

5. Edite backend/package.json y agregue un nuevo script para ejecutar el archivo e2e.js:

```
"e2e": "nodo src/e2e.js",
```

6. En la raíz del proyecto, instale simultáneamente una herramienta para ejecutar dos comandos en paralelo:

```
$ npm install --save-dev concurrentemente@8.2.2
```

Vamos a utilizar esta herramienta para ejecutar el backend y el frontend en paralelo.

7. Edite package.json en la raíz del proyecto y defina un script e2e que se ejecutará

Scripts e2e:client y e2e:server en paralelo:

```
"e2e": "simultáneamente \"npm run e2e:client\" \"npm run e2e:server\"",
```

8. Ahora, defina el script e2e:client, en el que simplemente ejecutamos el frontend prediseñado:

```
"e2e:client": "npm run compilación y npm run inicio",
```

Por razones de rendimiento, no ejecutamos el servidor de desarrollo. De lo contrario, ralentizaríamos nuestras pruebas de principio a fin. Podríamos omitir el script de compilación, pero entonces debemos recordar compilar nuestro frontend después de realizar cambios antes de ejecutar las pruebas, y esto también debe hacerse en CI. Alternativamente, al ejecutar las pruebas localmente, especialmente cuando solo ejecutamos ciertas pruebas y no todas, podríamos ejecutar el servidor de desarrollo en lugar de compilar.

9. \$en, definimos el script e2e:server, que ejecuta el script e2e en la carpeta backend:

```
"e2e:server": "cd backend/ && npm run e2e",
```

10. Edite playwright.config.js y configure la baseURL cambiando la siguiente línea:

```
usu: {  
    /* URL base para utilizar en acciones como `await page.goto('/')`.  
    */  
    URL base: 'http://localhost:5173',
```

11. Finalmente, edite playwright.config.js y reemplace la configuración webServer en la parte inferior del archivo con lo siguiente:

```
Servidor web: {  
    comando: 'npm run e2e',  
    URL: 'http://localhost:5173',  
},
```

Ahora que hemos configurado con éxito Playwright y preparado el backend para pruebas de extremo a extremo, ¡comencemos a escribir y ejecutar pruebas de extremo a extremo!

## Escritura y ejecución de pruebas de extremo a extremo

Ahora escribiremos y ejecutaremos nuestra primera prueba integral con Playwright. Empecemos con una prueba sencilla que verifica que hemos optimizado correctamente nuestro título para los motores de búsqueda. Sigue estos pasos para escribir y ejecutar tu primera prueba integral:

1. Cree un nuevo archivo tests/seo.spec.js. Dentro de este archivo, comprobaremos si...

El título de nuestra página está configurado correctamente.

2. Dentro de este archivo recién creado, primero importe las funciones de prueba y espera de @playwright/prueba:

```
importar { prueba, esperar } de '@playwright/prueba'
```

3. \$en, definimos una prueba en la que verificamos si el título del blog está configurado correctamente:

```
test('tiene título', async ({ página }) => {
```

Como puede ver, la función de prueba es similar a cómo definimos las pruebas en Jest. Playwright además nos permite acceder a contextos especiales en nuestra prueba, llamados `xtures`. El `xture` de página es el `xture` más esencial en Playwright y nos permite acceder a las funciones del navegador e interactuar con una página.

4. Dentro de la prueba, primero navegamos a la URL de nuestro frontend usando la función `page.goto`:

```
esperar página.goto('/')
```

5. \$en, usamos la función `expect` para verificar si la página muestra el título correcto:

```
esperar expect(página).toHaveTitle('Blog de React de pila completa')  
})
```

Como podemos ver, la sintaxis de Playwright es muy similar a la de Jest. También contamos con una función de expectativa para realizar afirmaciones, como que la página tenga un título determinado.

6. Antes de ejecutar las pruebas, asegúrese de que el contenedor Docker dbserver se esté ejecutando.

7. Ahora podemos ejecutar esta prueba abriendo una nueva Terminal y ejecutando el siguiente comando:

```
$ npx prueba de dramaturgo
```

¡Asegúrate de estar en la raíz de nuestro proyecto (carpeta ch9), y no dentro de la carpeta backend, cuando ejecutes este comando!

Verá que Playwright ejecuta nuestra prueba tres veces (en Chromium, Firefox y Webkit) y que todas ellas pasaron correctamente. La siguiente captura de pantalla muestra el resultado de ejecutar Playwright en la línea de comandos:

```
→ ~/D/F/ch9 ↵ main± > npx playwright test

Running 3 tests using 3 workers
3 passed (15.4s)

To open last HTML report run:

npx playwright show-report
```

Figura 9.1 – ¡Ejecutando nuestra primera prueba en Playwright!

Ahora que hemos ejecutado nuestra prueba con éxito, pasemos a ejecutar pruebas utilizando la extensión VS Code.

## Usando la extensión VS Code

En lugar de ejecutar manualmente todas las pruebas desde la línea de comandos, también podemos ejecutar pruebas específicas (o todas) con una extensión de VS Code, similar a lo que hicimos para Jest. Además, la extensión nos permite obtener una visión general de qué pruebas se ejecutan correctamente (o no), inspeccionarlas mientras se ejecutan en un navegador e incluso registrar nuestras interacciones en el navegador y generar pruebas a partir de ellas.

Comencemos configurando la extensión VS Code y ejecutando nuestra prueba desde ella:

1. Abra la pestaña Extensiones en VS Code y busque Playwright.
2. Haga clic en el botón Instalar para instalar Playwright Test para VS Code de Microso.
3. Haga clic en la pestaña Prueba en VS Code (el ícono %ask), que también usamos para la extensión Jest. Aquí, ahora verás Jest y Playwright en la lista.
4. Expanda la ruta Playwright | pruebas , haga clic en seo.spec.js para cargar el archivo y luego haga clic en el ícono Reproducir junto a seo.spec.js para ejecutar la prueba.

Como podemos ver en la siguiente captura de pantalla, la prueba se ejecutó con éxito y todas las pruebas están pasando:

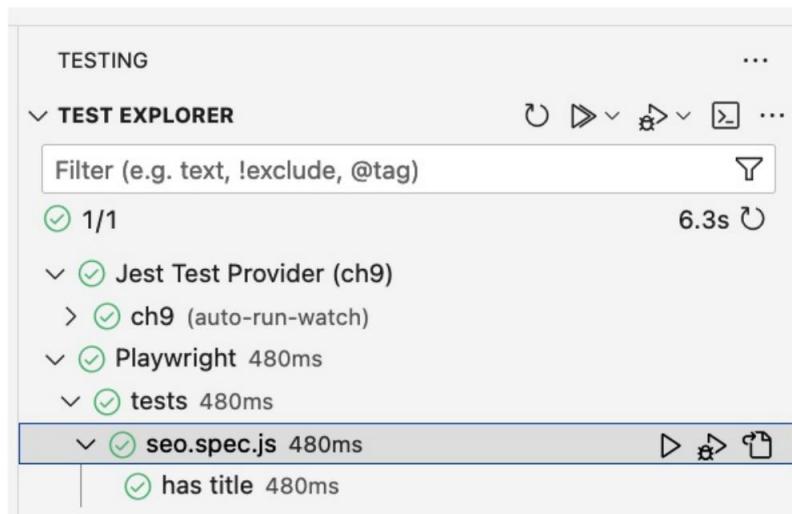


Figura 9.2 – ¡Nuestra prueba de dramaturgo se ejecuta con éxito desde la extensión VS Code!

Ahora que hemos ejecutado con éxito las pruebas en la extensión VS Code en modo sin cabeza, pasemos a ejecutarlas en modo con cabeza, donde mostramos lo que Playwright está haciendo en el navegador mientras ejecuta las pruebas.

#### Mostrar el navegador mientras se ejecutan pruebas

La extensión \$e Playwright de VS Code también cuenta con la útil opción "Mostrar navegador" , que abre el navegador mientras se ejecutan las pruebas. \$is nos permite depurar las pruebas o el frontend mientras se ejecutan. Probémoslo ahora:

1. En la parte inferior de la barra lateral de Pruebas , marque la casilla Mostrar navegador en la parte inferior de la barra lateral y ejecute la prueba nuevamente.

Se abrirá una ventana del navegador y se ejecutará la prueba. Sin embargo, nuestra prueba es muy rápida y sencilla, por lo que se ejecuta en poco tiempo y no hay mucho que ver.

2. Para inspeccionar mejor la prueba, podemos usar el visor de trazas. Marque la opción "Mostrar visor de trazas" en la parte inferior de la barra lateral de Pruebas y vuelva a ejecutar la prueba. Verá la siguiente ventana abierta:

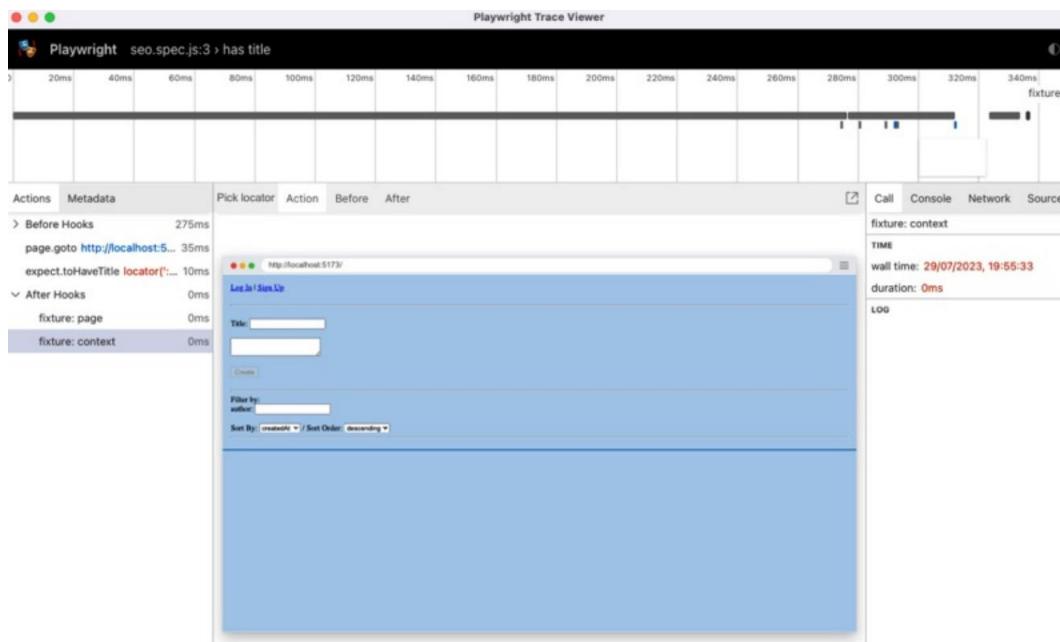


Figura 9.3 – El visor de trazas de Playwright

Como podemos ver, el visor de seguimiento de Playwright nos muestra que la prueba ejecutó `page.goto` y luego `expect.toHaveTitle`. También muestra el estado de la aplicación en cada paso de la prueba. En nuestro caso, solo tenemos un paso. Esta función es especialmente útil al desarrollar pruebas más grandes y complejas.

#### Nota

También es posible ejecutar Playwright en modo UI, lo que abre la aplicación Playwright en una ventana independiente que permite ejecutar las pruebas por separado y ver su ejecución, de forma similar a usar la función "Mostrar visor de seguimiento" en la extensión de VS Code. Puedes ejecutar Playwright en modo UI con el siguiente comando: `npx playwright test --ui`

Ahora que hemos aprendido a usar la extensión para ejecutar pruebas, podemos pasar a una función muy útil: registrar acciones para crear una nueva prueba. Hagámolo ahora.

## Grabando una prueba

La extensión \$e Playwright también puede grabar nuevas pruebas. Ahora, creamos una nueva prueba para la página de registro usando la función de grabación de pruebas de la extensión VS Code:

1. A diferencia de ejecutar una prueba de Playwright, la grabadora de pruebas no inicia automáticamente el frontend ni el backend, por lo que primero debemos iniciarlos manualmente. Abra una nueva terminal y ejecute el siguiente comando:

```
$ npm ejecuta e2e
```

2. En la sección inferior de la barra lateral de Pruebas , haga clic en "Grabar nuevo". Se abrirá una ventana del navegador.
3. En la ventana del navegador, navegue hasta la interfaz pegando `http://localhost:5173/` en la barra de URL.

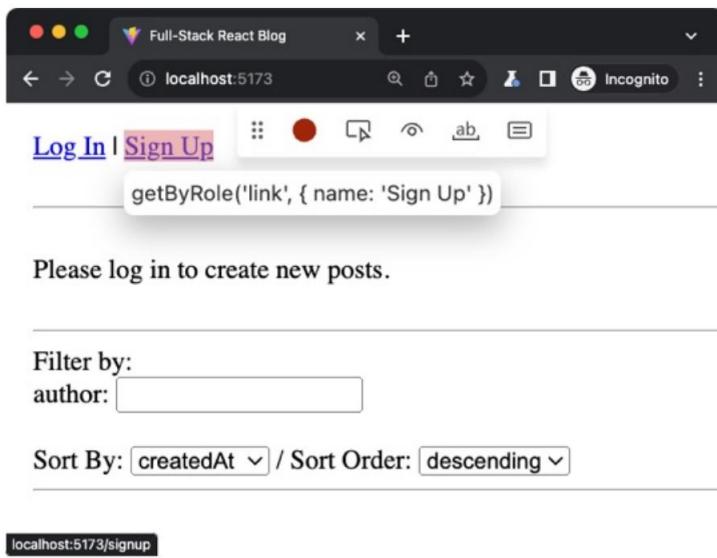


Figura 9.4 – La grabadora de prueba de Playwright al pasar el cursor sobre el enlace "Registrarse"

4. Luego, haga clic en el enlace Registrarse . Se debería abrir la página de registro.
5. En esta nueva página, ingrese un nombre de usuario y una contraseña; por ejemplo, prueba y prueba. Luego presione el botón Registrarse .
6. Serás redirigido a la página de inicio de sesión. Ahora, inicia sesión con el mismo nombre de usuario y contraseña. como antes.
7. Serás redirigido a la página principal y habrás iniciado sesión como prueba. Ahora puedes cerrar la ventana del navegador. Verás que dentro de VS Code tienes un nuevo archivo `test-1.spec.ts`. ¡que contiene todas las acciones que acabamos de realizar en el navegador!

8. Guarde el archivo y cierre el script extremo a extremo que ejecuta el backend y el frontend. Ahora verá el archivo test-1.spec.ts en la barra lateral de Pruebas . Si intenta ejecutar la prueba, notará que se bloquea en la parte de inicio de sesión, ya que nuestra prueba no espera la redirección a la página de inicio de sesión.

Si bien grabar pruebas es una función útil para acelerar la escritura de pruebas de principio a fin, no siempre nos permite escribir pruebas funcionales. Ahora necesitamos limpiar nuestra prueba grabada y añadirle aserciones.

Como referencia, aquí está el código completo generado por la grabadora de pruebas Playwright:

```
importar { prueba, esperar } de '@playwright/prueba'

prueba('prueba', async ({ página }) => {
    esperar página.goto('http://localhost:5173/')
    await page.getByRole('link', { name: 'Registrarse' }).click()
    esperar página.getByLabel('Nombre de usuario:').click()
    esperar página.getByLabel('Nombre de usuario:').fill('prueba')
    esperar página.getByLabel('Contraseña:').click()
    esperar página.getByLabel('Contraseña:').fill('prueba')
    await page.getByRole('botón', { nombre: 'Registrarse' }).click()
    esperar página.getByLabel('Nombre de usuario:').click()
    esperar página.getByLabel('Nombre de usuario:').fill('prueba')
    esperar página.getByLabel('Contraseña:').click()
    esperar página.getByLabel('Contraseña:').fill('prueba')
    esperar página.getByRole('botón', { nombre: 'Iniciar sesión' }).click()
})
```

Ahora que hemos grabado una prueba, limpiémosla para que se ejecute correctamente.

## Limpieza y finalización de la prueba grabada

Si revisas la prueba, verás que contiene todas las acciones que realizamos en el navegador, pero no verifica que hayamos iniciado sesión correctamente. Tampoco espera a que las páginas terminen de cargarse y algunos comparadores no coinciden con el texto correcto. Corrijamos estos problemas ahora:

1. Cambie el nombre de tests/test-1.spec.ts a tests/auth.spec.js.
2. Edite tests/auth.spec.js y cambie el nombre de la prueba para permitir el registro e inicio de sesión:

```
test('permite registrarse e iniciar sesión', async ({ page }) => {
```

3. Necesitamos definir un nombre de usuario único para poder ejecutar nuestra prueba varias veces sin tener que  
Para reiniciar el backend para borrar el servidor de memoria MongoDB:

```
const testUser = 'prueba' + Fecha.ahora()
```

**Nota**

Es importante no registrarse dos veces con el mismo nombre de usuario, ya que la base de datos MongoDB en memoria se reutiliza para todas las pruebas. Asegúrese de que las pruebas se ejecuten de forma independiente y no dependan de datos de otros archivos de prueba, ya que estos podrían ejecutarse en cualquier orden. Solo se garantiza el orden dentro de un mismo archivo de prueba. El uso de Date.now() devuelve la hora actual en milisegundos y es prácticamente seguro contra colisiones, siempre que no se ejecuten demasiadas pruebas en paralelo. Para una solución más segura contra colisiones, podría utilizar un generador de UUID.

4. Cambie la URL de page.goto() a / para asegurarse de que utilice la URL base que configuramos anteriormente:

```
esperar página.goto('/')
```

5. Complete el nombre de usuario generado al registrarse:

```
esperar página.getLabel('Nombre de usuario:').fill(testUser)
```

6. Despues de hacer clic en el botón Registrarse , espere a que la URL se actualice mediante la siguiente función:

```
await page.getByRole('botón', { nombre: 'Registrarse' }).click()  
esperar página.waitForURL("**/login")
```

Es necesario esperar a que se cargue la página siguiente porque la grabación no admite la detección de carga de página en este momento y, de lo contrario, volvería a ejecutar los comandos en la página anterior o durante la redirección, lo que provocaría que la prueba fallara.

7. Para iniciar sesión, también completamos el nombre de usuario generado:

```
esperar página.getLabel('Nombre de usuario:').fill(testUser)
```

8. Despues de eso, haz clic en el botón Iniciar sesión y espera a que la URL se actualice nuevamente:

```
esperar página.getByRole('botón', { nombre: 'Iniciar sesión' }).click()  
esperar página.waitForURL(**"/")
```

9. Para que coincida más fácilmente con el componente Header de React, edite src/components/Header.

jsx y convierte los elementos <div> en elementos <nav>:

```
función de exportación Header() {  
    const [token, setToken] = useAuth()  
  
    si (token) {  
        constante { sub } = jwtDecode(token)  
        devolver (  
            <navegación>  
            Inició sesión como <User id={sub} />  
            <br />  
            <button onClick={() => setToken(null)}>Cerrar sesión</button>
```

```
        </nav>
    )
}

devolver (
    <navegación>
        <Link to='/login'>Iniciar sesión</Link> | <Link to='/signup'>Registrarse
    Arriba</Link>
    </nav>
)
}
```

10. Al final de la prueba, ahora agregamos una afirmación que verifica si el encabezado (elemento `<nav>`) contiene el texto Iniciado sesión como y el nombre de usuario generado:

```
, await expect(page.locator('nav')).toContainText('Iniciado sesión como + testUser')
})
```

Usar `toContainText` en lugar de `toHaveText` garantiza que el texto no tenga que ser exactamente la cadena proporcionada. En nuestro caso, el texto "Cerrar sesión" también forma parte del elemento `<nav>`, por lo que el texto completo sería "Iniciar sesión" como `testXXXXLogout`.

11. Ejecute la prueba utilizando la extensión VS Code o ejecutando la prueba `npx playwright` comando en la Terminal (la que prefieras) ¡y verás que ahora se ejecuta correctamente!

#### Nota

Si la prueba no se ejecuta correctamente, es posible que haya registrado accidentalmente acciones adicionales y no las haya limpiado correctamente. Compare su prueba con el ejemplo de código de este libro para asegurarse de que esté correctamente definida y limpia.

Ahora que sabemos cómo funciona la definición de pruebas básicas en Playwright, aprendamos acerca de las configuraciones de pruebas reutilizables mediante accesorios.

### Configuraciones de prueba reutilizables mediante accesorios

Tras crear la prueba de autenticación, podrías pensar: ¿qué pasa si quiero definir una prueba para crear una nueva publicación? Primero tendríamos que registrarnos, iniciar sesión y crear la publicación. Esto es bastante tedioso, y cuanto más complejas sean nuestras pruebas, más tediosas serán las pruebas de definición. Afortunadamente, Playwright tiene una solución para este tipo de problemas. Playwright introduce un concepto llamado "xtures", que son contextos para la prueba que pueden contener funciones reutilizables. Por ejemplo, podríamos definir una "xture" de autenticación para proporcionar funciones de registro e inicio de sesión a todas las pruebas.

Cuando usábamos Jest, usábamos ganchos before/aer para preparar el entorno común para múltiples pruebas. Los accesorios tienen algunas ventajas sobre los ganchos before/aer. Principalmente, encapsulan la configuración y el desmontaje en un mismo lugar, son reutilizables entre archivos de prueba, componibles y más flexibles. Además, los accesorios se proporcionan a pedido, lo que significa que Playwright solo configurará los accesorios necesarios para ejecutar una determinada prueba.

El dramaturgo también incluye algunos accesorios listos para usar, que vamos a conocer ahora.

#### Descripción general de los accesorios empotrados

El dramaturgo viene con algunos accesorios incorporados, uno de los cuales ya hemos conocido: la página Accesorios. Ahora vamos a presentar brevemente los accesorios integrados más importantes que ofrece Playwright. Fuera de la caja:

- Navegador: permite controlar las funciones del navegador, como abrir una nueva página.
- browserName: contiene el nombre del navegador que actualmente está ejecutando la prueba
- página: Es, con diferencia, el accesorio integrado más importante, que se utiliza para controlar las interacciones con la página. Visitar URL, hacer coincidir elementos, realizar acciones y mucho más
- contexto: Un contexto aislado para la ejecución de la prueba actual
- solicitud: se utiliza para realizar solicitudes API de Playwright

Ahora que hemos aprendido sobre los accesorios integrados que proporciona Playwright, continuemos definiéndolos. Nuestro propio futuro.

#### Escribiendo nuestro propio fixture

Registrarse e iniciar sesión son acciones comunes que necesitaremos realizar en nuestras pruebas integrales, por lo que son el caso ideal para crear un dispositivo. Siga estos pasos para crear un nuevo dispositivo de autenticación:

1. Cree una nueva carpeta pruebas/accesorios/.
2. Dentro de él, creamos un nuevo archivo tests/fixtures/AuthFixture.js, donde definimos una Clase AuthFixture:

```
clase de exportación AuthFixture {
```

3. Esta clase recibirá el archivo xture de la página en el constructor:

```
constructor(página) {  
    esta.pagina = pagina  
}
```

4. Defina un método signUpAndLogIn, que sigue las acciones de la prueba de autenticación para generar un nombre de usuario único, luego regístrese e inicie sesión como usuario:

```
registro e inicio de sesión asíncrono() {
    const testUser = 'prueba' + Fecha.ahora()
    espera esta.página.goto('/signup')
    esperar this.page.getByLabel('Nombre de usuario:').fill(testUser)
    esperar esto.page.getByLabel('Contraseña:').fill('contraseña')
    esperar this.page.getByRole('botón', { nombre: 'Registrarse' }).click()

    esperar esta.página.waitForURL('**/login')
    esperar this.page.getByLabel('Nombre de usuario:').fill(testUser)
    esperar esto.page.getByLabel('Contraseña:').fill('contraseña')
    esperar this.page.getByRole('botón', { nombre: 'Iniciar sesión' }).click()

    esperar esta.página.waitForURL('**/')
    devolver usuario de prueba
}
```

5. Crea un nuevo archivo tests/fixtures/index.js. Dentro, importa la función de prueba de Playwright (renombrándola como baseTest) y el AuthFixture que acabamos de definir:

```
importar { prueba como baseTest } desde '@playwright/test'
importar { AuthFixture } desde './AuthFixture.js'
```

6. Sen, define y exporta una nueva función de prueba, extendiendo la función baseTest de Dramaturgo definiendo una nueva autoridad en su interior:

```
exportar const test = baseTest.extend({
    auth: async ({ página }, uso) => {
        constante authFixture = nueva AuthFixture(página)
        esperar uso(authFixture)
    },
})
```

Consejo

También es posible realizar una configuración adicional del contexto del dispositivo antes de llamar a use() Función y un desglose adicional tras llamarla. \$is se puede usar, por ejemplo, para crear un conjunto de publicaciones de ejemplo antes de ejecutar pruebas y eliminarlas posteriormente. Si el backend pudiera eliminar un usuario, crear un usuario temporal y eliminar el nombre de usuario creado después de usar el complemento sería una mejor opción para solucionar el problema de las colisiones de nombres de usuario.

7. Además, vuelva a exportar la función de espera desde Playwright para facilitar la importación de nuestros fixtures:

```
exportar { esperar } desde '@playwright/test'
```

Ahora que hemos definido nuestro accesorio personalizado, ¡usémoslo mientras creamos una nueva prueba!

### Uso de accesorios personalizados

Ahora definiremos una prueba integral para crear una nueva publicación. Para crear una publicación, necesitamos iniciar sesión para poder usar nuestra herramienta de autenticación y preparar el entorno. Siga estos pasos para definir la nueva prueba y usar nuestra herramienta personalizada:

1. Cree un nuevo archivo tests/create-post.spec.js. Para usar el dispositivo personalizado, necesitamos importar las funciones de prueba y espera del archivo fixtures/index.js:

```
importar { prueba, esperar } desde './fixtures/index.js'
```

2. Defina una nueva prueba para verificar que la creación de publicaciones funciona, utilizando la página y los accesorios de autorización:

```
test('permite crear una nueva publicación', async ({ page, auth }) => {
```

3. Ahora podemos usar el método signUpAndLogin de nuestro dispositivo de autenticación personalizado para crear e iniciar sesión como nuevo usuario:

```
constante testUser = await auth.signUpAndLogin()  
})
```

4. Podemos usar la generación de código de Playwright para grabar nuestra prueba. Primero, guarde el archivo y ejecute la prueba create-post.spec.js con la opción "Mostrar navegador" activada.

5. ¿Sí, crea una nueva línea después de llamar a la función auth.signUpAndLogin y presiona Grabar en el cursor.

6. Ahora podemos grabar acciones desde la ventana del navegador abierta (que ya tiene la sesión iniciada, ya que se invocaron los métodos de fixture). Haga clic en el campo de título e introduzca "Publicación de prueba" como título de la entrada. Luego, presione Tab para ir al siguiente campo, introduzca "¡Hola mundo!" como contenido de la entrada. Luego, presione Tab de nuevo y presione Retorno/Intro para crear una nueva entrada.

#### Nota

La publicación no se crea realmente, ya que Playwright cierra el backend justo después de ejecutarlo, por lo que, al momento de la grabación, el backend ya está cerrado. Si desea grabar con el backend en ejecución, explore webServer.reuseExistingServer. configuración en playwright.config.js.

7. Regresa al archivo y verás que todas las acciones se registraron correctamente. El siguiente código debería haberse registrado:

```
esperar página.getByLabel('Título:').click()
await page.getByLabel('Título:').fill('Publicación de prueba')
esperar página.getByLabel('Título:').press('Tab')
await page.locator('textareá').fill('¡Hola mundo!')
esperar página.locator('textareá').press('Tab')
esperar página.getByRole('botón', { nombre: 'Crear' }).press('Entrar')
```

8. Ahora solo nos falta añadir una comprobación para comprobar si la publicación se creó correctamente:

```
esperar esperar(página.getText('Publicación de prueba escrita por $ {testUser}')).toBeVisible()
```

Como estamos controlando el entorno de pruebas, basta con comprobar que en la página esté visible el texto Test PostWritten by testXXX (sin espacio entre “Post” y “Written”). \$is nos dirá que el post fue creado en la lista.

9. Ejecuta la prueba y verás que ¡pasa exitosamente!

Podríamos crear accesorios adicionales para gestionar publicaciones (crear, editar, eliminar) y usarlos para, por ejemplo, verificar que el enlace a una sola publicación funcione correctamente y ajuste el título en consecuencia. Sin embargo, extender las pruebas de extremo a extremo de esa manera es similar a lo que ya hemos hecho y, por lo tanto, es más sencillo. como ejercicio para ti.

## Visualización de informes de pruebas y ejecución en CI

Después de crear con éxito algunas pruebas de extremo a extremo para nuestra aplicación de blog, concluyamos el capítulo aprendiendo cómo ver informes de pruebas HTML y cómo ejecutar Playwright en CI.

### Visualización de un informe HTML

Playwright genera automáticamente informes HTML de las pruebas. Podemos generarlo ejecutando el siguiente comando para ejecutar todas las pruebas:

```
$ npx prueba de dramaturgo
```

\$en, ejecute el siguiente comando para servir y ver el informe HTML de la última ejecución:

```
$ npx informe del espectáculo del dramaturgo
```

El informe debería abrirse en una nueva ventana del navegador y tener el siguiente aspecto:

Search	All 9	Passed 9	Failed 0	Flaky 0	Skipped 0
Total time: 7.0s					
<b>auth.spec.js</b>					
✓ <b>allows sign up and log in</b> chromium					764ms
auth.spec.js:3					
✓ <b>allows sign up and log in</b> firefox					1.4s
auth.spec.js:3					
✓ <b>allows sign up and log in</b> webkit					880ms
auth.spec.js:3					
<b>create-post.spec.js</b>					
✓ <b>allows creating a new post</b> chromium					772ms
create-post.spec.js:3					
✓ <b>allows creating a new post</b> firefox					1.2s
create-post.spec.js:3					
✓ <b>allows creating a new post</b> webkit					653ms
create-post.spec.js:3					
<b>seo.spec.js</b>					
✓ <b>has title</b> chromium					303ms
seo.spec.js:3					
✓ <b>has title</b> firefox					722ms
seo.spec.js:3					
✓ <b>has title</b> webkit					324ms
seo.spec.js:3					

Figura 9.5 – Un informe de prueba HTML generado por Playwright

Como podemos ver, nuestras tres pruebas se ejecutaron correctamente en los tres navegadores. Haga clic en una de las pruebas para ver todos los pasos ejecutados en detalle.

## Ejecución de pruebas de Playwright en CI

Al inicializar Playwright, se nos preguntó si queríamos generar un archivo CI de GitHub Actions. Aceptamos, así que Playwright generó automáticamente una configuración de CI en `.github/workflows/playwright.yml` le. \$is work%ow extrae el repositorio, instala todas las dependencias, instala los navegadores de Playwright, ejecuta todas las pruebas de Playwright y luego carga el informe como un artefacto para que pueda visualizarse desde la ejecución de CI. Aún necesitamos ajustar el work%ow de CI para que también instale las dependencias de nuestro backend, así que hagámoslo ahora:

1. Edite `.github/workflows/playwright.yml` y agregue el siguiente paso:

```
- nombre: Instalar dependencias
  ejecutar: npm ci
- nombre: Instalar dependencias del backend
  ejecutar: cd backend/ && npm ci
```

El comando `$e npm ci` garantiza que el proyecto ya tenga un archivo `package-lock.json` y no escriba un archivo de bloqueo, lo que garantiza un estado limpio para que se ejecute CI.

2. Agregue, confirme y envíe todo a un repositorio de GitHub para ver Playwright ejecutándose en CI.

### Nota

Asegúrate de crear un nuevo repositorio solo con el contenido de la carpeta ch9 (¡no con toda la carpeta Full-Stack-React-Projects!), de lo contrario, GitHub Actions no detectará la carpeta `.github`.

3. Vaya a GitHub, haga clic en la pestaña Acciones , seleccione el flujo de trabajo Playwright Tests en la barra lateral, y luego haga clic en el último flujo de trabajo ejecutado.
4. En la parte inferior de la carrera, hay una sección de Artefactos , que contiene un informe del dramaturgo. objeto que se puede descargar para ver el informe HTML.

La siguiente captura de pantalla muestra las pruebas de Playwright ejecutándose en GitHub Actions, con el informe proporcionado como un artefacto:

The screenshot shows the GitHub Actions interface for a repository named 'omnidan / ch9'. The 'Actions' tab is selected, showing a recent run titled 'ci: define playwright tests #4'. The run was triggered via push 3 minutes ago by 'omnidan pushed -> 4a15ad2 main'. The status is 'Success' with a total duration of '1m 58s' and billable time of '2m', resulting in 1 artifact. The workflow file 'playwright.yml' is displayed, showing a single job named 'test' which completed successfully in '1m 48s'. Below the workflow, the 'Artifacts' section shows a single artifact named 'playwright-report' (418 KB) produced during runtime. On the left sidebar, the 'Summary' tab is active, followed by 'Jobs', 'test' (which is checked), 'Run details', 'Usage', and 'Workflow file'.

Figura 9.6 – Dramaturgo ejecutándose en Acciones de GitHub

Como podemos ver, ejecutar Playwright en CI utilizando la plantilla proporcionada es simple y directo.

## Resumen

En este capítulo, aprendimos a usar Playwright para pruebas integrales. Primero, configuramos Playwright en nuestro proyecto y preparamos nuestro backend para las pruebas integrales. Luego, escribimos y ejecutamos nuestra primera prueba.

A continuación, aprendimos sobre las herramientas para crear contextos de prueba reutilizables. Finalmente, revisamos el informe HTML generado y configuramos CI para ejecutar Playwright, generar un informe y guardarlo como un artefacto en la canalización.

En el próximo capítulo, Capítulo 10, Agregación y visualización de estadísticas usando MongoDB y Victory, aprenderemos cómo agregar datos usando MongoDB y exponer estos datos agregados a través de un backend. Luego, consumiremos los datos agregados en el frontend y los visualizaremos usando Victory con varios tipos de visualización.

Machine Translated by Google

# 10

## Agregación y visualización Estadísticas usando MongoDB y Victory

En este capítulo, aprenderemos a recopilar, agregar y visualizar estadísticas para nuestra aplicación de blog usando MongoDB y Victory. Comenzaremos aprendiendo cómo recopilar eventos de los usuarios que visualizan las entradas del blog. Luego, generamos algunos eventos aleatoriamente para tener un conjunto de datos con el que trabajar. Usamos este conjunto de datos para aprender a agregar datos con MongoDB y generar estadísticas resumidas, como el número de visitas por entrada o la duración promedio de la sesión. Este tipo de información ayudará a los autores a conocer el rendimiento de sus entradas. Finalmente, crearemos gráficos para visualizar estas estadísticas agregadas usando la biblioteca Victory.

En este capítulo cubriremos los siguientes temas principales:

- Recopilación y simulación de eventos
- Agregación de datos con MongoDB
- Implementación de agregación de datos en el backend
- Integración y visualización de datos en el frontend usando Victory

## 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/cap10>.

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/DmSq2P\\_IQQs](https://youtu.be/DmSq2P_IQQs).

## Recopilación y simulación de eventos

Antes de comenzar a agregar y visualizar estadísticas, necesitamos recopilar (y posteriormente simular) eventos que usaremos para crear las estadísticas. Empezaremos por pensar qué datos queremos recopilar y cuáles nos resultarían útiles. Por ahora, nos centraremos en las visualizaciones de publicaciones , por lo que nos gustaría mostrar las siguientes estadísticas por publicación:

- Número total de visitas a una publicación
- Vistas diarias de una publicación
- Duración promedio diaria de visualización de una publicación

Comencemos creando el modelo de base de datos de eventos que nos permitirá mostrar estas estadísticas.

## Creando el modelo de evento

Para crear estas estadísticas, necesitamos recopilar eventos de los usuarios. Los eventos contendrán una referencia a una publicación, un ID de sesión para rastrear eventos de la misma visualización, una acción (inicio de visualización, finalización de visualización) y la fecha en que ocurrió el evento.

Comencemos a definir el modelo de base de datos para eventos:

1. Copie la carpeta ch9 existente a una nueva carpeta ch10, de la siguiente manera:

```
$ cp -R ch9 ch10
```

2. Abra la carpeta ch10 en VS Code.

3. Cree un nuevo archivo backend/src/db/models/event.js. Dentro de este archivo, defina un esquema que contiene una referencia a una publicación:

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

const eventsSchema = nuevo Esquema(
{
    publicación: { tipo: Schema.Types.ObjectId, ref: 'publicación', requerido: verdadero },
}
```

4. Define una sesión, acción y fecha:

```
sesión: { tipo: String, requerido: verdadero },
acción: { tipo: String, requerido: verdadero },
fecha: { tipo: Fecha, requerido: verdadero },
},
{ marcas de tiempo: verdadero },
)
```

5. Finalmente, exporte el modelo:

```
exportar const Evento = mongoose.model('eventos', eventsSchema)
```

Ahora que hemos definido el modelo de base de datos, continuemos definiendo una función de servicio y una ruta para rastrear eventos.

## Definición de una función de servicio y una ruta para rastrear eventos

Ahora que hemos definido con éxito nuestro modelo de base de datos para eventos, creemos una función de servicio y una ruta para rastrear nuevos eventos, de la siguiente manera:

1. Para generar los ID de sesión, usaremos la biblioteca uuid, que genera identificadores únicos universales (UUID) . Instálela ejecutando los siguientes comandos:

```
$ cd backend/
$ npm install uuid@9.0.1
```

2. Cree un nuevo archivo backend/src/services/events.js. Dentro de él, importe la función v4 desde el UUID y el modelo de eventos, y defina una función para crear un nuevo documento de evento, como se indica a continuación:

```
importar { v4 como uuidv4 } desde 'uuid'
importar { Event } desde './db/models/event.js'

exportar función asíncrona trackEvent({
    ID de publicación,
    acción,
```

```

sesión = uuidv4(), fecha =
Date.now(), }) { const evento
= new
Event({ post: postId, acción, sesión, fecha })

devolver esperar evento.save()
}

```

En los argumentos de la función, establecemos el ID de sesión predeterminado en un UUID generado aleatoriamente y la fecha en la fecha actual.

3. Cree un nuevo archivo backend/src/routes/events.js. Dentro de él, importe las funciones trackEvent y getPostById:

```

importar { trackEvent } desde '../services/events.js' importar { getPostById } desde '../
services/posts.js'

```

4. Definir una nueva ruta POST /api/v1/events, en la que obtenemos el postId, sesión, y acción desde el cuerpo:

```

función de exportación eventRoutes(app) { app.post('/api/
v1/events', async (req, res) => {
try
{ const { postId, sesión, acción } = req.body

```

5. \$en, verificamos si existe una publicación con el ID indicado en la base de datos. De no ser así, devolvemos un error 400.

Código de estado de solicitud incorrecta:

```

const post = await getPostById(postId) si (post === null)
devuelve res.status(400).end()

```

6. Si la publicación existe, obtenemos el ID de la sesión y usamos la función trackEvent para crear un nuevo evento:

```

        const event = await trackEvent({postId, sesión, acción
})
devolver res.json({ sesión: evento.session })
} atrapar (err) {
    console.error('acción de seguimiento de errores', err) return
    res.status(500).end()

}
}

```

7. Edite backend/src/app.js e importe eventRoutes:

```

importar { eventRoutes } desde './routes/events.js'

```

8. \$en monta las rutas a la aplicación:

```
postRoutes(aplicación)
userRoutes(aplicación)
eventRoutes(aplicación)
```

9. Inicie el backend de la siguiente manera (y manténgalo en ejecución para el desarrollo futuro):

```
$ cd backend/ $ npm
run dev
```

Ahora que hemos definido con éxito una ruta de backend para rastrear eventos, implementemos la recopilación de eventos en el frontend.

## Recopilación de eventos en el frontend

Tras definir la ruta, crearemos la función de API para el frontend y definiremos una forma de rastrear cuándo un usuario empezó y terminó de ver una publicación. Siga estos pasos para recopilar eventos en el frontend:

1. Cree un nuevo archivo src/api/events.js y defina una función postTrackEvent, que toma un objeto de evento y lo envía a la ruta previamente definida:

```
exportar const postTrackEvent = (evento) =>
  fetch(`${import.meta.env.VITE_BACKEND_URL}/eventos`, { método: 'POST', encabezados:
    { 'Tipo de contenido':
      'application/json',
    },
    cuerpo: JSON.stringify(evento),
  }). entonces((res) => res.json())
```

2. Edite src/pages/ViewPost.jsx e importe useEffect, useState y Ganchos useMutation:

```
importar { useEffect, useState } de 'react' importar { useQuery, useMutation }
de '@tanstack/react-query'
```

3. Además, importe la función API postTrackEvent:

```
importar { postTrackEvent } desde '../api/events.js'
```

4. Ahora, dentro de la función ViewPost, defina un nuevo gancho de estado para almacenar el ID de sesión y una mutación para rastrear los eventos. Cuando un evento se rastrea correctamente, obtenemos un ID de sesión del backend. Lo almacenamos en el gancho de estado:

```
const [sesión, setSession] = useState() const trackEventMutation =
useMutation({
```

```

mutationFn: (acción) => postTrackEvent({ postId, acción, sesión }),

onSuccess: (datos) => setSession(datos?.sesión),
})

```

5. \$en, definimos un nuevo gancho elect en el que rastreamos un evento startView un segundo después de que el usuario haya abierto la publicación (para evitar el seguimiento de eventos accidentales, como actualizaciones rápidas), y un evento endView cuando se desmonta el gancho elect. No le asignamos dependencias para garantizar que el gancho elect solo se active al montar y desmontar la página:

```

usarEfecto(() => {
    dejar que el tiempo de espera = setTimeout(() => {
        trackEventMutation.mutate('startView')
        tiempo de espera = nulo
    }, 1000)
    devolver() => {
        si (tiempo de espera) clearTimeout(tiempo de espera)
        de lo contrario trackEventMutation.mutate('endView')
    }
}, [])

```

6. Inicie el frontend de la siguiente manera (y manténgalo en ejecución para el desarrollo futuro):

```
$ npm ejecuta dev
```

Asegúrese de ejecutar este comando en la raíz de la carpeta ch10, no dentro de la carpeta backend.

Si ahora abres una publicación en tu navegador y revisas la pestaña Red del inspector, verás que, después de un segundo, se registra el evento startView. Al salir de la página, se registra el evento endView.

Pasemos ahora a simular eventos para que tengamos más datos para agregar y visualizar más tarde.

## Simulación de eventos

Simular eventos es una excelente manera de generar datos de muestra para probar las agregaciones y visualizaciones. En nuestra simulación, primero borramos todos los usuarios actuales de la base de datos y luego creamos un conjunto de usuarios de muestra. Repetimos el mismo proceso para las publicaciones y luego para los eventos, simulando que un usuario aleatorio crea una publicación y que alguien la ve durante un tiempo aleatorio.

Siga estos pasos para implementar una simulación:

1. Primero, debemos cambiar la base de datos para evitar perder los datos creados previamente en los otros capítulos.

Edite el archivo backend/.env y cambie la siguiente línea de blog a blog-simulado:

```
URL_DE_BASE_DE_DATOS=mongodb://localhost:27017/blog-simulado
```

2. Ahora, crea un nuevo archivo backend/simulateEvents.js, en el que importamos dotenv, la función initDatabase y todos los modelos y funciones de servicio relevantes:

```
importar dotenv desde 'dotenv'  
dotenv.config()  
  
importar { initDatabase } desde './src/db/init.js'  
importar { Post } desde './src/db/models/post.js'  
importar { Usuario } desde './src/db/models/user.js'  
importar { Event } desde './src/db/models/event.js'  
importar { createUser } desde './src/services/users.js'  
importar { createPost } desde './src/services/posts.js'  
importar { trackEvent } desde './src/services/events.js'
```

3. Defina una hora de inicio para la simulación, que aquí se establece en hace 30 días (30 días \* 24 horas \* 60 segundos \* minutos \* 1000 milisegundos), y una hora de finalización, que es ahora:

```
constante simulaciónInicio = Fecha.ahora() - 1000 * 60 * 60 * 24 * 30  
fin de simulación constante = Fecha.ahora()
```

4. También definimos el número de usuarios, publicaciones y visualizaciones a simular:

```
const usuarios simulados = 5  
const simuladoPosts = 10  
const vistas simuladas = 10000
```

5. \$en, definimos la función simulatorEvents, en la que primero inicializamos la base de datos:

```
función asíncrona simularEventos() {  
    conexión constante = await initDatabase()
```

6. A continuación, elimine todos los usuarios existentes y cree nuevos usuarios inicializando una matriz vacía con el Número de usuarios a simular y mapeo sobre ellos:

```
esperar Usuario.deleteMany({})  
const createdUsers = await Promise.all(  
    Matriz(simulatedUsers)  
        .fill(nulo)  
        .mapa(  
            asíncrono (_, u) =>  
                esperar crearUsuario({  
                    nombre de usuario: `usuario-${u}`,  
                    contraseña: `contraseña-${u}`,  
                }),  
        ),
```

```
) console.log(`creó ${createdUsers.length} usuarios`)
```

## Información

La función `$e Array(X)` se puede utilizar para crear una matriz con X entradas, que luego debe completarse con un valor inicial antes de poder iterar sobre ella.

7. Ahora, repite el mismo proceso para las publicaciones:

```
esperar Post.deleteMany({}) const createdPosts
= esperar Promise.all(
    Array(simulatedPosts).fill(null).map(async
        (_, p) => { const
            randomUser =
                Usuarios creados[Math.floor(Math.random() *
simulados)] return await
            createPost(randomUser._id, { título: `Publicación de prueba ${p}`,
                contenido: `Esta
es una publicación de prueba ${p}` }),

)
console.log(`publicaciones creadas ${createdPosts.length}`)
```

## Información

Usamos `Math.floor(Math.random() * maxNumber)` para crear un entero aleatorio entre 0 y maxNumber (no inclusivo), que es perfecto para usar para indexar una matriz.

8. Por último, repetimos lo mismo para los eventos:

```
esperar Evento.deleteMany({}) const createdViews
= esperar Promesa.all(
    Matriz(simulatedViews)
        .fill(nulo)
        .map(async() => {
            constante randomPost =
                createdPosts[Math.floor(Math.random() *
publicaciones simuladas)]
```

9. Aquí, comenzamos la sesión en un momento aleatorio dentro de las fechas de simulación definidas:

```
constante sessionStart =  
    simulaciónInicio + Math.random() * (simulaciónFin -  
    simulaciónInicio)
```

10. Y lo terminamos aleatoriamente después de 0 a 5 minutos:

```
constante sessionEnd =  
    sessionStart + 1000 * Math.floor(Math.random() * 60 *  
    5)
```

11. Ahora, simulamos la colección de eventos, primero creando un evento startView:

```
constante evento = await trackEvent({ postId:  
    randomPost._id, acción: 'startView',  
    fecha: nueva Fecha(sessionStart), })
```

12. Y luego simulamos un evento endView, donde usamos el ID de sesión devuelto desde el primer evento:

```
esperar trackEvent({ postId:  
    randomPost._id, sesión: evento.session,  
  
    acción: 'endView', fecha: nueva  
    Fecha(sessionEnd), }) }),
```

```
) console.log(` ${createdViews.length} vistas simuladas con éxito`)
```

13. Por último, nos desconectamos de la base de datos y llamamos a la función:

```
esperar conexión.desconectar()  
}  
  
simularEventos()
```

14. ¡Nuestra simulación ya está lista! Ejecute el siguiente comando para iniciarla:

```
$ cd backend/ $ node  
simulatorEvents.js
```

Verás que la simulación primero crea 5 usuarios, luego 10 publicaciones y finalmente simula 10 000 visitas.

En la siguiente sección, usaremos este conjunto de datos para probar algunas agregaciones con MongoDB.

## Agregación de datos con MongoDB

A veces, no solo queremos recuperar datos de la base de datos, sino crear estadísticas a partir de ellos combinándolos y resumiéndolos. Este proceso se denomina agregación de datos y puede ayudarnos a comprender mejor los datos. Por ejemplo, podemos contar el número total de visualizaciones por publicación, obtener el número de visualizaciones diarias por publicación o calcular la duración promedio de la sesión al ver una publicación.

MongoDB admite una sintaxis de agregación especial utilizando la función `.aggregate()` en una colección.

El uso de esta funcionalidad de agregación de MongoDB nos permite consultar y procesar documentos de forma eficiente.

Las operaciones que proporciona son similares a las que se pueden realizar con las consultas de Lenguaje de Consulta Estructurado (SQL). Principalmente, utilizaremos las siguientes operaciones de agregación:

- `$match`: se utiliza para filtrar documentos
- `$group`: se utiliza para agrupar documentos por una determinada propiedad
- `$project`: Se utiliza para asignar propiedades a diferentes propiedades o procesarlas
- `$sort`: se utiliza para ordenar documentos

Información

MongoDB proporciona muchas más operaciones de agregación avanzadas, todas las cuales se pueden encontrar en su documentación (<https://www.mongodb.com/docs/manual/aggregation/>). También están agregando constantemente más operaciones para hacer que la agregación sea aún más poderosa.

La función de agregación `$e` funciona proporcionando un array de objetos, cada uno de los cuales define una etapa del proceso de agregación. En este capítulo, aprenderemos más sobre las agregaciones al usarlas en la práctica.

### Obtener el número total de vistas por publicación

La primera agregación que definiremos es una forma de obtener el número total de visualizaciones por publicación. Para ello, necesitaremos `$match` para filtrar todas las acciones de `startView` (de lo contrario, contaría las visualizaciones dos veces, ya que también hay una acción `endView` para cada visualización de la entrada del blog) y `$group` para agrupar los resultados por ID de publicación y luego devolver el número de documentos mediante `$count`.

Siga estos pasos para crear su primera canalización de agregación:

1. Cree una nueva carpeta backend/playground/ para nuestros scripts de playground.
2. Haga clic en la extensión MongoDB (el ícono de la hoja) en la barra lateral de VS Code.
3. Conéctese a la base de datos, luego expanda la sección Patios de recreo (si aún no está expandida), y haga clic en el botón Crear nuevo patio de juegos .

Se abrirá un nuevo archivo con código predefinido. Elimínelo, ya que lo reemplazaremos con el nuestro.

4. Primero, defina el uso y las globales de la base de datos que nos proporciona MongoDB Playground:

```
/* uso global, base de datos */
```

5. \$en, utiliza la base de datos simulada del blog:

```
use('blog-simulado')
```

6. Ahora, ejecute la siguiente función de agregación:

```
db.getCollection('eventos').aggregate([
```

7. La primera etapa del pipeline coincidirá con todas las acciones de startView:

```
{
  $match: { acción: 'startView' }, },
```

8. \$en, agrupamos por publicación. La etapa \$e \$group requiere que definamos un \_id, que contiene la propiedad por la que se agrupará. Necesitamos usar el operador \$ para resolver la variable a usar, por lo que \$post accederá a la propiedad event.post (que contiene un ID de publicación):

```
{
  $grupo: { _id:
    '$publicación', vistas:
    { $cuenta: {} }, }, },
])
```

9. Guarde el script como un archivo backend/playground/views-per-post.mongodb.js.

10. Haga clic en el icono de reproducción en la esquina superior derecha para ejecutar el script. Se abrirá una nueva pestaña con los resultados de la agregación:

```

1 [ 
2 {
3   "_id": {
4     "$oid": "64d294acb1fac3e6f41f5f1d"
5   },
6   "views": 994
7 },
8 {
9   "_id": {
10    "$oid": "64d294acb1fac3e6f41f5f15"
11  },
12   "views": 1005
13 },
14 {
15   "_id": {
16    "$oid": "64d294acb1fac3e6f41f5f1a"
17  },
18   "views": 971
19 },
20 {
21   "_id": {
22    "$oid": "64d294acb1fac3e6f41f5f1c"
23  },
24   "views": 1027
25 }
]

```

Figura 10.1 – ¡Nuestro primer resultado de agregación de MongoDB!

Después de crear y ejecutar nuestra primera agregación simple, continuemos practicando escribiendo agregaciones más avanzadas.

### Obtener el número de visualizaciones diarias por publicación

Ahora que ya conocemos el proceso general de creación de agregaciones de MongoDB, intentemos crear una agregación un poco más compleja: obtener el número de visualizaciones diarias por publicación. Siga estos pasos para crearla:

1. Cree un nuevo patio de juegos, como antes, con la siguiente función de agregación:

```

/* uso global, base de datos */
use('blog-simulado')
db.getCollection('eventos').aggregate([

```

2. Nuevamente, primero hacemos coincidir solo las acciones startView:

```
{  
    $match: { acción: 'startView' },  
},
```

3. Luego usamos \$project para mantener la propiedad de publicación y definir una nueva propiedad de día, que usa la función \$dateTrunc para simplificar la propiedad de fecha para que solo cubra los días (en lugar de contener la marca de tiempo completa):

```
{  
    $proyecto: {  
        publicación: '$post',  
        día: { $dateTrunc: { fecha: '$date', unidad: 'día' } },  
    },  
},
```

¡Algo importante a tener en cuenta con \$project es que solo las propiedades que se enumeran aquí se pasarán a las etapas posteriores del proceso, por lo que debemos enumerar aquí todas las propiedades que aún vamos a necesitar más adelante!

4. Finalmente, usamos \$group para agrupar los documentos por correo postal y día, pasando un objeto a la propiedad \_id. Usamos \$count de nuevo para contar el número de documentos en cada grupo:

```
{  
    $grupo: {  
        _id: { publicación: '$publicación', día: '$día' },  
        vistas: { $count: {} },  
    },  
},  
])
```

5. Guarde el script como backend/playground/views-per-post-per-day.mongodb.js le.

6. Ejecute este script haciendo clic en el botón Reproducir . Verá que ahora obtenemos una lista de documentos agrupados por publicación y día, y el número de visualizaciones correspondiente a una publicación determinada en un día determinado.

```

1  [
2   {
3     "_id": {
4       "post": {
5         "$oid": "65c8ef3599c4fdd0c066ed74"
6       },
7       "day": {
8         "$date": "2024-01-30T00:00:00Z"
9       }
10    },
11    "views": 36
12  },
13  {
14    "_id": {
15      "post": {
16        "$oid": "65c8ef3599c4fdd0c066ed72"
17      },
18      "day": {
19        "$date": "2024-01-12T00:00:00Z"
20      }
21    },
22    "views": 7
23  },

```

Figura 10.2 – Muestra el número de visualizaciones por publicación por día

Después de obtener el número de visitas diarias por publicación, continuemos practicando calculando la duración promedio de la sesión.

## Cálculo de la duración media de la sesión

Como recordará, primero enviamos una acción startView y luego una endView, ambas con fecha independiente. Usaremos agregaciones para agrupar estas dos acciones en un solo documento y luego calcular la duración de una sesión:

1. Cree un nuevo archivo de patio de juegos y comience a escribir una agregación que primero cree algunas propiedades nuevas usando \$project y mantenga la propiedad de sesión, ya que la necesitaremos más adelante:

```

/* uso global, base de datos */
use('blog-simulado')
db.getCollection('eventos').aggregate([
  {
    $project: {
      sesión: '$sesión',

```

```

Fecha de inicio: {
    $cond: [{ $eq: ['$acción', 'startView'] }, '$fecha', indefinido],
},
fechaFin: { $cond: [{ $eq: ['$acción', 'vistaFin'] }, '$fecha', indefinido] },
},
}
,
```

Aquí, usamos el operador `$cond` para crear una condición (similar a una sentencia ternaria/`if`). Acepta un array con tres elementos: el primero es una condición, el siguiente un resultado si la condición cumple y, por último, un resultado si la condición no cumple. En nuestro caso, comprobamos si la propiedad de acción es `startView` (usando el operador `$eq`). Si es verdadera, asignamos la fecha a la propiedad `startDate`. De lo contrario, no definimos `startDate`. Propiedad. De manera similar, si la acción es `endView`, creamos una propiedad `endDate`.

2. Ahora, podemos agrupar los documentos por ID de sesión y seleccionar la fecha de inicio más baja y la fecha de finalización más alta de una sesión:

```
{
    $grupo: {
        _id: '$sesión',
        fecha de inicio: { $min: '$fechadeinicio' },
        fecha de finalización: { $max: '$fecha de finalización' },
    },
}
```

De todos modos, solo debería haber una acción `startView` y `endView` por sesión, pero no podemos garantizarlo, por lo que debemos agregarlas en un solo valor.

3. Finalmente, usamos `$project` nuevamente para cambiar el nombre de la propiedad `_id` a `sesión` y calcular el duración restando la fecha de inicio de la fecha de finalización:

```
{
    $proyecto: {
        sesión: '$_id',
        duración: { $subtract: ['$endDate', '$startDate'] },
    },
},
])
```

4. Guarde el script como un archivo `backend/playground/session-duration.mongodb.js`.

5. Ejecute el script y verá una lista de documentos con un ID de sesión y un correspondiente duración en milisegundos:

```

1 [
2 {
3   "_id": "546c319f-45f3-4a8b-8115-ec3298aef3eb",
4   "session": "546c319f-45f3-4a8b-8115-ec3298aef3eb",
5   "duration": 35000
6 },
7 {
8   "_id": "e0adff0d-2dad-4137-8b40-0480e0a71a5e",
9   "session": "e0adff0d-2dad-4137-8b40-0480e0a71a5e",
10  "duration": 78000
11 },
12 {
13   "_id": "cde9dff5-7403-4725-a3b9-945d5e433b33",
14   "session": "cde9dff5-7403-4725-a3b9-945d5e433b33",
15   "duration": 85000
16 }
]

```

Figura 10.3 – Resultado de la agregación de las duraciones de las sesiones

Ahora que estamos más familiarizados con cómo funciona la agregación de datos en MongoDB, ¡implementemos agregaciones similares en nuestro backend!

## Implementación de la agregación de datos en el backend

Para nuestro backend, usaremos canales de agregación muy similares. Sin embargo, debemos ajustarlos ligeramente, ya que siempre queremos obtener los datos de una sola publicación. Por lo tanto, primero usaremos \$match para filtrar nuestros documentos. \$match también garantiza que la agregación se mantenga rápida, incluso con millones de eventos en nuestra base de datos, ya que primero filtramos todos los eventos de una sola publicación.

### Definición de funciones de servicio de agregación

Siga estos pasos para implementar las funciones de agregación en el backend:

1. Edite backend/src/services/events.js y defina una nueva función para obtener el número total de visualizaciones de una publicación. En este caso, podemos simplificar nuestro código usando countDocuments. función en lugar de la función agregada:

```

exportar función asíncrona getTotalViews(postId) {
  devolver {

```

```
vistas: esperar Event.countDocuments({ post: postId, acción: 'startView' }),  
}  
}
```

2. A continuación, defina una nueva función para obtener las visualizaciones diarias de una publicación con un ID determinado. Ahora usamos la operación '\$match' para obtener únicamente las acciones 'startView' de una publicación específica:

```
exportar función asíncrona getDailyViews(postId) {  
    devolver esperar Evento.agregado([  
        {  
            $match:  
                { publicación: postId,  
                    acción: 'startView', }, },
```

3. \$en, utilizamos la operación \$group en combinación con \$dateTrunc para obtener las vistas por día, tal como lo hicimos antes en el script MongoDB Playground:

```
{  
    $grupo: { _id:  
  
        { $dateTrunc: { fecha: '$date', unidad: 'día' }, }, vistas: { $count: {} }, }, },
```

4. Por último, utilizamos la operación \$sort para ordenar los documentos resultantes por \_id (que contiene el día):

```
{  
    $sort: { _id: 1 }, },  
  
])  
}
```

5. Para la última función, usamos nuestra agregación de duración de sesión, pero la ampliamos ligeramente para obtener la duración promedio por día. Nuevamente, primero necesitamos que coincida con un ID de publicación:

```
exportar función asíncrona getDailyDurations(postId) { devolver esperar Event.gregate([  
  
    {  
        $match:  
            { publicación: postId,
```

```
    }, },
```

6. \$en, usamos las mismas operaciones \$project y \$group para obtener la sesión, fecha de inicio y fecha de finalización, tal como lo hicimos antes:

```
{
  $proyecto:
    { sesión: '$sesión', fecha de inicio:
      { $constancia:
        [ { $eq: ['$acción', 'vistalnicio'] }, '$fecha', indefinido],
        },
      endDate:
        { $cond: [{ $eq: ['$acción', 'endView'] }, '$fecha', indefinido], }, }, },
    $grupo: { _id:
      '$sesión', fecha de inicio:
        { $min: '$fecha de inicio' }, fecha de finalización: { $max:
          '$fecha de finalización' }, }, },
```

7. Ahora, usamos la operación \$project para obtener el día a partir de nuestra fecha de inicio, como lo hicimos en el Agregación anterior donde obtuvimos el número de vistas diarias de una publicación:

```
{
  $proyecto: { día:
    { $dateTrunc: { fecha: '$startDate', unidad: 'día' } }
  },
  duración: { $subtract: ["$endDate", "$startDate"] }, }, },
```

8. Agrupamos los resultados por día y calculamos la duración media de un día:

```
{
  $grupo: { _id:
    '$día',
    duraciónPromedio: { $promedio: '$duración' }, }, },
```

9. Finalmente, ordenamos los resultados por día:

```
{  
    $sort: { _id: 1 }, },  
  
}  
})
```

Como podemos ver, las canalizaciones de agregación son extremadamente potentes y nos permiten realizar un gran procesamiento de datos directamente en la base de datos. En la siguiente sección, crearemos rutas para estas funciones de agregación.

## Definiendo las rutas

Definir las rutas es bastante sencillo: simplemente comprobamos si existe una publicación con el ID dado y, de ser así, devolvemos los resultados de la función de agregación correspondiente.

Comencemos definiendo las rutas:

1. Edite backend/src/routes/events.js e importe getTotalViews, Funciones getDailyViews y getDailyDurations:

```
importar  
    { trackEvent,  
        getTotalViews,  
        getDailyViews,  
        getDailyDurations, } desde '../  
services/events.js'
```

2. A continuación, dentro de la función eventRoutes, defina una nueva ruta para obtener el número total de vistas de una publicación, de la siguiente manera:

```
aplicación.get('/api/v1/events/totalViews/:postId', async (req, res) => { try { const { postId } = req.params  
const  
post =  
    await getPostById(postId) if (post === null) return  
    res.status(400).end() const stats = await getTotalViews(post._id)  
    return res.json(stats) } catch (err) {  
  
        console.error('error al obtener estadísticas', err) return  
        res.status(500).end()  
    } })
```

3. \$en define una ruta similar para el número de vistas diarias de una publicación:

```
aplicación.get('/api/v1/events/dailyViews/:postId', async (req, res) => { try { const { postId } = req.params
const
post =
await getPostById(postId) if (post === null) return
res.status(400).end() const stats = await getDailyViews(post._id)
return res.json(stats) } catch (err) {

}

console.error('error al obtener estadísticas', err) return
res.status(500).end()

} })
```

4. Y finalmente, define una ruta para la duración media diaria de visualización de una publicación:

```
aplicación.get('/api/v1/events/dailyDurations/:postId', async (req, res) => { try { const { postId } =
req.params
const
post = await getPostById(postId) if (post === null)
return res.status(400).end() const stats = await
getDailyDurations(post._id) return res.json(stats) } catch (err) {

}

console.error('error al obtener estadísticas', err)
devolver res.status(500).end()

} })
```

Ahora que hemos definido con éxito las rutas para nuestras funciones de agregación, ¡es hora de integrarlas en el frontend y comenzar a visualizar los datos que hemos estado simulando y recopilando!

## Integración y visualización de datos en el frontend usando Victoria

En esta sección final, vamos a integrar los puntos finales de agregación que definimos previamente. Luego, vamos a presentar la biblioteca Victory en el frontend para crear gráficos para visualizar nuestros datos agregados.

## Integración de la API de agregación

En primer lugar, necesitamos integrar las rutas API en el frontend, de la siguiente manera:

1. Edite el archivo `src/api/events.js` y agregue tres nuevas funciones de API para obtener las vistas totales, las vistas diarias y las duraciones diarias de una publicación:

```
exportar const getTotalViews = (postId) => buscar(`$  
{import.meta.env.VITE_BACKEND_URL}/events/ totalViews/${postId}`).luego( (res)  
=> res.json(),  
  
)  
  
exportar const getDailyViews = (postId) => buscar(`$  
{import.meta.env.VITE_BACKEND_URL}/events/dailyViews/${postId}`).luego( (res)  
=> res.json(),  
  
)  
  
exportar const getDailyDurations = (postId) =>  
buscar(  
` ${import.meta.env.VITE_BACKEND_URL}/eventos/duracionesdiarias/$  
{postId}`,  
). entonces((res) => res.json())
```

2. Cree un nuevo archivo `src/components/PostStats.jsx`, donde consultaremos estas nuevas rutas de API. Comience importando `useQuery`, `PropTypes` y las tres funciones de API:

```
importar { useQuery } desde '@tanstack/react-query' importar PropTypes desde  
'prop-types' importar { getTotalViews, getDailyViews,  
  
getDailyDurations, } desde  
'./api/events.js'
```

3. Defina un nuevo componente que tome un postId y obtenga todas las estadísticas que agregamos en el backend usando ganchos de consulta:

```
función de exportación PostStats({ postId }) { const totalViews =  
useQuery({  
clave de consulta: ['totalViews', postId], función de  
consulta: () => obtenerTotalViews(postId), }) const dailyViews  
=  
useQuery({  
clave de consulta: ['dailyViews', postId],
```

```

    consultaFn: () => getDailyViews(postId), }) const dailyDurations
  =
useQuery({
  clave de consulta: ['duracionesdiarias', IDdepublicación],
  funcióndeconsulta: () => obtenerDuracionesDiarias(IDdepublicación), })

```

4. Mientras se cargan las estadísticas, mostramos un mensaje de carga simple:

```

if
  ( totalViews.isLoading ||
    dailyViews.isLoading ||
    dailyDurations.isLoading { return
  )
  <div>cargando estadísticas...</div>
}

```

5. Una vez cargadas las estadísticas, podemos mostrarlas. Por ahora, simplemente mostramos el número total de visualizaciones y las respuestas JSON de las otras dos solicitudes de API:

```

devolver (
  <div>
    <b>{totalViews.data?.views} visualizaciones totales</b>
    <pre>{JSON.stringify(dailyViews.data)}</pre>
    <pre>{JSON.stringify(dailyDurations.data)}</pre>
  </div>
)
}

```

6. Todavía necesitamos definir los tipos de propiedad para este componente, de la siguiente manera:

```

PostStats.propTypes = {
  postId: PropTypes.string.isRequired,
}

```

7. Ahora podemos renderizar el componente PostStats en nuestro componente de página ViewPost. Edite src/pages/ViewPost.jsx e importe el componente PostStats allí:

```

importar { PostStats } desde '../components/PostStats.jsx'

```

8. \$en, en la parte inferior del componente, renderiza las estadísticas de la siguiente manera:

```

{publicar ? ( <div>
  <Publicar {...publicar} publicación completa />
  <hr />

```

```
<Estadísticas de publicación postId={postId} />
</div>
) : (
`No se encontró la publicación con id ${postId}.`
)}
</div>
)
}
```

Si abres una publicación en la interfaz ahora (quizás tengas que actualizarla si ves un error), verás que todas las estadísticas se han obtenido correctamente. ¡Ahora solo queda visualizar las estadísticas diarias con Victory!

## Visualización de datos con Victory

Victory es una biblioteca de React que proporciona componentes modulares que permiten crear gráficos y todo tipo de visualizaciones de datos. Incluso admite herramientas de visualización interactiva, como el cepillado y la agrupación (donde, por ejemplo, se selecciona una sección de un gráfico para examinarla con más detalle en otros). En este capítulo, solo exploraremos superficialmente las posibilidades de Victory, ya que la visualización de datos en React es un tema complejo en sí mismo.

Puede encontrar más información sobre Victory en su sitio web social: <https://commerce.nearform.com/open-source/victory/>

Creación de un gráfico de barras

Comencemos a visualizar nuestros datos usando Victory ahora:

1. Instale la biblioteca ejecutando el siguiente comando en la raíz del proyecto:

```
$ npm install victoria@36.9.1
```

2. Edite src/components/PostStats.jsx e importe los siguientes componentes desde Victory:

```
importar {
  Gráfico de la victoria,
  Información sobre herramientas de la victoria,
  Barra de la victoria,
  Línea de la victoria,
  VictoriaVoronoiContenedor,
} de 'victoria'
```

3. Reemplace las etiquetas <pre> al final del componente con los siguientes gráficos, comenzando con el gráfico de vistas diarias:

```
devolver (
  <div>
    <b>{totalViews.data?.views} visualizaciones totales</b>
    <div style={{ ancho: 512 }}>
      Vistas diarias
      <VictoryChart domainPadding={16}>
```

El componente \$e de VictoryChart es un contenedor que se utiliza para combinar todos los elementos de un gráfico Victory. Configuramos domainPadding a 16 píxeles, que es un relleno dentro del gráfico. Esto garantiza que las líneas y los gráficos de barras no se peguen a los bordes del gráfico, mejorando su aspecto.

4. \$en, define un gráfico de barras con VictoryBar, usando VictoryTooltip para mostrar las etiquetas:

```
<VictoryBar
  componente de etiqueta={<VictoryTooltip />}
```

La información sobre herramientas se ve así:

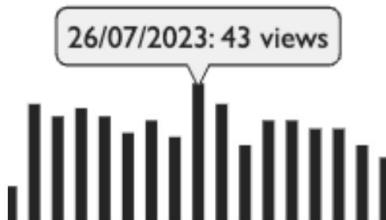


Figura 10.4 – Información sobre herramientas en un gráfico de barras en Victory

5. Ahora llegamos a la parte más importante: los datos. Aquí, mapeamos los datos de DailyViews devueltos por el gancho de consulta para convertirlos a un formato que Victory entienda:

```
datos={dailyViews.data?.map((d) => ({
```

6. Asignamos la propiedad \_id al valor del eje x (analizándolo como una fecha) y la propiedad views al valor del eje y:

```
  x: nueva Fecha(d._id),
  y: d.vistas,
```

7. Luego creamos una etiqueta, donde convertimos el día en una cadena de fecha local y luego mostramos el número de vistas en el día dado:

```
etiqueta: ` ${new Date(d._id).toLocaleDateString()}: ${d.views} vistas`,
})})
```

```
    />
    Gráfico de la victoria
  </div>
</div>
)
}
```

Hemos creado con éxito nuestra primera visualización con Victory. El gráfico tendrá ahora el siguiente aspecto:

**Daily Views**

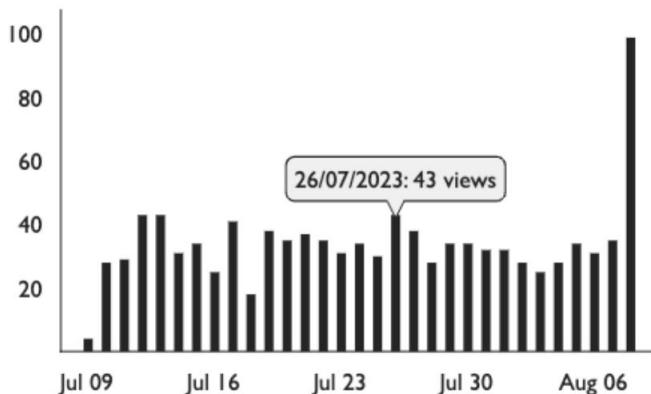


Figura 10.5 – Nuestro primer gráfico en Victory: ¡un gráfico de barras!

Como puedes ver, Victory formateó automáticamente las fechas para nosotros y ajustó los ejes para que nuestro gráfico se ajuste al espacio asignado.

A continuación, visualicemos la duración promedio de visualización diaria.

#### Creación de un gráfico de líneas

Crear un gráfico de líneas en Victory es bastante similar a crear un gráfico de barras, con una excepción: la información sobre herramientas. En los gráficos de líneas, no podemos usar descripciones emergentes directamente, ya que, en teoría, las líneas podrían ser continuas (no bloques de datos discretos), por lo que no está claro dónde colocarlas. En su lugar, en Victory, usamos un contenedor de Voronoi para mostrar descripciones emergentes en gráficos de líneas. El nombre Voronoi proviene de las matemáticas, donde un diagrama de Voronoi divide una región en varias secciones. En pocas palabras, el contenedor de Voronoi crea una intersección entre la posición del ratón y el gráfico de líneas, obtiene los datos de ese punto de intersección y luego muestra una descripción emergente allí.

Con esto en mente, comencemos ahora a crear el gráfico de líneas para la duración promedio diaria de visualización:

1. Edite src/components/PostStats.jsx y continúe donde lo dejamos con el otro gráfico, agregando un nuevo VictoryChart después del contenedor del gráfico de barras:

```
</VictoryChart> </div>
<div
  style={{ width: 512 }}> <h4>Duración
    promedio diaria de visualización</h4>
    <VictoryChart
      domainPadding={16}
```

2. En el componente VictoryChart, ahora definimos containerComponent, que será Contiene nuestro VictoryVoronoiContainer:

```
componente contenedor={
  <VictoriaVoronoiContenedor
    voronoiDimension='x'
```

Lo definimos para que solo se intersecte con los valores en el eje x, lo que significa que el puntero del mouse solo se intersecará con los días en nuestro gráfico.

3. Ahora podemos definir etiquetas para nuestro contenedor. Para ello, usamos la propiedad datum para obtener la entrada de datos que interseca con el puntero del ratón y crear una etiqueta. Nuestra etiqueta debe mostrar la fecha actual y la duración de la visualización en minutos, fijada con dos decimales.

```
etiquetas={({ dato }) => `$
  ${dato.x.toLocaleDateString()}>
  ${datum.y.toFixed(2)} minutos` }
```

4. Nuevamente, usamos VictoryTooltip para mostrar estas etiquetas:

```
componente de etiqueta=<VictoryTooltip /> />
```

```
}
```

5. Ahora podemos finalmente definir el gráfico VictoryLine, en el que mapeamos nuevamente los datos, analizando las fechas y dividiendo la duración promedio para convertirla de milisegundos a minutos:

```
<VictoryLine
  datos={dailyDurations.data?.map((d) => ({
    x: nueva Fecha(d._id), y:
    d.duraciónpromedio / (60 * 1000), }))} />
```

```
Gráfico de la victoria
</div>
</div>
)
}
```

Como pueden ver, el resto fue bastante sencillo y similar a la creación del gráfico de barras. Se ve así:

**Daily Average Viewing Duration**



Figura 10.6 – Un gráfico de líneas que utiliza Victory y que muestra la duración promedio diaria de visualización de una publicación.

Como puedes ver, Victory es una biblioteca bastante potente para crear gráficos con React, ¡y solo hemos empezado a explorar sus posibilidades! Puedes personalizar el tema de los gráficos y crear todo tipo de visualizaciones complejas. Sin embargo, en este capítulo nos centramos en los gráficos más esenciales y utilizados: los gráficos de barras y de líneas.

## Resumen

En este capítulo, aprendimos a rastrear eventos usando nuestro backend y frontend. Simulamos eventos para usarlos como un conjunto de datos de muestra para nuestras agregaciones y visualizaciones. A continuación, aprendimos a agregar datos con MongoDB usando MongoDB Playground. Implementamos funciones de agregación de datos en nuestro backend. Finalmente, integramos y visualizamos los datos en el frontend usando Victory.

En el próximo capítulo, Capítulo 11, Construyendo un backend con una API GraphQL, aprenderemos cómo usar una alternativa a REST, llamada GraphQL, para consultar objetos profundamente anidados más fácilmente.

Machine Translated by Google

## Construyendo un backend con una API GraphQL

Hasta ahora, solo hemos interactuado con API REST. Para API más complejas con objetos profundamente anidados, podemos usar GraphQL para permitir el acceso selectivo a ciertas partes de objetos grandes. En este capítulo, primero aprenderemos qué es GraphQL y cuándo es útil. Luego, experimentaremos con la creación de consultas y mutaciones GraphQL. Después, implementaremos GraphQL en un backend. Finalmente, cubriremos brevemente conceptos avanzados de GraphQL.

En este capítulo cubriremos los siguientes temas principales:

- ¿Qué es GraphQL?
- Implementación de una API GraphQL en un backend
- Implementación de autenticación y mutaciones GraphQL
- Descripción general de los conceptos avanzados de GraphQL

### 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/cap11>.

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/6gP0uM-XaVo>.

## ¿Qué es GraphQL?

Antes de empezar a aprender a usar GraphQL, centrémonos primero en qué es GraphQL. Al igual que REST, es una forma de consultar APIs. Sin embargo, es mucho más que eso. GraphQL incluye un entorno de ejecución del lado del servidor para ejecutar consultas y un sistema de tipos para definir los datos. Funciona con numerosos motores de bases de datos y se puede integrar en el backend existente.

Los servicios GraphQL se crean definiendo tipos (como el tipo Usuario), campos en tipos (como el campo de nombre de usuario) y funciones para resolver valores de campos. Supongamos que hemos definido el siguiente tipo Usuario con una función para obtener un nombre de usuario:

```
tipo Usuario {  
    nombreDeUsuario: String  
}  
  
función Usuario_nombreDeUsuario(usuario) {  
    devolver usuario.getUsername()  
}
```

Podríamos entonces definir un tipo de consulta y una función para obtener el usuario actual:

```
tipo Consulta {  
    usuarioActual: Usuario  
}  
  
función Consulta_UsuarioActual(req) {  
    devolver req.auth.user  
}
```

Información

El tipo de consulta \$e es un tipo especial que define el punto de entrada al esquema GraphQL. Nos permite definir qué campos se pueden consultar mediante la API GraphQL.

Ahora que hemos definido tipos con campos y funciones para resolverlos, podemos crear una consulta GraphQL para obtener el nombre de usuario del usuario actual. Las consultas GraphQL se parecen a objetos JavaScript, pero solo listan los nombres de los campos que se desean consultar. La API GraphQL devolverá un objeto JavaScript con la misma estructura que la consulta, pero con los valores completos. Veamos cómo se vería una consulta para obtener el nombre de usuario del usuario actual:

```
{  
    usuarioActual {
```

```

        nombre de usuario
    }
}
}
```

La consulta \$at devolvería entonces un resultado JSON que se vería así:

```
{
  "datos": {
    "UsuarioActual": {
      "nombre de usuario": "dan"
    }
  }
}
```

Como puede ver, el resultado tiene la misma forma que la consulta. \$is es uno de los conceptos esenciales de GraphQL: el cliente puede solicitar específicamente los campos que necesita y el servidor los devolverá exactamente. Si necesitamos más datos sobre un usuario, podemos simplemente agregar nuevos campos al tipo y a la consulta.

GraphQL valida consultas y resultados con los tipos definidos. Esto garantiza que no se rompa el contrato entre el cliente y el servidor. Los tipos GraphQL actúan como ese contrato entre el cliente y el servidor. Tras validar la consulta, un servidor GraphQL la ejecuta y devuelve un resultado exactamente igual a la forma solicitada. Cada campo solicitado ejecuta una función en el servidor. Estas funciones se denominan resolvers.

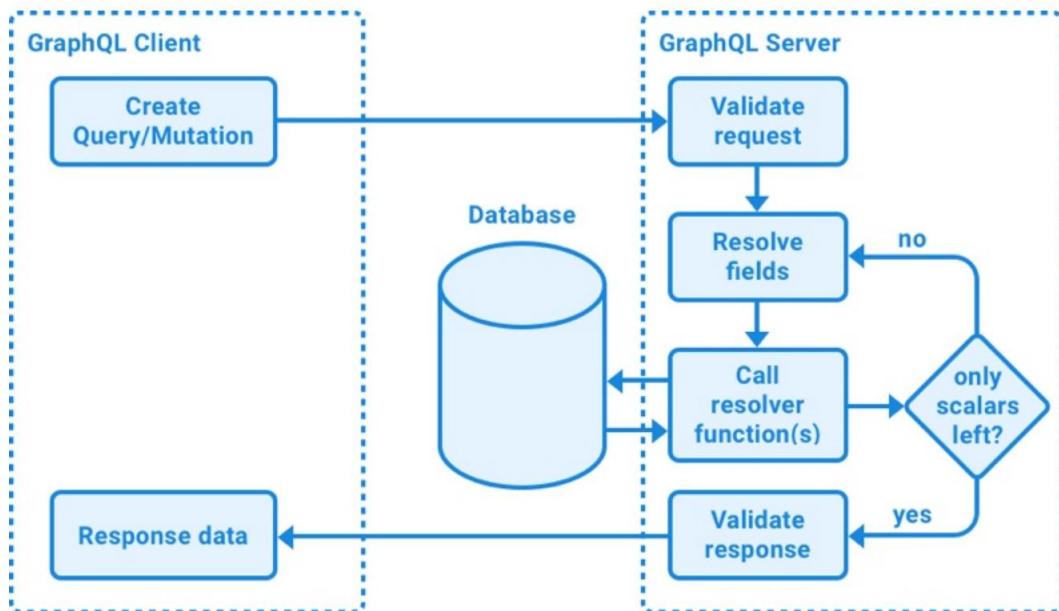


Figura 11.1 – Interacción entre el cliente y el servidor GraphQL

Los tipos y las consultas también pueden anidarse en profundidad. Por ejemplo, un usuario podría tener un campo que devuelva todas las publicaciones de las que es autor. También podemos realizar una subselección de campos en esos objetos de publicación . \$is funciona para objetos dentro de objetos e incluso para arrays de objetos dentro de objetos, en múltiples niveles de anidación. GraphQL seguirá resolviendo campos hasta que solo queden valores simples (escalares), como cadenas y números. Por ejemplo, la siguiente consulta podría obtener los ID y títulos de todas las publicaciones creadas por el usuario actual:

```
{  
  usuarioactual {  
    nombre de usuario  
    publicaciones {  
      identificación  
      título  
    }  
  }  
}
```

Además, GraphQL nos permite definir argumentos para campos, que se pasarán a las funciones que los resuelven. Podemos usar argumentos para, por ejemplo, obtener todas las publicaciones con una etiqueta específica. En GraphQL, podemos pasar argumentos a cualquier campo, incluso si está profundamente anidado. Incluso se pueden pasar argumentos a campos de un solo valor, por ejemplo, para transformar un valor. Por ejemplo, la siguiente consulta obtendría una publicación por ID y devolvería el título de la misma:

```
{  
  postById(id: "1234") {  
    título  
  }  
}
```

GraphQL es especialmente útil si creas el backend tú mismo o con él en mente, ya que permite patrones que facilitan la consulta de datos profundamente anidados e interconectados. Sin embargo, si existen backends REST que no controlas, no suele ser conveniente añadir GraphQL como una capa independiente, debido a sus restricciones basadas en esquemas.

Después de haber aprendido sobre las consultas, pasemos a las mutaciones.

## Mutaciones

En REST, cualquier solicitud podría causar un efecto secundario (como escribir datos en la base de datos). Sin embargo, como hemos aprendido, las solicitudes GET solo deben devolver datos y no causar tales efectos secundarios. Solo POST/PUT/PATCH/DELETE deberían provocar cambios en los datos de la base de datos. En GraphQL, existe un concepto similar: teóricamente, cualquier función eld podría provocar un cambio en el estado de la base de datos. Sin embargo, en GraphQL, definimos una mutación en lugar de una consulta para indicar explícitamente que queremos cambiar el estado de la base de datos. Además de definirse con la palabra clave mutation, las mutaciones tienen la misma estructura.

como consultas. Sin embargo, hay una diferencia: las consultas obtienen campos en paralelo, mientras que las mutaciones se ejecutan en serie, ejecutando primero la primera función de campo, luego la siguiente y así sucesivamente. Este comportamiento garantiza que no terminemos con condiciones de carrera en las mutaciones.

**Información**

Además del tipo de consulta incorporado, también hay un tipo de mutación para definir los campos de mutación permitidos.

Ahora que hemos aprendido los conceptos básicos de qué es GraphQL y cómo funciona, ¡comencemos a usarlo en la práctica implementando GraphQL en el backend de nuestra aplicación de blog!

## Implementación de una API GraphQL en un backend

Ahora configuraremos GraphQL en el backend de nuestra aplicación de blog, además de la API REST. Esto nos permitirá ver cómo GraphQL se compara y se diferencia de una API REST. Sigue estos pasos para empezar a configurar GraphQL en el backend:

1. Copie la carpeta ch10 existente a una nueva carpeta ch11, de la siguiente manera:

```
$ cp -R ch10 ch11
```

2. Abra la carpeta ch11 en VS Code.

3. Primero, instalemos una extensión de VS Code para añadir compatibilidad con el lenguaje GraphQL. Vaya a Extensiones. Busque la extensión GraphQL.vscode-graphql, desarrollada por GraphQL Foundation. Instálela.

4. A continuación, instale las bibliotecas graphql y @apollo/server en el backend usando el siguientes comandos:

```
$ cd backend/  
$ npm install graphql@16.8.1 @apollo/server@4.10.0
```

Apollo Server es una implementación de servidor GraphQL lista para producción que admite múltiples marcos web backend, incluido Express.

5. Cree una nueva carpeta `backend/src/graphql/`. Dentro de ella, cree `backend/src/`. graphql/query.js le, dentro del cual definimos un esquema de consulta, que es el punto de entrada de nuestra API GraphQL (que enumera todas las consultas compatibles con nuestro backend), de la siguiente manera:

```
exportar const querySchema = `#graphql  
    tipo Consulta {  
        prueba: Cadena  
    }`
```

Es importante agregar una directiva `#graphql` al inicio de la cadena de plantilla para que se reconozca como sintaxis GraphQL y se resalte correctamente en un editor de código. Dentro de nuestro esquema, definimos un campo de prueba, para el cual ahora definimos un resolver.

6. Defina un objeto `queryResolver` que contenga una función para resolver el campo de prueba en una cadena estática:

```
exportar const queryResolver = {
  Consulta: {
    prueba: () => {
      devuelve '¡Hola mundo desde GraphQL!'
    },
  },
}
```

7. Cree un nuevo archivo `backend/src/graphql/index.js` e importe el `querySchema` y `queryResolver` allí:

```
importar { querySchema, queryResolver } desde './query.js'
```

8. \$en, exporta una matriz llamada `typeDefs`, que incluye todos los esquemas (por ahora, solo el esquema de consulta) y una matriz llamada `resolvers`, que contiene todos los resolvers (por ahora, solo el resolver de consulta):

```
exportar const typeDefs = [querySchema]
exportar const resolvers = [queryResolver]
```

9. Edite `backend/src/app.js` e importe `ApolloServer` y `expressMiddleware` de la biblioteca `@apollo/server`:

```
importar { ApolloServer } desde '@apollo/server'
importar { expressMiddleware } desde '@apollo/server/express4'
```

10. \$en, importa `typeDefs` y `resolvers`:

```
importar { typeDefs, resolvers } desde './graphql/index.js'
```

11. Después de todo el resto del middleware y antes de las definiciones de ruta, cree un nuevo servidor Apollo usando Las definiciones de tipos de esquema y los solucionadores definidos:

```
constante apolloServer = nuevo ApolloServer({
  definiciones de tipos,
  resolutores,
})
```

12. \$en, una vez que el servidor esté listo, monte expressMiddleware en una ruta /graphql, de la siguiente manera:

```
apolloServer.start().then(() => app.use('/graphql',
  expressMiddleware(apolloServer)))
```

13. Inicie el backend en modo de desarrollo ejecutando el siguiente comando:

```
$ npm ejecuta dev
```

14. Vaya a <http://localhost:3001/graphql> en su navegador; debería ver la interfaz de Apollo para ingresar una consulta en el lado izquierdo y los resultados en el lado derecho.

15. Elimine todos los comentarios del editor en el archivo e ingrese la siguiente consulta GraphQL:

```
consulta EjemploConsulta {
  prueba
}
```

16. Presione el botón Reproducir para ejecutar la consulta y verá el siguiente resultado:

The screenshot shows the Apollo GraphQL playground interface. On the left, under 'Operation', there is a code editor with the following query:

```
query ExampleQuery {
  test
}
```

On the right, under 'Response', the results are displayed as a JSON object:

```
{
  "data": {
    "test": "Hello World from GraphQL!"
  }
}
```

At the top right, the status is shown as STATUS 200 | 11.0ms | 46B.

Figura 11.2 – Ejecución exitosa de nuestra primera consulta GraphQL!

Como puedes ver, nuestra consulta para el campo de prueba devuelve nuestra cadena estática definida previamente.

Después de implementar un campo básico, implementemos algunos campos que accedan a nuestras funciones de servicio y recuperen datos de MongoDB.

## Implementar campos que consultan publicaciones

Siga estos pasos para implementar los campos para consultar publicaciones:

1. Edite backend/src/graphql/query.js e importe las funciones de servicio relevantes:

```
importar
  { getPostById,
  listAllPosts,
```

```
listaPublicacionesPorAutor,  
listaPublicacionesPorEtiqueta,  
} de './servicios/posts.js'
```

2. Ajuste el esquema para incluir un campo de publicaciones, que devuelve una matriz de publicaciones:

```
exportar const querySchema = `#graphql  
    tipo Consulta {  
        prueba: Cadena  
        publicaciones: [!Publicación!]!
```

En GraphQL, la sintaxis [Type] significa que algo es un array de tipo. Definiremos el tipo Post más adelante. Type! es el modificador no nulo y significa que un tipo no es nulo (obligatorio), por lo que [Type!] significa que cada elemento es un tipo y no nulo (aunque el array puede estar vacío). [Type!]! significa que el array siempre existirá y nunca será nulo (pero puede estar vacío).

3. Además, define campos para consultar publicaciones por autor y etiqueta, los cuales aceptan un argumento obligatorio:

```
postsByAuthor(nombre de usuario: String!): [!Publicación!]!  
postsByTag(etiqueta: String!): [!Publicación!]!
```

4. Por último, define un campo para consultar una publicación por id:

```
postById(id: ID!): Publicación  
. }
```

5. Ahora que hemos definido el esquema, aún necesitamos proporcionar solucionadores para todos esos campos. Gracias a nuestras funciones de servicio, esto es bastante sencillo: podemos simplemente llamarlas con los argumentos relevantes en funciones asíncronas, de la siguiente manera:

```
exportar const queryResolver = {  
    Consulta: {  
        prueba: () => {  
            devolver '¡Hola mundo desde GraphQL!'  
        },  
        publicaciones: async () => {  
            devolver lista de esperaAllPosts()  
        },  
        postsByAuthor: async (padre, { nombre de usuario }) => {  
            devolver lista de esperaPostsByAuthor(nombre de usuario)  
        },  
        postsByTag: async (padre, { etiqueta }) => {  
            devolver lista de esperaPostsByTag(etiqueta)
```

```
    },
    postById: async (padre, { id }) => {
      devolver esperar getPostById(id)
    },
  },
}
```

Las funciones de resolución \$e siempre reciben el objeto padre como primer argumento y un objeto con todos los argumentos como segundo argumento.

Hemos definido correctamente los campos para consultar las publicaciones. Sin embargo, el tipo de publicación aún no está definido, por lo que nuestras consultas GraphQL no funcionarán. Hagámoslo a continuación.

## Definición del tipo de publicación

Luego de definir el tipo Consulta, continuamos definiendo el tipo Post, de la siguiente manera:

1. Crea un nuevo archivo backend/src/graphql/post.js, donde importamos la función getUserInfoById para resolver el autor de una publicación más tarde:

```
importar { getUserInfoById } desde '../services/users.js'
```

2. \$en, define postSchema. Tenga en cuenta que la publicación consta de id, título, autor, contenido, etiquetas y las marcas de tiempo createdAt y updatedAt:

```
exportar const postSchema = `#graphql
  tipo Publicación {
    ¡hice!
    Título: ¡Cuerda!
    autor: Usuario
    Contenido: Cadena
    etiquetas: [¡Cadena!]
    creadoEn: Flotador
    actualizadoEn: Flotador
  }
`
```

En este caso, utilizamos [String!] para las etiquetas, y no [String!]!, porque el campo de etiquetas también puede ser inexistente/nulo.

Las marcas de tiempo \$e createdAt y updatedAt son demasiado grandes para caber en un entero con signo de 32 bits, por lo que su tipo debe ser Float en lugar de Int.

3. A continuación, defina un solucionador para el campo autor que obtenga al usuario que utiliza la función de servicio:

```
exportar const postResolver = { Publicación:  
  { autor:  
    async (publicación) => { devolver  
      esperar getUserInfoByld(publicación.autor) }, },  
  
}
```

Los resolvidores \$e para obtener publicaciones ya forman parte del esquema de consulta, por lo que no es necesario definir cómo obtener una publicación aquí. GraphQL sabe que los campos de consulta devuelven matrices de publicaciones y, por lo tanto, nos permite resolver campos adicionales en las publicaciones.

4. Edite backend/src/graphql/index.js y agregue postSchema y postResolver:

```
importar { querySchema, queryResolver } desde './query.js' importar { postSchema,  
postResolver } desde './post.js'  
  
exportar const typeDefs = [querySchema, postSchema] exportar const  
resolvers = [queryResolver, postResolver]
```

Después de definir el tipo de Publicación, continuemos con el tipo de Usuario.

## Definición del tipo de usuario

Al definir el tipo de publicación, usamos el tipo de usuario para definir el autor de una publicación. Sin embargo, aún no hemos definido el tipo de usuario. Hagámoslo ahora:

1. Cree un nuevo archivo backend/src/graphql/user.js e importe la función listPostsByAuthor aquí, ya que vamos a agregar una forma de resolver las publicaciones de un usuario al obtener un objeto de usuario, para mostrar cómo GraphQL puede manejar relaciones profundamente anidadas:

```
importar { listPostsByAuthor } desde '../services/posts.js'
```

2. Defina el esquema de usuario. Cada usuario en nuestro esquema GraphQL tiene un nombre de usuario y una publicación. campo, en el cual resolveremos todos los posts que el usuario haya escrito:

```
exportar const userSchema = `#graphql  
  tipo Usuario  
    { nombre de usuario: String!  
      publicaciones: [!Publicación!]!  
    }`
```

## Información

No especificamos ninguna otra propiedad aquí, ya que solo devolvemos el nombre de usuario en nuestra función de servicio `getUserInfoById`. Si quisieramos obtener también el ID de usuario, tendríamos que devolverlo desde esa función. No devolvemos únicamente el objeto de usuario completo, ya que esto podría representar una vulnerabilidad de seguridad, exponiendo datos internos como la contraseña (o la información de facturación en algunas aplicaciones).

3. A continuación, defina `userResolver`, que obtiene todas las publicaciones del usuario actual:

```
exportar const userResolver = {
  Usuario: {
    publicaciones: async (usuario) => {
      devolver lista de esperaPostsByAuthor(usuario.nombredeusuario)
    },
  },
}
```

4. Edite `backend/src/graphql/index.js` y agregue `userSchema` y `userResolver`:

```
importar { querySchema, queryResolver } desde './query.js'
importar { postSchema, postResolver } desde './post.js'
importar { userSchema, userResolver } desde './user.js'

exportar const typeDefs = [querySchema, postSchema, userSchema]
exportar const resolvers = [queryResolver, postResolver, userResolver]
```

¡Después de definir el tipo de Usuario, probemos algunas consultas profundamente anidadas!

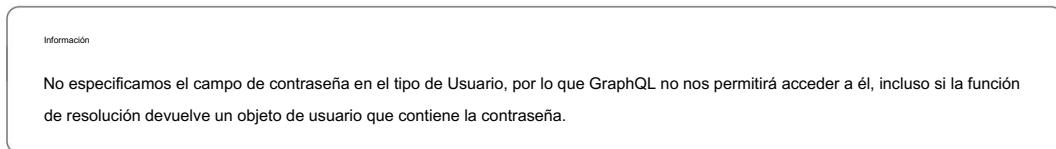
## Probando consultas profundamente anidadas

Ahora que hemos definido con éxito nuestros esquemas y resolvers GraphQL, ¡podemos comenzar a consultar nuestra base de datos usando GraphQL!

Por ejemplo, ahora podemos obtener una lista completa de todas las publicaciones, con su ID, título y nombre de usuario del autor, de la siguiente manera:

```
consulta GetPostsOverview {
  publicaciones {
    información
    título
    autor {
      nombre de usuario
    }
  }
}
```

Ejecute la consulta anterior en la interfaz de Apollo. Como podemos ver, la consulta obtiene todas las publicaciones, selecciona el ID, el título y el autor de cada publicación, y luego resuelve el nombre de usuario de cada instancia de autor. La consulta `$is` nos permite obtener todos los datos necesarios en la página de resumen en una sola solicitud, sin necesidad de realizar solicitudes separadas para resolver los nombres de usuario de los autores.



Ahora, probemos una consulta que obtiene una publicación por ID y luego encuentra otras publicaciones del mismo autor. `$is` se podría usar para, por ejemplo, recomendar otros artículos para ver del mismo autor después de que alguien haya terminado de leer una publicación:

1. Podemos generar automáticamente una consulta en la interfaz de Apollo borrando el contenido del cuadro de texto Operación y luego seleccionando Consulta de Tipos Raíz en la Documentación Barra lateral en la izquierda. Ahora haga clic en el botón + junto al campo `postById` en la izquierda, lo que define automáticamente una variable de consulta, con el siguiente aspecto:

The screenshot shows the Apollo GraphQL interface with two panels. The left panel displays the schema structure under 'Documentation' for 'Root'. It shows 'Root' has a 'query: Query' field and a 'mutation: Mutation' field. The 'query' field has a 'postById(...): Post' field selected. The right panel shows the generated query in the 'Operation' tab. The query is:

```
consulta PostById($postByIdId: ID!) {
  postById(id: $postByIdId) {
```

Below the query, the 'Documentation' sidebar shows the path: Root > Query > postById. The 'Arguments' section shows 'id: ID!' selected. The 'Fields' section lists fields like 'author: User', 'contents: String', 'createdAt: Float', etc.

Figura 11.3 – Generación automática de una consulta mediante la interfaz Apollo

2. Dentro de la publicación, ahora podemos obtener el título, el contenido y los valores del autor de la publicación:

```
título  
contenido  
autor {
```

3. Dentro del campo autor, obtenemos el nombre de usuario y los ID y títulos de sus publicaciones:

```
nombre de usuario  
publicaciones  
{ id  
    título  
}  
}  
}  
}
```

4. En la parte inferior de la interfaz de Apollo, hay una sección de Variables , que debemos completar con una identificación que existe en nuestra base de datos:

```
{  
  "postByIdId": "<INTRODUCIR ID DE LA BASE DE DATOS>"  
}
```

5. Ejecute la consulta y verá que la publicación y el autor están resueltos, y todas las publicaciones escritas por ese mismo autor también aparecen correctamente, como se muestra en la siguiente captura de pantalla:

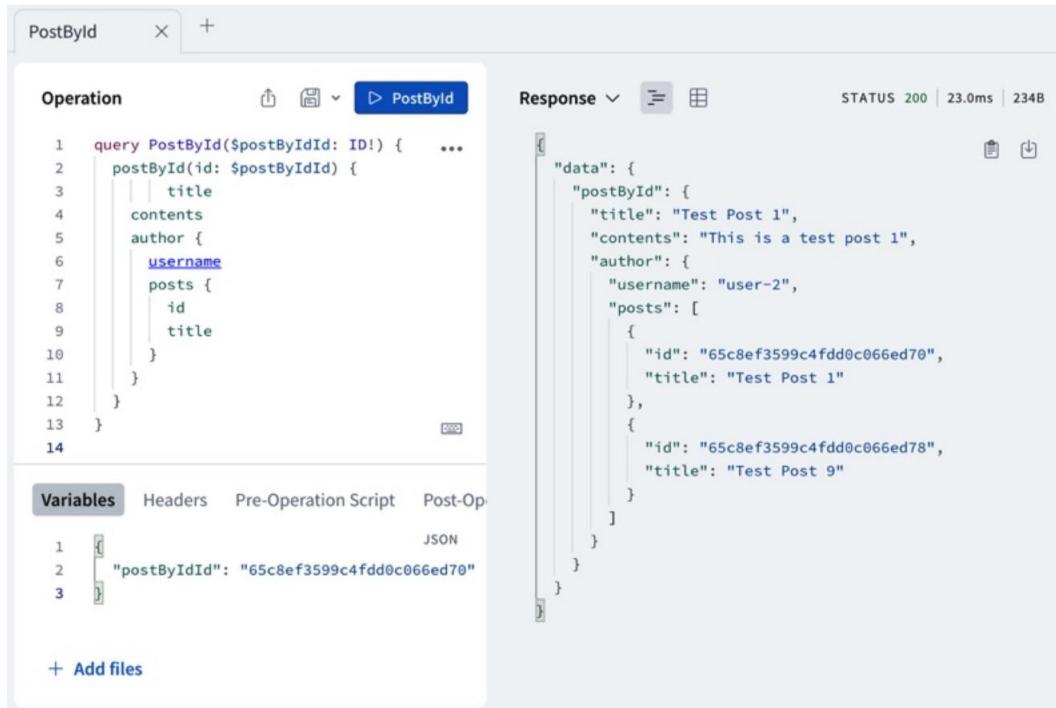


Figura 11.4 – Ejecución de consultas profundamente anidadas en GraphQL

A continuación, aprendamos cómo proporcionar argumentos a los campos definiendo tipos de entrada.

## Implementación de tipos de entrada

Ya aprendimos a definir tipos regulares en GraphQL, pero ¿qué ocurre si tenemos una forma común de proporcionar argumentos a los campos? Por ejemplo, las opciones para consultar las publicaciones son siempre las mismas (`sortBy` y `sortOrder`). No podemos usar un tipo regular para esto; en su lugar, necesitamos definir un tipo de entrada. Siga estos pasos para implementar opciones de consulta en GraphQL:

1. Edite `backend/src/graphql/query.js` y defina un tipo de entrada en el esquema:

```

exportar const querySchema = `#graphql
  entrada PostsOptions { sortBy:
    Cadena sortOrder:
    Cadena
  }
}

```

2. \$en, utiliza el tipo de entrada como argumento para elds, de la siguiente manera:

```
tipo Consulta {  
    prueba: String  
    posts(opciones: PostsOptions): [iPublicación!]!  
    postsByAuthor(nombre de usuario: String!, opciones: PostsOptions): [iPublicación!]!  
  
    postsByTag(etiqueta: String!, opciones: PostsOptions): [iPublicación!]! postById(id: ID!, opciones:  
        PostsOptions): Publicación  
    . }
```

3. Ahora, edite los solucionadores para pasar opciones a las funciones de servicio:

```
publicaciones: async (padre, { opciones }) => { devolver await  
    listAllPosts(opciones) }, publicacionesPorAutor: async  
  
(padre, { nombre de usuario, opciones }) => { devolver await listPostsByAuthor(nombre de  
    usuario, opciones) }, publicacionesPorTag: async (padre, { etiqueta,  
  
    opciones }) => { devolver await listPostsByTag(etiqueta, opciones) },
```

4. Pruebe la siguiente consulta para ver si las publicaciones están ordenadas correctamente:

```
consulta SortedPosts($opciones: PostsOptions)  
{ publicaciones(opciones: $opciones) {  
    identificación  
    título  
    creadoEn  
    actualizadoEn  
    }  
}
```

5. Establezca las siguientes variables:

```
{  
    "opciones": {  
        "sortBy": "updatedAt", "sortOrder":  
        "ascendente"  
    }  
}
```

6. Ejecute la consulta presionando el botón Reproducir y debería ver que la respuesta está ordenada por la marca de tiempo updatedAt en orden ascendente.

Ahora que hemos implementado con éxito la funcionalidad para consultar nuestra base de datos usando GraphQL, pasemos a implementar una forma de crear una nueva publicación usando mutaciones GraphQL.

## Implementación de autenticación y mutaciones GraphQL

Ahora implementaremos una forma de crear nuevas publicaciones con GraphQL. Para definir campos que cambien el estado de la base de datos, necesitamos crearlos bajo el tipo de mutación. Sin embargo, antes de hacerlo, debemos implementar la autenticación en GraphQL para poder acceder al usuario conectado al crear una publicación.

### Agregar autenticación a GraphQL

Dado que usamos GraphQL con Express, podemos usar cualquier middleware de Express con GraphQL y pasarlo a nuestros resolvers como contexto. Por lo tanto, podemos usar el middleware express-jwt existente para analizar el JWT. Comencemos ahora a agregar autenticación a GraphQL:

1. Nuestra configuración actual del middleware requireAuth garantiza que el usuario haya iniciado sesión y genera un error si no lo está. Sin embargo, esto supone un problema al pasar el contexto de autenticación a GraphQL, ya que no todas las consultas requieren autenticación. Ahora crearemos un nuevo middleware optionAuth que no requiere credenciales para procesar una solicitud. Edite backend/src/middleware/jwt.js y defina el siguiente middleware nuevo:

```
exportar const opcionalAuth = expressjwt({  
    secreto: () => proceso.env.JWT_SECRET,  
    algoritmos: ['HS256'],  
    credencialesRequeridas: falso,  
})
```

2. Ahora, edite backend/src/app.js e importe allí el middleware opcionalAuth:

```
importar {optionalAuth} desde './middleware/jwt.js'
```

3. Edite la llamada app.use() donde definimos la ruta /graphql y agregue el OptionalAuth middleware, de manera similar a como lo hicimos para las rutas:

```
apolloServer.start().then(() =>  
    aplicación.uso  
    '/graphql',  
    Autorización opcional,
```

4. \$en, agrega un segundo argumento al Apollo expressMiddleware, definiendo un contexto función que proporciona req.auth a los resolutores GraphQL como contexto:

```
expressMiddleware(apolloServer, { contexto: async  
  ({ req }) => { return { auth: req.auth } }, }),  
  
),  
)
```

A continuación, pasemos a la implementación de mutaciones en GraphQL.

## Implementando mutaciones

Ahora que hemos añadido la autenticación a GraphQL, podemos definir nuestras mutaciones. Sigue estos pasos para crear mutaciones para el registro, el inicio de sesión y la creación de publicaciones:

1. Cree un nuevo archivo backend/src/graphql/mutation.js e importe GraphQLError (para generar un error NO AUTORIZADO cuando el usuario no ha iniciado sesión), así como las funciones createUser, loginUser y createPost:

```
importar { GraphQLError } desde 'graphql' importar  
{ createUser, loginUser } desde './services/users.js' importar { createPost } desde './services/  
posts.js'
```

2. Defina mutationSchema, en el que primero definimos los campos para registrar e iniciar sesión a los usuarios. El campo \$e signupUser devuelve un objeto de usuario y el campo loginUser devuelve un JWT:

```
exportar const mutationSchema = `#graphql tipo Mutación {  
  
  signupUser(nombre de usuario: String!, contraseña: String!): Usuario loginUser(nombre  
  de usuario: String!, contraseña: String!): String
```

3. \$en, define un campo para crear una nueva publicación a partir de un título determinado, contenido (opcional) y Etiquetas (opcionales). Devuelve una publicación recién creada:

```
  createPost(título: String!, contenido: String, etiquetas:  
  [Cadena]): Publicar }
```

4. Definir el resolver, en el que primero definimos los campos signupUser y loginUser, que son bastante sencillos:

```
exportar const mutationResolver = {
  Mutación:
    { signupUser: async (parent, { nombre de usuario, contraseña }) => { return await
        createUser({ nombre de usuario, contraseña }), loginUser: async (parent,
        { nombre de usuario, contraseña }) => { return await loginUser({ nombre de usuario,
        contraseña }) },
    }
}
```

5. A continuación, definimos el campo createPost. Aquí, primero accedemos a los argumentos pasados al campo y, como tercer argumento de la función de resolución, obtenemos el contexto creado anteriormente:

```
createPost: async (padre, { título, contenidos, etiquetas }, { auth }) => {
```

6. Si el usuario no ha iniciado sesión, el contexto de autenticación será nulo. En ese caso, se generará un error. y no crees una nueva publicación:

```
  si (!auth) { lanzar
    nuevo GraphQLError(
      'Es necesario estar autenticado para realizar esta
      acción.',
    )
    extensiones: { código:
      'NO AUTORIZADO', },
  }
}
```

7. De lo contrario, utilizamos auth.sub (que contiene el ID del usuario) y los argumentos proporcionados para crear una nueva publicación:

```
    devolver esperar createPost(auth.sub, { título, contenidos, etiquetas
  })
}

}, },
```

8. Edite backend/src/graphql/index.js y agregue mutationSchema y mutationResolver:

```
importar { querySchema, queryResolver } desde './query.js' importar { postSchema,
postResolver } desde './post.js'
```

```
importar { userSchema, userResolver } desde './user.js'  
importar { mutationSchema, mutationResolver } desde './mutation.js'  
  
exportar const typeDefs = [querySchema, postSchema, userSchema, mutationSchema]  
  
exportar const resolvers = [  
    solucionador de consultas,  
    postResolver,  
    usuarioResolver,  
    Resolvedor de mutaciones,  
]  
]
```

Después de implementar mutaciones, aprendamos a usarlas.

## Uso de mutaciones

Tras definir las posibles mutaciones, podemos usarlas ejecutándolas en la interfaz de Apollo. Siga estos pasos para registrar un usuario, iniciar sesión y, finalmente, crear una publicación; todo esto con GraphQL:

1. Vaya a <http://localhost:3001/graphql> para ver la interfaz de Apollo. Defina una nueva mutación que registre a un usuario con un nombre de usuario y contraseña determinados, y devuelva el nombre de usuario si el registro se realizó correctamente:

```
mutación SignupUser($nombre de usuario: String!, $contraseña: String!) {  
    signupUser(nombre de usuario: $nombre de usuario, contraseña: $contraseña) {  
        nombre de usuario  
    }  
}
```

Consejo

Puede utilizar la sección Documentación en el archivo volviendo a Tipos de raíz, haciendo clic en Mutación y luego haciendo clic en el ícono + al lado de signupUser. \$en, haga clic en el ícono + al lado del campo de nombre de usuario . \$is creará automáticamente el código anterior.

2. Edite las variables en la parte inferior e ingrese un nombre de usuario y una contraseña:

```
{  
    "nombre de usuario": "graphql",  
    "contraseña": "gql"  
}
```

3. Ejecute la mutación SignupUser presionando el botón de reproducción.

4. A continuación, cree una nueva mutación para iniciar sesión como usuario:

```
mutación LoginUser($nombre de usuario: String!, $contraseña: String!) {  
    loginUser(nombre de usuario: $nombre de usuario, contraseña: $contraseña)  
}
```

5. Ingrese las mismas variables que antes y presione el botón de reproducción. La respuesta contiene un JWT. Copie y guarde el JWT para su uso posterior.

6. Define una nueva mutación para crear una publicación. \$is mutation devuelve Post, por lo que podemos obtener el id. Título y valores de nombre de usuario para el autor:

```
mutación CreatePost($título: Cadena!, $contenido: Cadena, $etiquetas: [Cadena]) {  
  
    createPost(título: $título, contenido: $contenido, etiquetas: $etiquetas) {  
        identificación  
        título  
        autor {  
            nombre de usuario  
        }  
    }  
}
```

\$is es un ejemplo de cómo GraphQL realmente destaca. Podemos resolver el nombre de usuario del autor después de crear la publicación para comprobar si realmente se creó con el usuario correcto, ya que podemos acceder a los solucionadores definidos para la publicación, ¡incluso en mutaciones! Como pueden ver, GraphQL es muy flexible.

7. Ingrese las siguientes variables:

```
{  
    "título": "Publicación GraphQL",  
    "contents": "¡Esto se publica desde GraphQL!"  
}
```

8. Seleccione la pestaña Encabezados , presione el botón Nuevo encabezado , ingrese Autorización como clave de encabezado y Portador <Pegar aquí el JWT previamente copiado> como valor. Luego presione el botón Reproducir para enviar la mutación.

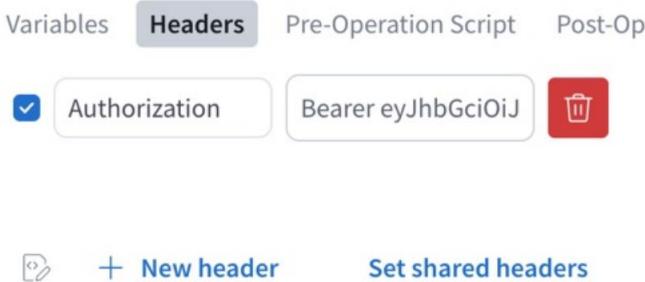


Figura 11.5 – Agregar el encabezado de autorización en la interfaz de Apollo

9. En la respuesta, puedes ver que la publicación se creó correctamente y que el autor se configuró y resolvió correctamente.

Después de haber implementado consultas y mutaciones GraphQL para nuestras aplicaciones de blog, concluyamos el capítulo brindando una descripción general de los conceptos avanzados de GraphQL.

## Descripción general de los conceptos avanzados de GraphQL

De fábrica, GraphQL viene con un conjunto de tipos escalares:

- Int: Un entero con signo de 32 bits
- Flotante: un valor de punto flotante con signo de doble precisión
- Cadena: una secuencia de caracteres codificada en UTF-8
- Booleano: puede ser verdadero o falso
- ID: Un identificador único, serializado como una cadena, pero que significa que no es legible para humanos.

GraphQL también permite la definición de enumeraciones, que son un tipo especial de escalar. Están restringidas a ciertos valores. Por ejemplo, podríamos usar la siguiente enumeración para distinguir diferentes tipos de publicaciones:

```
enumeración PostType {  
    INÉDITO,  
    NO ESTANTE EN LA LISTA,  
    PÚBLICO  
}
```

En Apollo, las enumeraciones se manejarán como cadenas que solo pueden tener ciertos valores, pero esto puede ser diferente en otras implementaciones de GraphQL.

Muchas implementaciones de GraphQL también permiten definir tipos escalares personalizados. Apollo, por ejemplo, admite la definición de tipos escalares personalizados.

## Fragments

Cuando se accede regularmente al mismo tipo de campos, podemos crear un fragmento para simplificar y estandarizar el acceso. Por ejemplo, si solemos resolver usuarios y estos tienen campos como nombre de usuario, foto de perfil, nombre completo y biografía, podríamos crear el siguiente fragmento:

```
fragment UserInfo sobre el usuario {  
    nombre de usuario  
    foto de perfil  
    nombre completo  
    biografía  
}
```

El fragmento \$is puede usarse en consultas. Por ejemplo, vea este fragmento:

```
{  
    publicaciones  
    { autor {  
        ...Información del usuario  
    }  
}
```

Los fragmentos son especialmente útiles cuando se utiliza el mismo tipo de estructura de campo varias veces en la misma consulta. Por ejemplo, si un autor tuviera "followBy" y "sigue campos", podríamos resolver todos los usuarios de la siguiente manera:

```
{  
    publicaciones  
    { autor {  
        ...Información del usuario  
        seguido por {  
            ...Información del usuario  
  
            } sigue  
            { ...Información del usuario  
        }  
    }  
}
```

## Introspección

La introspección nos permite consultar los propios esquemas definidos para comprender los datos que el servidor puede proporcionarnos. En esencia, se trata de consultar los esquemas definidos por el servidor GraphQL. Podemos usar el campo `__schema` para obtener todos los esquemas. Un esquema consta de tipos, cuyos nombres tienen valores.

Por ejemplo, podemos utilizar la siguiente consulta para obtener todos los tipos definidos por nuestro servidor:

```
{  
  __esquema {  
    tipos {  
      nombre  
    }  
  }  
}
```

Si ejecuta esta consulta en nuestro servidor, obtendrá (entre otros tipos) nuestros tipos definidos de consulta, publicación, usuario y mutación.

Las consultas de introspección son muy potentes y permiten obtener mucha información sobre las posibles consultas y mutaciones. De hecho, la interfaz de Apollo utiliza la introspección para generar la documentación.

¡Barra lateral y para completar automáticamente los campos para nosotros!

## Resumen

En este capítulo, aprendimos qué es GraphQL y cómo puede ser más flexible que REST, a la vez que requiere menos código repetitivo, especialmente al consultar objetos profundamente anidados. Luego, implementamos GraphQL en nuestro backend y creamos varios tipos, consultas y mutaciones. También aprendimos a integrar la autenticación JWT en GraphQL. Finalmente, concluimos el capítulo aprendiendo conceptos avanzados, como el sistema de tipos, los fragmentos y la introspección.

En el próximo capítulo, Capítulo 12, Interfaz con GraphQL en el frontend usando Apollo Client, aprenderemos cómo acceder e integrar GraphQL en nuestro frontend usando React y la biblioteca Apollo Client.

Machine Translated by Google

# 12

## Interfaz con GraphQL en el frontend usando Cliente Apollo

Tras implementar con éxito un backend GraphQL con Apollo Server en el capítulo anterior, ahora interactuaremos con nuestra nueva API GraphQL en el frontend mediante Apollo Client. Apollo Client es una biblioteca que facilita y agiliza la interacción con las API GraphQL. Comenzaremos reemplazando la obtención de la lista de publicaciones con consultas GraphQL y, a continuación, resolveremos los nombres de usuario de los autores sin necesidad de consultas adicionales, demostrando así la potencia de GraphQL. A continuación, añadiremos variables a la consulta para permitir la configuración de filtros y opciones de ordenación. Finalmente, aprenderemos a usar mutaciones en el frontend.

En este capítulo cubriremos los siguientes temas principales:

- Configuración del cliente Apollo y realización de nuestra primera consulta
- Uso de variables en consultas GraphQL
- Uso de mutaciones en el frontend

### 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-modernos-de-React-Full-Stack/árbol/principal/cap. 12>.

El video de \$e CiA para este capítulo se puede encontrar en: [https://youtu.be/GI\\_5i9DR\\_xA](https://youtu.be/GI_5i9DR_xA).

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.

## Configuración del cliente Apollo y realización de nuestra primera consulta

Antes de comenzar a realizar consultas GraphQL en el frontend, debemos configurar Apollo Client. Apollo Client es la contraparte frontend de Apollo Server, que ya hemos estado usando en el backend. Si bien no es obligatorio usar Apollo Client (también podríamos simplemente hacer un POST) Solicitud al endpoint /graphql, Apollo Client facilita y agiliza la interacción con GraphQL. Además, incluye funciones adicionales, como el almacenamiento en caché, de forma predeterminada.

Siga estos pasos para configurar Apollo Client:

1. Copie la carpeta ch11 existente a una nueva carpeta ch12, de la siguiente manera:

```
$ cp -R ch11 ch12
```

2. Abra la carpeta ch12 en VS Code.

3. Instale las dependencias @apollo/client y graphql:

```
$ npm install @apollo/client@3.9.5 graphql@16.8.1
```

4. Edite .env y agregue una nueva variable de entorno que apunte al punto final de nuestro servidor GraphQL:

```
VITE_GRAPHQL_URL="http://localhost:3001/graphql"
```

5. Edite src/App.jsx e importe ApolloClient, InMemoryCache y ApolloProvider del paquete @apollo/client:

```
importar { ApolloProvider } desde '@apollo/client/react/index.js'  
importar { ApolloClient, InMemoryCache } desde '@apollo/client/  
núcleo/index.js'
```

Al momento de escribir esto, hay un problema con las importaciones de ESM en Apollo Client, por lo que debemos importar directamente desde los archivos index.js.

6. Cree una nueva instancia de Apollo Client, que apunte al punto final GraphQL y usando InMemoryCache:

```
constante apolloClient = nuevo ApolloClient({  
    uri: import.meta.env.VITE_GRAPHQL_URL,  
    caché: nuevo InMemoryCache(),  
})
```

7. Ajuste el componente de la aplicación para agregar ApolloProvider, proporcionando el contexto del cliente Apollo a toda nuestra aplicación:

```
función de exportación App({ children }) {  
  devolver (  
    <Proveedor de casco>  
    <Cliente de ApolloProvider={apolloClient}>  
      <QueryClientProvider cliente={queryClient}>  
        <AuthContextProvider>{hijos}</AuthContextProvider>  
      </ProveedorClienteConsulta>  
    </ProveedorApolo>  
  </ProveedorDeCasco>  
)  
}
```

8. También crearemos un archivo de configuración GraphQL para que la extensión GraphQL de VS Code pueda autocompletar y validar las consultas. Cree un nuevo archivo graphql.config.json en la raíz del proyecto, con el siguiente contenido:

```
{  
  "esquema": "http://localhost:3001/graphql",  
  "documentos": "src/api/graphql/**/*.{js,jsx}"  
}
```

El esquema \$e define la URL del punto final de GraphQL y los documentos definen dónde encontrar los archivos que contienen consultas GraphQL. Colocaremos las consultas GraphQL en src/ carpeta api/graphql/ más adelante.

9. Asegúrese de que Docker y el contenedor de la base de datos se estén ejecutando, luego inicie el backend de la siguiente manera:

```
$ cd backend/  
$ npm ejecuta dev
```

Mantenga el backend en ejecución durante este capítulo, para que la extensión GraphQL pueda acceder al punto final GraphQL.

10. Reinicie la extensión GraphQL de VS Code. Puede hacerlo accediendo a la paleta de comandos de VS Code (Ctrl + Mayús + P en Windows/Linux y Cmd + Mayús + P en macOS) y escribiendo GraphQL: Reinicio manual.

## Consulta de publicaciones desde el frontend usando GraphQL

Ahora que Apollo Client está configurado y listo para usarse, definamos nuestra primera consulta GraphQL: una consulta simple para obtener todas las publicaciones.

Siga estos pasos para definir la consulta y utilizarla en nuestra aplicación:

1. Crea una nueva carpeta `src/api/graphql/`, donde colocaremos nuestras consultas GraphQL.
2. Dentro de esta carpeta, crea un nuevo archivo `src/api/graphql/posts.js`.
3. En el archivo `src/api/graphql/posts.js`, importe la función `gql` desde `@apollo/client`:

```
importar { gql } desde '@apollo/client/core/index.js'
```

4. Defina una nueva consulta `GET_POSTS`, que recupera todas las propiedades relevantes para una publicación (excepto el autor, que vendrá más adelante):

```
exportar const GET_POSTS = gql` consulta getPosts
  { publicaciones { id
    título
    contenido
    etiquetas actualizadas en
    creadoEn
  }
}
```

Deberías ver que la extensión GraphQL nos ofrece opciones de autocompletado para los tipos que definimos en nuestro backend. Si ingresamos un nombre de campo incorrecto, también nos advertirá que este no existe en el tipo.

5. Edite `src/pages/Blog.jsx` e importe el gancho `useQuery` desde `@apollo/client`:

```
importar { useQuery como useGraphQLQuery } desde '@apollo/client/react/index.js'
```

Cambiamos el nombre del gancho `useQuery` de Apollo Client a `useGraphQLQuery` para evitar confusiones con el gancho `useQuery` de TanStack React Query.

6. Importe la consulta `GET_POSTS` previamente definida:

```
importar { GET_POSTS } desde './api/graphql/posts.js'
```

7. Elimine las importaciones a `useQuery` y `getPosts`:

```
importar { useQuery } desde '@tanstack/react-query' importar { getPosts } desde
'./api/posts.js'
```

8. Elimine el gancho useQuery existente:

```
const postsQuery = useQuery({ queryKey:  
  ['posts', { autor, sortBy, sortOrder }], queryFn: () => getPosts({ autor, sortBy,  
  sortOrder }), }) const posts = postsQuery.data ?? []
```

9. Reemplácelo con el siguiente gancho:

```
const postsQuery = useGraphQLQuery(GET_POSTS) const posts =  
  postsQuery.data?.posts ?? []
```

10. Asegúrate de estar en la raíz del proyecto, luego ejecuta el frontend de la siguiente manera:

```
$ npm ejecuta dev
```

Ahora, abre el frontend en <http://localhost:5173/> y verás que los títulos de las entradas se muestran correctamente. Sin embargo, los enlaces a las entradas no funcionan y hay un error en la consola. Hay una ligera diferencia entre los resultados de GraphQL y la API REST: la API REST devuelve el ID de las entradas como una propiedad `_id`, mientras que GraphQL lo devuelve como una propiedad `id`.

Ajustemos nuestro código para adaptarlo a este cambio ahora:

1. Edite `src/components/Post.jsx` y cambie la propiedad `_id` a `id`:

```
función de exportación Post({ título,  
  
  contenido,  
  autor,  
  identificación,
```

2. Además, actualice el nombre de la variable donde se utiliza:

```
<Enlace a={`/posts/${id}/${slug(title)}}`>
```

3. Asegúrate de actualizar también `propTypes`:

```
Post.propTypes = {  
  título: PropTypes.string.isRequired, contenido:  
    PropTypes.string, autor: PropTypes.string,  
  id: PropTypes.string.isRequired,
```

4. Ahora que la propiedad ha cambiado, edite src/pages/ViewPost.jsx y pase la nueva propiedad como sigue:

```
{correo ? (
  <Publicar {...publicar} id={postId} publicación completa />
) : (
  `No se encontró la publicación con id ${postId}.`
)}
```

Tras guardar todos los archivos, la interfaz debería actualizarse y mostrar correctamente la lista de publicaciones con enlaces funcionales. Ahora, solo queda mostrar los nombres de usuario de los autores para restaurar la funcionalidad original.

### Resolver nombres de usuario de autores en una sola consulta

En lugar de resolver cada nombre de usuario de autor por separado, ahora podemos obtenerlos todos a la vez en una sola consulta, ¡gracias al poder de GraphQL! Aprovechemos esta capacidad para refactorizar un poco nuestro código y simplificarlo y mejorar el rendimiento:

1. Comience editando la consulta GraphQL en src/api/graphql/posts.js y agregando el autor.

campo de nombre de usuario, como sigue:

```
exportar const GET_POSTS = gql`  
consulta getPosts {  
  publicaciones {  
    autor {  
      nombre de usuario  
    }  
  }  
}
```

2. \$en, edita el componente src/components/User.jsx. Reemplaza todo el componente por el siguiente, más simple:

```
importar PropTypes desde 'prop-types'  
  
función de exportación Usuario({nombre de usuario}) {  
  devolver <b>{nombre de usuario}</b>  
}  
  
Usuario.propTypes = {  
  nombre de usuario: PropTypes.string.isRequired,  
}
```

Ya no es necesario obtener la información del usuario aquí, ya que podemos mostrar directamente el nombre de usuario desde la respuesta GraphQL.

3. A continuación, edite src/components/Post.jsx y pase todo el objeto de autor al usuario. componente, como sigue:

```
Escrito por <Usuario {...autor} />
```

4. También necesitamos ajustar propTypes ahora para aceptar un objeto de autor completo para la publicación. componente, en lugar de un ID de usuario:

```
autor: PropTypes.shape(Usuario.propTypes),
```

5. Edite src/pages/ViewPost.jsx y pase todo el objeto de autor al componente Post:

```
<Publicación {...publicación} id={postId} autor={userInfo} publicación completa  
/>
```

Afortunadamente, ya estamos resolviendo el nombre de usuario para las metaetiquetas en esta página, por lo que tampoco necesitamos hacer una consulta adicional aquí.

6. Sin embargo, en el encabezado, necesitamos realizar una consulta adicional para resolver el nombre de usuario cuando un usuario inicia sesión.

Edite src/components/Header.jsx e importe el gancho useQuery y la función API getUserInfo:

```
importar { useQuery } desde '@tanstack/react-query'  
importar { getUserInfo } desde './api/users.js'
```

7. \$en, ajuste el componente para obtener el ID de usuario del token (el subcampo del JWT) y realizar una consulta de información del usuario:

```
función de exportación Header() {  
    const [token, setToken] = useAuth()  
  
    constante { sub } = token ? jwtDecode(token) : {}  
    constante userInfoQuery = useQuery({  
        consultaKey: ['usuarios', sub],  
        consultaFn: () => obtenerInformaciónUsuario(sub),  
        habilitado: Boolean(sub),  
    })  
    constante userInfo = userInfoQuery.data
```

8. Por último, verificamos si pudimos resolver la consulta de información del usuario (en lugar de solo verificar) Para el token). En ese caso, pasamos la información del usuario al componente Usuario:

```
si (token && información del usuario) {  
    devolver (  
        <navegación>  
        Inició sesión como <User {...userInfo} />
```

También eliminamos la decodificación del token aquí, como ya hicimos anteriormente.

Ahora usamos GraphQL para obtener la lista de publicaciones y resolver los nombres de usuario de los autores en una sola solicitud. Sin embargo, los filtros y la ordenación ya no funcionan, ya que aún no pasamos esta información a la consulta GraphQL.

En la siguiente sección, vamos a presentar variables para filtrar y ordenar nuestras consultas GraphQL.

## Uso de variables en consultas GraphQL

Para añadir compatibilidad con filtros y ordenación, necesitamos añadir variables a nuestra consulta GraphQL. Podemos completar estas variables al ejecutar la consulta.

Siga estos pasos para agregar variables a la consulta:

1. Edite `src/api/graphql/posts.js` y ajuste la consulta para aceptar una variable `$options`:

```
export const GET_POSTS = gql`  
  consulta getPosts($opciones: PostsOptions) {
```

2. Ahora, pasa la variable `$options` al solucionador de publicaciones, para el cual ya implementamos un argumento de opciones en el capítulo anterior:

```
    publicaciones(opciones: $opciones) {
```

3. Ahora solo necesitamos pasar las opciones al ejecutar la consulta. Editar `src/pages/Blog.jsx` y pasa la variable, de la siguiente manera:

```
constante postsQuery = useGraphQLQuery(GET_POSTS, {  
  variables: { opciones: { sortBy, sortOrder } },  
})
```

4. ¡Vaya a la interfaz del blog y cambie el orden de clasificación a ascendente para ver la variable en acción!

## Uso de fragmentos para reutilizar partes de consultas

Ahora que la ordenación funciona, solo necesitamos añadir el filtro por autor. Para ello, necesitamos añadir una segunda consulta para `postsByAuthor`. Como se puede imaginar, esta consulta debería devolver los mismos campos que la consulta de publicaciones. Podemos usar un fragmento para reutilizar los campos en ambas consultas, como se indica a continuación:

1. Edite `src/api/graphql/posts.js` y defina un nuevo fragmento en GraphQL que contenga todos los campos que necesitamos de una publicación:

```
export const POST_FIELDS = gql`  
  fragment PostFields en Post {  
    identificación  
    título  
    contenido
```

```
etiquetas actualizadas en
creadoEn
autor {
    nombre de usuario
}
.
```

El fragmento se define dándole un nombre (PostFields) y especificando en qué tipo se puede utilizar (en Post). Luego, todos los campos del tipo especificado se pueden consultar en el fragmento.

2. Para utilizar el fragmento, primero tenemos que incluir su definición en la consulta GET\_POSTS:

```
exportar const GET_POSTS = gql` 
${POST_FIELDS}
consulta getPosts($opciones: PostsOptions) {
```

3. Ahora, en lugar de enumerar todos los campos manualmente, podemos usar el fragmento:

```
publicaciones(opciones: $opciones) {
    ...Campos de publicación
}
.
```

La sintaxis para usar un fragmento es como la desestructuración de objetos en JavaScript, donde todas las propiedades definidas en un objeto se propagan a otro objeto.

Nota

A veces es necesario reiniciar la extensión GraphQL de VS Code para detectar fragmentos correctamente. Puedes hacerlo accediendo a la paleta de comandos de VS Code (Ctrl + Mayús + P en Windows/ Linux y Crnd + Mayús + P en macOS) y escribiendo GraphQL: Reinicio manual.

4. A continuación, definimos una segunda consulta, donde consultamos las publicaciones por autor y obtenemos todos los campos necesarios con el fragmento:

```
exportar const GET_POSTS_BY_AUTHOR = gql` $ 
{POST_FIELDS}
consulta getPostsByAuthor($autor: String!, $opciones: PostsOptions)
{ postsByAuthor(nombre
    de usuario: $autor, opciones: $opciones) {
        ...Campos de publicación
}
.
}
```

Definimos la variable \$author como necesaria para esta consulta (usando un signo de exclamación después del tipo). Esto es necesario porque el campo postsByAuthor también requiere que se configure el primer argumento (nombre de usuario).

5. Edite src/pages/Blog.jsx e importe la consulta recién definida:

```
importar { OBTENER_POSTS, OBTENER_POSTS_POR_AUTOR } desde './api/graphql/publicaciones.js'
```

6. \$en, ajusta el gancho para usar la consulta GET\_POSTS\_BY\_AUTHOR si el autor está definido:

```
const postsQuery = useGraphQLQuery(autor ? OBTENER_POSTS_POR_AUTOR: GET_POSTS, {
```

7. Pase la variable autor a la consulta:

```
variables: { autor, opciones: { sortBy, sortOrder } },  
})
```

8. Por último, debemos ajustar cómo seleccionamos los resultados porque el campo postsByAuthor de la consulta GET\_POSTS\_BY\_AUTHOR devolverá los resultados en data.postsByAuthor, mientras que la consulta GET\_POSTS usa el campo posts, que devuelve los resultados en data.posts.

Como no existe ningún caso en el que se devuelvan ambos campos a la vez, simplemente podemos hacer lo siguiente:

```
const posts = postsQuery.data?.postsByAuthor ?? postsQuery.datos?.publicaciones ?? []
```

9. Vaya al frontend e intente filtrar por autor. ¡El filtro funciona nuevamente ahora!

Como podemos ver, los fragmentos son muy útiles para reutilizar los mismos campos en múltiples consultas. Ahora que nuestra lista de publicaciones está completamente refactorizada para usar GraphQL, pasemos al uso de mutaciones en el frontend, lo que nos permitirá migrar las funcionalidades de registro, inicio de sesión y creación de publicaciones a GraphQL.

## Uso de mutaciones en el frontend

Como aprendimos en el capítulo anterior, las mutaciones en GraphQL se utilizan para cambiar el estado del backend (similar a las solicitudes POST en REST). Ahora implementaremos mutaciones para el registro y el inicio de sesión en nuestra aplicación.

Siga estos pasos:

1. Cree un nuevo archivo src/api/graphql/users.js e importe gql:

```
importar { gql } desde '@apollo/client/core/index.js'
```

2. \$en, define una nueva mutación SIGNUP\_USER, que toma un nombre de usuario y una contraseña y llama al campo de mutación signupUser:

```
exportar const SIGNUP_USER = gql`  
    mutación signupUser($nombre de usuario: String!, $contraseña: String!) {  
        signupUser(nombre de usuario: $nombre de usuario, contraseña: $contraseña) {  
            nombre de usuario  
        }  
    }`
```

3. Edite src/pages/Signup.jsx y reemplace el gancho "useMutation" de TanStack React Query por el de Apollo Client. Al igual que hicimos con "useQuery", también renombraremos este gancho a "useGraphQLMutation" para evitar confusiones.

```
importar { useMutation como useGraphQLMutation } desde '@apollo/  
cliente/react/index.js'
```

4. Además, reemplace la importación de la función de registro con una importación de SIGNUP\_Mutación USUARIO:

```
importar { SIGNUP_USER } desde './api/graphql/users.js'
```

5. Reemplace el gancho de mutación existente con lo siguiente:

```
const [signupUser, { cargando }] = useGraphQLMutation(SIGNUP_  
USUARIO, {  
    variables: { nombre de usuario, contraseña },  
    onCompleted: () => navegar('/login'),  
    onError: () => alert('¡Error al registrarse!'),  
})
```

Como se puede observar, el gancho de mutación del Cliente Apollo tiene una API ligeramente diferente a la del gancho de mutación de Consultas de React de TanStack. Devuelve un array con una función para llamar a la mutación y un objeto con el estado de carga, el estado de error y los datos. Al igual que el gancho useGraphQLQuery, también acepta la mutación como primer argumento y un objeto con variables como segundo . Además, la función onSuccess se llama onCompleted en el Cliente Apollo.

6. Cambie la función handleSubmit de la siguiente manera:

```
constante handleSubmit = (e) => {  
    e.preventDefault()  
    signupUser()  
}
```

7. Por último, cambie el botón de envío de la siguiente manera:

```
<entrada
  tipo='enviar'
  valor={cargando ? 'Registrando...' : 'Registrarse'} deshabilitado={!usuario
  || !contraseña || cargando} />
```

Ahora la función de registro se ha migrado correctamente a GraphQL. A continuación, migre la función de inicio de sesión.

### Migración del inicio de sesión a GraphQL

Refactorizar la funcionalidad de inicio de sesión a GraphQL es muy similar a la funcionalidad de registro, así que repasemos rápidamente los pasos:

1. Edite src/api/graphql/users.js y defina una mutación para iniciar sesión:

```
exportar const LOGIN_USER = gql`  
  mutación loginUser($nombredeusuario: String!, $contraseña: String!) { loginUser(nombredeusuario:  
    $nombredeusuario, contraseña: $contraseña)  
  }
```

2. Edite src/pages/Login.jsx y reemplace las importaciones a TanStack React Query y la función de inicio de sesión con lo siguiente:

```
importar { useMutation como useGraphQLMutation } desde '@apollo/ client/react/index.js'  
importar { LOGIN_USER } desde  
'./api/graphql/users.js'
```

3. Actualiza también el gancho:

```
const [loginUser, { cargando }] = useGraphQLMutation(LOGIN_  
USUARIO,  
  { variables: { nombre de usuario, contraseña },  
  onCompleted: (datos) => {  
    setToken(data.loginUser) browse('/'),  
    onError: () =>  
  
    alert('¡Error al iniciar sesión!'), })
```

4. Actualice la función handleSubmit:

```
constante handleSubmit = (e) => {
    e.preventDefault()
    loginUser()
}
```

5. Finalmente, actualiza el botón de envío:

```
<entrada
    tipo='enviar'
    valor={cargando ? 'Iniciando sesión...' : 'Iniciar sesión'}
    deshabilitado={!usuario || !contraseña || cargando}
/>
```

Ahora que el registro y el inicio de sesión utilizan mutaciones GraphQL, pasemos a migrar la funcionalidad de creación de publicaciones a GraphQL.

## Migración de la publicación creada a GraphQL

La funcionalidad de crear publicaciones es un poco más complicada de implementar, ya que requiere que iniciemos sesión (lo que significa que debemos enviar el encabezado JWT) e invalidar las consultas de la lista de publicaciones, para que la lista se actualice después de crear una nueva publicación.

Ahora comencemos a implementar esto con Apollo Client:

1. Primero, definamos la mutación. Edite src/api/graphql/posts.js y agregue el siguiente código:

```
exportar const CREATE_POST = gql`  
    mutación createPost($título: Cadena!, $contenido: Cadena, $etiquetas: [!Cadena!]!) {  
        createPost(título: $título, contenido: $contenido, etiquetas: $etiquetas)  
    }  
    identificación  
    título  
    }  
`
```

Para esta mutación, usaremos la respuesta para obtener el ID y el título de la publicación creada. Usaremos estos datos para mostrar un enlace a la publicación una vez creada correctamente.

2. Luego, edita src/components/CreatePost.jsx y reemplaza la importación de TanStack React Query con una importación del gancho de mutación:

```
importar { useMutation como useGraphQLMutation } desde '@apollo/  
cliente/react/index.js'
```

3. Además, importe el componente Enlace y la función slug para mostrar un enlace a la publicación creada:

```
importar { Enlace } desde 'react-router-dom'  
importar slug desde 'slug'
```

4. Reemplace la importación de la función createPost con las importaciones de la mutación CREATE\_POST y las consultas GET\_POSTS y GET\_POSTS\_BY\_AUTHOR. Usaremos estas definiciones de consulta para que Apollo Client las recupere posteriormente:

```
importar {  
    CREAR_POST,  
    OBTENER_PUBLICACIONES,  
    OBTENER_PUBLICACIONES_POR_AUTOR,  
} de './api/graphql/posts.js'
```

5. Reemplace el cliente de consulta existente y los ganchos de mutación con la siguiente mutación GraphQL, donde pasamos las variables de título y contenido:

```
const [createPost, { cargando, datos }] =  
useGraphQLMutation(CREATE_POST, {  
    variables: { título, contenido },
```

6. A continuación, proporcionamos el encabezado JWT como contexto para la mutación:

```
contexto: { encabezados: { Autorización: `Portador ${token}` } },
```

7. También proporcionamos una opción refetchQueries a la mutación, indicándole al Cliente Apollo que vuelva a buscar ciertas consultas después de que se llamó a la mutación:

```
    Consultas de recuperación: [ OBTENER_PUBLICACIONES, OBTENER_PUBLICACIONES_POR_AUTOR ],
```

```
})
```

#### Nota

Dado que la recuperación tras una mutación es una operación común, Apollo Client ofrece una forma sencilla de hacerlo en el gancho de mutación. Simplemente pase todas las consultas que deban recuperarse allí y Apollo Client se encargará de ello.

8. Ajuste la función handleSubmit:

```
constante handleSubmit = (e) => {  
    e.preventDefault()  
    crearPost()  
}
```

9. Ajuste el botón de envío:

```
<entrada
  tipo='enviar'
  valor={cargando ? 'Creando...' : 'Crear'}
  deshabilitado={!title || cargando}
/>
```

10. Por último, vamos a cambiar el mensaje de éxito, mostrando un enlace a la publicación creada:

```
{datos?.createPost ?
  <>
    <br />
    Correo{' '}
    <Enlace
      a={`/posts/${data.createPost.id}/${slug(data.
        crearPost.title)}`}
    >
      {datos.createPost.título}
    </Link>' '
    ¡Creado exitosamente!
  </>
) : nulo}
```

Gracias al funcionamiento de los tipos y los resolutores en GraphQL, podemos acceder fácilmente a los campos del resultado de una mutación, como si obtuviéramos una sola publicación. Por ejemplo, ¡podríamos incluso indicarle a GraphQL que obtenga aquí el nombre de usuario del autor de la publicación creada!

11. Intenta crear una nueva publicación y verás que el mensaje de éxito ahora muestra un enlace a la publicación creada y la lista de publicaciones se recupera automáticamente para nosotros.

La siguiente captura de pantalla muestra una nueva publicación que se creó con éxito, mostrando el enlace a la nueva publicación en el mensaje de éxito y la nueva publicación en la lista de publicaciones (recuperada automáticamente por Apollo Client):

Title:

This post was created using  
GraphQL mutations!

Post [New Post](#) created successfully!

---

Filter by:  
author:

Sort By:  / Sort Order:

---

[New Post](#)

*Written by testuser*

Figura 12.1: Creación de una publicación mediante mutaciones GraphQL, con una lista de publicaciones que se vuelve a obtener

Ahora que hemos implementado con éxito la creación de publicaciones con GraphQL, nuestra aplicación de blog está completamente conectada a nuestro servidor GraphQL.

Hay muchos más conceptos avanzados en GraphQL que aún no hemos cubierto en este libro, como la recuperación avanzada, las suscripciones (obtener actualizaciones en tiempo real del servidor GraphQL), el manejo de errores, la suspensión, la paginación y el almacenamiento en caché. Los capítulos de GraphQL en este libro solo sirven como una introducción a GraphQL.

Si desea obtener más información sobre GraphQL y Apollo, le recomiendo consultar la extensa documentación de Apollo (<https://www.apollographql.com/docs/>), que contiene información detallada y ejemplos prácticos sobre el uso de Apollo Server y Apollo Client.

## Resumen

En este capítulo, conectamos nuestro backend GraphQL previamente creado con el frontend mediante Apollo Client. Comenzamos configurando Apollo Client y realizando una consulta GraphQL para obtener todas las publicaciones. Además, mejoramos el rendimiento de la lista de publicaciones al obtener los nombres de usuario de los autores en una sola solicitud, aprovechando así la potencia de GraphQL.

A continuación, introdujimos variables en nuestra consulta y reimplantamos la ordenación y el filtrado por autor. También introdujimos fragmentos en nuestras consultas para reutilizar los mismos campos. Por último, implementamos mutaciones de GraphQL en el frontend para registrarse, iniciar sesión y crear publicaciones. También aprendimos a recuperar consultas en Apollo Client y mencionamos brevemente conceptos avanzados de GraphQL y Apollo.

En el próximo capítulo, Capítulo 13, Construcción de un backend basado en una arquitectura basada en eventos usando Express y Socket.IO, nos alejaremos de las arquitecturas full-stack tradicionales y construiremos una nueva aplicación usando un tipo especial de arquitectura full-stack: una aplicación basada en eventos.

## Parte 4:

# Explorando un evento basado en Arquitectura de pila completa

En esta parte del libro, nos alejaremos de las arquitecturas full-stack tradicionales y exploraremos un tipo especial de arquitectura full-stack: las aplicaciones basadas en eventos. Ejemplos de aplicaciones basadas en eventos son las que gestionan datos en tiempo real, como las aplicaciones colaborativas (p. ej., Google Docs, pizarra digital, etc.) o las aplicaciones financieras (p. ej., la plataforma de intercambio de criptomonedas Kraken). Primero, desarrollaremos un backend basado en eventos con Express y Socket.IO. Luego, crearemos un frontend para consumir y enviar eventos. Por último, añadiremos persistencia y funcionalidad para reproducir eventos en nuestra aplicación mediante MongoDB.

Esta parte incluye los siguientes capítulos:

- Capítulo 13, Creación de un backend basado en eventos utilizando Express y Socket.IO
- Capítulo 14, Creación de una interfaz para consumir y enviar eventos
- Capítulo 15, Cómo agregar persistencia a Socket.IO usando MongoDB



# 13

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

En este capítulo, aprenderemos sobre las aplicaciones basadas en eventos y las ventajas de usar esta arquitectura en comparación con una más tradicional. A continuación, aprenderemos sobre WebSockets y su funcionamiento. Posteriormente, implementaremos un backend con Socket.IO y Express. Finalmente, aprenderemos a integrar la autenticación mediante JWT con Socket.IO.

En este capítulo cubriremos los siguientes temas principales:

- ¿Qué son las aplicaciones basadas en eventos?
- Configuración de Socket.IO
- Creación de un backend para una aplicación de chat usando Socket.IO
- Agregar autenticación mediante la integración de JWT con Socket.IO

### 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 los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-modernos-de-React-Full-Stack/árbol/principal/capítulo-13>.

Si clonó el repositorio completo de este 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/kHGvkopIHf4>.

## ¿Qué son las aplicaciones basadas en eventos?

A diferencia de las aplicaciones web tradicionales, donde tenemos un patrón de solicitud-respuesta, en las aplicaciones basadas en eventos, estamos tratando con eventos. El servidor y el cliente permanecen conectados y cada lado puede enviar eventos, que el otro lado escucha y reacciona a ellos.

El siguiente diagrama muestra la diferencia entre implementar una aplicación de chat en un patrón de solicitud-respuesta versus un patrón basado en eventos:

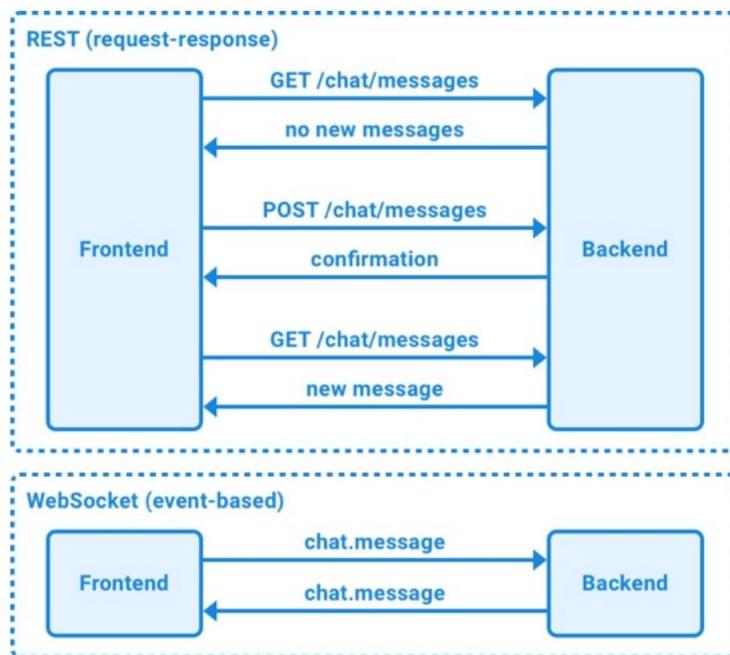


Figura 13.1 – Implementación de una aplicación de chat con patrones basados en solicitud-respuesta y eventos

Por ejemplo, para implementar una aplicación de chat con un patrón de solicitud-respuesta, necesitaríamos enviar periódicamente una solicitud a un endpoint GET /chat/messages para actualizar la lista de mensajes enviados en una sala de chat. Este proceso de envío periódico de solicitudes se denomina sondeo breve. Para enviar un mensaje de chat, haríamos una solicitud a POST /chat/messages. En un patrón basado en eventos, podríamos enviar un evento "chat.message" del cliente al servidor, que a su vez envía un evento "chat.message" a todos los usuarios conectados. Los clientes escuchan los eventos "chat.message" y muestran los mensajes a medida que llegan; ¡no se requieren solicitudes periódicas!

Por supuesto, cada patrón tiene sus ventajas y desventajas:

- REST/solicitud-respuesta:

Es útil cuando los datos no cambian con frecuencia.

Las respuestas se pueden almacenar en caché fácilmente

Las solicitudes no tienen estado, lo que facilita el escalado de los backends.

Malas actualizaciones en tiempo real (requiere sondeo periódico)

Más gastos generales por solicitud (malo cuando se envían muchas respuestas cortas)

- WebSockets/basados en eventos:

Bueno para aplicaciones que requieren actualizaciones frecuentes.

Más eficiente porque una conexión persistente entre el cliente y el servidor se reutiliza para múltiples solicitudes

Menos gastos generales por solicitud

Podría haber problemas de conexión con los servidores proxy (corporativos)

Tienen estado, lo que puede dificultar la escalabilidad de una aplicación.

Como podemos ver, para obtener datos que no cambian con frecuencia (y que pueden almacenarse en caché), como entradas de blog, es más adecuado un patrón de solicitud-respuesta. Para aplicaciones donde los datos cambian con frecuencia, como una sala de chat, es más adecuado un patrón basado en eventos.

## ¿Qué son los WebSockets?

La API de WebSockets es una función del navegador que permite a las aplicaciones web crear una conexión abierta entre el cliente y el servidor, similar a los sockets de estilo Unix. Con WebSockets, la comunicación... puede suceder en ambas direcciones al mismo tiempo. Esto contrasta con las solicitudes HTTP, donde ambas partes pueden comunicarse, pero no simultáneamente.

Los WebSockets utilizan HTTP para establecer una conexión entre el cliente y el servidor, y luego actualizan el protocolo de HTTP a WebSocket. Si bien tanto HTTP como WebSockets dependen del Protocolo de Control de Transmisión (TCP), son protocolos distintos en la capa de aplicación (Capa 7) del modelo de Interconexión de Sistemas Abiertos (OSI) .

Se establece una conexión a un WebSocket enviando una solicitud HTTP con el encabezado Upgrade: websocket y otros parámetros para establecer una conexión WebSocket segura. Luego, el servidor responde con un código de respuesta HTTP 101 Switching Protocols e información para establecer la conexión. Luego, el cliente y el servidor continúan hablando sobre el protocolo WebSocket.

### ¿Qué es Socket.IO?

Socket.IO es una implementación de una biblioteca de servidor y cliente basada en eventos. En la mayoría de los casos, establece una conexión con el servidor mediante un WebSocket. Si no es posible establecer una conexión WebSocket (debido a la falta de compatibilidad con el navegador o la configuración del firewall), Socket.IO también puede recurrir al sondeo HTTP prolongado. Sin embargo, Socket.IO no es una implementación WebSocket pura, ya que añade metadatos adicionales a cada paquete. Solo utiliza WebSockets internamente para transmitir datos.

Además de proporcionar una manera de enviar eventos entre el cliente y el servidor, Socket.IO ofrece las siguientes características sobre los WebSockets simples:

- Retorno al sondeo largo de HTTP: \$is ocurre si no se puede establecer la conexión WebSocket . \$is puede ser especialmente útil para empresas que utilizan servidores proxy o firewalls que bloquean las conexiones WebSocket.
- Reconexión automática: si se interrumpe la conexión WebSocket.
- Almacenamiento en búfer de paquetes: cuando el cliente se desconecta, los paquetes se pueden volver a enviar de forma automática al volver a conectarse.
- Agradecimientos: una forma conveniente de enviar eventos en un patrón de solicitud-respuesta, que puede a veces ser útil incluso en aplicaciones basadas en eventos.
- Transmisión: envío de un evento a todos (o a un subconjunto de todos) los clientes conectados.
- Multiplexación: Socket.IO implementa espacios de nombres, que pueden usarse para crear "canales" a los que solo ciertos usuarios pueden enviar y recibir eventos, como un "canal solo para administradores".

Ahora que hemos aprendido los conceptos básicos de qué es Socket.IO, profundicemos en cómo funcionan la conexión y la emisión/recepción de eventos.

## Conexión a Socket.IO

El siguiente diagrama muestra cómo se establece una conexión con Socket.IO:



Figura 13.2 – Establecer una conexión con Socket.IO

En primer lugar, Socket.IO envía un protocolo de enlace desde el cliente (en el frontend) al servidor (en el backend), que puede contener información para autenticarse con el servidor o parámetros de consulta para proporcionar información adicional al establecer la conexión.

Si no se puede establecer una conexión mediante WebSockets, Socket.IO se conecta al servidor mediante sondeo HTTP largo, lo que implica realizar una solicitud al servidor que se mantiene activa hasta que se produce un evento, momento en el que el servidor envía una respuesta. \$is permite esperar eventos sin tener que realizar una solicitud periódica para comprobar si hay nuevos eventos. Claro que esto no es tan eficiente como WebSockets, pero es una buena alternativa cuando WebSockets no está disponible.

## Emisión y recepción de eventos

Una vez conectados a Socket.IO, podemos empezar a emitir (enviar) y recibir eventos. Los eventos se gestionan mediante el registro de funciones de gestión de eventos, que se invocan cuando el cliente o el servidor reciben un determinado tipo de evento . Tanto el cliente como el servidor pueden emitir y recibir eventos. Además, los eventos se pueden transmitir desde el servidor a varios clientes. El siguiente diagrama muestra un ejemplo de cómo se emiten y reciben eventos en una aplicación de chat:



Figura 13.3 – Emisión y recepción de eventos con Socket.IO

Como podemos ver, el Usuario 1 envía un mensaje de "Hola a todos" , que el servidor (backend) transmite a todos los demás clientes (frontend). En este caso, el mensaje se transmite de vuelta tanto al Usuario 1 como al Usuario 2.

Si queremos restringir los clientes que reciben determinados eventos, Socket.IO permite la creación de salas. Los clientes pueden unirse a una sala y, en el servidor, también podemos transmitir eventos solo a salas específicas. Este concepto se puede utilizar para salas de chat, pero también para colaborar en un proyecto específico (como editar un documento en conjunto en tiempo real).

Además de emitir y recibir eventos de forma asíncrona, Socket.IO ofrece una forma de enviar un evento que espera una respuesta mediante acuses de recibo. Estos pueden usarse para modelar patrones de solicitud-respuesta en Socket.IO. Por ejemplo, podríamos solicitar información sobre un usuario específico mediante un archivo user.info. evento y esperar sincrónicamente la respuesta del servidor (acuse de recibo). Podemos observar esto en el diagrama anterior, donde el Usuario 2 solicita información sobre un usuario específico y recibe una respuesta con la información del usuario.

Ahora que hemos aprendido sobre aplicaciones basadas en eventos, WebSockets y Socket.IO, pongamos esta teoría en práctica y configuremos Socket.IO.

## Configuración de Socket.IO

Para configurar el servidor Socket.IO, basaremos nuestro código en lo que vimos en el Capítulo 6, "Añadir autenticación y roles con JWT", ya que incluye un código estándar para un backend y un frontend con autenticación JWT. Más adelante en este capítulo, en la sección "Añadir autenticación mediante la integración de JWT con Socket.IO", usaremos JWT para añadir autenticación a Socket.IO:

1. Copie la carpeta ch6 existente a una nueva carpeta ch13, de la siguiente manera:

```
$ cp -R ch6 ch13
```

2. Abra la carpeta ch13 en VS Code.

3. Ahora podemos empezar a configurar Socket.IO. Primero, instale el paquete socket.io en la carpeta del backend ejecutando los siguientes comandos:

```
$ cd backend/  
$ npm install socket.io@4.7.2
```

4. Edite backend/.env y cambie DATABASE\_URL para que apunte a una nueva base de datos de chat:

```
URL_DE_BASE_DE_DATOS=mongodb://localhost:27017/chat
```

5. Edite backend/src/app.js e importe la función createServer desde node:http  
y la función Servidor de socket.io:

```
importar { createServer } desde 'node:http'  
importar {Servidor} desde 'socket.io'
```

Necesitaremos crear un servidor node:http, ya que no podemos conectar Socket.IO directamente a Express. En su lugar, Socket.IO se conecta a un servidor node:http.

6. Afortunadamente, Express también se puede conectar fácilmente a un servidor node:http. Editar backend/src/app.js y, antes de exportar la aplicación, cree un nuevo servidor node:http desde la aplicación Express, de la siguiente manera:

```
const servidor = createServer(app)
```

7. Ahora, cree un nuevo servidor Socket.IO desde el servidor node:http:

```
const io = nuevo Servidor(servidor, {  
    cors: {  
        origen: '**',  
    },  
})
```

**Advertencia**

Al establecer el origen en \*, los sitios de phishing pueden imitar su sitio web y enviar solicitudes a su backend. En producción, el origen debe establecerse en la URL implementada de su frontend.

8. Podemos usar el servidor Socket.IO para escuchar las conexiones de los clientes e imprimir un mensaje:

```
io.on('conexión', (socket) => {
    console.log('usuario conectado:', socket.id)
```

9. La conexión de cliente activa se puede rastrear mediante el objeto socket. Por ejemplo, podemos Escuche los eventos de desconexión del cliente de la siguiente manera:

```
socket.on('desconectar', () => {
    console.log('usuario desconectado:', socket.id)
})
```

10. Por último, cambie la exportación para que utilice el servidor node:http en lugar de la aplicación Express directamente:

```
exportar { servidor como aplicación }
```

11. Inicie el backend ejecutando los siguientes comandos:

```
$ cd backend/
$ npm ejecuta dev
```

No olvides poner en funcionamiento Docker y el contenedor de la base de datos antes de iniciar el backend. Mantenga el backend en funcionamiento durante el resto de este capítulo.

Ahora que hemos configurado un servidor Socket.IO simple, continuemos configurando el cliente.

## Configuración de un cliente Socket.IO simple

Por ahora, usaremos el frontend existente. En el siguiente capítulo, Capítulo 14, "Creación de un frontend para consumir y enviar eventos", eliminaremos los componentes del blog y crearemos un nuevo frontend de React para nuestra aplicación de chat.

Comencemos configurando un cliente Socket.IO simple:

1. En la raíz del proyecto, instale el paquete socket.io-client para el frontend mediante ejecutando el siguiente comando:

```
$ npm install socket.io-client@4.7.2
```

¡Asegúrate de que ya no estás en la carpeta backend!

2. Edite src/App.jsx e importe la función io desde socket.io-client:

```
importar { io } desde 'socket.io-client'
```

3. Defina una nueva instancia del cliente Socket.IO utilizando la función io y pasando un nombre de host y puerto:

```
constante socket = io(import.meta.env.VITE_SOCKET_HOST)
```

Aquí, pasaremos localhost:3001 mediante una variable de entorno. No podemos pasar la URL HTTP, ya que Socket.IO intentará conectarse al nombre de host y al puerto mediante WebSockets.

4. Escuche el evento de conexión e imprima un mensaje si nos conectamos exitosamente al Servidor Socket.IO:

```
socket.on('conectar', () => {
    console.log('conectado a socket.io como', socket.id)
})
```

5. Además, escuche el evento connect\_error y registre un mensaje de error en caso de que se conecte el servidor Socket.IO falló:

```
socket.on('error_de_conexión', (err) => {
    console.error('Error de conexión de socket.io:', err)
})
```

6. Edite .env y agregue la siguiente variable de entorno:

```
VITE_SOCKET_HOST="localhost:3001"
```

7. Ejecute el frontend de la siguiente manera:

```
$ npm ejecuta dev
```

8. Ahora, abra la interfaz en su navegador yendo a <http://localhost:5173/>. Mantenga la interfaz se ejecutará durante el resto de este capítulo.

Verá un mensaje indicando que está conectado a socket.io en la consola del navegador. En la salida del servidor, verá que el cliente se conectó correctamente. Intente actualizar la página para ver que se desconecta y se conecta de nuevo (con un nuevo ID de socket).

```
successfully connected to database: mongodb://localhost:27017/ch3
express server running on http://localhost:3001
user connected: 8c0lseEPbgQ8d4AzAAAB
user disconnected: 8c0lseEPbgQ8d4AzAAAB
user connected: rmGIXt3HV8S1qusRAAAD
```

Figura 13.4 – Ver el cliente Socket.IO conectarse y desconectarse de nuestro servidor

Ahora que hemos configurado exitosamente un servidor Socket.IO, continuemos creando un backend para una aplicación de chat usando Socket.IO.

## Creación de un backend para una aplicación de chat usando Socket.IO

Ahora podemos empezar a implementar una aplicación de chat con Socket.IO. Desarrollaremos la siguiente funcionalidad para nuestra aplicación de chat:

- Emitir eventos para enviar mensajes de chat desde el cliente al servidor
- Transmitir mensajes de chat desde el servidor a todos los clientes
- Unirse a salas para enviar mensajes en
- Usar reconocimientos para obtener información sobre un usuario

¡Comencemos!

Emitir eventos para enviar mensajes de chat desde el cliente al servidor

Comenzaremos emitiendo un evento `chat.message` del cliente al servidor. Por ahora, emitiremos este evento justo después de conectarnos. Más adelante, lo integraremos en un frontend. Sigue estos pasos para enviar mensajes de chat desde el cliente y recibirlas en el servidor:

1. Edite `backend/src/app.js` y corte/elimine el siguiente código:

```
io.on('conexión', (socket) => { console.log('usuario  
conectado:', socket.id) socket.on('desconectar', () =>  
{ console.log('usuario desconectado:',  
socket.id) }) })
```

2. Cree un nuevo archivo `backend/src/socket.js`, defina allí una función `handleSocket` y pegue el siguiente código dentro de él:

```
función de exportación handleSocket(io)  
{ io.on('conexión', (socket) => { console.log('usuario  
conectado:', socket.id) socket.on('desconectar', () => {  
  
console.log('usuario desconectado:', socket.id) }) })
```

3. Ahora, agregue un nuevo oyente que escuche el evento `chat.message` y registre el mensaje enviado.  
Del cliente:

```
socket.on('chat.message', (mensaje) => { console.log(`$  
{socket.id}: ${mensaje}`) })
```

```
    })  
}
```

4. Edite backend/src/app.js e importe la función handleSocket:

```
importar { handleSocket } desde './socket.js'
```

5. Una vez creado el servidor Socket.IO, llame a la función handleSocket:

```
const io = new Server(servidor, { cors: { origen: '**', }, })  
  
handleSocket(io)
```

6. Edite src/App.jsx y emita un evento chat.message con algún texto, de la siguiente manera:

```
socket.on('conectar', () => {  
  console.log('conectado a socket.io como', socket.id) socket.emit('chat.message',  
  'hola del cliente') })
```

Información

Socket.IO nos permite enviar cualquier tipo de estructura de datos serializable en un evento, ¡no solo cadenas! Por ejemplo, es posible enviar objetos y matrices.

El backend y el frontend deberían actualizarse automáticamente y el servidor registrará el siguiente mensaje:

```
XXmWHjA_5zew70VIAAM: Hola del cliente
```

De lo contrario, asegúrese de (re)iniciar el backend y el frontend y actualizar la página manualmente.

Como puedes ver, es bastante sencillo enviar y recibir eventos de forma asíncrona en tiempo real utilizando Socket.IO.

## Transmitir mensajes de chat desde el servidor a todos los clientes

Ahora que el servidor backend puede recibir mensajes de un cliente, necesitamos difundirlos a todos los demás clientes para que puedan ver los mensajes de chat enviados. Hagamos esto ahora:

1. Edite backend/src/socket.js y amplíe el detector de eventos chat.message para que llama a io.emit y envía el mensaje de chat a todos:

```
socket.on('chat.mensaje', (mensaje) => { console.log('$  
  {socket.id}: ${mensaje}') io.emit('chat.mensaje', {
```

```

        nombre de usuario: socket.id,
        mensaje,
    })
}
})

```

**Nota**

Alternativamente, puede utilizar socket.broadcast.emit para enviar un evento a todos los clientes excepto al que envió el mensaje.

2. También necesitamos agregar un detector de mensajes de chat en el lado del cliente. \$is funciona igual que en el servidor. Edite src/App.jsx y agregue el siguiente detector de eventos:

```

socket.on('chat.mensaje', (msg) => {
    console.log(`${msg.nombreusuario}: ${msg.mensaje}`)
})

```

3. Ahora debería ver el mensaje registrado en el servidor y el cliente. Intente abrir una segunda ventana; verá los mensajes de ambos clientes en su navegador.

<code>connected to socket.io as ewy3yhSnl_59fGhbAAAJ</code>	<a href="#">App.jsx:13</a>
<code>ewy3yhSnl_59fGhbAAAJ: hello from client</code>	<a href="#">App.jsx:17</a>
<code>mYE_P9IrsqF0mVzaAAAL: hello from client</code>	<a href="#">App.jsx:17</a>

Figura 13.5 – Recepción de mensajes de otro cliente

## Unirse a salas para enviar mensajes en

Si bien es útil tener un chat funcional donde los mensajes se transmiten a todos, a menudo no queremos difundir nuestros mensajes a todo el mundo. En su lugar, podríamos querer enviar mensajes solo a un grupo específico. Para facilitar esto, Socket.IO ofrece salas. Las salas permiten agrupar clientes para que los eventos solo se envíen a todos los demás clientes de la sala. Esta función permite crear salas de chat, pero también colaborar en un proyecto (creando una nueva sala para cada proyecto). Veamos cómo se pueden usar las salas en Socket.IO:

1. Socket.IO nos permite pasar una cadena de consulta durante el protocolo de enlace. Podemos acceder a esta cadena para obtener la sala a la que el cliente desea unirse. Edite backend/src/socket.js y obtenga la sala a partir de la consulta del protocolo de enlace:

```

io.on('conexión', (socket) => {
    console.log('usuario conectado:', socket.id)
    const habitación = socket.handshake.query?.habitación ?? 'público'
}
)

```

2. Ahora, use socket.join para unir al cliente a la sala seleccionada:

```
socket.join(habitación)
console.log(socket.id, 'sala unida:', sala)
```

3. \$en, dentro del controlador chat.message, use .to(room) para asegurarse de que los mensajes de chat de ese cliente solo se envíen a una determinada sala:

```
io.to(sala).emit('chat.mensaje', {
    nombre de usuario: socket.id,
    mensaje,
})
```

4. En el cliente, necesitamos pasar una cadena de consulta para indicarle al servidor a qué sala nos gustaría unirnos.

Edite src/App.jsx de la siguiente manera:

```
constante socket = io(import.meta.env.VITE_SOCKET_HOST, {
    consulta: ventana.ubicación.búsqueda.subcadena(1),
})
```

La cadena de consulta \$e de Socket.IO es una cadena de consulta de URL, por lo que podemos simplemente pasársela la cadena de consulta de la página actual (sin ? al comienzo de la cadena).

5. Abra <http://localhost:5173/> y <http://localhost:5173/?room=test>

En dos ventanas del navegador independientes, envía mensajes desde ambas. Verás que el mensaje de la segunda ventana no se envía a la primera. Sin embargo, si abres otra ventana con la cadena de consulta ?room=test y envías un mensaje desde allí, verás que el mensaje se reenvía a la segunda ventana (pero no a la primera).

Como podemos ver, podemos usar las salas para tener un control más preciso sobre qué clientes reciben ciertos eventos. Dado que el servidor controla a qué salas se une un cliente, también podemos añadir comprobaciones de permisos antes de permitir que un cliente se una a una sala.

## Uso de reconocimientos para obtener información sobre un usuario

Como hemos visto, los eventos son una excelente manera de enviar mensajes asíncronos. Sin embargo, a veces necesitamos una API de solicitud-respuesta síncrona más tradicional, como la que teníamos con REST. En Socket.IO, podemos implementar eventos síncronos mediante acuses de recibo. Podemos usar acuses de recibo para, por ejemplo, obtener más información sobre un usuario en la sala de chat actual. Por ahora, solo devolveremos las salas en las que se encuentra el usuario. Más adelante, al añadir la autenticación, obtendremos el objeto de usuario de la base de datos. Comencemos a implementar los acuses de recibo:

1. Edite backend/src/socket.js y defina un nuevo detector de eventos:

```
socket.on('usuario.info', async (socketId, devolución de llamada) => {
```

Observe cómo pasamos una función de devolución de llamada como último argumento. \$is es lo que hace que el evento sea un reconocimiento.

2. En este detector de eventos, vamos a buscar todos los sockets en la sala con el ID de nuestro socket:

```
const sockets = await io.in(socketId).fetchSockets()
```

Internamente, Socket.IO crea una sala para cada socket conectado, para permitir el envío de eventos a un solo socket.

#### Nota

Podríamos acceder directamente a los sockets de la instancia actual, pero esto ya no funcionaría al escalar nuestro servicio a varias instancias en un clúster. Para que funcione incluso en un clúster, necesitamos usar la función de sala para obtener un socket por ID.

3. Ahora, debemos comprobar si encontramos un socket con el ID dado. De no ser así, devolvemos null:

```
si (sockets.length === 0) devolver callback(null)
```

4. De lo contrario, devolvemos el ID del socket y una lista de salas en las que se encuentra el usuario:

```
constante socket = sockets[0]
constante userInfo = { socketId,
    habitaciones: Array.from(socket.habitaciones),
} devolver devolución de llamada (userInfo)
})
```

5. Ahora podemos emitir el evento user.info en el cliente. Edite src/App.jsx y comience por convertir el detector de eventos de conexión en una función asíncrona:

```
socket.on('conectar', async () => {
    console.log('conectado a socket.io como', socket.id) socket.emit('chat.message', 'hola
    del cliente')
```

6. Para emitir un evento con un reconocimiento, podemos usar la función emitWithAck, que devuelve una Promesa que podemos esperar:

```
const userInfo = await socket.emitWithAck('usuario.info', socket.id)

console.log('información del usuario', userInfo })
```

7. Después de guardar el código, vaya a la ventana del navegador; verá la información del usuario.

Inició sesión en la consola:

```
connected to socket.io as 3i1MTX28X7H8xtXYAAAB
3i1MTX28X7H8xtXYAAAB: hello from client
user info
  ▼ {socketId: '3i1MTX28X7H8xtXYAAAB', rooms: Array(2)} ⓘ
    ► rooms: (2) ['3i1MTX28X7H8xtXYAAAB', 'public']
    socketId: "3i1MTX28X7H8xtXYAAAB"
```

Figura 13.6 – Obtención de información del usuario con un acuse de recibo

Ahora que hemos aprendido cómo enviar varios tipos de eventos, pasemos a un tema más avanzado: la autenticación con Socket.IO.

## Agregar autenticación mediante la integración de JWT con Socket.IO

Hasta ahora, todos los mensajes de chat se han enviado con el ID del socket como "nombre de usuario". Esto no es una buena manera de identificar a los usuarios en una sala de chat. Para solucionar esto, vamos a introducir cuentas de usuario mediante la autenticación de sockets con JWT. Siga estos pasos para implementar JWT con Socket.IO:

1. Edite backend/src/socket.js e importe jwt desde el paquete jsonwebtoken y

getUserInfoById de nuestras funciones de servicio:

```
importar jwt desde 'jsonwebtoken'
importar { getUserInfoById } desde './services/users.js'
```

2. Dentro de la función handleSocket, defina un nuevo middleware Socket.IO utilizando io.use().

El middleware en Socket.IO funciona de manera similar al middleware en Express: definimos una función que se ejecuta antes de que se procesen las solicitudes, de la siguiente manera:

```
función de exportación handleSocket(io) {
  io.use((socket, siguiente) => {
```

3. Dentro de esta función, verificamos si el token se envió mediante el objeto auth (similar a cómo pasamos la sala anteriormente mediante la cadena de consulta). Si no se pasó ningún token, enviamos un error a la función next() y provocamos un fallo en la conexión:

```
    si (!socket.handshake.auth?.token) {
      devolver siguiente(nuevo Error('Error de autenticación: no hay token')
      proporcionó))
    }
```

**Nota**

Es importante no pasar un JWT a través de la cadena de consulta, ya que esta forma parte de la URL. Se expone en la barra de direcciones del navegador y, por lo tanto, podría almacenarse en el historial, donde un posible atacante podría extraerlo. En su lugar, el objeto de autenticación se envía a través de la carga útil de la solicitud durante el protocolo de enlace, que no se expone en la barra de direcciones.

4. De lo contrario, llamamos a `jwt.verify` para verificar el token utilizando el `JWT_SECRET` existente variable de entorno:

```
jwt.verify(
  socket.handshake.auth.token,
  proceso.env.JWT_SECRET,
```

5. Si el token no es válido, devolvemos nuevamente un error en la función `next()`:

```
async (err, token decodificado) => {
  si (err) {
    devolver siguiente(nuevo Error('Error de autenticación: no válido
simbólico'))
  }
}
```

6. De lo contrario, guardamos el token decodificado en `socket.auth`:

```
socket.auth = token decodificado
```

7. Además, obtenemos la información del usuario de la base de datos y, para mayor comodidad, la almacenamos en `socket.user`:

```
socket.usuario = await getUserInfoById(socket.auth.sub)
devolver siguiente()
},
)
})
```

**Nota**

Asegúrese de que `next()` siempre se llame en el middleware de Socket.IO. De lo contrario, Socket.IO mantendrá la conexión abierta hasta que se cierre tras un tiempo de espera determinado.

El objeto de usuario `$e` contiene un valor de nombre de usuario. Ahora, podemos reemplazar el ID del socket en el mensaje de chat con el nombre de usuario:

```
socket.on('chat.message', (mensaje) => {
  console.log(`#${socket.id}: ${mensaje}`)
  io.to(sala).emit('chat.mensaje', {
    nombre de usuario: socket.usuario.nombredeusuario,
```

```

        mensaje,
    })
}
}
```

9. También podemos devolver la información del usuario desde el evento user.info:

```

constante userInfo = {
    ID de socket,
    habitaciones: Array.from(socket.habitaciones),
    usuario: socket.usuario,
}
}
```

10. Aún necesitamos enviar el objeto de autenticación desde el lado del cliente, editar src/App.jsx y obtener el token de localStorage, de la siguiente manera:

```

constante socket = io(import.meta.env.VITE_SOCKET_HOST, {
    consulta: ventana.ubicación.búsqueda.subcadena(1),
    aut: {
        token: ventana.localStorage.getItem('token'),
    },
})
}
```

#### Nota

Para simplificar, en este ejemplo almacenamos y leemos el JWT en localStorage. Sin embargo, no es recomendable almacenar un JWT de esta forma en producción, ya que un atacante podría leer localStorage si encuentra la forma de inyectar JavaScript. Una mejor manera de almacenar un JWT sería usar una cookie con los atributos Secure, HttpOnly y SameSite="Strict".

11. Ahora que el servidor está configurado, podemos intentar iniciar sesión en el cliente. Inicialmente, vamos a... ver un mensaje de error:

```

✖ ► socket.io connect error: Error: Authentication failed: no App.jsx:24
      token provided
      at Socket2.onpacket (socket.js:466:29)
      at Emitter.emit (index.mjs:136:20)
      at manager.js:204:18

```

Figura 13.7 – Un mensaje de error de Socket.IO porque no se proporcionó JWT

12. Para obtener un token, podemos registrarnos e iniciar sesión normalmente usando la interfaz del blog. En la pestaña Red del inspector, podemos encontrar la solicitud /login con un token en la respuesta:

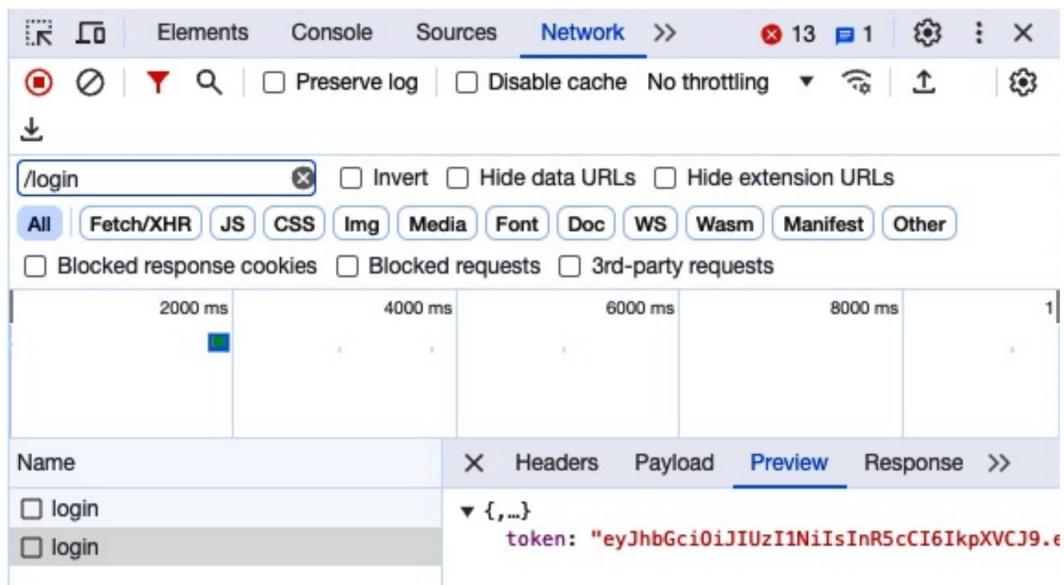


Figura 13.8 – Copia del JWT desde la pestaña Red

13. Copie este token y agréguelo a localStorage ejecutando localStorage.

`setItem('token', '<JWT>')` en la consola del navegador (reemplazando <JWT> con el token copiado). Al actualizar la página, ¡debería funcionar! Como podemos ver, al iniciar sesión con dos usuarios diferentes, podemos ver sus mensajes con sus respectivos nombres de usuario:

```
connected to socket.io as BVas04Ss1_UsnVY3AAAN           App.jsx:17
test: hello from client                                App.jsx:28
user info                                              App.jsx:20
▶ {socketId: 'BVas04Ss1_UsnVY3AAAN', rooms: Array(2), user: {...}}
daniel: hello from client                             App.jsx:28
```

Figura 13.9 – Recepción de mensajes de diferentes usuarios

¡Nuestro backend de chat ya está completamente funcional! En el siguiente capítulo, crearemos un frontend para completar nuestra aplicación de chat.

## Resumen

En este capítulo, aprendimos sobre aplicaciones basadas en eventos, WebSockets y Socket.IO. Luego, configuramos Socket.IO en el backend (servidor) y el frontend (cliente). Posteriormente, aprendimos a enviar mensajes entre el servidor y el cliente, a unirse a salas y a difundir mensajes. También usamos acuses de recibo para obtener información sobre un usuario en un patrón de solicitud-respuesta con Socket.IO. Finalmente, implementamos la autenticación mediante JWT en Socket.IO, lo que finalizó el backend de nuestra aplicación de chat.

En el próximo capítulo, Capítulo 14, Creación de un frontend para consumir y enviar eventos, vamos a crear un frontend para nuestra aplicación de chat, que interactuará con el backend que creamos en este capítulo.

# 14

## Creando un frontend para Consumir y enviar eventos

Después de crear con éxito un backend de Socket.IO en el capítulo anterior y de realizar nuestros primeros experimentos con el cliente de Socket.IO, ahora nos centraremos en implementar un frontend para conectarnos al backend y consumir y enviar eventos.

Primero, limpiaremos nuestro proyecto eliminando archivos de la aplicación de blog creada previamente. Luego, implementaremos un contexto de React para inicializar y almacenar nuestra instancia de Socket.IO, utilizando elAuthProvider existente para proporcionar el token de autenticación con el backend. Después, implementaremos una interfaz para nuestra aplicación de chat y una forma de enviar mensajes de chat, así como de mostrar los mensajes recibidos. Finalmente, implementaremos comandos de chat con reconocimientos para mostrar en qué salas nos encontramos.

En este capítulo cubriremos los siguientes temas principales:

- Integración del cliente Socket.IO con React
- Implementación de la funcionalidad de chat
- Implementar comandos de chat con reconocimientos

### 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 de los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-modernos-de-React-Full-Stack/tree/principal/capítulo-14>.

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/d\\_TZK6S\\_XDU](https://youtu.be/d_TZK6S_XDU).

## Integración del cliente Socket.IO con React

Comencemos por limpiar el proyecto y eliminar todos los archivos antiguos copiados de la aplicación del blog. ¿En qué vamos a configurar un contexto de Socket.IO para facilitar su inicialización y uso en componentes de React. Finalmente, crearemos nuestro primer componente que utiliza este contexto para mostrar el estado de nuestra conexión de Socket.IO.

### Limpiando el proyecto

Primero eliminemos las carpetas y archivos de la aplicación de blog que creamos anteriormente:

1. Copie la carpeta ch13 existente a una nueva carpeta ch14, de la siguiente manera:

```
$ cp -R ch13 ch14
```

2. Abra la carpeta ch14 en VS Code.

3. Elimine las siguientes carpetas y archivos, ya que solo eran necesarios para el backend de la aplicación del blog:

```
backend/src/__pruebas__/  
backend/src/ejemplo.js  
backend/src/db/modelos/post.js  
backend/src/rutas/posts.js  
backend/src/servicios/posts.js
```

4. En backend/src/app.js, elimine la siguiente importación:

```
importar postRoutes desde './routes/posts.js'
```

5. Además, elimine postRoutes:

```
postRoutes(aplicación)
```

6. Elimine las siguientes carpetas y archivos, ya que solo eran necesarios para la interfaz de la aplicación del blog:

```
src/api/posts.js  
src/componentes/CreatePost.jsx  
src/componentes/Post.jsx  
src/componentes/PostFilter.jsx  
src/componentes/PostList.jsx  
src/componentes/PostSorting.jsx  
src/pages/Blog.jsx
```

Ahora que hemos limpiado nuestro proyecto, comenzemos a implementar un contexto Socket.IO para nuestra nueva aplicación de chat.

## Creación de un contexto Socket.IO

Hasta ahora, hemos estado inicializando la instancia del cliente Socket.IO en el componente `src/App.jsx`. Sin embargo, hacer esto tiene algunas desventajas:

- Para acceder al zócalo en otros componentes, necesitaremos pasarlo a través de accesorios.
- Solo podemos tener una conexión de socket para toda la aplicación.
- No es posible obtener el token dinámicamente desde `AuthContext`, por lo que es necesario almacenar el token en el almacenamiento local.
- Nuestra aplicación requiere una actualización completa para poder cargar el nuevo token y conectarse con él.
- Todavía intentamos conectarnos y recibimos un error cuando no iniciamos sesión.

Para solucionar estos problemas, podemos crear un contexto Socket.IO. A continuación, podemos usar el componente proveedor para hacer lo siguiente:

- Conéctese a Socket.IO solo cuando el token esté disponible en `AuthContext`.
- Almacenar el estado de la conexión Socket.IO y usarlo dentro de los componentes para, por ejemplo, mostrar la interfaz de chat solo cuando se inicia sesión.
- Almacenar el objeto de error y mostrar errores en la interfaz de usuario.

El siguiente diagrama muestra cómo se rastreará el estado de nuestra conexión:

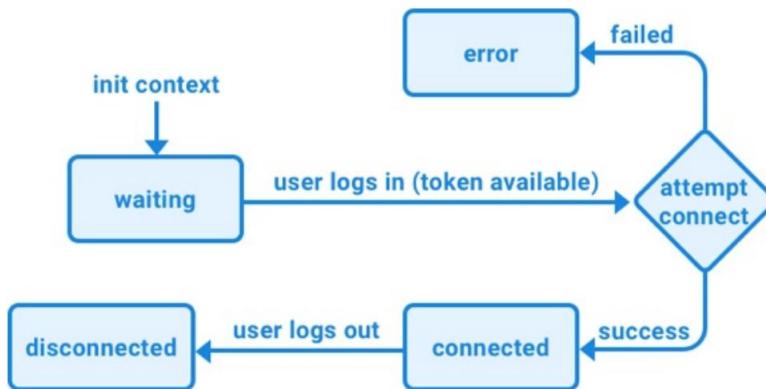


Figura 14.1 – Los diferentes estados de la conexión

Como se puede ver, la conexión del socket espera inicialmente a que el usuario inicie sesión. Una vez que el token está disponible, intentamos establecer una conexión. Si se establece correctamente, el estado cambia a conectado; de lo contrario, a error. Si el socket se desconecta (por ejemplo, al perder la conexión a internet), el estado cambia a desconectado.

Ahora, comenzemos a crear un contexto Socket.IO:

1. Cree un nuevo archivo src/contexts/SocketIOContext.jsx.
2. Dentro de este archivo, importe las siguientes funciones de react, socket.io-client y prop-tipos:

```

importar { createContext, useState, useContext, useEffect } desde 'react'

importar { io } desde 'socket.io-client'
importar PropTypes desde 'prop-types'

```

3. Además, importe el gancho useAuth desde AuthContext para obtener el token actual:

```
importar { useAuth } desde './AuthContext.jsx'
```

4. Ahora, define un contexto de React con algunos valores iniciales para socket, estado y error:

```

exportar const SocketIOContext = createContext({
    socket: nulo,
    estado: 'en espera',
    error: nulo,
})

```

5. A continuación, defina un componente proveedor, en el que primero crearemos ganchos de estado para los diferentes valores del contexto:

```
exportar const SocketIOContextProvider = ({ hijos }) => { const [socket, setSocket] = useState(null)
  const [status, setStatus] = useState('esperando') const [error, setError]
  = useState(null)}
```

6. \$en, use el gancho useAuth para obtener el JWT (si está disponible):

```
constante [token] = useAuth()
```

7. Cree un gancho elect que verifique si el token está disponible y, de ser así, intente conectarse al backend de Socket.IO:

```
useEffect(() => { if (token)
  { const socket =
    io(import.meta.env.VITE_SOCKET_HOST, { consulta: ventana.ubicación.búsqueda.substring(1),
    autenticación: { token }, })}
```

Al igual que antes, pasamos el host, la cadena de consulta y el objeto de autenticación. Sin embargo, ahora obtenemos el token del gancho useAuth en lugar del almacenamiento local.

8. Cree controladores para los eventos connect, connect\_error y disconnect y configure los cadenas de estado y el objeto de error, respectivamente:

```
socket.on('conectar', () => {
  setStatus('conectado')
  setError(null) }

socket.on('error_de_conexión', (err) => {
  setStatus('error') setError(err)

})
socket.on('desconectar', () => setStatus('desconectado'))
```

9. Establezca el objeto de socket y enumere todas las dependencias necesarias para el gancho elect:

```
setSocket(socket)
}
}, [token, setSocket, setStatus, setError])
```

10. Ahora podemos devolver el proveedor, pasándole todos los valores de los ganchos de estado:

```
devolver (
    <SocketIOContext.Provider valor={{ socket, estado, error }}>
        {niños}
    </SocketIOContext.Provider>
)
}
```

11. Finalmente, establecemos PropTypes para el componente proveedor de contexto y definimos un useSocket gancho que simplemente devolverá el contexto completo:

```
SocketIOContextProvider.propTypes = {
    hijos: PropTypes.element.isRequired,
}

función de exportación useSocket() {
    devolver useContext(SocketIOContext)
}
```

Ahora que tenemos un contexto para inicializar nuestro cliente Socket.IO, conectémoslo y mostremos el estado de la conexión del socket.

## Conectando el contexto y mostrando el estado

Ahora podemos eliminar el código para conectarnos a Socket.IO desde el componente App y usar el proveedor en su lugar, de la siguiente manera:

1. Edite src/App.jsx y elimine la siguiente importación:

```
importar { io } desde 'socket.io-client'
```

2. Agregue una importación a SocketIOContextProvider:

```
importar { SocketIOContextProvider } desde './contexts/ SocketIOContext.jsx'
```

3. \$en, elimine el siguiente código relacionado con la conexión Socket.IO:

```
const socket = io(import.meta.env.VITE_SOCKET_HOST, { consulta:
    ventana.ubicación.búsqueda.substring(1), autenticación: { token:

        ventana.localStorage.getItem('token'), }, })

socket.on('conectar', async () => {
```

```
console.log('conectado a socket.io como', socket.id) socket.emit('chat.message', 'hola
del cliente') const userInfo = await socket.emitWithAck('user.info', socket.id)
console.log('información del usuario', userInfo })
```

```
socket.on('error_de_conexión', (err) => { console.error('error de
conexión de socket.io:', err) })
```

```
socket.on('chat.message', (mensaje) => {
  console.log(mensaje) })
```

4. Dentro del componente App, renderiza el proveedor de contexto:

```
función de exportación App() { return (
```

```
  <QueryClientProvider cliente={queryClient}>
    <Proveedor de contexto de autenticación>
      <Proveedor de contexto de socket IO>
        <RouterProvider router={enrutador} />
          Proveedor de contexto de socket IO
          Proveedor de contexto de autenticación
        </ProveedorClienteConsulta>
      )
}
```

Después de conectar el contexto Socket.IO, pasemos a la creación de un componente Estado para mostrar el estado.

#### Creación de un componente de estado

Ahora, creamos un componente Estado para mostrar el estado actual del socket:

1. Cree un nuevo archivo src/components/Status.jsx.
2. Dentro de él, importe el gancho useSocket desde nuestro SocketIOContext:

```
importar { useSocket } desde '../contextos/SocketIOContext.jsx'
```

3. Defina un componente de estado, en el que obtenemos la cadena de estado y el objeto de error. el gancho:

```
función de exportación Estado() { const
  { estado, error } = useSocket()
```

4. Representa el estado del socket:

```
devolver (
  <div>
    Estado del socket: <b>{status}</b>
```

5. Si tenemos un objeto de error, ahora podemos mostrar adicionalmente el mensaje de error:

```
{error && <i> - {mensaje de error}</i>} </div>
)
}
```

Ahora que tenemos un componente Estado, creamos un componente de página Chat, donde representamos los componentes Encabezado y Estado.

#### Creación de un componente de página de chat

Anteriormente teníamos una página de blog para nuestra aplicación de blog, que eliminamos anteriormente en este capítulo. Ahora, creamos un nuevo componente de página de chat para nuestra aplicación de chat:

1. Cree un nuevo archivo src/pages/Chat.jsx.
2. Dentro de él, importa el componente Header (que vamos a reutilizar desde la aplicación Blog) y el componente Estado:

```
importar { Encabezado } desde '../componentes/Header.jsx' importar { Estado }
desde '../componentes/Status.jsx'
```

3. Renderiza un componente Chat en el que mostramos los componentes Encabezado y Estado:

```
función de exportación Chat() { return
  ( <div
    style={{ padding: 8 }}>
      <Encabezado />
      <br />
      <hr />
      <br />
      <Estado /> </div>

    )
}
```

4. Edite src/App.jsx y localice la siguiente importación:

```
importar { Blog } desde './pages/Blog.jsx'
```

Reemplácelo con una importación al componente Chat:

```
importar { Chat } desde './pages/Chat.jsx'
```

5. Finalmente, reemplace el componente <Blog /> en la ruta principal de nuestro enrutador con el componente <Chat /> componente:

```
enrutador constante = crearRouterBrowser([
  {
    ruta: '/',
    elemento:
      <Chat />,
  },
])
```

## Iniciando y probando la interfaz de nuestra aplicación de chat

Ahora podemos comenzar y probar la interfaz de nuestra aplicación de chat:

1. Ejecute el frontend de la siguiente manera:

```
$ npm ejecuta dev
```

2. Ejecute el backend de la siguiente manera (¡asegúrese de que Docker y el contenedor de la base de datos se estén ejecutando!):

```
$ cd backend/ $ npm
run dev
```

3. Ahora vaya a <http://localhost:5173/> y debería ver la siguiente interfaz:

[Log In](#) | [Sign Up](#)

Socket status: **waiting**

Figura 14.2 – Conexión de socket esperando que el usuario inicie sesión

4. Inicie sesión (cree un nuevo usuario si aún no tiene uno) y el socket debería conectarse correctamente:

Logged in as **test**  
[Logout](#)

Socket status: **connected**

Figura 14.3 – Socket conectado después de que el usuario inicia sesión

## Desconectar el socket al cerrar sesión

Quizás hayas notado que, al presionar Cerrar sesión, el socket permanece conectado. Arreglemos esto ahora, desconectando el socket al cerrar sesión:

1. Edite src/components/Header.jsx e importe el gancho useSocket:

```
importar { useSocket } desde './contextos/SocketIOContext.jsx'
```

2. Obtenga el socket del gancho useSocket, de la siguiente manera:

```
función de exportación Header() { const  
[token, setToken] = useAuth() const { socket } = useSocket()
```

3. Defina una nueva función handleLogout, que desconecte el socket y restablezca el token:

```
const handleLogout = () => { socket.disconnect()  
setToken(null)  
}
```

4. Por último, configure el controlador onClick en la función handleLogout:

```
<button onClick={handleLogout}>Cerrar sesión</button>
```

Ahora, al cerrar sesión, el socket se desconectará, como se puede ver en la siguiente captura de pantalla:

[Log In](#) | [Sign Up](#)

---

Socket status: **disconnected**

Figura 14.4 – Socket desconectado después de cerrar sesión

Ahora que el cliente Socket.IO está integrado exitosamente con nuestro frontend React, podemos continuar implementando la funcionalidad de chat en el frontend.

## Implementación de la funcionalidad de chat

Ahora implementaremos la funcionalidad para enviar y recibir mensajes en nuestra aplicación de chat. Primero, implementaremos todos los componentes necesarios. Luego, crearemos un gancho useChat para implementar la lógica que interactúa con la conexión del socket y proporcionar funciones para enviar y recibir mensajes. Finalmente, crearemos una sala de chat para completarlo todo.

## Implementando los componentes de chat

Vamos a implementar los siguientes componentes de chat:

- ChatMessage: para mostrar mensajes de chat
- Ingresar mensaje: Un campo para ingresar nuevos mensajes y un botón para enviarlos

### Implementación del componente ChatMessage

Comencemos implementando el componente ChatMessage:

1. Cree un nuevo archivo src/components/ChatMessage.jsx, que representará un mensaje de chat.
2. Importe PropTypes y defina una nueva función con propiedades de nombre de usuario y mensaje:

```
importar PropTypes desde 'prop-types'

función de exportación ChatMessage({nombre de usuario, mensaje}) {
```

3. Representa el nombre de usuario en negrita y el mensaje que aparece junto a él:

```
devolver (
  <div>
    <b>{nombre de usuario}</b>: {mensaje}
  </div>
)
}
```

4. Defina los tipos de propiedad, de la siguiente manera:

```
ChatMessage.propTypes = {
  nombre de usuario: PropTypes.string.isRequired,
  mensaje: PropTypes.string.isRequired,
}
```

### Implementación del componente EnterMessage

Ahora, creemos el componente EnterMessage, que permitirá a los usuarios enviar un nuevo mensaje de chat:

1. Cree un nuevo archivo src/components/EnterMessage.jsx.
2. Importe el gancho useState y PropTypes:

```
importar { useState } de 'react' importar PropTypes de
'prop-types'
```

3. Defina un nuevo componente EnterMessage, que recibe una función onSend como propiedades:

```
función de exportación EnterMessage({ onSend }) {
```

4. Almacenamos el estado actual del mensaje ingresado:

```
const [mensaje, setMessage] = useState("")
```

5. Entonces, definimos una función para manejar el envío de la solicitud y limpiar el campo posteriormente:

```
función handleSend(e) { e.preventDefault()
    onSend(mensaje) setMessage("")  
}
```

#### Recordatorio

Dado que estamos enviando un formulario mediante un botón de envío, necesitamos llamar a `e.preventDefault()` para evitar que el formulario actualice la página.

6. Renderiza un formulario con un campo de entrada para ingresar el mensaje y un botón para enviarlo:

```
devolver (
    <form onSubmit={handleSend}> <input
        type='text'
        value={message}
        onChange={(e) =>
            setMessage(e.target.value)} /> <input type='submit' value='Enviar' />
    </form>  
)
```

7. Defina los tipos de propiedad de la siguiente manera:

```
EnterMessage.propTypes = {
    onSend: PropTypes.func.isRequired,  
}
```

## Implementando un gancho useChat

Para agrupar toda la lógica, implementaremos un gancho `useChat`, que se encargará del envío y la recepción de mensajes, así como del almacenamiento de todos los mensajes actuales en un gancho de estado. Siga estos pasos para implementarlo:

1. Crea una nueva carpeta llamada `src/hooks/`. Dentro de ella, crea un nuevo archivo llamado `src/hooks/useChat.js`.

2. Importa los ganchos useState y useEffect desde React:

```
importar { useState, useEffect } desde 'react'
```

3. Importa el gancho useSocket desde nuestro contexto:

```
importar { useSocket } desde './contextos/SocketIOContext.jsx'
```

4. Defina una nueva función useChat, donde obtenemos el socket del gancho useSocket, y definir un gancho de estado para almacenar una matriz de mensajes:

```
función de exportación useChat() {
  const { socket } = useSocket() const
  [mensajes, setMessages] = useState([])
```

5. A continuación, defina una función receiveMessage, que añade un nuevo mensaje a la matriz:

```
función recibirMensaje(mensaje) {
  setMessages((mensajes) => [...mensajes, mensaje])
}
```

6. Ahora, crea un gancho elect, en el que creamos un oyente usando socket.on:

```
useEffect(() =>
  { socket.on('chat.message', recibirMensaje)
```

7. Debemos asegurarnos de eliminar el oyente nuevamente usando socket.off cuando se desmonte el gancho effect, de lo contrario podríamos terminar con múltiples oyentes cuando el componente se vuelva a renderizar o desmonta:

```
return () => socket.off('chat.message', recibirMensaje) }, [])
```

8. Ahora, recibir mensajes debería funcionar correctamente. Pasemos al envío de mensajes. Para ello, creamos la función sendMessage, que usa socket.emit para enviar el mensaje:

```
función enviarMensaje(mensaje) {
  socket.emit('chat.message', mensaje)
}
```

9. Por último, devuelve la matriz de mensajes y la función sendMessage para que podamos usarlos en nuestros componentes:

```
devolver { mensajes, enviarMensaje }
```

Ahora que hemos implementado con éxito el gancho useChat, ¡usémoslo!

## Implementación del componente ChatRoom

Finalmente, podemos integrarlo todo e implementar un componente ChatRoom. Sigue estos pasos para empezar:

1. Cree un nuevo archivo src/components/ChatRoom.jsx.
2. Importe el gancho useChat y los componentes EnterMessage y ChatMessage:

```
importar { useChat } desde './hooks/useChat.js' importar { EnterMessage }
desde './EnterMessage.jsx' importar { ChatMessage } desde './ChatMessage.jsx'
```

3. Defina un nuevo componente, que obtenga la matriz de mensajes y la función sendMessage del gancho useChat:

```
función de exportación ChatRoom() {
  const { mensajes, enviarMensaje } = useChat()
```

4. \$en, representa la lista de mensajes como componentes ChatMessage:

```
devolver (
  <div>
    {mensajes.map((mensaje, índice) => (
      <ChatMessage key={índice} {...mensaje} /> ))}
```

5. A continuación, renderice el componente EnterMessage y pase la función sendMessage como propiedad onSend:

```
<IngresarMensaje alEnviar={enviarMensaje} /> </div>

)
}
```

6. Edite src/pages/Chat.jsx e importe el componente ChatRoom y el gancho useSocket:

```
importar { ChatRoom } desde './components/ChatRoom.jsx' importar { useSocket } desde
'./contexts/SocketIOContext.jsx'
```

7. Obtenga el estado del gancho useSocket en el componente de la página Chat:

```
función de exportación Chat() { const
  { status } = useSocket()
```

8. Si el estado está conectado, mostramos el componente ChatRoom:

```
devolver (
    <div style={{ relleno: 8 }}>
        <Encabezado />
        <br />
        <hr />
        <br />
        <Estado />
        <br />
        <hr />
        <br />
    {estado === 'conectado' && <Sala de chat />}
```

9. Ahora, ve a <http://localhost:5173/> en tu navegador e inicia sesión con tu nombre de usuario y contraseña. El socket \$e se conecta y se muestra la sala de chat. Introduce un mensaje de chat y envíalo pulsando Intro o haciendo clic en el botón Enviar . Verás que el mensaje se ha recibido y mostrado.

10. Abra una segunda ventana del navegador e inicie sesión con otro usuario. Envíe otro mensaje desde allí. Verá que ambos usuarios reciben el mensaje, como se muestra en la siguiente captura de pantalla:

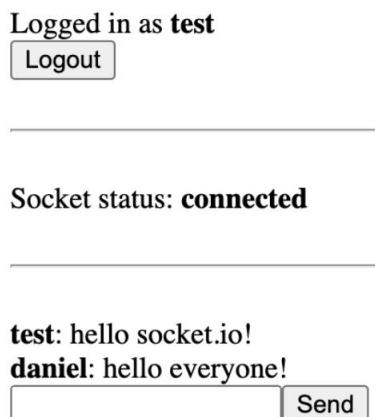


Figura 14.5 – Envío y recepción de mensajes de diferentes usuarios

Ahora que tenemos una aplicación de chat básica en funcionamiento, exploremos cómo podríamos implementar comandos de chat usando reconocimientos.

## Implementar comandos de chat con reconocimientos

Además de enviar y recibir mensajes, las aplicaciones de chat suelen ofrecer una forma de enviar comandos al cliente o al servidor. Por ejemplo, podríamos enviar el comando /clear para borrar nuestra lista de mensajes local. O podríamos enviar el comando /rooms para obtener la lista de salas en las que estamos. Sigue estos pasos para implementar los comandos de chat:

1. Edite src/hooks/useChat.js y ajuste la función sendMessage. Primero,

Hagámoslo una función asíncrona:

```
función asíncrona sendMessage(mensaje) {
```

2. Reemplace el contenido de la función con lo siguiente. Primero, verificamos si el mensaje comienza con una barra diagonal (/). De ser así, obtenemos el comando eliminando la barra diagonal y usando una declaración de cambio:

```
si (mensaje.iniciaCon('/')) {  
    const comando = mensaje.substring(1)  
    cambiar (comando) {
```

3. Para el comando clear, simplemente establecemos la matriz de mensajes en una matriz vacía:

```
caso 'claro':  
    establecerMensajes([])  
    romper
```

4. Para el comando de salas, obtenemos la información del usuario usando socket.emitWithAck y nuestro propio socket.id:

```
caso 'habitaciones': {  
    const userInfo = await socket.emitWithAck('usuario.info', socket.id)
```

5. Luego, obtenemos la lista de habitaciones, filtrando nuestra propia habitación (con el nombre de nuestro socket.id) que unimos automáticamente en Socket.IO:

```
const habitaciones = userInfo.habitaciones.filter((habitación) => habitación !=  
    socket.id)
```

6. Reutilizamos la función receiveMessage para enviar un mensaje desde el servidor, indicándonos la

Habitaciones en las que nos encontramos:

```
recibirMensaje({  
    mensaje: 'Estás en: ${rooms.join(', ')}'  
})  
romper  
}
```

Tenga en cuenta que no enviamos un nombre de usuario, solo un mensaje. Tendremos que adaptar el componente ChatMessage para que esto sea posible más adelante.

7. Si recibimos cualquier otro comando, mostramos un mensaje de error:

```
por defecto:  
receiveMessage({ mensaje:  
    'Comando desconocido: ${comando}', })  
  
romper  
}
```

8. De lo contrario (si el mensaje no comienza con una barra), simplemente emitimos el mensaje de chat, como antes:

```
} else  
    { socket.emit('chat.message', mensaje)  
    }  
}
```

9. Finalmente, edite src/components/ChatMessage.jsx y adapte el componente para que muestre un mensaje del sistema si no se proporcionó ningún nombre de usuario:

```
función de exportación ChatMessage({nombre de usuario, mensaje}) {  
    devolver (  
        <div>  
            {nombre de usuario ? (  
                <span>  
                    <b>{nombre de usuario}</b>: {mensaje}  
                ) : (  
                    <i>{mensaje}</i> )></div>  
    )  
}
```

10. No olvide ajustar PropTypes para que el nombre de usuario sea opcional (eliminando .isRequired de la propiedad de nombre de usuario):

```
ChatMessage.propTypes = {  
    nombre de usuario: PropTypes.string,
```

11. Accede a `http://localhost:5173/` en tu navegador e intenta enviar un par de mensajes. Escribe `/clear` y verás que se borraron todos los mensajes. A continuación, escribe `/rooms` para obtener la lista de salas en las que te encuentras, como se muestra en la siguiente captura de pantalla:

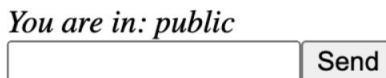


Figura 14.6 – Envío del comando `/rooms`

Nota

Unirse a diferentes salas actualmente no funciona debido a que el parámetro de consulta se borra después de iniciar sesión. En el próximo capítulo, vamos a refactorizar la aplicación de chat e implementar un / Comando "Únete" para unirte a una sala diferente.

## Resumen

En este capítulo, implementamos un frontend para el backend de nuestra aplicación de chat. Empezamos integrando el socket. Cliente IO con React, creando un contexto y un gancho personalizado. Usamos AuthProvider para obtener el token que autentica al usuario al conectarse al socket. Posteriormente, mostramos el estado de nuestro socket. Implementamos una interfaz de chat para enviar y recibir mensajes. Finalmente, implementamos comandos de chat mediante reconocimientos para obtener las salas en las que nos encontramos.

En el próximo capítulo, Capítulo 15, Agregar persistencia a Socket.IO usando MongoDB, aprenderemos cómo almacenar y reproducir mensajes enviados previamente usando MongoDB con Socket.IO.

## Cómo añadir persistencia a Socket.IO con MongoDB

Ahora que hemos implementado un backend y un frontend de Socket.IO, dediquemos tiempo a integrarlo con la base de datos MongoDB. Para ello, almacenaremos temporalmente los mensajes en la base de datos y los reproduciremos cuando se una un nuevo usuario, de modo que puedan ver el historial de chat tras su incorporación. Además, refactorizaremos nuestra aplicación de chat para que esté preparada para futuras expansiones y mantenimiento. Finalmente, probaremos la nueva estructura implementando nuevos comandos para unirse y cambiar de sala.

En este capítulo cubriremos los siguientes temas principales:

- Almacenamiento y reproducción de mensajes mediante MongoDB
- Refactorizar la aplicación para que sea más extensible
- Implementar comandos para unirse y cambiar de sala

### Requisitos técnicos

Antes de comenzar, instale todos los requisitos del Capítulo 1, Preparación para el desarrollo de pila completa, 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-modernos-de-React-Full-Stack-árbol/principal/capítulo-15>.

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/Mi7Wj\\_jxjhM](https://youtu.be/Mi7Wj_jxjhM).

## Almacenamiento y reproducción de mensajes mediante MongoDB

Actualmente, si un nuevo usuario se une al chat, no verá ningún mensaje hasta que alguien lo envíe activamente. Por lo tanto, los nuevos usuarios no podrán participar adecuadamente en las conversaciones en curso. Para solucionar este problema, podemos almacenar los mensajes en la base de datos y reproducirlos cuando un usuario se une.

### Creación del esquema Mongoose

Siga estos pasos para crear un esquema Mongoose para almacenar mensajes de chat:

1. Copie la carpeta ch14 existente a una nueva carpeta ch15, de la siguiente manera:

```
$ cp -R ch14 ch15
```

2. Abra la nueva carpeta ch15 en VS Code.
3. Cree un nuevo archivo backend/src/db/models/message.js.
4. Dentro de él, defina un nuevo messageSchema, que vamos a utilizar para almacenar los mensajes de chat. en la base de datos:

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

```
const messageSchema = nuevo esquema({
```

5. El esquema del mensaje \$e debe contener el nombre de usuario (persona que envió el mensaje), el mensaje, la sala en la que se envió y la fecha de envío del mensaje:

```
    nombre de usuario: { tipo: String, requerido: verdadero },
    mensaje: { tipo: String, requerido: true },
    habitación: { tipo: String, requerido: true },
    enviado: { tipo: Fecha, caduca: 5 * 60, predeterminado: Fecha.ahora, requerido:
    verdadero },
})
```

Para la fecha de envío, especificamos "expires" para que los mensajes expiren automáticamente después de 5 minutos (5 x 60 segundos). \$is garantiza que nuestra base de datos no se sature con muchos mensajes de chat. También establecemos el valor predeterminado en "Date.now" para que todos los mensajes se marquen como enviados a la hora actual.

#### Información

En realidad, MongoDB solo verifica la expiración de los datos una vez por minuto, por lo que los documentos expirados podrían persistir hasta un minuto después de su tiempo de expiración definido.

6. Cree un modelo a partir del esquema y expórtelo:

```
exportar const Mensaje = mongoose.model('mensaje', messageSchema)
```

Después de crear el esquema y el modelo de Mongoose, pasemos a la creación de las funciones de servicio para gestionar los mensajes de chat.

## Creación de las funciones de servicio

Necesitamos crear funciones de servicio para guardar un mensaje nuevo en la base de datos y obtener todos los mensajes enviados en una sala determinada, ordenados por fecha de envío, mostrando primero los más antiguos. Siga estos pasos para implementar las funciones de servicio:

1. Cree un nuevo archivo backend/src/services/messages.js.

2. Dentro de él, importe el modelo Mensaje:

```
importar { Mensaje } desde './db/models/message.js'
```

3. \$en, define una función para crear un nuevo objeto Mensaje en la base de datos:

```
exportar función asíncrona createMessage({nombre de usuario, mensaje, sala}) {  
  
    const messageDoc = new Message({nombre de usuario, mensaje, sala}) return await  
    messageDoc.save()  
}
```

4. Además, define una función para obtener todos los mensajes de una determinada sala, enumerando primero los mensajes más antiguos:

```
exportar función asíncrona getMessagesByRoom(room) {  
    devolver esperar Mensaje.find({ habitación }).sort({ enviado: 1 })  
}
```

A continuación, vamos a utilizar estas funciones de servicio en nuestro servidor de chat.

## Almacenamiento y reproducción de mensajes

Ahora que tenemos todas las funciones, necesitamos implementar el almacenamiento y la reproducción de mensajes en nuestro servidor de chat. Siga estos pasos para implementar la funcionalidad:

1. Edite backend/src/socket.js e importe las funciones de servicio que definimos anteriormente:

```
importar { createMessage, getMessagesByRoom } desde './services/ messages.js'
```

2. Cuando un nuevo usuario se conecta, recibe todos los mensajes de la sala actual y envíalos (reproducidos). al usuario que utiliza socket.emit:

```
función de exportación handleSocket(io) {  
    io.on('conexión', async (socket) => { const sala =  
        socket.handshake.query?.sala ?? 'público' socket.join(sala) console.log(socket.id, 'sala unida:',  
        sala)  
  
        const mensajes = await getMessagesByRoom(room) mensajes.forEach(({ nombre  
        de usuario, mensaje }) =>  
            socket.emit('chat.message', { nombredeusuario, mensaje }),  
        )  
    })  
}
```

3. Además, cuando un usuario envía un mensaje, guárdelo en la base de datos:

```
socket.on('chat.message', (mensaje) => { console.log(`${socket.id}: ${  
    mensaje}`) io.to(room).emit('chat.message', { nombredeusuario:  
        socket.usuario.nombredeusuario,  
  
        mensaje,  
    })  
    createMessage({ nombredeusuario: socket.usuario.nombredeusuario, mensaje, sala }) })
```

4. Inicie el servidor frontend de la siguiente manera:

```
$ npm ejecuta dev
```

5. \$en, inicia el servidor backend (¡no olvides iniciar el contenedor Docker para la base de datos!):

```
$ cd backend/ $ npm  
run dev
```

6. Vaya a <http://localhost:5173/>, inicie sesión y envíe algunos mensajes. \$en, abra una nueva pestaña, inicie sesión con un usuario diferente y verá que los mensajes enviados anteriormente se reproducen:

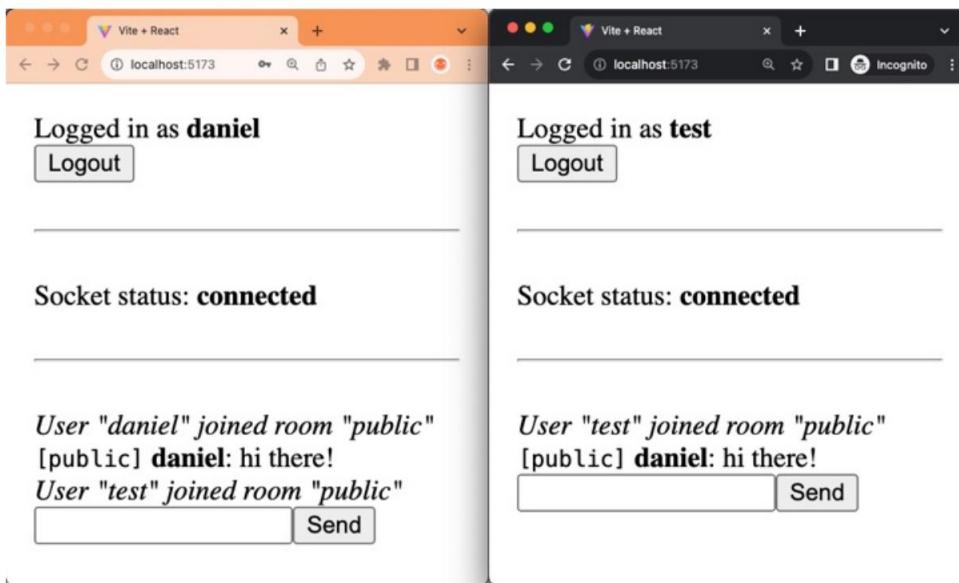


Figura 15.1 – Reproducción exitosa de mensajes almacenados

#### Nota

La captura de pantalla de la Figura 15.1 corresponde a una versión posterior de la aplicación, donde mostramos mensajes cuando un usuario se une a una sala (implementaremos estos mensajes más adelante en este capítulo). Aquí, usamos estos mensajes para mostrar que la reproducción funciona cuando el usuario se une después de enviar un mensaje.

Si esperas 5 minutos y vuelves a unirte al chat, verás que los mensajes existentes han expirado y ya no se reproducen.

Ahora, vamos a aclarar qué mensajes se reprodujeron en la interfaz de usuario.

## Distinguir visualmente los mensajes reproducidos

Actualmente, parece que el otro usuario envió el mensaje justo después de que nos uniéramos. No está claro si el mensaje se reprodujo desde el servidor. Para solucionar este problema, podemos distinguir visualmente los mensajes reproducidos, por ejemplo, haciéndolos ligeramente más grises. Procedamos de la siguiente manera:

1. Edite backend/src/socket.js y agregue un %ag reproducido a los mensajes reproducidos:

```
const mensajes = await getMessagesByRoom(habitación)
mensajes.paraCada(({nombredeusuario, mensaje}) =>
    socket.emit('chat.message', { nombre de usuario, mensaje, reproducido: verdadero }),
)
```

2. Ahora, edite src/components/ChatMessage.jsx y, si se configuró el %ag reproducido, muestre los mensajes con una opacidad menor:

```
función de exportación ChatMessage({ nombre de usuario, mensaje, reproducido }) {  
    devolver (  
        <div style={{opacidad: reproducido ? 0.5 : 1.0 }}>
```

3. No olvides actualizar propTypes y agregar el %ag reproducido:

```
ChatMessage.propTypes = {  
    nombre de usuario: PropTypes.string,  
    mensaje: PropTypes.string.isRequired,  
    reproducido: PropTypes.bool,  
}
```

4. Vuelva a <http://localhost:5173> y repita el mismo procedimiento (enviando mensajes desde un usuario y luego iniciando sesión con un usuario diferente en otra pestaña). Verá que ahora es fácil distinguir los mensajes reproducidos de los nuevos.



Figura 15.2 – Los mensajes reproducidos ahora se muestran en un color más claro

Ahora que hemos almacenado exitosamente nuestro historial de mensajes en la base de datos, centrémonos un poco en refactorizar la aplicación de chat para que sea más extensible y fácil de mantener en el futuro.

## Refactorizar la aplicación para que sea más extensible

Para la refactorización, comenzaremos por definir funciones de servicio para todas las funcionalidades de chat que proporciona nuestro servidor.

### Definición de funciones de servicio

Siga estos pasos para comenzar a definir funciones de servicio para la funcionalidad de chat:

1. Cree un nuevo archivo backend/src/services/chat.js.

2. Dentro de él, importe las funciones de servicio relacionadas con los mensajes:

```
importar { createMessage, getMessagesByRoom } desde './messages.js'
```

3. Definir una nueva función para enviar un mensaje privado directamente a un usuario:

```
función de exportación sendPrivateMessage(  
    enchufe,  
    { nombre de usuario, sala, mensaje, reproducido },  
){  
    socket.emit('chat.message', { nombre de usuario, mensaje, sala, reproducido })  
  
}
```

Los mensajes privados se utilizarán, por ejemplo, para reproducir mensajes a un usuario específico y no se almacenan en la base de datos.

4. Además, define una función para enviar un mensaje del sistema:

```
función de exportación sendSystemMessage(io, { sala, mensaje }) {  
    io.to(sala).emit('chat.message', { mensaje, sala })  
}
```

Los mensajes del sistema se usarán, por ejemplo, para anunciar que un usuario se ha unido a una sala. Tampoco queremos almacenarlos en la base de datos.

5. \$en, define una función para enviar un mensaje público:

```
función de exportación sendPublicMessage(io, { nombre de usuario, sala, mensaje }) {  
  
    io.to(room).emit('chat.message', { nombre de usuario, mensaje, sala })  
    createMessage({nombre de usuario, mensaje, sala})  
}
```

Los mensajes públicos se utilizarán para enviar mensajes de chat regulares a una sala. Estos mensajes se almacenan en la base de datos para que podamos reproducirlos más tarde.

6. También definimos una nueva función para unir un socket dado a una habitación:

```
exportar función asíncrona joinRoom(io, socket, { room }) {
    socket.join(habitación)
```

7. Dentro de esta función, envía un mensaje del sistema avisando a todos en la sala que alguien se unió:

```
enviarMensajeDelSistema(io, {
    habitación,
    mensaje: `El usuario "${socket.user.username}" se unió a la sala "${room}"`, })
```

8. \$en, reproduce todos los mensajes que se enviaron a la sala de forma privada al usuario que acaba de unirse:

```
const mensajes = await getMessagesByRoom(room) mensajes.forEach(({ nombre
de usuario, mensaje }) =>
    sendPrivateMessage(socket, { nombre de usuario, mensaje, sala, reproducido: verdadero })

)
}
```

9. Por último, defina una función de servicio para obtener la información del usuario del socketId. Simplemente copie y pegue el código que teníamos previamente en backend/src/socket.js aquí:

```
exportar función asíncrona getUserInfoBySocketId(io, socketId) {
    const sockets = await io.in(socketId).fetchSockets()
    si (sockets.length === 0) devuelve nulo const socket =
        sockets[0] const userInfo = { socketId,
        rooms: Array.from(socket.rooms),
        usuario:
        socket.user,

    }
    devolver información del usuario
}
```

Ahora que hemos creado las funciones de servicio para la funcionalidad de chat, usémoslas en el servidor Socket.IO.

## Refactorización del servidor Socket.IO para utilizar las funciones de servicio

Ahora que hemos definido las funciones de servicio, refactoricemos el código del servidor de chat para usarlas. Siga estos pasos:

1. Abra backend/src/socket.js y encuentre la siguiente importación:

```
importar { createMessage, getMessagesByRoom } desde './services/  
mensajes.js'
```

Reemplace la importación anterior con la siguiente importación a las nuevas funciones del servicio de chat:

```
importar {  
    unirse a la sala,  
    enviarMensajePúblico,  
    obtenerInformaciónDeUsuarioPorIdDeSocket,  
} de './services/chat.js'
```

2. Reemplace la función handleSocket por completo con el siguiente código. Al establecer una conexión, nos unimos automáticamente a la sala pública mediante la función de servicio joinRoom:

```
función de exportación handleSocket(io) {  
    io.on('conexión', (socket) => {  
        joinRoom(io, socket, { habitación: 'pública' })  
    })  
}
```

3. \$en, define un oyente para el evento chat.message y envíalo a la sala indicada usando

La función del servicio sendPublicMessage:

```
socket.on('chat.message', (sala, mensaje) =>  
    sendPublicMessage(io, { nombre de usuario: socket.usuario.nombredeusuario,  
    habitación, mensaje }),  
)
```

### Nota

Hemos modificado la firma del evento chat.message para que ahora se requiera pasar una sala, lo que nos permitirá implementar una mejor gestión de varias salas más adelante. Más adelante, debemos asegurarnos de ajustar el código del cliente para que esto sea posible.

4. A continuación, defina un detector para el evento user.info, en el que utilizamos la función de servicio asíncrono getUserInfoBySocketId y devolvemos el resultado en una devolución de llamada para convertir este evento en un reconocimiento:

```
socket.on('usuario.info', async (socketId, devolución de llamada) =>  
    devolución de llamada (espera getUserInfoBySocketId (io, socketId)),  
)  
})
```

5. Finalmente, podemos reutilizar el middleware de autenticación de antes:

```
io.use((socket, siguiente) => {
    si (!socket.handshake.auth?.token) {
        devolver siguiente(nuevo Error('Error de autenticación: no hay token')
proporcionó'))
    }
    jwt.verify(
        socket.handshake.auth.token,
        proceso.env.JWT_SECRET,
        async (err, token decodificado) => {
            si (err) {
                devolver siguiente(nuevo Error('Error de autenticación: no válido
simbólico'))
            }
            socket.auth = token decodificado
            socket.usuario = await getUserInfoById(socket.auth.sub)
            devolver siguiente()
        },
    )
})
}
```

Ahora que nuestro servidor de chat está refactorizado, continuemos con la refactorización del código del lado del cliente.

## Refactorización del código del lado del cliente

Ahora que nuestro código del lado del servidor usa funciones de servicio para encapsular la funcionalidad de la aplicación de chat, hagamos una refactorización similar del código del lado del cliente extrayendo los comandos del lado del cliente en funciones separadas, de la siguiente manera:

1. Edite src/hooks/useChat.js y dentro del gancho useChat, defina una nueva función para borrar los mensajes:

```
función clearMessages() {
    establecerMensajes([])
}
```

2. \$en, define una función asíncrona para obtener todas las salas en las que se encuentra el usuario:

```
función asíncrona getRooms() {
    const userInfo = await socket.emitWithAck('usuario.info', socket.id)

    const rooms = userInfo.rooms.filter((room) => room !== socket.id)
```

```
habitaciones de retorno  
}
```

3. Ahora podemos utilizar estas funciones en la función sendMessage, de la siguiente manera:

```
función asíncrona sendMessage(mensaje) { if  
(mensaje.startsWith('/')) {  
    const comando = mensaje.substring(1) switch(comando) {  
  
        caso 'claro':  
            clearMessages()  
            romper  
  
        caso 'habitaciones': {  
            const rooms = await getRooms()  
            receiveMessage({ mensaje:  
                `Estás en: ${rooms.join(', ')}` , })  
  
            romper  
    }  
}
```

4. Por último, ajustamos el evento chat.message para enviar la sala además del mensaje. Por ahora,  
Siempre enviamos mensajes a la sala 'pública':

```
por defecto:  
receiveMessage({ mensaje:  
    `Comando desconocido: ${comando}` , })  
  
romper  
  
} } else  
{ socket.emit('chat.message', 'público', mensaje)  
}  
}
```

En la siguiente sección, ampliaremos esto para poder cambiar entre diferentes salas.

5. Vaya a <http://localhost:5173/> y verifique que la aplicación de chat siga funcionando de la misma manera.  
como antes.

Ahora que hemos refactorizado con éxito nuestra aplicación de chat para que sea más extensible, probemos la flexibilidad de la nueva estructura implementando nuevos comandos para unirse y cambiar de sala.

## Implementar comandos para unirse y cambiar de sala

Ahora probemos la nueva estructura implementando comandos para unirnos y cambiar de sala en la aplicación de chat, de la siguiente manera:

1. Edite backend/src/socket.js y defina un nuevo oyente debajo del chat.message oyente, que llamará a la función de servicio joinRoom cuando recibamos un evento chat.join del cliente:

```
socket.on('chat.join', (sala) => joinRoom(io, socket, { sala
}))
```

Como podemos ver, tener una función de servicio "joinRoom" facilita enormemente la reutilización del código para unirse a una nueva sala. Ya envía un mensaje del sistema que informa a todos que alguien se ha unido a la sala, tal como ocurre cuando el usuario se une a la sala pública por defecto al conectarse.

2. Edite src/components/ChatMessage.jsx y muestre la sala:

```
función de exportación ChatMessage({ sala, nombre de usuario, mensaje, reproducido }) {

    devolver (
        <div style={{opacidad: reproducido ? 0.5 : 1.0 }}>
            {nombre de usuario} (
                <span>
                    <code>[{sala}]</code> <b>{nombre de usuario}</b>: {mensaje}
```

3. Agregue la propiedad de la habitación a la definición de propTypes:

```
ChatMessage.propTypes = {
    nombre de usuario: PropTypes.string,
    mensaje: PropTypes.string.isRequired,
    reproducido: PropTypes.bool,
    habitación: PropTypes.string,
}
```

4. Ahora, edite src/hooks/useChat.js y defina un gancho de estado para almacenar la sala en la que estamos.

Actualmente en:

```
función de exportación useChat() {
    const { socket } = useSocket()
    const [mensajes, setMessages] = useState([])
    const [habitaciónActual, establecerHabitaciónActual] = useState('público')
```

Por defecto, estamos en la sala pública.

5. Definir una nueva función para cambiar de habitación:

```
función switchRoom(habitación)
  { setCurrentRoom(habitación)
  }
```

Por el momento, solo estamos llamando a setCurrentRoom aquí, pero es posible que queramos ampliar esta función más adelante, por lo que es una buena práctica abstraerla de antemano en una función separada.

6. Defina una nueva función para unirse a una sala enviando el evento chat.join y cambiando el habitación actual:

```
función joinRoom(sala)
  { socket.emit('chat.join', sala) switchRoom(sala)

  }
```

7. Cambie la función sendMessage para aceptar argumentos para los comandos, de la siguiente manera:

```
función asíncrona sendMessage(mensaje) {
  (mensaje.startsWith('/')) {
    const [comando, ...args] = mensaje.substring(1).split(' ')
    switch (comando) {
```

Ahora podemos enviar comandos como /join <room-name> y el nombre de la sala se almacenará en args[0].

8. Definir un nuevo comando para unirse a una sala, en el que primero verificamos si se pasaron argumentos al comando:

```
caso 'unirse': { si
  (args.length === 0) { devolver
    recibirMensaje({
      Mensaje: 'Proporcione un nombre de sala: /join
<habitación>',
    })
  }
}
```

9. Enseguida nos aseguramos de que no nos hemos unido ya a la sala usando la función getRooms:

```
const room = args[0] const rooms
= await obtenerRooms() si (rooms.includes(room))
{ devolver recibirMensaje({
  mensaje: `Ya estás en la habitación "${room}"` , })
}
```

10. Finalmente, podemos unirnos a la sala utilizando la función joinRoom:

```
joinRoom(sala)
romper
{}
```

11. De manera similar, podemos implementar el comando /switch de la siguiente manera:

```
caso 'cambiar': { si
    (args.length === 0) { devolver
        recibirMensaje({|
            Mensaje: 'Proporcione un nombre de sala: /switch
<habitación>',
        |})
    }

    } const room = args[0] const
    rooms = await obtenerRooms() si (!
        rooms.includes(room)) { devolver
            recibirMensaje({|
                Mensaje: 'No estás en la habitación "${room}" . Escribe "/"'
                "Únete a ${room}" para unirte primero.`,
            |})
        }

    } switchRoom(room)
    receiveMessage({ mensaje:
        `Cambiado a la habitación "${room}" .` , })
}

romper
{}
```

En este caso, comprobamos si el usuario ya está en la sala. De no ser así, le indicamos que debe unirse a la sala antes de cambiar a ella.

12. Ajuste el evento chat.message para enviarlo a la sala actual, de la siguiente manera:

```
} else
{ socket.emit('chat.message', salaactual, mensaje)
}
```

13. Ve a <http://localhost:5173/>, envía un mensaje a la sala pública y únete a la sala de React ejecutando el comando /join react. Envía un mensaje diferente a esa sala.

14. Abre otra ventana del navegador, inicia sesión con un usuario diferente y verás que se reproduce el primer mensaje de la sala pública. Sin embargo, no vemos el mensaje de la sala de reacción, ya que aún no nos hemos unido.

15. Ahora, en la segunda ventana del navegador, llama también a /join React. Verás que el segundo mensaje se reproduce.

16. Intenta usar /switch public para volver a la sala pública y enviar otro mensaje . Verás que ambos clientes lo reciben porque ambos están en la sala pública.

El resultado de estas acciones se puede ver en la siguiente captura de pantalla:

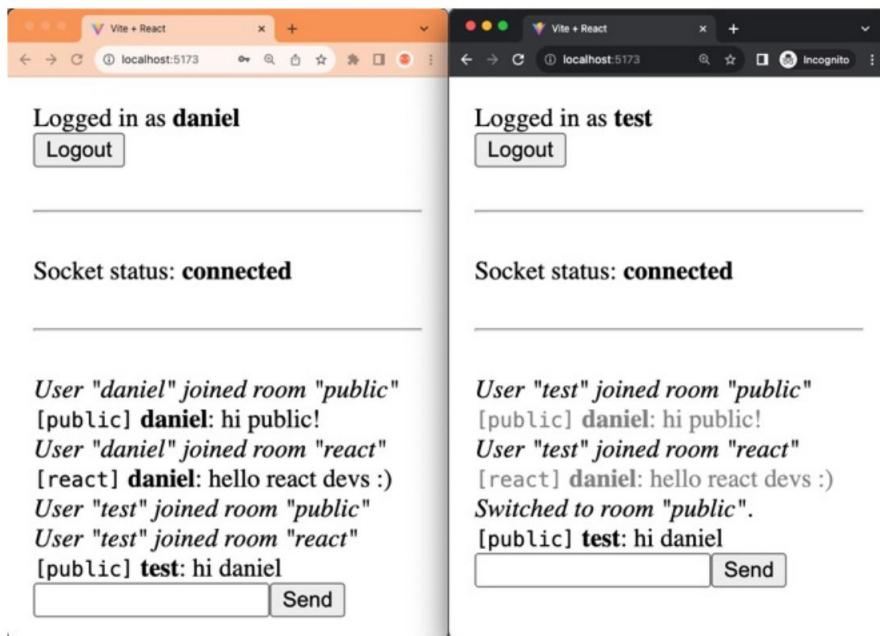


Figura 15.3 – Chat en diferentes salas

## Resumen

En este capítulo, primero conectamos nuestra aplicación de chat a la base de datos almacenando mensajes en MongoDB. También aprendimos a hacer que los documentos caduquen tras un tiempo determinado. Luego, implementamos la función de reproducir mensajes cuando un nuevo usuario se une al chat. A continuación, dedicamos tiempo a refactorizar la aplicación de chat para que sea más extensible y fácil de mantener en el futuro. Finalmente, implementamos formas de unirse a nuevas salas y cambiar entre ellas.

Hasta ahora, solo hemos usado bibliotecas para desarrollar nuestras aplicaciones. En el siguiente capítulo, Capítulo 16, Introducción a Next.js, aprenderemos a usar un framework React completo para desarrollar aplicaciones. Frameworks como Next.js proporcionan mayor estructura a nuestras aplicaciones y nos ofrecen numerosas funciones, como la renderización del lado del servidor, listas para usar.



## Parte 5:

# Avanzando hacia la preparación para la empresa

## Aplicaciones de pila completa

En esta parte, presentaremos Next.js como un framework de aplicaciones full-stack listo para empresas. Aprenderemos cómo funciona y cuáles son sus ventajas sobre usar **solo** React . A continuación, crearemos una aplicación usando Next.js y el nuevo paradigma App Router. Posteriormente, presentaremos los componentes y acciones de servidor de React como una forma de interactuar directamente con la base de datos, sin requerir una API REST o GraphQL. A continuación, profundizaremos en el framework Next.js y aprenderemos sobre el almacenamiento en caché, **las rutas** de API , la adición de metadatos y cómo cargar imágenes y fuentes de forma óptima. A continuación, aprenderemos a implementar una aplicación Next.js usando Vercel y una configuración de implementación personalizada con Docker. Finalmente, ofrecemos una visión general y cubrimos brevemente varios temas avanzados en desarrollo full-stack que aún no se han tratado en este libro. Esto incluye conceptos como el mantenimiento de proyectos a gran escala, la optimización del tamaño del paquete, una visión general de las bibliotecas de UI y soluciones avanzadas de gestión de estado.

Esta parte incluye los siguientes capítulos:

- Capítulo 16, Introducción a Next.js
- Capítulo 17, Introducción a los componentes del servidor React
- Capítulo 18, Conceptos y optimizaciones avanzadas de Next.js
- Capítulo 19, Implementación de una aplicación Next.js
- Capítulo 20, Profundizando en el desarrollo full-stack



# 16

## Introducción a Next.js

Hasta ahora, hemos estado utilizando varias bibliotecas y herramientas para desarrollar aplicaciones web completas. Ahora, presentamos Next.js como un marco de aplicación web de pila completa listo para la empresa para React. Next.js combina todas las funciones y herramientas que necesitas para el desarrollo web completo en un solo paquete. En este libro, utilizamos Next.js porque actualmente es el framework más popular que admite todas las nuevas funciones de React, como los Componentes de Servidor y las Acciones de Servidor de React, que representan el futuro del desarrollo full-stack con React. Sin embargo, existen otros frameworks para React full-stack, como Remix, que recientemente también han empezado a ser compatibles con las nuevas funciones de React.

En este capítulo, aprenderemos cómo funciona Next.js y cuáles son sus ventajas. Luego, recrearemos nuestro proyecto de blog en Next.js para destacar las diferencias entre usar un empaquetador simple como Vite y un framework completo como Next.js. A lo largo del camino, aprenderemos cómo funciona el enrutador de aplicaciones de Next.js. Por último, vamos a recrear nuestra aplicación de blog (estática) creando componentes y páginas y luego definiendo enlaces entre ellos.

En este capítulo cubriremos los siguientes temas principales:

- ¿Qué es Next.js?
- Configuración de Next.js
- Presentamos el enrutador de aplicaciones
- Creación de componentes y páginas estáticas

## 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-modernos-de-React-Full-Stack/tree/principal/capítulo-16>.

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

## ¿Qué es Next.js?

Next.js es un framework de React que reúne todo lo necesario para crear una aplicación web completa con React. Sus principales características son las siguientes:

- Buena experiencia de desarrollador lista para usar, incluida la recarga de módulos en caliente, el manejo de errores, y más.
- Enrutamiento basado en archivos y diseños anidados, controladores de ruta para definir puntos finales de API, todo desde Next.js.
- Soporte de internacionalización (i18n) en enrutamiento, permitiéndonos crear rutas internacionalizadas.
- Obtención de datos mejorada del lado del servidor y del lado del cliente con almacenamiento en caché listo para usar.
- Middleware para ejecutar el código antes de que se completen las solicitudes.
- Opciones para ejecutar puntos finales de API en entornos de ejecución sin servidor.
- Soporte listo para usar para la generación estática de páginas.
- Transmisión dinámica de componentes cuando son necesarios, lo que nos permite mostrar una página inicial rápidamente y luego cargar otros componentes más tarde.
- Renderizado avanzado de cliente y servidor, que nos permite no solo renderizar componentes React en el lado del servidor (renderizado del lado del servidor (SSR)) sino también utilizar componentes de servidor React, que nos permiten renderizar componentes React exclusivamente en el servidor sin enviar JavaScript adicional al cliente.
- Acciones del servidor para mejorar progresivamente los formularios y acciones enviados desde el cliente al servidor, Permitiéndonos enviar formularios incluso sin JavaScript en el cliente.
- Optimizaciones integradas para imágenes, fuentes y scripts para mejorar Core Web Vitals.
- Además, Next.js proporciona una plataforma para implementar fácilmente nuestras aplicaciones en Vercel.

En resumen, Next.js integra todo lo aprendido sobre desarrollo full-stack a lo largo de este libro, perfecciona cada concepto, haciéndolo más avanzado y personalizable, y lo ofrece todo en un solo paquete. Ahora vamos a recrear la aplicación de blog de capítulos anteriores, pero desde cero con Next.js. Esto nos permitirá ver las diferencias entre desarrollar una aplicación con y sin un framework full-stack.

## Configuración de Next.js

Ahora vamos a configurar un nuevo proyecto con la herramienta `create-next-app`, que configura todo automáticamente. Sigue estos pasos para empezar:

1. Abra una nueva ventana de Terminal. Asegúrese de estar fuera de las carpetas de proyecto. Ejecute el siguiente comando para crear una nueva carpeta e inicializar allí un proyecto Next.js:

```
$ npx crear-siguiente-aplicación@14.1.0
```

2. Cuando se le pregunte si desea continuar, presione y confirme presionando Retorno/Entrar.
3. Dale un nombre al proyecto, como por ejemplo `ch16`.
4. Responda las preguntas de la siguiente manera:

¿Te gustaría utilizar TypeScript?: No

¿Te gustaría utilizar ESLint?: Sí

¿Te gustaría utilizar Tailwind CSS?: No

¿Te gustaría utilizar el directorio `src`?: Sí

¿Te gustaría utilizar App Router?: Sí

¿Quieres personalizar el alias de importación predeterminado?: No

5. Después de responder todas las preguntas, se creará una nueva aplicación Next.js en la carpeta `ch16`. \$e  
La salida debería verse así:

```
● ➔ ~/D/Full-Stack-React-Projects ↵ main: > npx create-next-app@14.1.0
Need to install the following packages:
create-next-app@14.1.0
Ok to proceed? (y) y
✓ What is your project named? ... ch16
✓ Would you like to use TypeScript? ... No / Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... No / Yes
✓ Would you like to use `src/*` directory? ... No / Yes
✓ Would you like to use App Router? (recommended) ... No / Yes
✓ Would you like to customize the default import alias (@/*)? ... No / Yes
Creating a new Next.js app in /Users/dan/Development/Full-Stack-React-Projects/ch16.

Using npm.

Initializing project with template: app

Installing dependencies:
- react
- react-dom
- next

Installing devDependencies:
- eslint
- eslint-config-next

added 305 packages, and audited 306 packages in 3s

120 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
Success! Created ch16 at /Users/dan/Development/Full-Stack-React-Projects/ch16
```

Figura 16.1 – Creación de un nuevo proyecto Next.js

6. Abra la carpeta ch16 recién creada en VS Code.

7. En la nueva ventana de VS Code, abra una Terminal y ejecute el proyecto con el siguiente comando:

```
$ npm ejecuta dev
```

8. Abra <http://localhost:3000> en su navegador para ver la aplicación Next.js ejecutándose. Debería verse así:

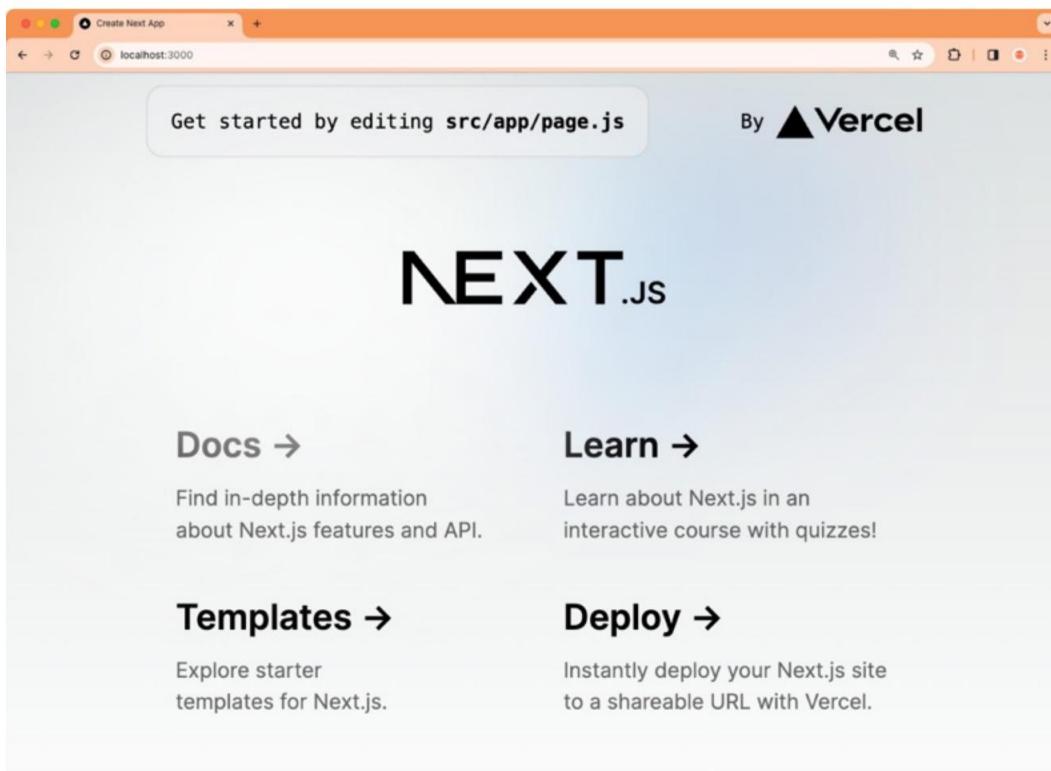


Figura 16.2 – Nuestra aplicación Next.js recién creada ejecutándose en el navegador

9. Lamentablemente, `create-next-app` no configura Prettier para nosotros, así que hágámoslo rápidamente. Ahora. Instale Prettier ejecutando el siguiente comando:

```
$ npm install --save-dev prettier@2.8.4 \
  eslint-config-prettier@8.6.0
```

10. Cree un nuevo archivo `.prettierrc.json` en la raíz del proyecto, con el siguiente contenido:

```
{
  "trailingComma": "todos", "tabWidth": 2, "printWidth": 80,
  "semi": false,
  "jsxSingleQuote": verdadero,
  "singleQuote": verdadero
}
```

11. Edite el archivo `.eslintrc.json` existente para extenderlo desde Prettier, de la siguiente manera:

```
{  
  "extiende": ["next/core-web-vitals", "más bonito"]  
}
```

12. Vaya a la configuración del espacio de trabajo de VS Code, cambie la configuración Editor: Formateador predeterminado a Prettier, y marque la casilla de verificación Editor: Formato al guardar.

¡Hemos creado con éxito un nuevo proyecto Next.js con ESLint y Prettier! Podríamos configurar Husky y lint-staged, como hicimos antes, pero por ahora, nos quedaremos con esta sencilla configuración. A continuación, aprenderemos más sobre cómo se estructuran las aplicaciones en Next.js.

## Presentamos el enrutador de aplicaciones

Next.js incluye un paradigma especial para estructurar aplicaciones llamado App Router. El App Router utiliza la estructura de carpetas de la carpeta `src/app/` para crear rutas para nuestras aplicaciones. La carpeta raíz (`/path`) es `src/app/`. Si queremos definir una ruta, como `/posts`, necesitamos crear una carpeta `src/app/posts/`. Para que esta carpeta sea una ruta válida, debemos incluir un archivo `page.js` dentro, que contiene el componente de página que se renderizará al acceder a esa ruta.

### Nota

Como alternativa, podemos colocar un archivo `route.js` en una carpeta para convertirlo en una ruta API en lugar de renderizar una página. Aprenderemos más sobre las rutas API en el Capítulo 18, Avanzado. Conceptos y optimizaciones de js.

Además, Next.js nos permite definir un archivo `layout.js`, que se usará como diseño para una ruta determinada. El componente de diseño acepta hijos, que pueden contener otros diseños o páginas. Su flexibilidad nos permite definir rutas anidadas con subdiseños.

Hay otros archivos especiales en el paradigma de App Router, como el archivo `error.js`, que se procesará cuando haya un error en la página, y el archivo `loading.js`, que se procesará mientras se carga la página (usando React Suspense).

Eche un vistazo al siguiente ejemplo de una estructura de carpeta con App Router:

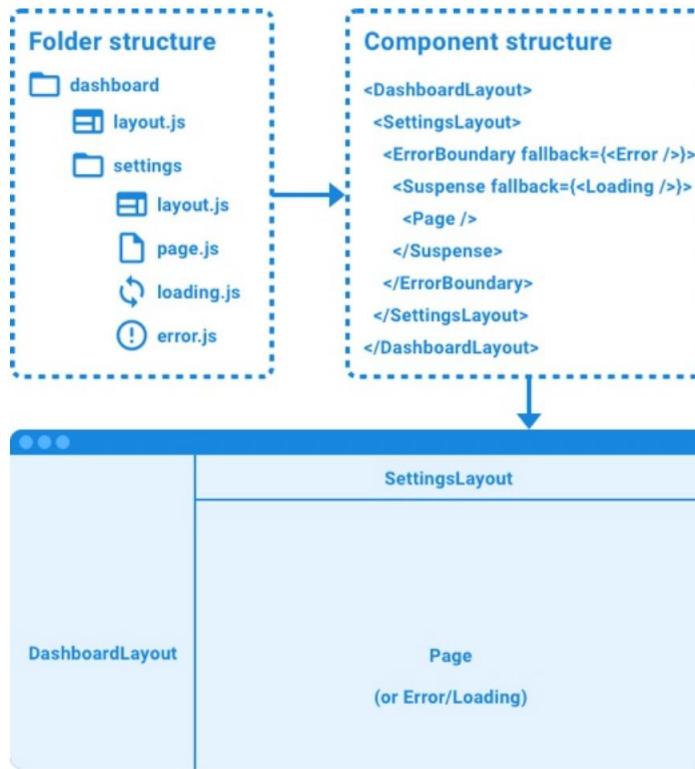


Figura 16.3 – Ejemplo de una estructura de carpetas con el App Router

En el ejemplo anterior, tenemos una ruta dashboard/settings/, definida por las carpetas dashboard y settings. La carpeta dashboard no tiene un archivo page.js, por lo que vamos a dashboard/. Dará como resultado un error 404 No encontrado. Sin embargo, la carpeta del panel tiene un archivo layout.js, que define el diseño principal del panel. La carpeta de configuración tiene otro diseño. js, que define el diseño de la página de configuración en el panel. También incluye un archivo page.js, que se renderiza al acceder a la ruta dashboard/settings/. Además, incluye un archivo loading.js, que se renderiza dentro del diseño de configuración mientras esta se carga. También contiene un archivo error.js, que se representa dentro del diseño de configuración si hay un error al cargar la página de configuración.

Como podemos ver, App Router facilita la implementación de casos de uso comunes, como rutas anidadas, diseños, errores y la carga de componentes. Comencemos ahora a definir la estructura de carpetas para nuestra aplicación de blog.

## Definición de la estructura de carpetas

Recapitulemos y refinemos la estructura de enrutamiento de la aplicación de blog de los capítulos anteriores:

- / – la página de índice de nuestro blog, que contiene una lista de publicaciones
- /login – la página de inicio de sesión para iniciar sesión en una cuenta existente
- /signup – la página de registro para crear una nueva cuenta
- /create – una página para crear una nueva publicación de blog (esta ruta es nueva)
- /posts/:id – una página para ver una sola publicación del blog

Todas estas páginas comparten un diseño común con una barra de navegación en la parte superior, que nos permite navegar entre las distintas páginas de nuestra aplicación.

Ahora vamos a crear esta estructura de enrutamiento como una estructura de carpeta en el enrutador de aplicaciones:

1. Elimine la carpeta src/app/ existente.
2. Crea una nueva carpeta llamada src/app/. Dentro de ella, crea un archivo src/app/layout.js con el siguiente contenido:

```
export const metadata = { title: 'Blog Full-
Stack de Next.js', description: 'Un blog sobre React y
Next.js',
}

exportar función predeterminada RootLayout({ children }) {
    devolver (
        <html lang="es"> <body>

            <main>{hijos}</main> </body> </html>

        )
}
```

El objeto de metadatos \$e es un objeto exportado especial en Next.js que se usa para proporcionar metaetiquetas, como las etiquetas <title> y <meta name="description">. La exportación predeterminada de archivos en App Router debe ser el componente que se debe representar para el diseño/página respectivo.

3. Cree un nuevo archivo src/app/page.js con el siguiente contenido de marcador de posición:

```
exportar función predeterminada HomePage() {
    volver a la <strong>página de inicio del blog</strong>
}
```

4. Cree una nueva carpeta llamada `src/app/login/`. Dentro de ella, cree un archivo `src/app/login/page.js` con el siguiente contenido de marcador de posición:

```
exportar función predeterminada LoginPage() { devolver
  <strong>Iniciar sesión</strong>
}
```

5. Cree una nueva carpeta llamada `src/app/signup/`. Dentro de ella, crea una carpeta llamada `src/app/signup/page.js` le con el siguiente contenido de marcador de posición:

```
exportar función predeterminada SignupPage() { devolver
  <strong>Signup</strong>
}
```

6. Cree una nueva carpeta llamada `src/app/create/`. Dentro de ella, crea una carpeta llamada `src/app/create/page.js` le con el siguiente contenido de marcador de posición:

```
exportar función predeterminada CreatePostPage() { devolver
  <strong>CreatePost</strong>
}
```

7. Cree una nueva carpeta llamada `src/app/posts/`. Dentro de ella, crea una nueva carpeta llamada `src/app/posts/[id]/` . \$is es una carpeta especial que contiene el id del parámetro de ruta, que podemos usar al renderizar la página.

8. Cree un nuevo archivo `src/app/posts/[id]/page.js` con el siguiente contenido de marcador de posición:

```
exportar función predeterminada ViewPostPage({ params }) { devolver
  <strong>ViewPost {params.id}</strong>
}
```

Como puedes ver, obtenemos la identificación del objeto de parámetros proporcionado por Next.js.

9. Si ya no se ejecuta, inicie el servidor de desarrollo Next.js con el siguiente comando:

```
$ npm ejecuta dev
```

10. Luego, ve a `http://localhost:3000/` (o actualiza la página) en tu navegador para ver la ruta principal funcionando.

Ve a las diferentes rutas, como `/login` y `/posts/123`, para ver cómo se renderizan las diferentes páginas y cómo funciona el parámetro de la ruta.

Ahora que hemos definido la estructura de carpetas para nuestro proyecto, continuemos creando componentes y páginas estáticas.

## Creación de componentes y páginas estáticas

Para los componentes de nuestro blog, podemos reutilizar gran parte del código que escribimos en capítulos anteriores, ya que no es muy diferente en Next.js que en React. Solo componentes específicos, como la barra de navegación, serán diferentes, ya que Next.js tiene su propio enrutador. Crearemos la mayoría de nuestros componentes en una carpeta separada, `src/components/`. Esta carpeta solo contendrá componentes de React que se puedan reutilizar en varias páginas. Todos los componentes de página y diseño seguirán estando en `src/app/`.

### Nota

En Next.js, también es posible ubicar los componentes regulares junto con los componentes de página y diseño, lo cual es recomendable en proyectos a gran escala para componentes que solo se usan en esas páginas específicas. En proyectos pequeños, esto no es tan importante, y podemos simplemente colocar todos nuestros componentes regulares en una carpeta separada para que sea más fácil distinguirlos de los componentes de página y diseño.

## Definición de componentes

Ahora comenzemos a crear los componentes para nuestra aplicación de blog:

1. Cree una nueva carpeta `src/components/`.
2. Cree un nuevo archivo `src/components/Login.jsx`. Dentro, defina un `<form>` con un campo de nombre de usuario, uno de contraseña y un botón de envío:

```
función de exportación Login() {  
    devolver (  
        <formulario>  
        <div>  
            <label htmlFor='username'>Nombre de usuario: </label>  
            <input type='text' name='nombre de usuario' id='nombre de usuario' />  
        </div>  
        <br />  
        <div>  
            <label htmlFor='password'>Contraseña: </label>  
            <input type='contraseña' name='contraseña' id='contraseña' />  
        </div>  
        <br />  
        <input type='submit' value='Iniciar sesión' />  
    </form>  
)  
}
```

**Nota**

Aquí utilizamos campos de entrada no controlados (es decir, no hay ganchos useState) a propósito, ya que no es necesario crear campos de entrada controlados para enviar formularios con Acciones de servidor, lo que aprenderemos en el próximo capítulo, Capítulo 17, Introducción a los componentes del servidor React.

Sin embargo, es importante definir adecuadamente la propiedad de nombre de los campos de entrada, ya que eso es lo que se utilizará para identificar el campo cuando se envíe el formulario.

3. De manera similar, cree un nuevo archivo src/components/Signup.jsx y defina un formulario con los mismos campos:

```
función de exportación Signup() { return (  
  
    <formulario>  
    <div>  
        <label htmlFor='username'>Nombre de usuario: </label> <input type='text'  
            name='username' id='username' />  
    </div>  
    <br />  
    <div>  
        <label htmlFor='password'>Contraseña: </label> <input type='password'  
            name='password' id='password' /> </div> <br /> <input type='submit' value='Registrarse' /> </  
    form>  
  
    )  
}
```

4. Cree un nuevo archivo src/components/CreatePost.jsx y defina un formulario con un campo de entrada de título obligatorio, un área de texto para definir el contenido y un botón de envío:

```
función de exportación CreatePost() { return (  
  
    <formulario>  
    <div>  
        <label htmlFor='title'>Título: </label> <input type='text' name='title'  
            id='title' required /> </div> <br /> <textarea name='contents' id='contents' /> <br /> <br />  
        <input  
            type='submit' value='Create' /> </form>
```

```

    )
}
```

5. Cree un nuevo archivo src/components/Post.jsx. Como mejora respecto a la estructura de los capítulos anteriores, el componente Post se usará en la lista de publicaciones y solo mostrará el título y el autor de una entrada de blog, con un enlace a la entrada completa.

```

importar PropTypes desde 'prop-types'

función de exportación Post({ _id, título, autor }) {
    devolver (
        <artículo>
            <h3>{título}</h3>
            <em>
                Escrito por <strong>{author.username}</strong> </em> </article>

        )
}
```

6. También necesitamos definir propTypes. En este caso, usaremos una estructura similar al resultado de una consulta a la base de datos, ya que podremos usar directamente los resultados de la base de datos cuando presentemos los componentes de React Server en el siguiente capítulo:

```

Post.propTypes = { _id:
    PropTypes.string.isRequired, título:
    PropTypes.string.isRequired, autor: PropTypes.shape({
        nombre de usuario: PropTypes.string.isRequired,
    }).isRequired, contenido:
    PropTypes.string,
}
```

7. Crea un nuevo archivo src/components/PostList.jsx. Aquí, reutilizaremos los propTypes del componente Post, así que también importaremos dicho componente:

```

importar { Fragmento } de 'react' importar PropTypes de
'prop-types' importar { Publicación } de './Post.jsx'
```

8. Luego, definimos el componente PostList, que representa cada publicación del blog con la Componente de publicación:

```

función de exportación PostList({ posts = [] }) {
    devolver (
        <div>
```

```
{posts.map((post) => ( <Clave del
    fragmento={'post-${post._id}'}>
    <Post _id={post._id} título={post.title} autor={post.author} /> <hr /> </Fragmento> ))} </div>

)
}
```

#### Nota

Se recomienda utilizar una identificación única para la propiedad clave, como una identificación de base de datos, de modo que React pueda realizar un seguimiento de los elementos que cambian en una lista.

9. Ahora definimos los propTypes para el componente PostList haciendo uso de la Post.propTypes existentes:

```
ListaDePostes.propTypes = {
  publicaciones:
    PropTypes.arrayOf(PropTypes.shape(Post.propTypes)).isRequired,
}

}
```

10. Por último, creamos un nuevo archivo src/components/FullPost.jsx, en el que mostramos el post completo con todo su contenido:

```
importar PropTypes desde 'prop-types'

función de exportación FullPost({ título, contenido, autor }) {
  devolver (
    <artículo>
      <h3>{título}</h3>
      <div>{contenido}</div> <br />

      <em>
        Escrito por <strong>{author.username}</strong> </em> </article>

    )
  }
}
```

11. En lugar de reutilizar los propTypes del componente Post, los estamos redimensionando aquí, porque el componente FullPost necesita diferentes props que el componente Post (no tiene el prop \_id, pero en su lugar tiene el prop contents):

```
FullPost.propTypes = {
  título: PropTypes.string.isRequired,
  autor: PropTypes.shape({
    nombre de usuario: PropTypes.string.isRequired,
  }).se requiere,
  contenido: PropTypes.string,
}
```

Ahora que hemos definido todos los componentes que vamos a necesitar para nuestra aplicación de blog, pasemos a definir correctamente los componentes de la página.

## Definición de páginas

Después de crear los componentes necesarios para nuestra aplicación de blog, reemplazamos los componentes de la página de marcador de posición con páginas adecuadas que representen los componentes correspondientes. Siga estos pasos para comenzar:

1. Edite src/app/login/page.js e importe el componente Login, luego renderícelo:

```
importar { Login } desde '@/components/Login'

exportar función predeterminada LoginPage() {
  regresar <Iniciar sesión />
}
```

### Nota

¿Recuerdan cuando configuramos Next.js y nos preguntaron si queríamos personalizar el alias de importación predeterminado? \$is import alias nos permite referenciar la carpeta src/ de nuestro proyecto, haciendo que nuestras importaciones sean absolutas en lugar de relativas. Por defecto, esto se hace usando el alias @. Así, ahora podemos importar desde '@/components/Login' para importar desde src/components/ Login.jsx le, en lugar de tener que importar desde ../../components/Login.jsx.

Las importaciones absolutas con alias de importación se vuelven especialmente útiles en proyectos grandes y facilitan la reestructuración de proyectos más adelante.

2. Edite src/app/signup/page.js y, de manera similar, importe y renderice el Componente de registro:

```
importar { Signup } desde '@/components/Signup'

exportar función predeterminada SignupPage() {
```

```
    regresar <Registrarse />
}
```

3. Repita el proceso editando el archivo src/app/create/page.js de la siguiente manera:

```
importar { CreatePost } desde '@/components/CreatePost'

exportar función predeterminada CreatePostPage() {
    devolver <CreatePost />
}
```

4. Ahora, edite el archivo src/app/posts/[id]/page.js e importe el componente FullPost:

```
importar { FullPost } desde '@/components/FullPost'
```

5. \$en, define un objeto de publicación de muestra:

```
exportar función predeterminada ViewPostPage({ params }) { const post = {

    título: 'Hola Next.js (${params.id})', contenido: 'Esto se obtendrá de
    la base de datos más tarde', autor: { nombre de usuario: 'Daniel Bug!' },

}
```

Para demostrar que el parámetro aún funciona, también ponemos el id en el título.

6. Representa el componente FullPost de la siguiente manera:

```
devolver (
    <Publicación completa
        título={post.title}
        contenido={post.contents}
        autor={post.author} />

)
```

7. Por último, edite src/app/page.js importando el componente PostList, creando un

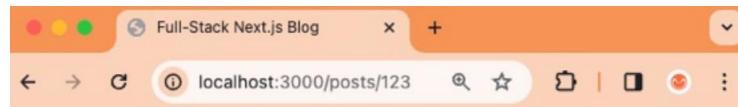
Ejemplo de matriz de publicaciones y representación del componente PostList:

```
importar { PostList } desde '@/components/PostList'

exportar función predeterminada HomePage() { const posts
    =
    [ { _id: '123', título: 'Hola Next.js', autor: { nombre de usuario: 'Daniel Bug!' } },
    ]
```

```
    devolver <PostList posts={posts} />
}
```

8. Vaya a <http://localhost:3000/posts/123> para ver cómo se renderiza el componente FullPost con el parámetro id en el título. Puede cambiar el id en la URL para ver cómo cambia el título. La siguiente captura de pantalla muestra cómo se renderiza el componente FullPost en la ruta /posts/123:



## Hello Next.js (123)

This will be fetched from the database later

*Written by Daniel Bugl*

Figura 16.4 – Representación del componente FullPost con un parámetro de ruta Next.js en el título

Después de definir con éxito todas nuestras páginas, aún necesitamos una forma de navegar entre ellas, así que continuemos agregando enlaces entre páginas.

### Agregar enlaces entre páginas

Como se mencionó anteriormente en este capítulo, Next.js ofrece su propia solución de enrutamiento: App Router. Las rutas se definen mediante la estructura de carpetas del directorio `src/app` y todas funcionan correctamente. Ahora solo queda añadir enlaces entre ellas. Para ello, necesitamos usar el componente `Link` de `next/link`. Sigue estos pasos para empezar a implementar una barra de navegación:

1. Creamos un nuevo archivo `src/components/Navigation.jsx`, donde importamos el enlace componente y `PropTypes`:

```
importar enlace desde 'next/link'
importar PropTypes desde 'prop-types'
```

2. Defina un componente UserBar, que se representará cuando el usuario inicie sesión y lo permita.

un usuario para acceder a la página Crear publicación y cerrar sesión:

```
función de exportación UserBar({ nombre de usuario }) {
    devolver (
        <formulario>
            <Link href='/create'>Crear publicación</Link> | Inició sesión como{'
        }
        <strong>{nombre de usuario}</strong> <button>Cerrar sesión</button> </
        form>
    )
}
```

UserBar.propTypes = { nombre  
 de usuario: PropTypes.string.isRequired,  
}

3. \$en, define un componente LoginSignupLinks, que se renderizará cuando el usuario aún no haya iniciado sesión. Proporciona enlaces a las páginas /login y /signup para que los usuarios puedan registrarse e iniciar sesión en nuestra aplicación.

```
función de exportación LoginSignupLinks() {
    devolver (
        <div>
            <Link href='/login'>Iniciar sesión</Link> | <Link href='/ signup'>Registrarse</
            Link> </div>

    )
}
```

4. A continuación, defina un componente de navegación que agregue un enlace a la página de inicio y luego represente condicionalmente el componente UserBar o el componente LoginSignupLinks, dependiendo de si el usuario ha iniciado sesión o no.

```
función de exportación Navegación({ nombre de usuario }) {
    devolver (
        <>
        Inicio

    )
}
```

```
Navegación.propTypes = {  
    nombre de usuario: PropTypes.string,  
}
```

5. Ahora solo necesitamos renderizar el componente de Navegación. Para asegurarnos de que aparezca en todas las páginas de nuestra aplicación de blog, lo colocaremos en el diseño raíz. Edite `src/app/layout.js` e importe el componente de Navegación:

```
importar {Navegación} desde '@/componentes/Navegación'
```

6. \$en, define un objeto de usuario de muestra para simular que un usuario ha iniciado sesión:

```
exportar función predeterminada RootLayout({ children }) { const usuario = { nombre  
    de usuario: 'dan' } }
```

7. Representa el componente de navegación de la siguiente manera:

```
devolver (  
    <html lang='es'> <cuerpo>  
  
        <navegación>  
            <Navegación nombredeusuario={usuario?.nombredeusuario} /> </  
            nav> <br />  
        <principal>{hijos}</principal> </cuerpo>  
    </html>  
  
)  
}
```

8. Aún necesitamos agregar un enlace desde una sola publicación de la lista a la página completa de la publicación. Editar `src/componentes/Post.jsx` e importe el componente Enlace:

```
importar enlace desde 'next/link'
```

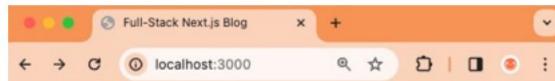
9. \$en, agrega un enlace al título, de la siguiente manera:

```
función de exportación Post({ _id, título, autor }) {  
    devolver (  
        <artículo>  
            <h3>  
                <Link href={`/posts/${_id}`}>{título}</Link> </h3>
```

10. Vaya a `http://localhost:3000` y verá que se muestra la barra de navegación.  
con el componente UserBar.

11. Haga clic en el enlace Crear publicación para ir a la página correspondiente, luego regrese usando el enlace Inicio . Además, intenta ir a la página de publicación completa haciendo clic en el título de la publicación del blog en la página de inicio.

La siguiente captura de pantalla muestra la página de inicio que se representa después de agregar la barra de navegación:



[Home](#) | [Create Post](#) | Logged in as **dan** [Logout](#)

## Hello Next.js

*Written by Daniel Bugl*

---

Figura 16.5 – ¡Nuestra aplicación de blog (estática) recreada en Next.js!

## Resumen

En este capítulo, aprendimos qué es Next.js y cómo puede ser útil para el desarrollo full-stack. Luego, configuramos un nuevo proyecto de Next.js y aprendimos sobre el paradigma de App Router. Finalmente, recreamos la aplicación de blog en Next.js creando componentes, páginas y una barra de navegación, utilizando el componente Link de Next.js para navegar entre las diferentes páginas de nuestra aplicación.

En el siguiente capítulo, Capítulo 17, Introducción a los Componentes de Servidor de React, aprenderemos a hacer que nuestra aplicación de blog sea interactiva mediante la creación de Componentes de Servidor de React. Estos componentes se ejecutan en el servidor y pueden, por ejemplo, ejecutar consultas a la base de datos. Además, aprenderemos sobre las Acciones de Servidor, que se utilizan para enviar formularios, como los de Inicio de Sesión, Registro y Creación de Publicaciones.



17

# Presentando React

## Componentes del servidor

Tras implementar nuestra aplicación de blog estática en Next.js, es hora de incorporarle interactividad. En lugar de usar el patrón tradicional de escribir un servidor backend independiente, del cual el frontend obtiene datos y al que realiza solicitudes, usaremos un nuevo patrón llamado Componentes de Servidor React. (RSCs). Este nuevo patrón nos permite acceder directamente a la base de datos desde los componentes de React mediante la ejecución de ciertos componentes de React (denominados componentes de servidor) únicamente en el servidor. Junto con las Acciones de Servidor (una forma de llamar a funciones en el servidor desde el cliente), este nuevo patrón nos permite desarrollar aplicaciones full-stack de forma fácil y rápida. En este capítulo, aprenderemos qué son los RSCs y las Acciones de Servidor, su importancia, sus ventajas y cómo implementarlos de forma correcta y segura.

En este capítulo cubriremos los siguientes temas principales:

- ¿Qué son las RSC?
- Agregar una capa de datos a nuestra aplicación Next.js
- Uso de RSC para obtener datos de la base de datos
- Usar acciones del servidor para registrarse, iniciar sesión y crear nuevas publicaciones

### 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 los capítulos 1 y 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-modernos-de-React-Full-Stack/árbol/principal/cap. 17>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/4hGZJRmZW6E>.

## ¿Qué son las RSC?

Hasta ahora, hemos estado utilizando la arquitectura tradicional de React, donde todos los componentes son componentes del cliente. Comenzamos con la renderización del lado del cliente. Sin embargo, esta renderización tiene algunas desventajas:

- El paquete de cliente JavaScript \$e debe descargarse del servidor antes de que el cliente pueda iniciarse renderizar cualquier cosa, retrasando la primera pintura con contenido (FCP) para el usuario.
- Los datos deben obtenerse del servidor (después de que se haya descargado y ejecutado todo el JavaScript) para mostrar algo significativo, lo que retrasa la primera pintura significativa (FMP) para el usuario.

La mayor parte de la carga recae en el cliente, incluso en páginas no interactivas, lo cual es especialmente problemático para clientes con procesadores lentos, como dispositivos móviles de gama baja o portátiles antiguos. Además, cargar una página renderizada del lado del cliente consume más batería.

- En ciertos casos, los datos se obtienen de forma secuencial (por ejemplo, primero se cargan las publicaciones y luego se resuelven los autores de cada publicación), lo que es especialmente un problema para conexiones lentas con alta latencia.

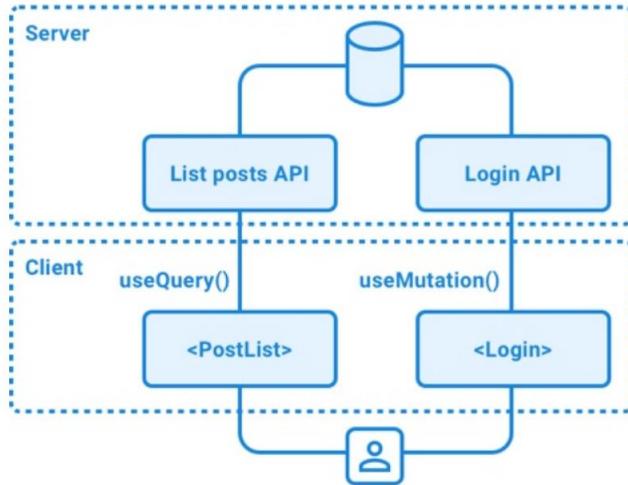
Para resolver estos problemas, se introdujo la representación del lado del servidor (SSR), pero todavía tiene una gran desventaja: la carga inicial de la página puede ser lenta debido a que todo se representa en el servidor. Esta ralentización se produce por las siguientes razones:

- Los datos deben obtenerse del servidor antes de poder mostrarlos.
- El paquete de cliente JavaScript \$e debe descargarse del servidor antes de que el cliente pueda hidratarse con él. La hidratación significa que la página está lista para que el usuario interactúe con ella. Para refresh sus conocimientos sobre el funcionamiento de la hidratación, consulte el Capítulo 7.
- La hidratación debe completarse en el cliente antes de poder interactuar con cualquier cosa.

Incluso cuando un componente de cliente se renderiza previamente en el servidor, su código se agrupará y se enviará al cliente para su hidratación. \$is significa que los componentes de cliente pueden ejecutarse tanto en el servidor (para SSR) como en el cliente, pero al menos deben poder ejecutarse en el cliente.

En una arquitectura React tradicional de pila completa con solo componentes cliente, si necesitábamos acceder al sistema de archivos del servidor o a una base de datos, debíamos escribir un backend independiente con Node.js y exponer una API (como una API REST). \$en, esta API se consultaba en los componentes cliente, por ejemplo, mediante TanStack Query. \$esas consultas también se pueden realizar en el servidor (como vimos en el Capítulo 7, Mejora del tiempo de carga mediante la representación del lado del servidor), pero deben ser al menos ejecutables en el cliente. \$esto significa que no podemos acceder directamente al sistema de archivos o a la base de datos desde un componente React, incluso si ese código pudiera ejecutarse en el servidor; se empaquetaría y enviaría al cliente, donde ejecutarlo no funcionaría (ni expondría información interna, como las credenciales, a la base de datos).

## Without React Server Components



## With React Server Components

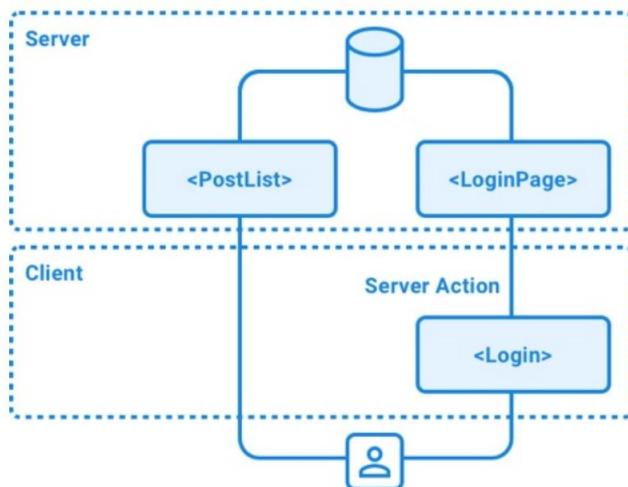


Figura 17.1 – La arquitectura de una aplicación full-stack sin y con RSC

React 18 introdujo una nueva función llamada RSC, que permite definir componentes que se ejecutarán exclusivamente en el servidor, enviando únicamente la salida al cliente. Los componentes del servidor pueden, por ejemplo, obtener datos de una base de datos o del sistema de archivos y luego renderizar componentes cliente interactivos, pasándoles esos datos como propiedades. Esta nueva función facilita la creación de una aplicación full-stack utilizando únicamente React, sin la sobrecarga que supone definir una API REST.

**Nota**

Todavía podría tener sentido definir API REST para ciertas aplicaciones, especialmente si el backend lo desarrolla otro equipo en un proyecto de mayor escala, o si lo consumen otros servicios y frontends.

Los RSC resuelven los problemas antes mencionados con la representación del lado del cliente y SSR al permitirnos ejecutar código exclusivamente en el servidor (¡sin necesidad de hidratación en el cliente!) y transmitir componentes de forma selectiva (de modo que no tenemos que esperar a que todo se pre-renderice antes de servir los componentes al cliente).

La siguiente figura compara la representación del lado del cliente (CSR) con SSR y RSC:

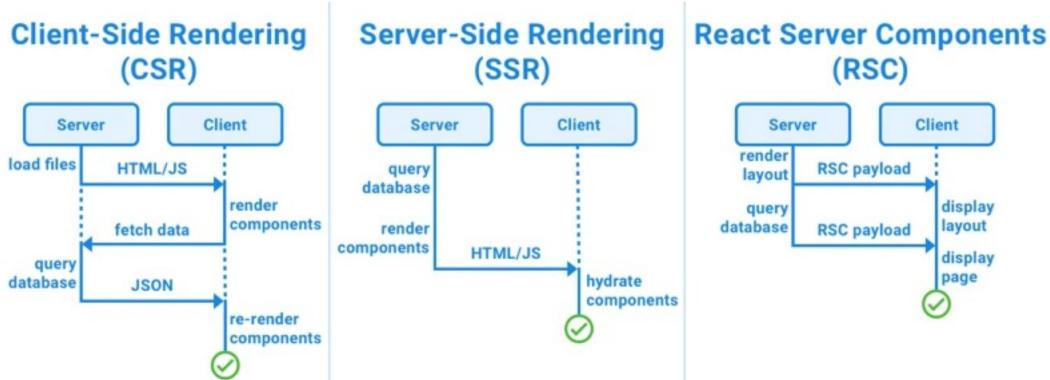


Figura 17.2 – Comparación entre CSR, SSR y RSC

Como puede ver, los RSC no solo son más rápidos en general (como resultado de menos viajes de ida y vuelta a través de la red), sino que también pueden mostrar el diseño de una aplicación inmediatamente mientras esperan que se carguen el resto de los componentes.

Resumamos las características más importantes de los RSC:

- Pueden ejecutarse con anticipación y se excluyen del paquete de JavaScript, lo que reduce el tamaño del paquete y mejora el rendimiento.
  - \$ey puede ejecutarse durante la compilación (lo que genera HTML estático) o en el %y al recibir una solicitud.
- Curiosamente, los componentes de servidor también pueden ejecutarse exclusivamente durante la compilación, lo que genera un paquete HTML estático. \$is puede ser útil para aplicaciones CMS o blogs personales compilados estáticamente. Los RSC también permiten una combinación: la caché inicial se prepara con una compilación estática y luego se revalida mediante Acciones de Servidor o Webhooks. Aprenderemos más sobre el almacenamiento en caché en el Capítulo 18, Conceptos y Optimizaciones Avanzadas de Next.js.

Pueden pasar datos serializables a los componentes del cliente. Además, los componentes del cliente pueden renderizarse en el servidor para mejorar aún más el rendimiento.

Dentro de un componente de servidor, se pueden pasar otros componentes de servidor como props a los componentes de cliente, lo que permite patrones de composición donde los componentes de servidor se integran en componentes de cliente interactivos. Sin embargo, todos los componentes importados dentro de componentes de cliente se considerarán componentes de cliente; ya no podrán ser componentes de servidor.

En frameworks como Next.js, por defecto, un componente React se considera un componente de servidor. Si queremos convertirlo en un componente cliente, debemos escribir la directiva "use client" al principio del archivo. Esto es necesario para poder añadir interactividad (detectores de eventos) o usar efectos de estado/ciclo de vida y APIs exclusivas para navegador.

#### Nota

La directiva "use client" define un límite de red entre los componentes del servidor y del cliente. Todos los datos enviados desde un componente servidor a un componente cliente se serializarán y enviarán por la red. Al usar la directiva "use client" en un archivo, todos los demás módulos que se importen, incluidos los componentes secundarios, se consideran parte del paquete cliente.

La siguiente figura proporciona una descripción general de cuándo utilizar un componente de servidor o un componente de cliente:

What do you need to do?	Server Component	Client Component
Fetch data	✓	✗
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server (reduces client-side JavaScript)	✓	✗
Add interactivity and event listeners (onClick, onChange, etc)	✗	✓
Use state and lifecycle effects (useState, useReducer, useEffect, etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use React class components	✗	✓

Figura 17.3 – Descripción general de cuándo utilizar componentes de servidor y componentes de cliente

En general, los RSC son una optimización de los componentes cliente. Podrías simplemente escribir "use client". en la parte superior de cada le y listo, ¡pero estarías dejando atrás todas las ventajas de los RSC! Por lo tanto, intente usar componentes de servidor siempre que sea posible, pero no dude en recurrir a definir un componente como cliente si resulta demasiado complicado dividirlo en partes del lado del servidor y del lado del cliente. Siempre se puede optimizar posteriormente.

Esta nueva forma de escribir aplicaciones React full-stack puede ser difícil de comprender en teoría, así que no dudes en volver a esta sección al final de este capítulo. Por ahora, implementaremos RSC en nuestra aplicación Next.js, ya que esto nos ayudará a comprender cómo funcionan los nuevos conceptos en la práctica. Primero, añadiremos una capa de datos a nuestra aplicación Next.js, que nos permitirá acceder a la base de datos desde los RSC más adelante.

## Agregar una capa de datos a nuestra aplicación Next.js

En la estructura tradicional del backend, teníamos la capa de base de datos, la capa de servicios y la capa de rutas. En una aplicación moderna de Next.js full-stack, no necesitamos la capa de rutas de nuestro backend, ya que podemos interactuar directamente con ella en los RSC. Por lo tanto, solo necesitamos la capa de base de datos y una capa de datos para proporcionar funciones que accedan a la base de datos. Teóricamente, podríamos acceder directamente a la base de datos en los RSC, pero se recomienda tener funciones específicas que accedan a ella de ciertas maneras. Definir estas funciones nos permite definir claramente qué datos son accesibles (y así evitar la filtración accidental de demasiada información). Además, son más reutilizables y facilitan las pruebas unitarias y la detección de posibles vulnerabilidades (por ejemplo, mediante una prueba de penetración) en la capa de datos.

Para resumir, hay tres enfoques principales de manejo de datos:

- API HTTP: Las usamos en capítulos anteriores para implementar nuestra aplicación de blog. Estas pueden ser útiles cuando equipos separados trabajan en el backend y el frontend. Por ello, este enfoque se recomienda para proyectos y organizaciones grandes existentes.

Capa de acceso a datos: \$is es el patrón que usaremos en esta sección. Se recomienda para proyectos nuevos que utilizan la arquitectura RSC, ya que facilita la implementación de proyectos full-stack al separar las preocupaciones sobre el manejo de datos (y todos los desafíos de seguridad asociados ) de la interfaz de usuario (mostrar los datos en componentes de React). Abordar cada problema por separado es más fácil de resolver y menos propenso a errores que gestionar la complejidad de ambos a la vez.

Acceso a datos a nivel de componente: \$is es un patrón en el que la base de datos se consulta directamente en los RSC. Este enfoque puede ser útil para el prototipado y el aprendizaje rápidos. Sin embargo, no debe utilizarse en una aplicación de producción debido a problemas de escalabilidad y la posible introducción de problemas de seguridad.

No se recomienda combinar estos enfoques, por lo que es mejor elegir uno y mantenerlo. En nuestro caso, optamos por el enfoque de "capa de acceso a datos", ya que es el más seguro para una arquitectura RSC moderna.

## Configuración de la conexión a la base de datos

Comencemos configurando los paquetes necesarios e inicializando una conexión de base de datos:

1. Copie la carpeta ch16 existente a una nueva carpeta ch17, de la siguiente manera:

```
$ cp -R ch16 ch17
```

2. Abra la carpeta ch17 en VS Code y abra una Terminal.

3. Vamos a utilizar un paquete llamado server-only para asegurarnos de que el código de la base de datos y de la capa de datos solo se ejecuten en el lado del servidor y no se importen accidentalmente en el cliente.

Instálelo de la siguiente manera:

```
$ npm install solo-servidor@0.0.1
```

4. También vamos a necesitar el paquete mongoose para conectarnos a la base de datos y crear Esquemas y modelos de base de datos. Ejecute el siguiente comando para instalarlo:

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

5. Cree una nueva carpeta src/db/.

6. Dentro de esta carpeta, crea un nuevo archivo src/db/init.js, en el que primero importamos el paquete exclusivo para servidor para asegurarnos de que el código solo se ejecute en el servidor:

```
importar 'solo servidor'
```

7. A continuación, importe mongoose:

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

8. Defina y exporte una función asíncrona para inicializar la base de datos:

```
exportar función asíncrona initDatabase() { const conexión =
    await mongoose.connect(process.env.
DATABASE_URL)
    conexión de retorno
}
```

9. Ahora, necesitamos definir DATABASE\_URL en un archivo .env. Para ello, cree un nuevo archivo .env en la raíz del proyecto y agregue la siguiente línea:

```
URL_DE_BASE_DE_DATOS=mongodb://localhost:27017/blog
```

Ahora que se ha configurado la conexión a la base de datos, podemos pasar a la creación de los modelos de base de datos.

## Creación de los modelos de base de datos

Ahora, crearemos modelos de base de datos para publicaciones y usuarios. Estos serán muy similares a los que creamos para nuestra aplicación de blog en capítulos anteriores. Sigue estos pasos para empezar a crear los modelos de base de datos:

1. Cree una nueva carpeta src/db/models/.
2. Dentro de él, creamos un nuevo archivo src/db/models/user.js, donde primero importamos el servidor.

Solo paquetes y mangosta:

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

3. Defina userSchema, que consta de un nombre de usuario único y obligatorio y una contraseña obligatoria:

```
const esquema de usuario = nuevo esquema ({  
    nombre de usuario: { tipo: String, requerido: verdadero, único: verdadero },  
    contraseña: { tipo: String, requerido: true },  
})
```

4. Creamos el modelo Mongoose si aún no se ha creado:

```
exportar const Usuario = mongoose.models.user ?? mangosta.  
modelo('usuario', esquemaDeUsuario)
```

### Nota

Es necesario devolver el modelo si ya existe y solo crear uno nuevo si no existe para evitar un problema OverwriteModelError, que ocurre cuando el modelo se importa (y, por lo tanto, se vuelve a definir) varias veces.

5. Crea un nuevo archivo src/db/models/post.js, donde primero importamos el archivo solo para servidor. y paquetes de mangosta:

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

6. Dene postSchema, que consta de un título y un autor obligatorios (que hacen referencia al modelo de usuario) y contenidos opcionales:

```
const postSchema = nuevo esquema(  
{  
    título: { tipo: Cadena, requerido: verdadero },  
    autor: { tipo: Schema.Types.ObjectId, ref: 'usuario', requerido: verdadero },  
  
    Contenido: Cadena,
```

```

},
{ marcas de tiempo: verdadero },
)

```

7. Creamos el modelo Mongoose si aún no se ha creado:

```

exportar const Post = mongoose.models.post ?? mangosta.
modelo('post', postSchema)

```

8. Cree un nuevo archivo src/db/models/index.js y vuelva a exportar los modelos:

```

importar 'solo servidor'
exportar * de './usuario'
exportar * desde './post'

```

Reexportamos los modelos de esta carpeta para garantizar que podamos, por ejemplo, cargar una publicación y resolver el autor consultando al usuario correspondiente. \$is requeriría definir el usuario

Modelo, aunque no se utiliza directamente. Para evitar problemas como estos, simplemente cargamos los modelos desde un archivo que define todos los modelos al importarlos.

Después de definir los modelos de base de datos, podemos definir las funciones de la capa de datos, que proporcionarán varias formas de acceder a la base de datos.

## Definición de funciones de la capa de datos

Ahora que tenemos una conexión de base de datos y esquemas, comenzemos a definir las funciones de la capa de datos que acceden a la base de datos.

### Definición de la capa de datos de publicaciones

Comenzaremos definiendo la capa de datos para las publicaciones. \$is nos permite acceder a todas las funciones relevantes para manejar publicaciones en nuestra aplicación:

1. Cree una nueva carpeta src/data/.
2. Dentro de él, creamos un nuevo archivo src/data/posts.js, donde importamos el archivo solo para servidor.  
paquete y el modelo Post:

```

importar 'solo servidor'
importar { Post } desde '@/db/models'

```

3. Defina una función createPost que tome un ID de usuario, un título y un contenido y cree

una nueva publicación:

```

exportar función asíncrona createPost(userId, { título, contenidos }) {
  const post = new Post({ autor: userId, título, contenidos })
  devolver esperar post.save()
}

```

4. A continuación, defina una función `listAllPosts`, que primero obtiene todas las publicaciones de la base de datos ordenadas por fecha de creación de manera descendente (mostrando primero las publicaciones más nuevas):

```
exportar función asíncrona listAllPosts() {
    devolver esperar Post.find({})
        .sort({ createdAt: 'descendente' })
```

5. \$en, debemos completar el campo de autor resolviendo el modelo de usuario y obteniendo de él el valor del nombre de usuario:

```
.populate('autor', 'nombre de usuario')
```

En Mongoose, la función `populate` funciona como una sentencia JOIN en SQL: toma el ID almacenado en el campo autor y luego verifica a qué modelo hace referencia el ID consultando el esquema de publicación. En el esquema de publicación, definimos que el campo autor hace referencia al usuario.

esquema, por lo que Mongoose consultará el modelo de usuario para la ID dada y devolverá un objeto de usuario. Al proporcionar el segundo argumento, especificamos que solo queremos obtener el valor del nombre de usuario del objeto de usuario (de todos modos, el ID siempre se devolverá). Esto se hace para evitar filtrar información interna, como la contraseña (en hash) de un usuario.

6. Después de completar los objetos de publicación, usamos `.lean()` para convertirlo en un objeto serializable simple.

Objeto JavaScript:

```
.inclinarse()
}
```

Tener un objeto serializable es necesario para poder pasar los datos desde un RSC a un componente normal del lado del cliente más tarde, ya que todos los datos que se pasan al cliente deben cruzar el límite de la red y, por lo tanto, deben ser serializables.

7. Por último, debemos definir una función `getPostById`, que encuentra una sola publicación por ID, rellena el campo de autor y convierte el resultado en un objeto JavaScript simple mediante `lean()`:

```
exportar función asíncrona getPostById(postId) {
    regreso en espera Post.findById(postId)
        .populate('autor', 'nombre de usuario')
        .inclinarse()
}
```

## Definición de la capa de datos para los usuarios

Ahora vamos a definir la capa de datos para los usuarios. Esto implicará crear un JWT para la autenticación. Nuevamente, gran parte del código será muy similar a lo que implementamos previamente para nuestra aplicación de blog. Siga estos pasos para comenzar a definir la capa de datos para los usuarios:

1. Instale `bcrypt` (para hacer hashes de la contraseña del usuario) y `jsonwebtoken` (para manejar JWT):

```
$ npm install bcrypt@5.1.1 jsonwebtoken@9.0.2
```

2. Crea un nuevo archivo src/data/users.js, donde importamos solo el servidor, bcrypt, jwt, y el modelo de Usuario:

```
importar 'solo servidor'  
importar bcrypt desde 'bcrypt'  
importar jwt desde 'jsonwebtoken'  
importar { Usuario } desde '@/db/models'
```

3. Defina una función createUser, donde hacemos un hash de la contraseña dada y luego creamos una nueva instancia del modelo Usuario y la guardamos:

```
exportar función asíncrona createUser({nombre de usuario, contraseña}) {  
    const hashedPassword = await bcrypt.hash(contraseña, 10)  
    const usuario = nuevo Usuario({nombre de usuario, contraseña: hashPassword})  
    devolver esperar usuario.save()  
}
```

4. A continuación, defina una función loginUser, que primero intenta encontrar un usuario con el nombre de usuario dado y genera un error si no se encuentra ningún usuario:

```
exportar función asíncrona loginUser({ nombre de usuario, contraseña }) {  
    const usuario = await Usuario.findOne({ nombre de usuario })  
    si (!usuario) {  
        arrojar nuevo Error('¡nombre de usuario inválido!')  
    }
}
```

#### Nota

Dependiendo de sus requisitos de seguridad, podría considerar no informar a un posible atacante de la existencia de un nombre de usuario y, en su lugar, devolver un mensaje genérico como "Nombre de usuario o contraseña no válidos". Sin embargo, en nuestro caso, se asume que los nombres de usuario son información pública, ya que cada usuario es autor del blog y sus nombres de usuario se publican con sus artículos.

5. \$en, usa bcrypt para comparar la contraseña proporcionada con la contraseña en hash de la base de datos y arroja un error si la contraseña no es válida:

```
const isPasswordCorrect = await bcrypt.compare(contraseña, usuario.  
contraseña)  
si (!esContraseñaCorrecta) {  
    lanzar nuevo Error('¡contraseña inválida!')  
}
```

6. Por último, genere, firme y devuelva un JWT:

```
const token = jwt.sign({ sub: usuario._id }, proceso.env.JWT_SECRET, { expira en: '24h' }) devuelve token

}
```

7. Ahora, definiremos una función para obtener la información del usuario (por ahora, solo obtendremos el nombre de usuario, pero esto podría ampliarse más adelante) a partir de un ID de usuario. Si el ID de usuario no existe, se generará un error:

```
exportar función asíncrona getUserInfoById(userId) { const usuario = await
User.findById(userId) if (!usuario) throw new Error('¡usuario no
encontrado!') return { nombre de usuario: usuario.nombredeusuario }

}
```

8. A continuación, defina una función para obtener el ID de usuario de un token, asegurándose de verificar la firma del token además de decodificar el JWT, mediante jwt.verify:

```
función de exportación getUserIdByToken(token) {
    si (!token) devuelve nulo
    const decodedToken = jwt.verify(token, proceso.env.JWT_SECRET) return decodedToken.sub

}
```

9. Finalmente, defina una función para obtener la información del usuario de un token combinando la Funciones getUserIdByToken y getUserInfoById:

```
exportar función asíncrona getUserInfoByToken(token) {
    const userId = getUserIdByToken(token) si (!userId) devuelve
    nulo
    constante usuario = await getUserInfoById(userId)
    usuario que regresa
}
```

10. Aún necesitamos definir la variable de entorno JWT\_SECRET para que nuestro código funcione. Edita el archivo .env y agrégalo de la siguiente manera:

```
JWT_SECRET=reemplazar con secreto aleatorio
```

**Nota**

\$is es una implementación muy básica de autenticación con Next.js. Para proyectos a gran escala, se recomienda considerar una solución de autenticación integral, como Auth.js (anteriormente next-auth), Auth0 o Supabase. Consulte la documentación de Next.js para obtener más información sobre la autenticación : <https://nextjs.org/docs/app/building-your-application/auth>.

Ahora que tenemos una capa de datos para acceder a la base de datos, podemos comenzar a implementar RSCs y Acciones de Servidor, que llamarán funciones de la capa de datos para acceder a la información de la base de datos y renderizar componentes React que la muestren, convirtiendo nuestra aplicación de blog estático en un blog completamente funcional.

## Uso de RSC para obtener datos de la base de datos

Como hemos aprendido, al usar Next.js, los componentes de React se consideran componentes de servidor por defecto, por lo que todos los componentes de página ya se ejecutan y renderizan en el servidor. Solo si necesitamos usar funciones exclusivas del cliente, como ganchos o campos de entrada, debemos convertir nuestros componentes en componentes de cliente mediante la directiva "use client". Los componentes que no requieren interacción del usuario se pueden mantener como componentes de servidor, y solo se renderizarán estáticamente y se servirán como HTML (codificado en la carga útil de RSC), sin hidratarse en el cliente. Para el cliente (el navegador), parecerá que estos componentes de React ni siquiera existen, ya que solo verá código HTML estático. El patrón \$is mejora considerablemente el rendimiento de nuestra aplicación web, ya que el cliente no necesita cargar JavaScript para renderizar dichos componentes. También reduce el tamaño del paquete, ya que se necesita menos código JavaScript para cargar nuestra aplicación web.

Ahora, implementemos RSC para obtener datos de la base de datos.

### Obtener una lista de publicaciones

Comenzaremos implementando la página de inicio, donde obtenemos y mostramos una lista de publicaciones:

1. Edite src/app/page.js e importe las funciones initDatabase y listAllPosts:

```
importar { initDatabase } desde '@/db/init'  
importar { listAllPosts } desde '@/data/posts'
```

2. Convierte el componente HomePage en una función asíncrona, lo que nos permite esperar hasta que

Los datos se obtienen antes de renderizar el componente:

```
exportar función asíncrona predeterminada HomePage() {
```

3. Reemplace la matriz de publicaciones de muestra con el siguiente código:

```
esperar initDatabase()  
const posts = await listaTodosLosPosts()
```

## Obteniendo una sola publicación

Ahora que podemos ver una lista de publicaciones, implementemos el proceso de obtener una sola publicación para ViewPostPage. Siga estos pasos para comenzar:

1. Edite src/app/posts/[id]/page.js e importe notFound, getPostById, y funciones initDatabase:

```
importar { notFound } desde 'next/navigation'  
importar { getPostById } desde '@/data/posts'  
importar { initDatabase } desde '@/db/init'
```

2. Convierte el componente de página en una función asíncrona:

```
exportar función asíncrona predeterminada ViewPostPage({params}) {
```

3. Reemplace el objeto de publicación de muestra con llamadas a initDatabase y getPostById:

```
esperar initDatabase()  
const post = await getPostById(params.id)
```

Si no se encontró ninguna publicación, llamamos a la función notFound, que arroja un NEXT\_NOT\_FOUND error y finaliza la representación del segmento de ruta:

```
si (!publicación) no se encontró()
```

Ahora necesitamos crear un archivo not-found.js para capturar el error y renderizar un componente diferente en su lugar.

4. Crea un nuevo archivo src/app/posts/[id]/not-found.js, donde mostramos un mensaje de “¡Publicación no encontrada!”, de la siguiente manera:

```
exportar función predeterminada ViewPostError() {  
    volver <strong>¡Publicación no encontrada!</strong>  
}
```

### Consejo

También podemos agregar un archivo app/not-found.js para gestionar las URL no coincidentes en toda la aplicación. Si los usuarios acceden a una ruta no definida por la aplicación, el componente definido en ella... En su lugar se representará le.

5. Además, podemos crear un componente de error que se renderizará ante cualquier error, como la imposibilidad de conectarse a la base de datos. Cree un nuevo componente `src/app/posts/[id]/error`.

is le, donde mostramos un mensaje de "Error al cargar la publicación!", de la siguiente manera:

'usar cliente'

```
exportar función predeterminada ViewPostError() {  
    devolver <strong>¡Error al cargar la publicación!</strong>  
}
```

Las páginas de error deben ser componentes del cliente, por eso agregamos una directiva 'use client'.

## Información

La razón por la que las páginas de error deben ser componentes cliente es que utilizan la función `ErrorBoundary` de React, que se implementa como componentes de clase (mediante `componentDidCatch`). Los componentes de clase de React no pueden ser componentes de servidor, por lo que debemos convertir la página de error en un componente cliente.

6. Aún necesitamos hacer un pequeño ajuste al componente Post, ya que `_id` ya no es una cadena, sino un objeto `ObjectId`. Editar `src/components/`

Post.jsx y cambia el tipo, de la siguiente manera:

```
Post.propTypes = {  
    id: PropTypes.object.isRequired,
```

7. ¡Asegúrese de que Docker y el contenedor MongoDB estén funcionando correctamente!

8. Ejecute el servidor de desarrollo de la siguiente manera:

\$ npm ejecuta dev

9. Vaya a <http://localhost:3000> y haga clic en una de las publicaciones de la lista; verá que se carga correctamente. Si una publicación no existe (por ejemplo, si cambia un solo dígito del ID), se mostrará el mensaje "¡Publicación no encontrada!". Si se produjo algún otro error (por ejemplo, un ID no válido), se mostrará el mensaje "¡Error al cargar la publicación!".

[Home](#) | [Create Post](#) | Logged in as **dan** | [Logout](#)

[Home](#) | [Create Post](#) | Logged in as **dan** [Logout](#)

[Home](#) | [Create Post](#) | Logged in as **dan** [Logout](#)

## Post not found!

Error while loading the post!

Full-Stack React Projects

Let's become full-stack developers!

*Written by Daniel Bugl*

Figura 17.4 – Muestra una publicación y los componentes no encontrados/de error

**Nota**

Si aún no hay publicaciones en su base de datos, cree una nueva publicación utilizando la aplicación de blog de los capítulos anteriores o espere hasta que implementemos la funcionalidad de creación de publicaciones utilizando `Next.js` al final de este capítulo.

Tras implementar RSC para obtener entradas, nuestra aplicación de blog ya está conectada a la base de datos. Sin embargo, por ahora solo puede mostrar entradas; el usuario aún no puede interactuar con la aplicación. Ahora, haremos que nuestra aplicación de blog sea interactiva añadiéndole Acciones de Servidor.

### Usar acciones del servidor para registrarse, iniciar sesión y crear nuevos publicaciones

Hasta ahora, solo hemos obtenido datos de la base de datos en el servidor y los hemos enviado al cliente, pero para la interactividad del usuario, necesitamos poder enviar datos del cliente al servidor. Para ello, React introdujo un patrón llamado Acciones del Servidor.

Las acciones de servidor son funciones que se ejecutan en el servidor, pero que pueden activarse desde el cliente, por ejemplo, al enviar un formulario. Esto se puede hacer incluso con JavaScript deshabilitado en el cliente, que enviará el formulario mediante el envío de formularios normal. Al habilitar JavaScript, el formulario se mejorará progresivamente y no requerirá una actualización completa para enviarse. Estas funciones se definen etiquetando funciones JavaScript normales con la directiva "use server" y luego importándolas a un componente cliente o pasándolas a un componente cliente mediante propiedades. Si bien las funciones JavaScript normales no se pueden pasar a componentes cliente (ya que no son serializables), las acciones de servidor sí.

**Nota**

Puedes definir un archivo completo para que contenga acciones de servidor añadiendo la directiva "use server" al principio. \$is indicará al empaquetador que todas las funciones de este archivo son acciones de servidor; no define los componentes dentro de él como componentes de servidor (para que algo se ejecute en el servidor, usa el paquete exclusivo para servidor, como se explicó). Despues, puedes importar funciones de dicho archivo en componentes de cliente.

En los componentes de cliente, podemos utilizar el gancho `useFormState`, que tiene una firma similar a `useState` pero nos permite ejecutar acciones del servidor (en el servidor) y obtener el resultado en el cliente. La firma del gancho `useFormState` se ve así:

```
const [estado, formAction] = useFormState(fn, initialState)
```

**Nota**

En la versión 19 de React, el gancho useFormState pasará a llamarse useStateAction. Consulta <https://react.dev/reference/react/useActionState> para obtener más información.

Como podemos ver, pasamos una función (Acción del Servidor) y un estado inicial. El gancho \$e devuelve el estado actual y una función formAction. El estado \$e se establece inicialmente en el estado inicial y se actualiza con el resultado de la Acción del Servidor tras llamar a esta función. En el lado del servidor, la firma de la Acción del Servidor se ve así:

```
función ejemploServerAction(previousState, formData) {  
  "usar servidor"  
  //...haz algo...  
}
```

Como podemos ver, la función Server Action acepta previousState (que inicialmente se establecerá en initialState desde el cliente) y un objeto formData (que es un objeto formData normal del estándar web XMLHttpRequest API). El objeto formData contiene toda la información enviada en los campos del formulario. Esto nos permite enviar fácilmente formularios para realizar una acción en el servidor y devolver el resultado al cliente.

Ahora, comencemos a usar Acciones de servidor para implementar la página de registro en nuestra aplicación de blog.

## Implementando la página de registro

Lo primero que debe hacer un usuario para interactuar con la aplicación del blog es registrarse, así que comencemos implementando esta función. Siga estos pasos para empezar:

1. Comenzamos implementando el componente cliente. Edite src/components/Signup.jsx y marcarlo como un componente de cliente, luego importar el gancho useFormState y PropTypes:

```
'usar cliente'  
importar { useFormState } desde 'react-dom'  
importar PropTypes desde 'prop-types'
```

2. El componente de registro \$e ahora debe aceptar una signupAction, que vamos a Dene en el lado del servidor más tarde:

```
función de exportación Signup({ signupAction }) {
```

3. Defina un gancho useFormState, que toma una acción del servidor y un estado inicial (en nuestro caso, un objeto vacío) y devuelve el estado actual y una acción:

```
const [estado, formAction] = useFormState(signupAction, {})
```

4. Ahora, podemos agregar acción a la etiqueta <form>, de la siguiente manera:

```
devolver (  
    < acción de formulario={AcciónDeFormulario}>
```

La acción \$e se llamará automáticamente al enviar el formulario. Como alternativa, se pueden llamar las acciones del servidor ejecutándolas como una función asíncrona normal; por ejemplo, llamando await formAction() dentro de una función del controlador onClick.

5. Además, podemos mostrar un mensaje de error debajo del botón “Registrarse” si recibimos un estado.

Mensaje de error del servidor:

```
<input type='submit' value='Registrarse' />  
{state.error ? <strong>Error al registrarse: {state.error}</strong>  
fuerte : nulo}  
</form>  
)  
}
```

6. No olvidemos definir propTypes para el componente Signup. signupAction es una función:

```
Registrarse.propTypes = {  
    signupAction: PropTypes.func.isRequired,  
}
```

7. Ahora podemos empezar a implementar la acción del servidor. Edite src/app/signup/page.js e importar la función de redirección de next/navigation (para navegar a la página de inicio de sesión después de registrarse exitosamente), así como las funciones createUser e initDatabase:

```
importar { redireccionar } desde 'siguiente/navegación'  
importar { createUser } desde '@/data/users'  
importar { initDatabase } desde '@/db/init'  
importar { Signup } desde '@/components/Signup'
```

8. \$en, fuera del componente SignupPage, define una nueva función asíncrona que acepte el estado anterior (en nuestro caso, este es el objeto vacío que definimos como estado inicial, por lo que podemos ignorarlo) y un objeto formData:

```
función asíncrona signupAction(prevState, formData) {
```

9. Necesitamos etiquetar la función con la directiva 'use server' para convertirla en una Acción de Servidor:

```
"utilizar servidor"
```

10. \$en, podemos inicializar la base de datos e intentar crear un usuario:

```
intentar {
    esperar initDatabase()
    esperar crearUsuario({
        nombre de usuario: formData.get('nombre de usuario'),
        contraseña: formData.get('contraseña'),
    })
}
```

Como puede ver, las Acciones de Servidor se basan en las API web existentes y utilizan la API FormData para el envío de formularios. Simplemente podemos llamar a .get() con el nombre del campo de entrada y contendrá el valor proporcionado en dicho campo.

11. Si hay un error, devolvemos el mensaje de error (que luego se mostrará en el Registro) componente cliente):

```
} atrapar (err) {
    devolver { error: err.message }
}
```

12. De lo contrario, si todo ha ido bien, redirigimos a la página de inicio de sesión:

```
redirigir('/login')
}
```

13. Luego de definir la Acción del Servidor, podemos pasársela al componente Signup, de la siguiente manera:

```
exportar función predeterminada SignupPage() {
    devolver <Signup signupAction={signupAction} />
}
```

Como alternativa, el componente cliente podría importar directamente la función signupAction desde un archivo. Siempre que la función tenga la directiva "use server", se ejecutará en el servidor. En este caso, solo necesitamos la función en esta página específica, por lo que tiene más sentido definirla en la página y pasársela al componente.

14. Ejecute el servidor de desarrollo de la siguiente manera:

```
$ npm ejecuta dev
```

15. Vaya a <http://localhost:3000/signup> e intente ingresar un nombre de usuario y una contraseña.

Debería funcionar correctamente y redirigirlo a la pantalla de inicio de sesión (el cambio es sutil, pero el botón de enviar cambia de Registrarse a Iniciar sesión).

16. Vaya a <http://localhost:3000/signup> nuevamente e intente ingresar el mismo nombre de usuario.

Recibirás el siguiente error:

Username:

Password:

**Sign Up Error signing up: E11000 duplicate key error collection: ch3.users index: username\_1 dup key: { username: "test42" }**

Figura 17.5 – Se muestra un error cuando el nombre de usuario ya existe

Por supuesto, este mensaje de error no es muy intuitivo, por lo que podríamos mejorarlo . Pero por ahora, este ejemplo es suficiente para mostrar cómo funcionan las Acciones del Servidor.

Como puede ver, los RSC y las Acciones de Servidor facilitan la implementación de funciones que interactúan con la base de datos . Además, todas las Acciones de Servidor enviadas mediante <form> funcionan incluso con JavaScript desactivado. ¡Pruébelo repitiendo los pasos 15 y 16 con JavaScript desactivado!

## Implementación de la página de inicio de sesión y manejo de JWT

Ahora que los usuarios pueden registrarse, necesitamos una forma de iniciar sesión. \$is también significa que necesitaremos implementar la funcionalidad para crear y almacenar JWT. Ahora que tenemos más control sobre la interacción servidor-cliente con Next.js, podemos almacenar el JWT en una cookie en lugar de en memoria. \$is significa que la sesión del usuario persistirá incluso al actualizar la página.

Comencemos a implementar la página de inicio de sesión y el manejo de JWT:

1. Comenzaremos implementando el componente cliente. Edite src/components/Login.jsx y convertirlo en un componente cliente:

'usar cliente'

2. \$en, importa el gancho useFormState y PropTypes:

```
importar { useFormState } desde 'react-dom'  
importar PropTypes desde 'prop-types'
```

3. Acepta loginAction como propiedad. Usaremos esto para definir el gancho useFormState:

```
función de exportación Login({ loginAction }) {  
  const [estado, formAction] = useFormState(loginAction, {})
```

4. Pase formAction, que se devolvió desde el gancho, al elemento <form>:

```
devolver (  
< acción de formulario={AcciónDeFormulario}>
```

5. Ahora, podemos mostrar posibles errores al final del componente:

```
<input type='submit' value='Iniciar sesión' /> {state.error ?  
  <strong>Error al iniciar sesión: {state.error}</strong> : null} </form>  
  
)  
}
```

6. Por último, define propTypes, de la siguiente manera:

```
Inicio de sesión.propTypes = {  
  loginAction: PropTypes.func.isRequired,  
}
```

7. Ahora podemos crear la acción de servidor loginAction. Edite src/app/login/page.js e importe las cookies y las funciones de redirección de Next.js, así como las funciones loginUser e initDatabase de nuestra capa de datos:

```
importar { cookies } desde 'next/headers' importar { redirect }  
desde 'next/navigation' importar { loginUser } desde '@/data/users'  
importar { initDatabase } desde '@/db/init' importar { Login } desde  
'@/components/Login'
```

8. Defina una nueva loginAction fuera del componente LoginPage, en la que intentamos

Para iniciar sesión con el nombre de usuario y contraseña proporcionados:

```
función asíncrona loginAction(prevState, formData) {  
  'utilizar servidor'  
  dejar token  
  intenta  
    { await initDatabase() token =  
      await loginUser({  
        nombre de usuario: formData.get('nombre de usuario'),  
        contraseña: formData.get('contraseña'), })
```

9. Si esto falla, devolvemos el mensaje de error:

```
} atrapar (err) {  
  devolver { error: err.message }  
}
```

10. De lo contrario, establecemos una cookie AUTH\_TOKEN con un vencimiento de 24 horas (el mismo tiempo de vencimiento como el JWT que creamos), y hacerlo seguro y httpOnly:

```
cookies().set({
    nombre: 'AUTH_TOKEN',
    valor: token,
    camino: '/',
    edad máxima: 60 * 60 * 24,
    seguro: verdadero,
    httpOnly: verdadero,
})
```

#### Nota

El atributo \$e httpOnly impide el acceso a las cookies por parte del cliente JavaScript, lo que reduce la posibilidad de ataques de scripts entre sitios en nuestra aplicación. El atributo \$e secure garantiza que la cookie se configure en la versión HTTPS del sitio web. Para mejorar la experiencia de desarrollo, esto no aplica a localhost.

11. Tras instalar la cookie, redirigimos a la página de inicio:

```
redirigir('/')
}
```

12. Finalmente, pasamos loginAction al componente Login:

```
exportar función predeterminada LoginPage() {
    devolver <Iniciar sesión loginAction={loginAction} />
}
```

13. Ve a <http://localhost:3000/login> e intenta ingresar un nombre de usuario不存在的; recibirás un error. \$en, intenta ingresar el mismo nombre de usuario y contraseña que usaste para registrarte . Debería funcionar correctamente y te redirigirá a la página de inicio.

#### Comprobación de si el usuario ha iniciado sesión

Quizás hayas notado que, después de que el usuario inicia sesión, la barra de navegación no cambia. Aún debemos verificar si el usuario ha iniciado sesión y ajustar la barra de navegación según corresponda. Hagámoslo ahora:

1. Edite src/app/layout.js e importe la función de cookies de Next.js y la función getUserInfoByToken de nuestra capa de datos:

```
importar { cookies } desde 'next/headers'
importar { getUserInfoByToken } desde '@/data/users'
importar {Navegación} desde '@/componentes/Navegación'
```

2. Convierte RootLayout en una función asíncrona:

```
exportar función asíncrona predeterminada RootLayout({ children }) {
```

3. Obtenga la cookie AUTH\_TOKEN y pase su valor a la función getUserInfoByToken para obtener el objeto de usuario, reemplazando el objeto de usuario de muestra que definimos anteriormente:

```
const token = cookies().get('AUTH_TOKEN') const usuario = await
getUserInfoByToken(token?.valor)
```

4. Si aún tiene abierta la página de inicio de antes, debería recargarse automáticamente y mostrar su nombre de usuario y el botón de cerrar sesión.

¡Ya estamos pasando user?.username al componente Navegación, así que eso es todo!

#### Implementación del cierre de sesión

Ahora que podemos mostrar una barra de navegación diferente cuando el usuario inicia sesión, por fin podemos ver el botón de cierre de sesión. Sin embargo, aún no funciona. Implementaremos el botón de cierre de sesión ahora:

1. Edite src/app/layout.js y defina una acción de servidor logoutAction fuera del componente RootLayout:

```
función asíncrona logoutAction() {
  'utilizar servidor'
```

2. Dentro de esta acción, simplemente eliminamos la cookie AUTH\_TOKEN:

```
cookies().delete('AUTH_TOKEN')
{}
```

3. Pase logoutAction al componente Navegación, de la siguiente manera:

```
<Navegación
  nombredeusuario={usuario?.nombredeusuario}
  accióndecierre ...
```

4. Edite src/components/Navigation.jsx y agregue logoutAction a UserBar y

el formulario de cierre de sesión:

```
función de exportación UserBar({nombre de usuario, cierre de sesión }) {
  devolver
    ( <form acción={logoutAction}>
```

5. Agregue la acción a propTypes del componente UserBar, de la siguiente manera:

```
UserBar.propTypes = { nombre de
    usuario: PropTypes.string.isRequired, acción de cierre de sesión:
        PropTypes.func.isRequired,
}
```

6. \$en, agrega logoutAction como propiedad al componente Navigation y pásalo al componente UserBar:

```
función de exportación Navigation({nombre de usuario, acción de cierre de sesión }) {
    devolver (
        <>
            <Link href="/">Inicio</Link> {nombre de usuario ?
                (
                    <Barra de usuario
                        nombre de usuario={nombre de
                            usuario} cierre de sesión={cierre de sesión} />

                ) : (
                    <Enlaces de registro de inicio de sesión /
                        > )} </>
            )
    }
}
```

7. Finalmente, cambie los propTypes del componente Navegación, de la siguiente manera:

```
Navegación.propTypes = {
    nombre de usuario: PropTypes.string, acción de
    cierre de sesión: PropTypes.func.isRequired,
}
```

8. Haga clic en el botón Cerrar sesión para ver cómo la barra de navegación vuelve a mostrar Iniciar sesión e iniciar sesión.

Enlaces ascendentes .

Ahora, nuestros usuarios finalmente pueden iniciar y cerrar sesión correctamente. Pasemos a implementar la creación de publicaciones.

## Implementando la creación de publicaciones

La última función que falta en nuestra aplicación de blog es la creación de entradas. Podemos usar Acciones de Servidor y un JWT para autenticar al usuario y permitirle crear una entrada. Sigue estos pasos para implementar la creación de entradas:

1. Ahora, comenzamos implementando la acción del servidor. Editamos src/app/create/page.js e importamos las funciones cookies, redirect, createPost, getUserIdByToken e initDatabase.

```
importar { cookies } desde 'next/headers' importar { redirect }
desde 'next/navigation' importar { createPost } desde '@/data/posts'
importar { getUserIdByToken } desde '@/data/users' importar
{ initDatabase } desde '@/db/init' importar { CreatePost } desde '@/components/
CreatePost'
```

2. Dentro del componente CreatePostPage, obtenga el token de la cookie:

```
exportar función predeterminada CreatePostPage() {
  token constante = cookies().get('AUTH_TOKEN')
```

3. Aún dentro del componente CreatePostPage, defina una Acción del Servidor:

```
función asíncrona createPostAction(formData) {
  'utilizar servidor'
```

Esta vez no usaremos el gancho useFormState porque no necesitamos gestionar el estado ni el resultado de la acción en el lado del cliente. Por lo tanto, la acción del servidor no tiene la firma (prevState, formData), sino una (formData).

4. Dentro de la Acción del Servidor, obtenemos el valor de userId del token, luego inicializamos la base de datos Conectarse y crear una nueva publicación:

```
constante userId = getUserIdByToken(token?.valor)
esperar initDatabase()
const post = await createPost(userId, { título: formData.get('título'),
  contenido: formData.get('contenido'), })
```

5. Por último, redirigimos a la página ViewPost de la publicación recién creada:

```
redirigir('/posts/${post._id}')
}
```

6. Si el usuario no ha iniciado sesión, ahora podemos mostrar un mensaje de error:

```
si (!token?.valor) {
    volver <strong>¡Debes iniciar sesión para crear publicaciones!</strong>
fuerte>
}
```

7. De lo contrario, renderizamos el componente CreatePost y le pasamos createPostAction:

```
devolver <CreatePost createPostAction={createPostAction} />
}
```

8. Ahora podemos ajustar el componente CreatePost. No es necesario convertirlo en un componente cliente esta vez, ya que no usaremos el gancho useState. Edite src/components/CreatePost.jsx e importar PropTypes:

```
importar PropTypes desde 'prop-types'
```

9. \$en, agrega createPostAction como propiedad y pásalo al elemento de formulario:

```
función de exportación CreatePost({ createPostAction }) {
    devolver (
        < acción del formulario={crearPostAcción}>
```

10. Finalmente, defina propTypes, de la siguiente manera:

```
CreatePost.propTypes = {
    createPostAction: PropTypes.func.isRequired,
}
```

11. Vaya a <http://localhost:3000>, inicie sesión de nuevo y haga clic en el enlace "Crear publicación". Introduzca un título y el contenido, y haga clic en el botón "Crear". Será redirigido a "ViewPost". ¡Página de la entrada del blog recién creada!

## Resumen

En este capítulo, aprendimos sobre los RSC, por qué se introdujeron, cuáles son sus ventajas y cómo se integran en nuestra arquitectura full-stack. Luego, aprendimos a implementar RSC de forma segura mediante la incorporación de una capa de datos en nuestra aplicación. Posteriormente, obtuvimos datos de nuestra base de datos y renderizamos componentes mediante RSC. Finalmente, aprendimos sobre las Acciones de Servidor y añadimos funciones interactivas a nuestra aplicación de blog.

¡Ahora nuestra aplicación de blog está nuevamente completamente funcional!

En el siguiente capítulo, Capítulo 18, Conceptos y Optimizaciones Avanzadas de Next.js, profundizaremos en el funcionamiento de Next.js y cómo podemos optimizar aún más nuestra aplicación al usarlo. Aprenderemos sobre el almacenamiento en caché, la optimización de imágenes y fuentes, y cómo definir metadatos para la optimización SEO.

## Conceptos y optimizaciones avanzadas de Next.js

Ahora que conocemos las características esenciales de Next.js y los Componentes de Servidor React (RSC), profundicemos en el framework Next.js. En este capítulo, aprenderemos cómo funciona el almacenamiento en caché en Next.js y cómo se puede usar para optimizar nuestras aplicaciones. También aprenderemos a implementar rutas de API en Next.js. Además, aprenderemos a optimizar una aplicación Next.js para motores de búsqueda y redes sociales añadiendo metadatos. Finalmente, aprenderemos a cargar imágenes y fuentes de forma óptima en Next.js.

En este capítulo cubriremos los siguientes temas principales:

- Definición de rutas API en Next.js
- Almacenamiento en caché en Next.js
- Optimización de motores de búsqueda (SEO) con Next.js
- Carga optimizada de imágenes y fuentes en Next.js

### 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 2.

Puedes encontrar el código de este capítulo en GitHub: <https://github.com/PacktPublishing/Proyectos-modernos-de-React-Full-Stack/árbol/principal/cap. 18>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/jzCRoJPG0G0>.

## Definición de rutas API en Next.js

En el capítulo anterior, usamos RSC para acceder a nuestra base de datos mediante una capa de datos; ¡no se necesitaron rutas de API para ello! Sin embargo, a veces, aún tiene sentido exponer una API externa. Por ejemplo, podríamos permitir que aplicaciones de terceros consulten entradas de blog. Afortunadamente, Next.js también incluye una función para definir rutas de API, llamada "Manejadores de Ruta".

Los manejadores de rutas también se definen dentro del directorio `src/app/`, pero en un archivo `route.js` en lugar de `page.js` (una ruta solo puede ser una ruta o una página, por lo que solo uno de estos archivos debe ubicarse dentro de una carpeta). En lugar de exportar un componente de página, necesitamos exportar allí funciones que gestionen varios tipos de solicitudes. Por ejemplo, para gestionar una solicitud GET, debemos definir y exportar la siguiente función:

```
exportar función asíncrona GET() {
```

Next.js admite los siguientes métodos HTTP para controladores de ruta: GET, POST, PUT, PATCH, DELETE, HEAD y OPTIONS. Para los métodos no compatibles, Next.js devolverá una respuesta 405 "Método no permitido".

Next.js admite la solicitud nativa (<https://developer.mozilla.org/en-US/docs/Web/API/Solicitud>) y Respuesta (<https://developer.mozilla.org/en-US/docs/Web/API/Respuesta>) API web pero las extiende a `NextRequest` y `NextResponse` API que facilitan la gestión de cookies y encabezados. En el capítulo anterior, usamos la función `cookies()` de Next.js para crear, obtener y eliminar fácilmente una cookie para el JWT. \$e headers() Esta función facilita la obtención de encabezados de una solicitud. Estas funciones se pueden utilizar de la misma manera en RSC y controladores de ruta.

### Creación de una ruta API para listar publicaciones de blog

Comencemos definiendo una ruta API para listar publicaciones de blog:

1. Copie la carpeta ch17 existente a una nueva carpeta ch18, de la siguiente manera:

```
$ cp -R ch17 ch18
```

2. Abra la carpeta ch18 en VS Code.
3. Para que las rutas de la API sean más fáciles de distinguir de las páginas de nuestra aplicación, cree un nuevo `src/app/` carpeta `api/`.
4. Dentro de la carpeta `src/app/api/`, cree una nueva carpeta `src/app/api/v1/` para asegurarse de que nuestra API esté versionada para posibles cambios posteriores en la API.
5. A continuación, cree una carpeta `src/app/api/v1/posts/` para la ruta `/api/v1/posts`.

6. Creamos un nuevo archivo `src/app/api/posts/route.js`, donde importamos el `initDatabase` función y la función `listAllPosts` de la capa de datos:

```
importar { initDatabase } desde '@/db/init' importar { listAllPosts }  
desde '@/data/posts'
```

7. \$en, define y exporta una función GET. \$is manejará las solicitudes HTTP GET a la ruta `/api/v1/posts`:

```
exportar función asíncrona GET() {
```

8. Dentro de ella, debemos inicializar la base de datos y obtener una lista de todas las publicaciones:

```
esperar initDatabase()  
const posts = await listaTodosLosPosts()
```

9. Utilice la API web de respuesta para generar una respuesta JSON:

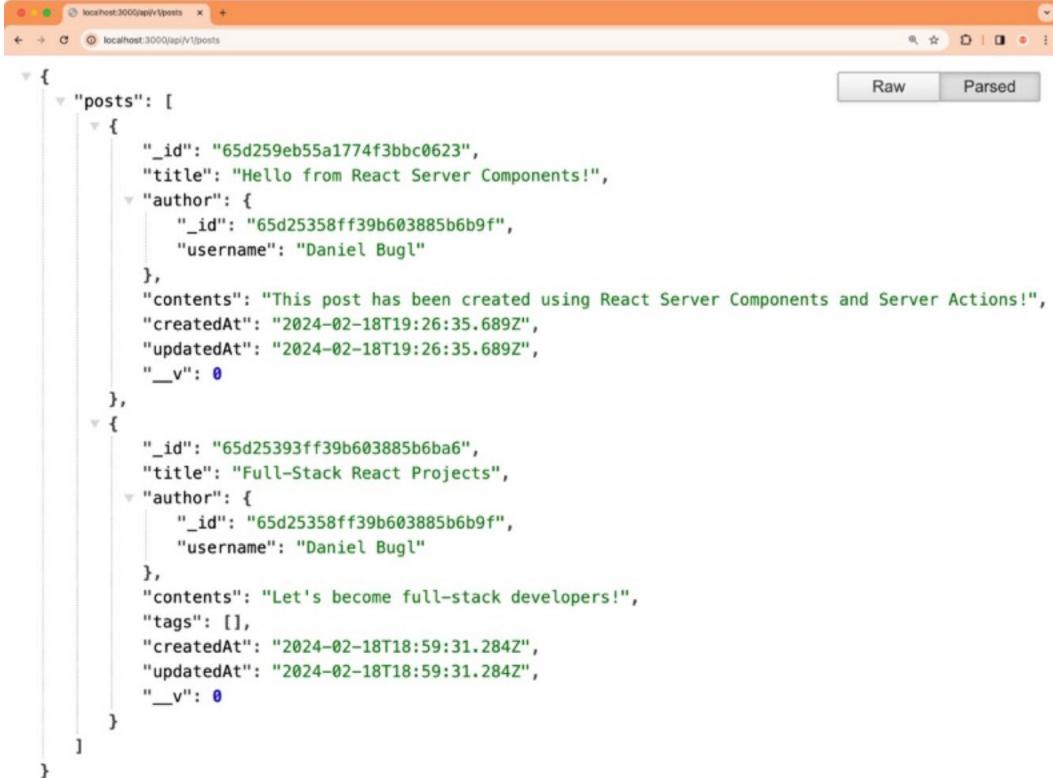
```
devolver Response.json({ publicaciones })  
}
```

10. ¡Asegúrese de que Docker y el contenedor MongoDB estén funcionando correctamente!

11. Inicie la aplicación Next.js de la siguiente manera:

```
$ npm ejecuta dev
```

12. Ahora, vaya a `http://localhost:3000/api/v1/posts` para ver las publicaciones que se devuelven como JSON, como se muestra en la siguiente figura:



The screenshot shows a browser window with two tabs open. The active tab is titled 'localhost:3000/api/v1/posts' and displays a JSON object. The JSON structure is as follows:

```
{ "posts": [ { "_id": "65d259eb55a1774f3bbc0623", "title": "Hello from React Server Components!", "author": { "_id": "65d25358ff39b603885b6b9f", "username": "Daniel Bugl" }, "contents": "This post has been created using React Server Components and Server Actions!", "createdAt": "2024-02-18T19:26:35.689Z", "updatedAt": "2024-02-18T19:26:35.689Z", "__v": 0 }, { "_id": "65d25393ff39b603885b6ba6", "title": "Full-Stack React Projects", "author": { "_id": "65d25358ff39b603885b6b9f", "username": "Daniel Bugl" }, "contents": "Let's become full-stack developers!", "tags": [], "createdAt": "2024-02-18T18:59:31.284Z", "updatedAt": "2024-02-18T18:59:31.284Z", "__v": 0 } ] }
```

Figura 18.1 – Respuesta JSON con publicaciones generadas desde el controlador de ruta Next.js

¡Ahora, las aplicaciones de terceros también pueden obtener las publicaciones a través de nuestra API! Continúe aprendiendo más sobre el almacenamiento en caché en Next.js.

### Almacenamiento en caché en Next.js

Hasta ahora, siempre hemos usado Next.js en modo de desarrollo. En este modo, la mayor parte del almacenamiento en caché de Next.js está desactivado para facilitar el desarrollo de nuestras aplicaciones con recarga en caliente y datos siempre actualizados. Sin embargo, al pasar al modo de producción, el renderizado estático y el almacenamiento en caché se activan por defecto. El renderizado estático significa que si una página solo contiene componentes estáticos (como una página "Sobre nosotros" o "Aviso legal", que solo contiene contenido estático), se renderizará estáticamente y se servirá como HTML o como texto/JSON estático para las rutas. Además, Next.js intentará almacenar en caché los datos y los componentes renderizados del lado del servidor tanto como sea posible para mantener el rendimiento de la aplicación.

Next.js tiene cuatro tipos principales de caché:

- Caché de datos: un caché del lado del servidor para almacenar datos en solicitudes e implementaciones de usuarios. Es persistente pero puede revalidarse.
- Memorización de solicitudes: un caché del lado del servidor para los valores de retorno de las funciones si se las llama varias veces en una sola solicitud.
- Caché de ruta completa: un caché del lado del servidor de las rutas de Next.js. Su caché es persistente pero puede ser revalidado.
- Caché de enrutador: un caché de rutas del lado del cliente para reducir las solicitudes del servidor en la navegación, para una sola sesión de usuario o en función del tiempo.

Los dos primeros tipos de caché (caché de datos y memorización de solicitudes) se aplican principalmente al uso de la función `fetch()` en el servidor para, por ejemplo, extraer datos de una API de terceros. Sin embargo, recientemente, también es posible usar estos dos tipos de caché para cualquier función, integrándolos con la función `unstable_cache()`. A pesar de su nombre, esta función ya se puede usar de forma segura en producción. Se denomina "inestable" porque la API puede cambiar y requerir cambios de código con el lanzamiento de nuevas versiones de Next.js. Consulte [https://nextjs.org/docs/app/api-reference/functions/unstable\\_cache](https://nextjs.org/docs/app/api-reference/functions/unstable_cache) para obtener más información.

#### Nota

Como alternativa, se podría usar la función `cache()` de React para memorizar los valores de retorno de las funciones, pero la función `unstable_cache()` de Next.js es más flexible, lo que permite revalidar dinámicamente la caché mediante una ruta o etiqueta. Más adelante en esta sección, aprenderemos más sobre la revalidación de la caché .

La caché de ruta completa es una caché adicional que garantiza que, cuando los datos no cambian, ni siquiera sea necesario volver a renderizar las páginas en el servidor, de modo que Next.js pueda devolver directamente el HTML estático pre-renderizado y la carga útil RSC. Sin embargo, invalidar la caché de datos también invalidará la caché de ruta completa correspondiente y activará una nueva renderización.

El caché del enrutador \$e es un caché del lado del cliente y se utiliza principalmente cuando el usuario navega entre páginas, lo que nos permite mostrar instantáneamente las páginas que ya ha visitado sin tener que buscarlas nuevamente en el servidor.

Además, si Next.js detecta que una página o ruta solo contiene contenido estático, lo prerenderizará y lo almacenará como tal. El contenido estático ya no se puede revalidar, por lo que debemos tener cuidado y asegurarnos de que Next.js considere "dinámico" todo el contenido dinámico de nuestras aplicaciones y no lo detecte accidentalmente como tal.

#### Nota

En este libro, este proceso se denomina renderizado estático. Sin embargo, en otros recursos, también se le puede llamar «optimización estática automática» o «generación de sitios estáticos».

Next.js optará por no realizar la representación estática y considerará una página o ruta dinámica en los siguientes casos:

- Al utilizar una función dinámica, como cookies(), headers() o searchParams
- Al configurar export const dynamic = 'force-dynamic' o export const revalidate = 0
- Cuando un controlador de ruta maneja una solicitud que no es GET

Para obtener información más detallada sobre los diferentes tipos de almacenamiento en caché, consulte la documentación de Next.js sobre almacenamiento en caché: <https://nextjs.org/docs/app/building-your-application/caching>.

Ahora, exploremos cómo funciona la representación estática en la práctica observando cómo se comporta nuestra ruta en una compilación de producción de nuestra aplicación.

## Explorando la representación estática en rutas API

En este capítulo, implementamos un controlador de rutas para obtener entradas de blog. Ahora, exploremos cómo se comporta esta ruta en los modos de desarrollo y producción:

1. Edite src/app/api/v1/posts/route.js y agregue un valor currentTime con Fecha.

now() a la respuesta, de la siguiente manera:

```
devolver Response.json({ publicaciones, horaActual: Fecha.ahora() })
```

2. Actualice la página en <http://localhost:3000/api/v1/posts> un par de veces; verá que currentTime es siempre la última marca de tiempo.

3. Salga del servidor de desarrollo Next.js utilizando Ctrl + C.

4. Cree la aplicación Next.js para producción e iníciela de la siguiente manera:

```
$ npm ejecuta la compilación  
$ npm start
```

5. Actualice la página en <http://localhost:3000/api/v1/posts> un par de veces. ¡Ahora, currentTime no cambia en absoluto!

Incluso si reiniciamos el servidor Next.js, currentTime...

Todavía no cambia. La respuesta de la ruta GET /api/v1/posts se procesa estáticamente durante el tiempo de compilación.

La representación estática funciona de forma similar para rutas y páginas, por lo que las páginas también se representarán estáticamente por defecto. \$is significa que los RSC no requieren un servidor; también pueden ejecutarse durante la compilación. Solo necesitamos un servidor Node.js si queremos tener páginas/rutas dinámicas. \$is significa que podríamos, por ejemplo, crear un blog o sitio web en Next.js y exportar un paquete estático, lo que nos permite alojarlo en un servidor web sencillo.

**Nota**

La exportación de una aplicación Next.js como un paquete estático se puede lograr especificando la opción `output: 'export'` en el archivo `next.config.js`.

Curiosamente, si creamos una nueva entrada de blog, nuestra página de inicio se actualiza. Sin embargo, esto solo ocurre porque `RootLayout` usa `cookies()` para comprobar si el usuario ha iniciado sesión, lo que hace que todas las páginas de nuestra aplicación de blog sean dinámicas (y, por lo tanto, no se rendericen estáticamente). Esto también se puede ver en la salida de `npm run build`:

```
● → ~/D/F/ch18 ↵ main± > npm run build

> ch16@0.1.0 build
> next build

  ▲ Next.js 14.1.0
    - Environments: .env

    Creating an optimized production build ...
    ✓ Compiled successfully
    ✓ Linting and checking validity of types
    ✓ Collecting page data
    ✓ Generating static pages (8/8)
    ✓ Collecting build traces
    ✓ Finalizing page optimization

  Route (app)          Size   First Load JS
  └─ λ /               178 B   91.1 kB
      └─ λ /_not-found  885 B   85.1 kB
          └─ o /api/v1/posts  0 B   0 B
          └─ λ /create     143 B   84.3 kB
          └─ λ /login       875 B   85 kB
          └─ λ /posts/[id]  144 B   84.3 kB
          └─ λ /signup      875 B   85 kB
  + First Load JS shared by all  84.2 kB
      └─ chunks/69-db30a0e38e2695f2.js  28.9 kB
      └─ chunks/fd9d1056-cc48c28d170fddc2.js  53.4 kB
          └─ other shared chunks (total)  1.87 kB

  o (Static)  prerendered as static content
  λ (Dynamic) server-rendered on demand using Node.js
```

Figura 18.2 – Visualización de qué rutas se representan estática y dinámicamente en la salida de la compilación

Como se puede ver en la Figura 18.2, la ruta `/api/v1/posts` se “pre-renderiza como contenido estático”, mientras que todas las demás rutas se “renderizan en el servidor a pedido usando Node.js”.

**Nota**

Si quisieramos renderizar estáticamente algunas páginas de nuestro blog, tendríamos que asegurarnos de que la barra de usuario no sea visible en ellas. Por ejemplo, podríamos crear un grupo de rutas (<https://nextjs.org/docs/app/building-your-application/routing/route-groups>) para todas las páginas que tienen una barra de usuario, con un diseño separado que contiene la barra de usuario. Sin embargo, podemos eliminar la barra de usuario del diseño raíz. De esta manera, podríamos crear, por ejemplo, una página Acerca de que se represente estáticamente mientras mantiene dinámico el resto del blog.

Como hemos visto, en Next.js, las páginas y rutas se renderizan estáticamente por defecto (si es posible). Sin embargo, en el caso de nuestra ruta de API, ¡esto no es lo que buscamos! Queremos poder obtener entradas dinámicamente de la API. El renderizado estático y el almacenamiento en caché en Next.js pueden ser confusos al comenzar a desarrollar aplicaciones, pero se convierten en una herramienta poderosa para mantener nuestras aplicaciones optimizadas.

Ahora, aprendamos cómo manejar adecuadamente el caché para hacer que nuestras páginas y rutas sean dinámicas cuando sea necesario y mantenerlas en caché siempre que sea posible.

## Dinamizar la ruta

Para que la ruta sea dinámica, necesitamos establecer la exportación `const dynamic = 'force-dynamic'` %ag en él. Sigue estos pasos:

1. Edite `src/app/api/v1/posts/route.js` y agregue el siguiente código:

```
export const dynamic = 'fuerza-dinámica'
```

2. Salga del servidor Next.js que esté ejecutándose actualmente.

3. Cree la aplicación Next.js para producción e iníciela de la siguiente manera:

```
$ npm ejecuta la compilación  
$ npm start
```

4. Actualice la página `http://localhost:3000/api/v1/posts` un par de veces. Ahora,  
¡La ruta API se comporta de la misma manera que en el servidor de desarrollo!

Lamentablemente, ahora hemos desactivado por completo la caché, por lo que no obtenemos ninguno de sus beneficios. A continuación, aprenderemos a activar la caché para funciones específicas.

### Funciones de almacenamiento en caché en la capa de datos

Para almacenar en caché funciones de nuestra capa de datos, podemos usar la función `unstable_cache()` de Next.js. La función `$e unstable_cache(fetchData, keyParts, options)` acepta tres argumentos:

- `fetchData`: el primer argumento es la función que se llamará. La función también puede tener argumentos.

- **keyParts:** El segundo argumento es una matriz de claves únicas que identifican la función en la caché. Los argumentos que se pasan a la función en el primer argumento también se añadirán automáticamente a esta matriz.
- **opciones:** el tercer argumento es un objeto que contiene opciones para el caché, donde podemos especificar etiquetas para revalidar el caché más tarde y un tiempo de espera de revalidación para revalidar automáticamente el caché después de una cierta cantidad de segundos.

Ahora, habilitemos esta caché para todas las funciones donde sea necesario. Siga estos pasos para comenzar:

1. Edite src/data/posts.js e importe la función `unstable_cache()`, creando un alias como `caché()`:

```
importar { unstable_cache como caché } desde 'next/cache'
```

2. Envuelva la función `listAllPosts` con `caché()`, de la siguiente manera:

```
exportar const listAllPosts = caché(  
    función asíncrona listAllPosts() {  
        devolver esperar Post.find({})  
            .sort({ createdAt: 'descendente' })  
            .populate('autor', 'nombre de usuario')  
            .inclinarse()  
    },  
    ['publicaciones', 'listaTodasLasPublicaciones'],  
    { etiquetas: ['publicaciones'] },  
)
```

Como clave de caché, definimos una matriz que contiene el nombre del archivo (`publicaciones`) y el nombre de la función (`listaTodasLasPublicaciones`) para identificar de forma única la función en nuestra capa de datos. Además, añadimos una etiqueta de `publicaciones`, que usaremos más adelante para revalidar la caché al crear nuevas publicaciones.

3. A continuación, envuelva la función `getPostById`:

```
exportar const getPostById = caché(  
    función asíncrona getPostById(postId) {  
        devolver esperar Post.findById(postId).populate('autor', 'nombre de usuario').lean()  
    },  
    ['publicaciones', 'getPostById'],  
)
```

4. Puede que notes que ahora hay un error al obtener publicaciones porque la caché serializa el ObjectId de MongoDB en una cadena. Edita src/components/Post.jsx y ajusta propType como se indica a continuación:

```
Post.propTypes = { _id:  
  PropTypes.string.isRequired,
```

5. Edite src/data/users.js e importe unstable\_cache allí:

```
importar { unstable_cache como caché } desde 'next/cache'
```

6. Envuelva la función getUserInfoById:

```
exportar const getUserInfoById = cache(  
  función asíncrona getUserInfoById(userId) {  
    const usuario = await User.findById(userId) if (!usuario) throw new  
    Error('¡Usuario no encontrado!') return { nombredeusuario:  
      usuario.nombredeusuario }, ['usuarios', 'getUserInfoById'],  
  
  })
```

7. Salga del servidor Next.js que esté ejecutándose actualmente.

8. Reconstruye e inicia la aplicación en producción. Notarás que después de crear una nueva publicación,...

No actualice más la página de inicio (ni la ruta API):

```
$ npm run compilación $  
npm start
```

¡Es porque nuestras publicaciones ahora están almacenadas en caché!

9. Si cache funciona incluso en modo de desarrollo. Cierre el servidor Next.js y vuelva a iniciararlo, como se indica a continuación:

```
$ npm ejecuta dev
```

10. Crea una nueva publicación; verás que ni la página de inicio ni la ruta API tienen la nueva publicación.

Publicación creada en la lista.

Ahora que se ha configurado el almacenamiento en caché, aprendamos cómo lidiar con la revalidación del caché (haciendo que los datos en el caché se actualicen).

## Revalidación de la caché mediante acciones del servidor

La mejor manera de gestionar los datos obsoletos es revalidar la caché cuando llegan nuevos datos, por ejemplo, mediante Acciones del Servidor. Para ello, tenemos dos opciones:

- Revalidar todos los segmentos de ruta en una ruta específica mediante la función `revalidatePath`
- Revalidar con una etiqueta específica (y, por lo tanto, potencialmente revalidar múltiples rutas) mediante la función `revalidateTag`

La revalidación significa que la próxima vez que se soliciten datos de la función almacenada en caché, se llamará a la función y se devolverán y almacenarán en caché datos nuevos (en lugar de devolver datos almacenados previamente en caché). Ambas funciones revalidan el caché de datos y, por lo tanto, revalidan el caché completo del enrutador y el del lado del cliente. caché del enrutador también.

Siga estos pasos para llamar a la función `revalidateTag` después de crear nuevas publicaciones:

1. Edite `src/app/create/page.js` e importe la función `revalidateTag`:

```
importar { revalidateTag } desde 'next/cache'
```

2. Dentro de `createPostAction`, llame a la función `revalidateTag` en la etiqueta de publicaciones después de creando la nueva publicación:

```
función asíncrona createPostAction(formData) {
  'utilizar servidor'

  constante userId = getUserIdByToken(token?.valor)
  esperar initDatabase()

  const post = await createPost(userId, {
    título: formData.get('título'),
    contenido: formData.get('contenido'),
  })
  revalidateTag('publicaciones')
  redirigir('/posts/${post._id}')
}
```

3. Ahora, crea una nueva publicación y ve a la página principal. Verás que la publicación recién creada aparece en la lista. La ruta API \$e también la mostrará.

Revalidar la caché cuando se modifican los datos mediante Acciones del Servidor es la forma más directa de actualizarla . Sin embargo, en ocasiones, obtendremos datos de API de terceros, donde no es posible revalidarla. Analizaremos este caso a continuación.

## Revalidar la caché a través de un webhook

Si los datos provienen de una fuente externa, podemos revalidar la caché mediante un webhook. Los webhooks son API que pueden usarse como devoluciones de llamada. Por ejemplo, cuando los datos cambian, la fuente externa llama a nuestro punto final de API para avisarnos que necesitamos volver a obtener los datos.

### Integración de una API de terceros

Antes de implementar un webhook, integremos una API de terceros en nuestra aplicación. Para este ejemplo, usaremos WorldTimeAPI (<https://worldtimeapi.org/>), pero puedes usar la API que prefieras.

Comencemos a implementar una página que obtiene información de una API de terceros:

1. Crea una nueva carpeta llamada `src/app/time/`. Dentro de ella, crea un nuevo archivo `src/app/time/page.js`.
2. Edite `src/app/time/page.js` y defina un componente de página asincrónico:

```
exportar función asincrona predeterminada TimePage() {
```

3. Dentro del componente, obtenga la hora actual de WorldTimeAPI y analice la respuesta como JSON:

```
const timeRequest = await fetch('https://worldtimeapi.org/api/  
zona horaria/UTC')  
tiempo constante = esperar timeRequest.json()
```

4. Representa la marca de tiempo actual:

```
devolver <div>Marca de tiempo actual: {time?.datetime}</div>  
}
```

5. Si accede a la página `http://localhost:3000/time` en su navegador, verá que muestra la hora actual. Sin embargo, al actualizar, la hora nunca se actualiza. Esto se debe a que las solicitudes con `fetch` se almacenan en caché por defecto, similar a lo que ocurrió después de añadir `unstable_cache()` a nuestras funciones de la capa de datos.

### Implementando el Webhook

Ahora, creemos un punto final de API de Webhook en nuestra aplicación que, cuando se llama, revalida el caché para los datos de terceros:

1. Crea una nueva carpeta llamada `src/app/api/v1/webhook/`. Dentro de ella, crea un nuevo archivo `src/app/api/v1/webhook/route.js`.
2. Edite `src/app/api/v1/webhook/route.js` e importe la función `revalidatePath`:

```
importar { revalidatePath } desde 'next/cache'
```

3. Ahora, defina un nuevo controlador de ruta GET que llame a revalidatePath en la página /time y

Luego devuelve una respuesta diciéndonos que fue exitoso:

```
exportar función asíncrona GET() {  
    revalidatePath('/tiempo')  
    devolver Response.json({ ok: true })  
}  
  
exportar const dynamic = 'fuerza-dinámica'
```

Normalmente, los webhooks se definen como controladores de ruta POST (ya que influyen en el estado de la aplicación), pero para simplificar su activación al visitar la página en nuestro navegador, lo hemos definido como controlador de ruta GET. Una ruta POST no se renderiza estáticamente, pero una ruta GET no, por lo que debemos especificar force-dynamic.

4. Visite <http://localhost:3000/api/v1/webhook> en su navegador, luego visite <http://localhost:3000/time> de nuevo; ¡debería ver que la hora se ha actualizado! En la práctica, estaríamos añadiendo la URL de nuestro webhook a la interfaz del sitio web externo que proporciona la API.

#### Nota

Alternativamente, podríamos agregar una etiqueta a la solicitud pasando la opción next.tags en la función fetch(), de la siguiente manera: `fetch('https://worldtimeapi.org/api/zona horaria/UTC', { next: { tags: ['time'] } })`. \$en, podríamos revalidar el caché llamando a `revalidateTag('time')`.

Como podemos ver, revalidar la caché mediante webhooks funciona de maravilla. Sin embargo, a veces ni siquiera podemos añadir un webhook a una API de terceros. Veamos qué hacer cuando no tenemos control sobre la API de terceros.

## Revalidar la caché periódicamente

Si no tenemos ningún control sobre la fuente de datos de terceros, podemos indicarle a Next.js que re valide la caché periódicamente. Configurémoslo ahora:

1. Edite `src/app/time/page.js` y ajuste la función `fetch()`, agregando lo siguiente.

revalidar la opción:

```
const timeRequest = await fetch('https://worldtimeapi.org/api/  
zona horaria/UTC', {  
    siguiente: { revalidar: 10 },  
})
```

En este caso, le dijimos a Next.js que revalidara el caché de datos la próxima vez que se solicitara la API si habían pasado al menos 10 segundos desde la última solicitud.

**Nota**

Con `unstable_cache()`, podemos pasar la opción de revalidar en el tercer argumento.

Para rutas y páginas, podemos especificar `export const revalidate = 10`, lo que revalidará la ruta/página correspondiente.

- Actualice la página `http://localhost:3000/time` en su navegador. Verá la hora actualizada. Actualice la página de nuevo; la hora no se actualizará. Si la actualiza después de al menos 10 segundos, la hora se actualizará de nuevo.

Ahora que hemos aprendido cómo revalidar el caché periódicamente, aprendamos cómo desactivar el almacenamiento en caché.

Cómo optar por no usar el almacenamiento en caché

En ocasiones, es posible que desee desactivar completamente el almacenamiento en caché para ciertas solicitudes. Para ello, pase la siguiente opción a la función de recuperación:

```
obtener('<URL>', { caché: 'no-store' })
```

Para las páginas/rutas, podemos definir `export const dynamic = 'force-dynamic'` para optar por no almacenar en caché toda la ruta (¡aunque los datos aún pueden almacenarse en caché!).

Ahora que hemos aprendido a usar el caché en Next.js para optimizar nuestra aplicación, aprendamos sobre SEO con Next.js.

## SEO con Next.js

En el capítulo 8, aprendimos sobre SEO en aplicaciones full-stack. Next.js ofrece funcionalidades de SEO listas para usar. Exploraremos esta funcionalidad ahora, empezando por añadir títulos dinámicos y metaetiquetas.

### Agregar títulos dinámicos y metaetiquetas

En Next.js, podemos definir metadatos estáticamente exportando un objeto de metadatos desde un archivo `page.js`, o dinámicamente exportando una función `generateMetadata`. Ya hemos añadido metadatos estáticos al diseño raíz, como se puede ver en `src/app/layout.js`:

```
exportar const metadatos = {
  Título: 'Blog Full-Stack Next.js',
  Descripción: 'Un blog sobre React y Next.js',
}
```

Ahora, generaremos dinámicamente metadatos para nuestras páginas de publicaciones:

1. Edite `src/app/posts/[id]/page.js` y defina la siguiente función fuera de la componente de página:

```
exportar función asíncrona generateMetadata({ parámetros }) {  
  constante id = parámetros.id
```

2. Obtener la publicación; si no existe, llamar a `notFound()`:

```
const post = awaitgetPostById(id) si (!post) no se  
encontró()
```

3. De lo contrario, devuelva un título y una descripción:

```
return  
  { título: `${post.title} | Blog Full-Stack Next.js`, descripción: `Escrito por $  
  {post.author.username}`,  
  }  
}
```

¡Eso es todo! Next.js configurará el título y las metaetiquetas correctamente.

#### Nota

Los metadatos se heredan de los diseños. Por lo tanto, es posible definir valores predeterminados para los metadatos en el diseño y luego sobrescribirlos selectivamente para páginas específicas.

Ahora que hemos agregado exitosamente un título dinámico y metaetiquetas, continuemos creando un archivo `robots.txt` para que los motores de búsqueda sepan que pueden indexar nuestra aplicación de blog.

## Creando un archivo robots.txt

Next.js tiene dos formas de crear un archivo `robots.txt`:

- Creación de un archivo `robots.txt` estático en `src/app/robots.txt`
- Creación de un archivo `robots.txt` dinámico mediante la creación de un script `src/app/robots.js`, que devuelve un objeto especial que Next.js convierte en un archivo `robots.txt`

#### Nota

Si necesita un repaso de qué es un archivo `robots.txt` y cómo funcionan los motores de búsqueda, consulte el Capítulo 8.

Solo vamos a crear un archivo robots.txt estático ya que no es necesario un archivo dinámico por ahora.

Siga estos pasos para comenzar:

1. Cree un nuevo archivo src/app/robots.txt.
2. Edite src/app/robots.txt y agregue el siguiente contenido para permitir que todos los rastreadores indexen

todas las páginas:

```
Agente de usuario: *
Permitir: /
```

Ahora que hemos creado un archivo robots.txt, creamos URL significativas.

## Creación de URL significativas (slugs)

Ahora, vamos a crear slugs para nuestras publicaciones de blog, de forma similar a lo que hicimos en el Capítulo 8. Comencemos:

1. Cambie el nombre de la carpeta src/app/posts/[id]/ a src/app/posts/[...path]/. Esto la convierte en una ruta general, que coincide con todo lo que viene después de /posts.
2. Edite src/app/posts/[...path]/page.js y ajuste el código para obtener la primera parte de la URL (el valor de id) del parámetro de ruta:

```
exportar función asíncrona predeterminada ViewPostPage({params}) {
    esperar initDatabase() const [id]
        = params.path const post = esperar
            getPostById(id)
```

3. Además, ajuste el código para la función generateMetadata:

```
exportar función asíncrona generateMetadata({ params }) { const [id] = params.path
```

Con esto, nuestro enrutador ha sido configurado para aceptar un slug opcional en la URL.

4. Instale el paquete slug npm:

```
$ npm install slug@8.2.3
```

5. Edite src/components/Post.jsx e importe la función slug:

```
importar slug desde 'slug'
```

6. Ajuste el enlace a la publicación del blog agregando el slug, de la siguiente manera:

```
<Link href={`/posts/${_id}/${slug(title)}`}>{titulo}</
Enlace>
```

7. Abra un enlace desde la lista de publicaciones; verá que la URL ahora contiene el slug.

Ahora que nos hemos asegurado de que nuestras URL sean significativas, finalizaremos esta sección creando un mapa del sitio para nuestra aplicación de blog.

## Creación de un mapa del sitio

Como aprendimos en el Capítulo 8, 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 de manera más eficiente, asegurándose de que se encuentre todo el contenido de nuestro blog.

Siga estos pasos para configurar un mapa del sitio dinámico en Next.js:

1. Primero, defina una `BASE_URL` para nuestra aplicación como variable de entorno. Edite `.env` y agregue el siguiente línea:

```
URL_BASE=http://localhost:3000
```

2. Crea un nuevo archivo `src/app/sitemap.js`, donde importamos las funciones `initDatabase`, `listAllPosts` y `slug`:

```
importar { initDatabase } desde '@/db/init' importar { listAllPosts }  
desde '@/data/posts' importar slug desde 'slug'
```

3. Defina y exporte una nueva función asíncrona que generará el mapa del sitio:

```
exportar función asíncrona predeterminada sitemap() {
```

4. Primero, enumeraremos todas las páginas estáticas:

```
constante staticPages = [  
  {  
    URL: `${process.env.BASE_URL}` , }, {  
  
    URL: `${process.env.BASE_URL}/crear` , }, {  
  
    URL: `${process.env.BASE_URL}/login` , }, {  
  
    URL: `${process.env.BASE_URL}/signup` , }, {  
  
    URL: `${process.env.BASE_URL}/tiempo` , },  
]
```

5. \$en, obtenemos todas las publicaciones de la base de datos:

```
esperar initDatabase()
const posts = await listaTodosLosPosts()
```

6. Genere una entrada para cada publicación creando la URL y agregando una marca de tiempo de última modificación:

```
const postsPages = posts.map((post) => ({
    URL: `${process.env.BASE_URL}/posts/${post._id}/${slug(post.título)}`,
    última modificación: post.updatedAt,
}))
```

7. Finalmente, devuelve staticPages y postsPages en una matriz:

```
devolver [...páginasestáticas, ...páginaspublicadas]
}
```

8. Vaya a <http://localhost:3000/sitemap.xml> en su navegador; verá que

¡Next.js generó el XML para nosotros a partir de la matriz de objetos!

#### Nota

Se recomienda agregar el mapa del sitio al archivo robots.txt, pero necesitamos convertirlo en un archivo robots.js dinámico para poder proporcionar la URL completa del mapa del sitio (usando la variable de entorno BASE\_URL). Esto es un ejercicio.

Ahora que hemos optimizado nuestra aplicación de blog para motores de búsqueda, aprendamos sobre la carga optimizada de imágenes y fuentes en Next.js.

## Carga optimizada de imágenes y fuentes en Next.js

Cargar imágenes y fuentes de forma optimizada puede ser tedioso, pero Next.js lo hace muy sencillo al proporcionar los componentes Fuente e Imagen.

### El componente Fuente

A menudo, querrás usar una fuente específica para tu página para que sea única y destaque. Si tu fuente está en Google Fonts, puedes configurar Next.js para que la aloje automáticamente. Si usas esta función, tu navegador no enviará solicitudes a Google. Además, las fuentes se cargarán de forma óptima sin necesidad de modificar el diseño.

Descubramos cómo se pueden alojar automáticamente las fuentes de Google con Next.js:

1. Vamos a cargar la fuente Inter importándola desde next/font/google. Editar src/app/layout.js y agregue la siguiente importación:

```
importar { Inter } desde 'next/font/google'
```

2. Ahora, cargue la fuente, de la siguiente manera:

```
constante inter = Inter({  
    subconjuntos: ['latin'],  
    pantalla: 'intercambiar',  
})
```

Inter es una fuente variable, por lo que no es necesario especificar el peso que queremos cargar. Si la fuente no es variable, no olvide especificar el peso. La propiedad `$display: 'swap'` implica que la fuente tiene un periodo de carga extremadamente corto. Si no se carga para entonces, se usará una fuente alternativa. Una vez cargada, se intercambiará.

3. Especifique la fuente en la etiqueta <html>, de la siguiente manera:

```
<html lang='es' className={inter.className}>
```

4. Vaya a <http://localhost:3000/> en su navegador; verá que nuestra aplicación de blog ahora está

¡Usando la fuente Inter! Vea la siguiente captura de pantalla como referencia:



## Hello from React Server Components!

*Written by Daniel Bugl*

## Full-Stack React Projects

*Written by Daniel Bugl*

Figura 18.3 – Nuestra aplicación de blog renderizada con la fuente Inter

Como puedes ver, ¡es muy sencillo usar fuentes de Google autohospedadas con Next.js!

**Nota**

Si quieras usar una fuente que no está en Google Fonts, usa la función `localFont` de `next/font/local`. `$is` te permite cargar una fuente desde un archivo en tu proyecto. Para más información sobre el componente `Font`, consulta la documentación de Next.js: <https://nextjs.org/docs/app/creando-su-aplicacion/optimizando/fuentes>.

A continuación, vamos a aprender sobre el componente `Imagen`, que nos permite cargar imágenes fácilmente y de forma optimizada.

## El componente `Imagen`

Las imágenes representan una gran parte del tamaño de descarga de su aplicación web y, por lo tanto, pueden tener un gran impacto en el rendimiento de Last Contentful Paint (LCP) . Next.js ofrece el componente `Image`, que extiende el elemento `<img>` de la siguiente manera:

- Entrega automática de imágenes redimensionadas para cada dispositivo y resolución
- Prevenir automáticamente el cambio de diseño cuando se cargan las imágenes
- Cargar imágenes únicamente cuando ingresan a la ventana gráfica ("carga diferida"), con imágenes de marcador de posición borrosas opcionales
- Ofrecemos redimensionamiento de imágenes a pedido, incluso si están almacenadas de forma remota

Usar el componente `Imagen` es sencillo: simplemente impórtalo y carga tus imágenes como lo harías con el elemento `<img>`. Probémoslo ahora:

1. Consigue una imagen para usarla como logotipo en tu blog. Puedes usar cualquier imagen, pero asegúrate de que esté en un formato no vectorial (como PNG). En los formatos vectoriales, no es necesario cambiar el tamaño, así que no notarás ningún efecto.
2. Guarda la imagen como un archivo `src/app/logo.png`.
3. Edite `src/app/layout.js` e importe el componente `Imagen` y el logotipo:

```
importar imagen desde 'next/image'  
importar logotipo desde './logo.png'
```

4. Sobre el elemento `<nav>`, renderiza el componente `<Image>`, de la siguiente manera:

```
devolver (  
    <html lang='es' className={inter.className}>  
        <cuerpo>  
            <Image  
                src={logotipo}  
                alt='Logotipo del blog Full-Stack Next.js'  
                ancho={500}
```

```
altura={47}
/>
<navegación>
  <Navegación nombredeusuario={usuario?.nombredeusuario}
  logoutAction={Acción de cierre de sesión} />
</nav>
```

Es importante especificar el ancho y la altura de la imagen para que Next.js pueda inferir la relación de aspecto correcta y evitar cambios en el diseño cuando se carga la imagen.

5. Vaya a <http://localhost:3000/> en su navegador; verá que se muestra el logotipo.

¡Correctamente! Vea la siguiente captura de pantalla como referencia:



## Hello from React Server Components!

*Written by Daniel Bugl*

## Full-Stack React Projects

*Written by Daniel Bugl*

Figura 18.4 – Uso del componente Imagen para mostrar un logotipo para nuestro blog

Si inspecciona la imagen en el navegador, verá que tiene la propiedad srcset con diferentes tamaños proporcionados para que el navegador pueda elegir cuál cargar dependiendo de la resolución de la pantalla.

### Nota

En este ejemplo, cargamos una imagen local, pero el componente Imagen también admite la carga de imágenes desde un servidor remoto y las redimensionará correctamente. Para usar URL externas, habilite el servidor remoto mediante la configuración images.remotePatterns en next.config.js y, luego simplemente pase una URL en lugar de un archivo local al componente de imagen.

## Resumen

En este capítulo, aprendimos a definir rutas de API en Next.js. Sobre el almacenamiento en caché, cómo revalidarlo y cómo desactivarlo. A continuación, aprendimos sobre SEO en Next.js añadiendo metadatos a nuestras páginas, creando URLs relevantes, definiendo un archivo robots.txt y generando un mapa del sitio. Finalmente, aprendimos sobre los componentes Fuente e Imagen, que nos permitieron cargar fuentes e imágenes de forma fácil y óptima en nuestra aplicación.

Todavía hay muchas más características que ofrece Next.js que no hemos cubierto todavía en este libro, como las siguientes:

- **Internacionalización:** Nos permite configurar el proceso de enrutamiento y renderizado de contenidos para varios idiomas
- **Middleware:** Nos permite ejecutar el código antes de que se completen las solicitudes, de forma similar a cómo funciona el middleware. trabaja en Express
- **Node.js sin servidor y entornos de ejecución de Edge:** nos permiten escalar nuestras aplicaciones aún más al no tener que ejecutar un servidor Node.js completo
- **Enrutamiento avanzado:** Nos permite modelar escenarios de enrutamiento complejos, como rutas paralelas (mostrando dos páginas a la vez)

En el próximo capítulo, Capítulo 19, Implementación de una aplicación Next.js, aprenderemos cómo implementar una Next. Aplicación js que utiliza Vercel y una configuración de implementación personalizada.

## Implementación de una aplicación Next.js

Tras aprender los conceptos avanzados de Next.js, es hora de aprender a implementar una aplicación Next.js. La forma más sencilla de implementar aplicaciones Next.js es mediante la plataforma Vercel, proporcionada por la empresa que desarrolla el framework Next.js. Después de aprender a implementar nuestra aplicación en la plataforma Vercel, aprenderemos a crear una configuración de implementación personalizada con Docker.

En este capítulo cubriremos los siguientes temas principales:

- Implementación de una aplicación Next.js con Vercel
- Creación de una configuración de implementación personalizada para aplicaciones Next.js

## 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/cap19>.

El video de \$e CiA para este capítulo se puede encontrar en: <https://youtu.be/ERBFy5mHwek>.

## Implementación de una aplicación Next.js con Vercel

Comenzaremos implementando nuestra aplicación en Vercel, una plataforma donde podemos implementar nuestras aplicaciones de forma gratuita, sencilla y cómoda. Sigue estos pasos para empezar a implementar nuestra aplicación Next.js con Vercel:

1. Copie la carpeta ch18 existente a una nueva carpeta ch19 ejecutando el siguiente comando:

```
$ cp -R ch18 ch19
```

2. Abra la carpeta ch19 en VS Code.

3. Instale la herramienta CLI de Vercel como un paquete global con el siguiente comando:

```
$ npm install -g vercel@33.5.3
```

4. Ejecute la CLI de Vercel:

```
$ vercel
```

5. Se le pedirá que inicie sesión en Vercel. Seleccione uno de los métodos de inicio de sesión y siga los pasos que Vercel le indica para iniciar sesión.

6. Despues de iniciar sesión exitosamente, se le harán preguntas sobre la implementación de su proyecto, confirme todas con los valores predeterminados proporcionados presionando Enter/Return hasta que Vercel CLI intente construir su proyecto.

```
⌚ ➔ ~/D/F/ch19 ↵ main > vercel
Vercel CLI 33.5.3
> > No existing credentials found. Please log in:
? Log in to Vercel github
> Success! GitHub authentication complete for me@omnidan.net
? Set up and deploy “~/Development/Full-Stack-React-Projects/ch19”? [Y/n] y
? Which scope do you want to deploy to? omnidan
? Link to existing project? [y/N] n
? What's your project's name? ch19
? In which directory is your code located? ../
Local settings detected in vercel.json:
Auto-detected Project Settings (Next.js):
- Build Command: next build
- Development Command: next dev --port $PORT
- Install Command: `yarn install`, `pnpm install`, `npm install`, or `bun install`
- Output Directory: Next.js default
? Want to modify these settings? [y/N] n
🔗 Linked to omnidan/ch19 (created .vercel)
🔍 Inspect: https://vercel.com/omnidan/ch19/7PhHxuGSJog1MGxReTYY8kKZgtvj [2s]
✅ Production: https://ch19-kwlu28dbp-omnidan.vercel.app [2s]
```

Figura 19.1 – Intentando implementar nuestra aplicación en Vercel

7. Mientras se construye el proyecto, puede visitar la URL proporcionada en la CLI para ver el estado actual del proceso de compilación (asegúrese de iniciar sesión en Vercel en el mismo navegador), como se muestra en la siguiente captura de pantalla:

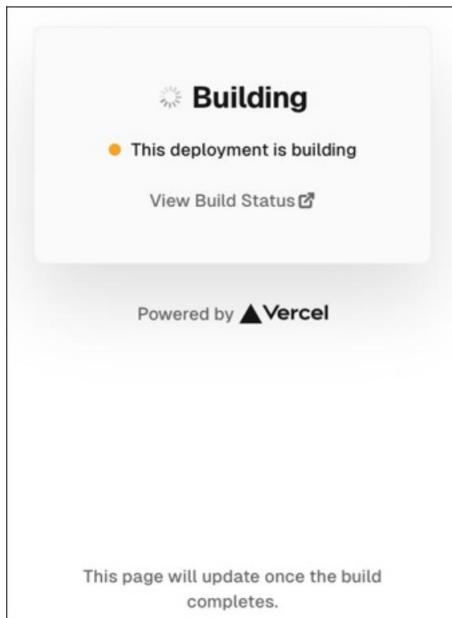


Figura 19.2 – Monitoreo del proceso de compilación en el navegador

8. Desafortunadamente, la compilación falla porque la variable de entorno DATABASE\_URL está configurada a `mongodb://localhost:27017/blog`.

Ahora necesitamos ajustar esta variable de entorno en Vercel.

## Configuración de variables de entorno en Vercel

Siga estos pasos para configurar las variables de entorno necesarias en Vercel:

1. Reutilice el clúster de bases de datos existente creado en MongoDB Atlas o siga los pasos de la sección "Creación de una base de datos MongoDB Atlas" del Capítulo 5 para crear un nuevo clúster de bases de datos. Ahora debería tener una cadena de conexión para su base de datos.

2. Verifique que la cadena de conexión funcione ejecutando el siguiente comando:

```
$ mongosh "<cadena de conexión>"
```

3. Si reutiliza el clúster de bases de datos existente, asegúrese de borrar la base de datos y las colecciones, ya que las publicaciones y los usuarios solían tener un formato ligeramente diferente en el Capítulo 5. Ejecute los siguientes comandos dentro de MongoDB Shell para borrar las colecciones:

```
> db.posts.drop()
> db.usuarios.drop()
```

4. Vaya a <https://vercel.com/> e inicie sesión con el mismo proveedor de inicio de sesión que utilizó anteriormente.

5. Deberías ver una descripción general de tus proyectos, incluido el proyecto ch19 que creamos anteriormente a través de la CLI de Vercel, como se muestra en la siguiente captura de pantalla:

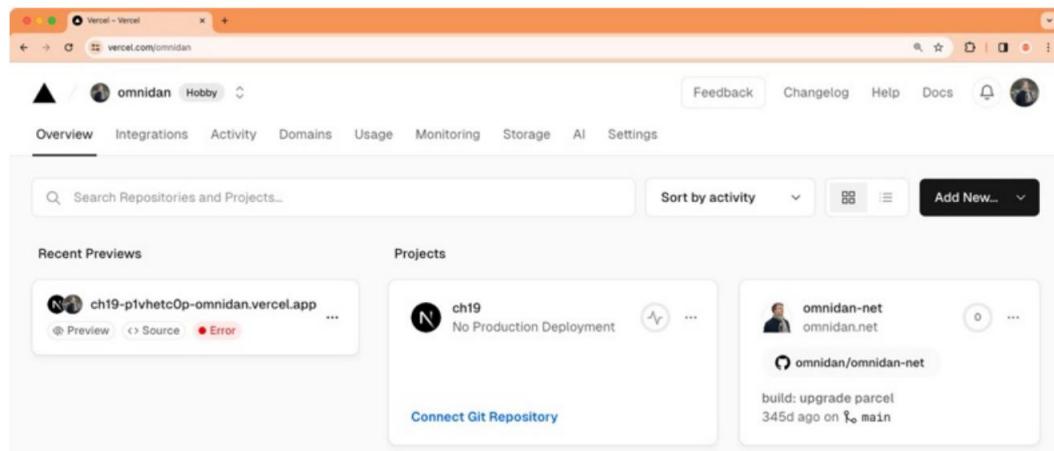


Figura 19.3 – El panel de control de Vercel

6. Haga clic en el proyecto ch19, luego vaya a la pestaña Configuración , seleccione Variables de entorno en la barra lateral y cree una nueva variable de entorno ingresando DATABASE\_URL como Clave y la cadena de conexión obtenida previamente como Valor, como se muestra en la siguiente captura de pantalla:

The screenshot shows the Vercel interface with the project 'omnidan' and environment 'ch19'. The 'Settings' tab is selected. On the left, a sidebar lists various settings: General, Domains, Environment Variables (which is currently selected), Git, Integrations, Deployment Protection, Functions, Data Cache, Cron Jobs, Security, and Advanced. The 'Environment Variables' section contains a note about deployment requirements and a 'Create new' button. It also includes sections for 'Sensitive' variables (disabled) and 'Environment' variables (Production and Preview branches selected). A table allows setting key-value pairs, with one entry for 'DATABASE\_URL' with the value '<enter connection string here>'. There are also icons for file operations.

Figura 19.4 – Agregar una variable de entorno en Vercel

#### Nota

Para las aplicaciones de producción, también se debe establecer la variable de entorno JWT\_SECRET con un secreto aleatorio. Además, se debe establecer la variable de entorno BASE\_URL con la URL de la implementación de producción de la aplicación. Por ejemplo, si la URL pública de tu blog es <https://ch19-omnidan.vercel.app/>, se debe establecer BASE\_URL con esa URL.

7. Haga clic en el botón Guardar debajo de las variables de entorno para guardar los cambios.

8. Ejecute la CLI de Vercel nuevamente para intentar otra implementación:

```
$ vercel
```

Alternativamente, puede activar una reconstrucción desde la interfaz de usuario web de Vercel.

9. Verá que ahora se implementa correctamente, visite la URL de vista previa proporcionada por la CLI de Vercel en su navegador para ver nuestra aplicación de blog cargándose exitosamente:

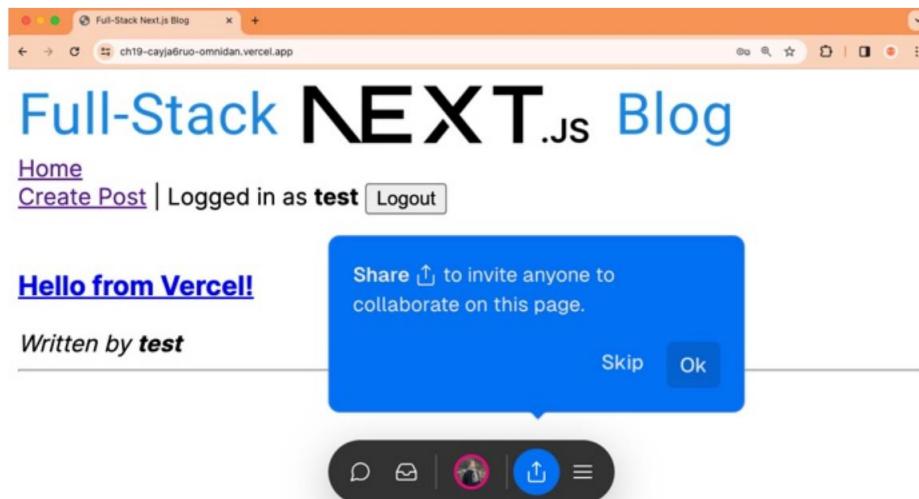


Figura 19.5 – Implementación de “Vista previa” funcional de nuestra aplicación

Curiosamente, la CLI de Vercel ahora ha creado una implementación de vista previa . Este es el comportamiento predeterminado en Vercel. Primero se implementará en un entorno de vista previa , donde podemos probar todo para asegurarnos de que nuestra aplicación funcione correctamente. El entorno de vista previa solo es accesible si se ha iniciado sesión a través de Vercel. También podemos invitar a otros a probar nuestra aplicación aquí y añadir comentarios mediante la barra de herramientas de Vercel en la parte inferior.

10. Ahora que hemos confirmado que nuestra aplicación funciona, podemos implementarla en producción, de la siguiente manera:

```
$ vercel --prod
```

La siguiente captura de pantalla muestra una vista previa y una implementación de producción que se realizan con la CLI de Vercel:

```
● ➔ ~/D/F/ch19 ↵ main > vercel
Vercel CLI 33.5.3
🔍 Inspect: https://vercel.com/omnidan/ch19/2HXmvcW1cR6TmB4yurf8kr5LpAJq [3s]
✅ Preview: https://ch19-cayja6ruo-omnidan.vercel.app [3s]
📝 To deploy to production (ch19.vercel.app), run `vercel --prod`
● ➔ ~/D/F/ch19 ↵ main± > vercel --prod
Vercel CLI 33.5.3
🔍 Inspect: https://vercel.com/omnidan/ch19/3kwA6ozErmsT15heLEDChQUG3Yyj [1s]
✅ Production: https://ch19-f230clhd2-omnidan.vercel.app [1s]
```

Figura 19.6 – Implementación de nuestra aplicación en entornos de “Vista previa” y “Producción” con la CLI de Vercel

¡Ahora nuestra aplicación está implementada en el entorno de producción y es accesible para cualquier persona, sin tener que iniciar sesión a través de Vercel!

**Nota**

La URL proporcionada en la salida CLI de Vercel no es accesible para nadie; debe usar uno de los dominios especificados en la sección Dominios en el panel de Vercel. El valor predeterminado debe ser <https://ch19-<vercel-username>.vercel.app/>.

Como podemos ver, implementar nuestra aplicación con Vercel es muy fácil y práctico. Sin embargo, en algunos casos, queremos implementarla en nuestra propia infraestructura. Aprendamos ahora a crear una configuración de implementación personalizada para aplicaciones Next.js.

### Creación de una configuración de implementación personalizada para aplicaciones Next.js

Ahora aprenderemos a configurar una implementación personalizada para aplicaciones Next.js con Docker. Ya aprendimos los conceptos básicos de la implementación de aplicaciones con Docker en el Capítulo 5, así que consulta ese capítulo si tienes alguna duda o necesitas un repaso de Docker. Comencemos ahora a configurar nuestra aplicación Next.js para una implementación con Docker:

1. Primero, necesitamos cambiar el formato de salida de Next.js a independiente. La opción \$is le indica a Next.js que cree una carpeta .next/standalone que solo contenga los archivos necesarios para una implementación en producción, incluyendo únicamente los node\_modules necesarios. La carpeta \$is se puede implementar sin tener que volver a instalar node\_modules. Edite next.config.mjs y ajuste la configuración como se indica a continuación:

```
/** @type {import('siguiente').NextConfig} */
constante nextConfig = {
    salida: 'independiente',
}

exportar predeterminado nextConfig
```

2. Ahora, creamos un archivo .dockerignore para ignorar ciertos archivos que no deberían incluirse en nuestra imagen:

```
módulos_de_nodo
.env*
.vscode
.git
```

3. Cree un nuevo Dockerfile, comience definiendo una imagen base desde el nodo:20:

```
DESDE nodo:20 COMO base
```

4. \$en, define una nueva imagen para construir la aplicación, basada en la imagen base:

DESDE la base COMO construir

5. Establezca el directorio de trabajo en la carpeta /app y copie el package.json y archivos package-lock.json:

WORKDIR /app  
COPIA paquete.json .  
COPIA paquete-lock.json .

6. Ahora, instale todas las dependencias y, además, instale Sharp, que Next.js utiliza para cambiar el tamaño y optimizar imágenes en producción:

EJECUTAR npm install  
EJECUTAR npm install sharp

7. Copia todos los archivos de nuestro proyecto:

COPIAR . . .

8. A continuación, defina los argumentos para el proceso de compilación. Definiremos todas las variables de entorno aquí, ya que Next.js también las usa durante el proceso de compilación para construir estáticamente ciertas rutas:

URL DE LA BASE DE DATOS ARG  
ARG JWT\_SECRETO  
URL BASE DE ARGENTINA

9. Ahora podemos ejecutar el comando de compilación, de la siguiente manera:

EJECUTAR npm run build

10. Define una nueva imagen para la aplicación final, basada también en la imagen base:

DESDE la base COMO final

11. También definimos el directorio de trabajo:

WORKDIR /aplicación

12. Configuramos los permisos para ejecutar nuestra aplicación como un usuario especial de nextjs en lugar de root:

EJECUTAR addgroup --system --gid 1001 nodejs  
EJECUTAR adduser --system --uid 1001 nextjs

13. Ahora, copie los archivos necesarios para ejecutar un servidor Next.js independiente desde la imagen de compilación:

```
COPIA --from=build /app/public ./public EJECUTAR mkdir  
-p .next EJECUTAR chown  
nextjs:nodejs .next COPIA --from=build /  
app/.next/standalone ./ COPIA --from=build /app/.next/  
static ./next/static
```

14. Definimos las variables PORT, HOSTNAME y NODE\_ENV:

```
EXPONER 3000  
PUERTO ENV 3000  
ENV NOMBRE DE HOST "0.0.0.0"  
Producción ENV NODE_ENV
```

15. \$en, ejecutamos el servidor independiente Next.js como el usuario nextjs que definimos anteriormente:

```
USUARIO nextjs  
CMD ["nodo", "server.js"]
```

16. Asegúrese de que el servidor de base de datos se esté ejecutando en un contenedor Docker.

17. Ahora podemos construir la imagen de Docker ejecutando el siguiente comando:

```
$docker build\  
-t blog-nextjs \  
--build-  
arg "URL_DE_LA_BASE_DE_DATOS=mongodb://host.docker.  
interno:27017/blog" \  
--build-arg "JWT_SECRET=reemplazar-con-secreto-aleatorio" \  
--build-arg  
"BASE_URL=http://localhost:3000" \  
.
```

En el comando anterior, especificamos blog-nextjs como nombre para nuestra imagen y las variables de entorno necesarias para compilarla. No olvides el punto (.) al final del comando, ya que este define el contexto de compilación, incluyendo dónde buscar el Dockerfile.

#### Nota

Puede consultar el ejemplo oficial Dockerfile de Next.js para obtener una versión actualizada : <https://github.com/vercel/next.js/blob/canary/examples/with-docker/Dockerfile>

18. Finalmente, ejecute un nuevo contenedor Docker, de la siguiente manera:

```
$docker run \
-d \
--nombre blog-app \
-p 3000:3000 \
-e "URL_DE_BASE_DE_DATOS=mongodb://host.docker.internal:27017/blog" \
-e "JWT_SECRET=reemplazar-con-secreto-aleatorio" \
-e "URL_BASE=http://localhost:3000" \
--reiniciar a menos que se detenga \
blog-nextjs
```

En el comando anterior, especificamos la ejecución de un contenedor con el nombre blog-app

En segundo plano (modo demonio), se publicó en el puerto 3000. Luego, se especificaron las variables de entorno y se le indicó a Docker que reiniciara el contenedor si fallaba. Finalmente, se especificó el nombre de la imagen, que es blog-nextjs (la imagen que creamos en el paso anterior).

19. Visita <http://localhost:3000> y verás que el blog funciona correctamente.

Ahora que tenemos un contenedor Docker, podemos implementarlo en un servicio en la nube (o en nuestro propio servidor), tal como lo hicimos en el Capítulo 5. Si bien es un poco más complicado configurar una implementación personalizada para una aplicación Next.js, sigue siendo bastante sencillo realizar una configuración simple.

Para configuraciones más avanzadas, como múltiples instancias, necesitará configurar un volumen compartido entre las instancias para que la caché y las imágenes optimizadas se puedan compartir (en Vercel, esto se realiza automáticamente en segundo plano). Sin embargo, esta configuración queda fuera del alcance de este libro. Puede consultar Next. Documentación de js sobre autohospedaje para obtener más información sobre cómo hacer esto: <https://nextjs.org/docs/aplicación/construcción-de-su-aplicación/implementación#autohospedaje>.

## Resumen

En este capítulo, primero aprendimos cómo implementar una aplicación Next.js usando Vercel. Luego, aprendimos cómo crear una configuración de implementación personalizada usando Docker.

En el siguiente y último capítulo, el Capítulo 20, Profundizando en el desarrollo full-stack, cubriremos brevemente varios temas avanzados de desarrollo full-stack que se han omitido en este libro hasta ahora, lo que le dará una idea de cómo continuar su viaje de aprendizaje del desarrollo web full-stack con React.

# 20

## Profundizando en Desarrollo de pila completa

Tras aprender a crear e implementar aplicaciones Next.js, hemos concluido nuestro recorrido hacia el desarrollo full-stack con React. En este último capítulo, quiero ofrecerles una visión general y abordar brevemente varios temas avanzados que se han tratado en este libro. Incluye conceptos como el mantenimiento de proyectos a gran escala, la optimización del tamaño de los paquetes, una descripción general de las bibliotecas de interfaz de usuario (UI) y soluciones avanzadas de gestión de estado.

En este capítulo cubriremos los siguientes temas principales:

- Descripción general de otros marcos de pila completa
- Descripción general de las bibliotecas de UI
- Descripción general de las soluciones avanzadas de gestión de estados
- Consejos para el mantenimiento de proyectos a gran escala

### Nota

Como este capítulo solo brinda una descripción general de temas avanzados en desarrollo full-stack con enlaces para lectura adicional, no hay ejemplos de código y, como tal, tampoco requisitos técnicos para este capítulo.

### Descripción general de otros marcos de trabajo de pila completa

En este libro, aprendimos sobre Next.js, el framework full-stack más popular para React. Sin embargo, otros frameworks full-stack podrían interesarte, cada uno con sus propias ventajas y desventajas.

Sin embargo, antes de poder comparar los marcos, recapitulemos los diferentes métodos de renderizado en React:

- Representación del lado del cliente (CSR): Representa los componentes en el navegador.
- Representación del lado del servidor (SSR): representa los componentes en el servidor y sirve el resultado.
- Generación de sitios estáticos (SSG): Representa los componentes en el servidor y los almacena como estáticos. HTML, luego sirve el HTML estático
- Generación estática incremental (ISR): realiza SSG en el % y almacena en caché el resultado durante un tiempo determinado. cantidad de tiempo
- Generación diferida de sitios (DSG): almacena en caché todos los datos en el momento de la compilación y cuando se vuelven a renderizar las páginas. hace uso de esos datos almacenados en caché

Además, muchos frameworks (y proveedores de nube) son compatibles con Edge Runtime, un subconjunto de API web estándar que se utiliza para ejecutar código en el borde. En este caso, "edge" se refiere a entornos informáticos sin servidor que pueden implementarse en muchas ubicaciones lo más cerca posible del cliente. Por ejemplo, si alguien accede a su sitio web desde Austria, el código se ejecutará en el servidor más cercano, que puede estar en Austria o Alemania. Sin embargo, para alguien desde Estados Unidos, el código se ejecutará en un servidor estadounidense. Esto reduce la latencia de la red y acelera la carga de nuestra aplicación.

Ahora, echemos un vistazo a los diferentes marcos de trabajo full-stack.

## Next.js

Ya hemos aprendido sobre Next.js en este libro: es el framework web full-stack más popular en el momento de escribir este artículo, compatible con CSR, SSR, SSG e ISR. Últimamente, Next.js usa SSG por defecto para mantener el máximo rendimiento de la aplicación, pero aún ofrece la posibilidad de revalidar páginas en caché y proporcionar SSR cuando sea necesario.

Next.js también es compatible con Edge Runtime, pero debe habilitarse específicamente para usarlo en lugar del entorno de ejecución Node.js (predeterminado). Algunas funciones tampoco están disponibles en Edge Runtime.

Puedes consultar Next.js aquí: <https://nextjs.org/>.

## Remezcla

Remix es un marco completo que se centra en los estándares web.

A diferencia de Next.js, no ofrece SSG y, en cambio, se centra en mejorar el rendimiento de renderizado dinámico y la integración con la infraestructura web mediante SSR. Dado que Remix está completamente basado en estándares web, no requiere Node.js para su ejecución, por lo que puede ejecutarse de forma nativa en entornos de ejecución de borde, como Cloud Worker.

Actualmente, Remix no es compatible con React Server Components (RSC), pero tiene sus propios patrones que generan las mismas ventajas que el uso de RSC.

---

Al igual que Next.js, admite rutas anidadas (con diseños anidados), enrutamiento dinámico y renderizado paralelo en el servidor. \$e Remix Router se basa en React Router, lo que facilita su comprensión si ya has trabajado con React Router. También admite estados de carga/error y un tipo de acciones del servidor.

En general, Remix es una excelente alternativa a Next.js, especialmente si prefieres trabajar con API web estándar y te preocupa la compatibilidad con entornos de ejecución en edge. Su objetivo es simplificar el desarrollo web al usar los estándares al máximo.

Puedes escuchar Remix aquí: <https://remix.run/>.

## Gatsby

Gatsby se centra principalmente en SSG. Si bien ahora también puede realizar SSR, los autores del framework recomiendan usar SSG en la medida de lo posible. En lugar de ISR, Gatsby ofrece DSG, que mejora la consistencia de los datos en sitios web grandes, pero a costa de ofrecer datos potencialmente obsoletos.

Gatsby ha comenzado recientemente a ofrecer compatibilidad con RSC y también con Edge Runtime. Sin embargo, al igual que Next.js, también depende de las API de Node.js y, por lo tanto, solo ofrece una parte de sus funciones para Edge Runtime.

Una ventaja de Gatsby es su amplio ecosistema de complementos, que permite a los desarrolladores integrar fácilmente nuevas funciones.

Sin embargo, una desventaja de Gatsby es que no admite rutas anidadas con diseños anidados.

Si bien Next.js y Remix ofrecen compatibilidad con REST y GraphQL, Gatsby se centra principalmente en GraphQL, y solo admite las API REST como algo secundario. Sin embargo, esto le permite ofrecer plugins que integran fácilmente diversas fuentes de datos.

En general, Gatsby puede ser un excelente framework si buscas principalmente SSG, la capacidad de integrar datos de diversas fuentes y una herramienta fácil de aprender. En lugar de imponerte toda la complejidad de un framework de golpe, Gatsby la revela progresivamente a través de su ecosistema de plugins.

Puedes ver Gatsby aquí: <https://www.gatsbyjs.com/>.

A continuación, proporcionaremos una descripción general de algunas bibliotecas de UI seleccionadas.

## Descripción general de las bibliotecas de UI

En este libro, he omitido deliberadamente las bibliotecas de interfaz de usuario, ya que son muy obstinadas, cambian constantemente y harían que los ejemplos de código fueran significativamente más largos. En esta sección, me gustaría ofrecer un resumen de algunas bibliotecas de interfaz de usuario seleccionadas. ¡Síntete libre de explorarlas por tu cuenta y estate atento a otras opciones y nuevas versiones en este campo!

#### Interfaz de usuario de materiales (MUI)

MUI es una de las bibliotecas de componentes más populares para React. Admite una amplia gama de componentes, incluyendo componentes complejos como tablas de datos. Además, cuenta con un sistema de temas muy extensible que te permite adaptarlo a tu estilo. Sin embargo, su motor de estilos, al momento de escribir este artículo, es incompatible con RSC, algo que se mejorará en futuras versiones. Usa MUI si te gusta su estilo en general, pero quieras personalizar los colores, la tipografía y el espaciado para personalizarlo.

Puedes consultar MUI aquí: <https://mui.com/>.

#### Viento de cola CSS

Tailwind CSS es un framework CSS orientado a la utilidad y no requiere React. Sin embargo, se integra bien con React, lo que te permite diseñar fácilmente tus componentes personalizados. Al ser exclusivamente CSS, puedes adaptar los componentes React a tus necesidades. Esto también significa que los RSC son totalmente compatibles, ya que Tailwind es simplemente un conjunto de clases CSS. Usa Tailwind si quieras implementar un estilo totalmente personalizado para tus aplicaciones de forma rápida y sencilla, en comparación con usar CSS directamente.

Puedes consultar Tailwind CSS aquí: <https://tailwindcss.com/>.

#### Interfaz de usuario de Tailwind

Los creadores de Tailwind CSS también ofrecen un conjunto de componentes prediseñados de estilo exclusivo que utilizan Tailwind CSS, llamado Tailwind UI. Consultalo si necesitas inspiración para crear componentes con Tailwind aquí: <https://tailwindui.com/>.

#### Reaccionar Aria

React Aria es un conjunto sencillo de componentes con excelente compatibilidad con accesibilidad e internacionalización . De forma predeterminada, los componentes no tienen estilo, lo que permite crear diseños personalizados. También se puede usar en combinación con Tailwind. Usa React Aria si quieras crear un sistema de diseño pero no quieras afrontar los retos de crear componentes accesibles.

Puedes consultar React Aria aquí: <https://react-spectrum.adobe.com/react-aria/>.

#### NextUI

NextUI es una biblioteca de interfaz de usuario (UI) de próxima aparición que utiliza el estilo de Vercel (la empresa creadora de Next.js). Está desarrollada con Tailwind CSS, pero ofrece varios componentes basados en React Aria, lo que garantiza una accesibilidad de primera clase. Al igual que MUI, también ofrece numerosos componentes y es muy personalizable mediante temas. Además, es compatible con RSC gracias a que está basada en Tailwind CSS. Usa NextUI si te gusta el estilo y quieres personalizarlo un poco, especialmente si desarrollas con un framework compatible con RSC.

Puedes consultar NextUI aquí: <https://nextui.org/>.

A continuación, proporcionaremos una descripción general de las soluciones avanzadas de gestión de estado.

## Descripción general de las soluciones avanzadas de gestión de estados

En este libro, nos centramos en soluciones sencillas de gestión de estados en React, como useState y los contextos. Sin embargo, en proyectos a gran escala, puede ser conveniente usar bibliotecas avanzadas de gestión de estados para gestionar estados complejos. Aquí presentaré una descripción general de algunas bibliotecas de gestión de estados seleccionadas , pero tenga en cuenta que existen muchas más, así que no dude en consultarlas y decidir cuál se adapta mejor a su proyecto.

### Retroceso

Recoil es una biblioteca de gestión de estados para React, desarrollada por Facebook Open Source. Por lo tanto, comparte muchos de los principios de React. Es un sistema muy simple pero potente, donde el estado se almacena en átomos y luego se deriva mediante selectores. \$is nos permite, por ejemplo, almacenar solo la entrada del usuario de un formulario en átomos y la carga útil resultante en un selector, que deriva su estado de los átomos.

Puedes consultar Recoil aquí: <https://recoiljs.org/>.

### Jotai

Jotai adopta un enfoque similar a Recoil, pero simplifica el sistema eliminando los selectores y trabajando únicamente con átomos. Los átomos pueden entonces derivar el estado de otros átomos. Si busca una solución de gestión de estados que siga siendo sencilla, pero más potente que useState, Jotai es una excelente solución.

Puedes consultar Jotai aquí: <https://jotai.org>.

### Redux

Redux adopta un enfoque diferente: ofrece un almacén central que contiene todo tu estado y solo te permite modificarlo mediante acciones. Esto garantiza que tu aplicación se comporte de forma consistente y que las mismas acciones del usuario siempre resulten en los mismos cambios de estado. Redux es ideal para aplicaciones donde las acciones son esenciales y cuando se necesita la función de deshacer/rehacer (como ciertos editores).

Puedes consultar Redux aquí: <https://redux.js.org>.

### MobX

MobX es una biblioteca de gestión de estado basada en señales que utiliza observables para rastrear cambios de estado. Cuando un valor se convierte en observable, se puede mutar directamente, como una variable JavaScript normal, pero cualquier cambio en él activa los observadores del estado para ejecutar y volver a renderizar los componentes.

Puedes consultar MobX aquí: <https://mobx.js.org>.

## xstate

xstate es una biblioteca de máquina de estados que puede resultar muy útil cuando tienes interfaces de usuario complejas con varios estados que necesitan ser rastreados.

Puedes consultar xstate aquí: <https://stately.ai/docs/xstate>.

## Estado actual

Zustand es una pequeña biblioteca de gestión de estados con una API basada en ganchos que combina valores y funciones que modifican los valores de las tiendas. Cada tienda expone un gancho donde se pueden usar sus valores y funciones.

Puedes consultar Zustand aquí: <https://docs.pmnd.rs/zustand/getting-started/introduction>.

Ahora, terminaremos aprendiendo algunos consejos sobre el mantenimiento de proyectos a gran escala.

## Consejos para el mantenimiento de proyectos a gran escala

Para que este libro sea lo más breve y conciso posible, y accesible para un público amplio, omití intencionalmente algunos temas y tecnologías. Sin embargo, es fundamental conocerlos al mantener proyectos a gran escala, por lo que quiero abordarlos brevemente aquí.

## Usando TypeScript

TypeScript es una extensión de JavaScript con sintaxis para tipos. Un sistema de tipos puede ser muy útil para detectar errores a tiempo y brindar confianza al refactorizar una base de código extensa. Si bien puede llevar tiempo acostumbrarse a escribir todo, resulta una ventaja cuando uno se da cuenta de que todos los problemas aparecen como errores de tipo en el editor de código, en lugar de errores de ejecución para los usuarios.

Recomiendo usar TypeScript para todos los proyectos nuevos. Es fácil de aprender cuando ya se conoce JavaScript y se integra bien con frameworks como Next.js.

Puede obtener más información sobre TypeScript aquí: <https://www.typescriptlang.org>.

## Configurar un Monorepo

En este libro, siempre tratábamos una sola aplicación a la vez. Sin embargo, los proyectos a gran escala suelen constar de varias aplicaciones, con la posibilidad de compartir varias bibliotecas internas. Por ejemplo, podrías tener dos aplicaciones que comparten componentes de interfaz de usuario en una biblioteca de interfaz de usuario común. Tener todas esas bibliotecas y aplicaciones en repositorios Git separados suele generar una sobrecarga organizativa.

Para simplificar las cosas, los equipos de desarrollo a menudo deciden configurar un Monorepo, que contiene todas las aplicaciones y bibliotecas en un solo repositorio. Esto también hace que sea más fácil mantener la base de código consistente y realizar un seguimiento de refactorizaciones a gran escala.

Puede obtener más información sobre Monorepos aquí: <https://monorepo.tools>.

Para configurar un Monorepo, utilice un administrador de paquetes que admite espacios de trabajo, como pnpm (<https://pnpm.io>) o yarn (<https://yarnpkg.com>). Algunas herramientas , como Turborepo, facilitan la creación y el mantenimiento de Monorepos. Consulta las guías en <https://turbo.build> para aprender a configurar un Monorepo con Turborepo.

### Optimización del tamaño del paquete

A medida que tu proyecto crece, el paquete de JavaScript que se envía al navegador también crece. Esto puede ser problemático para dispositivos con conexiones o procesadores lentos. A veces, ciertas dependencias aumentan considerablemente el tamaño del paquete, por lo que conviene comprobar periódicamente cómo los cambios en tu proyecto afectan al tamaño del paquete.

Para Vite, puedes usar vite-bundle-visualizer para descubrir qué dependencias están aumentando el tamaño de tu paquete: <https://github.com/KusStar/vite-bundle-visualizer>.

Para Next.js, puedes usar el complemento social @next/bundle-analyzer: <https://nextjs.org/docs/app/construyendo-su-aplicacion/optimizando/analizador-de-paquetes>.

## Resumen

En este libro, comenzamos con la motivación de convertirnos en desarrolladores full-stack. Luego, configuramos nuestro entorno de desarrollo y aprendimos sobre herramientas que nos simplifican la vida. Después, conocimos Node.js y MongoDB, dando nuestros primeros pasos como desarrolladores backend. Luego, implementamos un backend para una aplicación de blog con Express y Mongoose, y escribimos pruebas unitarias con Jest. Posteriormente, integramos un frontend con nuestro backend con React y TanStack Query, creando así nuestra primera aplicación web full-stack. Después, aprendimos a implementar nuestra aplicación con Docker y a configurar CI/CD. Luego, añadimos autenticación a nuestra aplicación con JWT. Aprendimos a mejorar el tiempo de carga de nuestra aplicación con SSR y, en el proceso, desarrollamos nuestra propia solución SSR (sencilla). Después, aprendimos cómo funcionan los motores de búsqueda y cómo asegurarnos de que los clientes puedan encontrar nuestra aplicación web facilitando el SEO y proporcionando metadatos para la integración en redes sociales. A continuación, implementamos pruebas de extremo a extremo utilizando Playwright, asegurándonos de que nuestra aplicación siempre funcione como se espera. Luego, aprendimos a agregar y visualizar estadísticas usando MongoDB y Victory.

Posteriormente, nos desviamos de las API REST y desarrollamos un backend con una API GraphQL, aprendiendo qué es GraphQL y cuáles son sus beneficios. Después, desarrollamos un frontend que consume esta API GraphQL. Después, dejamos de lado nuestra aplicación de blog y creamos una aplicación de chat basada en eventos con Socket.IO. Mientras lo hacíamos, aprendimos a crear un backend y un frontend, y a añadir persistencia, en el paradigma basado en eventos. En los últimos capítulos de este libro, aprendimos sobre Next.js, un framework de desarrollo web integral. Presentamos el enrutador de aplicaciones, una nueva forma de estructurar tus aplicaciones, y los RSC, que nos permitieron fusionar aún más el backend y el frontend, reduciendo la necesidad de código repetitivo para crear API y, en su lugar, permitiéndonos acceder directamente al código desde una capa de datos dentro de los RSC. También aprendimos sobre conceptos avanzados y optimizaciones en Next.js, como el almacenamiento en caché, el SEO y la carga optimizada de fuentes e imágenes. Finalmente, aprendimos a implementar una aplicación Next.js usando Vercel, una plataforma en la nube proporcionada por los creadores de Next.js, y creamos una configuración de implementación personalizada con Docker para poder implementar nuestra aplicación en cualquier otro proveedor de nube (o en nuestros propios servidores).

Ha sido un largo camino. Pero, como hemos visto en este capítulo, aún quedan muchos temas por profundizar, y el ecosistema del desarrollo web está cambiando rápidamente. Constantemente surgen nuevas tecnologías, especialmente en cuanto a RSC y Acciones de Servidor, que, al momento de escribir esto, aún son nuevas y están en desarrollo. Espero que se lancen muchas más funciones en este ámbito, así que estén atentos a los anuncios innovadores en el mundo de React.

Mantén el hambre. Mantén la valentía. Nunca pierdas el apetito por ir en busca de nuevas ideas, nuevas experiencias y nuevas aventuras. – Steve Jobs

# Índice

## A

agradecimientos 327, 334 acciones  
137

Conceptos avanzados de GraphQL 301

- Fragmentos 302
- Introspección 303

Representación avanzada del lado del servidor

209-211 Soluciones avanzadas de gestión de estados 459

- Jotai 459
- MobX 459
- Retroceso 459
- Redux 459
- xstate 460
- Zustand 460

manejo avanzado de tokens 180 API

de agregación que

- integra 273, 274

Servicios web de Amazon (AWS) 130

Rutas API

- creando, para listar entradas de blog 424, 425
- creando, con Express 83, 84 dening
- 77 dening, en
- Next.js 424 dinámico,
- haciendo 430 renderizado
- estático 428-430

Cliente Apollo 306

- configuración 306, 307

Enlace de referencia

- de la documentación del Apolo 320

Tiempo de carga de la aplicación

- Apollo Server 285 , evaluación comparativa
- 184-190 aplicación, optimización para motores de búsqueda
- títulos dinámicos, adición
- 222, 223 URL significativas (slugs), creación 221, 222
- metaetiquetas, adición 223
- archivo robots.txt, creación 215, 216
- páginas separadas, creación para publicaciones
- 216-220 mapa del sitio, creación
- 226-228 App Router 382,
- 383 estructura de enrutamiento,  
    como estructura de carpetas 384, 385

rutas autenticadas dening

- 162, 163 autenticación
- añadiendo, a

GraphQL 296

## B

backend Dockerle

- creando 120-122

- Implementación de una imagen  
Docker de backend en Cloud Run 134, 135,  
implementación de un  
punto final de backend para obtener  
información del usuario 176
- backend para aplicación de chat, con reconocimientos de  
Socket.IO 331, que se utiliza para obtener  
información del usuario 334, 335
- Mensajes de chat que se transmiten desde el  
servidor a los clientes 332, 333
- eventos, emitiendo para enviar mensajes de chat del  
cliente al servidor 331, 332
- Salas, unirse para enviar mensajes 333, 334 servicio  
backend
- Diseño de la estructura  
de carpetas 52, 53, creación de gráficos de  
barras 53-55
- creación, en Victory 275-277 aplicación  
de blog ,  
juntando 105, 106 modelo de  
publicación de blog  
usando 57, 58
- creación de  
publicaciones de blog y fechas de última  
actualización,  
definiendo 59, 60
- obtención 108-110 modelo,  
definiendo para 56, 57
- accesorios integrados 245 solicitudes agrupadas 191
- almacenamiento en caché, en Next.js 426, 427  
caché de datos 427  
caché de ruta completa 427  
enlace de referencia 428  
solicitud de memorización 427  
caché del enrutador  
427 pila de  
llamadas 26 metaetiqueta de  
conjunto de caracteres  
226 interfaz de  
la aplicación de  
chat iniciando 349
- pruebas 349 implementación  
de comandos de chat , con  
reconocimientos 356-358 componentes de chat  
Implementando 351
- funcionalidad de chat  
Implementando 350
- Componente ChatMessage que  
implementa 351
- Creación de componentes de  
página de chat 348, 349
- Implementación del componente  
ChatRoom 354, 355 componentes  
del cliente 398 refactorización del  
código del lado del  
cliente 368, 369
- renderizado del lado del cliente (CSR) 398, 456  
desventajas 398 nube
- do  
Aplicación de pila completa, implementada en 130
- Imagen Docker de  
backend de Cloud Run 133 ,  
implementándose en 134,
- Imagen Docker de frontend de 135, implementándose en 135
- colecciones 39
- cache  
invalidando, periódicamente 435, 436  
revalidando, mediante Acciones del servidor 433  
revalidando, mediante Webhook 434
- almacenamiento en caché, inhabilitando 436

- comandos  
Implementando, para unirse y cambiar de salas en la aplicación de chat 370-373
- commitlint  
Establecimiento  
de 18 congresos convencionales de compromiso  
URL 19
- mensajes de confirmación  
en el congreso convencional de commitlint 19
- Acceso a datos a nivel de componentes 402
- Creación de  
componentes para la aplicación de blog 386-390
- conurrencia  
con JavaScript en el navegador y  
Node.js 25-27
- contenedores 30  
creando 32, 33
- Content Delivery Network (CDN) 229 entrega continua (CD) 119 implementación continua (CD)  
configuración, para automatizar la  
implementación 141 integración continua (CI) 119 agregar, para el backend 138, 139 agregar, para el frontend 137, 138 configuración, para automatizar las pruebas 136, 137
- Pruebas de dramaturgo, ejecutando 250, 251
- integración continua/entrega continua (CI/CD) 30  
controlador 52
- Elementos esenciales de la Web 184, 188 rastreador 214
- Componente CreatePost que implementa la función de servicio createPost 99, 100  
casos de prueba, definición para 64-66  
escritura 63  
crear, leer, actualizar y eliminar (CRUD) 42, 53
- intercambio de recursos de origen cruzado (CORS) 87 se utiliza para permitir el acceso desde otras URL 88
- Creación de configuraciones de implementación personalizadas de Cumulative Layout  
Sh! (CLS) 184 para aplicaciones Next.js 451-454  
accesorios personalizados usando 247, 248
- D**
- Capa de acceso a datos 402, agregación de datos, en funciones de servicio de agregación de back-end, definiendo 268-271, implementando 268 rutas, definiendo 271, 272, conexión de base de datos  
Configuración de modelos de base de datos 403  
Creación de 404, 405  
enfoques de manejo de datos acceso a datos a nivel de componente 402 capa de acceso a datos 402 API HTTP 402
- Capa de datos 53, definición para usuarios 406-408, funciones de almacenamiento en caché 430-432, definición de funciones de la capa de datos 405, generación diferida de sitios (DSG), 456, función de eliminación posterior, definición 71-74, instalación de dependencias 11, flujo de trabajo de implementación, definición 142-144, metatiqueta de descripción 224

- Docker 30  
accediendo, vía VS Code 34  
instalando 32  
Cliente Docker 31  
docker-compose  
múltiples imágenes, gestión 127, 128  
Docker Compose 127  
Docker Compose el 120  
Contenedores Docker 31  
Demonio Docker 31  
Escritorio Docker  
enlace de descarga 32  
Instalación de la  
extensión  
Docker 6 Dockerfile  
120 Docker Host 31  
Docker Hub 31  
Obtención de credenciales de  
Docker Hub 141  
Creación  
de .dockerignore  
le 122 Creación, para el frontend  
126 Imágenes Docker  
31 Contenedor, creación 123  
Contenedor, ejecución 124  
Creación 120, 122, 123  
Implementación, en el registro de Docker 132,  
133 Registro Docker 31  
dotenv  
utilizado para configurar variables de  
entorno 79-81
- mi  
Estándar del módulo ECMAScript (ESM) 10, 62  
Tiempo de ejecución de Edge 456
- pruebas de extremo a extremo  
corriendo 237  
escribiendo 237  
Componente EnterMessage que  
implementa 351, 352  
variables de entorno  
Ajuste, con dotenv 79-81  
ESLint 11  
configuración 14, 15  
nuevo script, agregar para ejecutar linter 16  
aplicaciones basadas en eventos 324, 325  
bucle de eventos  
26 modelo de eventos  
creando 254, 255  
eventos 137  
recopilando, en el frontend 257, 258  
simulando 258-261  
formulario de ejecución 122  
Expreso 77  
Rutas API, creación con 83, 84 rutas,  
definición con cuerpo de solicitud  
JSON 86, 87 configuración  
78, 79
- F
- Fastify  
URL 78  
Manejo de archivos en Node.js 24,  
25  
filtros que implementan  
110-113 primera pintura con contenido (FCP)  
184, 398 primera pintura significativa (FMP)  
398 accesorios 237  
para configuraciones de prueba  
reutilizables 244 escritura 245-247

- 
- Componente de fuente 440, 441
    - enlace de referencia 442
  - fragmentos 302
  - contenedor frontend
    - creando 127
    - ejecutando 127
  - frontend Dockerfile
    - Creación de una imagen
  - Docker frontend 124-126 , creación de una 126
    - implementación en una aplicación de pila completa en Cloud Run 135
  - Implementando, en la nube
    - 130 marcos de pila completa 455
    - Gatsby 457
    - Next.js 456
    - Remix 456
  - Configuración de proyecto React de pila completa 93, 94
    - funciones de almacenamiento en caché, en la capa de datos 430-432
    - dening, para enumerar publicaciones 66, 67
  - Autenticación GraphQL
    - 282-284 , agregando a 296 nombres de usuario de autor, resolviendo en una sola consulta 310, 311
    - funcionalidad de creación de publicaciones, migrando a 317-319
    - funcionalidad de inicio de sesión, migrando a 316, 317
    - mutaciones 284, 285
    - mutaciones, implementando 297, 298
    - usando 299-301 publicaciones, consultando desde el frontend 307-310
  - API GraphQL, implementación en el backend 285-287
    - consultas profundamente anidadas, probando campos 291-293, implementando para consultar publicaciones 287-289
    - tipos de entrada, implementando 294, 295
    - Tipo de poste, dening 289, 290
    - Tipo de usuario, dening 290, 291
  - Fragments de consultas
    - GraphQL , utilizando para reutilizar partes de las consultas 312-314
    - variables, utilizando 312

GRAMO

- Gatsby 457
- URL 457
- obtener función de publicación única dening 71-74
- Acciones de GitHub 136
- Registro de artefactos de Google 133
- Googlebot 214
- Cuenta de Google
  - Cloud , creación de 131
  - credenciales, obtención de 141, 142
- Consola de búsqueda de Google 215

## H

- Informe HTML de ejecuciones de prueba visualizadas 248, 249
- API HTTP 402
- Configuración de Husky 16 16, 17

## I

- idempotencia 76
- Componente de imagen 442, 443
- generación estática incremental (ISR) 456 índice 214

- 
- |  |  |
|--|--|
| <p>entorno de desarrollo integrado<br/>(IDE) 3 internacionalización</p> <p>(i18n) 378</p> <p>Números asignados de Internet<br/>Autoridad (IANA) 151</p> <p>introspección 303</p> <p>representación isomórfica 192</p> <p><b>Yo</b></p> <p>JavaScript, en la arquitectura<br/>del navegador versus Node.js 22</p> <p>Extensión Jest VS Code<br/>usando 74, 75</p> <p>trabajos</p> <p>137 Jotai 459</p> <p>URL 459</p> <p>Extensión 85 del formateador JSON</p> <p>Token web JSON (JWT) 149, 150</p> <p>creando 152, 153</p> <p>encabezado</p> <p>150 encabezado, enviando al crear publicaciones</p> <p>178-180 carga útil</p> <p>150, 151 reclamos</p> <p>privados 151 reclamos</p> <p>públicos 151 reclamos</p> <p>registrados 151 firma</p> <p>150, 152 almacenando</p> <p>155, 172-174 usando 153, 154</p> <p><b>K</b></p> <p>Koa</p> <p>URL 78</p> | <p>Yo</p> <p>Proyectos a gran escala, manteniendo el tamaño<br/>del paquete 460, optimizando 461</p> <p>Monorepo, estableciendo 460</p> <p>TypeScript, utilizando 460</p> <p>Pintura con contenido más grande (LCP) 184</p> <p>Última pintura con contenido (LCP) 442 libuv 25</p> <p>Gráfico de líneas de la<br/>herramienta Faro 184</p> <p>creando, en Victoria 277-279</p> <p>Componente de enlace<br/>utilizado para enlazar con otras rutas 170, 171 lint-staged</p> <p>16 configuración 16,</p> <p>17 lista de publicaciones</p> <p>obteniendo 409</p> <p>lista de<br/>publicaciones</p> <p>casos de prueba, definiendo para 67-71</p> <p>lista de publicaciones</p> <p>definiendo la función</p> <p>66, 67 cargadores</p> <p>204 usuario<br/>conectado accediendo</p> <p>164, 165</p> <p>página de inicio de<br/>sesión creando</p> <p>172-174 ruta de<br/>inicio de sesión</p> <p>creando 160 servicio de inicio de sesión creando 158-160</p> <p>Enlace de<br/>referencia de la herramienta Lupa 27</p> <p>METRO</p> <p>Interfaz de usuario de materiales (MUI) 4.458</p> <p>URL 458</p> |
|--|--|

- mensajes  
reproduciendo 361, 362  
almacenable 361, 362
- metaetiquetas 223  
metaetiqueta de conjunto de  
caracteres 226 metaetiqueta de  
descripción 224 metaetiqueta  
de robots 225 metaetiqueta de ventana gráfica 225, 226
- MobX 459  
URL 459
- modelo 52  
dening, para publicaciones de blog 56,  
57 patrón modelo-vista-controlador (MVC) 52
- MongoDB 34-36, 262  
documento, eliminar 42  
documento, insertar en una colección 39, 40 documento,  
consultar y ordenar 40, 41 documento, actualizar 41
- MongoDB Atlas 130 Creación  
de base de datos MongoDB Atlas  
130, 131 Acceso a  
base de datos MongoDB ,  
a través de Node.js 45-47 Aplicación
- MongoDB Express React Node.js (MERN)  
119 MongoDB, para agregación
- de datos Duración promedio de sesión,  
cálculo 266-268 Número de  
vistas diarias por  
publicación, obtención 264-266 Número  
total de vistas por  
publicación, obtención 262-264
- Configuración del  
servidor MongoDB 37, 38
- MongoDB Shell 36, 38
- Mongoose 55
- biblioteca de mongoose  
configuración 55, 56
- Creación de esquemas  
Mongoose para almacenar mensajes de chat 360, 361
- Monorepo 460  
URL 461
- mutaciones 284, 285  
usando, en el frontend 314-316
- norte
- Rutas API de Next.js 378,  
455, 456 , definiendo en 424  
características 378  
configuración 379-382  
URL 456
- Configuración  
de implementación personalizada de  
la aplicación Next.js , creación  
para la implementación 451-454, con Vercel  
446, 447 NextUI 458  
URL 458
- Node.js 22  
archivos, manejo 24, 25  
Base de datos MongoDB, acceso a través de 45-47
- Arquitectura Node.js versus  
JavaScript en el navegador 22 Creación  
de scripts Node.js  
23, 24 nodemon
- usando 81, 82
- Tipos de bases de datos  
NoSQL 35
- Oh
- modelado de datos de objetos (ODM) 51
- Metaetiquetas del artículo OG  
que utilizan 230, 231

- Metaetiquetas Open Graph (OG) 229, 230
- Interconexión de sistemas abiertos
- Modelo (OSI) 325
- PAG
- páginas
- creando 386
  - creando, para la aplicación de blog
  - 390-392 enlace, agregando entre 392-395
  - objeto JavaScript simple (POJO) 72
- Dramaturgo 234
- backend, preparándose para
  - pruebas de extremo a extremo 235,
  - 236 encabezado
  - 234 sin cabeza
  - 234 instalación 234, 235
- Pruebas de dramaturgo
- en marcha, en CI 250, 251
- pnpm
- URL 461
- Podman 30
- Implementación del
- componente posterior 98, 99
- Implementación del componente
- PostFilter 101
- Componente PostList
- Implementando 102-104
- Publicaciones que crean
- consultas 114-117, invalidando la
  - capa de datos de 117
  - publicaciones, definiendo 405, 406
- Implementación del componente
- PostSorting 101
- Más bonita 11
- configurando 12
- Doctores más bonitos
- URL 12
- Extensión más bonita
- configuración 13
- Prettier ignora la creación
- 13 proyecto
- limpieza
- 342, 343 configuración, con
  - Vite 8-10
- R
- Rancher Desktop 30
- Reaccionar
- principios 92
- Cliente Socket.IO, integración con 342 TanStack
- Consulta, configuración para 107, 108 React Aria
- 458
- URL 458
- Componentes de React, renderizado en
- el servidor 191, 192
- punto de entrada del lado del cliente, definiendo 196,
- 197 index.html, actualizando 197
- package.json, actualizando 197
- configuración del servidor
- 193-196 punto de entrada del lado del servidor, definiendo 196
- Enrutador React 166
- utilizando, para implementar múltiples rutas 166, 167
- Enrutador React, con renderizado del lado
- del servidor 198, 199
- enrutador del lado del cliente, dening 199
- Solicitud expresa, mapeo a
- Solicitud de obtención 200,
- 201 enrutador del lado del servidor, dening 202, 203
- Componentes del servidor React
- (RSC) 397-399, 402, 423, 456
- datos, obtención de la base de datos 409
- características 400, 401

- lista de publicaciones, obteniendo  
409 publicación única, obteniendo 410, 411
- Retroceso 459
- URL 459
- Redux 459
- URL 459
- metainformación relevante 226
- Remezcla 456
- URL 457
- mensajes reproducidos
- distinguiendo, visualmente 363, 364
- transferencia de estado representacional
- (REST) Rutas API
- 75 , definiendo 77
- Arquitectura y métodos basados en REST
- ELIMINAR 76
- OBTÉN 76
- PARCHE 76
- PUESTO 76
- PONER 76
- Ventajas de REST/solicitud-
- respuesta 325
- desventajas 325
- configuraciones de prueba
- reutilizables con
- accesorios 244 metaetiqueta
- robots 225 archivo robots.txt 215
- creando 215, 216
- habitaciones 327, 333
- grupo de rutas 430
- capa de ruta 53
- rutas
- dening, para rastrear eventos 255-257
- probando, en el navegador 88, 89
- corredor 137
- S
- Rastreador de motores de búsqueda
- 214 Optimización de motores de búsqueda (SEO) 192
- puntuación 213
- motores de búsqueda 214
- Secretos API 134
- SEO, con Next.js 436 títulos
- dinámicos, agregando 436, 437 URL
- significativas (slugs), creando 438 metaetiquetas,
- agregando 436, 437 archivos
- robots.txt, creando 437 mapas del
- sitio, creando 439, 440
- Acciones del servidor 412, 413
- caché, revalidando vía 433
- Manejo de JWT 416-418 página
- de inicio de sesión, implementación 416-418
- cierre de sesión, implementación 419, 420
- creación posterior, implementación 421, 422 página
- de registro, implementación 413-416 inicio de
- sesión de usuario, verificación 418, 419
- obtención de datos del lado del servidor
- 204 enfoque de hidratación 204-209
- enfoque de datos iniciales 204, 205
- representación del lado del servidor
- (SSR) 191, 378, 398, 456 funciones
- de servicio
- creando, para gestionar mensajes de chat 361 dening,
- para la funcionalidad de chat 365, 366 dening, para
- rastrear eventos 255-257
- Capa de servicio 53,
- formulario de shell 122
- página de registro
- creando 168-170 ruta
- de registro
- creando 157, 158
- servicio de registro
- creando 156, 157

- Implementación  
  de cierre de sesión simple  
  174, 175 aplicación de página única (SPA)  
172 obtención  
  de publicación única 410, 411  
principio de responsabilidad única 96 mapa  
del sitio 226  
  creación 226-228  
incrustaciones en redes sociales  
  
Mejorando la  
  
desconexión del socket 229 , al cerrar sesión 350  
Autenticación  
  Socket.IO 326 , agregando por  
    Integración de JWT 336-339  
  backend, creación de aplicaciones de chat  
  331, conexión a 326  
  eventos, emisión de 327  
  eventos, recepción de 327  
  funciones 326  
  configuración 328, 329  
Integración del cliente  
  Socket.IO con React 342  
  configuración 329, 330  
Contexto de Socket.IO  
  Creando 343-346  
  conectando 346, 347  
servidor Socket.IO  
  refactorización, para usar funciones de servicio 367  
ordenación  
  implementación 110-113  
componentes estáticos  
  creación 386  
componentes estáticos de React  
  implementación 97  
representación estática  
  en las rutas API 428-430  
  
generación de sitio estático (SSG) 456  
Componente de estado  
  creando 347, 348 JWT  
almacenados  
  usando 174, 175  
Lenguaje de consulta estructurado  
  Bases de datos (SQL) 34  
Lenguaje de consulta estructurado  
  Consultas (SQL) 262

## T

- etiqueta 32  
Viento de cola CSS 458  
URL 458  
Interfaz de usuario de Tailwind  
URL 458  
Configuración de  
  consultas de TanStack para React  
  107, 108, utilizado para integrar el servicio backend 106  
casos de prueba  
  dening, para la función de servicio  
    createPost 64-66  
  dening, para las publicaciones de la lista  
67-71 entorno de prueba  
  configuración 60-62  
API de terceros que  
  integra la revocación  
  del token 434 174  
Tiempo total de bloqueo (TBT) 184  
Protocolo de control de transmisión (TCP) 325  
TurboRepo  
  URL 461  
TypeScript 460  
  URL 460

**Tú**

identificadores únicos universales (UUID) 255 contenedores no utilizados limpieza 129

actualización posterior

definición de función 71-74

ganchito useChat

que implementa 352, 353

Componente de usuario

Implementando, para obtener y renderizar el nombre de usuario 177, 178

interfaz de usuario (UI) 4

bibliotecas de interfaz de usuario (UI) 455, 457

Interfaz de usuario de materiales (MUI) 458

NextUI 458

Reaccionar Aria 458

Modelo de usuario

Tailwind CSS 458

creando 155, 156

nombres de usuario

obteniendo 175

**V**

variables

que utilizan, en consultas GraphQL 312

Variables

de entorno de Vercel , configuración 447-450

Aplicación Next.js, implementación con 446, 447 URL 448

Gráfico de barras

de Victoria 275 , creación de 275,

gráfico de líneas 276, creación de enlaces de referencia 277-279 275

vista 52

etiqueta meta de la ventana gráfica 225, 226

Instalación de Visual Studio Code (VS Code) 5

5-7

URL 5

Vite 8, 124

alternativas 10

proyecto, configuración 8-10

Contenedor Voronoi 277

Base de

datos de VS Code , acceso mediante 42-44

Docker, accediendo vía 34

Navegador de extensiones

de VS Code , que se muestra mientras se ejecuta

pruebas 239, 240

Pruebas de extremo a extremo, ejecución

de 238 pruebas grabadas, limpieza de 242-244

pruebas grabadas, finalización de 244

pruebas, grabación de 241, 242

**O**

Caché de

webhook , revalidando mediante 434

implementando 434, 435 servidor

web

creando 28, 29

ejecutando, para servir JSON le 29

WebSockets 325

Ventajas de los WebSockets

basados en eventos

325 Desventajas 325

Flujos de trabajo 137

Carga de trabajo Federación de identidad 142

inodriga

xstate 460

URL 460

## Y

hilo

URL 461

## Z

Nivel 460

URL 460



[packtpub.com](http://packtpub.com)

Suscríbete a nuestra biblioteca digital en línea para acceder a más de 7000 libros y videos, además de herramientas líderes en la industria que te ayudarán a planificar tu desarrollo personal y a impulsar tu carrera profesional. Para más información, visita nuestro sitio web.

## ¿Por qué suscribirse?

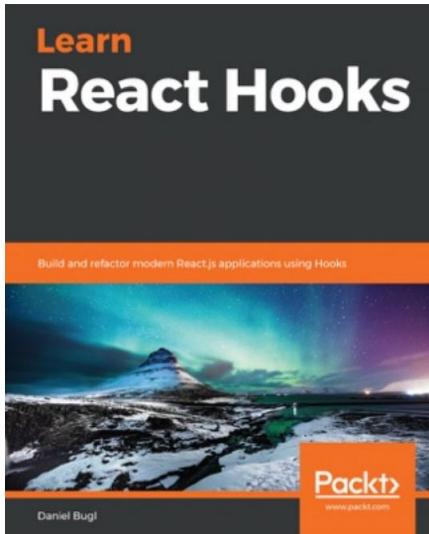
- Dedique menos tiempo a aprender y más tiempo a codificar con libros electrónicos y videos prácticos de más de 4000 profesionales de la industria.
- Mejora tu aprendizaje con planes de habilidades diseñados especialmente para ti
- Obtenga un libro electrónico o un video gratuito cada mes
- Totalmente buscable para un fácil acceso a información vital
- Copiar y pegar, imprimir y marcar contenido

¿Sabías que Packt ofrece versiones electrónicas de todos los libros publicados, con archivos PDF y ePub disponibles? Puedes actualizar a la versión electrónica en [packtpub.com](http://packtpub.com) y, como cliente de la versión impresa, tienes derecho a un descuento en la copia electrónica. Contáctanos en [customercare@packtpub.com](mailto:customercare@packtpub.com) para más detalles.

En [www.packtpub.com](http://www.packtpub.com), también puede leer una colección de artículos técnicos gratuitos, suscribirse a una variedad de boletines gratuitos y recibir descuentos y ofertas exclusivas en libros y libros electrónicos de Packt.

# Otros libros que te pueden gustar

Si te ha gustado este libro, puede que te interesen estos otros libros de Packt:

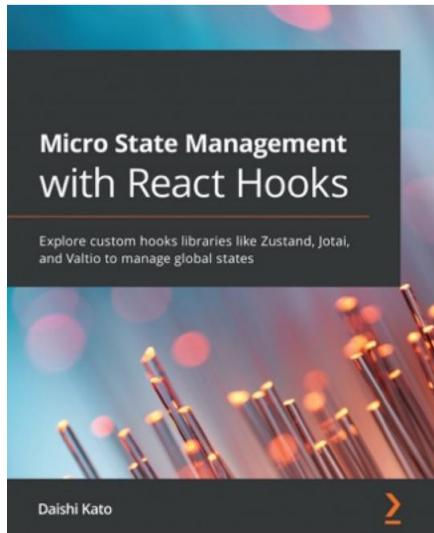


Aprenda los ganchos de React

Daniel Bugl

ISBN: 978-1-83864-144-3

- Comprender los fundamentos de React Hooks y cómo modernizan la gestión de estados en Aplicaciones de React
- Crea tus propios Hooks personalizados y aprende a probarlos
- Utilice los Hooks de la comunidad para implementar un diseño responsive y más
- Aprenda las limitaciones de los Hooks y para qué debería y no debería usarlos • Familiarícese con la implementación del contexto de React usando Hooks
- Refactorice su aplicación web basada en React, reemplazando los componentes de clase React existentes con Hooks
- Utilice soluciones de gestión de estados como Redux y MobX con React Hooks



Gestión de microestados con React Hooks

Daishi Kato

ISBN: 978-1-80181-237-5

- Comprender la gestión del microestado y cómo puede lidiar con el estado global.
- Cree bibliotecas utilizando la gestión de microestados junto con React Hooks
- Descubra cómo los enfoques micro son fáciles usando React Hooks
- Comprenda la diferencia entre el estado del componente y el estado del módulo
- Explorar varios enfoques para implementar un estado global
- Familiarícese con ejemplos concretos y bibliotecas como Zustand, Jotai y Valtio

## Packt está buscando autores como tú

Si te interesa ser autor para Packt, visita [authors.packtpub.com](http://authors.packtpub.com) y postúlate hoy mismo. Hemos trabajado con miles de desarrolladores y profesionales de la tecnología como tú para ayudarles a compartir sus conocimientos con la comunidad tecnológica global. Puedes enviar una solicitud general, postularte para un tema de actualidad específico para el que estemos buscando autores o enviar tu propia idea.

---

## Comparte tus pensamientos

Hola,

Soy Daniel Bugl, autor de Proyectos React Full-Stack Modernos. Espero que hayas disfrutado de este libro y te haya resultado útil para aumentar tu productividad y eficiencia con React.

Realmente me ayudaría (¡y también a otros lectores potenciales!) si pudieras dejar una reseña en Amazon compartiendo tus pensamientos sobre este libro.

Vaya al siguiente enlace para dejar su reseña:

<https://packt.link/r/1837637954>

Su reseña me ayudará a comprender qué ha funcionado bien en este libro y qué se podría mejorar para futuras ediciones, por lo que realmente se agradece.



Los mejores deseos,

Daniel Bugl

## 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.

Machine Translated by Google