Validação de Dados e Tratamento de Mensagens de Erro em uma API

A validação de dados e o tratamento adequado de erros são componentes essenciais ao desenvolver uma API robusta. Eles garantem que as entradas dos usuários sejam verificadas antes do processamento e que mensagens de erro úteis sejam fornecidas em caso de falhas. Neste laboratório, exploraremos como validar dados de entrada, definir mensagens de erro significativas e utilizar códigos de status HTTP adequados.

Parte 1: Validação de Dados

Por que validar dados?

A validação de dados é o processo de garantir que as entradas fornecidas pelo cliente atendam a determinados critérios antes de serem processadas pela API. Isso ajuda a prevenir problemas como:

- **Injeção de SQL ou XSS:** Dados maliciosos que podem comprometer a segurança da aplicação.
- **Erros Lógicos:** Dados incorretos que podem causar erros durante o processamento.
- **Integridade dos Dados:** Garantir que os dados armazenados estejam em um formato esperado.

Tipos de validação de dados

1. **Validação de Tipo:** Verifica se o tipo de dado (string, número, etc.) é o esperado.
2. **Validação de Formato:** Verifica se os dados estão em um formato específico, como um endereço de e-mail ou data.
3. **Validação de Limites:** Verifica se o dado está dentro de limites aceitáveis, como o comprimento de uma string ou o valor numérico mínimo/máximo.
4. **Validação de Presença:** Verifica se um campo obrigatório está presente.
5. **Validação de Relacionamento:** Verifica se a relação entre os campos é válida, como uma data de início ser anterior a uma data de fim.

Parte 2: Tratamento de Erros e Códigos de Status HTTP

Principais Códigos de Status HTTP para Erros

Os códigos de status HTTP indicam o resultado de uma requisição. Quando ocorre um erro, é crucial retornar o código correto para que o cliente saiba como lidar com ele.

1. **400 Bad Request:** A requisição não pôde ser processada devido a um erro do cliente (por exemplo, validação de dados falhou).
    - Mensagem Padrão: "A requisição contém parâmetros inválidos."
2. **401 Unauthorized:** A requisição requer autenticação do usuário, mas não foi fornecida ou é inválida.
    - Mensagem Padrão: "Autenticação necessária."
3. **403 Forbidden:** O cliente não tem permissão para acessar o recurso, mesmo que autenticado.
    - Mensagem Padrão: "Você não tem permissão para acessar este recurso."
4. **404 Not Found:** O recurso solicitado não foi encontrado no servidor.
    - Mensagem Padrão: "O recurso solicitado não foi encontrado."
5. **409 Conflict:** A requisição entrou em conflito com o estado atual do servidor (por exemplo, tentativa de criar um recurso que já existe).
    - Mensagem Padrão: "Conflito ao processar a requisição."
6. **429 Too Many Requests:** O cliente enviou muitas requisições em um curto período de tempo.
    - Mensagem Padrão: "Muitas requisições. Tente novamente mais tarde."
7. **500 Internal Server Error:** Ocorreu um erro inesperado no servidor.
    - Mensagem Padrão: "Erro interno do servidor."
8. **503 Service Unavailable:** O servidor está temporariamente indisponível, geralmente devido a manutenção ou sobrecarga.

- Mensagem Padrão: "Serviço temporariamente indisponível."

Estrutura de Mensagens de Erro

Ao retornar erros, é importante fornecer uma mensagem clara que ajude o cliente a entender o que deu errado e como corrigir.

- **Mensagem de Erro:** Uma breve descrição do problema.
- **Campo (Opcional):** Se o erro estiver relacionado a um campo específico, informe-o.
- **Código de Erro (Opcional):** Um código único que representa o erro (útil para depuração e suporte).

```
{
    "error": {
        "message": "O campo 'email' é inválido.",
        "field": "email",
        "code": "INVALID_EMAIL_FORMAT"
    }
}
```

Parte 3: Validação de Dados na Prática

Usando o framework Express com TypeScript, podemos implementar uma simples validação de dados. Vamos validar uma rota de registro de usuário.

Exemplo de Validação

1. **Configuração do Projeto**

   Certifique-se de ter um projeto configurado com TypeScript e Express.

2. **Middleware de Validação**

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

const validateUser = (req: Request, res: Response, next: NextFunction) => {
    const { username, email, password } = req.body;

    if (!username || typeof username !== 'string') {
        return res.status(400).json({
            error: {
                message: "O campo 'username' é obrigatório e deve ser uma string.",
                field: 'username',
                code: 'INVALID_USERNAME'
            }
        });
    }

    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    if (!email || !emailRegex.test(email)) {
        return res.status(400).json({
            error: {
                message: "O campo 'email' é inválido.",
                field: 'email',
                code: 'INVALID_EMAIL_FORMAT'
            }
        });
    }
```

```
    if (!password || password.length < 6) {
        return res.status(400).json({
            error: {
                message: "O campo 'password' é obrigatório e deve ter pelo menos 6
caracteres.",
                field: 'password',
                code: 'INVALID_PASSWORD'
            }
        });
    }

    next(); // Prosseguir para a próxima função de middleware ou rota
};

export default validateUser;
```

3. **Uso do Middleware na Rota:**

```
import express, { Request, Response } from 'express';
import validateUser from './middleware/validateUser';

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

app.post('/register', validateUser, (req: Request, res: Response) => {

    // Se a validação passar, prossiga com o processamento da requisição
    const { username, email, password } = req.body;

    // Simulação de criação do usuário
    res.status(201).json({ message: 'Usuário registrado com sucesso!' });
});

app.listen(3000, () => {
    console.log('Servidor rodando na porta 3000');
});
```

Parte 4: Melhores Práticas para Validação e Tratamento de Erros

1. **Validação no Lado do Servidor:** Nunca confie apenas na validação do lado do cliente; sempre valide no servidor.
2. **Códigos de Status HTTP Corretos:** Use códigos de status HTTP adequados para indicar o tipo de erro.
3. **Mensagens de Erro Úteis:** Forneça mensagens de erro claras e úteis que ajudem o cliente a entender e corrigir o problema.
4. **Não Expor Detalhes Internos:** Evite expor detalhes internos do sistema ou pilhas de erros, pois isso pode ser uma vulnerabilidade de segurança.
5. **Documentação:** Documente claramente os erros possíveis em sua API e os códigos de status associados.
6. **Manejo de Erros Gerais:** Implemente um middleware de tratamento de erros globais para capturar e retornar erros inesperados.