DESENVOLVIMENTO DE APIS COM EXPRESS.JS

Sumário:

1. Introdução ao Express.js

O **Express.js** é um framework minimalista e flexível para construir aplicações web e APIs em Node.js. Desenvolvido com o intuito de simplificar o processo de criação de servidores e facilitar o desenvolvimento de aplicativos escaláveis, ele se tornou uma das ferramentas mais populares para desenvolvedores de JavaScript no lado do servidor. Vamos explorar os fundamentos do Express.js, seu histórico, vantagens e desvantagens.

1.1 Fundamentos do Express.js

O Node.js é extremamente poderoso para construir aplicações de rede e servidores de alto desempenho. No entanto, algumas funcionalidades comuns no desenvolvimento web não são oferecidas diretamente por ele. Por exemplo, gerenciar diferentes métodos HTTP (como GET, POST, DELETE, PUT), lidar com várias URLs (rotas) ou gerar respostas dinâmicas com templates pode se tornar trabalhoso usando apenas o Node.js. Para resolver essas limitações e facilitar o desenvolvimento, é comum utilizar **web frameworks** que ajudam a estruturar e organizar o código.

Um dos frameworks mais populares para Node.js é o Express.js. O Express.js atua como uma camada sobre o Node.js, fornecendo uma interface simples e poderosa para criar aplicações web e APIs de maneira organizada e modular. O Express simplifica o processo de desenvolvimento com soluções que permitem:

- **Gerenciar múltiplos métodos HTTP em diferentes rotas:** O Express facilita a definição de rotas específicas para cada método HTTP, organizando os pontos de entrada da aplicação para diferentes tipos de requisição.
- **Integrar view engines para gerar páginas dinâmicas:** Embora o Express seja mais frequentemente usado para APIs RESTful (que geralmente retornam JSON), ele também suporta view engines (motores de renderização de templates) para gerar HTML dinâmico.
- **Configurações globais da aplicação:** No Express, você pode definir configurações como a porta de conexão e o diretório onde os templates HTML (caso use) estão localizados, padronizando a estrutura da aplicação.
- **Uso de Middlewares:** Middlewares são funções que processam requisições antes que a resposta final seja enviada ao cliente. Eles são úteis para tarefas como autenticação, validação de dados, tratamento de erros e logging.

1.2 Histórico do Express.js

O Express.js foi criado em 2010 por TJ Holowaychuk com o objetivo de fornecer uma maneira mais rápida e conveniente de desenvolver servidores em Node.js. O Express surgiu em um momento em que o Node.js ainda estava se estabelecendo como uma tecnologia popular para desenvolvimento de back-end, e o framework ajudou a acelerar a adoção do Node.js, especialmente entre desenvolvedores que já conheciam o JavaScript.

Desde seu lançamento, o Express se tornou parte fundamental do **MEAN stack** (MongoDB, Express, Angular, Node.js) e do **MERN stack** (MongoDB, Express, React, Node.js), frameworks amplamente usados para o desenvolvimento de aplicações full-stack em JavaScript. Em 2014, o projeto foi transferido para a **OpenJS Foundation** e continua a ser mantido ativamente, recebendo atualizações e melhorias da comunidade.

1.3 Principais Características do Express.js

1.3.1 Minimalista

O Express é um framework **minimalista**, ou seja, ele fornece apenas funcionalidades essenciais e deixa outras responsabilidades ao desenvolvedor. Essa simplicidade permite que os desenvolvedores implementem apenas as ferramentas de que realmente precisam, adicionando pacotes de middleware conforme necessário. O minimalismo do Express é uma vantagem em projetos que exigem flexibilidade e baixo consumo de recursos.

O código abaixo, por exemplo, cria um servidor que responde a requisições GET na rota raiz (/). Observe que o Express fornece apenas o necessário para criar o servidor e definir a rota.

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Bem-vindo ao Express!");
});

app.listen(3000, () => {
  console.log("Servidor rodando em http://localhost:3000");
});
```

1.3.2 Modularidade com Middlewares

Middlewares são uma das funcionalidades mais poderosas do Express. Eles são funções que executam antes, durante ou após o processamento de uma requisição, e podem ser usados para manipular a requisição, a resposta ou controlar o fluxo de execução. O Express permite que middlewares sejam adicionados de forma modular, permitindo que a lógica da aplicação seja dividida em etapas e simplificando a manutenção.

Veja abaixo alguns tipos de Middlewares no Express:

- **Middleware de Aplicação:** Executado para todas as rotas e métodos HTTP, útil para configurar autenticação, logs e validação de dados.
- **Middleware de Rota:** Executado em rotas específicas para adicionar lógica antes de uma resposta ser enviada.
- **Middleware Integrado:** O Express vem com middlewares embutidos, como **express.json()** para processar JSON enviados em requisições. Ela é usada para converter automaticamente o JSON recebido em um objeto JavaScript acessível na rota.
- **Middleware de Tratamento de Erros:** Manipula erros que ocorrem na aplicação e pode enviar mensagens de erro específicas para o cliente.

Middlewares permitem adicionar funcionalidades incrementais, como autenticação e validação, de forma modular e reutilizável. Veja no quadro abaixo um exemplo de uso de Middleware:

```
// Middleware para registrar o método, a URL e o horário de cada requisição
app.use((req, res, next) => {
  const now = new Date().toISOString();
  console.log(`[${now}] Requisição recebida: ${req.method} ${req.url}`);
  next();
});
```

Este código é um middleware que registra no console todas as requisições feitas ao servidor. Ele mostra o método HTTP (como GET ou POST) e a URL da requisição. Após registrar a informação, ele chama **next()** para que a execução continue para o próximo middleware ou rota, garantindo que o fluxo do aplicativo não seja interrompido.

1.3.3 Flexibilidade de Roteamento

O Express fornece um sistema de **roteamento** poderoso e flexível, que permite associar URLs específicas a funções de tratamento. É possível definir rotas para diferentes métodos HTTP (como GET, POST, PUT e DELETE) e para URLs dinâmicas que aceitam parâmetros.

Sobre o roteamento dinâmico e hierárquico:

- **Parâmetros de Rota:** O Express permite capturar partes variáveis da URL, como IDs de recursos, e passar essas informações para o manipulador de rota.
- **Roteamento Hierárquico:** As rotas podem ser organizadas em um roteador modular, facilitando a organização do código em aplicações maiores.

Vejamos abaixo um exemplo de roteamento com parâmetros de rota:

```
app.get("/produtos/:id", (req, res) => {
  const { id } = req.params;
  res.send(`Produto ID: ${id}`);
});
```

Esse exemplo define uma rota GET no Express.js para acessar um produto específico com base em seu ID. A rota "**/produtos/:id**" usa um parâmetro de rota (**:id**), que permite que qualquer valor inserido no lugar de **:id** na URL seja capturado. Quando um cliente acessa a URL, como "/produtos/123", o valor 123 será armazenado em **req.params.id**. No corpo da função, ele extrai o **id** da requisição (**req.params**) e envia uma resposta (r**es.send**) que exibe o ID do produto solicitado. Essa implementação permite que o servidor responda dinamicamente a diferentes IDs de produto, sem a necessidade de definir uma rota específica para cada um.

1.3.4 Não-Opinativo

O Express é um framework **não-opinativo**, o que significa que ele não impõe uma maneira específica de estruturar seu projeto ou de organizar as rotas e middlewares. Diferente de frameworks opinativos, que possuem "opiniões" sobre como resolver problemas específicos, o Express oferece liberdade total ao desenvolvedor para estruturar o projeto da maneira que preferir.

Veja abaixo uma descrição comparativa entre os frameworks opinativos e os não-opinativos:

- **Frameworks opinativos:** oferecem uma estrutura rígida e padrões definidos para organizar o projeto. Isso pode ser útil para iniciantes, pois traz uma organização padronizada, mas limita a flexibilidade.
- **Frameworks não-opinativos:** permitem que o desenvolvedor escolha a estrutura e os componentes do projeto. Você pode usar os middlewares e bibliotecas que desejar, definir sua própria estrutura de pastas e arquivos, e organizar o projeto de acordo com as necessidades específicas.

Neste contexto, veja na imagem abaixo um exemplo de estrutura de projeto personalizada em Node.js usando o Express:

```
projeto/
├── server.js           # Arquivo principal que inicia o servidor
├── routes/             # Pasta para armazenar as rotas
├── controllers/        # Pasta para lógica das rotas
├── middlewares/        # Pasta para middlewares personalizados
└── models/             # Pasta para modelos de dados, caso haja um banco de dados
```

Figura 1: Exemplo de estrutura de projeto personalizada para o Node.js usando o Express

Descrição: A imagem mostra a estrutura de um projeto em Node.js organizada em pastas e arquivos. Na raiz do projeto, há um arquivo chamado server.js, descrito como o arquivo principal que inicia o servidor. Existem quatro pastas: routes/, que armazena as rotas do aplicativo; controllers/, que contém a lógica das rotas; middlewares/, destinada a middlewares personalizados; e models/, que armazena modelos de dados, caso o projeto utilize um banco de dados.

Cabe salientar que essa estrutura não é imposta pelo Express, mas é uma prática comum para organizar projetos de forma modular e escalável. Entretanto, o Express.js oferece em seu site oficial o **Express Application Generator**, ferramenta que permite criar rapidamente a estrutura de um aplicativo Express básico. Ele gera uma estrutura padrão de diretórios e arquivos para ajudar os desenvolvedores a começar um projeto rapidamente, promovendo boas práticas e organizando o código de maneira modular e escalável.

1.3.5 Suporte a Templates (View Engines)

Embora o Express seja amplamente usado para criar APIs RESTful que retornam JSON, ele também suporta **view engines** (ou mecanismo de visualização) que permitem gerar HTML

dinâmico. No Node.js, uma view engine é uma ferramenta que permite a renderização de templates dinâmicos em aplicativos web. Ela gera HTML com base nos dados que o servidor fornece, permitindo a criação de páginas dinâmicas que podem se adaptar a diferentes dados ou condições. No contexto de frameworks como o Express.js, as view engines são usadas para renderizar conteúdo HTML antes de enviá-lo ao cliente (navegador).

Veja abaixo as principais view engines em projetos com Node.js:

- **Pug:** Motor de templates com uma sintaxe concisa, popular no Express.
- **EJS:** Motor de templates que permite inserir código JavaScript diretamente no HTML.
- **Handlebars:** Uma opção com suporte a templates mais sofisticados, útil para renderizar layouts complexos.

Veja abaixo um exemplo de uso do EJS em projetos com Node.js:

```
// Configurando EJS como view engine
app.set('view engine', 'ejs');

// Rota para renderizar uma página HTML
app.get('/home', (req, res) => {
  res.render('home', { title: 'Página Inicial', mensagem: 'Bem-vindo ao
Express!' });
});
```

No exemplo acima, a rota **/home** renderiza uma página HTML usando o motor de templates EJS.

1.4 Vantagens do Express.js

O Express.js é amplamente adotado por várias razões. Aqui estão algumas das principais vantagens do framework:

- **Simplicidade e Flexibilidade:** A simplicidade do Express permite que você comece rapidamente e adicione apenas as funcionalidades necessárias. Sua flexibilidade o torna ideal para uma grande variedade de projetos, desde APIs RESTful até aplicações full-stack.

- **Enorme Comunidade e Ecossistema:** Express tem uma vasta comunidade, com muitas bibliotecas e pacotes que facilitam a expansão de funcionalidades da aplicação.
- **Modularidade com Middlewares:** A capacidade de adicionar middlewares de forma modular torna o Express altamente extensível. Existem middlewares para quase tudo, como autenticação, validação, tratamento de erros, compressão e muito mais.

1.5 Desvantagens do Express.js

Embora o Express.js seja amplamente utilizado e ofereça diversas vantagens, ele também apresenta algumas desvantagens que devem ser consideradas:

- **Falta de Estrutura Padrão:** Como o Express não impõe uma estrutura rígida, ele pode se tornar difícil de organizar em projetos grandes. Isso exige que o desenvolvedor adote boas práticas para manter o código organizado.
- **Configuração Manual:** Algumas funcionalidades, como autenticação ou integração com banco de dados, precisam ser configuradas manualmente, o que pode aumentar a complexidade em projetos maiores.

2. Criando um Servidor HTTP com Express.js

Criar um servidor HTTP com Express.js é simples. Primeiro, crie e acesse uma pasta para o projeto, onde serão organizados todos os arquivos e configurações.

```
mkdir node-express
cd node-express
```

Em seguida, inicie o projeto criando o arquivo package.json, que será responsável por gerenciar as dependências e configurações do projeto.

```
npm init -y
```

Com o projeto inicializado, instale o **Express.js**:

```
npm i express
```

Instale o **Nodemon** no projeto para que ele monitore mudanças nos arquivos e reinicie o servidor automaticamente ao salvar. No terminal, digite o comando de instalação:

```
npm i nodemon --save-dev
```

Você pode, agora, **abrir a aplicação no editor Visual Studio Code**.

```
code .
```

Abra o arquivo **package.json** e, na seção "scripts", adicione o script "dev" para iniciar o servidor com o Nodemon.

```
...
"scripts": {
   "test": "echo \"Error: no test specified\" && exit 1",
   "start": "node server.js",
   "dev": "nodemon server.js"
 },
...
```

Para executar o servidor usando o Nodemon, digite o seguinte comando:

```
npm run dev
```

3. Criação de Rotas e Middlewares

Nesta seção, vamos explorar o roteamento no Express.js com mais profundidade, incluindo como definir rotas e manipuladores de requisições e como utilizar middlewares para processar requisições e respostas. Esses conceitos são essenciais para construir aplicações web organizadas e eficientes no Express.

Roteamento, em Node.js, refere-se ao processo de definir como uma aplicação web responde às solicitações de clientes em endpoints específicos. Um endpoint é uma combinação de uma URL (ou caminho) e um método HTTP (como GET, POST, PUT ou DELETE) que identifica a operação solicitada. Basicamente, o roteamento estabelece um conjunto de regras e caminhos que guiam a aplicação a realizar ações específicas quando o usuário acessa determinadas URLs.

No framework Express.js, o roteamento é configurado definindo **rotas** que correspondem a esses endpoints da aplicação. Cada rota pode:

- Especificar um caminho, como **/produtos** ou **/usuarios**, para identificar o recurso.
- Usar métodos HTTP, como **GET**, **POST**, **PUT** ou **DELETE**, para determinar o tipo de operação a ser realizada.
- Executar uma **função** ou **middleware** que define o comportamento da aplicação ao acessar esse endpoint específico, controlando, por exemplo, quais dados devem ser exibidos, criados, atualizados ou excluídos. Cabe lembrar que **middlewares** são funções intermediárias que processam dados antes ou depois de uma requisição ser atendida, adicionando funcionalidades ou verificações.

Esse sistema de roteamento é fundamental para organizar a lógica da aplicação, especialmente em APIs RESTful, pois permite responder de maneira consistente e modular a diferentes solicitações dos usuários.

3.1 Definindo Rotas e Manipuladores de Requisições

No Express.js, cada rota é composta por um **URL** (caminho) e um **método HTTP** (como GET, POST, PUT ou DELETE), além de um **manipulador de requisição** (ou handler) que lida com a requisição e a resposta. Neste contexto, a estrutura básica de uma rota no Express é a seguinte:

```
app.METODO_HTTP(caminho, (req, res) => {
  // Função de tratamento da rota
});
```

Onde:

- **METODO_HTTP:** Um método HTTP específico como GET, POST, PUT e DELETE.
- **caminho:** A URI que o cliente acessa para chegar a essa rota (ex.: "/produtos", "/usuarios/:id").
- **(req, res) => { ... }:** A função que define a lógica da resposta para essa rota.

Para ilustrar, no exemplo abaixo, definimos uma rota simples no Express.js para responder a uma requisição HTTP do tipo GET no endpoint "/" (a página inicial da aplicação). Desta forma, crie um arquivo chamado **server.js** com o seguinte código:

```
// Importa o módulo Express
const express = require("express");

// Cria uma instância do aplicativo Express
const app = express();

// Define a porta em que o servidor irá rodar
const PORT = 3000;

// Define uma rota GET para o caminho raiz ("/")
app.get("/", (req, res) => {
  // Envia uma resposta em JSON para o cliente
  res.json({ message: "Olá, Mundo!" });
});

// Inicia o servidor e faz com que ele escute na porta definida
app.listen(PORT, () => {
  // Exibe uma mensagem no console informando que o servidor está em execução
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Neste curso, utilizamos a extensão REST Client do VS Code para testar os endpoints da aplicação. Neste contexto, crie uma pasta nome “**request**” em seu projeto .Em seguida, dentro desta pasta, crie um arquivo com o nome de sua preferência com a extensão “.rest”, por exemplo, **testes.rest**.

No arquivo testes.rest, insira a seguinte requisição requisição HTTP para acessar a rota raiz (/) do servidor:

```
GET http://localhost:3000/
```

Após escrever a requisição, um link “Send Request” aparecerá acima dela. Clique nesse link para enviar a requisição. O REST Client enviará a requisição GET para o servidor Node.js no endereço http://localhost:3000/. A resposta do servidor será exibida no painel de saída do REST Client, mostrando o código de status HTTP e o corpo da resposta em JSON.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 26
ETag: W/"1a-hFwWj6/Mus85PVt7ROvapHkqfGA"
Date: Wed, 06 Nov 2024 08:34:16 GMT
Connection: close

{

  "message": "Olá, Mundo!"
}
```

3.1.1 Métodos de Roteamento

No Express.js, métodos de roteamento se referem aos métodos HTTP que o Express disponibiliza para lidar com requisições. Cada método define como o servidor deve reagir a diferentes tipos de requisições de clientes. Os métodos de roteamento mais comuns no Express são GET, POST, PUT, DELETE, entre outros.

O código a seguir é um exemplo de rotas o caminho “/produtos” e que estão definidas para os métodos GET, POST, PUT e DELETE. Atualize o arquivo server.js com este código:

```
const express = require("express");
const app = express();
const PORT = 3000;
// Rota GET para listar todos os produtos
app.get("/produtos", (req, res) => {
  res.json({ message: "Listando todos os produtos", produtos: [] }); // Exemplo
com um array vazio de produtos
});
```

```
// Rota POST para criar um novo produto
app.post("/produtos", (req, res) => {
  res.json({
    message: "Criando um novo produto",
    produto: { id: Date.now(), nome: "Produto Exemplo" },
  });
});
// Rota PUT para atualizar um produto por ID
app.put("/produtos/:id", (req, res) => {
  const { id } = req.params;
  res.json({
    message: "Atualizando o produto",
    produto: { id: id, nome: "Produto Atualizado" },
  });
});
// Rota DELETE para excluir um produto por ID
app.delete("/produtos/:id", (req, res) => {
  const { id } = req.params;
  res.json({ message: "Excluindo o produto", produtoId: id });
});
// Inicia o servidor e faz com que ele escute na porta definida
app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Neste exemplo:

- GET /produtos: Rota para listar produtos.
- POST /produtos: Rota para criar um novo produto.
- PUT /produtos/:id: Rota para atualizar um produto pelo ID.
- DELETE /produtos/:id: Rota para excluir um produto pelo ID.

Ao usar o REST Client, é possível declarar variáveis para definir valores reutilizáveis, como o endpoint da API, facilitando a manutenção e a consistência do script. Para isso, basta declarar a variável em uma nova linha com a seguinte sintaxe: **@variableName = valor**. As requisições dentro do arquivo podem então referenciar essa variável usando a notação **{{variableName}}**. Isso permite que, se o valor do endpoint precisar ser alterado, seja suficiente modificá-lo apenas na declaração da variável, atualizando automaticamente todas as requisições que a utilizam.

Neste contexto, você pode alterar o arquivo testes.rest da seguinte forma:

```
@baseURL = http://localhost:3000

### Teste da rota GET /produtos
GET {{baseURL}}/produtos

### Teste da rota POST /produtos
POST {{baseURL}}/produtos
Content-Type: application/json

{
  "nome": "Novo Produto",
  "preco": 100.0
}

### Teste da rota PUT /produtos/:id
PUT {{baseURL}}/produtos/1
Content-Type: application/json

{
  "nome": "Produto Atualizado",
  "preco": 150.0
}

### Teste da rota DELETE /produtos/:id
DELETE {{baseURL}}/produtos/1
```

Após escrever cada requisição, um link “Send Request” aparecerá acima delas. Clique nesse link para enviar cada requisição. Por exemplo, veja abaixo o resultado da requisição POST:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 93
ETag: W/"5d-OhYGsOVj/OqVgL0PCjlccB2767g"
Date: Wed, 06 Nov 2024 08:42:52 GMT
Connection: close

{

  "message": "Criando um novo produto",
  "produto": {

    "id": 1730882572049,
    "nome": "Produto Exemplo"
  }
}
```

3.1.2 Tipos de Caminhos de Rotas

O Express permite definir caminhos de rotas de diversas maneiras para lidar com diferentes padrões de URL. Os caminhos de uma rota podem conter uma sequência de caracteres, padrões de sequência ou expressões regulares.

OBS: O Express Route Tester é uma ferramenta online que permite testar e visualizar rotas para aplicativos Express.js sem precisar configurar um servidor completo.

3.1.2.1 Caminhos Simples

Um caminho simples representa uma URL exata que o servidor deve corresponder:

```
app.get("/sobre", (req, res) => {
  res.send("Página Sobre");
});
```

Aqui, uma requisição GET para "/sobre" dispara a função de tratamento que responde com "Página Sobre".

No exemplo abaixo, o caminho corresponde a solicitações à rota raiz **/:**

```
app.get("/", (req, res) => {
  res.send("raiz");
});
```

3.1.2.2 Padrões de Sequência

Abaixo mostra um exemplo de caminho de rota que corresponde ao padrão de sequência **acd** e **abcd:**

```
app.get("/ab?cd", (req, res) => {
  res.send("ab?cd");
});
```

Este caminho de rota irá corresponder ao **abcd**, **abxcd**, **abRANDOMcd**, **ab123cd**, etc.

```
app.get("/ab*cd", (req, res) => {
  res.send("ab*cd");
});
```

3.1.2.3 Expressões Regulares

O Express também permite o uso de expressões regulares para criar rotas mais complexas, restringindo padrões válidos de URL.

```
app.get("/item/:codigo([0-9]{5})", (req, res) => {
  res.send(`Código do item: ${req.params.codigo}`);
});
```

Aqui, **:codigo([0-9]{5})** permite apenas números de 5 dígitos. Se alguém tentar acessar **/item/abcde**, o Express não encontrará uma correspondência. Mas se acessar **/item/12345**, a resposta será **Código do item: 12345**.

3.1.2.4 Caminhos Dinâmicos com Parâmetros de Rota

Parâmetros de rota permitem capturar partes da URL como variáveis, tal como **:id**:

```
app.get("/produtos/:id", (req, res) => {
  const id = req.params.id;
  res.send(`Produto ID: ${id}`);
});
```

Neste caso:

- "**:id**" é um parâmetro de rota que captura o valor que vier depois de **/produtos/**.
- O valor de id estará disponível em **req.params.id**.

Se o cliente acessar **/produtos/123**, o servidor responderá com **Produto ID: 123**.

3.1.2.5 Caminhos com vários Parâmetros de Rota

O Express permite usar múltiplos parâmetros de rota na mesma URL:

```
app.get("/usuarios/:usuarioId/posts/:postId", (req, res) => {
  const { usuarioId, postId } = req.params;
  res.send(`Usuário: ${usuarioId}, Post: ${postId}`);
});
```

Acessando **/usuarios/45/posts/67**, a resposta será **Usuário: 45, Post: 67**.

3.1.2.6 Caminhos com Roteamento Opcional

O Express permite tornar parte do caminho opcional usando um **?** após o nome do parâmetro:

```
app.get("/produtos/:id?", (req, res) => {
  if (req.params.id) {
    res.send(`Produto ID: ${req.params.id}`);
  } else {
    res.send("Lista de produtos");
  }
});
```

Neste exemplo:

- Se **id** for fornecido, a resposta será Produto ID: valor.
- Se **id** não for fornecido (acesso apenas em /produtos), a resposta será Lista de produtos.

3.1.3 Manipuladores de Rota

No Express, os **manipuladores de rota** (ou **route handlers**) são funções de callback que definem o que o servidor deve fazer quando uma rota específica é acessada. Eles lidam com a requisição e a resposta, podendo executar várias ações, como acessar um banco de dados, validar dados ou enviar uma resposta ao cliente.

Um manipulador de rota, neste contexto, é uma função que define como a aplicação deve responder a uma requisição feita para uma URL específica e um método HTTP específico (como GET, POST, PUT, DELETE). A função recebe três parâmetros principais:

- **req (Request):** Representa a requisição feita pelo cliente. Contém informações como parâmetros, query strings, cabeçalhos e o corpo da requisição.
- **res (Response):** Representa a resposta que será enviada ao cliente. É usado para definir o conteúdo da resposta, o status HTTP e outros cabeçalhos.
- **next (opcional):** Uma função que passa o controle para o próximo manipulador de rota ou middleware na sequência.

Desta forma, uma rota pode ser manipulada com uma única função, que lida com toda a lógica de resposta, como no quadro abaixo:

```
app.get("/home", (req, res) => {
  res.send("Bem-vindo à página inicial");
});
```

Assim, os manipuladores de rota lidam diretamente com o processamento final da requisição, muitas vezes enviando uma resposta ao cliente.

3.2 Middlewares

No Express.js, um **middleware** é uma função que intercepta as requisições e respostas, podendo processar, modificar ou validar dados antes que a resposta final seja enviada ao cliente. Middleware é uma das principais funcionalidades do Express, permitindo que você adicione diversas camadas de processamento às requisições.

Cada middleware tem acesso aos objetos de requisição (**req**), resposta (**res**), e à função **next**. A função **next()** permite passar o controle para o próximo middleware na sequência. Isso permite criar uma cadeia de funções que podem executar ações, como autenticar o usuário, verificar permissões, registrar logs, entre outras.

Vimos que no Express.js tanto manipuladores de rota quanto middlewares são funções que lidam com as requisições e respostas do servidor, podendo ser usados juntos. Considere um middleware que verifica se o usuário está autenticado antes de acessar qualquer rota:

```
// Middleware para verificar autenticação
const checkAuth = (req, res, next) => {
  const autenticado = true; // Altere para `false` para simular um usuário não autenticado

  if (autenticado) {
    next(); // Usuário autenticado, continua para o próximo middleware ou rota
  } else {
    res.status(401).send("Acesso negado"); // Usuário não autenticado, responde com erro 401
  }
};
```

Esse middleware chama **next()** se o usuário estiver autenticado, permitindo que a requisição prossiga para o próximo middleware ou manipulador de rota. Caso contrário, ele encerra a requisição com uma mensagem de "Acesso negado".

Imagine que queremos aplicar o middleware de autenticação apenas à rota **/dashboard**. O middleware **checkAuth** será aplicado antes do **manipulador de rota**, garantindo que somente usuários autenticados possam acessar o conteúdo do painel:

```
// Rota com middleware específico
app.get("/dashboard", checkAuth, (req, res) => {
  res.send("Bem-vindo ao painel");
});
```

Neste exemplo:

- Quando a rota **/dashboard** é acessada, o Express executa o middleware **checkAuth**.
- Se **checkAuth** chama **next()**, o Express então executa o manipulador de rota que responde com "Bem-vindo ao painel".

Veja como pode ficar seu arquivo **server.js** para testar estas funcionalidades:

```
const express = require("express");
const app = express();
const PORT = 3000;
```

```
// Middleware para verificar autenticação
const checkAuth = (req, res, next) => {
  const autenticado = true; // Altere para `false` para simular um usuário não
autenticado
  if (autenticado) {
    next(); // Usuário autenticado, continua para o próximo middleware ou rota
  } else {
    res.status(401).json({ message: "Acesso negado" }); // Usuário não
autenticado, responde com erro 401 em JSON
  }
};
// Rota com middleware específico
app.get("/dashboard", checkAuth, (req, res) => {
  res.json({ message: "Bem-vindo ao painel" }); // Responde com JSON
});
// Inicia o servidor e faz com que ele escute na porta definida
app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Para testar cada rota usando o REST Client no Visual Studio Code, você pode alterar o arquivo testes.rest da seguinte forma:

```
### Teste da rota GET /dashboard
GET http://localhost:3000/dashboard
```

Após escrever a requisição, um link “Send Request” aparecerá acima dela. Clique nesse link para enviar a requisição.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 33
ETag: W/"21-ScZeBm9ysiqiqAehxTso2RmhW/A"
Date: Wed, 06 Nov 2024 08:49:40 GMT
Connection: close

{

  "message": "Bem-vindo ao painel"
}
```

3.2.1 Tipos de Middleware

No Express, você pode usar middlewares de três tipos principais:

- **Middleware de Aplicação:** Aplica-se globalmente a todas as rotas da aplicação.
- **Middleware de Rota:** Aplica-se a rotas específicas.
- **Middleware Incorporado e de Terceiros:** São middlewares que vêm com o Express ou de bibliotecas externas, como express.json() para processar JSON.

O **Middleware de Aplicação** será executado para todas as requisições da aplicação. Por exemplo, um middleware para registrar logs de todas as requisições.

```
const express = require("express");
const app = express();
const PORT = 3000;

// Middleware de log para todas as requisições
app.use((req, res, next) => {
  console.log(`${req.method} - ${req.url}`);
  next(); // Passa para o próximo middleware ou rota
});
// Rota para a página inicial
app.get("/", (req, res) => {
  res.json({ message: "Página inicial" });
});
// Rota para a página "sobre"
app.get("/sobre", (req, res) => {
  res.json({ message: "Página sobre" });
});
// Inicia o servidor e faz com que ele escute na porta definida
app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Neste exemplo, o middleware de log será executado para qualquer requisição, registrando o método HTTP e a URL no console antes de passar para a próxima etapa.

Para testar cada rota usando o REST Client no Visual Studio Code, você pode alterar o arquivo testes.rest da seguinte forma:

```
### Teste da rota GET /
GET http://localhost:3000/

### Teste da rota GET /sobre
GET http://localhost:3000/sobre
```

Após escrever a requisição, um link “Send Request” aparecerá acima dela. Clique nesse link para enviar a requisição. Por exemplo, veja o resultado após executar a rota raiz (/):

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 29
ETag: W/"1d-tAAM9qVuh5w3cNpjTPRkUjzu214"
Date: Wed, 06 Nov 2024 08:53:18 GMT
Connection: close

{

  "message": "Página inicial"
}
```

E, agora, o resultado após executar a requisição para a rota /sobre:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 27
ETag: W/"1b-xO8duZEKfbisXjf6Zg7R1fasJdI"
Date: Wed, 06 Nov 2024 08:53:32 GMT
Connection: close

{

  "message": "Página sobre"
}
```

Finalmente, veja o resultado no console do Node.js:

```
Servidor rodando em http://localhost:3000
GET - /
GET - /sobre
```

Figura 1: Resultado da aplicação no console do Node.js

Descrição: A imagem mostra uma página web aberta no navegador, no endereço http://localhost:3000/dashboard. No corpo da página, há o texto simples "Bem-vindo ao painel" em preto sobre fundo branco. Isso indica que o servidor está funcionando corretamente e a rota /dashboard está retornando essa mensagem como resposta.

Você pode aplicar **middlewares apenas para uma rota específica**. Por exemplo, vamos usar a rota **/perfi**l com um middleware que verifica se o usuário tem acesso autorizado ao perfil:

```
// Middleware para verificar acesso ao perfil
const checkAccess = (req, res, next) => {
  const authorized = true; // Altere para `false` para simular um usuário não autorizado
  if (authorized) {
    next(); // Usuário autorizado, passa para o próximo manipulador
  } else {
    res.status(403).json({ message: "Acesso ao perfil negado" }); // Resposta em JSON para não autorizado
  }
};
// Rota /perfil com middleware específico
app.get("/perfil", checkAccess, (req, res) => {
  res.json({ message: "Bem-vindo ao seu perfil!" }); // Resposta em JSON para autorizado
});
```

O Express vem com alguns **middlewares incorporados** para processar diferentes tipos de dados. Por exemplo:

- **express.json():** Analisa requisições com conteúdo JSON.
- **express.urlencoded():** Processa dados codificados de formulários (tipo **application/x-www-form-urlencoded**).

Veja um exemplo deste tipo de middleware abaixo:

```
app.use(express.json()); // Processa JSON em requisições
app.use(express.urlencoded({ extended: true })); // Processa dados de
formulário
app.post("/usuarios", (req, res) => {
  const { nome, email } = req.body;
  res.json({ message: `Usuário ${nome} com email ${email} criado!` });
});
```

Bibliotecas de terceiros também podem ser usadas como middlewares. Um exemplo popular é o **morgan**, que registra informações detalhadas de cada requisição. Para utilizar o Morgan, primeiro instale-o como uma dependência do seu projeto:

```
npm i morgan
```

Agora, no arquivo **server.js**, importe e configure o Morgan usando **app.use(morgan('dev'))**. O formato **'dev'** exibe logs com informações detalhadas de cada requisição:

```
const express = require("express");
const morgan = require("morgan"); // Importa o Morgan para log de requisições
const app = express();
const PORT = 3000;
// Configura o Morgan como middleware global para log de requisições
app.use(morgan("dev"));
// Rota para a página inicial
app.get("/", (req, res) => {
  res.json({ message: "Página inicial" }); // Resposta em JSON
});
// Rota para a página "sobre"
app.get("/sobre", (req, res) => {
  res.json({ message: "Página sobre" }); // Resposta em JSON
});
// Inicia o servidor
app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Observe que:

- **app.use(morgan('dev')):** Adiciona o Morgan como middleware global para registrar logs de cada requisição que o servidor recebe. O formato 'dev' mostra informações como método HTTP, URL, status e tempo de resposta de forma concisa.
- **Rotas:** Definimos duas rotas (**/** e **/sobre**) para que possamos observar o Morgan registrando logs quando cada rota é acessada.
- **Iniciar o Servidor:** O servidor escuta na porta 3000 e exibe uma mensagem no console quando está ativo.

Quando você acessa uma das rotas, o Morgan registra automaticamente a requisição no console. A saída do console pode ser parecida com esta:

```
Servidor rodando em http://localhost:3000
GET / 200 2.275 ms - 15
GET /sobre 200 0.356 ms - 13
```

Figura 2: Saída do console com os logs gerados pelo Morgan

Descrição: A imagem mostra um terminal onde o servidor em http://localhost:3000 registrou duas requisições. A primeira é uma requisição GET para a rota /, que retornou o status 200 (sucesso) em 2.275 ms, com um tamanho de resposta de 15 bytes. A segunda é uma requisição GET para a rota /sobre, também com status 200, respondida em 0.356 ms e com 13 bytes. Isso indica que ambas as rotas foram acessadas e responderam com sucesso.

4. Manipulação de Parâmetros de Rota e Query Strings

A manipulação de parâmetros de rota e query strings em Express.js é uma parte essencial do desenvolvimento de APIs, pois permite que o servidor receba e responda a requisições com base em valores fornecidos pelo cliente. Nesta seção está uma explicação detalhada com exemplos de como lidar com essas duas formas de passar dados em URLs.

4.1 Parâmetros de Rota

Os parâmetros de rota fazem parte da URL e são definidos com o uso de **:** seguido do nome do parâmetro. Eles são usados quando queremos capturar partes da URL para processar no servidor, como o ID de um recurso.

Veja no quadro abaixo um exemplo de manipulação de parâmetros de rota:

```
const express = require("express");
const app = express();
const PORT = 3000;

// Rota com parâmetro de rota :id
app.get("/usuarios/:id", (req, res) => {
  const { id } = req.params; // Extrai o parâmetro 'id' da URL
  res.json({ message: `Buscando o usuário com ID: ${id}` });
});

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Se o cliente fizer uma requisição para http://localhost:3000/usuarios/123, a resposta será:

```
{
  "message": "Buscando o usuário com ID: 123"
}
```

4.2 Query Strings

As query strings são pares de chave-valor passados após o símbolo **?** na URL. Elas são usadas para enviar dados adicionais na requisição de forma não hierárquica.

Veja abaixo um exemplo de manipulação de query strings:

```
app.get("/produtos", (req, res) => {
  const { categoria, precoMaximo } = req.query; // Extrai os valores da query string
  res.json({
    message: "Listando produtos",
    filtros: {
      categoria: categoria || "Todas",
      precoMaximo: precoMaximo || "Sem limite"
    }
  });
});
```

Desta forma, se o cliente fizer uma requisição para o caminho http://localhost:3000/produtos?categoria=eletronicos&precoMaximo=1000, a resposta será:

```
{
  "message": "Listando produtos",
  "filtros": {
    "categoria": "eletronicos",
    "precoMaximo": "1000"
  }
}
```

Se a URL for acessada sem query strings, a resposta usará valores padrão:

```
{
  "message": "Listando produtos",
  "filtros": {
    "categoria": "Todas",
    "precoMaximo": "Sem limite"
  }
}
```

Referências

O material desenvolvido para este capítulo baseou-se nas seguintes obras:

- ALVES, W. P. Projetos de sistemas Web: conceitos, estruturas, criação de banco de dados e ferramentas de desenvolvimento. São Paulo: Érica, 2019.
- CODEWELL, Krishna Rungta. Learn NodeJS in 1 Day. Independently published, 2017.
- FREITAS, P. E. C., et al. Programação back end 3. Porto Alegre: Sagah, 2021.
- LEDUR, C. L.; et al. Programação back end II. Porto Alegre: Sagah, 2019.
- OLIVEIRA, C. L. V.; et al. JavaScript descomplicado - Programação para a Web, IoT e dispositivos móveis. São Paulo: Érica, 2020.
- OLIVEIRA, C. L. V. Node.js - Programe de forma rápida e prática. São Paulo: Expressa, 2021.
- PEREIRA, Caio Ribeiro. Building APIs with Node.js. 1. ed. Birmingham: Packt Publishing, 2016.
- RODRIGUES, T. N.; et al. Integração de aplicações. Porto Alegre: Sagah, 2020.
- SKINNER, David Mark Clements. Node Cookbook. 2. ed. Birmingham: Packt Publishing, 2014.
- VITALINO, J. F. N.; CASTRO, M. A. N. Descomplicando o Docker. Editora Brasport, 2016.