relevantes durante la compilación de la imagen:

```
ARG VITE_BACKEND_URL=http://localhost:3001/api/v1
```

Nota

Mientras que la instrucción ARG define una variable de entorno que puede modificarse durante la compilación mediante --build-arg %ag, la instrucción ENV establece la variable de entorno en un valor fijo, que persistirá al ejecutar un contenedor desde la imagen resultante. Por lo tanto, si queremos personalizar las variables de entorno durante la compilación, debemos usar la instrucción ARG. Sin embargo, si queremos personalizar las variables de entorno durante el tiempo de ejecución, ENV es más adecuado.

4. Establecemos el directorio de trabajo en /build para la etapa de compilación y luego repetimos las mismas instrucciones que definimos para el backend para instalar todas las dependencias necesarias y copiar los archivos necesarios:

```
WORKDIR /construir  
COPIA paquete.json .  
COPIA paquete-lock.json .  
EJECUTAR npm install  
COPIAR . .
```

5. Además, ejecutamos npm run build para crear una compilación estática de nuestra aplicación Vite:

```
EJECUTAR npm run build
```

6. Ahora, nuestra etapa de compilación está completa. Usamos la instrucción FROM nuevamente para crear el archivo final.

Etapa. Es hora de que nos basemos en la imagen nginx, que ejecuta un servidor web nginx:

```
DESDE nginx COMO final
```

7. Establecemos el directorio de trabajo para esta etapa en /var/www/html, que es la carpeta que nginx sirve archivos estáticos desde:

```
DIRECTORIO DE TRABAJO /usr/share/nginx/html
```

8. Por último, copiamos todo de la carpeta /build/dist (que es donde Vite pone el archivo compilado). archivos estáticos) desde la etapa de construcción hasta la etapa final:

```
COPIA --from=build /build/dist .
```

En este caso no se necesita una instrucción CMD, ya que la imagen nginx ya contiene una para ejecutar el servidor web correctamente.

Hemos creado con éxito un Dockerfile multietapa para nuestro frontend. Ahora, vamos a crear el archivo .dockerignore.

Creando el archivo .dockerignore para el frontend

También necesitamos crear un archivo .dockerignore para el frontend. Aquí, además de la carpeta node_modules/ y los archivos .env, excluimos la carpeta backend/, que contiene nuestro servicio backend, y las carpetas .vscode, .git y .husky. Ahora, creamos el archivo .dockerignore:

1. Cree un nuevo archivo .dockerignore en la raíz de nuestro proyecto.
2. Dentro de este archivo recién creado, ingrese el siguiente contenido:

```
módulos_de_nodo  
.env*  
backend  
.vscode  
.git  
.fornido  
.commitlintrc.json
```

Ahora que hemos ignorado los archivos que no son necesarios para la imagen de Docker, ¡construyámosla!

Construyendo la imagen de Docker del frontend

Al igual que antes, ejecutamos el comando docker build para construir la imagen, dándole el nombre blog-frontend y especificando el directorio raíz como ruta:

```
$ docker build -t blog-frontend .
```

Docker ahora usará la imagen del nodo para construir nuestra interfaz en la etapa de compilación. \$en, cambiará a la etapa final, usará la imagen nginx y copiará los archivos compilados desde la etapa de compilación.

Ahora, vamos a crear y ejecutar el contenedor frontend.

Creación y ejecución del contenedor frontend

De manera similar a lo que hicimos para el contenedor backend, también podemos crear y ejecutar un contenedor desde la imagen blog-frontend ejecutando el siguiente comando:

```
$ docker run -it -p 3000:80 interfaz del blog
```

La imagen nginx ejecuta el servidor web en el puerto 80, por lo tanto, si queremos usar el puerto 3000 en nuestro host, necesitamos reenviar del puerto 80 al 3000 pasando -p 3000:80.

Después de ejecutar este comando y navegar a <http://localhost:3000> en su navegador, debería ver que el frontend se sirve correctamente y se muestran publicaciones del blog desde el backend.

Ahora que hemos creado imágenes y contenedores para el backend y el frontend, vamos a aprender una forma de administrar múltiples imágenes más fácilmente.

Administrar múltiples imágenes con Docker Compose

Docker Compose es una herramienta que nos permite definir y ejecutar aplicaciones multi-contenedor con Docker. En lugar de crear y ejecutar manualmente los contenedores de backend, frontend y base de datos, podemos usar Compose para crearlos y ejecutarlos todos juntos. Para empezar a usar Compose, necesitamos crear un archivo `compose.yaml` en la raíz de nuestro proyecto, como se indica a continuación:

1. Cree un nuevo archivo `compose.yaml` en la raíz de nuestro proyecto.
2. Abra el archivo recién creado y comience por definir la versión de Docker Compose.

La especificación:

```
versión: '3.9'
```

3. Ahora, definamos un objeto de servicios, en el cual vamos a definir todos los servicios que queramos.

Para utilizar:

```
servicios:
```

4. Primero, tenemos `blog-database`, que usa la imagen de mongo y reenvía el puerto 27017:

```
base de datos del blog:
```

```
imagen: mongo
```

```
puertos:
```

```
- '27017:27017'
```

Nota

En los archivos YAML, la sangría de las líneas es muy importante para distinguir dónde están anidadas las propiedades, así que tenga cuidado de colocar la cantidad correcta de espacios antes de cada línea.

5. A continuación, tenemos blog-backend, que utiliza el Dockerfile definido en la carpeta backend/, define las variables de entorno para PORT y DATABASE_URL, reenvía el puerto al host y depende de blog-database:

```
backend del blog:
compilación: backend/
ambiente:
- PUERTO=3001
- URL DE LA BASE DE DATOS=mongodb://host.docker.internal:27017/blog
puertos:
- '3001:3001'
depende_de:
- base de datos de blogs
```

6. Por último, tenemos blog-frontend, que utiliza el Dockerfile definido en la raíz, define el argumento de compilación VITE_BACKEND_URL, reenvía el puerto al host y depende de blog-backend:

```
interfaz del blog:
construir:
contexto: .
argumentos:
URL_DE_VITE_BACKEND: http://localhost:3001/api/v1
puertos:
- '3000:80'
depende_de:
- blog-backend
```

7. Luego de definir los servicios, guarde el archivo.

8. \$en, detenga los contenedores backend y frontend que se ejecutan en la terminal utilizando la combinación de teclas Ctrl + C.

9. Además, detenga el contenedor dbserver que ya está en ejecución, de la siguiente manera:

```
$ docker stop dbserver
```

10. Finalmente, ejecute el siguiente comando en la Terminal para iniciar todos los servicios usando Docker Compose:

```
$ docker componer
```

Docker Compose creará contenedores para la base de datos, el backend y el frontend, y los iniciará todos. Verá que se imprimen los registros de los diferentes servicios. Si accede a <http://localhost:3000>. Puedes ver que el frontend está ejecutándose. Crea una nueva publicación para verificar que la conexión con el backend y la base de datos también funciona.

La siguiente captura de pantalla muestra la salida de Docker Compose Up creando e iniciando todos los contenedores:

```
○ → ~/d/F/ch5 | main: > docker compose up
[+] Running 3/0
  ✓ Container ch5-blog-database-1  Created
  ✓ Container ch5-blog-backend-1   Created
  ✓ Container ch5-blog-frontend-1 Created
Attaching to ch5-blog-backend-1, ch5-blog-database-1, ch5-blog-frontend-1
```

Figura 5.2 – Creación y ejecución de múltiples contenedores con Docker Compose

Luego, la salida de \$e en la captura de pantalla es seguida por mensajes de registro de varios servicios, incluido el servicio de base de datos MongoDB y nuestros servicios frontend y backend.

Como siempre, puedes presionar Ctrl + C para detener todos los contenedores de Docker Compose.

Ahora que hemos configurado Docker Compose, es muy fácil iniciar todos los servicios a la vez y administrarlos desde un solo lugar. Si revisas tus contenedores Docker, notarás que aún quedan muchos contenedores obsoletos de la creación previa del backend y el frontend del blog. Contenedores. Ahora aprendamos a limpiarlos.

Limpieza de contenedores no utilizados

Después de experimentar con Docker un tiempo, habrá muchas imágenes y contenedores que ya no se usen. Docker generalmente no elimina objetos a menos que se lo solicites explícitamente, lo que provoca un uso considerable de espacio en disco. Si quieras eliminar objetos, puedes eliminarlos uno por uno o usar uno de los comandos de poda de Docker:

- contenedor docker prune: \$is elimina todos los contenedores detenidos
- poda de imágenes de Docker: \$is elimina todas las imágenes colgantes (imágenes no etiquetadas y no referenciadas por ningún contenedor)
- docker image prune -a: \$is elimina todas las imágenes que no utiliza ningún contenedor
- docker volume prune: \$is elimina todos los volúmenes no utilizados por ningún contenedor
- Docker Network Prune: \$is limpia las redes que no utiliza ningún contenedor
- docker system prune: \$is poda todo excepto los volúmenes
- docker system prune --volumes: \$is poda todo

Entonces, si desea, por ejemplo, eliminar todos los contenedores no utilizados, primero debe asegurarse de que todos los contenedores que aún desea utilizar estén en ejecución. \$en, ejecutar docker container prune en la terminal.

Ahora que hemos aprendido a usar Docker localmente para empaquetar nuestros servicios como imágenes y ejecutarlos en contenedores, pasemos a implementar nuestra aplicación completa en la nube.

Implementando nuestra aplicación full-stack en la nube

Tras crear imágenes y contenedores Docker localmente, es hora de aprender a implementarlos en la nube para que todos puedan acceder a nuestros servicios. En este libro, usaremos Google Cloud como ejemplo, pero el proceso general también aplica a otros proveedores como Amazon Web Services (AWS) y Microsoft Azure. Para la base de datos MongoDB, usaremos MongoDB Atlas, pero puede usar cualquier proveedor que pueda alojar una base de datos MongoDB.

Creación de una base de datos MongoDB Atlas

Para alojar nuestra base de datos, utilizaremos MongoDB Atlas, la solución de nube social del equipo de MongoDB. Comencemos ahora con el registro y la configuración de la base de datos:

1. Vaya a <https://www.mongodb.com/atlas> y presione Probar gratis para crear una nueva cuenta o inicie sesión con su cuenta existente.

Nota

Las siguientes instrucciones pueden variar ligeramente debido a las actualizaciones de la interfaz de usuario de MongoDB Atlas. Si las opciones no están disponibles exactamente como se indican, intente seguir las instrucciones del sitio web para crear una base de datos y un usuario para acceder a ella. Esto aplica a todos los servicios en la nube que configuraremos en este capítulo.

2. Seleccione "Base de datos" en la barra lateral y pulse "Crear" para crear una nueva implementación de base de datos. Si creó una cuenta nueva, se le solicitará que cree una nueva implementación de base de datos automáticamente.
3. Seleccione Shared / M0 Sandbox (instancia gratuita) en Google Cloud y su región preferida.
4. Dale a tu clúster un nombre de tu elección.
5. Pulse "Crear" para crear su clúster de sandbox M0. El acceso a la base de datos tardará un tiempo (normalmente alrededor de un minuto). Sin embargo, puede continuar configurando el usuario mientras espera a que se configure el clúster.
6. Vaya a la sección Base de datos en la barra lateral y haga clic en el botón Conectar junto a su nueva base de datos clúster creado.
7. En la ventana emergente, seleccione Permitir acceso desde cualquier lugar y luego presione Agregar dirección IP.
8. Establezca un nombre de usuario y una contraseña para el usuario de su base de datos y presione Crear usuario de base de datos.
9. Presione Elegir un método de conexión y seleccione Controladores.
10. Se mostrará una cadena de conexión; cópiala y guárdela para más tarde, insertando la contraseña previamente establecida en lugar de la cadena <password>. La cadena de conexión \$e debe tener el siguiente formato:

```
mongodb+srv://<nombre de usuario>:<contraseña>@<nombre-del-clúster>. <id-del-clúster>
mongodb.net/?retryWrites=true&w=majority
```

11. Verifique que la cadena de conexión funcione abriendo un terminal y conectándose a él usando Mongo Shell:

```
$ mongosh "<cadena de conexión>"
```

La siguiente captura de pantalla muestra cómo se ve la pestaña Implementaciones de base de datos en MongoDB Atlas:

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with 'Project 0' selected under 'Database'. The main area is titled 'Database Deployments' and shows a single deployment named 'Full-Stack-React-Projects'. This deployment is listed as 'FREE' and 'SHARED'. It has 0 R and 0 W nodes, both with 0 connections and 0 B/s throughput. The data size is 0.0 B and it was last updated 2 minutes ago. Below this, there's a table with columns: VERSION, REGION, CLUSTER TIER, TYPE, BACKUPS, LINKED APP SERVICES, ATLAS SQL, and ATLAS SEARCH. The data for the cluster is: VERSION 6.0.5, REGION GCP / Belgium (europe-west1), CLUSTER TIER M0 Sandbox (General), TYPE Replica Set - 3 nodes, BACKUPS Inactive, LINKED APP SERVICES None Linked, ATLAS SQL Connect, and ATLAS SEARCH Create Index.

Figura 5.3 – Un nuevo clúster de base de datos M0 Sandbox implementado en MongoDB Atlas

Ahora que hemos creado con éxito nuestra base de datos MongoDB en la nube, podemos pasar a configurar Google Cloud para implementar nuestro backend y frontend.

Crear una cuenta en Google Cloud

Comencemos con Google Cloud creando una cuenta. Al crearla, deberá ingresar sus datos de facturación, pero recibirá \$300 en créditos gratuitos para probar Google Cloud gratis:

1. Vaya a <https://cloud.google.com> en su navegador.
2. Presione Comenzar gratis si aún no tiene una cuenta o presione Iniciar sesión si ya la tiene.
3. Inicia sesión con tu cuenta de Google y sigue las instrucciones hasta que tengas acceso a la Consola de Google Cloud.

Ahora debería ver una pantalla similar a la siguiente figura:

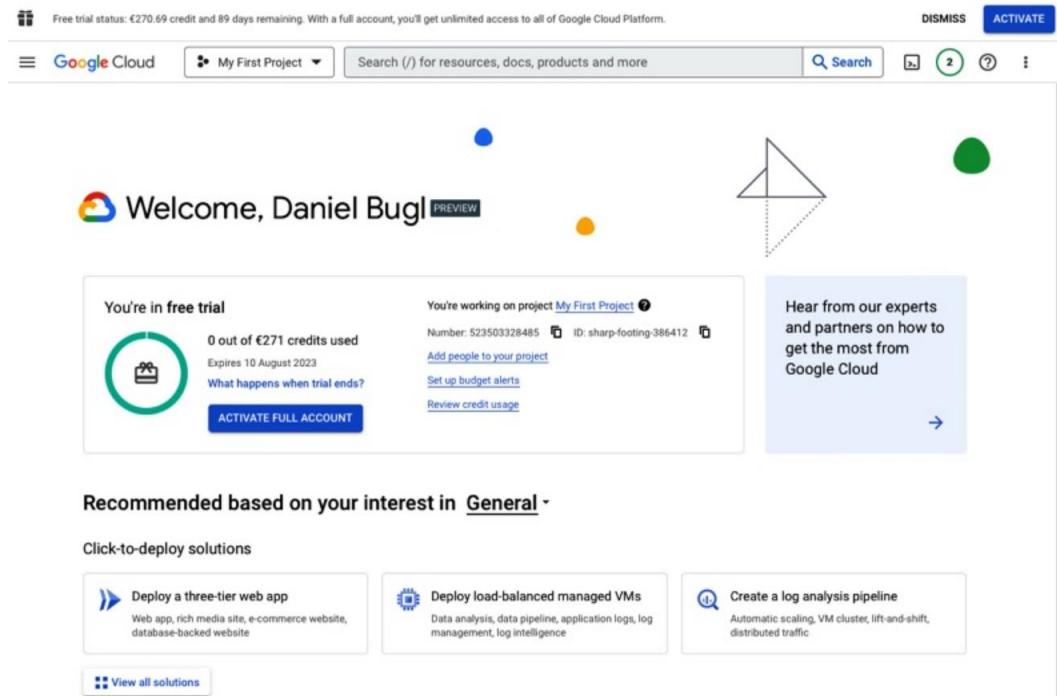


Figura 5.4 – La consola de Google Cloud después del registro

Ahora que tiene una cuenta configurada y lista, comencemos a implementar nuestros servicios.

Implementando nuestras imágenes Docker en un registro Docker

Antes de poder implementar un servicio en un proveedor de nube, primero debemos implementar nuestra imagen Docker en un registro Docker para que el proveedor de nube pueda acceder a ella desde allí y crear un contenedor a partir de ella. Siga estos pasos para implementar nuestras imágenes Docker en Docker Hub, el registro social de Docker:

1. Vaya a <https://hub.docker.com> e inicie sesión o registre una cuenta allí.
2. Presione el botón Crear repositorio para crear un nuevo repositorio. El repositorio \$e contendrá nuestra imagen.
3. Ingrese blog-frontend como nombre del repositorio y deje la descripción vacía y la visibilidad pública. Luego presione el botón Crear .
4. Repita los pasos 2 y 3, pero esta vez ingrese blog-backend como nombre del repositorio.

5. Abra una nueva terminal e ingrese el siguiente comando para iniciar sesión en su cuenta de Docker Hub:

```
$ inicio de sesión de docker
```

Ingrese su nombre de usuario y contraseña de Docker Hub y presione la tecla Retorno o Enter.

6. Reconstruya su imagen para Linux (para poder implementarla en Google Cloud más adelante), etiquete su imagen con el nombre de su repositorio (reemplace [USERNAME] con su nombre de usuario de Docker Hub) y envíela al repositorio:

```
$ docker build --platform linux/amd64 -t blog-frontend .
$ docker tag blog-frontend [NOMBRE DE USUARIO]/blog-frontend
$ docker push [NOMBRE DE USUARIO]/blog-frontend
```

7. Navegue hasta backend/ en la terminal y repita el paso 6 para la imagen del blog-backend:

```
$ cd backend/
$ docker build --platform linux/amd64 -t blog-backend .
$ docker tag blog-backend [NOMBRE DE USUARIO]/blog-backend
$ docker push [NOMBRE DE USUARIO]/blog-backend
```

Ahora que ambos repositorios están configurados y las imágenes se han enviado a ellos, deberían aparecer en Docker Hub con la siguiente información: Contiene: Imagen | Última inserción: hace unos segundos:

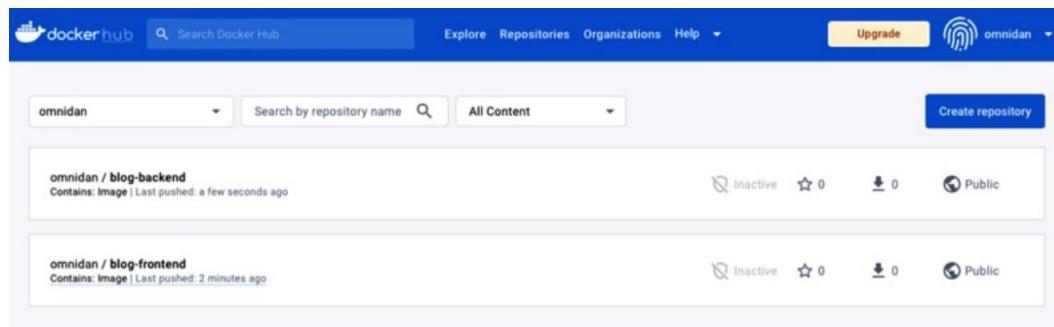


Figura 5.5 – Docker Hub ofrece una descripción general de nuestros repositorios

Ahora que nuestras imágenes de Docker están publicadas en un registro público de Docker (Docker Hub), podemos continuar configurando Google Cloud para implementar nuestros servicios.

Nota

Los repositorios creados en Docker Hub en este libro son públicos. También puede crear hasta un repositorio privado en Docker Hub de forma gratuita. De lo contrario, deberá tener una suscripción a Docker Hub, usar un registro diferente o alojar el suyo propio. Por ejemplo, Google Artifact Registry podría usarse para implementar imágenes privadas de Docker en Cloud Run.

Implementación de la imagen de Docker de backend en Cloud Run

Tras publicar correctamente nuestras imágenes de Docker en el registro de Docker Hub, es hora de implementarlas con Google Cloud Run. Cloud Run es una plataforma de computación administrada. Nos permite ejecutar contenedores directamente en la infraestructura de Google Cloud, lo que simplifica y agiliza la implementación de aplicaciones. Otras alternativas a Cloud Run serían infraestructuras basadas en Kubernetes, como AWS ECS Fargate o DigitalOcean.

Siga estos pasos para implementar el backend en Google Cloud Run:

1. Vaya a <https://console.cloud.google.com/>.
2. En la barra de búsqueda en la parte superior, ingrese Cloud Run y seleccione el producto Cloud Run – Serverless para aplicaciones en contenedores .
3. Pulse el botón Crear servicio para crear un nuevo servicio.

Nota

Es posible que primero deba crear un proyecto antes de crear un servicio. En ese caso, siga las instrucciones del sitio web para crear un nuevo proyecto con el nombre que desee. Posteriormente, pulse el botón "Crear servicio" para crear un nuevo servicio.

4. Ingrese [NOMBRE DE USUARIO]/blog-backend en el cuadro URL de la imagen del contenedor .
5. Ingrese blog-backend en el cuadro Nombre del servicio , seleccione una región de su elección, deje seleccionada la opción La CPU solo se asigna durante el procesamiento de la solicitud y seleccione Todo: Permitir acceso directo a su servicio desde Internet y Autenticación: Permitir invocaciones no autenticadas.
6. Expanda la sección Contenedor, Redes, Seguridad , desplácese hacia abajo hasta Variables de entorno y haga clic en Agregar variable.
7. Nombre la nueva variable de entorno DATABASE_URL y, como valor, ingrese la conexión cadena de MongoDB Atlas, que guardó anteriormente.

Nota

Para simplificar, usamos una variable de entorno normal. Para que las variables que contienen credenciales sean más seguras, se deben agregar como un secreto. Esto requiere habilitar la API de Secretos, agregar el secreto al gestor de secretos, referenciarlo y seleccionar su exposición como variable de entorno.

8. Deje el resto de las opciones como predeterminadas y presione Crear.
9. Serás redirigido al servicio recién creado, donde se está implementando el contenedor. Espera a que finalice la implementación (puede tardar un par de minutos).

10. Cuando el servicio termine de implementarse, verá una marca de verificación y una URL. Haga clic en la URL para abrir el backend y verá nuestro mensaje "¡Hola a todos de Express! ", lo que significa que nuestro backend se implementó correctamente en la nube.

Un servicio implementado se ve así en Google Cloud Run:



The screenshot shows the Google Cloud Run interface. At the top, there's a navigation bar with 'Cloud Run' and 'Service details' buttons, along with 'EDIT AND DEPLOY NEW REVISION' and 'SET UP' buttons. Below this, the service name 'blog-backend' is displayed with a green checkmark icon. To its right, the region 'europe-west1' is listed, followed by the URL 'https://blog-backend-tgncljyq66q-ew.a.run.app'. There are also icons for a clipboard and a help button.

Figura 5.6 – Un servicio implementado exitosamente en Google Cloud Run

Implementación de la imagen de Docker del frontend en Cloud Run

Para el frontend, primero necesitamos reconstruir el contenedor para cambiar VITE_BACKEND_URL Variable de entorno, que está integrada estáticamente en nuestro proyecto. Hagámoslo primero:

1. Abra una terminal y ejecute el siguiente comando para reconstruir el frontend con el entorno conjunto de variables:

```
$ docker build --platform linux/amd64 --build-arg "VITE_BACKEND_URL=[URL]/api/v1" -t blog-frontend .
```

Asegúrese de reemplazar [URL] con la URL del servicio backend implementado en Google Cloud Run.

2. Etiquételo con su nombre de usuario de Docker Hub e implemente la nueva versión de la imagen en Docker Hub:

```
$ docker tag blog-frontend [NOMBRE DE USUARIO]/blog-frontend  
$ docker push [NOMBRE DE USUARIO]/blog-frontend
```

Ahora, podemos repetir pasos similares a los que hicimos para implementar el backend para implementar también nuestro frontend:

1. Cree un nuevo servicio de Cloud Run, ingrese [NOMBRE DE USUARIO]/blog-frontend en el contenedor Cuadro de URL de la imagen y blog-frontend en el cuadro Nombre del servicio .
2. Elija una región de su elección y habilite Permitir invocaciones no autenticadas.
3. Expanda Contenedor, Redes, Seguridad y cambie el puerto del contenedor de 8080 a 80.
4. Presione Crear para crear el servicio y esperar a que se implemente.
5. Abra la URL en su navegador y debería ver el frontend implementado. Ahora también es posible agregar y listar entradas de blog enviando una solicitud al backend implementado, que almacena las entradas en nuestro clúster de MongoDB Atlas.

Hemos implementado manualmente con éxito nuestra primera aplicación full-stack de React y Node.js con una base de datos MongoDB en la nube. En las siguientes secciones, nos centraremos en la automatización de las pruebas y la implementación mediante CI/CD.

Configuración de CI para automatizar las pruebas

La Integración Continua (IC) abarca la automatización de la integración de cambios de código para detectar errores con mayor rapidez y mantener la base de código fácilmente mantenible. Generalmente, esto se facilita mediante la ejecución automática de scripts cuando un desarrollador realiza una solicitud de extracción/combinación, antes de que el código se integre en la rama principal. Esta práctica nos permite detectar problemas en nuestro código con antelación, por ejemplo, ejecutando el linter y las pruebas antes de que se pueda fusionar el código. Como resultado, la IC nos brinda mayor confianza en nuestro código y nos permite realizar e implementar cambios con mayor rapidez y frecuencia.

La siguiente figura muestra una descripción general simple de una posible secuencia de CI/CD:

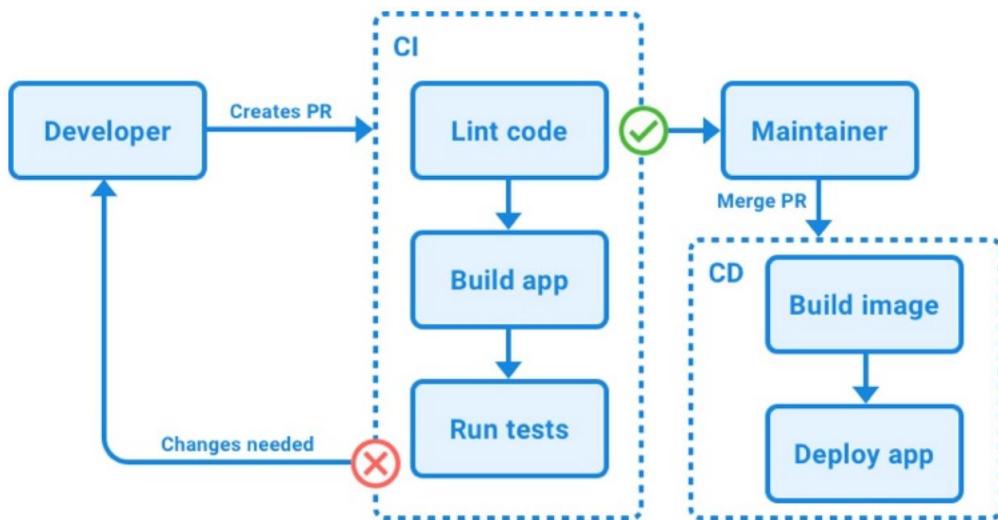


Figura 5.7 – Descripción general simple de una canalización de CI/CD

Nota

En este libro, usaremos GitHub Actions para CI/CD. Si bien la sintaxis y los archivos de configuración pueden tener un aspecto y un funcionamiento diferentes en otros sistemas, como GitLab CI/CD o CircleCI, los principios generales son similares.

En GitHub Actions, los flujos de trabajo se pueden activar cuando ocurren eventos en el repositorio, como el envío a una rama, la apertura de una nueva solicitud de extracción o la creación de una incidencia. Los flujos de trabajo pueden contener uno o varios trabajos, que se ejecutan en paralelo o secuencialmente. Cada trabajo se ejecuta dentro de su propio ejecutor, que recibe instrucciones de la definición de CI y las ejecuta dentro de un contenedor específico. Dentro de los trabajos, se pueden realizar acciones , que pueden ser acciones existentes en GitHub o acciones propias.

Añadiendo CI para el frontend

Comencemos a crear un flujo de trabajo que construirá el frontend cuando se crea una solicitud de extracción o se realiza un envío a la rama principal:

1. Crea una nueva carpeta .github/ en la raíz de nuestro proyecto. Dentro de ella, crea la carpeta workflows/.
2. Dentro de la carpeta .github/workflows/, crea un nuevo archivo llamado frontend-ci.yaml.
3. Abra el archivo .github/workflows/frontend-ci.yaml y comience dando la work%ow un nombre:

```
nombre: Blog Frontend CI
```

4. \$en, escucha eventos usando la palabra clave on. Ejecutaremos los trabajos cuando un nuevo La solicitud de extracción o inserción se realiza a la rama principal:

```
en:
```

```
empujar:
```

```
sucursales:
```

```
- principal
```

```
solicitud de extracción:
```

```
sucursales:
```

```
- principal
```

5. Ahora, definimos un trabajo que ejecutará el linter y construirá el frontend:

```
trabajos:
```

```
Pelusa y compilación:
```

6. Ejecutamos el trabajo en un contenedor ubuntu-latest:

```
se ejecuta en: ubuntu-latest
```

7. Podemos utilizar la estrategia matricial para ejecutar nuestras pruebas varias veces con diferentes variables.

En nuestro caso, queremos ejecutarlo en múltiples versiones de Node.js:

```
estrategia:
```

```
matriz:
```

```
versión del nodo: [16.x, 18.x, 20.x]
```

8. Ahora, definimos los pasos dentro de nuestro trabajo. Asegúrate de que los pasos estén definidos en el mismo...

Nivel de sangría como estrategia:

pasos:

9. Primero, usamos la acción actions/checkout, que extrae nuestro repositorio:

- usos: acciones/checkout@v3

10. \$en, usamos la acción actions/setup-node, que configura Node.js dentro de nuestro contenedor.

Aquí, utilizamos la variable node-version que definimos anteriormente:

- nombre: Usar Node.js \${{ matrix.node-version }}
usos: acciones/setup-node@v3
con:
versión del nodo: \${{ matrix.node-version }} caché: 'npm'

La opción de caché \$e especifica un administrador de paquetes que se utilizará para almacenar en caché las dependencias.

11. Finalmente, instalamos las dependencias, ejecutamos el linter y construimos nuestro frontend:

- nombre: Instalar dependencias ejecutar:
npm install
- nombre: Ejecutar linter en el frontend
ejecutar: npm run lint
- nombre: Construir frontend
ejecutar: npm run build

Agregar CI para el backend

Ahora que hemos agregado CI para el frontend, agreguemos también CI para el backend compilándolo y probándolo cuando se crea una solicitud de extracción o se realiza un push a la rama principal:

1. Dentro de la carpeta .github/workflows/, crea un nuevo archivo llamado backend-ci.yaml.

2. Abra el archivo .github/workflows/backend-ci.yaml, comience por darle un nombre y
Escuche los mismos eventos que escuchamos para la CI del frontend:

nombre: Blog Backend CI
en:
empujar:
sucursales:
- principal
solicitud de extracción:
sucursales:
- principal

3. Ahora, definimos un trabajo que compilará y probará el backend. Establecemos el directorio de trabajo predeterminado. a la carpeta backend/ para ejecutar todas las acciones dentro de esa carpeta:

```
trabajos:  
  Pelusa y prueba:  
    se ejecuta en: ubuntu-latest  
    estrategia:  
      matriz:  
        versión del nodo: [16.x, 18.x, 20.x]  
        valores predeterminados:  
          correr:  
            directorio de trabajo: ./backend
```

4. \$en, utilizamos las mismas acciones que para el frontend para verificar el repositorio y configurar Node.js:

```
pasos: -  
  usos: acciones/checkout@v3 - nombre: Usar Node.js  
  ${{ matrix.node-version }}  
  usos: acciones/setup-node@v3  
  con:  
    node-version: ${{ matrix.node-version }} cache: 'npm' - name: Instalar  
    dependencias run:  
      npm install
```

5. Finalmente, ejecutamos el linter en nuestro backend y ejecutamos las pruebas:

```
- nombre: Ejecutar linter en el backend  
  ejecutar: npm run lint - nombre:  
 Ejecutar pruebas de backend  
  ejecutar: npm test
```

6. Guarde los archivos de flujo de trabajo, confírmelo y envíelos a un repositorio de GitHub creando un nuevo repositorio en GitHub y siguiendo sus instrucciones para enviar un repositorio existente a GitHub.

7. Ve al repositorio en GitHub y selecciona la pestaña "Acciones" . Deberías ver tus flujos de trabajo. corriendo aquí.

La siguiente captura de pantalla muestra nuestros flujos de trabajo de CI ejecutándose correctamente en GitHub:

The screenshot shows the GitHub Actions interface. On the left, there's a sidebar with 'Actions' selected, showing 'All workflows'. Under 'All workflows', there are entries for 'Blog Backend CI' and 'Blog Frontend CI'. The main area is titled 'All workflows' and shows '4 workflow runs'. It lists two runs for 'ci: initial setup of ci workflows': one for 'Blog Backend CI' pushed by 'omnidan' and another for 'Blog Frontend CI' pushed by 'omnidan'. Both runs are in the 'main' branch and completed 2 minutes ago, with a duration of 50s for the Backend and 38s for the Frontend.

Figura 5.8 – Flujos de trabajo de CI de backend y frontend que se ejecutan correctamente en GitHub Actions

Si realizamos una nueva solicitud de extracción a la rama principal, también podemos ver que nuestros flujos de trabajo de integración continua (CI) se ejecutan correctamente en el nuevo código. Por ejemplo, si añadimos una forma de etiquetar publicaciones desde el frontend y, sin querer, establecemos que las etiquetas fueran obligatorias en el backend sin tener en cuenta nuestra regla anterior de que solo se requiriera el título, veremos que las pruebas correspondientes fallaron:

This screenshot shows a GitHub pull request page. At the top, it says 'Some checks were not successful' with '2 cancelled, 3 successful, and 1 failing checks'. Below this, there's a table of checks:

Check Status	Check Name	Status Details	Details Link
Cancelled	Blog Backend CI / lint-and-test (14.x) (pull_request)	Cancelled after 39s	Details
Successful	Blog Frontend CI / lint-and-build (14.x) (pull_request)	Successful in 33s	Details
Failing	Blog Backend CI / lint-and-test (16.x) (pull_request)	Failing after 22s	Details
Successful	Blog Frontend CI / lint-and-build (16.x) (pull_request)	Successful in 22s	Details
Cancelled	Blog Backend CI / lint-and-test (18.x) (pull_request)	Cancelled after 28s	Details
Successful	Blog Frontend CI / lint-and-build (18.x) (pull_request)	Successful in 16s	Details

Below the table, a green checkmark icon indicates 'This branch has no conflicts with the base branch' and notes that merging can be performed automatically. At the bottom, there's a 'Merge pull request' button and a note about opening the pull request in GitHub Desktop or viewing command line instructions.

Figura 5.9 – El flujo de trabajo de CI de backend falla en una solicitud de extracción

También podemos ver que GitHub Actions cancela automáticamente los trabajos que se están ejecutando para otras versiones de Node.js después de que uno de ellos ya falló, para evitar perder tiempo.

Ahora que hemos configurado exitosamente nuestros flujos de trabajo de CI, continuemos configurando CD para automatizar la implementación de nuestra aplicación de pila completa.

Configuración del CD para automatizar la implementación

Tras la fusión de la solicitud de extracción/combinación, entra en juego la entrega continua (CD) . La CD automatiza el proceso de lanzamiento mediante la implementación automática de los servicios y aplicaciones. Normalmente, esto implica un proceso de varias etapas, donde el código se implementa primero automáticamente en un entorno de pruebas y luego se puede implementar manualmente en otros entornos, hasta llegar a producción. Si la implementación en producción también es un proceso automatizado, se denomina implementación continua en lugar de entrega continua.

Primero, necesitamos obtener las credenciales para autenticarnos con Docker Hub y Google Cloud. Luego, podemos configurar el flujo de trabajo para implementar nuestro blog.

Obtener credenciales de Docker Hub

Comencemos por obtener las credenciales para acceder a Docker Hub:

1. Vaya a <https://hub.docker.com/>.
2. Haz clic en tu perfil y ve a la configuración de tu cuenta.
3. Haga clic en la pestaña Seguridad y presione el botón Nuevo token de acceso .
4. Como descripción, escribe "Acciones de GitHub" y pulsa el botón "Generar" . Otorga permisos de lectura, escritura y eliminación .
5. Copie el token de acceso y guárdelo en un lugar seguro.
6. Vaya a su repositorio de GitHub y luego a Configuración | Secretos y variables | Acciones.
7. Pulse el botón "Nuevo secreto de repositorio" para añadir un nuevo secreto. Escriba "DOCKERHUB_" como nombre. NOMBRE DE USUARIO y, como valor secreto, use su nombre de usuario en Docker Hub.
8. Agregue otro secreto con el nombre DOCKERHUB_TOKEN y pegue el secreto creado anteriormente. token de acceso como valor secreto.

Obtener credenciales de Google Cloud

Ahora, vamos a crear una cuenta de servicio para acceder a Google Cloud Run:

1. Vaya a <https://console.cloud.google.com/>.
2. En el cuadro de búsqueda de la parte superior, ingrese Cuentas de servicio y vaya a IAM y administración – Servicio Página de cuentas .
3. Presione el botón Crear cuenta de servicio .
4. En el cuadro Nombre de la cuenta de servicio , ingrese GitHub Actions. El ID \$e debería ser automáticamente Generado como acciones de GitHub. Pulse " Crear" y "Continuar".
5. Otorgue al servicio acceso al rol de administrador de Cloud Run y presione Continuar.
6. Presione Listo para terminar de crear la cuenta de servicio.

7. En la lista de descripción general, copie el correo electrónico de su cuenta de servicio recién creada y guárdelo para usarlo más adelante.
8. Acceda a la cuenta de servicio de cómputo predeterminada haciendo clic en su dirección de correo electrónico. Vaya a Permisos. pestaña y presione Otorgar acceso.
9. Pegue el correo electrónico de la cuenta de servicio recién creada en el campo "Nuevos administradores" y asigne el rol de Agente de servicio de Cloud Run . Pulse "Guardar" para confirmar.
10. En la lista de descripción general, presione el ícono de tres puntos para abrir acciones en sus acciones de GitHub. cuenta de servicio y seleccione Administrar claves.
11. En la nueva página, pulse "Añadir clave | Crear nueva clave" y pulse "Crear" en la ventana emergente. Se descargará un archivo JSON .
12. Ve a tu repositorio de GitHub y ve a Configuración | Secretos y variables | Acciones. Presiona el botón Nuevo botón secreto de repositorio para agregar un nuevo secreto.
13. Agrega un nuevo secreto en tu repositorio de GitHub llamado GOOGLECLOUD_SERVICE_ACCOUNT y pegue el correo electrónico previamente copiado de su cuenta de servicio recién creada como valor secreto.
14. Agregue un nuevo secreto en su repositorio de GitHub llamado GOOGLECLOUD_CREDENTIALS y como secreto, pegue el contenido del archivo JSON descargado.
15. Agregue un nuevo secreto en su repositorio de GitHub llamado GOOGLECLOUD_REGION y establezca el valor del secreto en la región que seleccionó al crear los servicios de Cloud Run.

Nota

Para mayor seguridad, Google recomienda usar la federación de identidades de carga de trabajo en lugar de exportar las credenciales JSON de la clave de la cuenta de servicio. Sin embargo, configurar la federación de identidades de carga de trabajo es un poco más complicado. Puede encontrar más información sobre cómo configurarla aquí: <https://github.com/google-github-actions/auth#setup>.

Definición del flujo de trabajo de implementación

Ahora que las credenciales están disponibles como valores secretos para nuestros flujos de trabajo de CI/CD, podemos comenzar a definir el flujo de trabajo de implementación:

1. Dentro de la carpeta .github/workflows/, crea un nuevo archivo llamado cd.yaml.
2. Abra el archivo .github/workflows/cd.yaml y comience por darle un nombre:

nombre: Implementar aplicación de blog

3. Para CD, solo ejecutamos el flujo de trabajo al enviar a la rama principal:

en:

empujar:

sucursales:

- principal

4. Comenzamos a definir un trabajo de implementación, en el que establecemos el entorno en producción y apuntamos la URL a la URL del frontend implementado:

```
trabajos:  
  implementar:  
    se ejecuta en: ubuntu-latest  
    ambiente:  
      nombre: producción url: ${  
        {{ steps.deploy-frontend.outputs.url }}  
      }
```

Más adelante definiremos un paso con el ID de implementación de frontend, que almacena una variable en steps.deploy-frontend.outputs.url.

5. Para los pasos que hicimos antes, primero debemos revisar nuestro repositorio:

```
pasos:  
  - usos: acciones/checkout@v3
```

6. \$en, iniciamos sesión en Docker Hub usando las credenciales que configuramos anteriormente en nuestros secretos:

```
- nombre: Iniciar sesión en Docker Hub  
  usos: docker/login-action@v2 con:  
  
    nombre de usuario: ${{ secrets.DOCKERHUB_USERNAME }}  
    contraseña: ${{ secrets.DOCKERHUB_TOKEN }}
```

7. A continuación, iniciamos sesión en Google Cloud con las credenciales que configuramos anteriormente:

```
- usos: google-github-actions/auth@v1  
  con:  
    cuenta_de_servicio: ${{ secretos.CUENTA_DE_SERVICIO_DE_GOOGLECLOUD }}  
  credenciales_json:  
    ${{ secretos.CREDENCIALES_DE_GOOGLECLOUD }}  
  }}
```

8. Ahora, construimos y enviamos la imagen de Docker de backend usando docker/build-push-action, que crea y envía una imagen a un registro de Docker:

```
- nombre: Construir y enviar imágenes de backend utiliza: docker/  
  build-push-action@v4 con:  
  
  contexto: ./backend archivo: ./  
  backend/Dockerfile  
  push: verdadero  
  etiquetas: ${{ secrets.DOCKERHUB_USERNAME }}/blog-backend:latest
```

9. Después de enviar la imagen de Docker para el backend, ahora podemos implementarla en Cloud Run, usando la acción google-github-actions/deploy-cloudrun:

```
- id: implementar-backend
  nombre: Implementar backend
  usos: google-github-actions/deploy-cloudrun@v1
  con:
    servicio: blog-backend
    imagen: ${{ secrets.DOCKERHUB_USERNAME }}/blog-
backend:último
    región: ${{ secrets.GOOGLECLOUD_REGION }}
```

Le otorgamos a este paso el ID de implementación del backend, ya que necesitamos usarlo para hacer referencia a la URL del backend para construir la imagen del frontend en el siguiente paso.

10. Despues de construir e implementar el backend, construimos el frontend de manera similar, asegurándonos para pasar VITE_BACKEND_URL como argumentos de compilación:

```
- nombre: Construir y enviar la imagen del frontend
  usos: docker/build-push-action@v4
  con:
    contexto: .
    archivo: ./Dockerfile
    empujar: verdadero
    etiquetas: ${{ secrets.DOCKERHUB_USERNAME }}/blog-
frontend:último
    argumentos de construcción: VITE_BACKEND_URL=${{ pasos.implementar-backend.
salidas.url }}/api/v1
```

11. Finalmente, podemos implementar el frontend, asignando a este paso el ID de implementación de frontend, para que la URL de nuestro entorno se pueda configurar correctamente:

```
- id: implementar-frontend
  nombre: Implementar frontend
  usos: google-github-actions/deploy-cloudrun@v1
  con:
    servicio: blog-frontend
    imagen: ${{ secrets.DOCKERHUB_USERNAME }}/blog-
frontend:último
    región: ${{ secrets.GOOGLECLOUD_REGION }}
```

12. Guarda el archivo, confirma y envía los cambios a la rama principal. Verás el blog de implementación.
Aplicación que se activa en GitHub Actions.

La siguiente captura de pantalla muestra el resultado de la implementación exitosa de nuestra aplicación de blog a través de Acciones de GitHub:

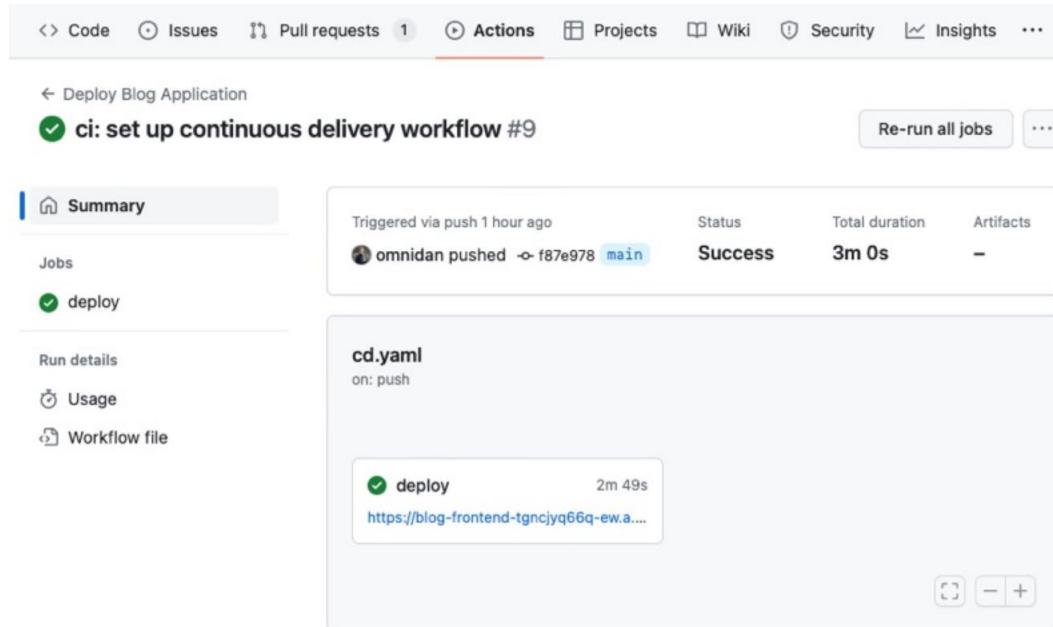


Figura 5.10 – Una implementación exitosa de nuestra aplicación full-stack usando GitHub Actions

Puede hacer clic en la URL para abrir la interfaz implementada y verá que funciona de la misma manera que la versión implementada manualmente.

¡Felicitaciones! ¡Has automatizado con éxito la integración y el despliegue de tu primera aplicación full-stack!

Nota

En este libro, creamos una implementación de una sola etapa, que se implementa automáticamente directamente en producción. En una aplicación real, es posible que desee definir varias etapas. Por ejemplo, CD podría implementarse automáticamente en un entorno de pruebas. La implementación en producción podría configurarse para requerir confirmación manual.

Resumen

En este capítulo, comenzamos aprendiendo a crear imágenes de Docker y a instanciar contenedores locales a partir de ellas. Luego, automatizamos este proceso con Docker Compose. A continuación, publicamos nuestras imágenes en el registro de Docker Hub para poder implementarlas en Google Cloud Run. Después, implementamos manualmente nuestra aplicación completa en Cloud Run. Finalmente, aprendimos a configurar CI/CD que funciona con GitHub Actions para automatizar la ejecución del linter, las pruebas y la implementación de la aplicación del blog.

Hasta ahora, todo en nuestra aplicación ha sido de acceso público. Sin gestión de usuarios, cualquiera puede crear publicaciones como cualquier autor. En el siguiente capítulo, Capítulo 6, "Añadiendo autenticación con JWT", aprenderemos a implementar cuentas de usuario y autenticación en nuestra aplicación de blog integral. Aprenderemos qué son los JSON Web Tokens (JWT) e implementaremos múltiples rutas para iniciar sesión y registrarse.

Parte 3:

Practicando el desarrollo de Aplicaciones web de pila completa

En esta parte, profundizaremos en el desarrollo web full-stack. Comenzaremos añadiendo autenticación a nuestra aplicación mediante tokens web JSON. Aprenderemos a mejorar el tiempo de carga mediante la renderización del lado del servidor. A continuación, aprenderemos a optimizar una aplicación para motores de búsqueda. También implementaremos pruebas integrales con Playwright para garantizar la robustez de nuestra aplicación. Aprenderemos a recopilar eventos, agregar datos con MongoDB y crear estadísticas para visualizar los datos agregados con Victory. Al final de esta parte, aprenderemos a crear un backend con una API GraphQL y a interactuar con GraphQL en el frontend con Apollo Client.

Esta parte incluye los siguientes capítulos:

- Capítulo 6, Agregar autenticación con JWT
- Capítulo 7, Mejora del tiempo de carga mediante la representación del lado del servidor
- Capítulo 8, Cómo asegurarse de que los clientes lo encuentren con la optimización de motores de búsqueda
- Capítulo 9, Implementación de pruebas de extremo a extremo con Playwright
- Capítulo 10, Agregación y visualización de estadísticas utilizando MongoDB y Victory
- Capítulo 11, Creación de un backend con una API GraphQL
- Capítulo 12, Interfaz con GraphQL en el frontend usando el cliente Apollo

Agregar autenticación con JWT

Tras desarrollar e implementar nuestra primera aplicación full-stack, ahora podemos crear entradas en nuestro blog. Sin embargo, dado que el autor es un campo de entrada, cualquiera podría introducir cualquier autor, ¡suplantando la identidad de otros! Eso no es bueno. En este capítulo, añadiremos autenticación con JSON Web Token (JWT) y funcionalidades para registrarse e iniciar sesión en nuestra aplicación añadiendo rutas adicionales con React Router.

En este capítulo cubriremos los siguientes temas principales:

- ¿Qué es JWT?
- Implementación de inicio de sesión, registro y rutas autenticadas en el backend usando JWT
- Integración de inicio de sesión y registro en el frontend usando React Router y JWT
- Manejo avanzado de tokens

Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones mencionadas en esos capítulos son las que se utilizan en este libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que algunos pasos podrían funcionar de forma diferente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas en el Capítulo 1 y el Capítulo 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/capítulo6>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/LloHmkgrLWk>.

¿Qué es JWT?

JWT, que se pronuncia "jot", es un estándar abierto de la industria (RFC 7519) para la transmisión segura de notificaciones entre múltiples partes. Las notificaciones pueden contener información sobre una parte u objeto específico, como la dirección de correo electrónico, el ID de usuario y los roles de un usuario. En nuestro caso, transmitiremos JWT entre nuestro backend y frontend.

Muchos productos y servicios utilizan JWT y son compatibles con proveedores de autenticación externos, como Auth0, Okta y Firebase Auth. Analizar JWT es sencillo, ya que solo necesitamos decodificarlos en base64 y analizar la cadena JSON. Tras verificar la firma, podemos garantizar la autenticidad del JWT y confiar en sus afirmaciones.

Los JWT constan de los siguientes componentes:

- Encabezado: Contiene el algoritmo y el tipo de token
- Carga útil: contiene los datos/reclamos del token
- Firma: para verificar que el token fue creado por una fuente legítima

Estos tres componentes forman un JWT ya que se unen en una sola cadena, separados por un punto (.), de la siguiente manera:

```
encabezado.carga útil.firma
```

Veamos cada componente por separado.

Encabezado JWT

El encabezado JWT \$e generalmente consta de un tipo de token (en nuestro caso, JWT), especificado por la propiedad typ, y el algoritmo utilizado para crear la firma (en nuestro caso, utilizaremos HMAC SHA256, un código de autenticación de mensajes basado en hash SHA256), especificado por la propiedad alg. El encabezado \$e se define como un objeto JSON, de la siguiente manera:

```
{  
  "alg": "HS256",  
  "típico": "JWT"  
}
```

Luego, el objeto JSON se codifica en base64 y forma la primera parte del JWT.

Carga útil JWT

La parte principal del JWT es la carga útil, que contiene todas las reclamaciones. Las reclamaciones contienen información sobre una entidad (como el usuario) y datos adicionales. El estándar JWT distingue tres tipos de reclamaciones:

- Reclamaciones registradas: Son reclamaciones predefinidas y se recomienda configurarlas. Incluyen información sobre lo siguiente:

\$e emisor (iss), que es la entidad que creó el token.

\$e tiempo de expiración (exp), que nos indica cuándo expira el token.

\$e subject (sub), que nos informa sobre la entidad identificada por el token (como el usuario que generó el token durante un inicio de sesión).

Audiencia electrónica (aud), que nos informa sobre los destinatarios previstos del token.

\$e emitido en el momento (iat), que nos indica cuándo se creó el token.

\$e no antes del tiempo (nbf), que especifica un tiempo antes del cual el token aún no es válido.

El ID de JWT (jti) proporciona un identificador único para el JWT. Se utiliza para evitar que los JWT se reproduzcan.

Nota

Las propiedades de los objetos JSON definidas en el estándar JWT son todos nombres de tres letras para mantener el JWT lo más compacto posible.

- Reclamos públicos: Son reclamos adicionales que se usan comúnmente y se comparten entre muchos servicios. Puede encontrar una lista de ellos en el sitio web de la Autoridad de Números Asignados en Internet (IANA) : <https://www.iana.org/assignments/jwt/jwt.xhtml>. Si desea almacenar información adicional, siempre debe consultar primero esta lista para ver si puede usar un nombre de reclamo estandarizado.
- Reclamos privados: Son reclamos personalizados, que no están registrados ni son públicos. Si necesitamos un reclamo especial que aún no esté definido, podemos crear uno privado que solo nuestros servicios comprenderán.

Todas las reivindicaciones son opcionales, pero tiene sentido incluir al menos una reivindicación para identificar al sujeto, como por ejemplo la reivindicación subregistrada.

Juntando lo que hemos aprendido, podemos crear la siguiente carga útil de ejemplo:

```
{  
    "sub": "1234567890",  
    "nombre": "Daniel Bugl",  
    "admin": verdadero  
}
```

En nuestro ejemplo, la reclamación secundaria es una reclamación registrada, la reclamación de nombre es una reclamación pública y el administrador

La reclamación es una reclamación privada.

La carga útil \$e también está codificada en base64 y constituye la segunda parte del JWT. Por lo tanto, esta información es públicamente legible para cualquier persona con acceso al token. ¡No incluya información secreta en la carga útil ni en el encabezado de un JWT! Sin embargo, la información no se puede modificar sin invalidar la firma existente, lo que garantiza la seguridad de todas las afirmaciones. Solo un servicio de backend con acceso a la clave privada puede generar una nueva firma para crear un JWT válido.

Firma JWT

La parte final de un JWT es su firma. La firma es lo que prueba que toda la información que hemos definido hasta ahora no ha sido alterada. La firma se crea tomando la base64-encabezado y carga útil codificados, uniendo esas cadenas con un símbolo de punto y utilizando el algoritmo especificado para firmarlos con una clave secreta:

```
HMACSHA256(  
    base64UrlEncode(encabezado) + "." + base64UrlEncode(carga útil),  
    secreto  
)
```

Ahora que hemos aprendido sobre los diferentes componentes de un JWT, juntemos todo esto para crear un JWT válido.

Creando un JWT

Siga estos pasos para crear un JWT:

1. Vaya al sitio web <https://jwt.io/> y desplácese hacia abajo hasta la sección Depurador .
2. Ingrese nuestro encabezado y carga útil previamente definidos.
3. Ingrese full-stack como secreto.
4. El JWT codificado \$e debe actualizarse en %y a medida que cambia los valores.

Como puedes ver, hemos creado con éxito nuestro primer JWT:

The screenshot shows the jwt.io debugger interface. At the top, there's a warning message: "Warning: JWTs are credentials, which can grant access to resources. Be careful where you paste them! We do not record tokens, all validation and debugging is done on the client side." Below this, there's a dropdown menu for the algorithm set to "HS256".

Encoded (PASTE A TOKEN HERE):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkRhbmllbCBcdWdsIiwiYWRTaW4iOnRydWV9.G5Nv3gt01UCJXn0cffq-_c3mwLjybv6xuKxdxp0iZXo
```

Decoded (EDIT THE PAYLOAD AND SECRET):

- HEADER: ALGORITHM & TOKEN TYPE**

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```
- PAYOUT: DATA**

```
{
  "sub": "1234567890",
  "name": "Daniel Bugl",
  "admin": true
}
```
- VERIFY SIGNATURE**

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  full-stack) □ secret base64 encoded
```

A note here says "Weak secret!" with a warning icon.

SHARE JWT

Signature Verified

Figura 6.1 – Nuestro primer JWT, creado con el depurador jwt.io

El JWT generado consta de tres componentes, cada uno codificado en base64 y separado por puntos. En el depurador, se resaltan en tres colores diferentes. Intenta cambiar la cadena base64 en la sección "Codificado" eliminando algunos caracteres; verá que el JWT ya no es válido debido a un problema de "Firma no válida". Ahora que hemos creado nuestro primer JWT, aprendamos a usarlo.

Uso de JWT

Durante el proceso de inicio de sesión, generaremos un JWT para el usuario conectado en el backend. Este JWT se devolverá al navegador del usuario. Cuando el usuario desee acceder a una ruta protegida, podemos enviar el JWT al servidor backend mediante el encabezado de autorización con el esquema de portador, como se indica a continuación:

Autorización: Portador <token>

El backend puede entonces buscar este encabezado, verificar la firma del token y otorgar al usuario acceso a ciertas rutas. Al enviar el token en un encabezado en lugar de una cookie, evitamos los problemas de CORS que tendríamos al usar cookies.

Nota

Tenga cuidado de no enviar demasiados datos en el encabezado, ya que algunos servidores no aceptan más de 8 KB. \$is significa que, por ejemplo, la información compleja de roles no debe almacenarse en las notificaciones JWT, ya que podría ocupar demasiado espacio. En su lugar, este tipo de información podría almacenarse en la base de datos asociada a un ID de usuario del JWT.

Una ventaja interesante de usar un JWT es que el servidor de autenticación y el backend de nuestra aplicación no tienen que ser el mismo. Podríamos tener un servicio de autenticación independiente, obtener un JWT y, en el backend, verificar la firma de los JWT para garantizar que fueron generados por el servicio de autenticación. \$is nos permite usar servicios externos para la autenticación, como Auth0, Okta o Firebase Auth.

El siguiente diagrama muestra el flujo de autorización para un JWT:

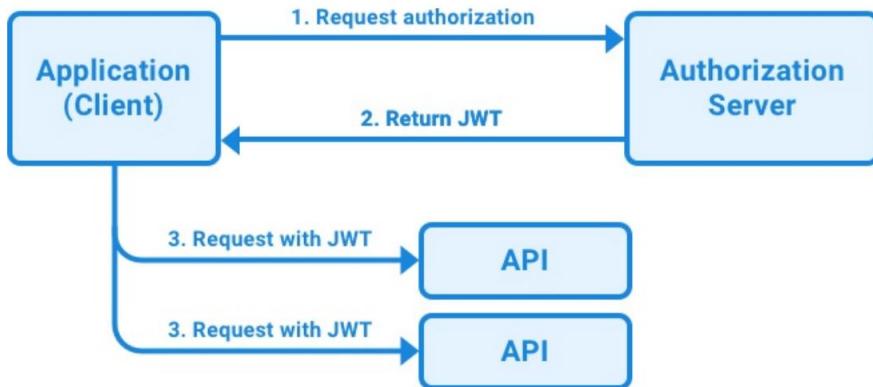


Figura 6.2 – Flujo de autorización para un JWT

Como podemos ver, la aplicación solicita autorización al servidor de autorización, que puede ser un proveedor externo, un servicio independiente o parte del servicio backend. \$en, cuando se concede la autorización (si los datos de inicio de sesión son correctos), el servidor de autorización devuelve un JWT. \$is JWT puede utilizarse para acceder a rutas protegidas en las API. Antes de conceder el acceso, se valida la firma JWT para garantizar que no haya sido alterada.

Almacenamiento de JWT

Debemos tener mucho cuidado con el almacenamiento del JWT. El almacenamiento local no es una buena opción para almacenar información de autenticación como un JWT. Las secuencias de comandos entre sitios pueden utilizarse para robar todos los datos del almacenamiento local. Para los tokens de corta duración, podemos almacenarlos en una variable de tiempo de ejecución de JavaScript (como un contexto de React). Para un almacenamiento a largo plazo, podríamos utilizar una cookie httpOnly, que tiene garantías de seguridad adicionales.

Ahora que aprendimos cómo funciona JWT, pongamos la teoría en práctica e implementemos el inicio de sesión, el registro y las rutas autenticadas en el backend usando JWT.

Implementación de inicio de sesión, registro y rutas autenticadas en el backend mediante JWT

Ahora que conocemos los JWT, los implementaremos en nuestro backend. Primero, necesitamos crear un modelo de usuario en nuestra base de datos, tras el cual podremos crear rutas para registrarse e iniciar sesión en nuestra aplicación. Finalmente, implementaremos rutas autenticadas a las que solo se pueda acceder con un JWT.

Creando el modelo de usuario

Comenzaremos la implementación del backend creando un modelo de usuario, de la siguiente manera:

1. Copie la carpeta ch5 a una nueva carpeta ch6, de la siguiente manera:

```
$ cp -R ch5 ch6
```

2. Abra la carpeta ch6 en VS Code.

3. Cree un nuevo archivo backend/src/db/models/user.js y defina un nuevo userSchema

En su interior:

```
importar mangosta, { Esquema } desde 'mangosta'
```

```
const esquema de usuario = nuevo esquema ({
```

4. El usuario debe tener un nombre de usuario único y una contraseña obligatoria:

```
nombre de usuario: { tipo: String, requerido: verdadero, único: verdadero },
contraseña: { tipo: String, requerido: true },
})
```

5. Crea y exporta el modelo:

```
exportar const Usuario = mongoose.model('usuario', userSchema)
```

6. En este punto, ajustemos también el modelo de publicación para que podamos almacenar una referencia a un ID de usuario en lugar del nombre de usuario como autor. Edite backend/src/db/models/post.js de la siguiente manera:

```
autor: { tipo: Schema.Types.ObjectId, ref: 'usuario', requerido: verdadero },
```

Cambiamos el tipo a ObjectId, con una referencia al modelo de usuario, e hicimos autor requerido (ya que deberá iniciar sesión para crear una publicación, agregaremos una ruta autenticada más adelante en este capítulo).

Hacer que el autor sea obligatorio significa que será necesario ajustar las pruebas unitarias, pero hacerlo es solo un ejercicio para usted.

Ahora que hemos creado con éxito el modelo de usuario, pasemos a la creación del servicio de registro para que tengamos una forma de crear nuevos usuarios.

Creación del servicio de registro

Cuando un usuario se registra, necesitamos generar un hash de la contraseña proporcionada por el usuario antes de almacenarla en la base de datos. Nunca debemos almacenar las contraseñas en texto plano, ya que esto significaría que, si nuestra base de datos se filtra, un atacante tendría acceso a las contraseñas de todos los usuarios. El hash es una función unidireccional que convierte una cadena en una diferente de forma determinista. Esto significa que, por ejemplo, si ejecutamos hash("contraseña1"), obtenemos una cadena específica cada vez. Sin embargo, si ejecutamos hash("contraseña2"), obtenemos una cadena completamente diferente. Al elegir una buena función hash, podemos garantizar que revertir un hash sea tan costoso computacionalmente que sea imposible hacerlo en un tiempo razonable. Cuando el usuario se registra, podemos almacenar el hash de su contraseña. Cuando un usuario ingresa su contraseña para iniciar sesión, podemos generar un hash de nuevo y compararlo con el hash en la base de datos.

Comencemos a implementar el servicio de registro con contraseñas en hash:

1. Instale el paquete npm bcrypt. Lo usaremos para generar un hash de la contraseña antes de almacenarla:

```
$ cd backend  
$ npm install bcrypt@5.1.1
```

2. Cree un nuevo archivo backend/src/services/users.js e importe bcrypt y el Modelo de usuario:

```
importar bcrypt desde 'bcrypt'  
importar { Usuario } desde './db/models/user.js'
```

3. Defina una función createUser que tome valores de nombre de usuario y contraseña:

```
exportar función asíncrona createUser({nombre de usuario, contraseña}) {
```

4. Dentro de esta función, usamos la función bcrypt.hash para crear un hash a partir de la contraseña de texto plano usando 10 rondas de sal (repetiendo el hash 10 veces para que sea aún más difícil revertirlo):

```
const hashPassword = await bcrypt.hash(contraseña, 10)
```

5. Ahora, podemos crear un nuevo usuario y almacenarlo en nuestra base de datos:

```
const usuario = nuevo Usuario({nombre de usuario, contraseña: hashPassword})
devolver esperar usuario.save()
}
```

Para abreviar, no abordaremos la creación de pruebas para los servicios de usuario. Consulta el Capítulo 3, "Implementación de un servicio backend con Express, Mongoose ODM y Jest", para obtener información sobre cómo crear pruebas para tus funciones de servicio. Puedes escribir pruebas similares a las que hicimos para las funciones de servicio de las publicaciones.

Después de crear el servicio de registro, podemos crear la ruta de registro.

Creando la ruta de registro

Ahora, expongamos la función del servicio de registro agregando una ruta API para ella:

1. Cree un nuevo archivo backend/src/routes/users.js e importe el servicio createUser:

```
importar { createUser } desde '../services/users.js'
```

2. Defina una nueva función userRoutes y exponga una ruta POST /api/v1/user/signup. \$is route crea un nuevo usuario a partir del cuerpo de la solicitud y devuelve el nombre de usuario:

```
función de exportación userRoutes(app) {
    app.post('/api/v1/usuario/registrar', async (req, res) => {
        intentar {
            const usuario = await crearUsuario(req.cuerpo)
            devolver res.status(201).json({ nombredeusuario: usuario.nombredeusuario })
        } atrapar (err) {
            devuelve res.status(400).json({
                error: 'Error al crear el usuario, ¿el nombre de usuario ya existe?'
            })
        }
    })
}
```

En este caso, definimos una ruta de usuario única en lugar de llamarla "usuarios", ya que solo trabajamos con un usuario a la vez. Para simplificar, la gestión de errores es muy rudimentaria. Sería recomendable distinguir entre los diferentes errores que pueden ocurrir y mostrar un mensaje de error diferente según el error.

3. Edite backend/src/app.js e importe la función userRoutes:

```
importar { postRoutes } desde './routes/posts.js'  
importar { userRoutes } desde './routes/users.js'
```

4. En el mismo archivo, llame a la función userRoutes después de la función postRoutes para montarlas:

```
postRoutes(aplicación)  
userRoutes(aplicación)
```

5. Asegúrese de que el contenedor dbserver se esté ejecutando en Docker.

6. Inicie el backend ejecutando el siguiente comando en una terminal dentro de la carpeta backend/:

```
$ cd backend  
$ npm ejecuta dev
```

7. Ahora, realiza una solicitud a la nueva ruta POST /api/v1/user/signup. Verás que la creación de un usuario funciona si se proporcionan correctamente los valores de nombre de usuario y contraseña. Introduce el siguiente código en la consola de tu navegador mientras el backend se está ejecutando, en una pestaña en blanco o en <http://localhost:3001/>:

```
const res = await fetch('http://localhost:3001/api/v1/usuario/  
inscribirse', {  
    método: 'POST',  
    encabezados: { 'Content-Type': 'application/json' },  
    cuerpo: JSON.stringify({ nombre de usuario: 'dan', contraseña: 'hunter2' })  
})  
console.log(await res.json())
```

8. Si intentamos crear otro usuario con el mismo nombre de usuario (ejecutando la misma búsqueda nuevamente), fallará porque el campo de nombre de usuario está definido para ser único en Mongoose.

Ahora que hemos creado exitosamente nuestro primer usuario, continuemos creando el servicio de inicio de sesión para permitir que nuestro usuario inicie sesión.

Creando el servicio de inicio de sesión

Hasta ahora, solo hemos creado un usuario en nuestra base de datos. Como aún no lo estamos autorizando, aún no hemos trabajado con JWT. Comencemos con eso ahora:

1. Abra una nueva terminal e instale la biblioteca jsonwebtoken, que contiene funciones para tratar con la creación y verificación de JWT:

```
$ cd backend  
$ npm install jsonwebtoken@9.0.2
```

2. Edite el archivo backend/src/services/users.js e importe jwt desde la biblioteca jsonwebtoken:

```
importar jwt desde 'jsonwebtoken'
```

3. Defina una nueva función loginUser, que acepta un nombre de usuario y una contraseña:

```
exportar función asíncrona loginUser({ nombre de usuario, contraseña }) {
```

4. Ahora, busque un usuario con el nombre de usuario indicado en nuestra base de datos:

```
const usuario = await Usuario.findOne({ nombre de usuario })
si (!usuario) {
    arrojar nuevo Error('¡nombre de usuario inválido!')
}
```

5. \$en, use bcrypt.compare para comparar la contraseña ingresada con la contraseña en hash de la base de datos:

```
const isPasswordCorrect = await bcrypt.compare(contraseña, usuario.
contraseña)
si (!esContraseñaCorrecta) {
    lanzar nuevo Error('¡contraseña inválida!')
}
```

6. Si el usuario introduce correctamente su nombre de usuario y contraseña, usamos jwt.sign() para crear un nuevo JWT y firmarlo con un secreto. Para el secreto, usamos una variable de entorno:

```
constante token = jwt.sign({ sub: usuario._id }, proceso.env.JWT_
SECRETO, {
    caducaEn: '24h',
})
```

En el último argumento, también especificamos que nuestro token debe ser válido por 24 horas.

Nota

Usamos el ID de usuario, no el nombre de usuario, para identificar al usuario. Esto se hace para asegurar el futuro del sistema, ya que el ID de usuario es un valor inmutable. En el futuro, podríamos añadir una forma de cambiar el nombre de usuario. Sería difícil gestionar dicho cambio si siempre usáramos el nombre de usuario para identificar al usuario.

7. Por último, devolvemos el token:

```
token de retorno
{
```

8. Ahora, definimos la variable de entorno `JWT_SECRET` editando el archivo `.env`:

```
JWT_SECRET=reemplazar con secreto aleatorio
```

Asegúrate de generar un secreto JWT seguro para el entorno de producción, que nunca expongas ni uses en entornos de desarrollo ni para depuración. Si quieras volver a implementar tu aplicación en Google Cloud Run, también deberás agregar este secreto como variable de entorno.

9. También agregaremos uno a `.env.template` como ejemplo:

```
JWT_SECRET=reemplazar con secreto aleatorio
```

Después de crear con éxito un servicio de inicio de sesión para crear y firmar JWT, podemos crear la ruta de inicio de sesión.

Creando la ruta de inicio de sesión

Aún necesitamos exponer el servicio de inicio de sesión como una ruta API para que los usuarios puedan iniciar sesión. Hagámoslo ahora:

1. Edite el archivo `backend/src/routes/users.js` e importe la función `loginUser`:

```
importar { createUser, loginUser } desde '../services/users.js'
```

2. Agregue una nueva ruta POST `/api/v1/user/login` dentro de la función `userRoutes`, donde Llamamos a la función `loginUser` y devolvemos el token:

```
app.post('/api/v1/usuario/inicio de sesión', async (req, res) => {
  try {
    { const token = await loginUser(req.body) return
      res.status(200).send({ token }) } catch (err) {

      devolver res.status(400).send({
        Error: 'Error de inicio de sesión, ¿ingresó la información correcta?
        '¿nombre de usuario/contraseña?'
      })
    }
  }
})
```

3. Si el backend ya no se ejecuta, inícielo nuevamente. \$en, realice una solicitud a `/api/v1/ user/ login` para probarlo ingresando el siguiente código en la consola de su navegador:

```
const res = await fetch('http://localhost:3001/api/v1/usuario/ inicio de sesión', {
  método: 'POST',
  encabezados: { 'Content-Type': 'application/json' }, cuerpo:
  JSON.stringify({ nombre de usuario: 'dan', contraseña: 'hunter2'
```

```

    })
  })
  consola.log(await res.json())
}

```

4. ¡Hemos creado un JWT válido! Para verificar su validez, podemos pegarlo en el depurador en <https://jwt.io/>. Asegúrate de cambiar también el secreto en la sección "Verificar Firma" de la página, como se muestra en la siguiente captura de pantalla:

Encoded PASTE A TOKEN HERE

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYOUT: DATA

```
{
  "sub": "647dad52623ec8a2bc1917ed",
  "iat": 1685961994,
  "exp": 1686048394
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  replace-with-random-s
) □ secret base64 encoded
```

SHARE JWT

Signature Verified

Figura 6.3 – Verificación del JWT creado desde el servicio de inicio de sesión

Nota

Al copiar el token de la respuesta JSON en su navegador, asegúrese de estar copiando el valor de la cadena completa y no el truncado (con ... en el medio de la cadena). De lo contrario, es posible que el JWT no se decodifique correctamente en el depurador.

Después de iniciar sesión exitosamente con nuestro usuario y crear un token para él, ahora podemos proteger ciertas rutas y asegurarnos de que solo los usuarios que hayan iniciado sesión puedan acceder a ellas.

Definición de rutas autenticadas

Ahora que hemos creado correctamente un JWT válido, podemos empezar a proteger las rutas. Para ello, usaremos la biblioteca express-jwt, como se indica a continuación:

1. Instale el paquete npm express-jwt:

```
$ cd backend  
$ npm install express-jwt@8.4.1
```

2. Cree una nueva carpeta llamada backend/src/middleware. Dentro de ella, cree un nuevo archivo backend/src/middleware/jwt.js le e importe expressjwt allí:

```
importar { expressjwt } desde 'express-jwt'
```

3. Cree y exporte un middleware requireAuth utilizando la función expressjwt y su configuración secreta y de algoritmo:

```
exportar const requireAuth = expressjwt({  
    secreto: () => proceso.env.JWT_SECRET,  
    algoritmos: ['HS256'],  
})
```

Necesitamos usar una función para el secreto, ya que dotenv aún no se inicializa al importar, por lo que la variable de entorno solo estará disponible más adelante. Es necesario especificar los algoritmos para evitar posibles ataques de degradación.

4. Edite backend/src/routes/posts.js e importe el middleware requireAuth:

```
importar { requireAuth} desde '../middleware/jwt.js'
```

5. Agregue el middleware a la ruta creada. En Express, el middleware se puede agregar a rutas específicas. pasándolo como segundo argumento a la función, de la siguiente manera:

```
aplicación.post('/api/v1/posts', requireAuth, async (req, res) => {
```

6. Repita lo mismo para la ruta de edición:

```
aplicación.patch('/api/v1/posts/:id', requireAuth, async (req, res) => {
```

7. Ahora, haz esto para la ruta de eliminación:

```
aplicación.delete('/api/v1/posts/:id', requireAuth, async (req, res) => {
```

8. Intenta acceder a las rutas sin iniciar sesión. Verás que fallan con un estado 401 " No autorizado" . Ejecuta el siguiente código en la consola de tu navegador:

```
const res = await fetch('http://localhost:3001/api/v1/posts', {
  método: 'POST',
  encabezados: {
    'Tipo de contenido': 'application/json'
  },
  cuerpo: JSON.stringify({ título: 'Publicación de prueba' })
})
console.log(await res.json())
```

Puedes ver los resultados de la ejecución del código en la siguiente captura de pantalla:

```
> fetch('http://localhost:3001/api/v1/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ title: 'Test Post' })
})
.then(res => res.json())
.then(console.log)
<-- ▶ Promise {<pending>}

✖ ▶ POST http://localhost:3001/api/v1/posts 401 (Unauthorized) VM899:1 ⓘ
✖ ▶ Uncaught (in promise) SyntaxError: Unexpected token '<', "<!DOCTYPE "... is not valid JSON" VM900:1

> fetch('http://localhost:3001/api/v1/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2NDdkYWQ1MjYyM2VjOGEyYmMxOTE
3ZWQiLCJpYXQiOjE2ODU5NjE5OTQsImV4cCI6MTY4NjA0ODM5NH0.PM2y4aqcGhvVAHUhfoxsafn
0GF8-xYtPN1q9L_0snX4',
  },
  body: JSON.stringify({ title: 'Test Post' })
})
.then(res => res.json())
.then(console.log)
<-- ▶ Promise {<pending>}

▶ {title: 'Test Post', tags: Array(0), _id: '647dd5a928f9824089ee712f', creat
edAt: '2023-06-05T12:31:37.512Z', updatedAt: '2023-06-05T12:31:37.512Z', ...}
```

Figura 6.4 – Intento de acceder a una ruta protegida sin un JWT y luego con un JWT

Nota

En lugar de utilizar la biblioteca express-jwt, también podríamos extraer manualmente el token del encabezado de autorización y usar la función jwt.verify del jsonwebtoken. biblioteca para verificarlo.

Las rutas \$e ahora están protegidas, pero no consideramos qué usuario accedió a ellas. Hagámoslo ahora accediendo al usuario conectado desde el token.

Acceder al usuario actualmente conectado

Tras añadir rutas autenticadas, logramos proteger algunas rutas para que solo puedan acceder a ellas los usuarios registrados. Sin embargo, aún es posible editar las publicaciones de otros usuarios o crearlas con un nombre de usuario diferente. Vamos a cambiar esto:

1. Edite el archivo backend/src/services/posts.js y agregue un argumento userId a la función createPost, eliminando al autor del objeto:

```
exportar función asíncrona createPost(userId, { título, autor,  
contenidos, etiquetas }) {
```

2. En lugar de configurar el autor a través del cuerpo de la solicitud, configuraremos el autor con el ID del usuario conectado:

```
const post = new Post({ título, autor: userId, contenidos, etiquetas })
```

3. Ajustamos las funciones updatePost y deletePost de manera similar (agregando el userId argumento, eliminando el argumento del autor y eliminando la variable del autor del objeto \$set), asegurando que el usuario que ha iniciado sesión actualmente sea el autor de la publicación:

```
exportar función asíncrona updatePost(userId, postId, { título,  
autor, contenidos, etiquetas }) {  
    devolver esperar Post.findOneAndUpdate(  
        { _id: postId, autor: usuarioid },  
        { $set: { título, autor, contenido, etiquetas } },  
        { nuevo: verdadero },  
        )  
    }  
  
exportar función asíncrona deletePost(userId, postId) {  
    retorno en espera Post.deleteOne({ _id: postId, autor: userId })  
}
```

En nuestro caso, simplemente recuperaremos una publicación con el ID dado y un autor como usuario actual. Podríamos extender este código para recuperar primero la publicación con el ID dado, comprobar si existe (de no ser así, devolver un error 404 "No encontrado ") y, si existe, verificar que el autor sea el usuario conectado (de no ser así, devolver un error 403 "Prohibido ").

Nota

\$is es un cambio importante en la API y requiere modificar las pruebas. Para abreviar, no explicaremos cómo ajustar las pruebas paso a paso, así que esto es solo un ejercicio.

4. Edite el archivo backend/src/routes/posts.js y use la variable req.auth.sub

Para pasar el ID del usuario a la función createPost:

```
const post = await createPost(req.auth.sub, req.body)
```

5. Haga lo mismo para la función updatePost:

```
const post = await updatePost(req.auth.sub, req.params.id, req.body)
```

6. Además, haga lo siguiente para la función deletePost:

```
const { removedCount } = await deletePost(req.auth.sub, req.  
parámetros.id)
```

7. Intenta crear una nueva publicación; verás que la crea el usuario identificado en el JWT. Puedes hacerlo ejecutando el siguiente código en la consola del navegador (no olvides reemplazar <TOKEN> con el JWT generado previamente):

```
const res = await fetch('http://localhost:3001/api/v1/posts', {  
    método: 'POST',  
    encabezados: {  
        'Tipo de contenido': 'application/json',  
        'Autorización': 'Portador <TOKEN>'  
    },  
    cuerpo: JSON.stringify({ título: 'Publicación de prueba' })  
})  
console.log(await res.json())
```

También es posible editar y eliminar tus publicaciones, ¡pero ya no para las publicaciones de otros usuarios!

Información

El middleware \$e express-jwt almacena todas las reclamaciones decodificadas del JWT en un req.auth objeto. Por lo tanto, podemos acceder a cualquier declaración realizada al crear nuestro JWT aquí. Por supuesto, el middleware valida primero la firma del JWT con el secreto definido para garantizar que se recibió un JWT auténtico.

Ahora que hemos configurado las rutas de inicio de sesión, registro y autenticación, continuemos integrando el inicio de sesión y el registro en la interfaz.

Integración del inicio de sesión y el registro en el frontend usando React Enrutador y JWT

Ahora que hemos implementado correctamente la autorización en el backend, comenzemos a extender el frontend con páginas de registro e inicio de sesión y a conectarlas al backend. Primero, aprenderemos a implementar varias páginas en una aplicación React usando React Router. A continuación, implementaremos la interfaz de registro y la conectaremos al backend. Posteriormente, implementaremos una interfaz de inicio de sesión, almacenaremos el token en el frontend y configuraremos redirecciones automáticas al iniciar sesión correctamente. Finalmente, actualizaremos el código para crear publicaciones para pasar el token en el encabezado de autorización y acceder correctamente a nuestra ruta autenticada.

Comencemos con la integración del frontend configurando React Router.

Uso de React Router para implementar múltiples rutas

React Router es una biblioteca que nos permite gestionar el enrutamiento en nuestra aplicación definiendo varias páginas en diferentes rutas, igual que hicimos en Express para las rutas de la API, ¡pero para el frontend! Configuremos React Router:

1. Instale la biblioteca react-router-dom en el proyecto frontend (la raíz de la carpeta ch6, no dentro de la carpeta backend):

```
$ npm install react-router-dom@6.21.0
```

2. Edite src/App.jsx e importe la función createBrowserRouter y

Componente RouterProvider:

```
importar { createBrowserRouter, RouterProvider } desde 'react-router-dom'
```

3. Cree un nuevo enrutador y defina las rutas. Primero, definiremos una ruta de índice para la renderización.

Nuestro componente Blog:

```
enrutador constante = crearRouterBrowser([
  {
```

```
        ruta: '/', elemento:  
        <Blog />, },  
  
    ])
```

4. \$en, en el componente App, reemplace el componente <Blog> con <RouterProvider>, como sigue:

```
función de exportación App() {  
    devolver (  
        <QueryClientProvider cliente={queryClient}>  
            <RouterProvider router={enrutador} />  
            </ProveedorClienteConsulta>  
    )  
}
```

5. Inicie el frontend ejecutando el siguiente comando en la raíz de la carpeta ch6:

```
$ npm ejecuta dev
```

6. El blog debería visualizarse de la misma manera que antes, pero ahora, ¡podemos comenzar a definir nuevas rutas!

Puedes verificar que React Router funciona accediendo a una página que no definimos, por ejemplo, <http://localhost:5173/test>. React Router mostrará la página 404 predeterminada, como se muestra en la siguiente captura de pantalla:



Unexpected Application Error!

404 Not Found

Hey developer 🤦

You can provide a way better UX than this when your app throws errors by providing your own `ErrorBoundary` or `errorElement` prop on your route.

Figura 6.5 – La página 404 predeterminada proporcionada por React Router

Ahora que hemos configurado correctamente React Router, podemos pasar a crear la página de registro.

Creando la página de registro

Comenzaremos actualizando nuestra estructura de carpetas para que admita varias páginas. \$en, implementaremos un componente Signup y definiremos una ruta /signup para enlazarlo. Siga estos pasos:

1. Cree una nueva carpeta src/pages/.
2. Mueva el archivo src/Blog.jsx a la carpeta src/pages/. Cuando VS Code le solicite actualizar todas las importaciones, seleccione Sí. También puede actualizar la importación en src/App.jsx, como se indica a continuación:

```
importar { Blog } desde './pages/Blog.jsx'
```

3. Cree un nuevo archivo src/api/users.js y defina una función API para la ruta de registro. como sigue:

```
export const signup = async ({ nombre de usuario, contraseña }) => {
  const res = await fetch(`${import.meta.env.VITE_BACKEND_URL}/
  usuario/registrar`, {
    método: 'POST',
    encabezados: { 'Content-Type': 'application/json' },
    cuerpo: JSON.stringify({ nombre de usuario, contraseña }),
  })
  si (!res.ok) arroja nuevo Error('no se pudo registrar')
  devolver esperar res.json()
}
```

Aquí estamos comprobando res.ok, que será falso cuando el código de estado de respuesta sea un código de error, como 400.

4. Cree un nuevo archivo src/pages/Signup.jsx, importe los ganchos useState, useMutation y useNavigate de react-router-dom, así como la función signup, y defina allí un componente Signup:

```
importar { useState } desde 'react'
importar { useMutation } desde '@tanstack/react-query'
importar { useNavigate } desde 'react-router-dom'

importar { signup } desde '../api/users.js'

función de exportación Signup() {
```

5. En este componente, primero creamos ganchos de estado para los campos de nombre de usuario y contraseña:

```
const [nombreusuario, setUsername] = useState('')
const [contraseña, establecerContraseña] = useState("")
```

6. \$en, usamos el gancho useNavigate para obtener una función para navegar a una ruta diferente:

```
constante navegar = usarNavigate()
```

7. También definimos un gancho useMutation para enviar la solicitud de registro. Si la solicitud se realiza correctamente, navegamos a la ruta /login, que definiremos próximamente:

```
const signupMutation = useMutation({ mutationFn: () =>
  signup({ nombre de usuario, contraseña }), onSuccess: () => browse('/login'),
  onError: () => alert('¡Error al registrarse!'), })
```

En caso de error, también podríamos usar el estado signupMutation.isError y la respuesta del backend para mostrar un mensaje de error con un formato más agradable.

8. Luego, definimos una función para gestionar el envío del formulario, como hicimos para el componente CreatePost:

```
const handleSubmit = (e) => { e.preventDefault()
  signupMutation.mutate()

}
```

9. Ahora, creamos un formulario simple para ingresar un nombre de usuario, una contraseña y un botón para enviar la solicitud, similar al componente CreatePost:

```
devolver
  ( <formulario onSubmit={handleSubmit}>
    <div>
      <label htmlFor='create-username'>Nombre de usuario: </label> <input type='text'

      nombre='crear-nombre-de-usuario'
      id='crear-nombre-de-usuario'
      valor={nombreusuario}
      onChange={(e) => setUsername(e.target.value)} /> </div> <br />

    <div>
      <label htmlFor='create-password'>Contraseña: </label> <input type='password'

      name='create-password'
      id='create-password'
```

```

        valor={contraseña}
        onChange={(e) => setPassword(e.target.value)} /> </div> <br />

        <entrada
          tipo='enviar'
          valor={signupMutation.isPending ? 'Registrando...' : 'Registrarse'} deshabilitado={!usuario || !
        contraseña || signupMutation.isPending} /> </form>

      )
    }
  )
}

```

10. Edite src/App.jsx e importe el componente de la página de registro:

```
importar { Signup } desde './pages/Signup.jsx'
```

11. Agregue una nueva ruta /signup que apunte al componente de la página de registro:

```

enrutador constante = crearRouterBrowser([
  {
    ruta: '/',
    elemento:
      <Blog />, },
    {
      ruta: '/signup',
      elemento:
        <Signup />, },
  ]
)

```

Después de definir la página de registro, aún necesitamos una forma de enlazarla. Añadimos el enlace ahora.

Vinculación a otras rutas mediante el componente Enlace

Ahora que tenemos varias páginas en nuestra aplicación de blog, necesitamos enlazarlas. Para ello, podemos usar el componente Link de React Router. También podríamos usar un enlace normal con ``, pero eso provocaría una actualización completa de la página. El componente `$e Link` utiliza enrutamiento del lado del cliente y, por lo tanto, evita una actualización completa de la página. En su lugar, renderiza inmediatamente el nuevo componente en el lado del cliente.

Siga estos pasos para crear un enlace desde la página de índice a la página de registro:

1. Cree un nuevo archivo `src/components/Header.jsx` e importe el componente `Link` Desde `react-router-dom`:

```
importar { Enlace } desde 'react-router-dom'
```

2. Defina un componente y devuelva el componente `Enlace` para definir un enlace a la ruta de registro. como sigue:

```
función de exportación Header() {  
    devolver (  
        <div>  
            <Link to='/signup'>Regístrate</Link> </div>  
  
    )  
}
```

3. Edite `src/pages/Blog.jsx` e importe el componente `Encabezado`:

```
importar { Header } desde '../components/Header.jsx'
```

4. \$en, renderiza el componente `Encabezado` en el componente `Blog`:

```
devolver  
( <div estilo={{ relleno: 8 }}>  
    <Encabezado />  
    <br />  
    <hr />  
    <br />  
    <CrearPublicación />
```

5. Edite `src/pages/Signup.jsx` e importe el componente `Enlace`:

```
importar { useNavigate, Link } desde 'react-router-dom'
```

6. Agregue el componente `Enlace` para vincular a la página de índice:

```
devolver  
( <formulario onSubmit={handleSubmit}>  
    <Link to='/'>Volver a la página principal</Link> <hr /> <br />
```

Ahora que hemos vinculado exitosamente nuestra página de registro, continuemos creando la página de inicio de sesión.

Creación de la página de inicio de sesión y almacenamiento del JWT

Ahora que hemos definido correctamente la página de registro, podemos crear la página de inicio de sesión. Sin embargo, primero debemos encontrar una forma de almacenar el JWT. No deberíamos almacenarlo en el almacenamiento local, ya que un posible atacante podría robar el token desde allí (por ejemplo, mediante inyección de script). En una aplicación de página única (SPA), donde no hay recargas de página, una forma segura y sencilla de almacenar el token es hacerlo en tiempo de ejecución mediante un contexto de React. Hagámoslo ahora:

1. Crea una nueva carpeta `src/contexts/`. Dentro, crea un archivo `src/contexts/AuthContext.jsx` e importa las funciones `createContext`, `useState` y `useContext` desde React:

```
importar { createContext, useState, useContext } de 'react' importar PropTypes de 'prop-types'
```

2. Ahora, define el siguiente contexto:

```
exportar const AuthContext = createContext({ token: null, setToken: () => {}, })
```

3. A continuación, defina un componente `AuthContextProvider` que proporcione el contexto con un gancho de estado:

```
exportar const AuthContextProvider = ({ hijos }) => {
  const [token, setToken] = useState(null)
  devolver (
    <AuthContext.Provider valor={{ token, setToken }}> {hijos}

    </AuthContext.Provider>
  )
}

AuthContextProvider.propTypes = {
  hijos: PropTypes.element.isRequired,
}
```

4. Además, define un gancho para usar el contexto con una API similar a `useState`:

```
función de exportación useAuth() { const
  { token, setToken } = useContext(AuthContext) return [token, setToken]
}
```

5. Edite src/App.jsx e importe AuthContextProvider:

```
importar { AuthContextProvider } desde './contexts/AuthContext.jsx'
```

6. Envuelva RouterProvider con AuthContextProvider para que esté disponible para todas las páginas:

```
<Proveedor de contexto de autenticación>
<RouterProvider router={enrutador} />
Proveedor de contexto de autenticación
```

7. Edite src/api/users.js y defina una nueva función de inicio de sesión:

```
export const login = async ({ nombre de usuario, contraseña }) => {
  const res = await fetch(`$import.meta.env.VITE_BACKEND_URL}/usuario/inicio de sesión`, {
    método: 'POST',
    encabezados: { 'Content-Type': 'application/json' }, cuerpo: JSON.stringify({ nombre
    de usuario, contraseña }), }) si (!res.ok) arrojar nuevo Error('no se pudo iniciar

    sesión') devolver esperar res.json()

}
```

8. Copia el archivo src/pages/Signup.jsx a un nuevo archivo src/pages/Login.jsx y ajusta la importación y el nombre del componente. Además, agrega una nueva importación para el gancho useAuth:

```
importar { login } desde '../api/users.js' importar { useAuth } desde
'../contexts/AuthContext.jsx'

función de exportación Login() {
```

9. A continuación, edite src/pages/Login.jsx, agregue el gancho useAuth, ajuste signupMutation para llamar al inicio de sesión, configure el token y navegue a la página de índice después de iniciar sesión correctamente:

```
constante [, setToken] = useAuth()

const loginMutation = useMutation({ mutationFn: () =>
  login({ nombre de usuario, contraseña }), onSuccess: (datos) => {

    setToken(datos.token)
    browse('/')
  }, onError:
  () => alert('¡Error al iniciar sesión!'), })
```

```
constante handleSubmit = (e) => {
  e.preventDefault()
```

```
    loginMutation.mutate()
}
```

10. Ajuste el botón de envío, de la siguiente manera:

```
<entrada
  tipo='enviar'
  valor={loginMutation.isPending ? 'Iniciando sesión...' : 'Iniciar sesión
En'}
  deshabilitado={!nombre de usuario || !contraseña || loginMutation.
está pendiente}
/>
```

11. Edite src/App.jsx e importe la página de inicio de sesión:

```
importar { Login } desde './pages/Login.jsx'
```

12. Por último, defina la ruta /login, de la siguiente manera:

```
{
  ruta: '/login',
  elemento: <Login />,
},
```

Con esto, nuestras páginas de registro e inicio de sesión funcionan correctamente, pero aún necesitamos enlazar a la página de inicio de sesión y mostrar el usuario conectado en la página de inicio. Hagámoslo ahora.

Usando el JWT almacenado e implementando un cierre de sesión simple

En esta sección, comprobaremos si el usuario ya ha iniciado sesión verificando si hay un JWT válido almacenado en el contexto. \$en, usaremos el gancho de contexto de autenticación para cerrar la sesión del usuario simplemente eliminando el token. \$is no es un cierre de sesión completo, ya que el JWT sigue siendo técnicamente válido. Para un cierre de sesión completo, tendríamos que invalidar el token en el backend (por ejemplo, poniéndolo en la lista negra de la base de datos del servicio de autenticación). \$is se denomina revocación de token.

Comencemos a usar el JWT almacenado e implementemos un cierre de sesión simple:

1. Instale la biblioteca jwt-decode en la raíz de nuestro proyecto (el frontend):

```
$ npm install jwt-decode@4.0.0
```

2. Edite src/components/Header.jsx e importe la función jwtDecode y el Gancho useAuth:

```
importar { jwtDecode } desde 'jwt-decode'
importar { useAuth } desde '../contextos/AuthContext.jsx'
```

3. Obtenga el token del gancho useAuth en el componente Encabezado:

```
función de exportación Header() {
  const [token, setToken] = useAuth()
```

4. Añada una comprobación para comprobar si el token está configurado correctamente. De ser así, analice el token y represente el ID de usuario a partir de él:

```
si (token) {
  constante { sub } = jwtDecode(token)
  devolver (
    <div>
      Inició sesión como <b>{sub}</b>
```

Nota

En este caso, solo decodificamos el token en un lugar. Si esta funcionalidad se utiliza en varios lugares, sería lógico abstraer la decodificación en un gancho independiente.

5. Además, mostraremos un botón para cerrar sesión aquí, que simplemente restablece el token:

```
<br />
<button onClick={() => setToken(null)}>Cerrar sesión</button>
</div>
)
}
```

6. Mientras estamos en ello, también agreguemos un enlace a la página de inicio de sesión en el encabezado, si el usuario aún no ha iniciado sesión:

```
devolver (
  <div>
    <Link to='/login'>Iniciar sesión</Link> | <Link to='/signup'>Registrarse
    Arriba</Link>
  </div>
)
```

¡Felicitaciones! Hemos implementado con éxito un flujo de autenticación de usuarios JWT simple. Sin embargo, habrás notado que todos los usuarios de nuestro blog aparecen con su ID de usuario, no con su nombre de usuario. Cambiemos eso.

Obteniendo los nombres de usuario

Para mostrar los nombres de usuario en lugar de los ID de usuario, crearemos un componente de usuario que obtendrá la información del usuario desde un endpoint en nuestro backend, que crearemos ahora. Por ahora, solo mostraremos el nombre de usuario, pero en el futuro, esta función podría utilizarse para obtener otra información, como el avatar o el nombre completo del usuario.

Implementación del punto final del backend

Comencemos implementando el punto final del backend para obtener información del usuario:

1. Edite backend/src/services/users.js y agregue una nueva función para obtener la información del usuario por ID.

Como alternativa, devolvemos el ID del usuario si no encontramos un usuario coincidente:

```
exportar función asíncrona getUserInfoById(userId) {
    intentar {
        const usuario = await Usuario.findById(userId)
        si (!usuario) devuelve { nombre de usuario: userId }
        devolver { nombre de usuario: usuario.nombredeusuario }
    } atrapar (err) {
        devolver { nombre de usuario: userId }
    }
}
```

¡Nos aseguramos específicamente de devolver solo el nombre de usuario aquí, para evitar filtrar la contraseña u otra información confidencial del usuario!

2. Edite backend/src/routes/users.js e importe allí la función recién definida:

```
importar { createUser, loginUser, getUserInfoById } desde '../
servicios/usuarios.js'
```

3. \$en, define una nueva ruta dentro de la función userRoutes, que obtendrá un usuario con un ID específico. Para esta ruta, usamos el plural "usuarios", ya que aquí se trata con varios usuarios:

```
aplicación.get('/api/v1/users/:id', async (req, res) => {
    const userInfo = espera getUserInfoById(req.params.id)
    devolver res.status(200).send(userInfo)
})
```

4. Como ya estamos trabajando en el backend, modifiquemos también el filtro de autor existente para que funcione con nombres de usuario. Editemos backend/src/services/posts.js e importemos el modelo de usuario:

```
importar { Usuario } desde '../db/models/user.js'
```

5. Refactorice la función listPostsByAuthor encontrando un usuario con el nombre de usuario dado.

Luego enumera todas las publicaciones por ID de usuario (si se encontró uno):

```
exportar función asíncrona listPostsByAuthor(nombreUsuarioAutor, opciones) {

    const usuario = await Usuario.findOne({ nombre de usuario: nombreDeUsuarioAutor })
    si (!usuario) devuelve []
    devolver lista de esperaPosts({ autor: usuario._id }, opciones)
}
```

Ahora que tenemos un punto final que devuelve información de usuario para un ID de usuario determinado, ¡usémoslo en el frontend!

Implementar un componente de usuario para obtener y representar el nombre de usuario

En el frontend, crearemos un componente que obtendrá y renderizará el nombre de usuario. React Query nos ayuda mucho en este caso, ya que no tenemos que preocuparnos por obtener los mismos ID de usuario varias veces: almacenará el resultado en caché y lo devolverá al instante, en lugar de realizar otra solicitud.

Siga estos pasos para implementar un componente de Usuario:

1. Primero, necesitamos definir la función de la API. Edite src/api/users.js y agregue una función a

obtener la información del usuario por id:

```
exportar const getUserInfo = async (id) => {
    const res = await fetch(`${import.meta.env.VITE_BACKEND_URL}/
    usuarios/${id}`, { método:
        'GET', encabezados:
        { 'Content-Type': 'application/json' }, }) devolver await res.json()

}
```

2. Cree un nuevo archivo src/components/User.jsx e importe useQuery, PropTypes, y la función API:

```
importar { useQuery } desde '@tanstack/react-query' importar PropTypes
desde 'prop-types' importar { getUserInfo } desde
'./api/users.js'
```

3. Ahora, defina el componente y obtenga la información del usuario a través del gancho de consulta:

```
función de exportación Usuario({ id }) {
    const userInfoQuery = useQuery({ queryKey:
        ['usuarios', id], queryFn: () =>
        getUserInfo(id), }) const userInfo =
        userInfoQuery.data ?? {}
```

4. Representamos el nombre de usuario si está disponible y, en caso contrario, recurrimos al ID:

```
devolver <strong>{userInfo?.nombreusuario ?? id}</strong>
{}
```

5. Por último, definimos los tipos de propiedades para el componente:

```
Usuario.propTypes = {  
  id: PropTypes.string.isRequired,  
}
```

6. Ahora, podemos hacer uso del componente recién creado e importarlo en src/components/

Encabezado.jsx:

```
importar { Usuario } desde './User.jsx'
```

7. \$en, podemos editar el código existente para renderizar el componente Usuario en lugar de renderizarlo directamente. el ID del usuario:

```
Inició sesión como <User id={sub} />
```

8. A continuación, repetimos el mismo proceso para src/components/Post.jsx e importamos el

Componente de usuario:

```
importar { Usuario } desde './User.jsx'
```

9. \$en, ajustamos el código para renderizar el componente Usuario:

```
Escrito por <User id={author} />
```

Ahora, nuestros nombres de usuario se representarán correctamente nuevamente, como se muestra en la siguiente captura de pantalla:

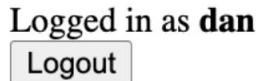
A screenshot of a user interface showing a login status message "Logged in as dan" above a "Logout" button.

Figura 6.6 – Obtención y visualización correcta del nombre de usuario

Ahora que los nombres de usuario se muestran correctamente, necesitamos hacer una cosa más: enviar el encabezado JWT al crear publicaciones.

Envío del encabezado JWT al crear publicaciones

Al crear una publicación, ya no necesitamos enviar el autor. En su lugar, necesitamos enviar el JWT con el encabezado de autenticación.

Refactoricemos el código para que podamos hacer esto:

1. Edite src/api/posts.jsx y ajuste la función createPost para que acepte un JWT como primer argumento, que luego se pasa dentro de un encabezado de autenticación:

```
exportar const createPost = async (token, post) => {
    const res = await fetch(`.${{import.meta.env.VITE_BACKEND_URL}}/ posts`, { método: 'POST', encabezados:
    { 'Tipo de
        contenido': 'application/
        json',
        Autorización: `Portador ${token}` },
        cuerpo:
        JSON.stringify(post), }) return await res.json()

}
```

2. Edite src/components/CreatePost.jsx e importe el gancho useAuth:

```
importar { useAuth } desde './contextos/AuthContext.jsx'
```

3. Obtenga el JWT desde el gancho useAuth dentro del componente:

```
función de exportación CreatePost() { const [token] =
useAuth()
```

4. Eliminar el estado del autor:

```
const [autor, establecerAutor] = useState("")
```

5. Además, elimine el estado del autor de la función createPost y, en su lugar, pase el estado del token como primer argumento:

```
mutationFn: () => createPost(token, { título, autor, contenidos }),
```

6. Antes de renderizar el componente, verifique si el usuario ha iniciado sesión verificando si existe un token. Si el usuario no ha iniciado sesión, le decimos que inicie sesión primero:

```
si (!token) devuelve <div>Inicie sesión para crear nuevas publicaciones.</div>
```

```
devolver
( <formulario onSubmit={handleSubmit}>
```

7. Elimine el siguiente código para eliminar el campo de autor:

```
<br />
<div>
    <label htmlFor='create-author'>Autor: </label>
    <entrada
        tipo='texto'
        nombre='crear-autor'
        id='crear-autor'
        valor={autor}
        onChange={(e) => setAuthor(e.target.value)}>
    />
</div>
```

¡Ahora la creación de publicaciones vuelve a funcionar correctamente! Almacena el ID de usuario del usuario conectado en la base de datos como autor y lo resuelve como el nombre de usuario al mostrar la publicación.

A continuación, aprenderemos sobre el manejo avanzado de tokens.

Manejo avanzado de tokens

Es posible que haya notado que a nuestra solución de autenticación simple aún le faltan algunas características que una solución totalmente avanzada debería tener, como las siguientes:

- Utilizamos claves asimétricas para los tokens, lo que nos permite verificar la autenticidad (con la clave pública) sin exponer nuestro secreto (la clave privada) a todos los servicios. Hasta ahora, hemos utilizado una clave simétrica, lo que significa que necesitamos el mismo secreto para generar y verificar un JWT.
- Almacenar tokens en cookies httpOnly seguras para que se pueda acceder a ellos nuevamente, incluso cuando se actualice o se cierre la página.
- Iniciar sesión después de cerrar sesión en el backend.

Implementar estas cosas requiere mucho esfuerzo manual, por lo que se recomienda usar una solución de autenticación como Auth0 o Firebase Auth. Estas soluciones funcionan de manera similar a nuestra implementación de JWT simple, pero brindan un servicio de autenticación externo para crear y manejar los tokens por nosotros. Este capítulo tiene como objetivo presentar cómo funcionan esos proveedores detrás de escena para que pueda comprender e integrar fácilmente cualquiera de los proveedores como lo ve en sus proyectos.

Hasta ahora, todos los usuarios eran iguales: cada uno podía crear publicaciones, pero solo actualizar y eliminar las suyas. En un blog público, sería conveniente que los administradores pudieran eliminar las publicaciones de otros para moderar el contenido de la plataforma. Una buena manera de agregar roles es almacenarlos y recuperarlos de la base de datos. Si bien agregar roles en el JWT es técnicamente posible, tiene algunas desventajas, como la necesidad de invalidar los tokens existentes y crear uno nuevo cuando cambian los roles.

Resumen

En este capítulo, aprendimos a fondo el funcionamiento de los JWT. Primero, aprendimos la teoría de la autenticación y los JWT, y cómo crearlos manualmente. Luego, implementamos rutas de inicio de sesión, registro y autenticación en el backend. A continuación, integramos estas rutas en el frontend creando nuevas páginas y enrutando entre ellas con React Router. Finalmente, concluimos este capítulo aprendiendo sobre el manejo avanzado de tokens y ofreciendo consejos sobre aspectos adicionales de la autenticación y la gestión de roles.

En el siguiente capítulo, Capítulo 7, "Mejora del tiempo de carga mediante renderizado del lado del servidor", aprenderemos a implementar el renderizado del lado del servidor para mejorar el tiempo de carga inicial de nuestro blog. Ya estamos realizando muchas solicitudes en la primera carga (obteniendo todas las entradas del blog y luego los nombres de usuario de cada autor). Podemos agruparlas haciendo esto en el backend.

Machine Translated by Google

Mejorando el tiempo de carga

Uso de la renderización del lado del servidor

Tras implementar la autenticación mediante JWT, nos centraremos en optimizar el rendimiento de nuestra aplicación de blog. Comenzaremos evaluando el tiempo de carga actual de nuestra aplicación y aprenderemos sobre diversas métricas a considerar. A continuación, aprenderemos a renderizar componentes de React y a obtener datos en el servidor. Al final de este capítulo, abordaremos brevemente conceptos avanzados de renderizado del lado del servidor.

En este capítulo cubriremos los siguientes temas principales:

- Evaluación comparativa del tiempo de carga de nuestra aplicación
- Representación de componentes React en el servidor
- Obtención de datos del lado del servidor
- Representación avanzada del lado del servidor

Requisitos técnicos

Antes de comenzar, instale todos los requisitos mencionados en el Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones que aparecen en esos capítulos son las que se utilizan en este libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que algunos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas en los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub, <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/arbol/principal/capitulo7>.

El video de Se CiA para este capítulo se puede encontrar en: <https://youtu.be/0OlmicibYWQ>

Evaluación comparativa del tiempo de carga de nuestra aplicación

Antes de que podamos comenzar a mejorar el tiempo de carga, primero debemos aprender sobre las métricas para comparar el rendimiento de nuestra aplicación. Las principales métricas para medir el rendimiento de las aplicaciones web se denominan Core Web Vitals y son las siguientes:

- First Contentful Paint (FCP): mide el rendimiento de carga de una aplicación informando el tiempo transcurrido hasta que se renderiza la primera imagen o bloque de texto en la página. Un buen objetivo sería reducir esta métrica a menos de 1,8 segundos.
- Largest Contentful Paint (LCP): \$is mide el rendimiento de carga de una aplicación informando el tiempo que transcurre hasta que la imagen o el bloque de texto más grande es visible en la ventana gráfica. Un buen objetivo sería reducir esta métrica a menos de 2,5 segundos.
- Tiempo total de bloqueo (TBT): mide la interactividad de una aplicación informando el tiempo transcurrido entre el FCP y el momento en que el usuario puede interactuar con la página. Un buen objetivo sería reducir esta métrica por debajo de los 200 milisegundos.
- Cumulative Layout Shi! (CLS): mide la estabilidad visual de una aplicación al informar movimientos inesperados en la página durante la carga, como por ejemplo, un enlace que primero se carga en la parte superior de la página, pero luego se empuja hacia abajo cuando se cargan otros elementos. Si bien esta métrica no mide directamente el rendimiento real de la aplicación, sigue siendo una métrica importante a tener en cuenta, ya que puede molestar a los usuarios cuando intentan hacer clic en algo, pero el diseño cambia.

Todas estas métricas se pueden medir con la herramienta de código abierto Lighthouse , que también está disponible en Google Chrome DevTools, en el panel de Lighthouse . Comencemos a realizar evaluaciones comparativas.

Aplicación ahora:

1. Copie la carpeta ch6 a una nueva carpeta ch7, de la siguiente manera:

```
$ cp -R ch6 ch7
```

2. Abra la carpeta ch7 en VS Code, abra una terminal y ejecute el frontend con el siguiente comando:

```
$ npm ejecuta dev
```

3. Asegúrese de que el contenedor dbserver se esté ejecutando en Docker.

4. Abra una nueva terminal y ejecute el backend con el siguiente comando:

```
$ cd backend  
$ npm ejecuta dev
```

5. Vaya a <http://localhost:5173> en Google Chrome y abra el inspector (haga clic derecho y luego presione Inspeccionar).

Nota

Sería mejor hacer esto en una pestaña de incógnito para que las extensiones no interfieran con las mediciones.

6. Abra la pestaña Faro (puede que esté oculta por el menú >>). Debería verse así:

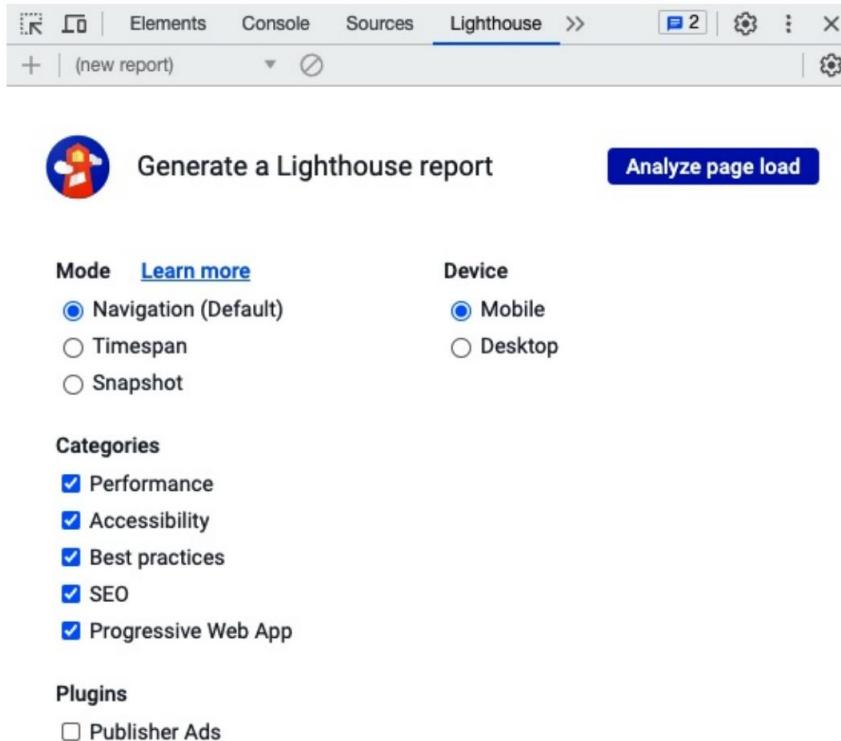


Figura 7.1 – La pestaña Lighthouse en Google Chrome DevTools

7. En la pestaña Faro , deje todas las opciones con sus configuraciones predeterminadas y haga clic en la página Analizar botón de carga .

Lighthouse comenzará a analizar el sitio web y generará un informe con métricas como el Primer Cuadro de Contenido , el Cuadro de Contenido más Grande, el Tiempo Total de Bloqueo y el Diseño Acumulado. Como podemos ver, nuestra aplicación ya tiene un buen rendimiento en cuanto a TBT y CLS, pero su rendimiento es especialmente bajo en cuanto a FCP y LCP. Vea la siguiente captura de pantalla como referencia:

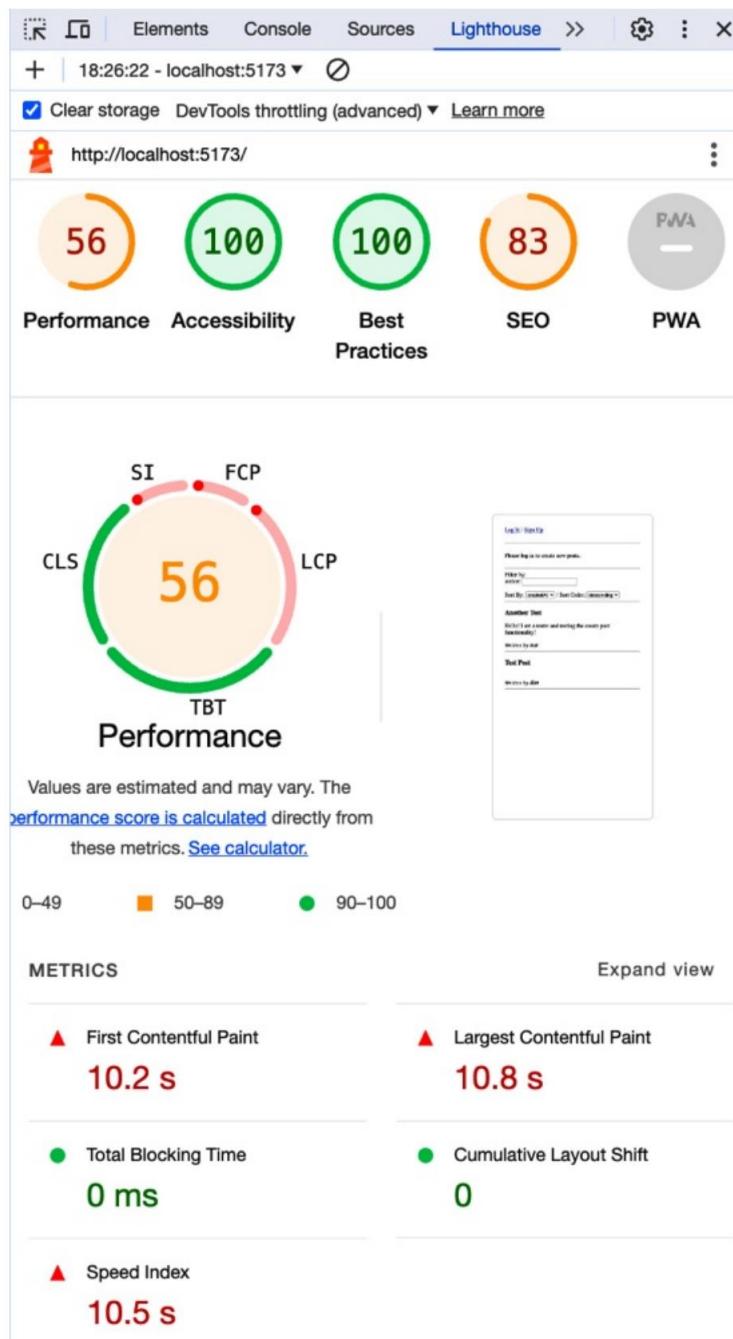


Figura 7.2 – Resultados de Lighthouse al analizar nuestra aplicación en modo de desarrollo

(mientras pasa el cursor sobre la puntuación de rendimiento)

Hay dos razones por las que la pintura tarda tanto. En primer lugar, estamos ejecutando el servidor en modo de desarrollo, lo que generalmente ralentiza todo. Además, estamos renderizando todo en el lado del cliente, lo que significa que el navegador primero debe descargar y ejecutar nuestro código JavaScript antes de poder empezar a renderizar la interfaz. Construyamos estáticamente nuestro frontend y el benchmark de nuevo:

1. Instale la herramienta de servicio globalmente con el siguiente comando, que es una herramienta que ejecuta un simple servidor web:

```
$ npm install -g serve
```

2. Construye el frontend con este comando (ejecútalo en la raíz de nuestro proyecto):

```
$ npm ejecuta la compilación
```

3. Sirva nuestra aplicación estáticamente ejecutando el siguiente comando:

```
$ servir dist/
```

4. Abra `http://localhost:3000` en Google Chrome y ejecute Lighthouse nuevamente (es posible que tenga que borrar los informes antiguos o hacer clic en la lista en la parte superior y seleccionar (nuevo informe) para analizar nuevamente).

Debería ver los resultados del nuevo benchmark en el frontend servido estáticamente, que se asemeja más a cómo se serviría en producción. Puede ver un ejemplo de los resultados en la siguiente captura de pantalla:

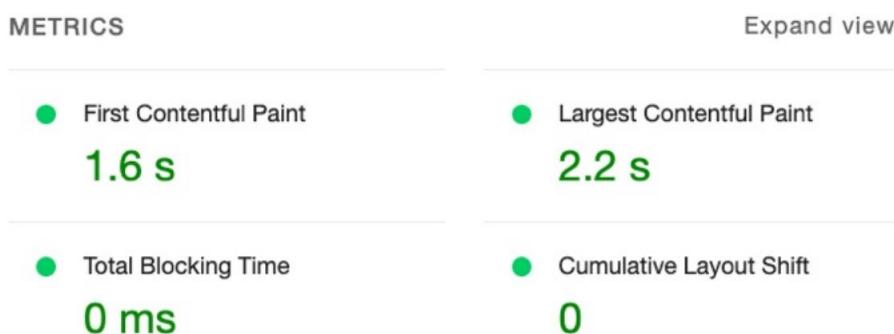


Figura 7.3 – Resultados del informe Lighthouse en nuestra aplicación creada estáticamente

¡Los resultados son bastante buenos! Sin embargo, aún se pueden mejorar. Además, Core Web Vitals no tiene en cuenta las solicitudes en cascada para obtener los nombres de usuario de los autores. Si bien las primeras y más grandes pinturas de contenido se cargan rápidamente en nuestra aplicación, los nombres de los autores ni siquiera se cargan en ese momento. Además del informe de Lighthouse, también podemos echar un vistazo a la pestaña Red para depurar aún más el rendimiento de nuestra aplicación, de la siguiente manera:

1. En DevTools, vaya a la pestaña Red .
2. Actualice la página con la pestaña abierta. Verá un diagrama de cascada y el tiempo medido.

para realizar solicitudes, como se muestra en la siguiente captura de pantalla:

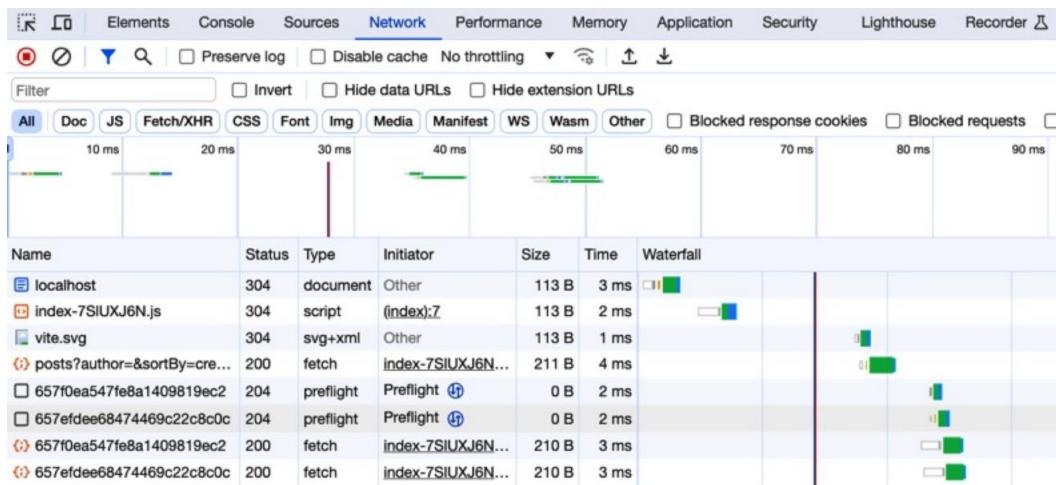


Figura 7.4 – El diagrama de cascada en la pestaña Red

Pero los tiempos son extremadamente bajos (todos por debajo de 10 ms). Esto se debe a que nuestro backend se ejecuta localmente, por lo que no hay retraso en la red. Esto no es un escenario realista. En producción, tendríamos latencia en cada solicitud, por lo que primero tendríamos que esperar a que se extraigan las entradas del blog y luego obtener los nombres de los autores por separado. Podemos usar DevTools para simular una conexión de red más lenta; hágámoslo ahora:

1. En la parte superior de la pestaña Red , haga clic en el menú desplegable Sin limitación .

2. Seleccione el ajuste preestablecido 3G lento . Consulte la siguiente captura de pantalla como referencia:

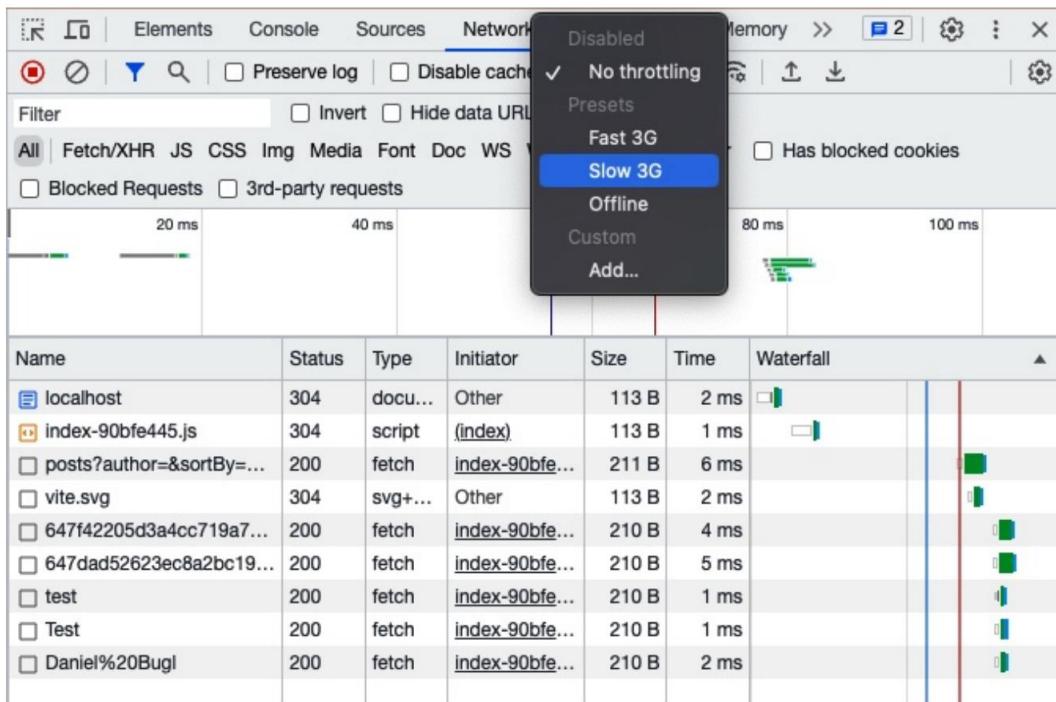


Figura 7.5 – Simulación de redes lentas en Google Chrome DevTools

Nota

Lighthouse incorpora una forma de limitación, similar a la limitación de red que usamos aquí, pero no igual. Mientras que la limitación de red en DevTools consiste en un retraso fijo que se añade a todas las solicitudes, la limitación en Lighthouse intenta simular un escenario más realista ajustándola en función de los datos observados en la carga inicial sin limitación.

3. Actualice la página. Verá que la aplicación carga lentamente el diseño principal y luego una lista de todos los elementos.

publicaciones y, finalmente, resolver los ID de autor en nombres de usuario.

Así es como se cargaría nuestra página en redes lentas. ¡Ahora, el tiempo total de carga de nuestra aplicación es de casi nueve segundos! Puedes consultar el diagrama de cascada para entender por qué ocurre esto:

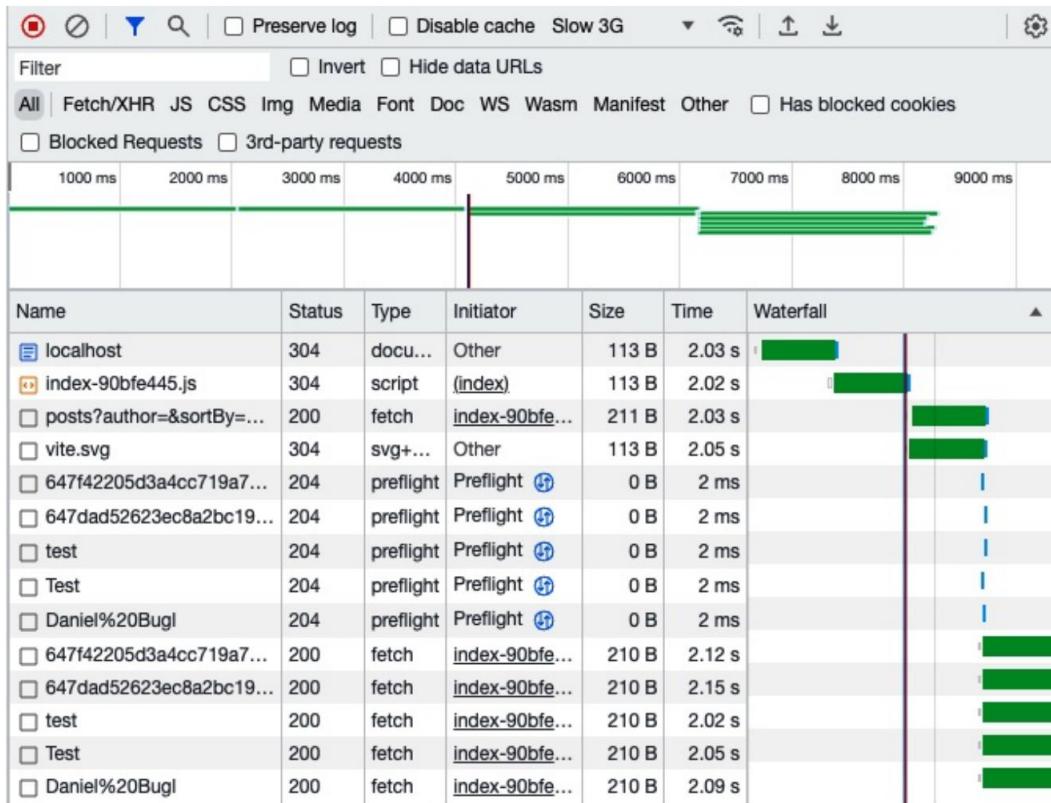


Figura 7.6 – Comprobación del diagrama de cascada con limitación lenta de 3G activada

El problema de nuestra aplicación es que las solicitudes se ejecutan en cascada. Primero, se carga el documento HTML, que a su vez carga el archivo JavaScript de nuestra aplicación. Este archivo JavaScript se ejecuta y comienza a renderizar el diseño y a obtener la lista de publicaciones. Tras cargar las publicaciones, se realizan múltiples solicitudes en paralelo para resolver los nombres de los autores. Como cada solicitud tarda poco más de dos segundos en nuestra red lenta simulada, el tiempo de carga total es de más de ocho segundos.

Ahora que hemos aprendido cómo evaluar una aplicación web y encontramos un cuello de botella en el rendimiento de nuestra aplicación (las solicitudes en cascada), ¡aprendamos cómo mejorar el rendimiento!

Representación de componentes React en el servidor

En la sección anterior, identificamos las solicitudes en cascada como el problema de nuestro bajo rendimiento en conexiones lentas. Las posibles soluciones son las siguientes:

- **Solicitudes agrupadas:** Obtener todo en el servidor y luego entregarlo todo al cliente en una sola solicitud. Si esto solucionaría las solicitudes en cascada al obtener los nombres de los autores, pero no el tiempo de espera inicial entre la carga de la página HTML y la ejecución de JavaScript para comenzar a obtener los datos. Con una latencia de dos segundos por solicitud, se añaden cuatro segundos (dos segundos para cargar el JavaScript y dos segundos para realizar la solicitud) tras obtener el HTML.
- **Renderizado del lado del servidor:** Renderiza la interfaz de usuario inicial con todos los datos en el servidor y la sirve en lugar del HTML inicial, que solo contiene una URL al archivo JavaScript. Esto significa que no se necesitan solicitudes adicionales para obtener los datos ni el JavaScript, y podemos mostrar las entradas del blog inmediatamente. Otra ventaja de este enfoque es que permite almacenar los resultados en caché, por lo que solo necesitamos regenerar la página en el servidor cuando se añade una entrada. Una desventaja es que sobrecarga el servidor, especialmente cuando las páginas son complejas de renderizar.

En casos donde los datos no cambian con tanta frecuencia o todos los usuarios acceden a los mismos datos, la renderización del lado del servidor resulta beneficiosa. En casos donde los datos cambian con frecuencia o se personalizan para cada usuario, podría ser más conveniente agrupar las solicitudes en una sola creando una nueva ruta o utilizando un sistema que pueda agregar solicitudes, como GraphQL, que aprenderemos más adelante en este libro, en el Capítulo 11, "Creación de un backend con una API GraphQL". Sin embargo, en este capítulo nos centraremos en el enfoque de renderización del lado del servidor.

Echemos un vistazo a las diferencias entre la renderización del lado del servidor y la renderización del lado del cliente:

- En la representación del lado del cliente, el navegador descarga una página HTML mínima, que, la mayoría de las veces, solo contiene información sobre dónde descargar un paquete de JavaScript, que contiene todo el código que representará la aplicación.
- En la representación del lado del servidor, los componentes de React se representan en el servidor y se sirven como HTML al navegador. Esto garantiza que la aplicación se pueda representar inmediatamente. El paquete de JavaScript se puede cargar más tarde.

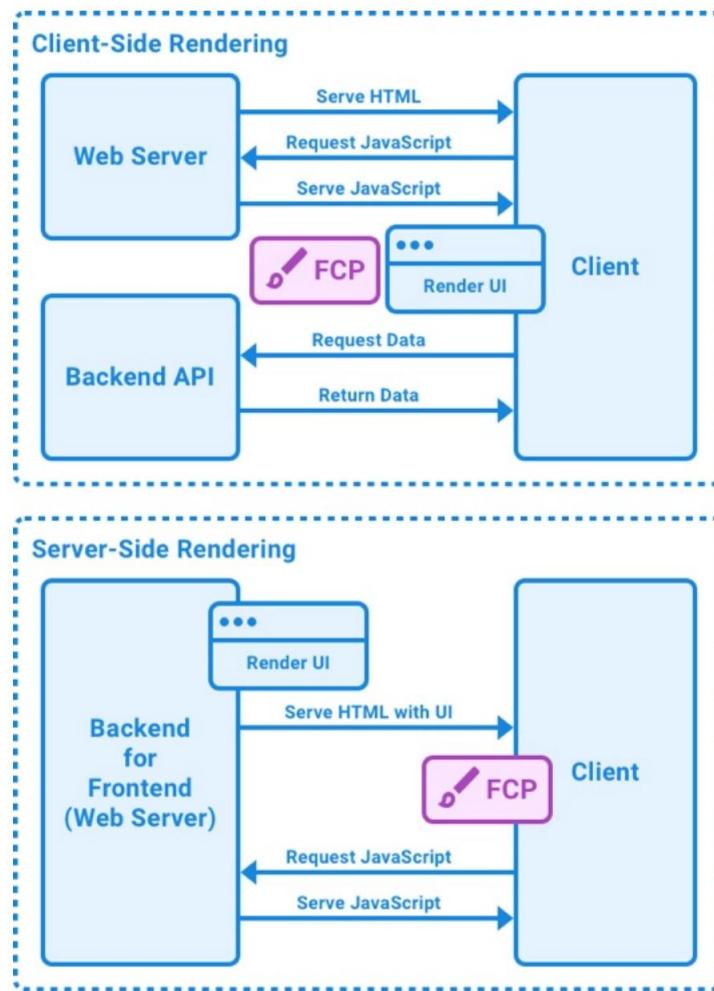


Figura 7.7 – Las diferencias entre la renderización del lado del cliente y la renderización del lado del servidor

También es posible combinar ambos en una representación isomórfica. Esto implica representar la página inicial en el servidor y luego continuar representando los cambios en el cliente. La representación isomórfica combina lo mejor de ambos mundos.

Además de las mejoras de rendimiento, la representación del lado del servidor también es buena para la optimización de motores de búsqueda (SEO), porque los rastreadores de motores de búsqueda no necesitan ejecutar JavaScript para ver la página. Aprenderemos más sobre SEO en el próximo capítulo, Capítulo 8, Cómo asegurarse de que los clientes lo encuentren con la optimización de motores de búsqueda.

Ahora que hemos aprendido sobre la renderización del lado del servidor, comenzemos a implementarla en nuestro frontend, de la siguiente manera:

- Configuración del servidor
- Definir el punto de entrada del lado del servidor
- Definir el punto de entrada del lado del cliente
- Actualización de index.html y package.json
- Hacer que React Router funcione con la renderización del lado del servidor

Comencemos configurando el servidor.

Configuración del servidor

Antes de poder comenzar con la renderización del lado del servidor, necesitamos configurar un código repetitivo para ejecutar un servidor Express en conjunto con Vite, de modo que no perdamos los beneficios de Vite, como la recarga en caliente. Sigamos estos pasos para configurar el servidor:

1. Instale las dependencias express y dotenv en la raíz de nuestro proyecto (el frontend). Las usaremos para crear un pequeño servidor web que sirva nuestra página renderizada del lado del servidor.

```
$ npm install express@4.18.2 dotenv@16.3.1
```

2. Edite .eslintrc.json y agregue el nodo env, ya que vamos a agregar el código del lado del servidor a Nuestro frontend ahora:

```
"env": {  
  "navegador": verdadero,  
  "nodo": verdadero  
},
```

3. Cree un nuevo archivo server.js en la carpeta ch7 e importe las dependencias fs, path, url, express y dotenv:

```
importar fs desde 'fs'  
ruta de importación desde 'ruta'  
importar { fileURLToPath } desde 'url'  
  
importar express desde 'express'  
importar dotenv desde 'dotenv'  
dotenv.config()
```

4. Guarde la ruta actual en una variable para usarla más adelante para hacer referencia a otros archivos en nuestro proyecto, utilizando la variable import.meta.url compatible con ESM, que contiene una URL file:// a nuestro proyecto:

```
const __dirname = ruta.dirname(fileURLToPath(import.meta.url))
```

Aquí convertimos esta URL en una ruta normal.

5. Defina una nueva función createDevServer, donde crearemos un servidor de desarrollo Vite con acceso activo.

Recarga y renderizado del lado del servidor:

```
función asíncrona createDevServer() {
```

6. Dentro de esta función, primero definimos la aplicación Express:

```
aplicación constante = express()
```

7. \$en, importa y crea un servidor de desarrollo de Vite. Usamos la sintaxis de importación dinámica para no tener que importar Vite al definir el servidor de producción posteriormente:

```
const vite = esperar (
    esperar
    importación('vite') .createServer({
        servidor: { middlewareMode: true }, tipo de aplicación:
        'personalizado', })

    aplicación.use(vite.middlewares)
```

El modo middleware ejecuta Vite como middleware en un servidor Express existente. Al configurar appType como personalizado, se desactiva la lógica de entrega de Vite para que podamos controlar el HTML que se entregará.

8. Ahora, defina una ruta que coincida con todas las rutas y comience cargando el archivo index.html:

```
aplicación.use('*', async (req, res, next) => {
    prueba
    { const templateHtml = fs.readFileSync(
        ruta.resolve(__nombredir, 'index.html'), 'utf-8',
    )
})
```

Asegúrese de cargarlo en modo UTF-8 para admitir varios idiomas y emojis en index.html.

9. A continuación, inyecte el cliente de reemplazo de módulo activo de Vite para permitir la recarga en caliente:

```
plantilla constante = await vite.transformIndexHtml(
    req.originalUrl, plantillaHtml

)
```

10. Cargue el archivo de punto de entrada para nuestra aplicación renderizada del lado del servidor, que definiremos en el siguiente paso:

```
const { render } = await vite.ssrLoadModule('/src/entry-server.jsx')
```

La función `$e ssrLoadModule` en Vite transforma automáticamente el código fuente de ESM para que pueda usarse en Node.js. Esto significa que podemos recargar en caliente el archivo de punto de entrada sin tener que ejecutar una compilación manual.

11. Renderiza la aplicación con React. Definiremos la función de renderizado más adelante, en el punto de entrada del servidor. Por ahora, simplemente llamamos a la función:

```
const appHtml = await render()
```

12. Inserte el HTML renderizado de nuestra aplicación en la plantilla HTML haciendo coincidir una cadena de marcador de posición, que definiremos más adelante en el archivo index.html:

```
const html = plantilla.replace(`<!--ssr-outlet-->`, appHtml)
```

13. Devuelve una respuesta 200 OK con el contenido HTML final:

```
res.status(200).set({ 'Tipo de contenido': 'texto/html' }).end(html)
```

14. Para finalizar la creación del servidor, capture todos los errores y deje que Vite reinicie el seguimiento de la pila, asignando los archivos fuente del seguimiento de la pila al código fuente real. \$en, devuelva la aplicación Express creada:

```
} captura (e) {
    vite.ssrFixStacktrace(e)
    siguiente(e)
}
})

aplicación de retorno
}
```

15. Por último, ejecute la función `createDevServer` y haga que la aplicación escuche en un puerto definido:

```
const app = await crearServidorDesarrollo()
aplicación.listen(proceso.env.PORT, () =>
    consola.log(
        `Servidor de desarrollo ssr ejecutándose en http://localhost:${process.env.
PUERTO}`,
        ),
    )
)
```

16. No olvidemos definir la variable de entorno PORT en el archivo .env. Edite el archivo .env, y agregue la variable de entorno PORT, de la siguiente manera:

```
VITE_BACKEND_URL="http://localhost:3001/api/v1"  
PUERTO=5173
```

Ahora que hemos creado con éxito el servidor Express con la integración de Vite, continuamos implementando el punto de entrada del lado del servidor.

Definición del punto de entrada del lado del servidor

El punto de entrada del servidor usará ReactDOMServer para renderizar nuestros componentes React en el servidor. Debemos distinguir este punto de entrada del cliente, ya que no todas las funciones de React son compatibles con el servidor. En concreto, algunos ganchos, como los de elect, no se ejecutarán en el servidor. Además, tendremos que gestionar el enrutador de forma diferente en el servidor, pero hablaremos de ello más adelante.

Ahora, comencemos a definir el punto de entrada del lado del servidor:

1. Primero, crea un nuevo archivo src/entry-server.jsx e importa ReactDOMServer y el componente App:

```
importar ReactDOMServer desde 'react-dom/server'  
importar { App } desde './App.jsx'
```

2. Defina y exporte la función de renderizado, que devuelve el componente de la aplicación utilizando el Función ReactDOMServer.renderToString:

```
exportar función asíncrona render() {  
    devuelve ReactDOMServer.renderToString(  
        <Aplicación />,  
    )  
}
```

Después de definir el punto de entrada del lado del servidor, continuaremos definiendo el punto de entrada del lado del cliente.

Definición del punto de entrada del lado del cliente

El punto de entrada del cliente usa ReactDOM normal para renderizar nuestros componentes React. Sin embargo, necesitamos que React utilice el DOM ya renderizado en el servidor. En lugar de renderizar, hidratamos el DOM existente. Como alregar las plantas, la hidratación revitaliza el DOM añadiendo toda la funcionalidad de React al DOM estático renderizado en el servidor.

Siga estos pasos para definir el punto de entrada del lado del cliente:

1. Cambie el nombre del archivo `src/main.jsx` existente a `src/entry-client.jsx`.
2. Reemplace la función `createRoot` con la función `hydrateRoot`, de la siguiente manera:

```
ReactDOM.hydrateRoot(  
  documento.getElementById('root'),  
  <React.StrictMode>  
    <Aplicación />  
  </React.StrictMode>,  
)
```

La función `$e hydrateRoot` acepta el componente como segundo argumento y no requiere que llamemos a `.render()`.

Ahora que hemos definido ambos puntos de entrada, actualicemos `index.html` y `package.json`.

Actualización de `index.html` y `package.json`

Aún necesitamos agregar la cadena de marcador de posición al archivo `index.html` y ajustar `package.json` para que ejecute nuestro servidor personalizado en lugar del comando `vite` directamente. Hagámoslo ahora:

1. Edite `index.html` y agregue un marcador de posición donde se inyectará el HTML renderizado por el servidor:

```
<div id="raíz"><!--ssr-outlet--></div>
```

2. Ajuste la importación del módulo para que apunte al punto de entrada del lado del cliente:

```
<script type="module" src="/src/entry-client.jsx"></script>
```

3. Ahora, edite `package.json` y reemplace el script `dev` con lo siguiente:

```
"dev": "servidor de nodo",
```

4. Además, reemplace el comando de compilación con comandos para compilar el servidor y el cliente:

```
"build": "npm run build:client && npm run build:server", "build:client": "vite build --outDir dist/client", "build:server": "vite build --outDir dist/server --ssr src/ entry-server.jsx",
```

Nuestra configuración ya está lista para la renderización del lado del servidor. Sin embargo, al iniciar el servidor, notarás inmediatamente que `React Router` no funciona con nuestra configuración actual. Vamos a corregirlo.

Cómo hacer que React Router funcione con la renderización del lado del servidor

Para que React Router funcione con la renderización del lado del servidor, necesitamos usar StaticRouter en el lado del servidor y BrowserRouter en el lado del cliente. Podemos reutilizar las mismas definiciones de ruta en ambos lados.

Comencemos a refactorizar nuestro código para que React Router funcione en el lado del servidor:

1. Edite src/App.jsx y elimine las importaciones relacionadas con el enrutador (las líneas resaltadas):

```
importar { QueryClient, QueryClientProvider } desde '@tanstack/ react-query' importar { createBrowserRouter,
RouterProvider } desde
'react-router-dom'

importar { AuthContextProvider } desde './contexts/AuthContext.jsx' importar { Blog } desde './pages/Blog.jsx'
importar { Registrarse } desde './pages/Signup.jsx' importar { Iniciar
sesión } desde './pages/Login.jsx'
```

2. Importa PropTypes, ya que los necesitaremos más adelante:

```
importar PropTypes desde 'prop-types'
```

3. A continuación, elimine las siguientes definiciones de ruta; las colocaremos en un nuevo archivo pronto:

```
enrutador constante = crearRouterBrowser([
  {
    ruta: '/', elemento:
    <Blog />, },
    {
      ruta: '/signup', elemento:
      <Signup />, },
      {
        ruta: '/login', elemento:
        <Login />, },
    ])
```

4. Ajuste la función para aceptar hijos y reemplace RouterProvider con {children}:

```
función de exportación App({ children }) {
  devolver (
    <QueryClientProvider cliente={queryClient}>
      <Proveedor de contexto de autenticación>
        {niños}
```

```
    Proveedor de contexto de autenticación
    </ProveedorClienteConsulta>
)
}
```

5. También necesitamos agregar las definiciones de propTypes para el componente App ahora:

```
Aplicación.propTypes = {
  hijos: PropTypes.element.isRequired,
}
```

6. Cree un nuevo archivo src/routes.jsx e importe allí las importaciones eliminadas anteriormente:

```
importar { Blog } desde './pages/Blog.jsx' importar { Registrarse }
desde './pages/Signup.jsx' importar { Iniciar sesión } desde './pages/
Login.jsx'
```

7. \$en, agrega las definiciones de ruta y expórtalas:

```
exportar const rutas = [
{
  ruta: '/',
  elemento: <Blog />, },
  ruta: '/signup', elemento:
  <Signup />, },
  ruta: '/login', elemento:
  <Login />, },
]
```

Ahora que hemos refactorizado la estructura de nuestra aplicación de manera tal que podamos reutilizar las rutas en los puntos de entrada del lado del cliente y del servidor, redefinamos el enrutador en el punto de entrada del cliente.

Definición del enrutador del lado del cliente

Siga estos pasos para volver a definir el enrutador en el punto de entrada del cliente:

1. Edite src/entry-client.jsx e importe RouterProvider, el
Función createBrowserRouter y rutas:

```
importar React desde 'react' importar
ReactDOM desde 'react-dom/client'
```

```
importar { createBrowserRouter, RouterProvider } desde 'react-
enrutador-dom'
importar { App } desde './App.jsx' importar { rutas }
desde './routes.jsx'
```

2. \$en, crea un nuevo enrutador del navegador basado en la definición de rutas:

```
const router = createBrowserRouter(rutas)
```

3. Ajuste la función de renderizado para renderizar la aplicación con RouterProvider:

```
ReactDOM.hydrateRoot(documento.getElementById('raíz'),
  <React.Modo estricto>
    <App>
      <RouterProvider router={enrutador} /> </App> </
        React.StrictMode>,
    )
)
```

A continuación, definamos el enrutador del lado del servidor.

Asignación de la solicitud Express a una solicitud Fetch

Del lado del servidor, recibiremos una solicitud Express, que primero debemos convertir en una solicitud Fetch para que React Router pueda interpretarla. Hagámoslo ahora:

1. Cree un nuevo archivo src/request.js y defina allí una función createFetchRequest.

que toma una solicitud Express como argumento:

```
función de exportación createFetchRequest(req) {
```

2. Primero, define el origen de la solicitud y crea la URL:

```
const origen = `${req.protocol}://${req.get('host')}` const url = new URL(req.originalUrl ||
req.url, origen)
```

Necesitamos usar req.originalUrl rst (si está disponible) para tener en cuenta que el middleware Vite podría cambiar la URL.

3. \$en, definimos un nuevo AbortController para manejar cuándo se cierra la solicitud:

```
const controlador = nuevo AbortController() req.on('cerrar', () =>
controlador.abort())
```

4. A continuación, asignamos los encabezados de solicitud Express a los encabezados Fetch:

```
const encabezados = nuevos encabezados()

para (const [clave, valores] de Object.entries(req.headers)) {
    si (!valores) continuar si
    (Array.isArray(valores)) {
        para (valor constante de valores)
        { headers.append(clave, valor)

    } } else
    { headers.set(clave, valores)
    }
}
```

5. Ahora, podemos construir el objeto init para la solicitud Fetch, que consta de un método, encabezados y AbortController:

```
const init = { método:
    req.method, encabezados,
    señal:
    controlador.signal,
}
```

6. Si nuestra solicitud no fue una solicitud GET o HEAD, también obtenemos el cuerpo, así que agreguemoslo a Fetch.

solicitud también:

```
si (req.método != 'GET' && req.método != 'HEAD') {
    init.cuerpo = req.cuerpo
}
```

7. Finalmente, creamos el objeto Fetch Request a partir de la información extraída:

```
devolver nueva Solicitud(url.href, init)
}
```

Ahora que tenemos una función de utilidad para convertir una solicitud Express en una solicitud Fetch, podemos usarla para definir el enrutador del lado del servidor.

Definición del enrutador del lado del servidor

El enrutador del servidor funciona de forma muy similar al enrutador del cliente, excepto que recibimos la información de la solicitud de Express en lugar de la página y usamos StaticRouter, ya que la ruta no puede cambiar en el servidor. Siga estos pasos para definir el enrutador del servidor:

1. Edite src/entry-server.jsx e importe StaticRouterProvider y las funciones createStaticHandler y createStaticRouter. Además, importe la definición de rutas y la función createFetchRequest que acabamos de definir:

```
importar ReactDOMServer desde 'react-dom/server' importar

{ createStaticHandler,
  createStaticRouter,
  Proveedor de enrutador estático,
} desde 'react-router-dom/server' importar { App }
desde './App.jsx' importar { rutas } desde './routes.jsx'
importar { createFetchRequest } desde './request.js'
```

2. Defina un controlador estático para las rutas:

```
const handler = createStaticHandler(rutas)
```

3. Ajuste la función de renderizado para aceptar un objeto de solicitud Express y luego cree una solicitud Fetch desde él utilizando nuestra función definida previamente:

```
exportar función asíncrona render(req) { const fetchRequest
  = createFetchRequest(req)
```

4. Ahora podemos usar esta solicitud convertida para pasarla a nuestro controlador estático, que crea contexto para la ruta, lo que permite a React Router ver a qué ruta intentamos acceder y con qué parámetros:

```
contexto constante = await manejador.consulta(fetchRequest)
```

5. A partir de las rutas definidas por el manejador y el contexto, podemos crear un enrutador estático:

```
const router = createStaticRouter(handler.dataRoutes, contexto)
```

6. Finalmente, podemos ajustar la representación para renderizar el enrutador estático y nuestra estructura de aplicación refactorizada:

```
devuelve ReactDOMServer.renderToString(
  <Aplicación>
    <StaticRouterProvider enrutador={enrutador} contexto={contexto} />
  </Aplicación>,
```

```
)  
}
```

7. Queda una cosa más por hacer. Necesitamos pasar la solicitud Express a render().

Función del punto de entrada del servidor. Edite la siguiente línea en el archivo server.js:

```
const appHtml = await render(req)
```

8. Si el frontend y el backend ya están ejecutándose, asegúrese de cerrarlos.

9. Inicie el frontend de la siguiente manera:

```
$ npm ejecuta dev
```

10. Además, inicie el backend en una terminal separada:

```
$ cd backend  
$ npm ejecuta dev
```

El frontend \$e ahora mostrará el servidor de desarrollo ssr ejecutándose en <http://localhost:5173>

¡Y renderizamos todas nuestras páginas correctamente en el servidor! Puedes verificar que se renderiza en el servidor abriendo

DevTools, haciendo clic en el ícono de engranaje en la esquina superior derecha, desplázate hacia abajo en el panel

Configuración | Preferencias hasta la sección Depurador y marcando la casilla Deshabilitar JavaScript, como se muestra a continuación:

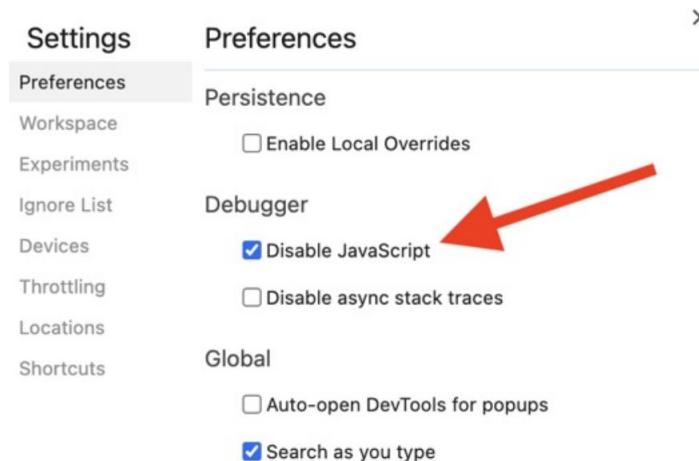


Figura 7.8 – Deshabilitar JavaScript en DevTools

Ahora, actualiza la página y verás que parte de la aplicación aún se renderiza. Actualmente, solo la parte superior de la aplicación se renderiza completamente en el servidor. La lista de publicaciones \$e aún no se renderiza en el servidor. Esto se debe a que los ganchos useQuery usan internamente un gancho elect para obtener datos después de montar el componente. Por lo tanto, no funcionan con la renderización del servidor. Sin embargo, aún podemos lograr que la obtención de datos funcione con la renderización del servidor. Veremos esto en la siguiente sección.

Obtención de datos del lado del servidor

Como hemos visto, la obtención de datos no funciona de inmediato en el lado del servidor. Hay dos enfoques para la obtención de datos del lado del servidor con React Query:

- Enfoque de datos iniciales: use la opción `initialData` en el gancho `useQuery` para pasar datos pre-buscados. Este enfoque es suficiente para obtener una lista de publicaciones, pero sería complicado para obtener datos profundamente anidados, como los nombres de usuario de cada autor.
- Enfoque de hidratación: `$is` nos permite obtener previamente cualquier solicitud y almacenar el resultado por su clave de consulta y obtener previamente cualquier solicitud en el lado del servidor, incluso si está profundamente anidada dentro de la aplicación, sin tener que pasar los datos obtenidos previamente mediante accesorios o un contexto.

Primero usaremos la opción `initialData` para obtener la lista de publicaciones del blog y luego ampliaremos nuestra solución al enfoque de hidratación para que podamos tener una idea de cómo funcionan ambos enfoques y cuáles son sus ventajas y desventajas.

Utilizando datos iniciales

React Router nos permite definir cargadores en las rutas, que podemos usar para obtener datos tanto del servidor como del cliente al cargar la ruta. Luego, podemos pasar los datos obtenidos de los cargadores al componente `Blog` y al gancho `useQuery` mediante la opción `initialData`. Veamos esto ahora:

1. Edite `src/routes.jsx` e importe el gancho `useLoaderData` desde `react-router-dom` y la función `getPosts`:

```
importar { useLoaderData } desde 'react-router-dom'
importar { Blog } desde './pages/Blog.jsx'
importar { Signup } desde './pages/Signup.jsx'
importar { Login } desde './pages/Login.jsx'
importar { getPosts } desde './api/posts.js'
```

2. Ajustar la ruta para definir una función de carga, en la que simplemente llamamos a la función `getPosts`. Luego podemos definir un método `Component()` en el que usamos el gancho `useLoaderData` para obtener los datos del cargador y pasarlo al componente `Blog`, de la siguiente manera:

```
exportar const rutas = [
  {
    camino: '/',
    cargador: getPosts,
    Componente() {
      constante posts = useLoaderData()
      devolver <Blog initialData={publicaciones} />
    },
  },
]
```

3. Edite src/pages/Blog.jsx e importe PropTypes allí, para que podamos definir un nuevo apoyo para el componente más adelante:

```
importar PropTypes desde 'prop-types'
```

4. \$en, agrega la propiedad initialData al componente Blog:

```
función de exportación Blog({ initialData }) {
```

5. Pase la propiedad initialData al gancho useQuery, de la siguiente manera:

```
constante postsQuery = useQuery({  
    clave de consulta: ['publicaciones', { autor, ordenar por, ordenar por }],  
    queryFn: () => getPosts({ autor, sortBy, sortOrder }),  
    Datos iniciales,  
})
```

6. Por último, define propTypes para el componente Blog:

```
Blog.propTypes = {  
    Datos iniciales: PropTypes.shape(PostList.propTypes.posts),  
}
```

Actualice la página del frontend (con JavaScript deshabilitado) y ahora mostrará la lista de publicaciones, pero sin resolver los nombres de usuario de los autores. Como podemos ver, el enfoque inicial para los datos es bastante simple. Sin embargo, si quisieramos obtener los nombres de usuario de todos los autores, tendríamos que almacenarlos en algún lugar y luego pasarlo a los componentes de usuario mediante propiedades o un contexto, lo cual sería bastante tedioso y no escalaría bien si necesitáramos realizar más solicitudes posteriormente. Afortunadamente, existe otro enfoque más avanzado que aprenderemos a continuación.

Usando la hidratación

Con el enfoque de hidratación, creamos un cliente de consulta para precargar cualquier solicitud que queramos realizar, la deshidratamos, la pasamos al componente mediante un cargador y la hidratamos de nuevo allí. Con este enfoque, podemos simplemente realizar cualquier consulta y almacenarla usando una clave de consulta. Si un componente usa la misma clave de consulta, podrá renderizar los resultados en el servidor. Implementemos ahora el enfoque de hidratación:

1. Edite src/routes.jsx e importe QueryClient, la función dehydrate y el componente Hydrate desde React Query:

```
importar { QueryClient, deshidratar, HydrationBoundary } desde '@  
tanstack/react-query'
```

2. Además, importa la función getUserInfo, ya que ahora también vamos a buscar nombres de usuario:

```
importar { getUserInfo } desde './api/users.js'
```

3. Ajuste el cargador; ahora vamos a crear un cliente de consulta allí:

```
{
  camino: '/',
  cargador: async() => {
    constante queryClient = nuevo QueryClient()
```

4. \$en, simulamos la solicitud getPosts del componente Blog pasando el mismo argumentos predeterminados como lo haría el componente:

```
const autor =
  "
constante sortBy = 'createdAt'
const sortOrder = 'descendente'
const posts = await getPosts({ autor, sortBy, sortOrder
})
```

Nota

La duplicación de argumentos predeterminados es un poco problemática. Sin embargo, con nuestra solución actual de renderizado del lado del servidor, la obtención de datos y el renderizado de componentes están demasiado separados como para compartir el código correctamente. Una solución de renderizado del lado del servidor más sofisticada, como Next.js o Remix, puede gestionar mejor este patrón.

5. Ahora, podemos llamar a queryClient.prefetchQuery, con la misma clave de consulta que utilizará el gancho useQuery en el componente, para obtener previamente los resultados de la consulta:

```
esperar queryClient.prefetchQuery({
  clave de consulta: ['publicaciones', { autor, ordenar por, ordenar por }],
  queryFn: () => publicaciones,
})
```

6. A continuación, utilizamos la matriz de publicaciones obtenidas para obtener una lista única de identificaciones de autores:

```
const uniqueAuthors = publicaciones
  .map((post) => post.autor)
  .filter((valor, índice, matriz) => matriz.indexOf(valor)
    === índice)
```

7. Ahora recorremos todos los ID de autor y recuperamos previamente su información:

```
para (const userId de autores únicos) {
  esperar queryClient.prefetchQuery({
    consultaKey: ['usuarios', ID de usuario],
    consultaFn: () => obtenerInfoUsuario(IdUsuario),
  })
}
```

8. Ahora que hemos obtenido previamente todos los datos necesarios, debemos llamar a `dehydrate` en `queryClient` para devolverlo en un formato serializable:

```
devolver deshidratar(consultaCliente),
```

9. En el método `Component()`, obtenemos este estado deshidratado y usamos el componente `Hydrate` para hidratarlo nuevamente. Este proceso de hidratación hace que los datos sean accesibles para el cliente de consulta renderizado del lado del servidor:

```
Componente() {
  const dehydratedState = useLoaderData() return (
    <Estado límite de hidratación={estadodeshidratado}>
      <Blog />
    </Límite de hidratación>
  )
}, },
```

10. Finalmente, podemos revertir el componente `src/pages/Blog.jsx` a su estado anterior. Empezamos eliminando la importación de `PropTypes`:

```
importar PropTypes desde 'prop-types'
```

11. \$en, eliminamos la propiedad `initialData`:

```
función de exportación Blog({ initialData }) {
```

12. También lo eliminamos en el gancho `useQuery`:

```
const postsQuery = useQuery({ queryKey: ['posts',
  { autor, sortBy, sortOrder }], queryFn: () => getPosts({ autor, sortBy, sortOrder }),

  Datos iniciales,
})
```

13. Por último, eliminamos la definición de `propTypes`:

```
Blog.propTypes = {
  Datos iniciales: PropTypes.shape(PostList.propTypes),
}
```

14. Salga del frontend mediante `Ctrl + C`, luego reinícielo de la siguiente manera:

```
$ npm ejecuta dev
```

15. Actualice la página y verá que la lista completa de publicaciones del blog, incluidos todos los nombres de los autores, está ¡Ahora se representa correctamente en el lado del servidor, incluso con JavaScript deshabilitado!

Hagamos otra prueba comparativa para ver cómo ha mejorado el rendimiento:

1. Abra Chrome DevTools.
2. Vuelva a habilitar JavaScript yendo al engranaje, Configuración | Preferencias y desmarcando Deshabilitar JavaScript.
3. Vaya a la pestaña Lighthouse . Haga clic en "Analizar carga de página" para generar un nuevo informe.

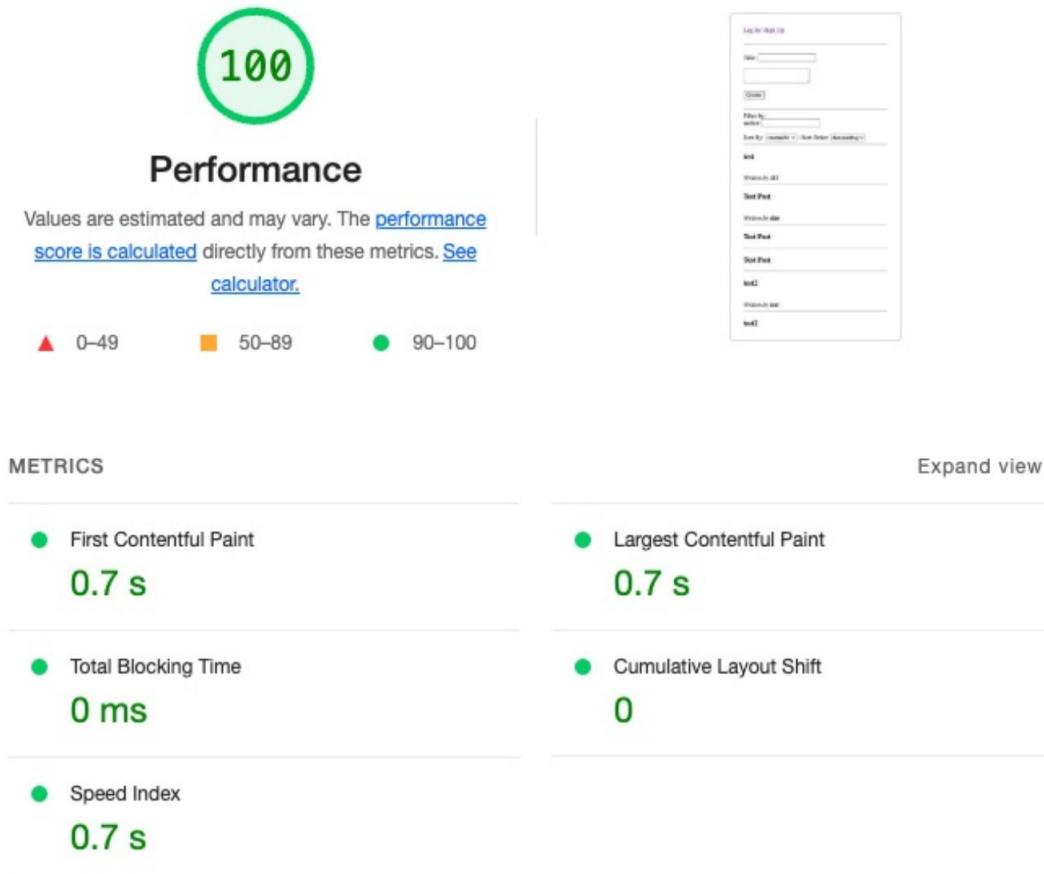


Figura 7.9 – Puntuación de rendimiento de Lighthouse de la aplicación renderizada del lado del servidor con el servidor de desarrollo

Los tiempos de FCP y LCP de \$e son casi la mitad de los tiempos reportados previamente para la renderización del lado del cliente en modo de producción. Al observar el diagrama de cascada en la pestaña Red , ahora podemos ver que solo hay una solicitud para obtener la página inicial.

Ahora terminaremos el capítulo aprendiendo sobre la renderización avanzada del lado del servidor.

Renderizado avanzado del lado del servidor

En las secciones anteriores, hemos creado con éxito un servidor que puede realizar renderizado del lado del servidor con recarga en caliente, lo que es muy útil para el desarrollo pero empeorará el rendimiento en producción.

Ahora, creemos otra función de servidor para un servidor de producción. Esta compilará archivos, usará compresión y no cargará el middleware de Vite para la recarga en caliente. Siga estos pasos para crear el servidor de producción:

1. En la raíz de nuestro proyecto, instale la dependencia de compresión con el siguiente comando:

```
$ npm install compresión@1.7.4
```

2. Edite server.js y defina una nueva función para el servidor de producción, encima de Función createDevServer:

```
función asíncrona createProdServer() {
```

3. En esta función, definimos una nueva aplicación Express y usamos el paquete de compresión y el paquete serve-static para servir a nuestro cliente:

```
aplicación constante = express()

aplicación.use((await import('compresión')).default())
aplicación.uso(
    espera importación('serve-static')).predeterminado(
        ruta.resolve(__dirname, 'dist/cliente'),
        {
            índice: falso,
        },
    ),
)
```

4. Luego, definimos una ruta que captura todas las rutas nuevamente, esta vez cargando la plantilla desde el archivos compilados en la carpeta dist/:

```
aplicación.use('*', async (req, res, next) => {
    intentar {
        dejar que la plantilla = fs.readFileSync(
            ruta.resolve(__nombredir, 'dist/cliente/index.html'),
            'utf-8',
        )
    }
})
```

5. Ahora también importamos y renderizamos directamente el punto de entrada del lado del servidor:

```
const render = (await import('./dist/server/entry-server.js')).render
```

6. Como antes, renderizamos la aplicación React, reemplazamos el marcador de posición en index.html con la aplicación renderizada y devolvemos el HTML resultante:

```
const appHtml = await render(req) const html =
  plantilla.replace('<!--ssr-outlet-->', appHtml) res.status(200).set({ 'Tipo de contenido': 'texto/
  html' }).end(html)
```

7. Para el manejo de errores, simplemente lo pasamos al siguiente middleware y devolvemos la aplicación:

```
} captura (e) {
  siguiente(e)

}

aplicación de retorno
}
```

8. En la parte inferior del archivo server.js, donde creamos el servidor de desarrollo, ahora verificamos la variable de entorno NODE_ENV y la usamos para decidir si iniciar el servidor de producción o el servidor de desarrollo:

```
si (proceso.env.NODE_ENV === 'producción') { const app = await
createProdServer() app.listen(proceso.env.PORT, () =>

  console.log(`servidor
  de producción ssr ejecutándose en http://localhost:${process.env.PORT}`,
),

) } else { const
  app = await createDevServer() app.listen(process.env.PORT, ()
=> console.log(`servidor de desarrollo ssr ejecutándose en
  http://localhost${
    {process.env.
  PUERTO},
),
)
}
```

9. Instale el paquete cross-env, de la siguiente manera:

```
$ npm install cross-env@7.0.3
```

10. Edite package.json y agregue un script de inicio, que inicia el servidor en modo de producción:

```
"inicio": "cross-env NODE_ENV= servidor de nodo de producción",
```

11. Salga del servidor de desarrollo frontend, compile e inicie el servidor de producción:

```
$ npm ejecuta la compilación
```

```
$ npm start
```

Como podemos ver, nuestro servidor sigue sirviendo la aplicación correctamente, pero ya no estamos en modo de desarrollo , por lo que no disponemos de recarga en caliente. ¡\$is completa nuestra implementación del renderizado del lado del servidor! Como pueden imaginar, la implementación del renderizado del lado del servidor en este capítulo es bastante básica, y aún quedan varios aspectos por gestionar:

- Redirecciones y códigos de estado HTTP adecuados
- Generación de sitios estáticos (almacenamiento en caché de las páginas HTML resultantes para que no tengamos que renderizarlas en el lado del servidor) ellos de nuevo cada vez)
- Mejor funcionalidad de obtención de datos
- Mejor división del código entre el servidor y el cliente
- Mejor manejo de las variables de entorno entre el servidor y el cliente

Para solucionar estos problemas, es mejor usar una implementación de renderizado del lado del servidor completamente optimizada en un framework web, como Next.js o Remix. Estos frameworks ya ofrecen métodos para renderizar, obtener datos y enrutar del lado del servidor de forma predeterminada, sin necesidad de configurar manualmente todo para que funcione en conjunto. Aprenderemos más sobre Next.js en el Capítulo 16, Introducción a Next.js.

Resumen

En este capítulo, aprendimos a realizar benchmarks de aplicaciones web con Lighthouse y Chrome DevTools. También aprendimos sobre métricas útiles para dichos benchmarks, llamadas Core Web Vitals. Luego, aprendimos sobre el renderizado de componentes de React en el servidor y las diferencias entre el renderizado del lado del cliente y el del servidor. A continuación, implementamos el renderizado del lado del servidor para nuestra aplicación con Vite y React Router. Luego, implementamos la obtención de datos del lado del servidor con React Query. Volvimos a realizar benchmarks de nuestra aplicación y observamos una mejora en el rendimiento de más del 40 %. Por último, aprendimos a preparar nuestro servidor de renderizado del lado del servidor para producción y los conceptos que un framework de renderizado del lado del servidor más sofisticado debe abordar.

En el siguiente capítulo, Capítulo 8, "Cómo asegurar que los clientes te encuentren con la optimización para motores de búsqueda", aprenderemos cómo hacer que nuestra aplicación web sea más accesible para los rastreadores de motores de búsqueda, mejorando así la puntuación SEO que vimos en el informe Lighthouse. Agregaremos metaetiquetas para obtener más información sobre nuestra aplicación web e integraremos nuestra aplicación con diversas redes sociales.

Asegurarse de que los clientes Encuentre con la búsqueda Optimización del motor

Al optimizar el rendimiento de nuestro blog en el capítulo anterior, habrás notado que el informe Lighthouse también incluye una puntuación de optimización para motores de búsqueda (SEO), en la que nuestra aplicación obtuvo una puntuación relativamente baja. Esta puntuación nos indica el grado de optimización de nuestra aplicación para ser indexada correctamente y encontrada por motores de búsqueda como Google o Bing. Después de desarrollar con éxito una aplicación de blog funcional, queremos que los usuarios la encuentren. En este capítulo, aprenderemos los fundamentos del SEO y cómo optimizar la puntuación SEO de nuestra aplicación React. Luego, aprenderemos a crear metaetiquetas para facilitar su integración en diversas redes sociales.

En este capítulo cubriremos los siguientes temas principales:

- Optimización de una aplicación para motores de búsqueda
- Mejorar las incrustaciones en redes sociales

Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones que aparecen en esos capítulos son las que se utilizan en este libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que algunos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente usar las versiones mencionadas en los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/capítulo8>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/1xN3l0MMTbY>

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

Optimización de una aplicación para motores de búsqueda

Antes de empezar a optimizar nuestra aplicación para motores de búsqueda, aprendamos brevemente cómo funcionan. Los motores de búsqueda almacenan información sobre los sitios web en un índice. El índice contiene... Ubicación, contenido y metainformación de sitios web. Añadir o actualizar páginas en el índice se denomina indexación y la realiza un rastreador. Un rastreador es un software automatizado que recupera sitios web y los indexa. Se le llama rastreador porque sigue enlaces adicionales en el sitio web para encontrar más sitios web. Los rastreadores más avanzados, como el robot de Google, también pueden detectar si se requiere JavaScript para mostrar el contenido de un sitio web e incluso mostrarlo.

El siguiente gráfico visualiza cómo funciona un rastreador de un motor de búsqueda:

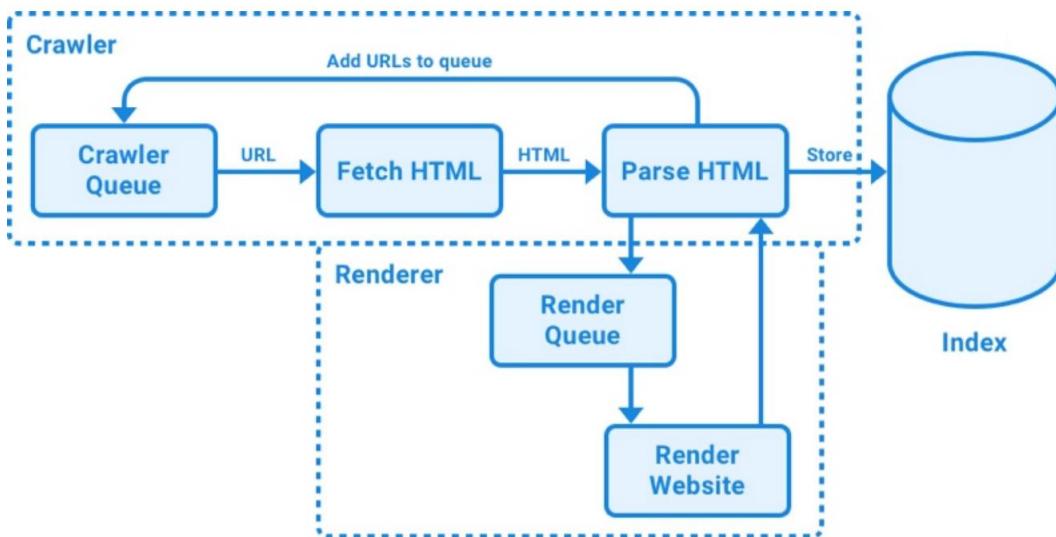


Figura 8.1 – Visualización de cómo funciona un rastreador de un motor de búsqueda

Como podemos ver, un rastreador de búsqueda tiene una cola con las URL que necesita rastrear e indexar. Luego, visita las URL una por una, recupera el HTML y, si se trata de un rastreador avanzado, detecta si necesita ejecutar JavaScript para renderizar el contenido. En ese caso, la URL se añade a una cola de renderizado y el HTML renderizado se devuelve al rastreador posteriormente. \$en, el rastreador extrae todos los enlaces a otras páginas y los añade a la cola. Finalmente, el contenido analizado se añade al índice.

Para comprobar si un sitio web ya está indexado por un motor de búsqueda, la mayoría de estos motores ofrecen el operador "site:", que permite comprobar si una URL ya está indexada. Por ejemplo, site:wikipedia.org muestra varias URL de Wikipedia que ya están indexadas. Si su sitio web aún no está indexado, puede enviarlo a herramientas como Google Search Console. Google Search Console también ofrece un resumen detallado del estado de indexación y de cualquier problema relacionado. Sin embargo, no es necesario enviar nuestro sitio para que lo encuentren, ya que la mayoría de los motores de búsqueda rastrean la web automáticamente y, finalmente, lo encontrarán.

Si su sitio web sigue sin indexarse, podría deberse a una configuración incorrecta. Primero, debe crear un archivo robots.txt para especificar si los motores de búsqueda pueden rastrear partes de su sitio web y cuáles.

Nota

El archivo robots.txt no debe usarse para ocultar páginas web de los resultados de búsqueda de Google. Se usa para reducir el tráfico de los rastreadores en páginas poco importantes o similares. Si desea ocultar completamente las páginas web de los resultados de búsqueda de Google, protéjalas con contraseña o use la metaetiqueta "noindex".

A continuación, debe asegurarse de que el contenido de su sitio web sea visible para el rastreador. La renderización del lado del servidor puede ser útil, ya que permite a los rastreadores ver el contenido de su sitio web sin ejecutar JavaScript. Además, añadir metainformación mediante etiquetas HTML especiales ayuda a los rastreadores a obtener información adicional sobre su sitio web. En sitios web pequeños, es necesario enlazar correctamente las páginas o añadir un mapa del sitio manualmente. En sitios web más grandes, como un blog con muchas entradas, siempre se debe definir un mapa del sitio. Por último, un buen rendimiento, tiempos de carga rápidos y una buena experiencia de usuario mejoran la posición de su sitio web en los motores de búsqueda.

Ya hemos añadido la renderización del lado del servidor para acelerar el rastreo, mostrando el contenido inmediatamente sin depender de JavaScript. Ahora, optimicemos aún más nuestra aplicación para los motores de búsqueda. Empezamos creando un archivo robots.txt.

Creando un archivo robots.txt

Primero, asegúrenos de que los rastreadores tengan permiso explícito para acceder a nuestra aplicación e indexar todas sus páginas. Para ello, necesitamos crear un archivo robots.txt que los rastreadores leerán para determinar a qué páginas pueden acceder (si las hay). Siga estos pasos para crear un archivo robots.txt que permita el acceso de todos los rastreadores a todas las páginas:

1. Copie la carpeta ch7 a una nueva carpeta ch8, de la siguiente manera:

```
$ cp -R ch7 ch8
```

2. Abra la carpeta ch8 en VS Code.

3. Cree un nuevo archivo public/robots.txt en la raíz de nuestro proyecto.
4. Abra el archivo recién creado e ingrese el siguiente contenido para permitir que todos los rastreadores lo indexen.

todas las páginas:

Agente de usuario: *

Permitir: /

El archivo robots.txt funciona definiendo bloques, cada uno de los cuales se define mediante la coincidencia con un agente de usuario. El agente de usuario puede coincidir con varios rastreadores, como Googlebot para Google, o puede usar * para coincidir con todos los rastreadores. Después del agente de usuario, se pueden usar uno o varios permisos y/o bloques.

Se pueden realizar declaraciones que decidan a qué rutas puede o no acceder un rastreador.

En nuestro caso, permitimos el acceso a todas las rutas. Además, se puede especificar un mapa del sitio, pero veremos más sobre esto más adelante en la subsección "Crear un mapa del sitio".

5. Abra un panel de Terminal e inicie la interfaz ejecutando el siguiente comando:

```
$ npm ejecuta dev
```

6. Abra otro panel de Terminal e inicie el backend ejecutando los siguientes comandos:

```
$ cd backend
```

```
$ npm ejecuta dev
```

7. Vaya a <http://localhost:5173/robots.txt> en su navegador para ver el archivo robots.

El archivo txt se está sirviendo correctamente.

Ahora que hemos permitido que los rastreadores accedan a nuestra aplicación, debemos mejorar la estructura de URL. Para ello, crearemos páginas independientes para cada publicación.

Creación de páginas separadas para publicaciones

Actualmente, no es posible ver solo una entrada en nuestra aplicación de blog; solo podemos ver la lista completa. Esto no es bueno para el SEO, ya que significa que un motor de búsqueda siempre enlazará a la página de índice, que podría contener artículos diferentes a los que el usuario buscaba. Refactoricemos ligeramente nuestra aplicación para mostrar solo los títulos y autores de las entradas en la página principal y luego enlazar a páginas separadas para cada entrada del blog:

1. Edite src/components/Post.jsx para permitir la visualización de una sola publicación completa y, al mismo tiempo, mostrar una versión reducida de la publicación en la lista, con un enlace a la versión completa. Primero, importamos el enlace componente de react-router-dom:

```
importar { Enlace } desde 'react-router-dom'
```

2. \$en, agregamos una propiedad `_id` y una propiedad `fullPost` al componente `Post`. La propiedad `fullPost` se establecerá en falso de manera predeterminada (cuando se muestre en la lista de publicaciones) y se establecerá en verdadero cuando se use en la página de publicación única:

```
función de exportación Post({ título,  
    contenido,  
    autor,  
    _id,  
    fullPost = falso, }) {
```

3. Hacemos algunos ajustes al componente para mostrar un enlace a la página de publicación única si estamos

Aún no está en una página de una sola publicación:

```
{Publicación completa?  
  ( <h3>{título}</h3>  
  )  :  (  
    <Enlace a={`/posts/${_id}`}> <h3>{título}</h3>  
    </Link> )}
```

4. Además, solo mostramos el contenido de la entrada del blog en una sola página y ajustamos el espaciado de la información del autor según corresponda.

```
{publicación completa && <div>{contenido}</div>} {autor && (  
  <em>  
    {Publicación completa && <br />}  
    Escrito por <User id={author} /> </em> )}
```

5. Ajuste los tipos de propiedad para agregar las propiedades recién definidas:

```
Post.propTypes = {  
  título: PropTypes.string.isRequired, contenido: PropTypes.string,  
  autor: PropTypes.string, _id:  
  PropTypes.string.isRequired, fullPost:  
  PropTypes.bool,  
}
```

6. Edite src/api/posts.js y agregue una nueva función para obtener una sola publicación por id:

```
exportar const getPostById = async (postId) => {
    const res = await fetch(`.${{import.meta.env.VITE_BACKEND_URL}}/ posts/${{postId}}`)

    devolver esperar res.json()
}
```

7. Crea un nuevo archivo src/pages/ViewPost.jsx y comienza importando todos los componentes y funciones que vamos a necesitar:

```
importar { Enlace } desde 'react-router-dom' importar PropTypes
desde 'prop-types' importar { useQuery } desde '@tanstack/
react-query' importar { Encabezado } desde './components/Header.jsx' importar
{ Publicación } desde './components/Post.jsx' importar { getPostById } desde './api/
posts.js'
```

8. \$en, define un componente que acepta un postId como propiedad:

```
función de exportación ViewPost({ postId }) {
```

9. En el componente, utilizamos un gancho de consulta para obtener una sola publicación por id:

```
constante postQuery = useQuery({
    queryKey: ['post', postId], queryFn: () =>
    getPostById(postId), }) const post = postQuery.data
```

10. A continuación, renderiza el encabezado y un enlace a la página principal:

```
devolver (
    <div style={{ relleno: 8 }}>
        <Encabezado />
        <br />
        <hr />
        <Link to="/">Volver a la página principal</Link> <br /> <hr />
```

11. \$en, si logramos obtener una publicación con el ID dado, renderiza una publicación con la propiedad fullPost
De lo contrario, se mostrará un mensaje de "no encontrado":

```
{post ? <Post {...post} fullPost /> : `No se encontró la publicación con id ${postId}.`} </
div>
```

```
    )  
}
```

12. Por último, defina los tipos de propiedad para el componente ViewPost:

```
ViewPost.propTypes = { postId:  
  PropTypes.string.isRequired,  
}
```

13. Edite src/routes.jsx e importe el componente ViewPost y la función getPostById (para la representación del lado del servidor):

```
importar { ViewPost } desde './pages/ViewPost.jsx' importar { getPosts,  
getPostById } desde './api/posts.js'
```

14. Defina una nueva ruta /posts/:postId para ver una sola entrada. En el cargador, recuperamos la entrada del blog y su autor, si lo tiene. Luego, devolvemos el estado deshidratado y el ID de la entrada:

```
{  
  ruta: '/posts/:postId', cargador: async  
  ({ params }) => { const postId = params.postId  
  
    constante queryClient = nuevo QueryClient()  
  
    const post = await getPostById(postId) await  
    queryClient.prefetchQuery({ clave de consulta:  
      ['post', postId], función de consulta: () =>  
      post, })  
  
    si (publicación?.autor)  
    { esperar consultaClient.prefetchQuery({  
      consultaKey: ['usuarios', post.autor], consultaFn: ()  
      =>  
      getUserInfo(post.autor),  
    })  
    }  
  
    devolver { dehydratedState: dehydrate(queryClient), postId } },
```

15. Defina un método Componente para la ruta, donde obtenemos dehydratedState y postId y pasarlos al componente ViewPost, de la siguiente manera:

```
Componente() {
  const {estado deshidratado, postId} = useLoaderData()
  devolver (
    <Estado límite de hidratación={estadodeshidratado}>
      <ViewPost postId={postId} />
    </Límite de hidratación>
  )
},
},
```

16. Accede a <http://localhost:5173/> en tu navegador y verás que todas las entradas de blog de la lista tienen un enlace en el título. Haz clic en el enlace para ver la entrada completa, como se muestra en la siguiente captura de pantalla:



Figura 8.2 – Visualización de una sola publicación de blog en una página separada

Ahora nuestra aplicación de blog está mucho más organizada, ya que no vemos el contenido completo de todas las entradas en la página principal. Solo vemos el título y el autor, y podemos decidir si el artículo nos interesa. Además, un motor de búsqueda puede ofrecer entradas separadas para cada entrada, lo que facilita encontrarlas en nuestra aplicación. Sin embargo, la estructura de las URL aún puede mejorarse , ya que actualmente solo contiene el ID de la entrada. En el siguiente paso, introduciremos URLs más significativas.

Creación de URL significativas (slugs)

Los sitios web suelen incluir palabras clave en las URL para que los usuarios puedan ver fácilmente lo que abrirán con solo mirar la URL. Las palabras clave en las URL también son un factor de posicionamiento para los motores de búsqueda, aunque no tan importante. El más importante siempre es un buen contenido. Sin embargo, una buena estructura de URL mejora la experiencia del usuario. Por ejemplo, si el enlace es `/posts/64a42dfd6a7b7ab47009f5e3/` Al asegurar que los clientes te encuentren con optimización para motores de búsqueda, en lugar de simplemente `/posts/64a42dfd6a7b7ab47009f5e3`, la URL ya deja claro qué contenido encontrarán en la página. Estas palabras clave en la URL se denominan slugs de URL, o "slugs" en periodismo, y se refieren al uso de descripciones cortas de artículos como nombres internos. Comencemos introduciendo slugs en nuestras páginas de entradas:

1. Edite `src/routes.jsx` y ajuste la ruta para permitir la inclusión opcional de un slug:

```
ruta: '/posts/:postId/:slug?',
```

Nota

No verificamos si el slug es correcto. De hecho, no es necesario, y muchas páginas no lo hacen. Siempre que tengamos un ID correcto, podemos mostrar la entrada del blog. Solo necesitamos asegurarnos de que todos los enlaces a la página incluyan el slug correcto.

Sin embargo, podríamos agregar adicionalmente un elemento `<link>` con el atributo `rel="canonical"` a una página, especificando la página canónica con el slug correcto. `$is` le indicaría a los rastreadores que no indexen páginas duplicadas cuando se utilizan slugs incorrectos.

2. En la raíz de nuestro proyecto, instala el paquete `npm slug`, que contiene una función para ejecutar correctamente

```
convertir un título en slug:
```

```
$ npm install slug@8.2.3
```

El paquete `$is` ya maneja Unicode y devuelve cadenas seguras para URL. Por lo tanto, no tenemos que preocuparnos por corregir la cadena de título nosotros mismos.

3. Edite `src/components/Post.jsx` e importe la función `slug`:

```
importar slug desde 'slug'
```

4. \$en, ajusta el enlace a la publicación del blog agregando el slug, de la siguiente manera:

```
<Enlace a={`/posts/${_id}/${slug(title)}`}>
```

5. Ahora, cuando abramos un enlace de la lista, la URL se verá así:

`http://localhost:5173/posts/64a42dfd6a7b7ab47009f5e3/`
Cómo asegurarse de que los clientes lo encuentren con la optimización para motores de búsqueda

¡Ahora tenemos URLs legibles para las entradas de nuestro blog! Sin embargo, quizás hayas notado que el título sigue siendo Vite + React en todas las páginas de nuestra aplicación. Vamos a cambiar esto introduciendo títulos dinámicos e incluyendo el título de la entrada del blog en el título de la página.

Agregar títulos dinámicos

El título de una página es incluso más importante para el SEO que las palabras clave en la URL, ya que es el que se mostrará en los resultados de búsqueda en la mayoría de los casos. Por lo tanto, debemos elegir el título con cuidado y, si tenemos contenido dinámico (como en nuestro blog), también debemos ajustarlo dinámicamente para que se ajuste al contenido. Podemos usar la biblioteca React Helmet para facilitar los cambios en la sección <head> del documento HTML. Esta biblioteca nos permite renderizar un componente especial de Helmet. Los componentes secundarios de este componente reemplazarán las etiquetas existentes en la sección <head>. Sigue estos pasos para usar React Helmet y configurar el título dinámicamente:

1. Primero, cambiemos el título general de nuestra aplicación, ya que sigue siendo Vite + React. Editar índice.html en la raíz de nuestro proyecto y cambiamos el título. Llamaremos a nuestra aplicación de blog "Full- Stack React Blog":

```
Blog de React de pila completa
```

2. En la raíz de nuestro proyecto, instala la dependencia react-helmet-async para poder cambiar dinámicamente el título:

```
$ npm install react-helmet-async@1.3.0
```

Nota

React Helmet Async es una bifurcación del React Helmet original que agrega soporte para versiones más nuevas de React.

3. Edite src/pages/ViewPost.jsx e importe el componente Helmet desde react- helmet-async:

```
importar { Casco } desde 'react-helmet-async'
```

4. Renderiza el componente Casco y define la etiqueta <title> dentro de él, de la siguiente manera:

```
devolver (
  <div style={{ relleno: 8 }}>
    {correo && (
      <Casco>
        <title>{post.title} | Blog de React Full-Stack</title>
      </Casco>
    )}
  </div>
)
```

5. Edite src/pages/Blog.jsx e importe Helmet:

```
importar { Casco } desde 'react-helmet-async'
```

6. \$en, restablece el título a Full-Stack React Blog en el componente Blog:

```
devolver
( <div estilos={{ relleno: 8 }}>
  <Casco>
    Blog de React de pila completa
  </Casco>
```

7. Edite src/App.jsx e importe HelmetProvider:

```
importar { HelmetProvider } desde 'react-helmet-async'
```

8. \$en, ajusta el componente de la aplicación para representar HelmetProvider:

```
función de exportación App({ children }) {
  devolver (
    <Proveedor de casco>
      <QueryClientProvider cliente={queryClient}>
        <Proveedor de contexto de autenticación>
          {niños}
        Proveedor de contexto de autenticación
      </ProveedorClienteConsulta>
    </ProveedorDeCasco>
  )
}
```

9. Haz clic en una sola publicación en la aplicación y verás que el título ahora se actualiza para incluir el título de la publicación.

Ahora que hemos establecido con éxito un título dinámico, prestemos atención a otra información importante en la sección <head>, las metaetiquetas.

Agregar otras metaetiquetas

Las metaetiquetas, como su nombre indica, contienen metainformación sobre una página. Además del título, podemos configurar metainformación como una breve descripción o información sobre cómo el navegador debe mostrar un sitio web. En esta sección, abordaremos las metaetiquetas más importantes para SEO, empezando por la metaetiqueta de descripción.

Etiqueta meta de descripción

La metaetiqueta 'e description' contiene una breve descripción del contenido de la página. Al igual que con la etiqueta de título, también podemos configurarla dinámicamente, como se indica a continuación:

1. Edite src/pages/Blog.jsx y agregue la siguiente etiqueta de descripción genérica <meta>:

```
<Casco>
  Blog de React de pila completa
  <meta
    nombre='descripción'
    content='Un blog lleno de artículos sobre desarrollo full-stack de React.'

  />
</Casco>
```

Ahora, agreguemos una etiqueta de meta descripción dinámica para cada entrada del blog. La meta descripción debe tener entre 50 y 160 caracteres, y como no tenemos un resumen breve de nuestras entradas, usaremos el contenido completo y lo acortaremos después de 160 caracteres. Por supuesto, sería aún mejor permitir que los autores agreguen un resumen breve al crear entradas, pero para simplificar, simplemente acortamos la descripción.

2. Edite el archivo src/pages/ViewPost.jsx y defina una función simple para truncar una cadena:

```
función truncar(str, max = 160) {
  si (!str) devuelve str
  si (str.length > max) {
    devuelve str.slice(0, max - 3) + '...'
  } demás {
    devolver cadena
  }
}
```

Limitamos la cadena a 160 caracteres y, si es superior a 160, la truncamos a 157 caracteres y agregamos tres puntos al final.

3. Agregue el contenido truncado como una etiqueta de meta descripción al componente Casco, de la siguiente manera:

```
{correo && (
  <Casco>
    <title>{post.title} | Blog de React Full-Stack</title>
    <meta nombre='descripción' contenido={truncar(publicación.
  contenido)} />
```

Después de agregar la metaetiqueta de descripción, aprendamos sobre otras metaetiquetas que podrían usarse.

Metaetiqueta Robots

La metaetiqueta \$e robots indica a los rastreadores si deben rastrear páginas web y cómo hacerlo. Puede usarse junto con robots.txt, pero solo si queremos restringir dinámicamente el rastreo de una página. Su aspecto es el siguiente:

```
<meta name="robots" content="índice, seguir">
```

La palabra clave \$e index indica a los rastreadores que indexen la página, la palabra clave follow indica a los rastreadores que rastreen más enlaces en la página. Las palabras clave \$e index y follow se pueden desactivar usando noindex ynofollow, respectivamente.

Etiqueta meta de la ventana gráfica

Otra metaetiqueta importante es la etiqueta viewport, que indica al navegador (y a los rastreadores) que tu sitio web es compatible con dispositivos móviles. Observa el siguiente ejemplo de cómo la metaetiqueta afecta la visualización de las páginas en dispositivos móviles:

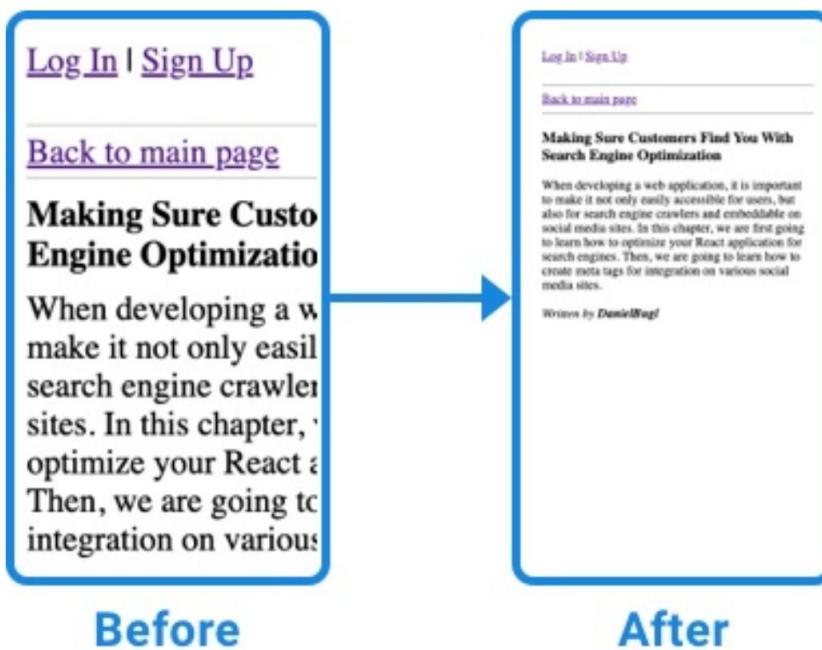


Figura 8.3 – Representación de una publicación de blog antes y después de agregar la metaetiqueta viewport

Vite ya agregó esta metaetiqueta automáticamente en la plantilla index.html que nos proporcionó. Puedes verla consultando el archivo index.html:

```
<meta name="viewport" content="ancho=ancho-del-dispositivo, escala-inicial=1.0" />
```

Después de aprender sobre la etiqueta viewport, continuamos aprendiendo sobre la etiqueta meta charset.

metaetiqueta Charset

La metaetiqueta \$e charset le informa al navegador y a los rastreadores sobre la codificación de caracteres de la página web. Normalmente, conviene configurarlo en UTF-8 para garantizar que todos los caracteres Unicode se representen correctamente. Vite ya añadió esta metaetiqueta automáticamente:

```
<meta charset="UTF-8" />
```

Ahora que hemos aprendido sobre las metaetiquetas relevantes, pasemos a la creación de un mapa del sitio, que ayuda a los rastreadores a encontrar todas las páginas de nuestra aplicación más fácilmente.

Otra metainformación relevante

Hay información meta adicional que puede ser relevante para un sitio web, como configurar el idioma en la etiqueta <html>, de la siguiente manera:

```
<html lang="es">
```

Configurar un favicon también mejora el fragmento de búsqueda, que es lo que ven los usuarios cuando deciden si deben hacer clic en un enlace.

Creación de un mapa del sitio

Un mapa del sitio contiene una lista de URL que forman parte de una aplicación, para que los rastreadores puedan detectar fácilmente contenido nuevo y rastrear la aplicación con mayor eficiencia. También garantiza que se encuentre todo el contenido, lo cual es especialmente importante para aplicaciones basadas en contenido con un gran número de páginas/entradas. Normalmente, los mapas del sitio se proporcionan en formato XML. No son obligatorios para SEO, pero facilitarán y agilizarán la búsqueda de contenido en la aplicación por parte de los rastreadores. Dado que nuestra aplicación de blog tiene contenido dinámico, también deberíamos crear un mapa del sitio dinámico. Sigue estos pasos para crear un mapa del sitio dinámico para nuestra aplicación de blog:

1. Primero, necesitaremos una URL base para nuestro frontend (desplegado) que preceda todas las rutas de nuestro mapa del sitio. Por ahora, simplemente la configuraremos con la URL de nuestro host local, pero en producción, esta variable de entorno debe cambiarse a la URL base correcta de la aplicación. Edite el archivo .env en la raíz de nuestro proyecto y agregue la variable de entorno FRONTEND_URL:

```
URL_INICIO="http://localhost:5173"
```

2. Crea un nuevo archivo generateSitemap.js en la raíz de nuestro proyecto, comienza importando el módulo slug y dotenv:

```
importar slug desde 'slug' importar
dotenv desde 'dotenv' dotenv.config()
```

3. \$en, guarda la variable de entorno creada previamente en una variable baseUrl:

```
constante baseUrl = proceso.env.FRONTEND_URL
```

4. Ahora, defina una función asíncrona para generar un mapa del sitio. En esta función, comenzamos obteniendo una lista de entradas de blog, ya que queremos que cada una forme parte del mapa del sitio:

```
exportar función asíncrona generateSitemap() {
    const postsRequest = await fetch(`process.env.VITE_BACKEND_URL}/posts`)
    const posts = await
    postsRequest.json()
```

5. A continuación, devolvemos una cadena que contiene el XML del mapa del sitio. Empezamos definiendo el XML, el encabezado y una etiqueta <urlset>:

```
devolver `<?xml versión="1.0" codificación="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
```

6. Dentro de la etiqueta <urlset>, podemos usar etiquetas <url> con etiquetas <loc> para vincular a varias páginas.

Primero enumeraremos todas las páginas estáticas:

```
<url>
${baseUrl}

<url>
<loc>${baseUrl}/registro</loc> </url>

<url>
<loc>${baseUrl}/iniciar sesión</loc> </url>
```

7. \$en, recorremos todas las publicaciones que obtuvimos del backend y generamos una etiqueta <url> para cada una de ellas, construyendo las URL a partir del ID de la publicación y el slug:

```
$

{posts .map( (publicación) =>
<url>
<loc>${baseUrl}/posts/${post._id}/${slug(post.title)}</
ubicación>
```

8. También podemos especificar opcionalmente una etiqueta <lastmod>, que le indica al rastreador cuándo se publicó el contenido. última modificación:

```
<lastmod>${publicación.actualizadaA las ?? publicación.creadaA las}</lastmod>
```

9. Por último, unimos todas las etiquetas <url> generadas en una sola cadena y cerramos la etiqueta <urlset>:

```
</url>`,  
)  
.unirse("}")  
</urlset>  
}
```

Ahora que tenemos una función para generar dinámicamente un mapa del sitio, todavía necesitamos incluir una ruta hacia él en nuestro servidor.

10. Edite server.js e importe allí la función generateSitemap:

```
importar { generateSitemap } desde './generateSitemap.js'
```

11. \$en, ve a la primera declaración app.use('*') dentro de la función createProdServer y comprueba si la URL es /sitemap.xml. De ser así, genera el mapa del sitio y devuélvelo como XML:

```
aplicación.use('*', async (req, res, next) => {  
    si (req.originalUrl === '/sitemap.xml') {  
        const sitemap = await generarSitemap()  
        devolver res  
            .estado(200)  
            .set({ 'Tipo-Contenido': 'application/xml' })  
            .end('mapa del sitio')  
    }  
}
```

Nota

En una configuración más sofisticada, podríamos almacenar en caché el mapa del sitio generado en nuestro servidor Express, nuestro propio servidor web o un servicio de almacenamiento en caché independiente.

12. Hacemos el mismo cambio que en el paso anterior para la segunda declaración app.use('*') dentro de la función createDevServer.
13. Reinicie el servidor y vaya a <http://localhost:5173/sitemap.xml> para ver el mapa del sitio generado dinámicamente, con enlaces a todas las publicaciones creadas y sus últimas marcas de tiempo modificadas.
14. Ahora podemos enlazar al mapa del sitio en el archivo robots.txt. Por ejemplo, estableceremos la URL en localhost. En una aplicación de producción, se ajustaría esta URL para que apunte al mapa del sitio en la URL de la aplicación implementada. Edite public/robots.txt y agregue la siguiente línea:

```
Mapa del sitio: http://localhost:5173/sitemap.xml
```

Ahora que hemos implementado con éxito medidas para mejorar nuestra aplicación para los motores de búsqueda, echemos un vistazo a nuestra puntuación SEO en el informe Lighthouse:



Figura 8.4 – ¡Nuestra puntuación SEO de Lighthouse ahora es 100!

Como podemos ver, nuestra puntuación SEO ahora es de 100 (antes era de 91). Esto podría parecer solo una ligera mejora, pero el informe de Lighthouse solo tiene en cuenta comprobaciones básicas, como tener un título, una descripción, una etiqueta de ventana gráfica y un archivo robots.txt. Hemos hecho mucho más para optimizar la experiencia del usuario para los visitantes y los motores de búsqueda, como mejorar la estructura de las URL y añadir títulos y descripciones dinámicos.

Podríamos optimizar aún más nuestra aplicación si sirviéramos recursos estáticos a través de una Red de Entrega de Contenido (CDN) y usáramos imágenes adaptables (sirviendo imágenes en diferentes tamaños para optimizar el rendimiento en conexiones lentas y evitar cargar las imágenes completas). Sin embargo, esto queda fuera del alcance de este libro.

Para finalizar este capítulo, vamos a echar un vistazo a cómo mejorar las incrustaciones en sitios de redes sociales.

Mejorar las incrustaciones en redes sociales

Ya hemos añadido las metaetiquetas importantes para los motores de búsqueda. Sin embargo, las redes sociales, como Facebook y X (anteriormente Twitter), utilizan metaetiquetas adicionales para mejorar la integración de tu aplicación en sus sitios y aplicaciones. La mayoría de las redes sociales utilizan un estándar llamado Open Graph Meta Tags, creado originalmente en Facebook. Estas etiquetas pueden contener información adicional sobre el tipo de página, un título especial, la descripción y una imagen para integrar la página en la red social.

Metaetiquetas de Open Graph

Las metaetiquetas Open Graph (OG) tienen cuatro propiedades genéricas que cada página puede tener:

- og:type: Describe el tipo de página; los tipos específicos pueden tener propiedades adicionales
- og:title: Describe el título de la página tal como debería aparecer en las incrustaciones
- og:image: Una URL a una imagen que debe usarse para la incrustación
- og:url: Una URL a un enlace que debe usarse para la incrustación

La metaetiqueta \$e og:type describe el tipo de contenido disponible en la página. Indica a las redes sociales cómo debe formatearse la incrustación. Entre otros, se pueden usar los siguientes valores:

- sitio web: valor predeterminado \$e, una incrustación básica
- artículo: \$is es para noticias y publicaciones de blogs, y tiene parámetros adicionales para publicado_
hora, hora de modificación, autor, sección y etiqueta
- perfil: para perfiles de usuario, con parámetros adicionales para nombre, apellido, nombre de usuario y género
- libro: para libros, con parámetros adicionales para autor, ISBN, fecha de lanzamiento y etiqueta
- tipos de música: incluye music.song, music.album, music.playlist y music.radio_station, cada uno de ellos con diferentes parámetros adicionales
- tipos de video: incluye video.movie, video.episode, video.tv_show y video.other, cada uno de ellos con diferentes parámetros adicionales

Puede encontrar una descripción completa de las metaetiquetas OG y todos los valores posibles en su sitio web social: <https://ogp.me/>.

Información

La mayoría de las redes sociales admiten metaetiquetas OG para incrustaciones. Sin embargo, algunos sitios web, como X (anteriormente Twitter), tienen sus propias metaetiquetas, que tienen prioridad sobre las metaetiquetas OG si se proporcionan. Sin embargo, X aún puede leer metaetiquetas OG, por lo que basta con proporcionarlas.

Ahora, nos centraremos en el tipo de artículo, ya que estamos desarrollando una aplicación de blog, por lo que podemos usar este tipo para proporcionar mejores incrustaciones para las publicaciones del blog.

Uso de las metaetiquetas del artículo OG

Como hemos aprendido, el tipo de artículo nos permite incluir metainformación sobre la fecha de publicación, la fecha de modificación y el autor de un artículo en nuestra página. Hagamos esto ahora para nuestra página de una sola entrada:

1. Edite src/pages/ViewPost.jsx e importe la función API getUserInfo, ya que necesitaremos resolver el nombre del autor para la metaetiqueta correspondiente:

```
importar { getUserInfo } desde './api/users.js'
```

2. Dentro del componente ViewPost, recuperamos la publicación y el nombre del autor. Nos aseguramos de realizar esta llamada solo si el atributo post?.author existe mediante la opción habilitada del gancho useQuery:

```
constante userInfoQuery = useQuery({
    clave de consulta: ['usuarios', publicación?.autor],
    queryFn: () => obtenerInfoUsuario(publicación?.autor),
    habilitado: Boolean(post?.author),
})
constante userInfo = userInfoQuery.data ?? {}
```

3. Dentro del componente Casco, definimos la etiqueta og:type como artículo y definimos el título, Hora de publicación y hora de modificación:

```
{correo && (
<Casco>
    <title>{post.title} | Blog de React Full-Stack</title>
    <meta nombre='descripción' contenido={truncar(publicación.
contenido)} />
    <meta propiedad='og:type' contenido='artículo' />
    <meta propiedad='og:título' contenido={post.título} />
    <meta propiedad='og:article:published_time'
contenido={post.createdAt} />
    <meta propiedad='og:article:modified_time'
contenido={post.updatedAt} />
```

4. \$en, establecemos og:article:author en el nombre de usuario resuelto:

```
<meta propiedad='og:article:author' contenido={información del usuario.
nombre de usuario} />
```

5. Por último, recorremos las etiquetas (si no hay ninguna, usamos una matriz vacía de forma predeterminada) y definimos una metaetiqueta para cada etiqueta:

```
{(post.tags ?? []).map((etiqueta) => (
    <meta clave={etiqueta} propiedad='og:artículo:etiqueta'
contenido={etiqueta} />
))}
```

Las matrices en las metaetiquetas OG funcionan redefiniendo la misma propiedad varias veces.

¡Ahora que hemos agregado exitosamente metaetiquetas, nuestra aplicación de blog está optimizada para motores de búsqueda y sitios de redes sociales!

Resumen

En este capítulo, aprendimos brevemente cómo funcionan los motores de búsqueda. Luego, creamos un archivo robots.txt.

Le, junto con páginas separadas para cada entrada del blog, para optimizarlo mejor para los motores de búsqueda. A continuación, creamos URLs relevantes (slugs) y establecimos títulos y metaetiquetas dinámicos. Luego, creamos un mapa del sitio y evaluamos la puntuación SEO de nuestro blog después de todas las optimizaciones. Finalmente, aprendimos cómo funcionan las incrustaciones en redes sociales y qué metaetiquetas se pueden usar para mejorar la incrustación de artículos, como las entradas del blog.

En el próximo capítulo, Capítulo 9, Implementación de pruebas de extremo a extremo con Playwright, aprenderemos a escribir pruebas de extremo a extremo para nuestra interfaz de usuario configurando Playwright. Luego, vamos a escribir algunas pruebas frontend para nuestra aplicación de blog.

Implementación de pruebas de extremo a extremo

Usando Dramaturgo

En los capítulos anteriores, escribimos pruebas unitarias para nuestro backend con Jest. Ahora, aprenderemos a escribir y ejecutar pruebas integrales en nuestra interfaz de usuario con Playwright. Primero, configuraremos Playwright en nuestro proyecto y VS Code para permitir la ejecución de pruebas frontend. A continuación, escribiremos algunas pruebas frontend para nuestra aplicación. A continuación, aprenderemos a reutilizar configuraciones de prueba con dispositivos. Finalmente, aprenderemos a ver informes de pruebas y a ejecutar Playwright en CI con GitHub Actions.

En este capítulo cubriremos los siguientes temas principales:

- Configuración de Playwright para pruebas de extremo a extremo
- Escribir y ejecutar pruebas de extremo a extremo
- Configuraciones de prueba reutilizables utilizando accesorios
- Visualización de informes de pruebas y ejecución en CI

Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo full-stack, y el Capítulo 2, Conozca Node.js y MongoDB.

Las versiones que se listan en esos capítulos son las que se utilizan en todo el libro. Si bien instalar una versión más reciente no debería ser un problema, tenga en cuenta que ciertos pasos podrían funcionar de forma diferente en una versión más reciente. Si tiene algún problema con el código y los pasos proporcionados en este libro, intente utilizar las versiones mencionadas en la sección de Requisitos técnicos de los Capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-React-Full-Stack-Modernos/árbol/principal/capítulo9>.

Si clonó el repositorio completo del libro, es posible que Husky no encuentre el directorio .git al ejecutar npm install. En ese caso, simplemente ejecute git init en la raíz de la carpeta del capítulo correspondiente.

El enlace de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/WjwEwUR8g2c>

Configuración de Playwright para pruebas de extremo a extremo

Playwright es un ejecutor de pruebas que facilita las pruebas integrales en diversos motores de renderizado web, como Chromium (Chrome, Edge, Opera, etc.), WebKit (Safari) y Firefox. Puede ejecutar pruebas en Windows, Linux y macOS, localmente o en CI. Playwright se puede ejecutar de dos maneras:

- Encabezado: Abre una ventana del navegador donde se puede ver lo que está haciendo el dramaturgo.
- Sin cabeza: ejecuta el motor de renderizado en segundo plano y solo muestra los resultados de las pruebas en la Terminal o un informe de prueba generado

En este capítulo, exploraremos ambas formas de ejecutar Playwright. Ahora, instalaremos Playwright en nuestro proyecto.

Instalación de Playwright

Para instalar Playwright, podemos usar npm init playwright, que ejecuta un comando que instala Playwright, crea una carpeta para las pruebas de extremo a extremo, agrega un flujo de trabajo de GitHub Actions para ejecutar pruebas en CI e instala los navegadores de Playwright para que pueda ejecutar pruebas en varios motores. Sigue estos pasos para instalar Playwright:

1. Copie la carpeta ch8 existente a una nueva carpeta ch9, de la siguiente manera:

```
$ cp -R ch8 ch9
```

2. Abra la carpeta ch9 en VS Code y abra una nueva Terminal.

3. Ejecute el siguiente comando:

```
$ npm init dramaturgo@1.17.131
```

Nota

Por lo general, es una buena idea instalar la última versión aquí ejecutando npm init playwright@Última versión. Sin embargo, para garantizar que las instrucciones de este libro sean reproducibles incluso cuando se publiquen nuevas versiones con cambios importantes, fijamos la versión aquí.

4. Cuando se le pregunte si desea continuar con la instalación del paquete create-playwright, presione Enter para confirmar. \$en select JavaScript. En cuanto al nombre del directorio, mantenga las pruebas. Nombre predeterminado y presione Enter para confirmar. Escriba "y" para agregar un flujo de trabajo de GitHub Actions. Escribe "y" de nuevo para instalar los navegadores Playwright. La descarga e instalación de los diferentes motores de navegación tardará un tiempo .

5. Necesitamos ajustar algunos archivos para que Playwright funcione con los módulos ES. Edite `playwright.config.js` y cambie la línea con la importación `require()` al principio del archivo por lo siguiente:

```
importar { defineConfig, dispositivos } desde '@playwright/test'
```

6. Además, cambie la exportación de `module.exports` a lo siguiente:

```
exportar predeterminado defineConfig({
```

7. Elimine la carpeta `tests-examples/` y el archivo `tests/example.spec.js`.

Después de instalar Playwright, necesitamos preparar nuestro backend para las pruebas de extremo a extremo, así que hágámoslo ahora.

Preparación del backend para pruebas de extremo a extremo

Para preparar el backend para las pruebas integrales, necesitamos iniciar una instancia del backend con el servidor MongoDB en memoria, de forma similar a como hicimos para las pruebas de Jest. Hagámoslo ahora:

1. Cree un nuevo archivo `backend/src/e2e.js`. Dentro, importe `dotenv`, `globalSetup` y las funciones `app` e `initDatabase`:

```
importar dotenv desde 'dotenv' dotenv.config()
```

```
importar globalSetup desde './test/globalSetup.js' importar { app } desde './app.js' importar { initDatabase } desde './db/init.js'
```

2. \$en, define una nueva función asíncrona para ejecutar un servidor de pruebas:

```
función asíncrona runTestingServer() {
```

3. Dentro de esta función, primero ejecutamos la función `globalSetup`, que ejecuta un servidor MongoDB en memoria. \$en, inicializamos la base de datos y ejecutamos la aplicación Express:

```
esperar globalSetup() esperar
initDatabase()
const PORT = process.env.PORT app.listen(PORT)
console.info('PRUEBA')
servidor express ejecutándose en http://
localhost:${PUERTO} } }
```

4. Finalmente, ejecutamos la función definida:

```
ejecutarServidorDePruebas()
```