

## Uso de bases de datos

En este documento, mostraremos cómo las aplicaciones Node.js pueden usar una base de datos relacional para almacenar y consultar datos. Este documento explica cómo trabajar directamente con una base de datos ejecutando consultas SQL y cómo adoptar un enfoque más sin duda con un paquete de mapeo relacional de objetos (ORM, Object Relational Mapping). La tabla 1 pone este documento en contexto y la tabla 2 resume lo que haremos.

Tabla 1: Poner las bases de datos en contexto.

Pregunta	Respuesta
¿Qué son?	Las bases de datos son los medios más comunes para almacenar datos persistentemente.
¿Por qué son útiles?	Las bases de datos pueden almacenar grandes volúmenes de datos y hacer cumplir una estructura de datos que permite realizar consultas eficientes.
¿Cómo se usan?	Las bases de datos son administradas por motores de base de datos, que se pueden instalar como paquetes npm, ejecutarse en servidores dedicados o consumirse como servicios en la nube.
¿Hay alguna trampa o limitaciones?	Las bases de datos pueden ser complejas y requieren conocimiento adicional, como poder formular consultas en SQL.
¿Hay alguna alternativa?	Las bases de datos no son la única forma de almacenar datos, pero son las más comunes y, en general, las más efectivas porque son robustas y se reducen fácilmente.

Tabla 2: Resumen del documento.

Problema	Solución	Listado
Almacenar datos de manera persistente.	Usar una base de datos.	7, 8, 12, 13
Simplificar el proceso de cambiar cómo se almacenan los datos.	Usar una capa de repositorio.	9-11
Mostrar datos almacenados.	Incluir resultados de consulta al renderizar plantillas.	14, 15
Evitar que los valores enviados por el usuario se interpreten como SQL.	Usar parámetros de consulta.	16, 17
Asegurarse de que los datos se actualicen de manera consistente.	Usar una transacción.	18-21
Usar una base de datos sin necesidad de escribir consultas SQL.	Usar un paquete ORM y describir los datos utilizados por la aplicación utilizando el código JavaScript.	22-25, 27, 28
Realizar operaciones que sean demasiado complejas para describir el uso de clases de modelos.	Usar la instalación del paquete ORM para ejecutar SQL.	26

Consultar y actualizar datos utilizando un ORM.	Usar los métodos definidos por las clases de modelo, con restricciones especificadas usando objetos JavaScript.	29-32
---	---	-------

## Preparándose para este documento

Este documento utiliza el proyecto part2app del documento anterior. Para prepararse para este documento, el listado 1 elimina el código de validación del lado del cliente, que no se utilizará en este documento.

Listado 1: El contenido del archivo client.js en la carpeta src/client.

```
document.addEventListener('DOMContentLoaded', () => {
  // do nothing
});
```

El listado 2 actualiza la configuración de enrutamiento para la aplicación de ejemplo.

Listado 2: El contenido del archivo server.ts en la carpeta src/server.

```
import { createServer } from "http";
import express, { Express } from "express";
import httpProxy from "http-proxy";
import helmet from "helmet";
import { engine } from "express-handlebars";
import { registerFormMiddleware, registerFormRoutes } from "../forms";

const port = 5000;

const expressApp: Express = express();

const proxy = httpProxy.createProxyServer({
  target: "http://localhost:5100", ws: true
});

expressApp.set("views", "templates/server");
expressApp.engine("handlebars", engine());
expressApp.set("view engine", "handlebars");

expressApp.use(helmet());
expressApp.use(express.json());

registerFormMiddleware(expressApp);
registerFormRoutes(expressApp);
```

```

expressApp.use("^/$", (req, resp) => resp.redirect("/form"));

expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));

expressApp.use((req, resp) => proxy.web(req, resp));

const server = createServer(expressApp);

server.on('upgrade', (req, socket, head) => proxy.ws(req, socket, head));

server.listen(port,
  () => console.log(` HTTP Server listening on port ${port}`));

```

La nueva configuración de enrutamiento elimina las entradas que ya no son necesarias. Todos los ejemplos de este documento utilizan plantillas, y la nueva ruta coincide con las solicitudes de la ruta predeterminada y responde con una redirección a la URL /form. La nueva ruta utiliza soporte Express para hacer coincidir patrones de URL, como este:

```

...
expressApp.use("^/$", (req, resp) => resp.redirect("/form"));
...

```

El patrón es necesario para hacer coincidir las solicitudes de `http://localhost:5000` y no las solicitudes que se manejan mediante otras rutas, como `http://localhost:5000/css/bootstrap.min.css` (que se maneja mediante el middleware de contenido estático) o `http://localhost:5000/bundle.js` (que se reenvía al servidor HTTP de desarrollo de webpack).

El listado 3 actualiza la plantilla de edad para agregar un campo que permite al usuario especificar una cantidad de años, solo para permitir más variaciones en los datos de los resultados. La estructura de la salida HTML se ha modificado para introducir un diseño de dos columnas y utilizar una plantilla parcial denominada history. Este listado también elimina los elementos de error de validación, que es algo que no se debe hacer en un proyecto real, pero que no son necesarios en este documento.

Listado 3: El contenido del archivo `age.handlebars` en la carpeta `templates/server`.

```

<div class="container fluid">
  <div class="row">
    <div class="col-7">
      {{#if name}}

```

```

    <div class="m-2">
      <h4>Hello {{ name }}. You will be {{ nextage }}
        in {{ years }} years.</h4>
    </div>
  {{/if}}
</div>
<div>
  <form id="age_form" action="/form" method="post">
    <div class="m-2">
      <label class="form-label">Name</label>
      <input name="name" class="form-control"
        value="{{ name }}" />
    </div>
    <div class="m-2">
      <label class="form-label">Current Age</label>
      <input name="age" class="form-control"
        value="{{ age }}" />
    </div>
    <div class="m-2">
      <label class="form-label">Number of Years</label>
      <input name="years" class="form-control"
        value="{{ years }}" />
    </div>
    <div class="m-2">
      <button class="btn btn-primary">Submit</button>
    </div>
  </form>
</div>
</div>
<div class="col-5">
  {{> history }}
</div>
</div>
</div>

```

Para crear la vista parcial, agrega un archivo denominado `history.handlebars` a la carpeta `templates/server/partials` con el contenido que se muestra en el listado 4.

Listado 4: El contenido de la carpeta `history.handlebars` en la carpeta `templates/server/partials`.

```

<h4>Recent Queries</h4>
<table class="table table-sm table-striped my-2">
  <thead>
    <tr>
      <th>Name</th><th>Age</th><th>Years</th><th>Result</th>

```

```

    </tr>
  </thead>
  <tbody>
    { {#unless history } }
    <tr><td colspan="4">No data available</td></tr>
    { {/unless } }
  </tbody>
</table>

```

La plantilla parcial muestra los datos proporcionados a través de una propiedad de contexto `history` y muestra un mensaje predeterminado cuando no hay datos disponibles. El listado 5 revisa el código que maneja la URL `/form` para eliminar las comprobaciones de validación introducidas en el documento (práctica) anterior.

Listado 5: El contenido del archivo `forms.ts` en la carpeta `src/server`.

```

import express, { Express } from "express";

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {
    resp.render("age");
  });

  app.post("/form", (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);
    const context = {
      ...req.body, nextage
    };
    resp.render("age", context);
  });
}

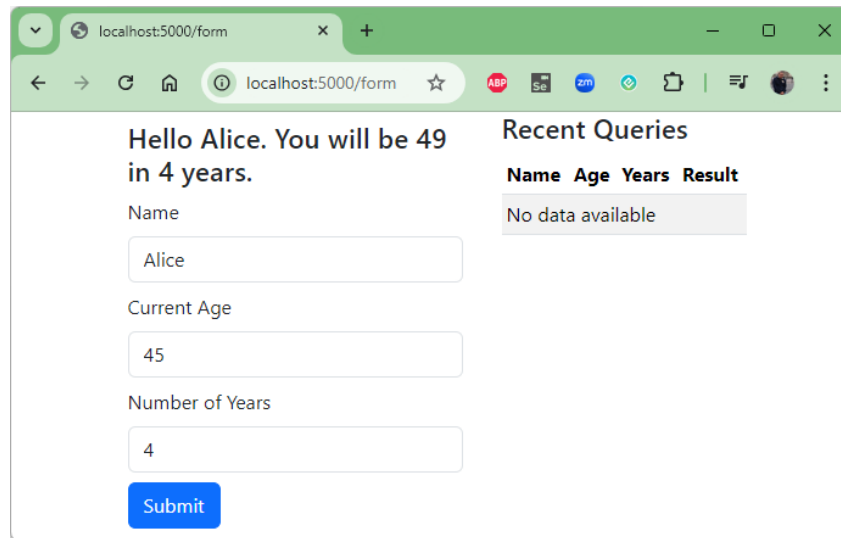
```

Ejecuta el comando que se muestra en el listado 6 en la carpeta `part2app` para iniciar las herramientas de desarrollo.

Listado 12.6: Inicio de las herramientas de desarrollo.

```
npm start
```

Usa un navegador para solicitar `http://localhost:5000`, completa el formulario y haz clic en el botón Enviar (Submit), como se muestra en la figura 1. No se mostrarán datos en la sección Consultas recientes (**Recent Queries**).



Name	Age	Years	Result
No data available			

Figura 1: Ejecución de la aplicación de ejemplo.

## Uso de una base de datos

Las bases de datos permiten que las aplicaciones web lean y escriban datos, que pueden utilizarse para generar respuestas a solicitudes HTTP. Existen muchos tipos de bases de datos, con opciones sobre cómo se almacenan y consultan los datos, cómo se implementa el software de base de datos y cómo se manejan los cambios en los datos.

El mercado de bases de datos es competitivo e innovador, y existen excelentes productos comerciales y de código abierto, pero el mejor consejo es que **la mejor base de datos es aquella que ya comprendes y con la que has trabajado antes**. La mayoría de los proyectos pueden utilizar la mayoría de las bases de datos, y el beneficio que otorga una tecnología de base de datos en particular se verá socavado por el tiempo que se necesita para aprender y dominar esa tecnología.

Si no tienes una base de datos, es fácil perderse entre las infinitas opciones, y nuestro consejo es comenzar con algo lo más simple posible. **Para aplicaciones pequeñas, recomendamos SQLite, que es la base de datos que utilizaremos en este documento**. Para aplicaciones más grandes, especialmente donde se utilizan varias instancias de Node.js para manejar solicitudes HTTP, recomendamos una de las excelentes bases de datos relacionales de código abierto, como:

MySQL (<https://www.mysql.com>) o PostgreSQL (<https://www.postgresql.org>).

Puedes ver un ejemplo de una de esas bases de datos más adelante.

Si no te gusta usar el lenguaje de consulta estructurado (SQL), existen buenas bases de datos NoSQL disponibles y un buen lugar para comenzar es MongoDB (<https://www.mongodb.com>).

---

## Quejas sobre bases de datos

Recibiremos quejas cada vez que escribamos sobre la elección de productos de bases de datos. Muchos desarrolladores tienen opiniones firmes sobre la superioridad de una base de datos en particular o un estilo de base de datos y se enojan cuando no se recomienda su producto preferido.

No es que pensemos que algún motor de base de datos en particular sea malo. De hecho, el mercado de bases de datos nunca ha sido tan bueno, hasta el punto de que casi cualquier producto de base de datos se puede usar en casi cualquier proyecto con poco impacto en la productividad o la escala.

Los motores de bases de datos son como los automóviles: los coches modernos son tan buenos que la mayoría de la gente puede arreglárselas con casi cualquier coche. Si ya tienes un coche, es probable que el beneficio de cambiarlo sea pequeño en comparación con el costo. Si no tienes coche, un buen punto de partida es el coche que tienen la mayoría de tus vecinos y en el que suelen trabajar los mecánicos locales. A algunas personas les gustan mucho los coches y tienen opiniones firmes sobre una marca o modelo en particular, y eso está bien, pero se puede llegar al exceso, y la mayoría de la gente no conduce de un modo en el que las mejoras marginales se vuelvan significativas.

Por lo tanto, comprendemos perfectamente por qué algunos desarrolladores se involucran profundamente en un motor de base de datos en particular (y se respeta ese nivel de compromiso y comprensión), pero la mayoría de los proyectos no tienen los tipos de requisitos de almacenamiento o procesamiento de datos que hacen que las diferencias entre los productos de bases de datos sean importantes.

---

## Instalación del paquete de base de datos

El motor de base de datos utilizado en este documento es SQLite. Opera dentro del proceso Node.js y es una buena opción para aplicaciones donde no es necesario compartir datos entre varias instancias de Node.js, algo que SQLite no admite porque no se ejecuta como un servidor independiente. SQLite se usa ampliamente y es, al menos según <https://sqlite.org>, el motor de base de datos más popular del mundo.

Ejecuta el comando que se muestra en el listado 7 en la carpeta part2app para agregar SQLite al proyecto. No se requieren paquetes de tipos TypeScript adicionales.

Listado 7: Agregar el paquete de base de datos al proyecto.

```
npm install sqlite3@5.1.6
```

Este paquete incluye el motor de base de datos, una API de Node.js y descripciones de esa API para el compilador TypeScript. Para describir la base de datos que se utilizará en esta sección, agrega un archivo llamado age.sql a la carpeta part2app con el contenido que se muestra en el listado 8.

Listado 8: El contenido del archivo age.sql en la carpeta part2app.

```
DROP TABLE IF EXISTS Results;
DROP TABLE IF EXISTS Calculations;
DROP TABLE IF EXISTS People;

CREATE TABLE IF NOT EXISTS `Calculations` (
  id INTEGER PRIMARY KEY AUTOINCREMENT, `age` INTEGER,
  years INTEGER, `nextage` INTEGER);

CREATE TABLE IF NOT EXISTS `People` (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name VARCHAR(255));

CREATE TABLE IF NOT EXISTS `Results` (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  calculationId INTEGER REFERENCES `Calculations` (`id`)
  ON DELETE CASCADE ON UPDATE CASCADE,
  personId INTEGER REFERENCES `People` (`id`)
  ON DELETE CASCADE ON UPDATE CASCADE);

INSERT INTO Calculations (id, age, years, nextage) VALUES
(1, 35, 5, 40), (2, 35, 10, 45);

INSERT INTO People (id, name) VALUES
(1, 'Alice'), (2, "Bob");

INSERT INTO Results (calculationId, personId) VALUES
(1, 1), (2, 2), (2, 1);
```

Las instrucciones SQL del listado 8 crean tres tablas, que registrarán los cálculos de edad realizados por la aplicación. La tabla Calculations realiza un seguimiento de los cálculos de edad que se han realizado y tiene columnas para los valores de edad y año proporcionados por el usuario y la edad futura que se ha calculado. La tabla People realiza un seguimiento de los nombres que proporcionan los usuarios.



La tabla Results realiza un seguimiento de los resultados haciendo referencia a un nombre y un cálculo.

---

### Nota

Los datos con los que trabaja la aplicación no justifican tres tablas, pero los datos simples combinados con varias tablas permiten demostrar más fácilmente algunos problemas comunes.

---

### Creación de una capa de repositorio

Un repositorio es una capa (layer) de código que aísla la base de datos del resto de la aplicación, lo que facilita cambiar la forma en que se leen y escriben los datos sin necesidad de cambiar el código que utiliza esos datos. No todo el mundo considera útil una capa de repositorio, pero el consejo es que la utilices a menos que estés completamente seguro de que el uso de los datos o los productos de la base de datos de tu aplicación no cambiará. Crea la carpeta src/server/data y agrégle un archivo llamado repositorio.ts con el contenido que se muestra en el listado 9.

Listado 9: El contenido del archivo repositorio.ts en la carpeta src/server/data.

```
export interface Result {  
  id: number,  
  name: string,  
  age: number,  
  years: number,  
  nextage: number  
}  
  
export interface Repository {  
  saveResult(r: Result): Promise<number>;  
  
  getAllResults(limit: number) : Promise<Result[]>;  
  
  getResultByName(name: string, limit: number): Promise<Result[]>;  
}
```

La interfaz Repository define métodos para almacenar nuevos objetos Result, consultar todos los resultados y los resultados que tienen un nombre específico. El tipo Result define propiedades para todas las columnas de datos en las tablas de la base de datos en una estructura simple y plana.

Los proyectos pueden utilizar tipos de datos que coincidan con la estructura de la base de datos, pero eso a menudo significa que los datos que llegan del usuario deben ensamblarse en una estructura compleja antes de ser extraídos y utilizados para crear una declaración SQL, mientras que el proceso inverso ensambla los datos de la base de datos en la misma estructura, solo para que se extraigan para su uso en plantillas. No siempre es posible, pero el uso de estructuras de datos simples y planas a menudo simplifica el desarrollo.

## Implementación del repositorio

El siguiente paso es implementar la interfaz `Repository` con una clase que utiliza el motor de base de datos `SQLite`. Vamos a implementar el repositorio en etapas, lo que facilitará la comprensión de la relación entre los datos de la base de datos y los objetos JavaScript en la aplicación.

Para comenzar la implementación, crea un archivo llamado `sql_repository.ts` en la carpeta `src/server/data` con el contenido que se muestra en el listado 10.

Listado 10: El contenido del archivo `sql_repository.ts` en la carpeta `src/server/data`.

```
import { Database } from "sqlite3";
import { Repository, Result } from "../repository";

export class SqlRepository implements Repository {
  db: Database;

  constructor() {
    this.db = new Database("age.db");
    this.db.exec(readFileSync("age.sql").toString(), err => {
      if (err !== undefined) throw err;
    });
  }

  saveResult(r: Result): Promise<number> {
    throw new Error("Method not implemented.");
  }

  getAllResults($limit: number): Promise<Result[]> {
    throw new Error("Method not implemented.");
  }

  getResultByName($name: string, $limit: number): Promise<Result[]> {
    throw new Error("Method not implemented.");
  }
}
```

La clase `SqlRepository` implementa la interfaz `Repository` y su constructor prepara la base de datos. El módulo `sqlite3` contiene la API de la base de datos y crea un nuevo objeto `Database`, especificando `age.db` como nombre de archivo. El objeto `Database` proporciona métodos para utilizar la base de datos y el método `exec` se utiliza para ejecutar instrucciones SQL; en este caso, para ejecutar las instrucciones en el archivo `age.sql`.

---

### Nota

Los proyectos reales no necesitan ejecutar SQL para crear la base de datos cada vez, pero al hacerlo se puede restablecer el ejemplo, y es por esta razón que el SQL en el listado 8 eliminará y recreará las tablas de la base de datos si ya existen. Las bases de datos generalmente se inicializan solo cuando se implementa una aplicación.

---

Para que el repositorio esté disponible para el resto de la aplicación, agrega un archivo llamado `index.ts` a la carpeta `src/server/data` con el contenido que se muestra en el listado 11.

Listado 11: El contenido del archivo `index.ts` en la carpeta `src/server/data`.

```
import { Repository } from "../repository";
import { SqlRepository } from "../sql_repository";

const repository: Repository = new SqlRepository();
export default repository;
```

Este archivo es responsable de instanciar el repositorio para que el resto de la aplicación pueda acceder a los datos a través de la interfaz `Repository` sin necesidad de saber qué implementación se ha utilizado.

### Consulta de la base de datos

El siguiente paso es implementar los métodos que brindan acceso a la base de datos, comenzando con aquellos que consultan datos. Agrega un archivo llamado `sql_queries.ts` a la carpeta `src/server/data`, con el contenido que se muestra en el listado 12.

Listado 12: El contenido del archivo `sql_queries.ts` en la carpeta `src/server/data`.

```
const baseSql = `
  SELECT Results.*, name, age, years, nextage FROM Results
  INNER JOIN People ON personId = People.id
  INNER JOIN Calculations ON calculationId = Calculations.id`;

const endSql = `ORDER BY id DESC LIMIT $limit`;

export const queryAllSql = `${baseSql} ${endSql}`;
```

```
export const queryByNameSql = `${baseSql} WHERE name = $name ${endSql}`;
```

Las consultas SQL se pueden formular como cualquier otra cadena de JavaScript, y la preferencia es evitar la duplicación definiendo una consulta base y luego construyéndola para crear las variaciones necesarias. En este caso, se ha definido cadenas `baseSql` y `endSql`, que se combinan para crear consultas, de modo que la consulta de datos que coincidan con un nombre será la siguiente:

```
...
SELECT Results.*, name, age, years, nextage FROM Results
INNER JOIN People ON personId = People.id
INNER JOIN Calculations ON calculationId = Calculations.id
WHERE name = $name
ORDER BY id DESC LIMIT $limit
...
```

Estas consultas utilizan parámetros nombrados, que se indican con un signo `$` y permiten que se proporcionen valores cuando se ejecuta la consulta. Como explicamos en la sección Comprensión de los parámetros de consulta SQL, esta es una característica que siempre se debe utilizar y que es compatible con todos los paquetes de bases de datos.

No somos administradores de bases de datos profesionales y existen formas más eficientes de crear consultas, pero usar una base de datos es más fácil cuando las consultas devuelven datos que se pueden analizar fácilmente para crear objetos JavaScript. En este caso, las consultas devolverán tablas de datos como esta:

id	calculationId	personId	name	age	years	nextage
1	1	1	Alice	35	5	40
3	2	1	Alice	35	10	45

El paquete `SQLite` convertirá la tabla de datos en un arreglo de objetos JavaScript cuyas propiedades corresponden a los nombres de las columnas de la tabla, como esta:

```
...
{
  id: 1,
  calculationId: 1,
  personId: 1,
  name: "Alice",
  age: 35,
```

```

    years: 5,
    nextage: 40
  }
  ...

```

La estructura de los datos recibidos de la base de datos es un superconjunto de la interfaz Result definida en el listado 9, lo que significa que los datos recibidos de la base de datos se pueden usar sin necesidad de procesamiento adicional. El listado 13 utiliza el SQL definido en el listado 12 para consultar la base de datos.

Listado 13: Consulta de la base de datos en el archivo sql\_repository.ts en la carpeta src/server/data.

```

import { readFileSync } from "fs";
import { Database } from "sqlite3";
import { Repository, Result } from "../repository";
import { queryAllSql, queryByNameSql } from "../sql_queries";

export class SqlRepository implements Repository {
  db: Database;

  constructor() {
    this.db = new Database("age.db");
    this.db.exec(readFileSync("age.sql").toString(), err => {
      if (err !== undefined) throw err;
    });
  }

  saveResult(r: Result): Promise<number> {
    throw new Error("Method not implemented.");
  }

  getAllResults($limit: number): Promise<Result[]> {
    return this.executeQuery(queryAllSql, { $limit });
  }

  getResultsByName($name: string, $limit: number): Promise<Result[]> {
    return this.executeQuery(queryByNameSql, { $name, $limit });
  }

  executeQuery(sql: string, params: any) : Promise<Result[]> {
    return new Promise<Result[]>((resolve, reject) => {
      this.db.all<Result>(sql, params, (err, rows) => {
        if (err == undefined) {
          resolve(rows);
        } else {

```

```

        reject(err);
    }
  })
});
}
}

```

El objeto Database creado en el constructor proporciona métodos para consultar la base de datos. El método `executeQuery` utiliza el método `Database.all`, que ejecuta una consulta SQL y devuelve todas las filas que produce la base de datos. Para una referencia rápida, la tabla 3 describe los métodos más útiles que proporciona la clase Database. La mayoría de estos métodos aceptan valores para los parámetros de consulta, que explicaremos más adelante.

Tabla 3: Métodos útiles de la base de datos.

Nombre	Descripción
<code>run(sql, params, cb)</code>	Este método ejecuta una sentencia SQL con un conjunto opcional de parámetros. No se devuelven datos de resultados. La función de devolución de llamada opcional se invoca si hay un error o cuando se completa la ejecución.
<code>get(sql, params, cb)</code>	Este método ejecuta una sentencia SQL con un conjunto opcional de parámetros y pasa la primera fila de resultados como un objeto a la función de devolución de llamada, con tipo T.
<code>all(sql, params, cb)</code>	Este método ejecuta una sentencia SQL con un conjunto opcional de parámetros y pasa todas las filas de resultados a la función de devolución de llamada como un arreglo de tipo T.
<code>prepare(sql)</code>	Este método crea una sentencia preparada, que se representa con un objeto Statement, y puede mejorar el rendimiento porque la base de datos no tiene que procesar SQL cada vez que se ejecuta la consulta. Este método no acepta parámetros de consulta.

## Visualización de datos

El listado 14 actualiza el código que maneja las solicitudes HTTP para crear una instancia del repositorio SQL y utiliza los métodos que proporciona para consultar la base de datos y pasar los resultados a la plantilla.

Listado 14: Uso del repositorio en el archivo `forms.ts` en la carpeta `src/server`.

```

import express, { Express } from "express";
import repository from "../data";

const rowLimit = 10;

export const registerFormMiddleware = (app: Express) => {

```

```

    app.use(express.urlencoded({extended: true}))
  }

export const registerFormRoutes = (app: Express) => {
  app.get("/form", async (req, resp) => {
    resp.render("age", {
      history: await repository.getAllResults(rowLimit)
    });
  });

  app.post("/form", async (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);
    const context = {
      ...req.body, nextage,
      history: await repository.getResultsByName(
        req.body.name, rowLimit)
    };
    resp.render("age", context);
  });
}

```

La palabra clave `async` se aplica a las funciones del controlador, lo que permite el uso de la palabra clave `await` al llamar a los métodos del repositorio. Los resultados se pasan a la plantilla mediante una propiedad denominada `history`, que se utiliza para rellenar la tabla del listado 15.

Listado 15: Cómo rellenar la tabla en el archivo `history.handlebars` de la carpeta `templates/server/partials`.

```

<h4>Recent Queries</h4>
<table class="table table-sm table-striped my-2">
  <thead>
    <tr>
      <th>Name</th><th>Age</th><th>Years</th><th>Result</th>
    </tr>
  </thead>
  <tbody>
    {{#unless history }}
      <tr><td colspan="4">No data available</td></tr>
    {{/unless }}
    {{#each history }}
      <tr>
        <td>{{ this.name }} </td>
        <td>{{ this.age }} </td>

```

```

        <td>{{ this.years }} </td>
        <td>{{ this.nextage }} </td>
      </tr>
    </tbody>
  </table>

```

Utiliza un navegador para solicitar `http://localhost:5000/form` y verás que el lado derecho muestra los datos de todos los usuarios. Completa y envía el formulario y solo se mostrarán las consultas de ese usuario, como se muestra en la figura 2. Las consultas en la base de datos de otros usuarios ya no se muestran.

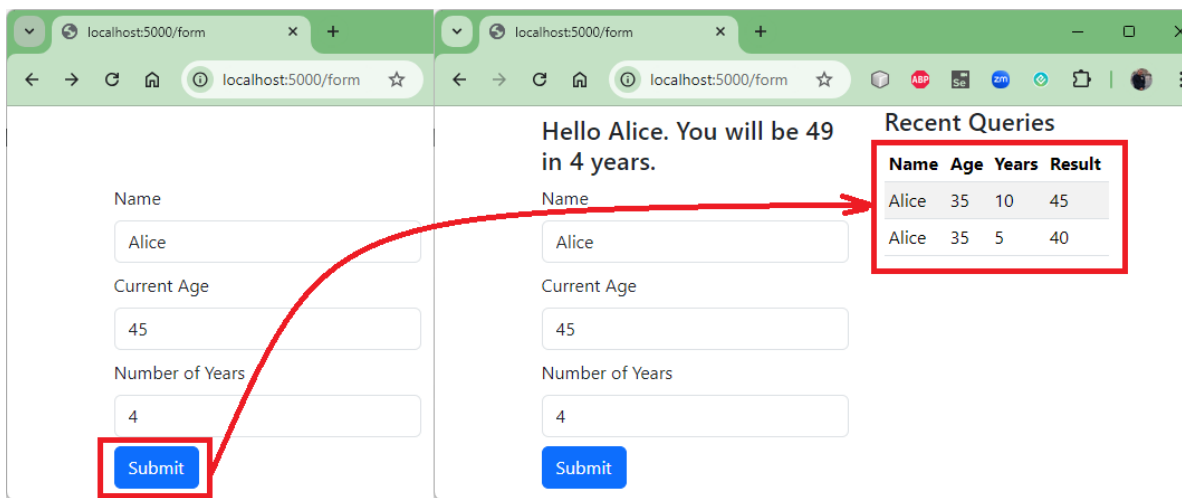


Figura 2: Cómo consultar la base de datos.

## Entender los parámetros de consulta SQL

Se debe tener cuidado al incluir valores recibidos de los usuarios en las consultas SQL. Como demostración, el listado 16 altera la implementación de `getResultsByName` definido por la clase `SQLRepository`.

Listado 16: Inclusión de la entrada del usuario en el archivo `sql_repository.ts` en la carpeta `src/server/data`.

```

...
getResultsByName($name: string, $limit: number): Promise<Result[]> {
  return this.executeQuery(`
    SELECT Results.*, name, age, years, nextage FROM Results
    INNER JOIN People ON personId = People.id
    INNER JOIN Calculations ON calculationId = Calculations.id
    WHERE name = "${$name}"`, {});
}

```



```
}
...
```

El error que se comete en este ejemplo es incluir el valor recibido del formulario directamente en la consulta. Si el usuario ingresa Alice en el formulario, la consulta se verá así:

```
...
SELECT Results.*, name, age, years, nextage FROM Results
  INNER JOIN People ON personId = People.id
  INNER JOIN Calculations ON calculationId = Calculations.id
  WHERE name = "Alice"
...
```

Este es el comportamiento previsto y recupera las consultas realizadas con ese nombre. Pero es fácil crear cadenas que alteren la consulta. Si el usuario ingresa Alice" o name = "Bob, por ejemplo, la consulta se verá así:

```
...
SELECT Results.*, name, age, years, nextage FROM Results
  INNER JOIN People ON personId = People.id
  INNER JOIN Calculations ON calculationId = Calculations.id
  WHERE name = "Alice" OR name = "Bob"
...
```

Esto no es lo que el desarrollador espera y significa que se muestran las consultas realizadas por dos usuarios, como se muestra en la figura 3.

The screenshot shows a web browser window with the URL `localhost:5000/form`. The page has a blue header and a white body. On the left, there is a form with the following elements:

- A heading: "Hello Alice" or name = "Bob. You will be 40 in 10 years."
- A text input field labeled "Name" containing the text "Alice" or name = "Bob".
- A text input field labeled "Current Age" containing the number "30".
- A text input field labeled "Number of Years" containing the number "10".
- A blue "Submit" button.

On the right, there is a section titled "Recent Queries" containing a table with the following data:

Name	Age	Years	Result
Alice	35	5	40
Bob	35	10	45
Alice	35	10	45

Figura 3: Ejecución de una consulta con la entrada del usuario.

Este es un ejemplo benigno, pero muestra que incluir valores proporcionados por el usuario directamente en las consultas permite a los usuarios malintencionados cambiar la forma en que se procesan las consultas.

Este problema no se soluciona con la limpieza de HTML descrita anteriormente, ya que los valores no se limpian hasta que se incluyen en una respuesta. En cambio, las bases de datos proporcionan compatibilidad con parámetros de consulta, que permiten insertar valores en las consultas de forma segura.

El parámetro se define en la consulta SQL y se denota con un carácter \$ inicial, de la siguiente manera:

```
...
export const queryByNameSql = `${baseSql} WHERE name = $name ${endSql}`;
...
```

Esta declaración se combina con la consulta base, lo que significa que la declaración SQL general se ve así:

```
...
SELECT Results.*, name, age, years, nextage FROM Results
  INNER JOIN People ON personId = People.id
  INNER JOIN Calculations ON calculationId = Calculations.id
  WHERE name = $name ORDER BY id DESC LIMIT $limit
...
```

Los dos parámetros de consulta están marcados en negrita e indican valores que se proporcionarán cuando el método `executeQuery` ejecute la declaración en la clase `SqlRepository`:

```
...
executeQuery(sql: string, params: any) : Promise<Result[]> {
  return new Promise<Result[]>((resolve, reject) => {
    this.db.all<RowResult>(sql, params, (err, rows) => {
      if (err == undefined) {
        resolve(rowsToObjects(rows));
      } else {
        reject(err);
      }
    })
  });
}
...
```

El listado 17 revierte los cambios en la clase `SqlRepository` para que la consulta realizada por el método `getResultsByName` use el método `executeQuery` y proporcione parámetros de consulta.

Listado 17: Uso de parámetros de consulta en el archivo `sql_repository.ts` en la carpeta `src/server/data`.

```
...
getResultsByName($name: string, $limit: number): Promise<Result[]> {
    return this.executeQuery(queryByNameSql, { $name, $limit });
}
...
```

El objeto que contiene los valores de los parámetros tiene nombres de propiedad que coinciden con los parámetros en la declaración SQL: `$name` y `$limit`. El signo `$` no es la única forma de indicar un parámetro de consulta en SQL, pero funciona bien con JavaScript porque el signo `$` está permitido en los nombres de variables.

Es por esta razón que el método `getResultsByName` define los parámetros `$name` y `$limit`, lo que permite que los valores se pasen sin necesidad de modificar los nombres.

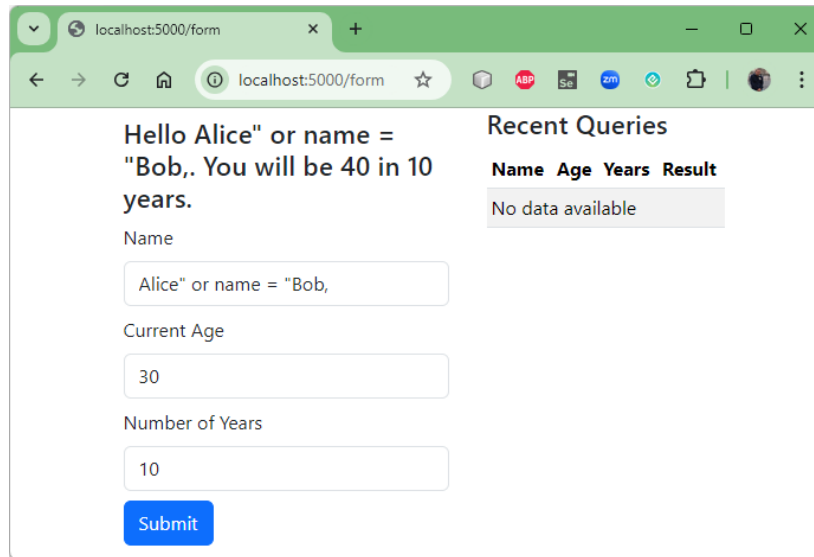
La última pieza del rompecabezas la proporciona el código que maneja los datos del formulario:

```
...
const context = {
    ...req.body, nextage,
    history: await repository.getResultsByName(req.body.name, rowLimit)
};
...
```

El valor que el usuario ingresó para el campo de nombre en el formulario se lee del cuerpo y se utiliza como valor para el parámetro de consulta `$name`. Los métodos descritos en la tabla 3 desinfectan automáticamente los parámetros de consulta, por lo que no alteran la forma en que se ejecuta la consulta, como se muestra en la figura 4.

## Escritura en la base de datos

El siguiente paso es escribir datos para que la base de datos contenga más que solo los datos semilla agregados cuando se crea la base de datos. El listado 18 define las instrucciones SQL que insertarán filas en las tablas de la base de datos.



localhost:5000/form

localhost:5000/form

**Hello Alice" or name = "Bob,. You will be 40 in 10 years.**

Name

Alice" or name = "Bob,

Current Age

30

Number of Years

10

Submit

**Recent Queries**

Name	Age	Years	Result
No data available			

Figura 4: El efecto de un parámetro de consulta desinfectado.

Listado 18: Agregar instrucciones en el archivo `sql_queries.ts` en la carpeta `src/server/data`.

```
const baseSql = `
    SELECT Results.*, name, age, years, nextage FROM Results
    INNER JOIN People ON personId = People.id
    INNER JOIN Calculations ON calculationId = Calculations.id`;

const endSql = `ORDER BY id DESC LIMIT $limit`;

export const queryAllSql = `${baseSql} ${endSql}`;

export const queryByNameSql = `${baseSql} WHERE name = $name ${endSql}`;

export const insertPerson = `
    INSERT INTO People (name)
    SELECT $name
    WHERE NOT EXISTS (SELECT name FROM People WHERE name = $name)`;

export const insertCalculation = `
    INSERT INTO Calculations (age, years, nextage)
    SELECT $age, $years, $nextage
    WHERE NOT EXISTS
    (SELECT age, years, nextage FROM Calculations
    WHERE age = $age AND years = $years AND nextage = $nextage)`;

export const insertResult = `
    INSERT INTO Results (personId, calculationId)
    SELECT People.id as personId, Calculations.id as calculationId from People
```

## CROSS JOIN Calculations

```

WHERE People.name = $name
AND Calculations.age = $age
AND Calculations.years = $years
AND Calculations.nextage = $nextage`;

```

Las instrucciones `insertPerson` e `insertCalculation` insertarán nuevas filas en las tablas `People` y `Calculation` solo si no hay filas existentes que tengan los mismos detalles. La instrucción `insertResult` crea una fila en la tabla `Results`, con referencias a las otras tablas.

Estas instrucciones deben ejecutarse dentro de una transacción para garantizar la coherencia. El motor de base de datos `SQLite` admite transacciones, pero estas no están expuestas de manera conveniente a `Node.js` y se requiere trabajo adicional para ejecutar instrucciones `SQL` en una transacción. Agrega un archivo llamado `sql_helpers.ts` a la carpeta `src/server/data` con el contenido que se muestra en el listado 19.

Listado 19: El contenido del archivo `sql_helpers.ts` en la carpeta `src/server/data`.

```

import { Database } from "sqlite3";

export class TransactionHelper {
  steps: [sql: string, params: any][] = [];

  add(sql: string, params: any): TransactionHelper {
    this.steps.push([sql, params]);
    return this;
  }

  run(db: Database): Promise<number> {
    return new Promise((resolve, reject) => {
      let index = 0;
      let lastRow: number = NaN;
      const cb = (err: any, rowID?: number) => {
        if (err) {
          db.run("ROLLBACK", () => reject());
        } else {
          lastRow = rowID ? rowID : lastRow;
          if (++index === this.steps.length) {
            db.run("COMMIT", () => resolve(lastRow));
          } else {
            this.runStep(index, db, cb);
          }
        }
      };
    });
  }
}

```

```

        db.run("BEGIN", () => this.runStep(0, db, cb));
    });
}
runStep(idx: number, db: Database, cb: (err: any, row: number) => void) {
    const [sql, params] = this.steps[idx];
    db.run(sql, params, function (err: any) {
        cb(err, this.lastID)
    });
}
}
}

```

La clase TransactionHelper define un método add que se utiliza para crear una lista de instrucciones SQL y parámetros de consulta. Cuando se llama al método run, se envía el comando BEGIN a SQLite y se ejecuta cada una de las instrucciones SQL. Si todas las sentencias se ejecutan correctamente, se envía el comando COMMIT y SQLite aplica los cambios a la base de datos. El comando ROLLBACK se envía si alguna de las sentencias falla y SQLite abandona los cambios realizados por sentencias anteriores. SQLite proporciona el ID de la fila modificada por sentencias INSERT y el método run devuelve el valor producido por la sentencia más reciente. Conocer el ID de la fila insertada más recientemente es generalmente una buena idea porque facilita la consulta de nuevos datos.

El Listado 20 utiliza la clase TransactionHelper para realizar una actualización ejecutando las tres sentencias del listado 18 dentro de una transacción SQL.

Listado 20: Inserción de datos en el archivo sql\_repository.ts en la carpeta src/server/data.

```

import { readFileSync } from "fs";
import { Database } from "sqlite3";
import { Repository, Result } from "../repository";
import { queryAllSql, queryByNameSql, insertPerson, insertCalculation, insertResult } from
"./sql_queries";
import { TransactionHelper } from "../sql_helpers";

export class SqlRepository implements Repository {
    db: Database;

    constructor() {
        this.db = new Database("age.db");
        this.db.exec(readFileSync("age.sql").toString(), err => {
            if (err != undefined) throw err;
        });
    }

    async saveResult(r: Result): Promise<number> {
        return await new TransactionHelper()

```

```

        .add(insertPerson, { $name: r.name })
        .add(insertCalculation, {
            $age: r.age, $years: r.years, $nextage: r.nextage
        })
        .add(insertResult, {
            $name: r.name,
            $age: r.age, $years: r.years, $nextage: r.nextage
        })
        .run(this.db);
    }

    getAllResults($limit: number): Promise<Result[]> {
        return this.executeQuery(queryAllSql, { $limit });
    }

    getResultsByName($name: string, $limit: number): Promise<Result[]> {
        return this.executeQuery(queryByNameSql, { $name, $limit });
    }

    executeQuery(sql: string, params: any) : Promise<Result[]> {
        return new Promise<Result[]>((resolve, reject) => {
            this.db.all<Result>(sql, params, (err, rows) => {
                if (err == undefined) {
                    resolve(rows);
                } else {
                    reject(err);
                }
            })
        });
    }
}

```

La implementación del método `saveResult` ejecuta las tres sentencias SQL. Cada sentencia requiere un objeto independiente para sus parámetros de consulta porque SQLite produce un error si hay propiedades sin usar en el objeto de parámetros. El listado 21 actualiza el controlador de solicitudes HTTP POST para escribir datos en la base de datos a través del repositorio.

Listado 21: Escritura de datos en el archivo `forms.ts` en la carpeta `src/server`.

```

import express, { Express } from "express";
import repository from "../data";

const rowLimit = 10;

```

```

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({extended: true}))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", async (req, resp) => {
    resp.render("age", {
      history: await repository.getAllResults(rowLimit)
    });
  });

  app.post("/form", async (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);

    await repository.saveResult({...req.body, nextage });

    const context = {
      ...req.body, nextage,
      history: await repository.getResultsByName(
        req.body.name, rowLimit)
    };
    resp.render("age", context);
  });
}

```

El uso de nombres consistentes para cada parte de la aplicación significa que el cuerpo de la solicitud se puede utilizar como base para la interfaz Result esperada por el repositorio. El efecto es que cada nueva solicitud se almacena en la base de datos y se refleja en la respuesta presentada al usuario, como se muestra en la figura 5.

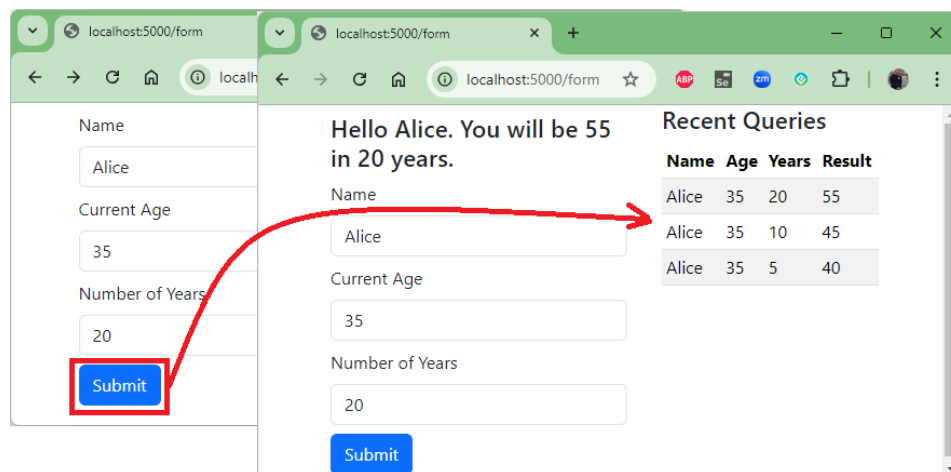


Figura 5: Escritura de datos en la base de datos.



## Uso de un paquete ORM

La ventaja de trabajar directamente con la base de datos es que tú tienes control sobre cómo se escribe y ejecuta cada sentencia. La desventaja es que puede ser un proceso complejo y que requiere mucho tiempo. Una alternativa es utilizar un paquete ORM que se ocupe de la base de datos en nombre del desarrollador, ocultando algunos aspectos de SQL y encargándose del mapeo entre la base de datos y los objetos JavaScript.

La gama de características que ofrecen los paquetes ORM varía ampliamente. Algunos adoptan un enfoque ligero y se centran en la transformación de datos, pero la mayoría de los paquetes se ocupan de la mayoría de los aspectos del uso de una base de datos, incluida la definición del esquema SQL, la creación de la base de datos e incluso la generación de consultas.

Los paquetes ORM pueden ser excelentes, pero aún, así es necesario tener un conocimiento básico de SQL, por eso comenzamos este documento con un ejemplo directo a la base de datos. Los paquetes ORM esperan que el desarrollador comprenda cómo se utilizarán sus características para crear y utilizar bases de datos, y no podrá obtener resultados útiles ni diagnosticar problemas sin algunas habilidades de SQL.

---

### El argumento a favor de las bases de datos de objetos

Una alternativa al uso de SQL y un paquete ORM es utilizar una base de datos que almacene objetos directamente, como MongoDB (<https://www.mongodb.com>). La razón por la que no hemos cubierto las bases de datos de objetos en este curso es que la mayoría de los proyectos utilizan bases de datos relacionales y la mayoría de las empresas estandarizan un motor de base de datos relacional específico.

Las bases de datos de objetos pueden ser una buena opción, pero no son la tecnología que la mayoría de los desarrolladores terminan utilizando. Las bases de datos SQL siguen siendo dominantes, a pesar de que existen algunas alternativas excelentes disponibles.

---

El paquete ORM que utilizaremos en este documento/práctica se llama Sequelize (<https://www.npmjs.com/package/sequelize>), que es el paquete ORM de JavaScript más popular. Sequelize tiene un conjunto completo de características y es compatible con los motores de base de datos más populares, incluido SQLite.

Ejecuta el comando que se muestra en el listado 22 en la carpeta part2app para instalar el paquete Sequelize, que incluye información de tipo TypeScript.

Listado 22: Instalación de los paquetes ORM.

```
npm install sequelize@6.35.1
```

## Definición de la base de datos utilizando objetos JavaScript

Al trabajar directamente con una base de datos, el primer paso es escribir las sentencias SQL que crean las tablas y las relaciones entre ellas, que es como comenzamos este documento. Al utilizar un ORM, la base de datos se describe mediante objetos JavaScript. Cada paquete ORM tiene su propio proceso y, para Sequelize, se requieren tres pasos.

### Creación de las clases del modelo

El primer paso es definir las clases que representarán los datos en la base de datos. Agrega un archivo llamado `orm_models.ts` a la carpeta `src/server/data`, con el contenido que se muestra en el listado 23.

Listado 23: El contenido del archivo `orm_models.ts` en la carpeta `src/server/data`.

```
import { Model, CreationOptional, ForeignKey, InferAttributes, InferCreationAttributes }
from "sequelize";

export class Person extends Model<InferAttributes<Person>,
  InferCreationAttributes<Person>> {

  declare id?: CreationOptional<number>;
  declare name: string
}

export class Calculation extends Model<InferAttributes<Calculation>,
  InferCreationAttributes<Calculation>> {

  declare id?: CreationOptional<number>;
  declare age: number;
  declare years: number;
  declare nextage: number;
}

export class ResultModel extends Model<InferAttributes<ResultModel>,
  InferCreationAttributes<ResultModel>> {

  declare id: CreationOptional<number>;
  declare personId: ForeignKey<Person["id"]>;
  declare calculationId: ForeignKey<Calculation["id"]>;
  declare Person?: InferAttributes<Person>;
  declare Calculation?: InferAttributes<Calculation>;
}
```

Sequelize utilizará cada clase para crear una tabla de base de datos y cada propiedad será una columna en esa tabla. Estas clases también describen los datos en la base de datos al compilador de TypeScript.

Todas las propiedades de clase en el Listado 23 se definen con la palabra clave `declare`, que le indica al compilador de TypeScript que se comporte como si las propiedades se hubieran definido, pero que no incluya esas propiedades en el JavaScript compilado. Esto es importante porque Sequelize agregará getters y setters a los objetos para proporcionar acceso a los datos, y definir propiedades de manera convencional evitará que esa característica funcione correctamente.

---

### Nota

Los nombres de las clases del modelo deben ser significativos. Se ha elegido `Person` y `Calculation`, que son bastante obvias, pero hemos utilizado `ResultModel` para evitar conflictos con el nombre del tipo utilizado por la interfaz `Repository`.

---

Las propiedades de clase cuyo tipo es un tipo de JavaScript regular se representarán mediante columnas regulares en la base de datos, como la propiedad `name` definida por la clase `Person`:

```
...
export class Person extends Model<InferAttributes<Person>, InferCreationAttributes<Person>> {
  declare id?: CreationOptional<number>;
  declare name: string
}
...
```

El tipo `CreationOptional<T>` se utiliza para describir una propiedad que no tiene que proporcionarse cuando se crea una nueva instancia de la clase del modelo, como esto:

```
...
export class Person extends Model<InferAttributes<Person>,
InferCreationAttributes<Person>> {
  declare id?: CreationOptional<number>;
  declare name: string
}
..
```

La propiedad `id` representa la llave primaria de un objeto `Person` cuando se almacena como una fila en una tabla de base de datos. La base de datos se configurará para asignar automáticamente una llave cuando se almacena una nueva fila, por lo que el uso del tipo

`CreationOptional<number>` evitará que TypeScript informe un error cuando se crea un objeto `Person` sin un valor `id`.

La clase base se utiliza para crear una lista de las propiedades definidas por la clase, que se utilizan para aplicar la seguridad de tipos cuando se leen o escriben datos:

```
...
export class Person extends Model<InferAttributes<Person>,
InferCreationAttributes<Person>> {
  declare id?: CreationOptional<number>;
  declare name: string;
}
...
```

El tipo `InferAttributes<Person>` selecciona todas las propiedades definidas por la clase `Person`, mientras que el tipo `InferCreationAttributes<Person>` excluye las propiedades cuyo tipo es `CreationOptional<T>`. Las clases del modelo también contienen propiedades para representar relaciones entre tablas en la base de datos:

```
...
export class ResultModel extends Model<InferAttributes<ResultModel>,
  InferCreationAttributes<ResultModel>> {

  declare id: CreationOptional<number>;
  declare personId: ForeignKey<Person["id"]>;
  declare calculationId: ForeignKey<Calculation["id"]>;

  declare Person?: InferAttributes<Person>;
  declare Calculation?: InferAttributes<Calculation>;
}
...
```

Las propiedades `personId` y `calculateId` almacenarán las llaves primarias de los datos relacionados, mientras que las propiedades `Person` y `Calculation` se completarán con objetos creados por Sequelize, como parte del proceso de hacer que los datos estén disponibles como objetos.

## Inicialización del modelo de datos

El siguiente paso es indicarle a Sequelize cómo debe representarse cada propiedad definida por las clases del modelo en la base de datos. Agrega un archivo llamado `orm_helpers.ts` a la carpeta `src/server/data`, con el contenido que se muestra en el listado 24.

Listado 24: El contenido del archivo `orm_helpers.ts` en la carpeta `src/server/data`.

```
import { DataTypes, Sequelize } from "sequelize";
import { Calculation, Person, ResultModel } from "../orm_models";

const primaryKey = {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  }
};

export const initializeModels = (sequelize: Sequelize) => {
  Person.init({
    ...primaryKey,
    name: { type: DataTypes.STRING }
  }, { sequelize });

  Calculation.init({
    ...primaryKey,
    age: { type: DataTypes.INTEGER },
    years: { type: DataTypes.INTEGER },
    nextage: { type: DataTypes.INTEGER },
  }, { sequelize });

  ResultModel.init({
    ...primaryKey,
  }, { sequelize });
}
```

La clase base del Model utilizada en el listado 24 define el método `init`, que acepta un objeto cuyas propiedades corresponden a las definidas por la clase. A cada propiedad se le asigna un objeto de configuración que le indica a Sequelize cómo representar los datos en la base de datos.

Las tres clases del modelo tienen una propiedad `id` que se configura como la llave principal. Para las otras propiedades, se selecciona un valor de la clase `DataTypes` para especificar el tipo de datos SQL que se utilizará cuando se cree la base de datos.

El segundo argumento aceptado por el método `init` se utiliza para configurar el modelo de datos general.

En el listado 24 solo se especifica la propiedad `sequelize`, que es un objeto `Sequelize` que se creará para administrar la base de datos. Hay otras opciones disponibles, que permiten cambiar el nombre de la tabla de la base de datos, configurar activadores de la base de datos y configurar otras características de la base de datos.

## Configuración de las relaciones (vínculos) del modelo

La clase base `Model` proporciona métodos para describir las relaciones entre las clases del modelo, como se muestra en el listado 25.

Listado 25: Definición de las relaciones del modelo en el archivo `orm_helpers.ts` en la carpeta `src/server/data`.

```
import { DataTypes, Sequelize } from "sequelize";
import { Calculation, Person, ResultModel } from "../orm_models";

const primaryKey = {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  }
};

export const initializeModels = (sequelize: Sequelize) => {
  Person.init({
    ...primaryKey,
    name: { type: DataTypes.STRING }
  }, { sequelize });

  Calculation.init({
    ...primaryKey,
    age: { type: DataTypes.INTEGER },
    years: { type: DataTypes.INTEGER },
    nextage: { type: DataTypes.INTEGER },
  }, { sequelize });

  ResultModel.init({
    ...primaryKey,
  }, { sequelize });
}

export const defineRelationships = () => {
  ResultModel.belongsTo(Person, { foreignKey: "personId" });
}
```

```

    ResultModel.belongsTo(Calculation, { foreignKey: "calculationId" });
  }

```

Sequelize define cuatro tipos de *asociación*, que se utilizan para describir la relación entre las clases del modelo de datos, como se describe en la tabla 4.

Tabla 4: Métodos de asociación de Sequelize.

Nombre	Descripción
hasOne(T, options)	Este método denota una relación de uno a uno entre la clase de modelo y T, con la llave foránea definida en T.
belongsTo(T, options)	Este método denota una relación de uno a uno entre la clase de modelo y T, con la llave foránea definida por la clase de modelo.
hasMany(T, options)	Este método denota una relación de uno a muchos, con la llave foránea definida por T.
belongsToMany(T, options)	Este método denota una relación de muchos a muchos mediante una tabla de unión.

Cada uno de los métodos definidos en la tabla 4 acepta un argumento de opciones que se utiliza para configurar la relación. En el listado 25, la propiedad `foreignKey` se utiliza para especificar la llave foránea en la clase `ResultModel` para las relaciones uno a uno con los tipos `Person` y `Calculation`.

(Existen otras opciones, descritas en:

<https://sequelize.org/api/v6/identifiers.html#associations>).

## Definición de los datos semilla

Aunque los paquetes ORM se encargan de muchos de los detalles, puede haber tareas que se realicen más fácilmente simplemente ejecutando expresiones SQL directamente, en lugar de utilizar objetos JavaScript. Para demostrarlo, el listado 26 utiliza SQL para inicializar la base de datos.

Listado 26: Cómo agregar datos de semilla en el archivo `orm_helpers.ts` en la carpeta `src/server/data`.

```

import { DataTypes, Sequelize } from "sequelize";
import { Calculation, Person, ResultModel } from "../orm_models";
import { Result } from "../repository";

const primaryKey = {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  }
}

```

```

    }
  };

export const initializeModels = (sequelize: Sequelize) => {
  Person.init({
    ...primaryKey,
    name: { type: DataTypes.STRING }
  }, { sequelize });

  Calculation.init({
    ...primaryKey,
    age: { type: DataTypes.INTEGER },
    years: { type: DataTypes.INTEGER },
    nextage: { type: DataTypes.INTEGER },
  }, { sequelize });

  ResultModel.init({
    ...primaryKey,
  }, { sequelize });
}

export const defineRelationships = () => {
  ResultModel.belongsTo(Person, { foreignKey: "personId" });
  ResultModel.belongsTo(Calculation, { foreignKey: "calculationId" });
}

export const addSeedData = async (sequelize: Sequelize) => {
  await sequelize.query(`
    INSERT INTO Calculations
      (id, age, years, nextage, createdAt, updatedAt) VALUES
      (1, 35, 5, 40, date(), date()),
      (2, 35, 10, 45, date(), date())`);

  await sequelize.query(`
    INSERT INTO People (id, name, createdAt, updatedAt) VALUES
      (1, 'Alice', date(), date()), (2, 'Bob', date(), date())`);

  await sequelize.query(`
    INSERT INTO ResultModels (calculationId, personId, createdAt, updatedAt) VALUES
      (1, 1, date(), date()), (2, 2, date(), date()),
      (2, 1, date(), date());`);
}

```



El método `Sequelize.query` acepta una cadena que contiene una declaración SQL. Las declaraciones del listado 26 crean los mismos datos de semilla utilizados anteriormente en el documento, pero con la adición de valores para las columnas `createdAt` y `updatedAt`. Una consecuencia de usar un paquete ORM para crear una base de datos es que a menudo se introducen características y restricciones adicionales y Sequelize agrega estas columnas para realizar un seguimiento de cuándo se crean y modifican las filas de la tabla. Las consultas que crean los datos de semilla utilizan la función `date()`, que devuelve la fecha y hora actuales.

## Cómo convertir modelos de datos en objetos planos

Usar objetos JavaScript para representar datos puede ser una experiencia de desarrollo más natural, pero puede significar que los objetos del modelo de datos no estén en el formato esperado en otras partes de la aplicación. En el caso de la aplicación de ejemplo, los objetos del modelo de datos ORM no cumplen con los requisitos del tipo `Result` utilizado por la interfaz `Repository`. Un enfoque sería modificar la interfaz, pero esto socavaría el beneficio de aislar la base de datos del resto de la aplicación. El listado 27 define una función que transforma los objetos `ResultModel` proporcionados por el paquete ORM en objetos `Result` requeridos por la interfaz `Repository`.

Listado 27: Transformación de datos en el archivo `orm_helpers.ts` en la carpeta `src/server/data`.

```
import { DataTypes, Sequelize } from "sequelize";
import { Calculation, Person, ResultModel } from "../orm_models";
import { Result } from "../repository";

const primaryKey = {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  }
};

//...funciones omitidas por brevedad...
export const fromOrmModel = (model: ResultModel | null) : Result => {
  return {
    id: model?.id || 0,
    name: model?.Person?.name || "",
    age: model?.Calculation?.age || 0,
    years: model?.Calculation?.years || 0,
    nextage: model?.Calculation?.nextage || 0
  }
}
```

Este tipo de transformación puede parecer complicado, pero JavaScript facilita la composición de nuevos objetos de esta manera, y es una técnica útil que facilita la integración entre módulos y paquetes, algo con lo que la mayoría de los proyectos de JavaScript tienen que lidiar.

## Implementación del repositorio

Todo está listo y es momento de implementar la interfaz del repositorio. Agrega un archivo llamado `orm_repository.ts` a la carpeta `src/server/data` con el contenido que se muestra en el listado 28, que configura el ORM pero aún no implementa consultas ni almacena datos.

Listado 28: El contenido del archivo `orm_repository.ts` en la carpeta `src/server/data`.

```
import { Sequelize } from "sequelize";
import { Repository, Result } from "../repository";
import { addSeedData, defineRelationships, fromOrmModel, initializeModels } from
"./orm_helpers";
import { Calculation, Person, ResultModel } from "../orm_models";

export class OrmRepository implements Repository {
  sequelize: Sequelize;

  constructor() {
    this.sequelize = new Sequelize({
      dialect: "sqlite",
      storage: "orm_age.db",
      logging: console.log,
      logQueryParameters: true
    });
    this.initModelAndDatabase();
  }

  async initModelAndDatabase() : Promise<void> {
    initializeModels(this.sequelize);
    defineRelationships();
    await this.sequelize.drop();
    await this.sequelize.sync();
    await addSeedData(this.sequelize);
  }

  async saveResult(r: Result): Promise<number> {
    throw new Error("Method not implemented.");
  }
}
```

```

    async getAllResults(limit: number): Promise<Result[]> {
      throw new Error("Method not implemented.");
    }

    async getResultsByName(name: string, limit: number): Promise<Result[]> {
      throw new Error("Method not implemented.");
    }
  }
}

```

Sequelize admite una variedad de motores de base de datos, incluido SQLite, por lo que el primer paso es crear un objeto Sequelize, proporcionando un objeto de configuración que especifica el motor de base de datos y las opciones para su uso. En el listado 28, la opción de dialecto especifica SQLite y la opción de almacenamiento especifica el nombre del archivo. Al utilizar un ORM, puede resultar útil ver las consultas SQL que se generan, por lo que se configuran las opciones logging y logQueryParameters.

Una vez que se ha creado un objeto Sequelize, se puede configurar. El método `initModelAndDatabase` llama a las funciones `initializeModels` y `defineRelationships` para configurar los objetos del modelo de datos y, a continuación, llama a estos métodos:

```

...
await this.sequelize.drop();
await this.sequelize.sync();
...

```

El método `drop` le indica a Sequelize que elimine las tablas de la base de datos. Esto no es algo que se deba hacer en un proyecto real, pero recrea los ejemplos anteriores de este documento. El método `sync` le indica a Sequelize que sincronice la base de datos con los objetos del modelo de datos, lo que tiene el efecto de crear tablas para los datos `ResultModel`, `Person` y `Calculation`. Una vez que se han creado las tablas, se llama a la función `addSeedData` para agregar los datos iniciales a la base de datos. Algunas de estas operaciones son asíncronas, por lo que se realizan con la palabra clave `await` dentro de un método `async`.

## Consultas de datos

Las consultas en un ORM se realizan utilizando una API que devuelve objetos, sin ninguna interacción directa con el SQL que se envía a la base de datos. Los paquetes ORM tienen diferentes filosofías sobre cómo se expresan las consultas.

Con Sequelize, las consultas se realizan utilizando las clases del modelo de datos, con métodos que se heredan de la clase base `Model`, los más útiles de los cuales se describen en

la tabla 5. (El conjunto completo de características del Modelo se puede encontrar en <https://sequelize.org/api/v6/class/src/model.js~model>.)

Tabla 5: Métodos útiles del Modelo.

Nombre	Descripción
findAll	Este método busca todos los registros coincidentes y los presenta como objetos de modelo.
findOne	Este método busca el primer registro coincidente y lo presenta como un objeto de modelo.
findByPk	Este método busca el registro con una llave primaria especificada.
findOrCreate	Este método busca un registro coincidente o crea uno si no hay ninguna coincidencia.
create	Este método crea un nuevo registro.
update	Este método actualiza los datos en la base de datos.
upsert	Este método actualiza una sola fila de datos o crea una fila si no hay ninguna coincidencia.

Los métodos de la tabla 5 se configuran con un objeto de configuración que cambia la forma en que se ejecuta la consulta o actualización. Las propiedades de configuración más útiles se describen en la tabla 6.

Tabla 6: Propiedades de configuración de consultas útiles.

Nombre	Descripción
include	Esta propiedad carga datos de tablas relacionadas siguiendo llaves foráneas.
where	Esta propiedad se utiliza para limitar una consulta, que se pasa a la base de datos mediante la palabra clave WHERE de SQL.
order	Esta propiedad configura el orden de la consulta, que se pasa a la base de datos mediante las palabras clave ORDER BY de SQL.
group	Esta propiedad especifica la agrupación de la consulta, que se pasa a la base de datos mediante las palabras clave GROUP BY de SQL.
limit	Esta propiedad especifica la cantidad de registros necesarios, que se pasa a la base de datos mediante la palabra clave LIMIT de SQL.
transaction	Esta propiedad realiza la consulta dentro de la transacción especificada, como se muestra en la sección Escritura de datos.
attributes	Esta propiedad restringe los resultados, de modo que incluyan solo los atributos o columnas especificados.

El listado 29 muestra una consulta básica de Sequelize que implementa el método `getAllResults`.

Listado 29: Realización de una consulta en el archivo `orm_repository.ts` en la carpeta `src/server/data`.

```
...
async getAllResults(limit: number): Promise<Result[]> {
  return (await ResultModel.findAll({
    include: [Person, Calculation],
    limit,
    order: [["id", "DESC"]]
  })).map(row => fromOrmModel(row));
}
...
```

El método `findAll` se llama en la clase `ResultModel` y se configura con un objeto que tiene propiedades `include`, `limit` y `order`. La propiedad más importante es `include`, que le indica a Sequelize que siga las relaciones de clave externa para cargar datos relacionados y crear objetos a partir de los resultados.

En este caso, el resultado será un objeto `ResultModel` cuyas propiedades `Person` y `Calculation` están completas. La propiedad `limit` restringe la cantidad de resultados y la propiedad `order` se utiliza para especificar cómo se ordenan los resultados.

La consulta se realiza de forma asíncrona y el resultado es una Promesa que genera un arreglo de objetos `ResultModel`, que se asignan a los objetos `Result` requeridos por la interfaz `Repository` mediante la función `fromOrmModel` definida en el listado 27.

La propiedad de configuración `where` se puede utilizar para seleccionar datos específicos, como se muestra en el Listado 30, que implementa el método `getResultsByName`.

Listado 30: Búsqueda de datos en el archivo `orm_repository.ts` en la carpeta `src/server/data`.

```
...
async getResultsByName(name: string, limit: number): Promise<Result[]> {
  return (await ResultModel.findAll({
    include: [Person, Calculation],
    where: {
      "$Person.name$": name
    },
    limit, order: [["id", "DESC"]]
  })).map(row => fromOrmModel(row));
}
...
```

Esta es la misma consulta utilizada en el listado 29, pero con la adición de la propiedad `where`, que le indica a Sequelize que siga la relación de clave externa y que haga coincidir los objetos

Person mediante la propiedad name. La sintaxis de la propiedad where puede requerir algo de tiempo para acostumbrarse, pero verá ejemplos adicionales en capítulos posteriores.

## Escritura de datos

El listado 31 completa el repositorio implementando el método saveResult, que solo almacena los objetos Person y Calculation si no hay datos coincidentes en la base de datos y realiza todos sus cambios mediante una transacción.

Listado 31: Escritura de datos en el archivo orm\_repository.ts en la carpeta src/server/data.

```
...
async saveResult(r: Result): Promise<number> {
  return await this.sequelize.transaction(async (tx) => {

    const [person] = await Person.findOrCreate({
      where: { name : r.name },
      transaction: tx
    });

    const [calculation] = await Calculation.findOrCreate({
      where: {
        age: r.age, years: r.years, nextage: r.nextage
      },
      transaction: tx
    });

    return (await ResultModel.create({
      personId: person.id, calculationId: calculation.id,
      {transaction: tx})).id;
    });
  }
}
...
```

La transacción se crea con el método Sequelize.transaction, que acepta una función de devolución de llamada que recibe un objeto Transaction. La propiedad transaction se utiliza para inscribir cada operación en la transacción, que se confirmará o revertirá automáticamente.

Dentro de la transacción, se utiliza el método findOrCreate para ver si hay objetos Person y Calculation en la base de datos que coincidan con los datos recibidos por el método saveResult. El resultado es el objeto existente, si lo hay, o el objeto recién creado si no hay coincidencia.

Se debe almacenar un nuevo objeto `ResultModel` para cada solicitud, y esto se hace mediante el método `create`. Los valores de las propiedades `personId` y `calculateId` se establecen utilizando los resultados del método `findOrCreate` y la operación de escritura se inscribe en la transacción. No se requiere ningún valor para la propiedad `id`, que será asignada por la base de datos cuando se almacenen los nuevos datos y que está contenida en el resultado del método `create`.

---

### Nota

El método `create` permite crear y almacenar objetos en un solo paso. Una alternativa es utilizar el método `build`, que crea un objeto de modelo que no se almacena hasta que se llama al método `save`, que permite realizar cambios antes de que los datos se escriban en la base de datos.

---

## Aplicación del repositorio

El beneficio de usar un repositorio es que los detalles de cómo se almacenan los datos se pueden cambiar sin afectar las partes de la aplicación que usan esos datos. Para completar la transición al paquete ORM, el listado 32 reemplaza el repositorio existente con el ORM.

Listado 32: Uso del repositorio ORM en el archivo `index.ts` en la carpeta `src/server/data`.

```
import { Repository } from "../repository";
//import { SqlRepository } from "../sql_repository";
import { OrmRepository } from "../orm_repository";

const repository: Repository = new OrmRepository();
export default repository;
```

No se requieren otros cambios porque el repositorio aísla la administración de datos de las plantillas y el código de manejo de solicitudes. Use un navegador para solicitar `http://localhost:5000` y verá los datos iniciales. Completa y envíe el formulario y verá la respuesta que se muestra en la figura 6, que muestra que los datos se han almacenado en la base de datos. Si examinas la salida de la consola de Node.js, verá las consultas SQL que Sequelize está formulando a partir de las operaciones realizadas en los objetos del modelo de datos.

Archivo completo `orm_reposiory.ts` de la carpeta `src/server/data`.

```
import { Sequelize } from "sequelize";
import { Repository, Result } from "../repository";
import { addSeedData, defineRelationships, fromOrmModel, initializeModels } from
"./orm_helpers";
import { Calculation, Person, ResultModel } from "../orm_models";
```

```

export class OrmRepository implements Repository {
  sequelize: Sequelize;

  constructor() {
    this.sequelize = new Sequelize({
      dialect: "sqlite",
      storage: "orm_age.db",
      logging: console.log,
      logQueryParameters: true
    });
    this.initModelAndDatabase();
  }

  async initModelAndDatabase() : Promise<void> {
    initializeModels(this.sequelize);
    defineRelationships();
    await this.sequelize.drop();
    await this.sequelize.sync();
    await addSeedData(this.sequelize);
  }

  async saveResult(r: Result): Promise<number> {
    return await this.sequelize.transaction(async (tx) => {

      const [person] = await Person.findOrCreate({
        where: { name : r.name },
        transaction: tx
      });

      const [calculation] = await Calculation.findOrCreate({
        where: {
          age: r.age, years: r.years, nextage: r.nextage
        },
        transaction: tx
      });

      return (await ResultModel.create({
        personId: person.id, calculationId: calculation.id,
        {transaction: tx})).id;
    });
  }

  async getAllResults(limit: number): Promise<Result[]> {

```



```

    return (await ResultModel.findAll({
      include: [Person, Calculation],
      limit,
      order: [["id", "DESC"]]
    })).map(row => fromOrmModel(row));
  }

  async getResultsByName(name: string, limit: number): Promise<Result[]> {
    return (await ResultModel.findAll({
      include: [Person, Calculation],
      where: {
        "$Person.name$": name
      },
      limit, order: [["id", "DESC"]]
    })).map(row => fromOrmModel(row));
  }
}

```

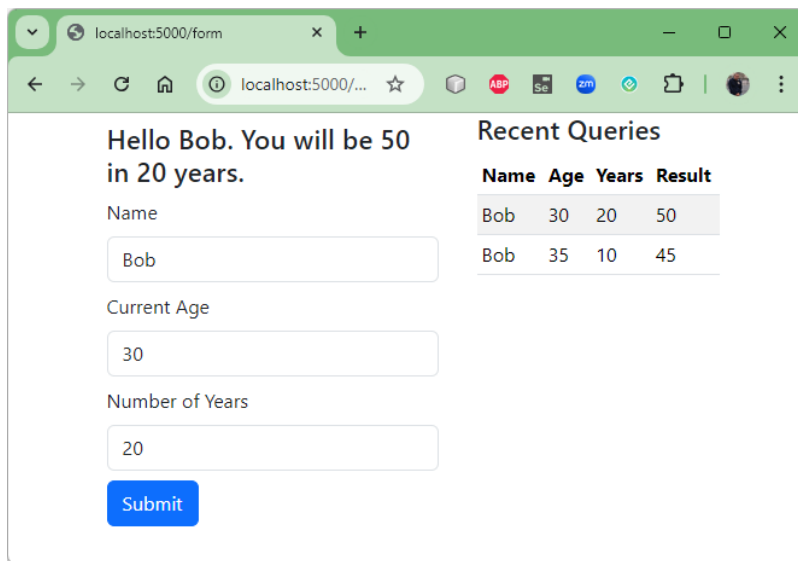


Figura 6: Uso de un paquete ORM.

## Resumen

En este documento, explicamos cómo una aplicación web JavaScript puede usar una base de datos, tanto directamente usando SQL como indirectamente usando un paquete ORM.

- Las bases de datos son la opción más común para el almacenamiento persistente de datos.

- Node.js se puede usar con motores de bases de datos populares, para los cuales existe una amplia gama de paquetes de código abierto.
- Las bases de datos se pueden usar directamente o mediante paquetes que expresan datos como objetos y generan consultas automáticamente.
- Un conocimiento básico de cómo funcionan las bases de datos y la capacidad de comprender la sintaxis core SQL facilitan el trabajo con bases de datos, incluso cuando se usa un paquete ORM.

En el próximo documento/práctica, se describirá cómo se pueden identificar las solicitudes HTTP relacionadas para crear sesiones.