Table of contents

Introduction

Express.js + TypeScript Documentation

a. What is Express.js?

Express.js is a lightweight, unopinionated Node.js framework for building server-side applications. It abstracts away the complexity of setting up a server with HTTP, streamlining the process of handling routing, middleware, and HTTP requests.

b. Why Express.js and not Node.js?

Node.js is a runtime that allows JavaScript to run on the server, but lacks built-in features for routing, request handling, and middleware. Express.js builds on Node.js, providing:

- **Simplicity**: Easy to use with minimal setup.
- **Middleware support**: Enable adding custom logic to requests at various stages.
- **Routing**: Simplified handling of HTTP requests like GET, POST, PUT, DELETE.

c. What Can Express.js Do?

- Serve **static files** like HTML, CSS, and images.
- Build **RESTful APIs**.
- Handle **middleware** for tasks like authentication, logging, and error handling.
- Integrate with **templating engines** (e.g., EJS, Pug) for dynamic web pages.
- Support real-time applications with WebSockets or libraries like **Socket.io**.

Getting Started

a. Create an Express App

1. **Initialize a Node project:**

```
npm init -y
```

2. **Install Express and TypeScript:**

```
npm install express
npm install --save-dev typescript @types/express tsx
```

3. **Set up TypeScript:** Create a `tsconfig.json`: and add

```
{
   "compilerOptions": {
      "target": "ESNext",   // Target ES6 or later to support modern
JavaScript features
      "module": "Node16",    // Use node modules version 16
      "rootDir": "./src",       // Source files location
      "outDir": "./dist",          // Output directory for compiled
files
      "esModuleInterop": true,        // Enables support for `import` in
CommonJS
      "strict": true,                    // Enables strict type-
checking options
      "skipLibCheck": true,             // Skips type checking of
declaration files
      "moduleResolution": "Node16",        // For resolving node modules
      "resolveJsonModule": true,              // Enable importing
`.json` files
      "allowSyntheticDefaultImports": true,    // Allow default imports
from modules
      "forceConsistentCasingInFileNames": true   // Ensure consistent
```

```
casing in file names
   },
   "include": ["src/**/*.ts"],          // Include all TypeScript files
in the `src` folder
   "exclude": ["node_modules"]          // Exclude `node_modules`
}
```

4. **Create a basic Express server:** In `src/server.ts`:

```
import express, { Request, Response } from 'express';

const app = express();
const port = 8000;

app.use(express.json()); // Middleware to parse JSON requests

app.get('/', (req: Request, res: Response) => {
  res.send('Hello, Express with TypeScript!');
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

5. **Run the server:** Add this to `package.json` scripts:

```
"scripts": {
  "dev": "tsx watch src/app.ts",
   "start": "node dist/index.js"
}
```

6. Test the app, install the REST Client vs code extension. create `app.http` and add

```
### HTTP/1.1
```

```
GET http://localhost:8000/
```

Start the server :

```
npm start
```

b. Frameworks Built on Express

Several frameworks extend Express to provide additional features and structure:

- **NestJS**: A framework built with TypeScript, offering a modular architecture similar to Angular. It enhances Express with dependency injection, strong typing, and structured patterns for scalable applications.
- **Loopback**: Extends Express to build APIs rapidly, with built-in connectors for databases, auto-generated REST APIs, and strong TypeScript support.
- **Sails.js**: Another Express-based framework, which provides a MVC architecture and supports real-time features out of the box.

c. Proper Express Folder Structure for REST API

A well-structured folder organization is essential for scalability and maintainability. Here is an ideal folder structure:

```
src/
  ├── controllers/        # Handles the business logic
  ├── routes/             # Defines routes for each resource
  ├── models/             # Database models or data definitions
  ├── services/           # Handles data processing and calls to
external APIs
  ├── middleware/         # Custom middleware logic (auth, logging,
etc.)
  ├── config/             # Environment variables, configuration
settings
  ├── utils/              # Helper functions and utility classes
  ├── types/              # Custom TypeScript types/interfaces
```

```
  ├── app.ts              # Express application initialization
  └── server.ts           # Server startup file
```

Example Breakdown:

i. Model, Routes, Controllers

1. **Models**: Defines the structure of your database schema or data interfaces.

```ts
// src/models/User.ts
export interface User {
  id: string;
  name: string;
  email: string;
}
```

2. **Controllers**: Implements the core business logic, processing requests and sending appropriate responses.

```ts
// src/controllers/userController.ts
import { Request, Response } from 'express';

export const getUsers = (req: Request, res: Response): void => {
  const users = [{ id: '1', name: 'John Doe', email: 'john@example.com'
}];
  res.json(users);
};
```

3. **Routes**: Maps routes to corresponding controllers.

```ts
// src/routes/userRoutes.ts
import express from 'express';
import { getUsers } from '../controllers/userController';
```

```
const router = express.Router();

router.get('/users', getUsers);

export default router;
```

4. **Middleware**: Add custom logic between request and response.

```
// src/middleware/authMiddleware.ts
import { Request, Response, NextFunction } from 'express';

export const authMiddleware = (req: Request, res: Response, next:
NextFunction) => {
  const authToken = req.headers['authorization'];

  if (authToken) {
    // Do some token verification here
    next(); // Proceed to the next middleware/controller
  } else {
    res.status(401).json({ message: 'Unauthorized' });
  }
};
```

5. **Server and App Initialization:**

```
// src/app.ts
import express from 'express';
import userRoutes from './routes/userRoutes';

const app = express();
app.use(express.json());

app.use('/api', userRoutes);

export default app;
```

```ts
// src/server.ts
import app from './app';

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

ii. Model, Routes, Controllers, Services

1. **Services**: Contains logic for database operations, external API calls, or complex data manipulation. Services abstract the business logic and keep the controller lean.

```ts
// src/services/userService.ts
import { User } from '../models/User';

export const getAllUsers = async (): Promise<User[]> => {
  // Simulate fetching data from a database
  return [{ id: '1', name: 'John Doe', email: 'john@example.com' }];
};
```

2. **Controller using a Service**:

```ts
// src/controllers/userController.ts
import { Request, Response } from 'express';
import { getAllUsers } from '../services/userService';

export const getUsers = async (req: Request, res: Response) => {
  try {
    const users = await getAllUsers();
    res.status(200).json(users);
  } catch (error) {
    res.status(500).json({ message: 'Internal Server Error' });
```

```
  }
};
```

App Object (Express Object)

The `app` object in Express represents the core of your Express application. It is an instance of the `express()` function, and it provides methods to define routes, middleware, and configure various settings for your web application or API.

Basic Express Application Setup

Before diving into the specific methods, here's a simple example of how to initialize the `app` object.

```
import express from 'express';

const app = express();
const port = 3000;

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

The `app` object provides several HTTP methods that map to HTTP request verbs (`GET`, `POST`, `PUT`, `DELETE`, `PATCH`, etc.). Additionally, `app.use()` allows you to mount middleware and handle requests before they reach the route handler.

a. app.get()

`app.get()` handles HTTP **GET** requests to a specific route. This is typically used to retrieve or read data from the server.

```
app.get('/users', (req, res) => {
  res.json([{ id: 1, name: 'John Doe' }]);
});
```

Parameters:

- **Path (string)**: The URL or endpoint you want to handle (e.g., `/users`).

- **Callback Function (middleware)**: A function that receives the `Request` and `Response` objects. It contains logic to process the request and return the response.

Example with Route Parameters:

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.json({ id: userId, name: 'John Doe' });
});
```

Example with Query Parameters:

```
app.get('/search', (req, res) => {
  const query = req.query.q; // Example: /search?q=express
  res.json({ message: `Searching for ${query}` });
});
```

b. app.post()

`app.post()` handles HTTP **POST** requests. It is commonly used to send data to the server, typically for creating new resources (e.g., user registration, creating a blog post).

```
app.post('/users', (req, res) => {
  const { name, email } = req.body; // Assuming body-parser middleware
is used
  res.status(201).json({ id: 1, name, email });
});
```

Parameters:

- **Path (string)**: The URL or endpoint you want to handle (e.g., `/users`).
- **Callback Function**: Receives the `Request` and `Response` objects. The `req.body` contains data sent from the client (usually in JSON format).

Example: Posting Form Data

```
app.use(express.json()); // Parse incoming JSON data
app.use(express.urlencoded({ extended: true })); // Parse URL-encoded
data

app.post('/users', (req, res) => {
  const { username, password } = req.body;
  res.status(201).json({ message: 'User created', username });
});
```

c. app.put()

`app.put()` handles HTTP **PUT** requests. This is used to update a resource completely (replacing it with the new data). The client typically sends the full resource, and the server updates the record.

```
app.put('/users/:id', (req, res) => {
  const userId = req.params.id;
  const { name, email } = req.body;
  // Update the user in the database here
  res.json({ message: `User ${userId} updated`, name, email });
});
```

Parameters:

- **Path (string)**: The URL or endpoint, often with a dynamic parameter (e.g., `/users/:id`).
- **Callback Function**: `req.body` contains the updated data sent by the client.

Example: Full Update

```
app.put('/articles/:id', (req, res) => {
  const articleId = req.params.id;
  const { title, content } = req.body;
  // Replace the old article with the new one
```

```
  res.json({ message: `Article ${articleId} updated`, title, content });
});
```

d. app.delete()

`app.delete()` handles HTTP **DELETE** requests, used to delete a resource on the server.

```
app.delete('/users/:id', (req, res) => {
  const userId = req.params.id;
  // Remove the user from the database here
  res.status(204).json({ message: `User ${userId} deleted` });
});
```

Parameters:

- **Path (string)**: The URL or endpoint, often with a dynamic parameter (e.g., `/users/:id`).
- **Callback Function**: The request usually only contains the ID of the resource to delete.

Example: Delete Request with Confirmation

```
app.delete('/posts/:id', (req, res) => {
  const postId = req.params.id;
  // Logic to delete post by postId
  res.status(200).json({ message: `Post ${postId} deleted` });
});
```

e. app.patch()

`app.patch()` handles HTTP **PATCH** requests, used to update part of a resource (as opposed to `PUT`, which replaces the entire resource).

```
app.patch('/users/:id', (req, res) => {
  const userId = req.params.id;
  const { email } = req.body; // Assume we're only updating the email
  // Update the user's email in the database here
```

```
  res.json({ message: `User ${userId} email updated`, email });
});
```

Parameters:

- **Path (string)**: The URL or endpoint, often with a dynamic parameter (e.g., `/users/:id`).
- **Callback Function**: Receives the partial update data in `req.body`.

Example: Partial Update

```
app.patch('/profiles/:id', (req, res) => {
  const profileId = req.params.id;
  const updates = req.body; // Partial updates like { age: 30 }
  // Apply updates to the profile in the database
  res.json({ message: `Profile ${profileId} updated`, updates });
});
```

f. app.use()

`app.use()` mounts middleware functions, which are functions that execute during the request-response cycle. Middleware can modify the `Request` and `Response` objects or terminate the request-response cycle.

Middleware Example: Global Middleware

```
app.use((req, res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
  next(); // Pass control to the next middleware or route handler
});
```

Middleware Example: Applying to a Specific Route

You can apply middleware to specific routes or route groups by passing a path as the first argument to `app.use()`.

```
const authMiddleware = (req, res, next) => {
  const token = req.headers['authorization'];
```

```
  if (!token) return res.status(401).json({ message: 'Unauthorized' });
  // Verify token here (e.g., JWT verification)
  next();
};

app.use('/api/protected', authMiddleware, (req, res) => {
  res.json({ message: 'You have access to the protected route' });
});
```

Example: Using Built-in Middleware

Express provides built-in middleware to handle common tasks like parsing JSON and serving static files.

```
// Built-in middleware to serve static files (HTML, CSS, images)
app.use(express.static('public'));

// Built-in middleware to parse incoming JSON data
app.use(express.json());

// Built-in middleware to parse URL-encoded data (from HTML forms)
app.use(express.urlencoded({ extended: true }));
```

Example: Third-Party Middleware

You can also use third-party middleware, such as `morgan` for logging or `cors` for Cross-Origin Resource Sharing.

```
import cors from 'cors';
import morgan from 'morgan';

// Use morgan for logging requests
app.use(morgan('dev'));

// Enable CORS for all routes
app.use(cors());
```

Request Object

The Request object (`req`) in Express represents the HTTP request and provides information about the request made to the server. It contains properties and methods to handle data such as query parameters, route parameters, headers, the request body, and much more.

a. What is Response Object?

While the Request object represents the incoming HTTP request, the **Response object (`res`)** represents the HTTP response that an Express app sends when it gets an incoming request. It is used to send data back to the client (e.g., JSON, HTML, status codes, etc.).

Example:

```
app.get('/example', (req, res) => {
  res.status(200).json({ message: 'Hello, world!' });
});
```

b. Request Object Methods

The Request object provides several methods and properties for handling client requests. Below are some important ones:

- **`req.query`**: Access query parameters (e.g., `/users?name=John`).
- **`req.params`**: Access route parameters (e.g., `/users/:id`).
- **`req.body`**: Access the request body, commonly used in POST and PUT requests.
- **`req.headers`**: Access headers sent by the client.
- **`req.method`**: Get the HTTP method used in the request (GET, POST, etc.).
- **`req.url`**: The full URL requested.

i. Query Parameters (req.query)

Query parameters are typically used to pass optional data to the server in the URL, usually for filtering, sorting, limiting, or pagination. They are appended to the URL after a `?` symbol.

Accessing Query Parameters:

```
app.get('/users', (req, res) => {
  const { name, age } = req.query;
  res.json({ name, age });
});
```

Example URL:

```
/users?name=John&age=25
```

The above URL would populate `req.query` as:

```
{
  "name": "John",
  "age": "25"
}
```

ii. Operations which Require Query Params

a. Filtering

Filtering allows clients to retrieve specific data based on certain conditions. You can use query parameters to filter results.

Example:

```
app.get('/products', (req, res) => {
  const { category } = req.query;
  // Logic to filter products based on category
  res.json({ message: `Filtered products by category: ${category}` });
});
```

b. Sorting

Sorting allows clients to receive data in a specific order (e.g., ascending or descending based on a field).

Example:

```
app.get('/products', (req, res) => {
  const { sortBy, order } = req.query;
  // Logic to sort products based on sortBy field and order ('asc' or
'desc')
  res.json({ message: `Sorted by ${sortBy} in ${order} order` });
});
```

Example URL:

```
/products?sortBy=price&order=asc
```

c. Limiting

Limiting allows you to control how many results are returned from a query. This is especially useful for large datasets.

Example:

```
app.get('/products', (req, res) => {
  const { limit } = req.query;
  // Logic to limit the number of products returned
  res.json({ message: `Limited to ${limit} products` });
});
```

Example URL:

```
/products?limit=10
```

d. Pagination

Pagination allows you to break large datasets into smaller chunks and fetch data page by page.

Example:

```
app.get('/products', (req, res) => {
  const { page, pageSize } = req.query;
  const offset = (parseInt(page as string) - 1) * parseInt(pageSize as
string);
  // Logic to return paginated results
  res.json({ message: `Page: ${page}, Page Size: ${pageSize}`, offset
});
});
```

Example URL:

```
/products?page=2&pageSize=10
```

ii. Route Parameters (req.params)

Route parameters are named URL segments that act as placeholders for data. They are defined in the route path with a colon (:) and can be accessed using `req.params`.

Example:

```
app.get('/users/:id', (req, res) => {
  const { id } = req.params;
  // Logic to fetch user by id
  res.json({ message: `User ID: ${id}` });
});
```

Example URL:

```
/users/123
```

In this case, `req.params` would contain:

```
{
  "id": "123"
}
```

Route parameters are often used when you want to reference a specific resource (like a user, product, or article) by its unique identifier.

i. Operations which Require Route Params

a. Fetching a Single Resource

You can use route parameters to fetch a specific resource based on a unique identifier (e.g., fetching a single user by their ID).

Example:

```
app.get('/users/:id', (req, res) => {
  const { id } = req.params;
  // Fetch user by id
  res.json({ message: `Fetching user with ID: ${id}` });
});
```

b. Updating a Specific Resource

Route parameters can also be used for updating a specific resource.

Example:

```
app.put('/users/:id', (req, res) => {
  const { id } = req.params;
  const { name, email } = req.body;
  // Update the user in the database
  res.json({ message: `User with ID ${id} updated`, name, email });
});
```

c. Deleting a Specific Resource

You can use route parameters to delete a specific resource based on its unique ID.

Example:

```
app.delete('/users/:id', (req, res) => {
  const { id } = req.params;
  // Delete the user from the database
```

```
  res.status(204).json({ message: `User with ID ${id} deleted` });
});
```

Summary

- **Query Parameters (`req.query`)**: Used for passing optional data like filtering, sorting, limiting, and pagination via URL queries (e.g., `/users?name=John`).
- **Route Parameters (`req.params`)**: Used for passing dynamic data directly in the URL path (e.g., `/users/:id`), commonly used for operations like fetching, updating, or deleting a resource.

Response Object

The **Response object (`res`)** in Express is used to send the HTTP response to the client after a request is processed. It contains various methods that allow you to manipulate the response in different ways, such as sending JSON, setting the status code, sending files, or ending the request-response cycle.

a. What is the Response Object?

The `res` object represents the HTTP response that an Express app sends when it receives an incoming request. It provides various methods to send different types of responses to the client, such as JSON data, plain text, HTML, or files.

Example:

```
app.get('/example', (req, res) => {
  res.status(200).json({ message: 'Hello, World!' });
});
```

b. Response Methods

Express provides several useful methods within the `res` object to handle various response types. Here's a breakdown of commonly used methods.

i. res.json()

The `res.json()` method is used to send a JSON-formatted response. This method automatically sets the `Content-Type` header to `application/json`.

Example:

```
app.get('/users', (req, res) => {
  const users = [{ id: 1, name: 'John Doe' }, { id: 2, name: 'Jane Doe'
}];
  res.json(users);
});
```

In this example, a JSON object (array of users) is sent as the response. The `res.json()` method is ideal for RESTful APIs, where you often need to send structured JSON data to clients.

How It Works:

- Automatically converts JavaScript objects to JSON strings.
- Sets the `Content-Type` header to `application/json`.

Example Output:

```
[
  { "id": 1, "name": "John Doe" },
  { "id": 2, "name": "Jane Doe" }
]
```

ii. res.status()

The `res.status()` method sets the HTTP status code of the response. Status codes are essential for indicating the result of an HTTP request (e.g., `200` for success, `404` for not found, `500` for server errors).

Example:

```
app.get('/notfound', (req, res) => {
  res.status(404).json({ message: 'Resource not found' });
});
```

In this example, a 404 (Not Found) status is returned, along with a JSON message.

How It Works:

- Sets the HTTP status code for the response.
- Must be called before sending the final response with `res.send()`, `res.json()`, etc.

Common Status Codes:

- `200` – OK (successful request)
- `201` – Created (resource successfully created)
- `400` – Bad Request (client-side error)
- `404` – Not Found (resource not found)
- `500` – Internal Server Error (server-side error)

iii. res.send()

The `res.send()` method is used to send a response body of various types, including strings, buffers, or objects. If an object is passed, Express will automatically convert it to JSON (similar to `res.json()`, but less explicit).

Example:

```
app.get('/hello', (req, res) => {
  res.send('Hello, World!');
});
```

This sends a plain text response to the client.

How It Works:

- Can send strings, arrays, buffers, or objects.
- Automatically sets the `Content-Type` based on the argument type (e.g., `text/plain` for strings, `application/json` for objects).

Example:

```
app.get('/data', (req, res) => {
  res.send({ message: 'This is an object response' });
});
```

iv. res.redirect()

The `res.redirect()` method is used to redirect the client to a different URL. It sends a `302 Found` status code by default, but this can be changed to another redirect status code like `301 Moved Permanently`.

Example:

```
app.get('/old-route', (req, res) => {
  res.redirect('/new-route');
});
```

In this example, a client requesting `/old-route` will be redirected to `/new-route`.

How It Works:

- By default, the status code is `302` (temporary redirect).
- You can pass a status code as the first argument to specify a different type of redirect.

Example:

```
app.get('/permanent-redirect', (req, res) => {
  res.redirect(301, '/new-permanent-route');
});
```

In this case, the server responds with a `301 Moved Permanently` status, indicating that the resource has been moved to a new location.

v. res.download()

The `res.download()` method prompts the client to download a file from the server. It automatically sets the appropriate headers so that the client knows it's receiving a file to download, rather than to render in the browser.

Example:

```
app.get('/download', (req, res) => {
  const filePath = './files/sample.pdf';
  res.download(filePath, 'sample.pdf', (err) => {
    if (err) {
```

```
      res.status(500).send('Error downloading file');
    }
  });
});
```

In this example, the client will download the file `sample.pdf` from the server.

How It Works:

- Automatically sets the `Content-Disposition` header to trigger the download.
- The second argument is optional and allows you to specify a custom filename for the downloaded file.

vi. res.end()

The `res.end()` method is used to end the response process without sending any data. It is useful in situations where you need to terminate the connection, and no further data is to be sent.

Example:

```
app.get('/close', (req, res) => {
  res.status(204).end(); // 204 No Content
});
```

In this example, the `res.end()` method is used to end the response without sending any content back to the client.

How It Works:

- Ends the response process.
- You should call this when you've finished writing to the response and want to terminate the connection.
- It does not send any data or body content, just ends the response.

Summary

- **`res.json()`**: Sends a JSON response. Automatically sets the `Content-Type` header to `application/json`.
- **`res.status()`**: Sets the HTTP status code for the response. Should be called before `res.send()` or `res.json()`.
- **`res.send()`**: Sends a response of various types (strings, objects, etc.). Automatically converts objects to JSON if needed.
- **`res.redirect()`**: Redirects the client to a different URL. The default status code is `302` (Found), but it can be changed to `301` for permanent redirects.
- **`res.download()`**: Prompts the client to download a file from the server.
- **`res.end()`**: Ends the response process without sending any additional data. Used when no further data needs to be sent to the client.

This collection of response methods helps you handle a variety of use cases, whether you're sending JSON data, redirecting users, or handling file downloads.

Routing (Express.Router())

Express provides a powerful and flexible way to define routing for your web application or API. The `express.Router` is a modular, mountable route handler that allows you to organize your routes into separate files and group them logically.

Routing refers to how an application responds to client requests for a particular endpoint (URL path and HTTP method). Express routes can be simple, complex, or even organized into separate files using `express.Router()`.

Basic Routing

Basic routing involves defining routes for HTTP methods (e.g., GET, POST, PUT, DELETE) using simple callback functions. This is typically used for handling a single endpoint.

Example:

```
import express, { Request, Response } from 'express';

const app = express();

app.get('/home', (req: Request, res: Response) => {
  res.send('Welcome to the Home Page!');
});

app.post('/submit', (req: Request, res: Response) => {
  res.send('Form submitted!');
});

app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

In this example:

- `app.get('/home', ...)`: Handles HTTP GET requests to `/home`.
- `app.post('/submit', ...)`: Handles HTTP POST requests to `/submit`.

Complex Routing with express.Router

As your application grows, managing all your routes in a single file becomes impractical. To address this, Express provides the `Router()` function to create modular and mountable route handlers. Each router can handle its own set of routes and be exported from a file for use in the main application.

Example: Using express.Router for Complex Routing

```
import express, { Request, Response } from 'express';

const router = express.Router();

// Route to get all users
router.get('/users', (req: Request, res: Response) => {
  res.json([{ id: 1, name: 'John Doe' }, { id: 2, name: 'Jane Doe' }]);
});

// Route to get a user by ID
router.get('/users/:id', (req: Request, res: Response) => {
  const { id } = req.params;
  res.json({ id, name: 'John Doe' });
});

// Route to create a new user
router.post('/users', (req: Request, res: Response) => {
  const { name } = req.body;
  res.status(201).json({ id: 3, name });
});

export default router;
```

In the main application:

```
import express from 'express';
import userRoutes from './routes/userRoutes'; // Importing router
```

```
const app = express();
app.use(express.json());

// Mount the user routes under /api
app.use('/api', userRoutes);

app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

In this example:

- We define several routes using `express.Router()`.
- We then mount the router to the main application using `app.use('/api', userRoutes)`.
- The API now has routes like `/api/users` and `/api/users/:id`.

Various Ways to Structure Routes

There are different ways to structure routes depending on the complexity and size of your application. Let's explore some common ways to organize routes in Express.

1. Flat Structure (Basic Applications)

For small applications, a simple flat structure is often sufficient:

```
src/
  ├── app.ts        # Main application file
  └── routes/
      └── index.ts  # All routes defined in one file
```

```
// src/routes/index.ts
import express from 'express';

const router = express.Router();

router.get('/', (req, res) => res.send('Hello World'));
```

```
router.get('/about', (req, res) => res.send('About Page'));

export default router;

// src/app.ts
import express from 'express';
import routes from './routes/index';

const app = express();
app.use('/', routes);

app.listen(3000, () => console.log('Server running'));
```

This is simple and ideal for applications with just a few routes.

2. Modular Structure (Medium Applications)

For larger applications, you might organize routes by resource (e.g., users, posts, products).

```
src/
  ├── app.ts
  └── routes/
      ├── userRoutes.ts
      ├── postRoutes.ts
      └── index.ts
```

```
// src/routes/userRoutes.ts
import express from 'express';
const router = express.Router();

router.get('/', (req, res) => res.send('Get all users'));
router.post('/', (req, res) => res.send('Create user'));

export default router;

// src/routes/postRoutes.ts
```

```
import express from 'express';
const router = express.Router();

router.get('/', (req, res) => res.send('Get all posts'));
router.post('/', (req, res) => res.send('Create post'));

export default router;

// src/routes/index.ts
import express from 'express';
import userRoutes from './userRoutes';
import postRoutes from './postRoutes';

const router = express.Router();

router.use('/users', userRoutes);
router.use('/posts', postRoutes);

export default router;

// src/app.ts
import express from 'express';
import routes from './routes';

const app = express();
app.use('/api', routes);

app.listen(3000, () => console.log('Server running'));
```

In this case:

- Routes are separated by resource (`users`, `posts`).
- The `routes/index.ts` file imports and consolidates the other route files.

3. Advanced Structure (Large Applications)

For large applications, you may want to include controllers, services, and middleware, each in separate directories to promote separation of concerns.

```
src/
  ├── app.ts
  ├── controllers/
  │   ├── userController.ts
  │   └── postController.ts
  ├── routes/
  │   ├── userRoutes.ts
  │   └── postRoutes.ts
  └── services/
      ├── userService.ts
      └── postService.ts
```

Example:

```
// src/controllers/userController.ts
import { Request, Response } from 'express';
import { getAllUsers } from '../services/userService';

export const getUsers = (req: Request, res: Response) => {
  const users = getAllUsers();
  res.json(users);
};

// src/routes/userRoutes.ts
import express from 'express';
import { getUsers } from '../controllers/userController';

const router = express.Router();
router.get('/', getUsers);

export default router;

// src/app.ts
import express from 'express';
```

```
import userRoutes from './routes/userRoutes';

const app = express();
app.use('/api/users', userRoutes);

app.listen(3000, () => console.log('Server running'));
```

This structure promotes clean code and separation of concerns. Each layer (controllers, services, routes) has its responsibility, which makes it easier to manage and scale.

Best Practices for Structuring Routes

1. **Organize by Feature or Resource**:
   - Group related routes by resource (e.g., `/users`, `/posts`) rather than HTTP method (e.g., `/getUser`, `/createUser`). This promotes RESTful practices and cleaner route structures.
2. **Use `express.Router` for Modular Code**:
   - Define routes in separate files/modules using `express.Router()`. This keeps your code modular, easier to maintain, and allows for cleaner imports in the main application.
3. **Use Route Controllers for Logic**:
   - Avoid placing complex logic directly in route definitions. Instead, delegate business logic to controller functions, which handle processing and return the response.
4. **Use Middleware for Reusable Logic**:
   - Implement middleware functions for common functionality like authentication, logging, or request validation. Apply middleware either globally or to specific routes.
5. **Separate Concerns**:
   - Keep route definitions, controllers, services, and business logic in separate directories to maintain clear boundaries between different parts of the application.
6. **Keep the Router DRY (Don't Repeat Yourself)**:

- Avoid repeating common route prefixes. For example, mount resource routes under `/api` once instead of specifying `/api` for every route.

7. **Use Named Route Parameters for Dynamic Data**:
   - Use route parameters like `/users/:id` for accessing specific resources instead of relying on query parameters.

8. **Handle Errors in Routes**:
   - Ensure that routes handle errors gracefully by catching exceptions and returning meaningful error responses with appropriate status codes.

Example of error handling:

```
router.get('/:id', async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const user = await getUserById(req.params.id);
    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }
    res.json(user);
  } catch (error) {
    next(error);
  }
});
```

Summary

- **Basic Routing**: Define simple routes using HTTP methods like `GET`, `POST`, `PUT`, etc.
- **Complex Routing**: Use `express.Router()` to organize routes in a modular way for larger applications.
- **Structure**: Depending on the application's size, routes can be organized into flat, modular, or advanced structures with separation of concerns (controllers, services, middleware).

- **Best Practices**: Group routes by resource, delegate logic to controllers, use middleware for reusable logic, and handle errors properly.

Serving static files & Uploading files

7. Serving Static Files & Uploading Files Using Multer

In Express.js, serving static files like HTML, CSS, JavaScript, and images, as well as handling file uploads (e.g., user profile pictures, documents), is a common requirement for many web applications. Express provides built-in functionality for serving static files, and file uploads can be managed using **Multer**, a popular middleware for handling multipart/form-data, which is primarily used for uploading files.

Serving Static Files in Express

To serve static files (e.g., images, CSS files, JavaScript files), you can use the built-in middleware `express.static`. This middleware serves the files directly from the directory you specify.

Example: Basic Setup for Serving Static Files

```
import express from 'express';
import path from 'path';

const app = express();
const PORT = 3000;

// Serve static files from the "public" directory
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', (req, res) => {
  res.send('Welcome to the homepage!');
});

app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Directory Structure:

```
src/
  ├── app.ts
  └── public/
      ├── index.html
      ├── styles.css
      └── script.js
```

In this example:

- **`express.static(path.join(__dirname, 'public'))`**: This middleware serves static files from the `public` directory. Any file inside the `public` folder can be accessed directly via a URL. For example, an image located at `public/images/logo.png` can be accessed via `http://localhost:3000/images/logo.png`.

Serving Static Files at a Specific Route:

You can also mount static file serving to a specific route. For example:

```
app.use('/static', express.static(path.join(__dirname, 'public')));
```

Now, files in the `public` folder will be served from `http://localhost:3000/static/`. For example, the file `public/images/logo.png` will be accessible via `http://localhost:3000/static/images/logo.png`.

Uploading Files Using Multer

Multer is a middleware for handling `multipart/form-data`, primarily used for uploading files in Express applications. It allows you to handle file uploads easily and store them locally or upload them to cloud storage (e.g., AWS S3, Google Cloud).

Installation:

```
npm install multer
npm install @types/multer --save-dev
```

Basic File Upload Example

1. **Set up Multer in your Express application**: Multer requires setting up a storage engine that defines how and where to store uploaded files.

```
import express from 'express';
import multer from 'multer';
import path from 'path';

const app = express();
const PORT = 3000;

// Set up Multer storage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/'); // Directory where files will be saved
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() *
1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix +
path.extname(file.originalname)); // Custom filename
  }
});

const upload = multer({ storage });

app.use(express.static('public'));

// Simple file upload route
app.post('/upload', upload.single('file'), (req, res) => {
  res.send('File uploaded successfully!');
});

app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

In this example:

- **`multer.diskStorage`**: Defines the storage engine. The uploaded file is stored in the `uploads` directory, and the file name is appended with a unique timestamp.
- **`upload.single('file')`**: Middleware to handle single file uploads. The string `'file'` should match the name attribute of the file input in your HTML form.

HTML Form for File Upload (Public Folder):

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>File Upload</title>
</head>
<body>
  <h1>Upload a File</h1>
  <form action="/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="file" />
    <button type="submit">Upload</button>
  </form>
</body>
</html>
```

In this form:

- **`enctype="multipart/form-data"`**: This is important for handling file uploads. Without it, the file won't be included in the form data sent to the server.

Accessing Uploaded File Information:

After the file is uploaded, you can access its details via `req.file`.

```
app.post('/upload', upload.single('file'), (req, res) => {
  console.log(req.file); // Information about the uploaded file
```

```
  res.send('File uploaded successfully!');
});
```

The `req.file` object contains useful information about the uploaded file, such as:

- `filename`: The name of the file on the server.
- `originalname`: The original name of the uploaded file.
- `mimetype`: The MIME type of the file.
- `size`: The size of the uploaded file.

Uploading Multiple Files

You can also configure Multer to handle multiple file uploads at once.

```
app.post('/uploads', upload.array('files', 5), (req, res) => {
  console.log(req.files); // Array of uploaded files
  res.send('Multiple files uploaded successfully!');
});
```

In this example:

- **`upload.array('files', 5)`**: This allows the client to upload up to 5 files at once. The `files` field in the HTML form should match the name attribute in the `input` tag.

Example HTML Form for Multiple Files:

```
<form action="/uploads" method="post" enctype="multipart/form-data">
  <input type="file" name="files" multiple />
  <button type="submit">Upload</button>
</form>
```

Handling File Types and Size Limits

You can add validation in Multer to control the file type and size.

Restrict File Type:

You can use the `fileFilter` option to allow only certain types of files (e.g., images).

```
const upload = multer({
  storage,
  fileFilter: (req, file, cb) => {
    const fileTypes = /jpeg|jpg|png|gif/;
    const extName =
fileTypes.test(path.extname(file.originalname).toLowerCase());
    const mimeType = fileTypes.test(file.mimetype);

    if (extName && mimeType) {
      return cb(null, true);
    } else {
      cb(new Error('Only images are allowed!'));
    }
  }
});
```

Restrict File Size:

You can restrict the size of uploaded files using the `limits` option.

```
const upload = multer({
  storage,
  limits: { fileSize: 1024 * 1024 * 2 } // Limit file size to 2MB
});
```

Organizing File Upload Logic

For larger applications, you should organize file upload logic into a more modular structure. You can separate concerns by moving your file upload configurations and logic into separate files.

Example Structure:

```
src/
  ├── app.ts
```

```
  ├── controllers/
  │   └── uploadController.ts
  ├── routes/
  │   └── uploadRoutes.ts
  └── config/
      └── multerConfig.ts
```

- `multerConfig.ts`: Configuration for file uploads.
- `uploadController.ts`: Logic for handling uploaded files.
- `uploadRoutes.ts`: Defines the routes for handling file uploads.

Example multerConfig.ts:

```
import multer from 'multer';
import path from 'path';

const storage = multer.diskStorage({
  destination: 'uploads/',
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() *
1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix +
path.extname(file.originalname));
  }
});

export const upload = multer({
  storage,
  limits: { fileSize: 1024 * 1024 * 5 }, // 5MB limit
  fileFilter: (req, file, cb) => {
    const fileTypes = /jpeg|jpg|png|gif/;
    const extName =
fileTypes.test(path.extname(file.originalname).toLowerCase());
    const mimeType = fileTypes.test(file.mimetype);

    if (extName && mimeType) {
```

```
      return cb(null, true);
    } else {
      cb(new Error('Only images are allowed!'));
    }
  }
});
```

Example uploadController.ts:

```
import { Request, Response } from 'express';

export const uploadFile = (req: Request, res: Response) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded');
  }
  res.send('File uploaded successfully!');
};
```

Example uploadRoutes.ts:

```
import express from 'express';
import { upload } from '../config/multerConfig';
import

 { uploadFile } from '../controllers/uploadController';

const router = express.Router();

router.post('/upload', upload.single('file'), uploadFile);

export default router;
```

Example app.ts:

```
import express from 'express';
import uploadRoutes from './routes/uploadRoutes';
```

```
const app = express();
app.use(express.json());
app.use('/api', uploadRoutes);

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Summary

- **Serving Static Files**: Use the `express.static()` middleware to serve static assets (like HTML, CSS, images) from a directory. These files can be accessed directly via their URL path.
- **File Uploads with Multer**: Use Multer to handle file uploads by configuring storage options (e.g., where to store files) and handling multipart/form-data. You can restrict file size, type, and name uploads for security and consistency.
- **Organizing Upload Logic**: For larger applications, separate concerns into different files (controllers, routes, configurations) to maintain modularity and clean code.

Middlewares

Express middlewares are functions that execute during the request-response cycle. They can perform various tasks such as parsing request bodies, logging, handling errors, rate-limiting, and more. Middlewares can be built-in, custom, or external libraries that you plug into your application.

Middleware Types:

- **Built-in**: Included in the Express framework.
- **External**: Additional middleware provided by third-party libraries.

a. Inbuilt Middlewares

Express has several built-in middlewares that handle common tasks like parsing request bodies. Two of the most commonly used built-in middlewares are `express.json()` and `express.urlencoded()`.

i. express.json()

The `express.json()` middleware is used to parse incoming JSON payloads. It makes the body of the request available in `req.body`. This is commonly used in APIs where JSON data is submitted in POST or PUT requests.

```
import express from 'express';
const app = express();

app.use(express.json()); // Parses JSON requests

app.post('/data', (req, res) => {
  console.log(req.body); // Access the JSON data sent in the request
  res.send('Data received');
});

app.listen(3000, () => {
```

```
  console.log('Server is running on http://localhost:3000');
});
```

ii. express.urlencoded({ extended: true })

This middleware is used to parse URL-encoded payloads, which are usually submitted by HTML forms. It makes the form data available in `req.body`.

- **`extended: true`**: Allows for richer objects and arrays to be encoded.
- **`extended: false`**: Uses the classic encoding of the query-string library.

```
app.use(express.urlencoded({ extended: true })); // Parses URL-encoded
requests

app.post('/submit', (req, res) => {
  console.log(req.body); // Access the form data submitted
  res.send('Form submitted');
});
```

b. External Middlewares

Express has a large ecosystem of third-party middlewares. Here are some useful external middlewares you can integrate into your Express application:

i. express-rate-limit

The `express-rate-limit` middleware helps you limit the number of requests a client can make to your API in a given time window. This is useful for preventing abuse (e.g., DDoS attacks) by limiting the frequency of requests.

Installation:

```
npm install express-rate-limit
```

Example Usage:

```
import express from 'express';
import rateLimit from 'express-rate-limit';

const app = express();

// Rate limit middleware
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per windowMs
  message: 'Too many requests, please try again later.'
});

app.use('/api', limiter); // Apply the rate limit to all API routes

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

ii. pino (with pino-pretty)

`Pino` is a fast and lightweight logging library for Node.js. It's a great alternative to more verbose logging systems like `winston`. When combined with `pino-pretty`, the logs are formatted in a more human-readable way for development.

Installation:

```
npm install pino pino-pretty
```

Example Usage:

```
import express from 'express';
import pino from 'pino';
import pinoHttp from 'pino-http';

const logger = pino({ prettyPrint: true });
const app = express();
```

```
app.use(pinoHttp({ logger })); // Integrate Pino with Express

app.get('/', (req, res) => {
  req.log.info('Request received'); // Example log
  res.send('Hello, Pino logger!');
});

app.listen(3000, () => {
  logger.info('Server running on http://localhost:3000');
});
```

With `pino-pretty`, the logs are displayed in a cleaner, human-friendly format.

iii. prom-client

The `prom-client` middleware helps integrate **Prometheus** metrics into your Express application. This allows you to track various performance metrics like response times, request counts, and more.

Installation:

```
npm install prom-client
```

Example Usage:

```
import express from 'express';
import client from 'prom-client';

const app = express();

// Create metrics for response time
const restResponseTimeHistogram = new client.Histogram({
  name: 'rest_response_time_duration_seconds',
  help: 'REST API response time in seconds',
  labelNames: ['method', 'route', 'status_code'],
});
```

```
const databaseResponseTimeHistogram = new client.Histogram({
  name: 'database_response_time_duration_seconds',
  help: 'Database response time in seconds',
  labelNames: ['operation'],
});

// Middleware to start timer for response time
app.use((req, res, next) => {
  const end = restResponseTimeHistogram.startTimer();
  res.on('finish', () => {
    end({ method: req.method, route: req.route?.path, status_code:
res.statusCode });
  });
  next();
});

app.get('/metrics', async (req, res) => {
  res.set('Content-Type', client.register.contentType);
  res.end(await client.register.metrics());
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Key Metrics:

1. `restResponseTimeHistogram`: Tracks the REST API's response time.
2. `databaseResponseTimeHistogram`: Tracks response times for database queries (you would call this in your database service).

iv. morgan

`morgan` is a popular HTTP request logger middleware. It provides detailed logs about requests made to your server, which can be very useful for debugging and monitoring.

Installation:

```
npm install morgan
```

Example Usage:

1. Basic Logging to Console:

```
import express from 'express';
import morgan from 'morgan';

const app = express();

// Use morgan for logging HTTP requests
app.use(morgan('dev')); // 'dev' is a pre-defined format

app.get('/', (req, res) => {
  res.send('Hello, Morgan Logger!');
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

2. Logging with pino (using pino-pretty):

You can replace `morgan` with `pino` for logging if you need high performance logging.

```
import express from 'express';
import pino from 'pino';
import pinoHttp from 'pino-http';

const logger = pino({ prettyPrint: true });

const app = express();
app.use(pinoHttp({ logger }));

app.get('/', (req, res) => {
  req.log.info('Logging request'); // Log using Pino
  res.send('Hello, Pino and Express!');
```

```
});

app.listen(3000, () => {
  logger.info('Server is running');
});
```

3. Writing Logs to a File:

You can also configure `morgan` to log requests to a file instead of the console.

```
import express from 'express';
import morgan from 'morgan';
import fs from 'fs';
import path from 'path';

const app = express();

// Create a write stream for logging into a file
const accessLogStream = fs.createWriteStream(path.join(__dirname,
'access.log'), { flags: 'a' });

// Use morgan for logging into a file
app.use(morgan('combined', { stream: accessLogStream }));

app.get('/', (req, res) => {
  res.send('Logging requests to a file');
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

v. Swagger (swagger-ui-express & swagger-jsdoc)

Swagger is a set of tools to help document and visualize REST APIs. You can use `swagger-ui-express` to serve a Swagger UI and `swagger-jsdoc` to generate the OpenAPI specification (Swagger docs) from comments in your code.

Installation:

```
npm install swagger-ui-express swagger-jsdoc
```

Example Setup:

```
import express from 'express';
import swaggerUi from 'swagger-ui-express';
import swaggerJsdoc from 'swagger-jsdoc';

const app = express();

const swaggerOptions = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Express API Documentation',
      version: '1.0.0',
      description: 'A sample API'
    },
  },
  apis: ['./src/routes/*.ts'], // Path to your API route files
};

const swaggerDocs = swaggerJsdoc(swaggerOptions);

// Serve Swagger UI
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocs));

// Example API route
app.get('/api', (req, res) => {
  res.send('API works!');
});

app.listen(3000, () => {
```

```
  console.log('Server running on http://localhost:3000');
});
```

Adding Swagger Documentation to Routes:

You can add Swagger documentation directly in your routes using comments.

```
/**
 * @swagger
 * /api:
 *   get:
 *     description: Get API status
 *

 responses:
 *       200:
 *         description: Success
 */
app.get('/api', (req, res) => {
  res.send('API works!');
});
```

In this setup:

- `swagger-ui-express`: Provides a UI to view and interact with the API documentation.
- `swagger-jsdoc`: Parses comments in your code and generates an OpenAPI specification.

Summary

- **Built-in Middlewares**:
  - `express.json()`: Parses incoming JSON requests.
  - `express.urlencoded({ extended: true })`: Parses URL-encoded data (from HTML forms).

- **External Middlewares**:
  - `express-rate-limit`: Prevents abuse by limiting request rates.
  - `pino`: High-performance logging library (with `pino-pretty` for human-readable logs).
  - `prom-client`: Provides Prometheus metrics to monitor API performance.
  - `morgan`: Simple HTTP request logging (to the console or a file).
  - `swagger-ui-express` & `swagger-jsdoc`: Automatically generate and serve API documentation via Swagger.

Schema Validation (Zod)

Zod is a TypeScript-first schema declaration and validation library. It provides a simple and declarative way to define and validate data schemas, which is especially useful when working with user input, API requests, and complex data models in Express.js applications.

Compared to other validation libraries (like Joi or Yup), Zod is tightly integrated with TypeScript, providing full type inference and static type safety, which can help prevent runtime errors and improve developer experience.

Why Zod?

- **TypeScript-first**: Zod automatically infers TypeScript types from your schema, reducing redundancy.
- **Declarative**: You define schemas once and use them for both validation and type-checking.
- **Simple API**: Zod provides a clean and simple interface for creating schemas and validating data.

Installation

To use Zod, first install it via npm:

```
npm install zod
```

Basic Usage of Zod

Zod allows you to define schemas for various data types and validate data against those schemas. Here's an example of how to create a simple schema and validate an object:

Example of Basic Validation:

```
import { z } from 'zod';
```

```
const userSchema = z.object({
  id: z.number(),
  name: z.string().min(3, 'Name must be at least 3 characters long'),
  email: z.string().email(),
  age: z.number().positive().int().optional(), // Optional field
});

const userData = {
  id: 1,
  name: 'John Doe',
  email: 'john@example.com',
};

try {
  userSchema.parse(userData); // Validation successful
  console.log('User data is valid');
} catch (err) {
  console.log(err.errors); // Print validation errors
}
```

In this example:

- **z.number()**: Defines a number type.
- **z.string()**: Defines a string type, with additional .min() for length validation and .email() for email format validation.
- **z.number().positive()**: Ensures the number is positive.
- **.optional()**: Specifies that the field is optional.

Zod with Express

You can integrate Zod with Express for request validation, ensuring incoming data is valid before processing it in your application. Below are examples of how to use Zod for validating incoming data in routes.

Example: Validating Request Body in an Express Route

```
import express from 'express';
import { z } from 'zod';

const app = express();
app.use(express.json());

const userSchema = z.object({
  name: z.string().min(3),
  email: z.string().email(),
  age: z.number().int().positive(),
});

app.post('/users', (req, res) => {
  try {
    // Validate request body against userSchema
    const userData = userSchema.parse(req.body);
    res.status(200).json({ message: 'User created successfully', data:
userData });
  } catch (err) {
    res.status(400).json({ message: 'Validation failed', errors:
err.errors });
  }
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

In this example:

- `userSchema.parse(req.body)`: Validates the incoming request body against the defined schema. If validation passes, the user data is processed; if it fails, an error is returned.
- **Error Handling**: If Zod's `parse()` method throws an error, it will contain an array of validation errors, which can be sent back to the client.

Example Request:

```
curl -X POST http://localhost:3000/users -H "Content-Type:
application/json" -d '{"name": "Jo", "email": "invalidemail", "age":
-5}'
```

This would return:

```
{
  "message": "Validation failed",
  "errors": [
    { "message": "String must contain at least 3 character(s)", "path":
["name"] },
    { "message": "Invalid email", "path": ["email"] },
    { "message": "Number must be greater than 0", "path": ["age"] }
  ]
}
```

Complex Zod Schemas

Zod allows you to build more complex schemas that include nested objects, arrays, enums, and more.

Nested Objects:

You can define schemas for nested objects and validate them.

```
const addressSchema = z.object({
  street: z.string(),
  city: z.string(),
  zipCode: z.string().regex(/^\d{5}$/, 'Invalid zip code format'),
});

const userSchema = z.object({
  id: z.number(),
  name: z.string(),
  email: z.string().email(),
  address: addressSchema, // Nested object
```

```
});

const userData = {
  id: 1,
  name: 'Jane Doe',
  email: 'jane@example.com',
  address: {
    street: '123 Main St',
    city: 'New York',
    zipCode: '10001',
  },
};

userSchema.parse(userData); // Validation passes
```

Arrays:

Zod allows you to validate arrays and ensure that every element in the array conforms to a schema.

```
const postSchema = z.object({
  id: z.number(),
  title: z.string(),
});

const blogSchema = z.object({
  name: z.string(),
  posts: z.array(postSchema), // Array of posts
});

const blogData = {
  name: 'My Blog',
  posts: [
    { id: 1, title: 'First Post' },
    { id: 2, title: 'Second Post' },
  ],
};
```

```
blogSchema.parse(blogData); // Validation passes
```

Enums:

You can define a set of valid values using Zod's `z.enum()` method.

```
const userTypeSchema = z.enum(['admin', 'user', 'guest']);

const userData = {
  id: 1,
  type: 'admin',
};

userTypeSchema.parse(userData.type); // Validation passes
```

Asynchronous Validation

In some cases, validation might need to be asynchronous (e.g., validating data from an external API or checking if a record exists in a database). Zod allows you to write async validations with `refine`.

Example: Async Validation with refine

```
const userSchema = z.object({
  email: z.string().email(),
}).refine(async (data) => {
  // Simulate checking if email exists in the database
  const emailExists = await fakeDatabaseCheck(data.email);
  return !emailExists;
}, {
  message: 'Email is already in use',
  path: ['email'],
});

async function fakeDatabaseCheck(email: string): Promise<boolean> {
  const existingEmails = ['existing@example.com'];
  return existingEmails.includes(email);
```

```
}

userSchema.parseAsync({ email: 'new@example.com' })
  .then(() => console.log('Valid email'))
  .catch(err => console.log('Validation failed', err.errors));
```

In this example:

- **refine()**: Adds a custom validation rule. Here, it checks if the email exists in a simulated database.
- **parseAsync()**: Used for asynchronous schema validation.

Custom Error Messages

You can customize error messages to provide more meaningful feedback to the user.

```
const userSchema = z.object({
  name: z.string().min(3, { message: 'Name must be at least 3 characters
long' }),
  email: z.string().email({ message: 'Invalid email format' }),
});

try {
  userSchema.parse({ name: 'Jo', email: 'invalidemail' });
} catch (err) {
  console.log(err.errors);
}
```

Output:

```
[
  { "message": "Name must be at least 3 characters long", "path":
["name"] },
  { "message": "Invalid email format", "path": ["email"] }
]
```

Integration with Express Middleware

You can create reusable validation middleware using Zod for Express. This middleware can be used to validate request bodies, query parameters, or route parameters.

Example: Zod Validation Middleware

```
import { Request, Response, NextFunction } from 'express';
import { z } from 'zod';

const validate = (schema: z.ZodSchema) => (req: Request, res: Response,
next: NextFunction) => {
  try {
    schema.parse(req.body);
    next();
  } catch (err) {
    res.status(400).json({ errors: err.errors });
  }
};

const userSchema = z.object({
  name: z.string().min(3),
  email: z.string().email(),
});

app.post('/users', validate(userSchema), (req, res) => {
  res.status(200).json({ message: 'User created successfully' });
});
```

In this example:

- `validate`: Middleware that validates the request body against the provided schema and returns errors if validation fails.

Summary

- **Zod** is a TypeScript-first validation library that provides type-safe schemas for data validation.
- **Basic Usage**: Zod schemas can be defined to validate simple types (strings, numbers) and more complex structures (nested objects, arrays, enums).
- **Express Integration**: Zod can be integrated into Express routes to validate incoming request data. You can create reusable validation middleware for request bodies, query parameters, or route parameters.
- **Asynchronous Validation

**: Use `refine()` for custom async validation logic (e.g., checking database entries).

- **Custom Error Messages**: You can provide more meaningful error messages using Zod's error message options.
- **Middleware**: Zod can be wrapped into a middleware for cleaner and reusable validation handling in Express.

JWT (Authentication & Authorization)

JWT (JSON Web Token) is a compact and self-contained way of transmitting information between parties as a JSON object. It is often used for securing APIs by authenticating users via tokens rather than sessions.

JWT consists of three parts:

1. **Header**: Contains information about the algorithm used (usually HS256).
2. **Payload**: Contains the claims (user data, permissions, etc.).
3. **Signature**: A unique string created by encoding the header and payload with a secret key.

In a typical JWT authentication setup, a client sends credentials (like email and password), the server validates them, and if successful, the server generates a JWT token, which the client can use to authenticate itself on subsequent requests.

a. Token Creation

To implement JWT in Express, you'll need the `jsonwebtoken` library to create and verify tokens.

Installation

```
npm install jsonwebtoken bcryptjs
```

Here, we'll also use `bcryptjs` for hashing passwords.

Example Setup: Generating a JWT Token

1. **Create a User Authentication Flow:**

- When the user logs in or signs up, the server generates a JWT token and sends it to the client. The client stores the token (usually in localStorage or cookies) and includes it in the `Authorization` header of subsequent requests.

Token Creation Example

```
import express from 'express';
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';

const app = express();
const SECRET_KEY = 'your-secret-key'; // Ideally store this in
environment variables

app.use(express.json());

// Mock user database
const users: { id: number, email: string, password: string }[] = [];

// Create a JWT token for a user
const createToken = (user: { id: number; email: string }) => {
  return jwt.sign({ id: user.id, email: user.email }, SECRET_KEY, {
expiresIn: '1h' });
};

// Example user registration
app.post('/register', async (req, res) => {
  const { email, password } = req.body;

  // Hash the password before storing it
  const hashedPassword = await bcrypt.hash(password, 10);
  const newUser = { id: users.length + 1, email, password:
hashedPassword };
  users.push(newUser);

  const token = createToken(newUser);
  res.status(201).json({ message: 'User registered successfully', token
});
});

// Example user login
```

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = users.find(u => u.email === email);

  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(401).json({ message: 'Invalid email or password'
});
  }

  const token = createToken(user);
  res.json({ message: 'Login successful', token });
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

In this setup:

- **Password Hashing**: User passwords are hashed using `bcryptjs` before being stored in the database (in this example, a simple in-memory array).
- **JWT Token**: Upon successful login or registration, a JWT token is generated using `jwt.sign()`, with the user's ID and email as the payload. The token expires in 1 hour.

Example Request for Token Creation (Login):

```
curl -X POST http://localhost:3000/login \
  -H "Content-Type: application/json" \
  -d '{"email": "john@example.com", "password": "secret"}'
```

b. User Management API (Authorization & Authentication)

JWT Authentication involves two key processes:

1. **Authentication**: Verifying the user's identity (logging in).

2. **Authorization**: Ensuring that a user has the correct permissions to access specific resources.

1. Authentication: Protecting Routes with JWT

Once the token is created, the client sends the token in the `Authorization` header for protected routes. The server will verify the token using `jwt.verify()` to ensure that the token is valid.

Example of Protecting a Route:

```
// Middleware to verify the JWT token
const authenticateToken = (req: express.Request, res: express.Response,
next: express.NextFunction) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Bearer token

  if (!token) return res.status(401).json({ message: 'No token provided'
});

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ message: 'Invalid or expired
token' });

    req.user = user; // Attach user info to the request object
    next(); // Proceed to the next middleware or route handler
  });
};

// Protected route
app.get('/protected', authenticateToken, (req, res) => {
  res.json({ message: `Welcome, user with email ${req.user.email}`,
user: req.user });
});
```

In this example:

- `authenticateToken` **Middleware**: Verifies the token sent by the client. If the token is valid, it adds the user data to the request object and allows access to the protected route.
- **Protected Route**: Only accessible if the client sends a valid JWT token.

Example Request to a Protected Route:

```
curl -X GET http://localhost:3000/protected \
  -H "Authorization: Bearer <your-jwt-token>"
```

If the token is valid, the server will return the protected data. Otherwise, it will respond with a `401 Unauthorized` or `403 Forbidden` status.

Authorization: Role-based Access Control (RBAC)

You can add an additional layer of authorization by assigning roles to users (e.g., "admin", "user") and verifying their permissions when accessing certain routes.

Example: Role-based Authorization

1. **Adding Roles to JWT Token**: When generating the JWT token, include the user's role in the payload.

```
const createToken = (user: { id: number; email: string; role: string })
=> {
  return jwt.sign({ id: user.id, email: user.email, role: user.role },
SECRET_KEY, { expiresIn: '1h' });
};
```

2. **Role-based Middleware**: Create middleware to check if the authenticated user has the correct role to access a resource.

```
const authorizeRole = (role: string) => {
  return (req: express.Request, res: express.Response, next:
express.NextFunction) => {
    if (req.user.role !== role) {
```

```
      return res.status(403).json({ message: 'Access forbidden:
Insufficient permissions' });
    }
    next();
  };
};

// Example protected route only accessible by admins
app.get('/admin', authenticateToken, authorizeRole('admin'), (req, res)
=> {
  res.json({ message: 'Welcome Admin!' });
});
```

In this setup:

- `authorizeRole` **Middleware**: Checks if the authenticated user has the required role to access the route.

Example User Login with Role:

```
// Mock user data with roles
const users: { id: number, email: string, password: string, role: string
}[] = [
  { id: 1, email: 'admin@example.com', password: 'hashedpassword', role:
'admin' },
  { id: 2, email: 'user@example.com', password: 'hashedpassword', role:
'user' },
];
```

Handling Token Expiration and Refresh Tokens

JWT tokens are often set to expire after a short time (e.g., 1 hour) to reduce security risks. You can issue a new token by implementing a **refresh token** strategy.

1. **Access Token**: Short-lived token used for API requests.
2. **Refresh Token**: Longer-lived token stored securely on the client side, used to request

a new access token when it expires.

Example of Refresh Token Flow:

```
const refreshTokens: string[] = []; // In-memory store for refresh
tokens

// Generate a new refresh token
const createRefreshToken = (user: { id: number; email: string }) => {
  const refreshToken = jwt.sign({ id: user.id, email: user.email },
SECRET_KEY, { expiresIn: '7d' });
  refreshTokens.push(refreshToken); // Store refresh token
  return refreshToken;
};

// Route to generate new access token using refresh token
app.post('/token', (req, res) => {
  const { token } = req.body;
  if (!token || !refreshTokens.includes(token)) {
    return res.status(403).json({ message: 'Invalid refresh token' });
  }

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ message: 'Invalid refresh
token' });

    const accessToken = createToken(user);
    res.json({ accessToken });
  });
});
```

In this setup:

- **Refresh Token Store**: Refresh tokens are stored in memory (for simplicity), but you should store them securely (e.g., in a database or Redis).

- **Token Generation**: When the access token expires, the client sends the refresh token to the server to get a new access token.

Complete User Management API Example

```
import express from 'express';
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';

const app = express();
const SECRET_KEY = 'your-secret-key';
const users:

 { id: number, email: string, password: string, role: string }[] = [];

app.use(express.json());

const createToken = (user: { id: number; email: string; role: string })
=> {
  return jwt.sign({ id: user.id, email: user.email, role: user.role },
SECRET_KEY, { expiresIn: '1h' });
};

const authenticateToken = (req: express.Request, res: express.Response,
next: express.NextFunction) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];
  if (!token) return res.status(401).json({ message: 'No token provided'
});

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ message: 'Invalid or expired
token' });
    req.user = user;
    next();
  });
};
```

```
// Register
app.post('/register', async (req, res) => {
  const { email, password, role } = req.body;
  const hashedPassword = await bcrypt.hash(password, 10);
  const newUser = { id: users.length + 1, email, password:
hashedPassword, role };
  users.push(newUser);
  const token = createToken(newUser);
  res.status(201).json({ message: 'User registered', token });
});

// Login
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = users.find(u => u.email === email);
  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }
  const token = createToken(user);
  res.json({ message: 'Login successful', token });
});

// Protected route
app.get('/profile', authenticateToken, (req, res) => {
  res.json({ message: `Welcome, user with email ${req.user.email}`,
user: req.user });
});

app.listen(3000, () => console.log('Server running on
http://localhost:3000'));
```

Summary

- **Token Creation**: JWT tokens are generated using `jsonwebtoken`. They contain user data and expire after a set time.

- **Authentication**: The token is sent in the `Authorization` header for protected routes and verified using middleware.
- **Authorization**: Role-based access control (RBAC) can be implemented to restrict certain routes based on user roles.
- **Refresh Tokens**: Used to refresh access tokens when they expire, allowing for secure and scalable authentication.

Developing An API

Start typing here...

Sending Emails Programmatically

Sending Emails Programmatically with Express.js and TypeScript

This guide provides a hands-on approach to sending emails programmatically using Express.js with TypeScript. We'll utilize the popular Nodemailer (https://nodemailer.com/about/) library to handle email sending. By the end of this tutorial, you'll have a simple API endpoint that can send emails based on client requests.

Setup

1. Initialize the Project

First, create a new directory for your project and initialize it with npm:

```
mkdir express-email-ts
cd express-email-ts
npm init -y
```

2. Install Dependencies

Install the necessary packages:

```
npm install express nodemailer dotenv
npm install --save-dev typescript @types/express @types/node ts-node-dev
```

- **express**: Web framework for Node.js
- **nodemailer**: Module to send emails
- **dotenv**: Loads environment variables from a `.env` file
- **typescript**, **ts-node-dev**: For TypeScript support
- **@types/***: Type definitions for TypeScript

3. Configure TypeScript

Initialize a TypeScript configuration:

```
npx tsc --init
```

Modify the `tsconfig.json` as needed. A basic configuration might look like this:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  }
}
```

4. Set Up Project Structure

Create the following structure:

```
express-email-ts/
├── src/
│   ├── index.ts
│   └── email.ts
├── .env
├── package.json
└── tsconfig.json
```

Implementation

1. Configure Environment Variables

Create a `.env` file in the root directory to store sensitive information like email credentials:

```
PORT=3000
EMAIL_SERVICE=Gmail
```

```
EMAIL_USER=your-email@gmail.com
EMAIL_PASS=your-email-password
```

Note: For Gmail, you might need to enable Less Secure Apps (https://myaccount.google.com/lesssecureapps) or use App Passwords (https://support.google.com/accounts/answer/185833).

2. Create the Email Module

Create `src/email.ts` to handle email sending logic:

```
// src/email.ts
import nodemailer from 'nodemailer';
import dotenv from 'dotenv';

dotenv.config();

interface EmailOptions {
  to: string;
  subject: string;
  text: string;
  html?: string;
}

const transporter = nodemailer.createTransport({
  service: process.env.EMAIL_SERVICE, // e.g., 'Gmail'
  auth: {
    user: process.env.EMAIL_USER,
    pass: process.env.EMAIL_PASS,
  },
});

export const sendEmail = async (options: EmailOptions) => {
  const mailOptions = {
    from: process.env.EMAIL_USER,
    to: options.to,
    subject: options.subject,
    text: options.text,
```

```
    html: options.html || options.text,
  };

  try {
    const info = await transporter.sendMail(mailOptions);
    console.log('Email sent: ', info.response);
    return info;
  } catch (error) {
    console.error('Error sending email: ', error);
    throw error;
  }
};
```

3. Set Up the Express Server

Create `src/index.ts` to define the API endpoint:

```
// src/index.ts
import express, { Request, Response } from 'express';
import { sendEmail } from './email';
import dotenv from 'dotenv';

dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware to parse JSON
app.use(express.json());

app.post('/send-email', async (req: Request, res: Response) => {
  const { to, subject, text, html } = req.body;

  if (!to || !subject || !text) {
    return res.status(400).json({ message: 'Missing required fields: to,
subject, text' });
  }
```

```
  try {
    await sendEmail({ to, subject, text, html });
    res.status(200).json({ message: 'Email sent successfully' });
  } catch (error) {
    res.status(500).json({ message: 'Failed to send email', error });
  }
});

app.get('/', (req: Request, res: Response) => {
  res.send('Email Service is running.');
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

4. Update package.json Scripts

Add a script to run the server with `ts-node-dev` for hot-reloading:

```
"scripts": {
  "start": "ts-node-dev src/index.ts"
}
```

Running the Application

1. **Start the Server**

   ```
   npm start
   ```

   You should see:

   ```
   Server is running on port 3000
   ```

2. **Send a Test Email**

   Use a tool like Postman (https://www.postman.com/) or `curl` to send a POST request

to `http://localhost:3000/send-email` with a JSON body:

```
{
  "to": "recipient@example.com",
  "subject": "Test Email",
  "text": "Hello! This is a test email sent from Express.js with
TypeScript.",
  "html": "<p>Hello! This is a <strong>test email</strong> sent from
Express.js with TypeScript.</p>"
}
```

Using `curl`:

```
curl -X POST http://localhost:3000/send-email \
  -H "Content-Type: application/json" \
  -d '{
    "to": "recipient@example.com",
    "subject": "Test Email",
    "text": "Hello! This is a test email sent from Express.js with
TypeScript.",
    "html": "<p>Hello! This is a <strong>test email</strong> sent from
Express.js with TypeScript.</p>"
  }'
```

Expected Response:

```
{
  "message": "Email sent successfully"
}
```

Check the recipient's inbox to verify the email was received.

Error Handling and Validation

For production-ready applications, consider enhancing error handling and validating input data. Libraries like Joi (https://joi.dev/) or express-validator (https://express-validator.github.io/docs/) can help with robust validation.

Managing Cron Jobs and Background Services

Managing Cron Jobs and Background Services with Express.js and TypeScript

This guide offers a hands-on approach to managing cron jobs and background services using Express.js with TypeScript. We'll utilize the node-cron (https://www.npmjs.com/package/node-cron) library for scheduling tasks and demonstrate how to integrate background services seamlessly into your Express application. By the end of this tutorial, you'll have a robust setup for running scheduled tasks alongside your API endpoints.

Setup

1. Initialize the Project

First, create a new directory for your project and initialize it with npm:

```
mkdir express-cron-ts
cd express-cron-ts
npm init -y
```

2. Install Dependencies

Install the necessary packages:

```
npm install express node-cron dotenv
npm install --save-dev typescript @types/express @types/node ts-node-dev
```

- **express**: Web framework for Node.js
- **node-cron**: Task scheduler for running cron jobs
- **dotenv**: Loads environment variables from a `.env` file
- **typescript**, **ts-node-dev**: For TypeScript support

- **@types/***: Type definitions for TypeScript

3. Configure TypeScript

Initialize a TypeScript configuration:

```
npx tsc --init
```

Modify the `tsconfig.json` as needed. A basic configuration might look like this:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  }
}
```

4. Set Up Project Structure

Create the following structure:

```
express-cron-ts/
├── src/
│   ├── index.ts
│   ├── jobs/
│   │   └── exampleJob.ts
├── .env
├── package.json
└── tsconfig.json
```

Implementation

1. Configure Environment Variables

Create a `.env` file in the root directory to store configuration variables:

```
PORT=3000
CRON_SCHEDULE=*/5 * * * *
```

- **PORT**: Port number for the Express server
- **CRON_SCHEDULE**: Cron expression defining the schedule (e.g., every 5 minutes)

2. Create a Cron Job Module

Create `src/jobs/exampleJob.ts` to define a sample cron job:

```
// src/jobs/exampleJob.ts
import cron, { ScheduledTask } from 'node-cron';

export class ExampleJob {
  private task: ScheduledTask;

  constructor() {
    // Define the cron schedule from environment variables or use a
default
    const schedule = process.env.CRON_SCHEDULE || '*/5 * * * *';
    this.task = cron.schedule(schedule, this.run, {
      scheduled: false, // Prevents the job from starting automatically
    });
  }

  // The task to run
  private async run() {
    try {
      console.log(`[${new Date().toISOString()}] ExampleJob is
running.`);
      // Add your task logic here (e.g., database cleanup, sending
reports)
    } catch (error) {
      console.error('Error running ExampleJob:', error);
    }
  }
```

```
  // Start the cron job
  public start() {
    this.task.start();
    console.log('ExampleJob has started.');
  }

  // Stop the cron job
  public stop() {
    this.task.stop();
    console.log('ExampleJob has stopped.');
  }
}
```

3. Set Up the Express Server

Create `src/index.ts` to define the Express server and manage background jobs:

```
// src/index.ts
import express, { Request, Response } from 'express';
import dotenv from 'dotenv';
import { ExampleJob } from './jobs/exampleJob';

dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware to parse JSON
app.use(express.json());

// Initialize background jobs
const exampleJob = new ExampleJob();

// Start all scheduled jobs
exampleJob.start();

// API Endpoint to check server status
app.get('/', (req: Request, res: Response) => {
```

```
  res.send('Background Service is running.');
});

// API Endpoint to manually trigger the ExampleJob
app.post('/trigger-job', async (req: Request, res: Response) => {
  try {
    await exampleJob['run'](); // Accessing the private run method for
demonstration
    res.status(200).json({ message: 'ExampleJob executed successfully.'
});
  } catch (error) {
    res.status(500).json({ message: 'Failed to execute ExampleJob.',
error });
  }
});

// API Endpoint to start the ExampleJob
app.post('/start-job', (req: Request, res: Response) => {
  exampleJob.start();
  res.status(200).json({ message: 'ExampleJob started.' });
});

// API Endpoint to stop the ExampleJob
app.post('/stop-job', (req: Request, res: Response) => {
  exampleJob.stop();
  res.status(200).json({ message: 'ExampleJob stopped.' });
});

// Handle graceful shutdown
const shutdown = () => {
  console.log('Shutting down server...');
  exampleJob.stop();
  process.exit(0);
};

process.on('SIGINT', shutdown);
process.on('SIGTERM', shutdown);
```

```
// Start the Express server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Notes:

- **ExampleJob Class**: Encapsulates the cron job logic, allowing you to start and stop the job as needed.
- **API Endpoints**:
  - `GET /`: Check if the service is running.
  - `POST /trigger-job`: Manually trigger the cron job.
  - `POST /start-job`: Start the cron job.
  - `POST /stop-job`: Stop the cron job.
- **Graceful Shutdown**: Ensures that cron jobs are stopped when the server shuts down.

4. Update package.json Scripts

Add a script to run the server with `ts-node-dev` for hot-reloading:

```
"scripts": {
  "start": "ts-node-dev src/index.ts"
}
```

Running the Application

1. **Start the Server**

   ```
   npm start
   ```

   You should see output similar to:

```
Server is running on port 3000
ExampleJob has started.
```

2. **Verify Scheduled Job Execution**

   Every 5 minutes (as per the default cron schedule), you should see logs like:

   ```
   [2024-04-27T12:00:00.000Z] ExampleJob is running.
   ```

3. **Interact with API Endpoints**

   - **Check Server Status**

     ```
     curl http://localhost:3000/
     ```

     Response:

     ```
     Background Service is running.
     ```

   - **Manually Trigger the Job**

     ```
     curl -X POST http://localhost:3000/trigger-job
     ```

     Response:

     ```
     {
       "message": "ExampleJob executed successfully."
     }
     ```

   - **Start the Job**

     If the job is stopped, you can start it:

     ```
     curl -X POST http://localhost:3000/start-job
     ```

     Response:

```
{
  "message": "ExampleJob started."
}
```

- **Stop the Job**

  To stop the cron job:

  ```
  curl -X POST http://localhost:3000/stop-job
  ```

  Response:

  ```
  {
    "message": "ExampleJob stopped."
  }
  ```

Error Handling and Validation

For production-ready applications, consider enhancing error handling and validating input data. Libraries like Joi (https://joi.dev/) or express-validator (https://express-validator.github.io/docs/) can help with robust validation. Additionally, implement logging mechanisms (e.g., using Winston (https://github.com/winstonjs/winston)) for better monitoring of background tasks.

Scaling Background Services

While `node-cron` is suitable for simple scheduling needs, more complex applications might require advanced job queues or background processing systems. Consider the following alternatives for scalability and reliability:

- **Bull (https://github.com/OptimalBits/bull)**: A Redis-based queue system for handling distributed jobs.
- **Agenda (https://github.com/agenda/agenda)**: A light-weight job scheduling library for Node.js backed by MongoDB.
- **Bee-Queue (https://github.com/bee-queue/bee-queue)**: A simple, fast, robust job queue for Node.js backed by Redis.

These tools offer features like job retries, concurrency control, and persistent storage, which are essential for large-scale applications.

EJS Templating Engine

Using EJS Templating Engine in Express.js with TypeScript

This guide provides a comprehensive introduction to using the EJS (Embedded JavaScript) templating engine within an Express.js application written in TypeScript. We'll explore the basic syntax of EJS, demonstrate its use in email templating, and cover other common applications such as rendering dynamic web pages. By the end of this tutorial, you'll be equipped to integrate EJS into your Express projects effectively.

Table of Contents

- 8. Update `package.json` Scripts
- 9. Running the Application

4. Conclusion

5. Additional Resources

Introduction to Templating Engines and EJS

What is a Templating Engine?

A templating engine allows you to generate dynamic HTML (or other text-based formats) by embedding code within templates. This enables the creation of dynamic content based on data, user input, or other variables. Templating engines are essential for rendering views in web applications, generating emails, and more.

Why EJS?

EJS (https://ejs.co/) is a simple and flexible templating engine for Node.js. It uses plain JavaScript syntax within templates, making it easy to learn and integrate, especially if you're already familiar with JavaScript. EJS is versatile and can be used for rendering web pages, generating emails, and other text-based outputs.

Key Features of EJS:

- **Simplicity:** Easy to learn with straightforward syntax.
- **Flexibility:** Can be used for various purposes beyond web pages, such as email templates.
- **Integration:** Seamlessly integrates with Express.js and TypeScript.
- **Extensibility:** Supports partials and includes for reusable components.

Basic Syntax of EJS

Understanding the basic syntax of EJS is crucial before integrating it into your projects. EJS templates are essentially HTML files with embedded JavaScript code.

Variables

To embed variables within your template, use `<%= %>` for escaping and `<%- %>` for unescaped content.

```
<!-- Escaped Output -->
<p>Hello, <%= username %>!</p>

<!-- Unescaped Output (Use with caution) -->
<p>Content: <%- content %></p>
```

Control Flow

EJS supports standard JavaScript control flow statements like `if`, `for`, `each`, etc.

```
<!-- If Statement -->
<% if (user.isAdmin) { %>
  <p>Welcome, admin!</p>
<% } else { %>
  <p>Welcome, user!</p>
<% } %>

<!-- For Loop -->
<ul>
  <% for(let i = 0; i < items.length; i++) { %>
    <li><%= items[i] %></li>
  <% } %>
</ul>

<!-- ForEach Loop -->
<ul>
  <% items.forEach(function(item) { %>
    <li><%= item %></li>
  <% }); %>
</ul>
```

Includes

EJS allows you to include other EJS files within a template, promoting reusability.

```
<!-- Including a Header Partial -->
<%- include('partials/header') %>

<!-- Including a Footer Partial -->
<%- include('partials/footer') %>
```

Comments

Comments in EJS templates are not rendered in the output.

```
<%# This is a comment and won't be rendered %>
```

Hands-On Implementation

Let's implement EJS in an Express.js application using TypeScript. We'll create an API endpoint that sends templated emails using EJS.

Prerequisites

- **Node.js** (v14 or later)
- **npm** or **yarn**
- Basic knowledge of **TypeScript** and **Express.js**

1. Initialize the Project

Create a new directory for your project and initialize it with npm:

```
mkdir express-ejs-email-ts
cd express-ejs-email-ts
npm init -y
```

2. Install Dependencies

Install the necessary packages:

```
npm install express nodemailer ejs dotenv
```

```
npm install --save-dev typescript @types/express @types/node ts-node-dev
```

Dependencies:

- **express**: Web framework for Node.js
- **nodemailer**: Module to send emails
- **ejs**: Templating engine
- **dotenv**: Loads environment variables from a `.env` file

Dev Dependencies:

- **typescript**, **ts-node-dev**: For TypeScript support
- **@types/***: Type definitions for TypeScript

3. Configure TypeScript

Initialize a TypeScript configuration:

```
npx tsc --init
```

Modify the `tsconfig.json` as needed. A basic configuration might look like this:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  }
}
```

4. Set Up Project Structure

Create the following structure:

```
express-ejs-email-ts/
├── src/
│   ├── templates/
│   │   └── welcomeEmail.ejs
│   ├── index.ts
│   └── email.ts
├── .env
├── package.json
└── tsconfig.json
```

5. Create EJS Templates

Create an EJS template for the email. For example, `src/templates/welcomeEmail.ejs`:

```ejs
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Welcome Email</title>
</head>
<body>
  <h1>Welcome, <%= username %>!</h1>
  <p>Thank you for signing up. We're excited to have you on board.</p>

  <% if (isAdmin) { %>
    <p>You have administrative privileges.</p>
  <% } else { %>
    <p>You are a regular user.</p>
  <% } %>

  <h2>Your Interests:</h2>
  <ul>
    <% interests.forEach(function(interest) { %>
      <li><%= interest %></li>
    <% }); %>
  </ul>
```

```
  <p>Best regards,<br/>The Team</p>
</body>
</html>
```

Explanation:

- **Variables:** `<%= username %>` injects the `username` variable.
- **Control Flow:** The `if` statement displays different content based on the `isAdmin` flag.
- **Loop:** Iterates over the `interests` array to list user interests.

6. Configure Express to Use EJS

While EJS is primarily used for rendering HTML pages, we'll leverage it for email templating by rendering EJS templates to HTML strings.

Create `src/email.ts` to handle email sending logic with EJS templating:

```
// src/email.ts
import nodemailer from 'nodemailer';
import dotenv from 'dotenv';
import ejs from 'ejs';
import path from 'path';

dotenv.config();

interface EmailOptions {
  to: string;
  subject: string;
  template: string; // Path to the EJS template
  data: any;        // Data to inject into the template
}

export const sendTemplatedEmail = async (options: EmailOptions) => {
  // Create a transporter
  const transporter = nodemailer.createTransport({
    service: process.env.EMAIL_SERVICE, // e.g., 'Gmail'
```

```
    auth: {
      user: process.env.EMAIL_USER,
      pass: process.env.EMAIL_PASS,
    },
  });

  // Resolve the template path
  const templatePath = path.join(__dirname, 'templates',
options.template);

  try {
    // Render the EJS template to HTML
    const html = await ejs.renderFile(templatePath, options.data);

    // Define mail options
    const mailOptions = {
      from: process.env.EMAIL_USER,
      to: options.to,
      subject: options.subject,
      html: html,
    };

    // Send the email
    const info = await transporter.sendMail(mailOptions);
    console.log('Email sent:', info.response);
    return info;
  } catch (error) {
    console.error('Error sending email:', error);
    throw error;
  }
};
```

Explanation:

- **Nodemailer Transporter:** Configured using environment variables for service, user, and password.

- **EJS Rendering:** `ejs.renderFile` is used to render the specified EJS template with provided data.
- **Mail Options:** Defines the sender, recipient, subject, and HTML content.
- **Sending Email:** Uses `transporter.sendMail` to send the email.

7. Implement Email Sending with EJS Templates

Create `src/index.ts` to define the Express server and API endpoints for sending templated emails:

```
// src/index.ts
import express, { Request, Response } from 'express';
import dotenv from 'dotenv';
import { sendTemplatedEmail } from './email';

dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware to parse JSON
app.use(express.json());

// API Endpoint to send a welcome email
app.post('/send-welcome-email', async (req: Request, res: Response) => {
  const { to, username, isAdmin, interests } = req.body;

  if (!to || !username || !isAdmin || !interests) {
    return res.status(400).json({ message: 'Missing required fields: to,
username, isAdmin, interests' });
  }

  try {
    await sendTemplatedEmail({
      to,
      subject: 'Welcome to Our Service!',
      template: 'welcomeEmail.ejs',
```

```
      data: { username, isAdmin, interests },
    });
    res.status(200).json({ message: 'Welcome email sent successfully.'
});
  } catch (error) {
    res.status(500).json({ message: 'Failed to send welcome email.',
error });
  }
});

// API Endpoint to check server status
app.get('/', (req: Request, res: Response) => {
  res.send('EJS Email Service is running.');
});

// Handle graceful shutdown
const shutdown = () => {
  console.log('Shutting down server...');
  process.exit(0);
};

process.on('SIGINT', shutdown);
process.on('SIGTERM', shutdown);

// Start the Express server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Explanation:

- **POST `/send-welcome-email`:** Accepts JSON data to send a welcome email using the `welcomeEmail.ejs` template.
- **GET `/`:** Simple endpoint to verify that the server is running.
- **Graceful Shutdown:** Ensures the server exits cleanly on termination signals.

8. Update package.json Scripts

Add a script to run the server with `ts-node-dev` for hot-reloading:

```
"scripts": {
  "start": "ts-node-dev src/index.ts"
}
```

9. Running the Application

1. Configure Environment Variables

Create a `.env` file in the root directory to store sensitive information and configuration variables:

```
PORT=3000
EMAIL_SERVICE=Gmail
EMAIL_USER=your-email@gmail.com
EMAIL_PASS=your-email-password
```

Important:

- Replace `your-email@gmail.com` and `your-email-password` with your actual email credentials.
- For Gmail, you might need to enable Less Secure Apps (https://myaccount.google.com/lesssecureapps) or use App Passwords (https://support.google.com/accounts/answer/185833).

2. Start the Server

```
npm start
```

You should see output similar to:

```
Server is running on port 3000
```

3. Send a Test Welcome Email

Use a tool like Postman (https://www.postman.com/) or `curl` to send a POST request to `http://localhost:3000/send-welcome-email` with a JSON body:

```
{
  "to": "recipient@example.com",
  "username": "JohnDoe",
  "isAdmin": true,
  "interests": ["Coding", "Reading", "Gaming"]
}
```

Using `curl`:

```
curl -X POST http://localhost:3000/send-welcome-email \
  -H "Content-Type: application/json" \
  -d '{
    "to": "recipient@example.com",
    "username": "JohnDoe",
    "isAdmin": true,
    "interests": ["Coding", "Reading", "Gaming"]
  }'
```

Expected Response:

```
{
  "message": "Welcome email sent successfully."
}
```

Check the recipient's inbox to verify that the email was received with the rendered EJS template.