

Creación de servicios web RESTful

En este documento se explica cómo se puede utilizar Node.js para crear servicios web que proporcionen acceso a datos mediante solicitudes HTTP, lo que es un elemento clave para las aplicaciones de una sola página (SPA, *Single-page applications*). El documento comienza con un servicio web básico y luego incorpora funciones más complejas, como actualizaciones parciales y validación de datos. La tabla 1 pone este documento en contexto. La tabla 2 resume el documento/práctica.

Tabla 1: Poner los servicios web RESTful en contexto.

| Pregunta | Respuesta |
|---------------------------------------|---|
| ¿Qué son? | Los servicios web RESTful proporcionan acceso a datos mediante solicitudes HTTP. En lugar de enviar datos incrustados en contenido HTML, el servidor responde con datos “sin procesar”, generalmente en formato JSON. |
| ¿Por qué son útiles? | Los servicios web permiten a los clientes realizar operaciones de datos, como consultar o actualizar datos, mediante solicitudes HTTP. Esto se utiliza con mayor frecuencia mediante código JavaScript que se ejecuta en el navegador, aunque cualquier tipo de cliente puede consumir un servicio web. |
| ¿Cómo se utilizan? | El método/verbo de solicitud HTTP se utiliza para indicar una operación y la URL de solicitud identifica los datos en los que se deben realizar las operaciones. |
| ¿Existen dificultades o limitaciones? | No existe una forma estándar de crear un servicio web, lo que genera variaciones significativas en la forma en que se diseñan. |
| ¿Existen alternativas? | La mayoría de las aplicaciones web modernas requieren algún tipo de servicio web para entregar datos a las aplicaciones JavaScript del lado del cliente. Dicho esto, los servicios web no son necesarios para las aplicaciones que son puramente de ida y vuelta y que no necesitan dar soporte a los clientes. |

Tabla 2: Resumen del documento/práctica.

| Problema | Solución | Listado |
|---|---|--------------|
| Definir un servicio web. | Utilizar los controladores de solicitud estándar y devolver datos JSON en lugar de HTML. | 9-15 |
| Consolidar el código necesario para crear un servicio web. | Separar el código que maneja las solicitudes HTTP para que el código de manejo de datos pueda aislarse. | 16-18, 43-45 |
| Actualizar los datos con un servicio web. | Manejar las solicitudes PUT y PATCH. | 19-26 |
| Describir cambios de datos complejos. | Utilizar la especificación JSON Patch. | 27-30 |
| Validar los valores de datos recibidos por el servicio web. | Realizar la validación antes de pasar los datos al código que los procesa. | 31-37 |

| | |
|--|------------------------------------|
| Validar las combinaciones de datos que recibe el servicio web. | Realizar la validación del modelo. |
|--|------------------------------------|

38-41

Preparación para este documento

Este documento utiliza el proyecto part2app del documento anterior. Los ejemplos de este documento son más fáciles de entender con una aplicación cliente de línea de comandos simple que envía solicitudes HTTP y muestra las respuestas que se reciben. Para prepararse, ejecuta los comandos que se muestran en el listado 1 en la carpeta part2app para instalar el paquete Inquirer (<https://github.com/SBoudrias/Inquirer.js>), que proporciona funciones para solicitar información al usuario.

Listado 14.1: Instalación de un paquete.

```
npm install @inquirer/prompts@3.3.0
```

Crea la carpeta src/cmdline y agrégle un archivo llamado main.mjs, con el contenido que se muestra en el listado 2. La extensión del archivo .mjs le indica a Node.js que trate este archivo como un módulo de JavaScript y permita el uso de la declaración import.

Listado 2: El contenido del archivo main.mjs en la carpeta src/cmdline.

```
import { select } from "@inquirer/prompts";
import { ops } from "../operations.mjs";

(async function run() {
  let loop = true;
  while (loop) {
    const selection = await select({
      message: "Select an operation",
      choices: [...Object.keys(ops).map(k => { return { value: k } })]
    });
    await ops[selection]();
  }
})();
```

Este código utiliza el paquete Inquirer para solicitar al usuario que elija una operación a realizar. Las opciones que se le presentan al usuario se obtienen de las propiedades de un objeto y, al hacer una elección, se ejecuta la función asignada a esa propiedad.

Agrega un archivo llamado operations.mjs a la carpeta src/cmdline con el contenido que se muestra en el listado 3.

Listado 3: El contenido del archivo `operations.mjs` en la carpeta `src/cmdline`.

```
export const ops = {
  "Test": () => {
    console.log("Test operation selected");
  },
  "Exit": () => process.exit()
}
```

Este archivo proporciona las operaciones que el usuario puede seleccionar, con una operación de prueba para comenzar y asegurarse de que todo funciona como debería, y una opción de salida que utiliza el método de Node.js `process.exit` para finalizar el proceso. El listado 4 agrega una entrada a la sección de scripts del archivo `package.json` para ejecutar el cliente de línea de comandos. Por brevedad no se muestra todo el archivo.

Listado 4: Agregar un script en el archivo `package.json` en la carpeta `part2app`.

```
{
  "name": "part2app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "server": "tsc-watch --noClear --onSuccess \"node dist/server/server.js\"",
    "client": "webpack serve",
    "start": "npm-run-all --parallel server client",
    "cmdline": "node --watch ./src/cmdline/main.mjs"
  },
  ...
}
```

La nueva entrada ejecutará el archivo `main.mjs` utilizando Node.js. El argumento `--watch` pone a Node.js en modo de observación, donde se reiniciará si se detectan cambios.

Preparación para un servicio web

Para prepararse para la introducción de un servicio web, crea la carpeta `src/server/api` y agrégale un archivo llamado `index.ts` con el contenido que se muestra en el listado 5.

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

```
import { Express } from "express";

export const createApi = (app: Express) => {
  // TODO - implement API
}
```

Este archivo es solo un marcador de posición por ahora, pero se usará para configurar Express para que gestione las solicitudes de la API HTTP. El cambio final es llamar a la función definida en el listado 5 para configurar el servicio web, como se muestra en el listado 6.

Listado 6: Configuración de Express en el 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";
import { createApi } from "../api";

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);

createApi(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}`));
```

Ejecuta el comando que se muestra en el listado 7 en la carpeta part2app para iniciar las herramientas de desarrollo.

Listado 7: Inicio de las herramientas de desarrollo.


```
npm start
```

Abre un segundo símbolo del sistema, navegue hasta la carpeta part2app y ejecuta el comando que se muestra en el listado 8 para iniciar el cliente de línea de comandos.

Listado 8: Inicio del cliente de línea de comandos.

```
npm run cmdline
```

Las características proporcionadas por el paquete Inquirer presentan una única opción, como esta:



```
Restarting './src/cmdline/main.mjs'  
? Select an operation (Use arrow keys)  
> Test  
Exit
```

Utiliza las teclas de flecha para desplazarte hacia arriba y hacia abajo en la lista de opciones. Si seleccionas la operación de prueba (Test), se mostrará un mensaje de prueba y, si seleccionas Salir (Exit), finalizará el proceso. Node.js se ejecuta en modo de observación, lo que significa que iniciará nuevamente el cliente de línea de comandos si se detecta un cambio. Presione Ctrl + C si deseas detener el cliente por completo.

Comprensión de los servicios web

No existe un acuerdo definitivo sobre qué es un servicio web, no hay un estándar único a seguir ni un conjunto de patrones ampliamente adoptados. Lo cierto es lo opuesto: existe una multitud infinita de opiniones, innumerables patrones y una interminable disputa en Internet sobre la forma “correcta” de entregar datos a los clientes.

El caos y el ruido que rodea a los servicios web pueden ser abrumadores y puede ser difícil saber por dónde empezar. Sin embargo, la falta de estandarización puede ser liberadora porque significa que un proyecto puede centrarse en brindar solo la funcionalidad que requieren los clientes, sin ninguno de los códigos repetitivos ni los costos generales que a veces puede traer la estandarización.

Los servicios web son solo una API de acceso a datos a las que se accede a través de HTTP. Un servicio web RESTful es simplemente un servicio web que utiliza aspectos de las solicitudes HTTP para determinar qué partes de la API desea utilizar un cliente.

El término RESTful proviene del patrón de transferencia de estado representacional (REST, *Representational state transfer*), pero ha habido tanta variación y adaptación en los servicios web que solo se utiliza ampliamente la premisa central de REST, que es que una API se define utilizando una combinación de métodos HTTP y URL. El método HTTP, como GET o POST, define el tipo de operación que se realizará, mientras que la URL especifica el objeto u objetos de datos a los que se aplicará la operación.

Los proyectos tienen la libertad de crear API de servicios web de cualquier manera, pero los mejores servicios web son los que son simples y fáciles de usar. Como ejemplo, aquí hay una URL que podría identificar datos administrados por la aplicación:

/api/results/1

No hay restricciones sobre cómo se utiliza la URL para identificar datos, siempre que tanto el cliente como el servidor comprendan el formato de la URL para que los datos se puedan identificar de forma inequívoca. Si una aplicación almacena datos en una base de datos, una URL generalmente identifica un valor específico mediante una llave primaria, pero esa es solo una convención común y no un requisito.

La URL identifica los datos, pero es el método de solicitud HTTP el que especifica qué se debe hacer con esos datos. La tabla 3 describe los métodos HTTP que se utilizan comúnmente en los servicios web y las operaciones que representan convencionalmente.

Tabla 3: Métodos HTTP de uso común.

| Método | Descripción |
|--------|--|
| GET | Este método se utiliza para recuperar uno o más valores de datos. |
| POST | Este método se utiliza para almacenar un nuevo valor de datos. |
| PUT | Este método se utiliza para reemplazar un valor de datos existente. |
| PATCH | Este método se utiliza para actualizar parte de un valor de datos existente. |
| DELETE | Este método se utiliza para eliminar un valor de datos. |

Un servicio web presenta una API combinando URL y métodos y normalmente devolverá datos JSON. Para las operaciones que no consultan datos, se devuelve una indicación del resultado, que también puede ser datos JSON. Un servicio web básico puede proporcionar las combinaciones descritas en la tabla 4, que también describe los resultados que producirá el servicio web.

Nota

Los primeros servicios web utilizaban XML en lugar de JSON. JSON se convirtió en el estándar de facto porque es simple y los clientes JavaScript lo analizan fácilmente, pero aún verás la referencia ocasional a XML, como los objetos XMLHttpRequest, que los navegadores proporcionan para enviar solicitudes HTTP (aunque estos han sido reemplazados por la API Fetch más moderna).

Tabla 4: Un servicio web típico.

| Método | URL | Descripción |
|--------|-------------------------|--|
| GET | /api/results/1 | Esta combinación obtiene el valor único con ID 1, expresado como una representación JSON de un objeto Result. Si no existe tal ID, se devolverá una respuesta 404. |
| GET | /api/results | Esta combinación obtiene todos los valores de datos disponibles, expresados como una representación JSON de un arreglo de objetos Result. Si no hay datos, se devolverá un arreglo vacío. |
| GET | /api/results?name=Alice | Esta combinación busca todos los valores con un valor de nombre de Alice y devuelve una representación JSON de un arreglo de objetos Result. Se devolverá un arreglo vacío si no hay datos coincidentes. |
| POST | /api/results | Esta combinación almacena un valor y devuelve una representación JSON de los datos almacenados. |
| DELETE | /api/results/1 | Esta combinación elimina el valor único con ID 1 y devuelve un objeto JSON con una propiedad de éxito con un valor booleano que indica el resultado. |

Entender los microservicios

Cualquier investigación sobre servicios web te llevará rápidamente al mundo de los microservicios, por eso lo sugerimos como término de búsqueda en un navegador web. Los microservicios son una forma de diseñar aplicaciones en torno a capacidades comerciales y que a menudo involucran servicios web. Puedes encontrar una buena descripción general de los microservicios en <https://microservices.io>, junto con patrones de diseño detallados.

Mi opinión personal sobre los microservicios es que son interesantes, pero se deben evitar para la mayoría de los proyectos. El problema principal que abordan los microservicios es una organización de desarrollo disfuncional que no se puede gestionar para proporcionar lanzamientos de software coordinados. Este es un problema que enfrentan muchos proyectos, dado que cualquier grupo de tres o más desarrolladores se divide inmediatamente en facciones que compiten por los recursos, discuten sobre problemas de diseño y se culpan entre sí por los retrasos.

Los microservicios intentan resolver estos problemas haciendo que los equipos de desarrollo trabajen en gran medida de forma aislada y acuerden solo cómo se integrarán

las diferentes partes del proyecto. Hay algunas herramientas excelentes diseñadas para dar soporte a los microservicios, la más conocida es Kubernetes, pero las herramientas son increíblemente complejas y adoptar microservicios parece como renunciar a las complejidades de la gestión del personal para centrarse en las complejidades de la gestión del software.

En mi experiencia con los proyectos de cursos y tesis, pocos problemas de RH se han resuelto aumentando la complejidad de las herramientas de desarrollo, por lo que soy un tanto escéptico de que los microservicios sean una forma práctica de resolver problemas organizacionales complejos. Debes formar tu propia opinión, pero mi consejo es que pienses detenidamente antes de adoptar microservicios y te preguntes si tus colegas se comportarán mejor en un modelo de desarrollo federado de lo que lo hacen hoy.

Creación de un servicio web RESTful básico

Como primer paso, el listado 9 crea un servicio web que implementa algunas de las combinaciones de métodos URL y HTTP descritas en la tabla 4.

Listado 9: Creación de un servicio web básico en el archivo index.ts de la carpeta src/server/api.

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

export const createApi = (app: Express) => {
  app.get("/api/results", async (req, resp) => {
    if (req.query.name) {
      const data = await repository.getResultsByName(
        req.query.name.toString(), 10);
      if (data.length > 0) {
        resp.json(data);
      } else {
        resp.writeHead(404);
      }
    } else {
      resp.json(await repository.getAllResults(10));
    }
    resp.end();
  });
}
```

El listado muestra lo fácil que es crear una API para clientes reutilizando las partes de la aplicación creadas para solicitudes de ida y vuelta (round-trip requests). Así es como comienzan la mayoría de los servicios web, pero hay algunos problemas y mejoras que se

pueden realizar, como se explica en secciones posteriores. Pero, para completar el proceso inicial, el listado 10 agrega operaciones al cliente para consumir la API.

Listado 10: Agregar operaciones en el archivo `operations.mjs` de la carpeta `src/cmdline`.

```
import { input } from "@inquirer/prompts";

const baseUrl = "http://localhost:5000";

export const ops = {
  "Get All": () => sendRequest("GET", "/api/results"),

  "Get Name": async () => {
    const name = await input({ message: "Name?" });
    await sendRequest("GET", `/api/results?name=${name}`);
  },

  "Exit": () => process.exit()
}

const sendRequest = async (method, url, body, contentType) => {
  const response = await fetch(baseUrl + url, {
    method, headers: { "Content-Type": contentType ?? "application/json" },
    body: JSON.stringify(body)
  });
  if (response.status === 200) {
    const data = await response.json();
    (Array.isArray(data) ? data : [data])
      .forEach(elem => console.log(JSON.stringify(elem)));
  } else {
    console.log(response.status + " " + response.statusText);
  }
}
```

Node.js admite la API Fetch, que se usa comúnmente en el código JavaScript basado en navegador para realizar solicitudes HTTP. Los cambios en el listado 10 agregan una función `sendRequest` que envía solicitudes HTTP y muestra sus resultados, y agrega las operaciones `Get All` y `Get Name`. La operación `Get Name` usa `Inquirer` para solicitar un nombre, que luego se agrega a la cadena de consulta de la solicitud HTTP.

El cliente de línea de comandos se reiniciará cuando se detecten los cambios en el listado. Si seleccionas la opción `Obtener todo` y presiona `Retorno`, se mostrarán todos los datos disponibles, de esta manera:

```
Restarting './src/cmdline/main.mjs'  
? Select an operation Get All  
{ "id": 3, "name": "Alice", "age": 35, "years": 10, "nextage": 45 }  
{ "id": 2, "name": "Bob", "age": 35, "years": 10, "nextage": 45 }  
{ "id": 1, "name": "Alice", "age": 35, "years": 5, "nextage": 40 }  
? Select an operation (Use arrow keys)  
> Get All  
  Get Name  
  Exit
```

Selecciona la operación Obtener nombre y presiona la tecla Retorno, y se te solicitará un nombre. Ingresa Alice y presione Retorno, y verás los resultados coincidentes:

```
? Select an operation Get Name  
? Name? Alice  
{ "id": 3, "name": "Alice", "age": 35, "years": 10, "nextage": 45 }  
{ "id": 1, "name": "Alice", "age": 35, "years": 5, "nextage": 40 }  
? Select an operation (Use arrow keys)  
> Get All  
  Get Name  
  Exit
```

Si ingresas un nombre que no existe, el servicio web responderá con una respuesta 404 No encontrado.

Obtención de datos para el servicio web

La razón por la que el servicio web solo admite dos de las combinaciones de la tabla 4 es que esas son las únicas operaciones que se pueden realizar utilizando el repositorio, que se creó para las necesidades de la aplicación de ida y vuelta.

No existe una única mejor manera de abordar este problema y se requieren concesiones. Algunos proyectos tienen requisitos de servicio web y de ida y vuelta que son lo suficientemente similares como para compartir un repositorio, pero son poco frecuentes y tratar de forzar la coherencia entre ambos puede terminar comprometiendo una o ambas partes de la aplicación.

Entendiendo GraphQL

GraphQL (<https://graphql.org>) es un enfoque diferente para proporcionar datos a los clientes. Un servicio web RESTful normal proporciona un conjunto específico de operaciones que producen los mismos resultados para todos los clientes. Si un cliente necesita datos adicionales en las respuestas, por ejemplo, entonces un desarrollador debe modificar el servicio web y luego todos los clientes recibirán esos nuevos datos.

GraphQL todavía utiliza solicitudes HTTP y los datos todavía se expresan como JSON, pero los clientes pueden ejecutar consultas personalizadas, que incluyen la selección de los

valores de datos que se incluirán y el filtrado de datos de diferentes maneras. Esto significa que los clientes pueden recibir solo los datos que necesitan y diferentes clientes pueden recibir datos diferentes.

GraphQL es genial, pero es más complejo que un servicio web RESTful normal, tanto en términos de desarrollo del lado del servidor como de realización de consultas de clientes, y la mayoría de los proyectos se adaptan mejor a los servicios web RESTful convencionales, que presentan un conjunto fijo de operaciones y resultados a todos los clientes. GraphQL destaca en proyectos que tienen grandes cantidades de datos y clientes que van a utilizar esos datos de formas muy diferentes, que los desarrolladores del lado del servidor no pueden anticipar. Pero, para la mayoría de los demás proyectos, GraphQL es demasiado complejo y un servicio web convencional es más sencillo de crear y consumir.

Una alternativa es crear repositorios separados para cada parte de la aplicación, lo que permite que cada una evolucione de forma independiente, pero inevitablemente conduce a cierto grado de duplicación de código, ya que es probable que algunas operaciones sean necesarias tanto para los clientes de ida y vuelta como para los clientes de servicios web.

Otra alternativa (y la que se utiliza en este documento/práctica) es crear una subclase del repositorio original y agregar las características faltantes. Esto funciona cuando las características requeridas por una parte de la aplicación son un subconjunto de las requeridas en otras partes, que es el caso de la aplicación de ejemplo. El listado 11 define una nueva interfaz que describe las características adicionales requeridas para el servicio web. Se necesitarán más métodos más adelante, pero esto es suficiente por el momento.

Sugerencia

Si no estás seguro de por dónde empezar, comienza creando una subclase. Si descubres que necesitas reemplazar la mayoría de las características heredadas de la clase base, entonces debes dividir el código en dos repositorios separados.

Listado 11: Definición de una nueva interfaz en el 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[]>;
  }

  export interface ApiRepository extends Repository {

    getResultById(id: number): Promise<Result | undefined>;

    delete(id: number) : Promise<boolean>;
  }

```

Los nuevos métodos permiten solicitar un objeto Result individual por su ID y eliminar datos especificando un ID. El listado 12 actualiza la clase OrmRepository para implementar la nueva interfaz.

Listado 12: Implementación de la interfaz en el archivo orm_repository.ts en la carpeta src/server/data.

```

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

export class OrmRepository implements ApiRepository {
  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));
}

async getResultById(id: number): Promise<Result | undefined> {
  const model = await ResultModel.findByPk(id, {
    include: [Person, Calculation ]
  });
  return model ? fromOrmModel(model): undefined;
}

```

```

    }

    async delete(id: number): Promise<boolean> {
      const count = await ResultModel.destroy({ where: { id }});
      return count == 1;
    }
  }
}

```

El listado 13 actualiza las exportaciones del módulo de datos para agregar el repositorio específico de la API.

Listado 13: Actualización de las exportaciones en el archivo index.ts en la carpeta src/server/data.

```

import { ApiRepository } from "../repository";
import { OrmRepository } from "../orm_repository";

const repository: ApiRepository = new OrmRepository();
export default repository;

```

El listado 14 utiliza la nueva interfaz de repositorio para agregar funciones al servicio web. Este código es difícil de leer, pero se abordará en la siguiente sección.

Listado 14: Adición de funciones en el archivo index.ts en la carpeta src/server/api.

```

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

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

  app.get("/api/results", async (req, resp) => {
    if (req.query.name) {
      const data = await repository.getResultsByName(
        req.query.name.toString(), 10);
      if (data.length > 0) {
        resp.json(data);
      } else {
        resp.writeHead(404);
      }
    } else {
      resp.json(await repository.getAllResults(10));
    }
    resp.end();
  });
}

```

```

app.all("/api/results/:id", async (req, resp) => {
  const id = Number.parseInt(req.params.id);
  if (req.method === "GET") {
    const result = await repository.getResultById(id);
    if (result === undefined) {
      resp.writeHead(404);
    } else {
      resp.json(result);
    }
  } else if (req.method === "DELETE") {
    let deleted = await repository.delete(id);
    resp.json({ deleted });
  }
  resp.end();
})

app.post("/api/results", async (req, resp) => {
  const { name, age, years } = req.body;
  const nextage = Number.parseInt(age) + Number.parseInt(years);
  const id = await repository.saveResult({ id: 0, name, age,
    years, nextage });
  resp.json(await repository.getResultById(id));
  resp.end();
});
}

```

Las nuevas rutas agregan compatibilidad para realizar consultas por ID, almacenar nuevos resultados y eliminar resultados existentes.

La capacidad de almacenar nuevos resultados depende de la capacidad de realizar consultas por ID, ya que existe una discrepancia entre el resultado devuelto por el método `Repository.saveResult` y el resultado requerido por el servicio web para las solicitudes POST. El método `saveResult` devuelve el Id del objeto recién almacenado, por lo que se requiere una consulta adicional para obtener el objeto `Result` que se ha almacenado para que pueda enviarse de vuelta al cliente. El listado 15 agrega nuevas operaciones al cliente de línea de comandos que dependen de las nuevas características del servicio web.

Listado 15: Adición de características en el archivo `operations.mjs` en la carpeta `src/cmdline`.

```

import { input } from "@inquirer/prompts";

const baseUrl = "http://localhost:5000";

export const ops = {
  "Get All": () => sendRequest("GET", "/api/results"),

```

```

"Get Name": async () => {
  const name = await input({ message: "Name?" });
  await sendRequest("GET", `/api/results?name=${name}`);
},

"Get ID": async () => {
  const id = await input({ message: "ID?" });
  await sendRequest("GET", `/api/results/${id}`);
},

"Store": async () => {
  const values = {
    name: await input({ message: "Name?" }),
    age: await input({ message: "Age?" }),
    years: await input({ message: "Years?" })
  };
  await sendRequest("POST", "/api/results", values);
},

"Delete": async () => {
  const id = await input({ message: "ID?" });
  await sendRequest("DELETE", `/api/results/${id}`);
},

"Exit": () => process.exit()
}

const sendRequest = async (method, url, body, contentType) => {
  const response = await fetch(baseUrl + url, {
    method, headers: { "Content-Type": contentType ?? "application/json" },
    body: JSON.stringify(body)
  });
  if (response.status === 200) {
    const data = await response.json();
    (Array.isArray(data) ? data : [data])
      .forEach(elem => console.log(JSON.stringify(elem)));
  } else {
    console.log(response.status + " " + response.statusText);
  }
}

```

Utilizando el cliente de línea de comandos, selecciona la opción Obtener Id e ingresa 3 cuando se te solicite, lo que producirá el siguiente resultado:


```
? Select an operation Get ID
? ID? 3
{"id":3,"name":"Alice","age":35,"years":10,"nextage":45}
? Select an operation (Use arrow keys)
```

El servicio web devolverá una respuesta 404 No encontrado para los ID que no existen en la base de datos. Selecciona la opción Almacenar (Store) e ingresa Drew, 50 y 5 cuando se te soliciten los valores de nombre, edad y años, y la respuesta mostrará el nuevo registro que se almacenó:

```
? Select an operation Get ID
? ID? 3
{"id":3,"name":"Alice","age":35,"years":10,"nextage":45}
? Select an operation Store
? Name? Drew
? Age? 50
? Years? 5
{"id":4,"name":"Drew","age":50,"years":5,"nextage":55}
? Select an operation (Use arrow keys)
```

Si seleccionas la opción Obtener todo (Get All), se mostrará el nuevo registro junto con los datos existentes en la base de datos (pero ten en cuenta que la base de datos se restablece y se vuelve a inicializar cada vez que se reinicia la aplicación del lado del servidor, por lo que no realices ningún cambio de código):

```
? Select an operation Get All
{"id":4,"name":"Drew","age":50,"years":5,"nextage":55}
{"id":3,"name":"Alice","age":35,"years":10,"nextage":45}
{"id":2,"name":"Bob","age":35,"years":10,"nextage":45}
{"id":1,"name":"Alice","age":35,"years":5,"nextage":40}
? Select an operation (Use arrow keys)
```

Selecciona la opción Eliminar (Delete) e ingresa el ID del elemento recién almacenado cuando se te solicite. El resultado es un objeto JSON con una propiedad eliminada que indica el resultado:

```
? Select an operation Delete
? ID? 4
{"deleted":true}
? Select an operation (Use arrow keys)
```

Si seleccionas la opción Obtener todo (Get All), se confirmará que los datos se eliminaron.

Entendiendo OpenAPI

La especificación OpenAPI (<https://www.openapis.org>) es un estándar para describir servicios web, que puede ayudar a los desarrolladores del lado del cliente a entender cómo se pretende utilizar un servicio web y proporciona una descripción de los datos a los que proporciona acceso. Hay herramientas y paquetes disponibles que generan código del lado del cliente automáticamente a partir de una descripción de OpenAPI, y algunos paquetes de JavaScript utilizados para definir servicios web generarán automáticamente documentos OpenAPI.

OpenAPI es una buena idea, pero a menudo se utiliza como sustituto de la documentación descriptiva, que tiende a dejar un vacío entre las características que proporciona un servicio web y cómo el desarrollador pretendía que se utilizaran. Si adoptas OpenAPI en tu proyecto, debes asegurarte de complementar la descripción que produce con notas que expliquen cómo se debe consumir tu servicio web.

Separación del código HTTP

El servicio web del listado 15 admite todas las combinaciones de método HTTP y URL descritas en la tabla 4, pero el código es difícil de entender. El servicio web tiene tres tareas que realizar:

Analizar (parsear) la solicitud HTTP, realizar una operación y preparar la respuesta HTTP. Cuando el mismo código es responsable de todas estas tareas, puede resultar difícil identificar las instrucciones que realizan las operaciones porque se pierden en todo el procesamiento HTTP.

El resultado también tiende a estar centrado en HTTP, es decir, la mayoría de los desarrolladores terminan escribiendo código con la menor cantidad posible de rutas, y eso complica aún más los resultados. Puedes ver esto en el listado 14, donde se utiliza el método Express all para hacer coincidir todas las solicitudes de una ruta URL, con el método HTTP identificado en el controlador de solicitudes, de esta manera:

```
app.all("/api/results/:id", async (req, resp) => {
  const id = Number.parseInt(req.params.id);
  if (req.method === "GET") {
    const result = await repository.getResultById(id);
    if (result === undefined) {
      resp.writeHead(404);
    } else {
      resp.json(result);
    }
  } else if (req.method === "DELETE") {
    let deleted = await repository.delete(id);
    resp.json({ deleted });
  }
  resp.end();
})
```

Terminamos escribiendo este tipo de código todo el tiempo. El código se compila y el servicio web funciona, pero es difícil de mantener porque los diferentes aspectos del servicio web están entrelazados.

Los servicios web se escriben y mantienen más fácilmente si el código que maneja las solicitudes HTTP se extrae en un adaptador, con el beneficio adicional de que los servicios web que requieren el mismo conjunto de métodos HTTP y formatos de URL pueden usar el mismo código de adaptador. Para describir la funcionalidad de un servicio web, agrega un archivo llamado `http_adapter.ts` a la carpeta `src/server/api` con el código del listado 16.

Listado 16: El contenido del archivo `http_adapter.ts` en la carpeta `src/server/api`.

```
import { Express, Response } from "express";

export interface WebService<T> {
  getOne(id: any) : Promise<T | undefined>;
  getMany(query: any) : Promise<T[]>;
  store(data: any) : Promise<T | undefined>;
  delete(id: any): Promise<boolean>;
}

export function createAdapter<T>(app: Express, ws: WebService<T>, baseUrl: string) {
  app.get(baseUrl, async (req, resp) => {
    try {
      resp.json(await ws.getMany(req.query));
      resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
  });

  app.get(`${baseUrl}/:id`, async (req, resp) => {
    try {
      const data = await ws.getOne((req.params.id));
      if (data == undefined) {
        resp.writeHead(404);
      } else {
        resp.json(data);
      }
      resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
  });

  app.post(baseUrl, async (req, resp) => {
    try {
      const data = await ws.store(req.body);
```

```

        resp.json(data);
        resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
});

app.delete(`${baseUrl}/${id}`, async (req, resp) => {
    try {
        resp.json(await ws.delete(req.params.id));
        resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
});

const writeErrorResponse = (err: any, resp: Response) => {
    console.error(err);
    resp.writeHead(500);
    resp.end();
}
}

```

La interfaz `WebService<T>` describe un servicio web que opera en el tipo `T`, con métodos que describen las operaciones necesarias para admitir las características básicas del servicio web. La función `createAdapter<T>` crea rutas `Express` que dependen de los métodos `WebService<T>` para producir resultados. Para crear una implementación de la interfaz `WebService<T>` para los datos de resultados, agrega un archivo llamado `results_api.ts` a la carpeta `src/server/api` con el contenido que se muestra en el listado 17.

Nota

Generalmente definimos funciones de JavaScript usando la sintaxis de flecha gruesa porque parece más natural. Sin embargo, usamos la palabra clave `function` en el listado 16 para definir la función `createAdapter<T>`, porque la forma en que se expresan los parámetros de tipo TypeScript en las funciones de flecha gruesa parece extraña. La firma de función equivalente en forma de flecha gruesa es:

```
export const createAdapter = <T>(app: Express, ws: WebService<T>, baseUrl: string) => {
```

Colocar el parámetro de tipo después del signo igual parece chocante, aunque tú eres libre de seguir cualquiera de las sintaxis según tus preferencias.

Listado 17: El contenido del archivo `results_api.ts` en la carpeta `src/server/api`.

```

import { WebService } from "../http_adapter";
import { Result } from "../data/repository";
import repository from "../data";

```

```

export class ResultWebService implements WebService<Result> {

  getOne(id: any): Promise<Result | undefined> {
    return repository.getResultById(Number.parseInt(id));
  }

  getMany(query: any): Promise<Result[]> {
    if (query.name) {
      return repository.getResultsByName(query.name, 10);
    } else {
      return repository.getAllResults(10);
    }
  }

  async store(data: any): Promise<Result | undefined> {
    const { name, age, years } = data;
    const nextage = Number.parseInt(age) + Number.parseInt(years);
    const id = await repository.saveResult({ id: 0, name, age, years, nextage });
    return await repository.getResultById(id);
  }

  delete(id: any): Promise<boolean> {
    return repository.delete(Number.parseInt(id));
  }
}

```

La clase ResultWebService implementa la interfaz WebService<Result> e implementa los métodos mediante las características del repositorio. El listado 18 utiliza el nuevo adaptador para registrar el servicio web, reemplazando el código mixto.

Listado 18: Uso del adaptador en el archivo index.ts en la carpeta src/server/api.

```

import { Express } from "express";
//import repository from "../data";
import { createAdapter } from "./http_adapter";
import { ResultWebService } from "../results_api";

export const createApi = (app: Express) => {
  createAdapter(app, new ResultWebService(), "/api/results");
}

```

No hay cambios en el comportamiento del servicio web, pero eliminar el código que se ocupa de las solicitudes y respuestas HTTP hace que el servicio web sea más fácil de entender y mantener.

Actualización de datos

Hay dos formas de admitir actualizaciones en los servicios web: reemplazar datos y aplicar parches a los datos. Se envía una solicitud HTTP PUT cuando el cliente desea reemplazar datos por completo y el cuerpo de la solicitud contiene todos los datos que necesitará el servicio web para el reemplazo. Se utiliza un método HTTP PATCH cuando el cliente desea modificar datos y el cuerpo de la solicitud contiene una descripción de cómo se deben modificar esos datos.

Admitir actualizaciones con solicitudes PUT es más simple de implementar, pero requiere que el cliente proporcione un reemplazo completo de los datos almacenados. Las solicitudes PATCH son más complejas, pero ofrecen más flexibilidad y pueden ser más eficientes porque solo se envían los cambios al servicio web.

Consejo

Puede resultar difícil saber qué enfoque adoptar al comienzo de un nuevo proyecto cuando se desconocen los tipos de actualizaciones que enviarán los clientes. El consejo es comenzar a respaldar las actualizaciones completas porque son más simples de implementar y pasar a las actualizaciones parciales solo si descubres que los valores de datos sin cambios comienzan a superar en número a los valores modificados.

Este documento muestra las solicitudes PUT y PATCH. Para prepararse, el listado 19 agrega un nuevo método a la interfaz ApiRepository que permitirá actualizar los datos.

Listado 19: Agregar un método en el 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[]>;
}

export interface ApiRepository extends Repository {
```

```

    getResultById(id: number): Promise<Result | undefined>;

    delete(id: number) : Promise<boolean>;

    update(r: Result) : Promise<Result | undefined>
  }

```

El listado 20 implementa este método utilizando el paquete ORM Sequelize.

Listado 20: Actualizar los datos en el archivo orm_repository.ts en la carpeta src/server/data.

```

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

export class OrmRepository implements ApiRepository {
  sequelize: Sequelize;

  //...constructor y métodos omitidos por brevedad...

  async update(r: Result) : Promise<Result | undefined > {
    const mod = await this.sequelize.transaction(async (transaction) => {
      const stored = await ResultModel.findByPk(r.id);
      if (stored !== null) {
        const [person] = await Person.findOrCreate({
          where: { name : r.name }, transaction
        });
        const [calculation] = await Calculation.findOrCreate({
          where: {
            age: r.age, years: r.years, nextage: r.nextage
          }, transaction
        });
        stored.personId = person.id;
        stored.calculationId = calculation.id;
        return await stored.save({transaction});
      }
    });
    return mod ? this.getResultById(mod.id) : undefined;
  }
}

```

Actualizar los datos en el ejemplo significa cambiar el nombre o el cálculo asociado con un resultado. La implementación del método de actualización realiza una actualización en cuatro pasos, todos los cuales se realizan como una transacción. El primer paso es leer los datos que se van a actualizar de la base de datos utilizando la propiedad `id` del parámetro `Result`:

```
...  
const stored = await ResultModel.findByPk(r.id);  
...
```

Si hay una entrada coincidente en la base de datos, se utiliza el método `findOrCreate` para localizar los datos de `Person` y `Calculation` que coinciden con el parámetro `Result` o crear nuevos datos si no hay coincidencias. El siguiente paso es actualizar los ID para que los datos almacenados hagan referencia a los nuevos registros de `Persona` y `Cálculo` y escribir los cambios en la base de datos, lo que se hace utilizando el método `save`:

```
...  
stored.personId = person.id;  
stored.calculationId = calculation.id;  
return await stored.save({transaction});  
...
```

El método `save` es lo suficientemente inteligente como para detectar cambios y solo actualizará la base de datos para las propiedades cuyos valores hayan cambiado. El paso final se realiza después de que se haya confirmado la transacción y devuelve los datos modificados utilizando el método `getResultById`.

Reemplazo de datos con solicitudes PUT

Las solicitudes PUT son las más simples de implementar porque el servicio web simplemente utiliza los datos enviados por el cliente para reemplazar los datos almacenados.

El listado 21 extiende la interfaz que describe los servicios web para agregar un nuevo método y extiende el contenedor HTTP para utilizar el método de interfaz para manejar las solicitudes PUT.

Nota

No todos los servicios web utilizan solicitudes PUT para actualizaciones. Las solicitudes POST se utilizan a menudo tanto para almacenar datos nuevos como para actualizarlos, utilizando la URL para diferenciar entre operaciones, de modo que la URL utilizada para una actualización incluirá un ID único (`/api/results/1`) y la URL utilizada para no almacenar datos (`/api/results`).

Listado 21: Adición de métodos en el archivo `http_adapter.ts` en la carpeta `src/server/api`.

```
import { Express, Response } from "express";

export interface WebService<T> {
  getOne(id: any) : Promise<T | undefined>;
  getMany(query: any) : Promise<T[]>;
  store(data: any) : Promise<T | undefined>;
  delete(id: any): Promise<boolean>;
  replace(id: any, data: any): Promise<T | undefined>;
}

export function createAdapter<T>(app: Express, ws: WebService<T>, baseUrl: string) {

  //...rutas omitidas por brevedad.

  app.put(`${baseUrl}/:id`, async (req, resp) => {
    try {
      resp.json(await ws.replace(req.params.id, req.body));
      resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
  });

  const writeErrorResponse = (err: any, resp: Response) => {
    console.error(err);
    resp.writeHead(500);
    resp.end();
  }
}
```

El método `replace` agregado a la interfaz `WebService<T>` acepta un `id` y un objeto de datos. La nueva ruta hace coincidir las solicitudes con el método `PUT`, extrae el `ID` de la URL y utiliza el cuerpo de la solicitud para los datos. Implementar el método en el servicio web es cuestión de recibir los datos del contenedor `HTTP` y pasarlos al repositorio, como se muestra en el listado 22.

Listado 22: Reemplazo de datos en el archivo `results_api.ts` en la carpeta `src/server/api`.

```
import { WebService } from "../http_adapter";
import { Result } from "../data/repository";
import repository from "../data";

export class ResultWebService implements WebService<Result> {

  getOne(id: any): Promise<Result | undefined> {
    return repository.getResultById(Number.parseInt(id));
  }
}
```

```

    }

    getMany(query: any): Promise<Result[]> {
      if (query.name) {
        return repository.getResultsByName(query.name, 10);
      } else {
        return repository.getAllResults(10);
      }
    }
  }

  async store(data: any): Promise<Result | undefined> {
    const { name, age, years } = data;
    const nextage = age + years;
    const id = await repository.saveResult({ id: 0, name, age, years, nextage });
    return await repository.getResultById(id);
  }

  delete(id: any): Promise<boolean> {
    return repository.delete(Number.parseInt(id));
  }

  replace(id: any, data: any): Promise<Result | undefined> {
    const { name, age, years, nextage } = data;
    return repository.update({ id, name, age, years, nextage });
  }
}

```

Los datos recibidos del contenedor (wrapper) HTTP se deconstruyen en valores constantes que se combinan con el parámetro id y se pasan al método de actualización del repositorio. El último paso es agregar una operación al cliente de línea de comandos que enviará la solicitud PUT, como se muestra en el listado 23.

Listado 23: Soporte de actualizaciones en el archivo operations.mjs en la carpeta src/cmdline.

```

...
export const ops = {

  //...propiedades/funciones omitidas por brevedad...

  "Replace": async () => {
    const id = await input({ message: "ID?" });
    const values = {
      name: await input({ message: "Name?" }),
      age: await input({ message: "Age?" }),
      years: await input({ message: "Years?" }),
    }
  }
}

```

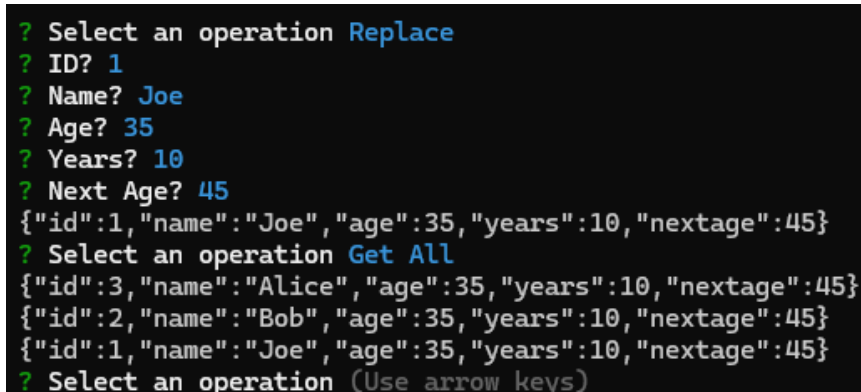
```

        nextage: await input({message: "Next Age?"})
    };
    await sendRequest("PUT", `/api/results/${id}`, values);
},

"Exit": () => process.exit()
}
...

```

La operación se llama Reemplazar (Replace) y solicita todos los valores necesarios para almacenar los datos y los envía al servicio web mediante una solicitud HTTP PUT. Selecciona la nueva opción Replace en la línea de comandos e ingresa 1, Joe, 35, 10 y 45 cuando se te solicite. Esta operación actualizará el resultado cuyo ID es 1 con un nuevo nombre, como este:



```

? Select an operation Replace
? ID? 1
? Name? Joe
? Age? 35
? Years? 10
? Next Age? 45
{"id":1,"name":"Joe","age":35,"years":10,"nextage":45}
? Select an operation Get All
{"id":3,"name":"Alice","age":35,"years":10,"nextage":45}
{"id":2,"name":"Bob","age":35,"years":10,"nextage":45}
{"id":1,"name":"Joe","age":35,"years":10,"nextage":45}
? Select an operation (Use arrow keys)

```

Se cambia el nombre pero los otros valores son los mismos, por lo que la relación entre el resultado y el cálculo en la base de datos permanece sin cambios.

Modificación de datos con solicitudes PATCH

Las solicitudes PATCH permiten que un cliente solicite a un servidor web que aplique actualizaciones parciales, sin tener que enviar un registro de datos completo. No existe una forma estándar de describir cambios parciales en una solicitud PATCH y se puede utilizar cualquier formato de datos, siempre y cuando tanto el cliente como el servicio web comprendan cómo se identifican los datos y cómo se describen los cambios. Para admitir solicitudes PATCH, el listado 24 agrega un nuevo método a la interfaz del servicio web y define una ruta que coincide con las solicitudes PATCH.

Listado 24: Compatibilidad con solicitudes PATCH en el archivo `http_adapter.ts` en la carpeta `src/server/api`.

```

import { Express, Response } from "express";

export interface WebService<T> {
  getOne(id: any) : Promise<T | undefined>;
  getMany(query: any) : Promise<T[]>;
  store(data: any) : Promise<T | undefined>;
  delete(id: any): Promise<boolean>;
  replace(id: any, data: any): Promise<T | undefined>;
  modify(id: any, data: any): Promise<T | undefined>;
}

export function createAdapter<T>(app: Express, ws: WebService<T>, baseUrl: string) {

  app.get(baseUrl, async (req, resp) => {
    try {
      resp.json(await ws.getMany(req.query));
      resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
  });

  app.get(`/${baseUrl}/:id`, async (req, resp) => {
    try {
      const data = await ws.getOne((req.params.id));
      if (data == undefined) {
        resp.writeHead(404);
      } else {
        resp.json(data);
      }
      resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
  });

  app.post(baseUrl, async (req, resp) => {
    try {
      const data = await ws.store(req.body);
      resp.json(data);
      resp.end();
    } catch (err) { writeErrorResponse(err, resp) }
  });

  app.delete(`/${baseUrl}/:id`, async (req, resp) => {
    try {
      resp.json(await ws.delete(req.params.id));
      resp.end();
    }
  });
}

```

```

    } catch (err) { writeErrorResponse(err, resp) }
  });

app.put(`${baseUrl}/${id}`, async (req, resp) => {
  try {
    resp.json(await ws.replace(req.params.id, req.body));
    resp.end();
  } catch (err) { writeErrorResponse(err, resp) }
});

app.patch(`${baseUrl}/${id}`, async (req, resp) => {
  try {
    resp.json(await ws.modify(req.params.id, req.body));
    resp.end();
  } catch (err) { writeErrorResponse(err, resp) }
});

const writeErrorResponse = (err: any, resp: Response) => {
  console.error(err);
  resp.writeHead(500);
  resp.end();
}
}

```

La forma más sencilla de admitir actualizaciones parciales es permitir que el cliente proporcione un objeto JSON que contenga solo valores de reemplazo y omita cualquier propiedad que deba dejarse sin cambios, como se muestra en el listado 25.

Listado 25: Modificación de datos en el archivo results_api.ts en la carpeta src/server/api.

```

import { WebService } from "../http_adapter";
import { Result } from "../data/repository";
import repository from "../data";

export class ResultWebService implements WebService<Result> {

  getOne(id: any): Promise<Result | undefined> {
    return repository.getResultById(Number.parseInt(id));
  }

  getMany(query: any): Promise<Result[]> {
    if (query.name) {
      return repository.getResultsByName(query.name, 10);
    } else {
      return repository.getAllResults(10);
    }
  }
}

```

```

    }
  }

  async store(data: any): Promise<Result | undefined> {
    const { name, age, years } = data;
    const nextage = age + years;
    const id = await repository.saveResult({ id: 0, name, age, years, nextage });
    return await repository.getResultById(id);
  }

  delete(id: any): Promise<boolean> {
    return repository.delete(Number.parseInt(id));
  }

  replace(id: any, data: any): Promise<Result | undefined> {
    const { name, age, years, nextage } = data;
    return repository.update({ id, name, age, years, nextage });
  }

  async modify(id: any, data: any): Promise<Result | undefined> {
    const dbData = await this.getOne(id);
    if (dbData !== undefined) {
      Object.entries(dbData).forEach(([prop, val]) => {
        (dbData as any)[prop] = data[prop] ?? val;
      });
      return await this.replace(id, dbData);
    }
  }
}

```

El método de implementación enumera las propiedades definidas por la interfaz Result y verifica si los datos recibidos de la solicitud contienen un valor de reemplazo. Los nuevos valores se aplican para actualizar los datos existentes, que luego se pasan al método replace del repositorio para almacenarlos. Observa que el repositorio se utiliza de la misma manera para reemplazos y actualizaciones y que es tarea del servicio web preparar los datos para el almacenamiento. El listado 26 agrega una operación al cliente de línea de comandos que envía solicitudes PATCH.

Listado 26: Envío de solicitudes PATCH en el archivo operations.mjs en la carpeta src/cmdline.

```

...
export const ops = {

```

//...propiedades/funciones omitidas por brevedad...

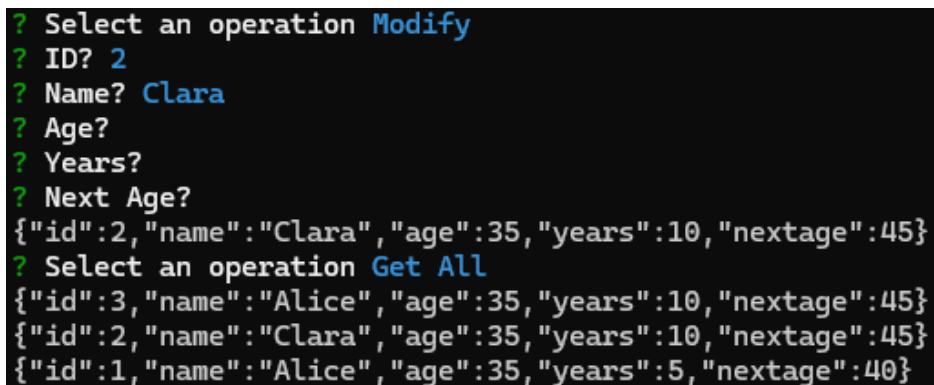
```
"Modify": async () => {
  const id = await input({ message: "ID?" });
  const values = {
    name: await input({ message: "Name?" }),
    age: await input({ message: "Age?" }),
    years: await input({ message: "Years?" }),
    nextage: await input({ message: "Next Age?" })
  };
  await sendRequest("PATCH", `/api/results/${id}`,
    Object.fromEntries(Object.entries(values)
      .filter(([p, v]) => v !== "")));
},

"Exit": () => process.exit()
}
...
```

Esta operación solicita valores de la misma manera que las solicitudes PUT, pero las funciones `Object.fromEntries`, `Object.entries` y `filter` de JavaScript se utilizan para excluir cualquier propiedad para la que no se proporciona ningún valor, de modo que se envíe una actualización parcial al servicio web.

Selecciona la nueva opción Modificar (Modify) desde la línea de comandos e ingresa 2 para el ID y Clara para el nombre, y luego presiona Retorno (Return) para las otras solicitudes.

Esta operación actualizará el resultado cuyo ID es 2 con un nuevo nombre, de esta manera:



```
? Select an operation Modify
? ID? 2
? Name? Clara
? Age?
? Years?
? Next Age?
{"id":2,"name":"Clara","age":35,"years":10,"nextage":45}
? Select an operation Get All
{"id":3,"name":"Alice","age":35,"years":10,"nextage":45}
{"id":2,"name":"Clara","age":35,"years":10,"nextage":45}
{"id":1,"name":"Alice","age":35,"years":5,"nextage":40}
```

El cliente solo envió un nuevo valor de nombre al servicio web y no necesitó enviar valores para las propiedades cuyos valores no se cambiaron.

Uso de JSON Patch

El enfoque utilizado en la sección anterior es útil cuando las únicas actualizaciones enviadas por el cliente son cambios en los valores de datos existentes. Muchos proyectos entran en esta categoría y es una técnica útil cuando los datos se vuelven demasiado complejos para las solicitudes de reemplazo y los únicos cambios son proporcionar valores actualizados.

El formato JSON Patch (<https://jsonpatch.com>) se puede utilizar para actualizaciones más complejas. Un documento JSON Patch contiene una serie de operaciones que se aplican a un documento JSON. Un documento JSON Patch para actualizar el valor de la propiedad name, por ejemplo, se vería así:

```
...
[{"op": "replace", "path": "/name", "value": "Bob" }]
...
```

Los documentos JSON Patch contienen un arreglo de objetos JSON, con propiedades op y path que describen la operación que se realizará y el objetivo de esa operación. Se requieren propiedades adicionales para algunas operaciones, como la propiedad value que se utiliza para especificar el nuevo valor para la operación de reemplazo. La tabla 5 describe las operaciones JSON Patch.

Tabla 5: Las operaciones JSON Patch.

| Operación | Descripción |
|-----------|---|
| add | Esta operación agrega una propiedad al documento JSON, con el nombre y el valor especificados por las propiedades path y value. |
| remove | Esta operación elimina una propiedad del documento JSON, especificada por la propiedad path. |
| replace | Esta operación cambia la propiedad especificada por la propiedad path utilizando el valor asignado a la propiedad value. |
| copy | Esta operación duplica una propiedad especificada por la propiedad from a la ubicación especificada por la propiedad path. |
| move | Esta operación mueve una propiedad especificada por la propiedad from a la ubicación especificada por la propiedad path. |
| test | Esta propiedad verifica que el documento JSON contenga una propiedad y un valor especificados por las propiedades path y value. No se realizará ninguna otra operación si esta operación falla. |

La propiedad path se utiliza para identificar valores en el documento JSON utilizando la sintaxis de apuntador JSON, que se describe en <https://datatracker.ietf.org/doc/html/rfc6901>, y que se puede utilizar para seleccionar propiedades y elementos del arreglo. La ubicación

/name, por ejemplo, denota una propiedad de nombre en el nivel superior del documento JSON.

Los documentos JSON Patch se pueden analizar y aplicar mediante código personalizado, pero es más fácil utilizar uno de los paquetes de JavaScript de código abierto disponibles. Ejecuta el comando que se muestra en el listado 27 en la carpeta part2app para instalar el paquete fast-json-patch (<https://github.com/StarcounterJack/JSON-Patch>), que es un paquete JSON Patch popular.

Listado 27: Instalación del paquete JSON Patch.

```
npm install fast-json-patch@3.1.1
```

El listado 28 actualiza el servicio web para que el método modify trate los datos que recibe como un documento JSON Patch y los aplique mediante el paquete fast-json-patch.

Listado 28: Uso de JSON Patch en el archivo results_api.ts en la carpeta src/server/api.

```
import { WebService } from "../http_adapter";
import { Result } from "../data/repository";
import repository from "../data";
import * as jsonpatch from "fast-json-patch";

export class ResultWebService implements WebService<Result> {

  getOne(id: any): Promise<Result | undefined> {
    return repository.getResultById(Number.parseInt(id));
  }

  getMany(query: any): Promise<Result[]> {
    if (query.name) {
      return repository.getResultsByName(query.name, 10);
    } else {
      return repository.getAllResults(10);
    }
  }

  async store(data: any): Promise<Result | undefined> {
    const { name, age, years } = data;
    const nextage = age + years;
    const id = await repository.saveResult({ id: 0, name, age, years, nextage });
    return await repository.getResultById(id);
  }

  delete(id: any): Promise<boolean> {
```

```

        return repository.delete(Number.parseInt(id));
    }

    replace(id: any, data: any): Promise<Result | undefined> {
        const { name, age, years, nextage } = data;
        return repository.update({ id, name, age, years, nextage });
    }

    async modify(id: any, data: any): Promise<Result | undefined> {
        const dbData = await this.getOne(id);
        if (dbData !== undefined) {
            return await this.replace(id,
                jsonpatch.applyPatch(dbData, data).newDocument);
        }
    }
}

```

El método `applyPatch` se utiliza para procesar el documento JSON Patch en un objeto. El objeto `result` define una propiedad `newDocument` que devuelve el objeto modificado, que se puede almacenar en la base de datos.

El encabezado HTTP Content-Type se establece en `application/json-patch+json` cuando se envía un documento JSON Patch y este tipo no se decodifica automáticamente mediante el componente de middleware JSON Express. El listado 29 configura el middleware JSON para que se decodifiquen las cargas útiles JSON normales y las cargas útiles JSON Patch.

Listado 29: Habilitación de la decodificación JSON Patch en el archivo `server.ts` en la carpeta `src/server`.

```

...

expressApp.use(helmet());
expressApp.use(express.json({
    type: ["application/json", "application/json-patch+json"]
}));

registerFormMiddleware(expressApp);
registerFormRoutes(expressApp);
...

```

El middleware JSON acepta un objeto de configuración cuya propiedad `type` se puede configurar con un arreglo de tipos de contenido para decodificar. El paso final es crear un documento JSON Patch en el cliente de línea de comandos, como se muestra en el listado 30.

Listado 30: Uso de JSON Patch en el archivo operations.mjs en la carpeta src/cmdline.

```
...
"Modify": async () => {
  const id = await input({ message: "ID?" });
  const values = {
    name: await input({ message: "Name?" }),
    age: await input({ message: "Age?" }),
    years: await input({ message: "Years?" }),
    nextage: await input({ message: "Next Age?" })
  };
  await sendRequest("PATCH", `/api/results/${id}`,
    Object.entries(values).filter(([p, v]) => v !== "")
      .map(([p, v]) => ({ op: "replace", path: "/" + p, value: v })),
    "application/json-patch+json");
},

"Exit": () => process.exit()
}
...
```

El paquete fast-json-patch es capaz de generar un documento JSON Patch, pero es más fácil crear parches con código personalizado que aplicarlos, y la declaración modified en el listado 30 crea operaciones de reemplazo para cada uno de los valores ingresados por el usuario.

No hay cambios en la forma en que se comportan el cliente y el servicio web, lo cual puedes confirmar seleccionando la opción Modificar desde la línea de comandos e ingresando 2 para el ID y Clara para el nombre, y luego presionando Enter para las otras indicaciones. Este es el mismo cambio realizado anteriormente en el documento y debería producir los mismos resultados, como este:

```
? Select an operation Modify
? ID? 2
? Name? Clara
? Age?
? Years?
? Next Age?
{"id":2,"name":"Clara","age":35,"years":10,"nextage":45}
? Select an operation Get All
{"id":3,"name":"Alice","age":35,"years":10,"nextage":45}
{"id":2,"name":"Clara","age":35,"years":10,"nextage":45}
{"id":1,"name":"Alice","age":35,"years":5,"nextage":40}
```

Validación de datos del cliente

Los servicios web no pueden confiar en los datos que se reciben de los clientes y están sujetos a los mismos tipos de problemas que afectan a los formularios HTML. Los usuarios malintencionados pueden crear solicitudes HTTP o alterar el código JavaScript del lado del cliente para enviar valores de datos que causarán errores o crearán resultados inesperados, similares a los problemas con los datos del formulario descritos en el documento del manejo de datos de formularios.

La dificultad con los servicios web es validar los datos de una manera que no socave la claridad del código que se obtuvo al aislar las declaraciones que manejan las solicitudes HTTP. Si cada método de servicio web valida sus datos directamente, el resultado es un lío de declaraciones de código duplicadas que ocultan la funcionalidad del servicio web y son difíciles de leer y entretener. El mejor enfoque para la validación es describir los requisitos de validación y aplicarlos fuera del servicio web.

Creación de la infraestructura de validación

Permitir que los requisitos de validación de un servicio web se expresen de manera clara y concisa requiere una infraestructura que oculte los detalles de implementación desordenados.

El punto de partida es definir los tipos que describen la validación de un servicio web completo, un método de servicio web y una única regla de validación. Agrega un archivo llamado `validation_types.ts` a la carpeta `src/server/api` con el contenido que se muestra en el listado 31.

Listado 31: El contenido del archivo `validation_types.ts` en la carpeta `src/server/api`.

```
export interface WebServiceValidation {
  keyValidator?: ValidationRule;
  getMany?: ValidationRequirements;
  store?: ValidationRequirements;
  replace?: ValidationRequirements;
  modify?: ValidationRequirements;
}

export type ValidationRequirements = {
  [key: string]: ValidationRule
}

export type ValidationRule =
  ((value: any) => boolean)[] |
  {
```

```

    required?: boolean,
    validation: ((value: any) => boolean)[],
    converter?: (value: any) => any,
  }

export class ValidationError implements Error {
  constructor(public name: string, public message: string) {}
  stack?: string | undefined;
  cause?: unknown;
}

```

El tipo `WebServiceValidation` describe los requisitos de validación para un servicio web. La propiedad `keyValidator` especifica los requisitos de validación para los valores de ID que identifican los registros de datos, utilizando el tipo `ValidationRule`. Una `ValidationRule` puede ser un arreglo de funciones de prueba que se aplicarán a un valor o un objeto que además especifica si se requiere un valor y un convertidor que transformará el valor en el tipo esperado por el método de servicio web.

Las otras propiedades definidas por el tipo `WebServiceValidation` corresponden a los métodos de servicio web que consumen datos. A estas propiedades se les puede asignar un objeto `ValidationRequirements`, que puede especificar la forma del objeto que espera el servicio web, y una `ValidationRule` para cada una de ellas. La clase `ValidationError` representa un problema al validar los datos enviados por el cliente en una solicitud.

El siguiente paso es definir funciones que aplicarán los requisitos descritos utilizando los tipos del listado 31 para validar los datos. Agrega un archivo llamado `validation_functions.ts` a la carpeta `src/server/api` con el código que se muestra en el listado 32.

Listado 32: El contenido del archivo `validation_functions.ts` en la carpeta `src/server/api`.

```

import { ValidationError, ValidationRequirements,
  ValidationRule, WebServiceValidation } from "../validation_types";

export type ValidationResult = [valid: boolean, value: any];

export function validate(data: any, reqs: ValidationRequirements): any {
  let validatedData: any = {};
  Object.entries(reqs).forEach(([prop, rule]) => {
    const [valid, value] = applyRule(data[prop], rule);
    if (valid) {
      validatedData[prop] = value;
    } else {
      throw new ValidationError(prop, "Validation Error");
    }
  })
}

```

```

    });
    return validatedData;
  }

  function applyRule(val: any,
    rule: ValidationRule): ValidationResult {
    const required = Array.isArray(rule) ? true : rule.required;
    const checks = Array.isArray(rule) ? rule : rule.validation;
    const convert = Array.isArray(rule) ? (v: any) => v : rule.converter;
    if (val === null || val == undefined || val === "") {
      return [required ? false : true, val];
    }
    let valid = true;
    checks.forEach(check => {
      if (!check(val)) {
        valid = false;
      }
    });
    return [valid, convert ? convert(val) : val];
  }

  export function validateIdProperty<T>(val: any,
    v: WebServiceValidation) : any {
    if (v.keyValidator) {
      const [valid, value] = applyRule(val, v.keyValidator);
      if (valid) {
        return value;
      }
      throw new ValidationError("ID", "Validation Error");
    }
    return val;
  }

```

La función de validación acepta un objeto de datos y un objeto ValidationRequirements. Cada propiedad especificada por el objeto ValidationRequirements se lee desde el objeto de datos y se valida. El resultado es un objeto que contiene datos validados en los que el servicio web puede confiar.

Se genera ValidationError si una propiedad de datos no cumple con sus requisitos de validación.

Para integrar el proceso de validación de la forma más fluida posible, vamos a insertar una capa de validación entre el adaptador HTTP y el servicio web. La capa de validación recibirá la solicitud y la respuesta del adaptador, validará los datos y los pasará al servicio web.

Agrega un archivo llamado `validation_adapter.ts` a la carpeta `src/server/api` con el contenido que se muestra en el listado 33.

Listado 33: El archivo `validation_adapter.ts` en la carpeta `src/server/api`.

```
import { WebService } from "../http_adapter";
import { validate, validateIdProperty } from "../validation_functions";
import { WebServiceValidation } from "../validation_types";

export class Validator<T> implements WebService<T> {

  constructor(private ws: WebService<T>,
    private validation: WebServiceValidation) {}

  getOne(id: any): Promise<T | undefined> {
    return this.ws.getOne(this.validateId(id));
  }

  getMany(query: any): Promise<T[]> {
    if (this.validation.getMany) {
      query = validate(query, this.validation.getMany);
    }
    return this.ws.getMany(query);
  }

  store(data: any): Promise<T | undefined> {
    if (this.validation.store) {
      data = validate(data, this.validation.store);
    }
    return this.ws.store(data);
  }

  delete(id: any): Promise<boolean> {
    return this.ws.delete(this.validateId(id));
  }

  replace(id: any, data: any): Promise<T | undefined> {
    if (this.validation.replace) {
      data = validate(data, this.validation.replace);
    }
    return this.ws.replace(this.validateId(id), data);
  }

  modify(id: any, data: any): Promise<T | undefined> {
    if (this.validation.modify) {
```

```

        data = validate(data, this.validation.modify);
    }
    return this.ws.modify(this.validateId(id), data);
}

validateId(val: any) {
    return validateIdProperty(val, this.validation);
}
}

```

Los valores de ID incluidos en las URL se validan mediante la función `validationIdProperty` y cualquier dato adicional se valida mediante la función `validation`. Si la validación falla, se lanzará `ValidationError`. El listado 34 actualiza el adaptador HTTP para capturar las excepciones lanzadas cuando se maneja una solicitud y genera una respuesta 400 Bad Request para errores de validación y una respuesta 500 Internal Server Error para cualquier otro problema.

Listado 34: Manejo de errores de validación en el archivo `http_adapter.ts` en la carpeta `src/server/api`.

```

import { Express, Response } from "express";
import { ValidationError } from "../validation_types";

export interface WebService<T> {
    getOne(id: any) : Promise<T | undefined>;
    getMany(query: any) : Promise<T[]>;
    store(data: any) : Promise<T | undefined>;
    delete(id: any): Promise<boolean>;
    replace(id: any, data: any): Promise<T | undefined>;
    modify(id: any, data: any): Promise<T | undefined>;
}

export function createAdapter<T>(app: Express, ws: WebService<T>, baseUrl: string) {

    //...rutas omitidas por brevedad..

    const writeErrorResponse = (err: any, resp: Response) => {
        console.error(err);
        resp.writeHead(err instanceof ValidationError ? 400 : 500);
        resp.end();
    }
}

```


Observa que no se incluyen detalles del problema de validación en el resultado enviado al cliente. Sería posible enviar al cliente un objeto JSON que describa los problemas de validación, pero en términos prácticos, los clientes rara vez pueden hacer un uso sensato de dicha información. Los requisitos de validación se encuentran durante el proceso de desarrollo y es mejor incluirlos en la documentación del desarrollador para que el cliente pueda validar los datos recibidos del usuario antes de enviarlos al servicio web.

Confiar en el servicio web para proporcionar errores de validación que se puedan presentar al usuario es un proceso problemático y que se debe evitar, incluso cuando el cliente y el servicio web están escritos por el mismo equipo. Cuando publiques un servicio web, debes esperar brindar soporte a los desarrolladores del lado del cliente a medida que consumen la funcionalidad que tu proporcionas.

Definición de la validación para la API de resultados

La complejidad de la validación está en la infraestructura, que permite definir de manera concisa los requisitos de validación para un servicio web. Agrega un archivo llamado `results_api_validation.ts` en la carpeta `src/server/api` con el contenido que se muestra en el listado 35.

Listado 35: El contenido del archivo `results_api_validation.ts` en la carpeta `src/server/api`.

```
import { ValidationRequirements, ValidationRule,
  WebServiceValidation } from "../validation_types";
import validator from "validator";

const intValidator : ValidationRule = {
  validation: [val => validator.isInt(val.toString())],
  converter: (val) => Number.parseInt(val)
}

const partialResultValidator: ValidationRequirements = {
  name: [(val) => !validator.isEmpty(val)],
  age: intValidator,
  years: intValidator
}

export const ResultWebServiceValidation: WebServiceValidation = {

  keyValidator: intValidator,

  store: partialResultValidator,
```

```

    replace: {
      ...partialResultValidator,
      nextage: intValidator
    }
  }
}

```

El objeto `ResultWebServiceValidation` define las propiedades `keyValidator`, `store` y `replace`, lo que indica que el servicio web requiere que se validen sus valores de ID, así como los datos utilizados por los métodos `store` y `replace`.

La `ValidationRule` denominada `intValidator` describe la validación de valores enteros, con una propiedad de validación que utiliza el paquete `validator` para garantizar que un valor sea un entero y una función de conversión que analiza el valor como un número.

El `intValidator` se utiliza por sí solo como el validador de llaves y en el objeto `ValidationRequirements` denominado `partialResultValidator`, que valida las propiedades `name`, `age` y `years` que requiere el método `store`. Los requisitos de validación para el método `replace` amplían los utilizados por el método `store` agregando una propiedad `nextage`.

Este enfoque de validación permite que los requisitos de datos de un servicio web se expresen por separado de la aplicación de esos requisitos. El listado 36 envuelve (wraps) el servicio web en su validador.

Listado 36: Aplicación de la validación en el archivo `index.ts` en la carpeta `src/server/api`.

```

import { Express } from "express";
import { createAdapter } from "../http_adapter";
import { ResultWebService } from "../results_api";
import { Validator } from "../validation_adapters";
import { ResultWebServiceValidation } from "../results_api_validation";

export const createApi = (app: Express) => {
  createAdapter(app, new Validator(new ResultWebService(),
    ResultWebServiceValidation), "/api/results");
}

```

El cambio final es pequeño, ya que aprovecha la conversión de tipos realizada por el sistema de validación, como se muestra en el listado 37.

Listado 37: Confianza en la conversión de tipos en el archivo `results_api.ts` en la carpeta `src/server/api`.

```

...
async store(data: any): Promise<Result | undefined> {

```

```

    const { name, age, years } = data;
    //const nextage = Number.parseInt(age) + Number.parseInt(years);
    const nextage = age + years;
    const id = await repository.saveResult({ id: 0, name, age, years, nextage});
    return await repository.getResultById(id);
  }
  ...

```

El método de almacenamiento no se llamará a menos que los botones de edad y años se hayan convertido a valores numéricos, lo que significa que el método de almacenamiento no necesita realizar sus propias conversiones.

Para probar la validación, seleccione la opción Obtener ID del cliente de línea de comandos e ingrese ABC cuando se te solicite.

La verificación de validación rechazará este valor y generará una respuesta 400 Solicitud incorrecta, como esta:

```

? Select an operation Get ID
? ID? ABC
400 Bad Request

```

Selecciona la opción Almacenar (Store) e ingresa Joe, 30 y Ten cuando se te solicite. El último valor no supera la validación y genera otra respuesta 400.

```

? Select an operation Store
? Name? Joe
? Age? 30
? Years? Ten
400 Bad Request

```

Realización de la validación del modelo

No todas las validaciones se pueden realizar antes de que la solicitud se pase al método del servicio web que generará una respuesta. Un ejemplo son las solicitudes PATCH, donde el cliente puede enviar actualizaciones parciales que pueden generar datos inconsistentes en la base de datos, como proporcionar un valor de año nuevo sin un valor de edad siguiente correspondiente, de modo que el resultado del cálculo no tenga sentido.

La validación de este tipo de actualización no se puede realizar hasta que la actualización se haya aplicado a los datos almacenados existentes, lo que significa que debe realizarse mediante el método de servicio web, que es el primer punto en el proceso de actualización

donde los cambios y los datos almacenados están disponibles. Esto a menudo se conoce como validación del modelo, aunque no existe una terminología uniforme.

El listado 38 define un nuevo tipo de datos que combina la validación existente con una nueva regla que se aplica a todo el objeto del modelo de datos.

Listado 38: Adición de un tipo en el archivo `validation_types.ts` en la carpeta `src/server/api`.

```
export interface WebServiceValidation {
  keyValidator?: ValidationRule;
  getMany?: ValidationRequirements;
  store?: ValidationRequirements;
  replace?: ValidationRequirements;
  modify?: ValidationRequirements;
}

export type ValidationRequirements = {
  [key: string]: ValidationRule
}

export type ValidationRule =
  ((value: any) => boolean)[] |
  {
    required?: boolean,
    validation: ((value: any) => boolean)[],
    converter?: (value: any) => any,
  }

export class ValidationError implements Error {
  constructor(public name: string, public message: string) {}
  stack?: string | undefined;
  cause?: unknown;
}

export type ModelValidation = {
  modelRule?: ValidationRule,
  propertyRules?: ValidationRequirements
}
```

El listado 39 utiliza el tipo `ModelValidation` en una nueva función que se puede utilizar para validar un objeto antes de almacenarlo.

Listado 39: Agregar una función en el archivo `validation_functions.ts` en la carpeta `src/server/api`.

```

import { ModelValidation, ValidationError, ValidationRequirements,
  ValidationRule, WebServiceValidation } from "../validation_types";

export type ValidationResult = [valid: boolean, value: any];

//...funciones omitidas por brevedad...

export function validateModel(model: any, rules: ModelValidation) : any {
  if (rules.propertyRules) {
    model = validate(model, rules.propertyRules);
  }
  if (rules.modelRule) {
    const [valid, data] = applyRule(model, rules.modelRule);
    if (valid) {
      return data;
    }
    throw new ValidationError("Model", "Validation Error");
  }
}

```

La función `validationModel` aplica las reglas para cada propiedad y luego aplica la regla de todo el modelo. Las reglas de propiedad pueden realizar conversiones de tipo y, por lo tanto, el resultado de las comprobaciones de propiedad se utiliza como entrada para la validación de todo el modelo. El listado 40 define la validación requerida para un objeto `Result`.

Listado 40: Definición de un validador de modelo en el archivo `results_api_validation.ts` en la carpeta `src/server/api`.

```

import { ModelValidation, ValidationRequirements, ValidationRule,
  WebServiceValidation } from "../validation_types";
import validator from "validator";

const intValidator : ValidationRule = {
  validation: [val => validator.isInt(val.toString())],
  converter: (val) => Number.parseInt(val)
}

const partialResultValidator: ValidationRequirements = {
  name: [(val) => !validator.isEmpty(val)],
  age: intValidator,
  years: intValidator
}

export const ResultWebServiceValidation: WebServiceValidation = {

```

```

    keyValidator: intValidator,

    store: partialResultValidator,

    replace: {
      ...partialResultValidator,
      nextage: intValidator
    }
  }

  export const ResultModelValidation : ModelValidation = {
    propertyRules: { ...partialResultValidator, nextage: intValidator },
    modelRule: [(m: any) => m.nextage === m.age + m.years]
  }

```

La propiedad `propertyRules` utiliza las reglas de validación creadas para los ejemplos anteriores. La propiedad `modelRule` comprueba que el valor `nextage` sea la suma de las propiedades `age` y `years`.

Se requiere un pequeño cambio en la regla utilizada para validar números enteros. El método `isInt` proporcionado por el paquete de validación solo funciona con valores de cadena, pero una actualización parcial puede combinar los valores de cadena recibidos de la solicitud HTTP con valores numéricos leídos de la base de datos. Para evitar excepciones, el valor que se está comprobando siempre se convierte en una cadena.

El listado 41 actualiza el servicio web para utilizar la función de validación de modelos para los métodos `replace` y `modify`, lo que garantiza que no se escriban datos inconsistentes en la base de datos.

Listado 41: Validación de datos en el archivo `results_api.ts` en la carpeta `src/server/api`.

```

import { WebService } from "../http_adapter";
import { Result } from "../data/repository";
import repository from "../data";
import * as jsonpatch from "fast-json-patch";
import { validateModel } from "../validation_functions";
import { ResultModelValidation } from "../results_api_validation";

export class ResultWebService implements WebService<Result> {

  getOne(id: any): Promise<Result | undefined> {
    return repository.getResultById(id);
  }
}

```

```

getMany(query: any): Promise<Result[]> {
  if (query.name) {
    return repository.getResultsByName(query.name, 10);
  } else {
    return repository.getAllResults(10);
  }
}

async store(data: any): Promise<Result | undefined> {
  const { name, age, years } = data;
  const nextage = age + years;
  const id = await repository.saveResult({ id: 0, name, age, years, nextage });
  return await repository.getResultById(id);
}

delete(id: any): Promise<boolean> {
  return repository.delete(Number.parseInt(id));
}

replace(id: any, data: any): Promise<Result | undefined> {
  const { name, age, years, nextage } = data;
  const validated = validateModel({ name, age, years, nextage },
    ResultModelValidation)
  return repository.update({ id, ...validated });
}

async modify(id: any, data: any): Promise<Result | undefined> {
  const dbData = await this.getOne(id);
  if (dbData !== undefined) {
    return await this.replace(id,
      jsonpatch.applyPatch(dbData, data).newDocument());
  }
}
}

```

La validación se puede realizar en el método replace, que permite validar los reemplazos y las actualizaciones de manera consistente.

Selecciona la opción Replace del cliente de línea de comandos e ingresa 1, Joe, 20, 10 y 25 cuando se te solicite.

Estos datos no son válidos porque el valor de nextage debería ser 30, por lo que el proceso de validación falla y se produce una respuesta 400 Bad Request, como esta:

```
? Select an operation Replace
? ID? 1
? Name? Joe
? Age? 20
? Years? 10
? Next Age? 25
400 Bad Request
```

La combinación de validación de solicitud y modelo garantiza que el servicio web solo reciba y almacene datos válidos, mientras que las características HTTP y de validación abstractas ayudan a simplificar la implementación del servicio web para que sea más fácil de entender y mantener.

Uso de un paquete para servicios web

Existen excelentes paquetes disponibles para crear servicios web, aunque la falta de estandarización significa que debes encontrar uno que se adapte a tus preferencias sobre cómo deben funcionar los servicios web, que pueden ser diferentes del enfoque que hemos adoptado en este documento. Utilizaremos el paquete Feathers (<https://feathersjs.com>), que funciona de manera similar al código personalizado de este documento y tiene buenas integraciones con bases de datos populares y otros paquetes, incluido Express.

Pero hay muchos buenos paquetes disponibles, y un buen consejo es buscar microservicios, que se ha convertido en un término tan popular que algunos paquetes se posicionan como parte del ecosistema de microservicios. Ejecuta los comandos que se muestran en el listado 42 en la carpeta part2app para instalar el paquete Feathers y las integraciones con Express.

Listado 42: Instalación de paquetes.

```
npm install @feathersjs/feathers@5.0.14
npm install @feathersjs/express@5.0.14
```

Los paquetes Feathers contienen declaraciones de tipo TypeScript, pero anulan las declaraciones del paquete Express. Se requiere un cambio en la configuración del compilador para solucionar este problema, como se muestra en el listado 43.

Listado 43: Cambio de la configuración del compilador en el archivo tsconfig.json en la carpeta part2app.

```
{
  "extends": "@tsconfig/node20/tsconfig.json",
  "compilerOptions": {
    "rootDir": "src/server",
    "outDir": "dist/server/",
```



```

    "noImplicitAny": false
  },
  "include": ["src/server/**/*"]
}

```

La integración de Feathers con Express funciona ampliando la API existente, y las declaraciones de tipo que proporciona el paquete son diferentes de las proporcionadas por el paquete @types/express.

Creación de un adaptador para servicios web

El paquete Feathers describe los servicios web mediante una serie de métodos, similares a la interfaz utilizada por el código personalizado anterior en el documento. Existen algunas pequeñas diferencias, pero los dos enfoques son lo suficientemente similares como para que un adaptador simple permita que el código de manejo de HTTP personalizado se reemplace con el paquete Feathers, sin necesidad de realizar cambios en el servicio web.

Agrega un archivo llamado feathers_adapter.ts a la carpeta src/server/api con el contenido que se muestra en el listado 44.

Listado 44: El contenido del archivo feathers_adapter.ts en la carpeta src/server/api.

```

import { Id, NullableId, Params } from "@feathersjs/feathers";
import { WebService } from "../http_adapter";

```

```

export class FeathersWrapper<T> {

  constructor(private ws: WebService<T>) {}

  get(id: Id) {
    return this.ws.getOne(id);
  }

  find(params: Params) {
    return this.ws.getMany(params.query);
  }

  create(data: any, params: Params) {
    return this.ws.store(data);
  }

  remove(id: NullableId, params: Params) {
    return this.ws.delete(id);
  }
}

```

```

    update(id: NullableId, data: any, params: Params) {
      return this.ws.replace(id, data);
    }

    patch(id: NullableId, data: any, params: Params) {
      return this.ws.modify(id, data);
    }
  }
}

```

La API Feathers proporciona tipos que representan valores de ID, cuerpos de solicitud y parámetros de consulta, pero solo hay un número limitado de formas en las que se puede representar una solicitud HTTP, por lo que es un proceso simple para unir el paquete Feathers y el código personalizado. Los ejemplos posteriores utilizarán la API Feathers directamente, pero este enfoque demuestra lo fácil que es adaptar el código existente para que funcione con paquetes de terceros.

La integración de Feathers con Express supone que Feathers extenderá la API Express para agregar funciones. El listado 45 utiliza la funcionalidad de Feathers para crear un servicio web sin cambiar el resto de la aplicación.

Listado 45: Uso de Feathers en el archivo index.ts en la carpeta src/server/api.

```

import { Express } from "express";
import { createAdapter } from "../http_adapter";
import { ResultWebService } from "../results_api";
import { Validator } from "../validation_adapters";
import { ResultWebServiceValidation } from "../results_api_validation";
import { FeathersWrapper } from "../feathers_adapters";
import { feathers } from "@feathersjs/feathers";
import feathersExpress, { rest } from "@feathersjs/express";
import { ValidationError } from "../validation_types";

export const createApi = (app: Express) => {
  //createAdapter(app, new Validator(new ResultWebService(),
  //  ResultWebServiceValidation), "/api/results");
  const feathersApp = feathersExpress(feathers(), app).configure(rest());

  const service = new Validator(new ResultWebService(),
    ResultWebServiceValidation);

  feathersApp.use('/api/results', new FeathersWrapper(service));
  feathersApp.hooks({
    error: {

```

```

    all: [(ctx) => {
      if (ctx.error instanceof ValidationError) {
        ctx.http = { status: 400};
        ctx.error = undefined;
      }
    }]
  }
});
}

```

La versión mejorada de Express se crea con esta declaración:

```

...
const feathersApp = feathersExpress(feathers(), app).configure(rest());
...

```

Este encantamiento habilita Feathers y lo configura para admitir consultas RESTful. Feathers se puede utilizar de diferentes maneras, y las solicitudes RESTful son solo una de las formas en las que los clientes pueden comunicarse con los componentes del lado del servidor de Feathers.

Feathers admite ganchos (*hooks*), que permiten ejecutar funciones en momentos clave del ciclo de vida de la solicitud. Los ganchos son una característica útil y se pueden utilizar para tareas que incluyen la validación y el manejo de errores.

En este ejemplo, la validación se maneja mediante el código personalizado, pero esta declaración define un gancho que se invocará cuando se genere una excepción al manejar una solicitud:

```

...
feathersApp.hooks({
  error: {
    all: [(ctx) => {
      if (ctx.error instanceof ValidationError) {
        ctx.http = { status: 400};
        ctx.error = undefined;
      }
    }]
  }
});
...

```

El código personalizado genera `ValidationError` cuando falla la validación, que Feathers maneja enviando una respuesta 500. Hooks recibe un objeto de contexto que proporciona detalles de la solicitud y su resultado, y esta declaración cambia el código de estado de respuesta si se produjo `ValidationError`. No hay cambios en la forma en que funciona el

servicio web porque utiliza el mismo código personalizado para manejar solicitudes. Pero, después de haber visto cómo funcionan los servicios web RESTful y cómo se pueden crear, pasar a un paquete como Feathers permite utilizar las mismas funciones sin la necesidad de código personalizado.

Resumen

En este documento/práctica, se demostró cómo se pueden utilizar las funciones HTTP proporcionadas por Node.js y mejoradas por el paquete Express para crear un servicio web RESTful.

- La URL de la solicitud HTTP identifica los datos y el método HTTP indica la operación que se realizará.
- La mayoría de los servicios web utilizan el formato JSON, que ha reemplazado a XML como formato de datos predeterminado.
- Existe poca estandarización en la forma en que se implementan los servicios web, aunque existen algunas convenciones comunes que se utilizan ampliamente, en particular en relación con las operaciones que representan los métodos HTTP.
- Los datos recibidos por los servicios web deben validarse antes de que se puedan utilizar de forma segura.
- Los servicios web se escriben más fácilmente separando la implementación del código que maneja las solicitudes HTTP y realiza la validación.

En el próximo documento, se mostrará cómo se pueden autenticar las solicitudes HTTP y cómo se puede utilizar la identidad del usuario para la autorización.