Questões Teóricas

1. O que caracteriza uma API RESTful?

Uma API RESTful é definida por sua aderência aos princípios arquiteturais REST, que incluem o uso explícito de métodos HTTP, statelessness, identificação única de recursos via URIs e representação padronizada de dados, geralmente em formatos como JSON ou XML. Sua característica central é a escalabilidade e a interoperabilidade entre sistemas heterogêneos.

2. Quais são os verbos HTTP mais usados em APIs e suas funções?

Os verbos HTTP fundamentais são GET (recuperar recursos), POST (criar recursos), PUT/PATCH (atualizar recursos) e DELETE (remover recursos). Cada verbo opera de forma idempotente (exceto POST) e segue convenções semânticas para operações CRUD.

3. Explique a vantagem de estruturar projetos em pastas como controllers, models e routes.

Esta estrutura promove separação de preocupações seguindo o padrão MVC, onde controllers gerenciam lógica de negócio, models representam dados e routes definem endpoints. Melhora manutenção, testabilidade e escalabilidade do código.

4. O que é Joi e para que serve?

Joi é uma biblioteca de validação de dados para JavaScript que permite definir esquemas declarativos para garantir integridade de entradas em APIs, verificando tipo, formato e obrigatoriedade dos dados antes do processamento.

5. Como a paginação melhora a performance de uma API?

A paginação divide grandes conjuntos de dados em subconjuntos menores (páginas), reduzindo carga no banco de dados e tempo de transferência. Técnicas como LIMIT e OFFSET em SQL são comumente utilizadas.

6. Qual é a finalidade do Swagger em projetos de API?

Swagger (OpenAPI) padroniza a documentação de APIs, descrevendo endpoints, métodos, parâmetros e respostas em formato legível por humanos e máquinas, facilitando integração e testes.

7. O que é o conceito de filtros em consultas?

Filtros permitem restringir resultados de consultas usando condições específicas (como WHERE em SQL) baseadas em parâmetros como datas, valores ou categorias, melhorando precisão e eficiência.

8. Explique como implementar um endpoint seguro para criar registros no MySQL.

Deve-se usar prepared statements para prevenir injeção SQL, validar entradas com Joi, implementar autenticação (JWT), criptografar dados sensíveis e utilizar transações para atomicidade.

9. Quais são os principais benefícios de validar dados na camada da API?

Previne erros, vulnerabilidades e inconsistências; melhora experiência do usuário com feedbacks adequados; e reduz processamento desnecessário em camadas posteriores.

10. Como o Express simplifica o desenvolvimento de APIs RESTful?

O Express oferece abstração para rotas, middlewares e tratamento de requisições/respostas com sintaxe simplificada, integrando-se facilmente com outras bibliotecas para criar APIs robustas de forma eficiente.

Questões Práticas

1. **Estruture um projeto para uma API RESTful com as pastas adequadas.**

Ver projeto em anexo

2. **Implemente um endpoint para listar todos os registros de uma tabela.**

```
app.get('/usuarios', (req, res) => {
  connection.query('SELECT * FROM usuarios', (err, results) => {
    if (err) {
      res.status(500).send(err);
      return;
    }
```

```
    res.json(results);
  });
});
```

3. Crie um endpoint para inserir um novo registro no banco de dados, validando os dados com Joi.

```
const Joi = require('joi');

// Definindo o schema de validação com Joi
const schema = Joi.object({
  nome: Joi.string().min(3).required(),
  email: Joi.string().email().required(),
  senha: Joi.string().min(6).required()
});

// Endpoint para inserir novo usuário
app.post('/usuarios', (req, res) => {
  const { error } = schema.validate(req.body);

  if (error) {
    res.status(400).send(error.details[0].message);
    return;
  }

  const { nome, email, senha } = req.body;

  connection.query(
    'INSERT INTO usuarios (nome, email, senha) VALUES (?, ?, ?)',
    [nome, email, senha],
    (err, result) => {
      if (err) {
        res.status(500).send(err);
        return;
```

```
      }
      res.status(201).send('Usuário criado com sucesso!');
    }
  );
});
```

4. Adicione paginação a um endpoint que lista registros.

```
const Joi = require('joi');

// Schema de validação
const schema = Joi.object({
  nome: Joi.string().min(3).required(),
  email: Joi.string().email().required(),
  senha: Joi.string().min(6).required(),
  page: Joi.number().integer().min(1).default(1),
  limit: Joi.number().integer().min(1).default(10)
});

app.post('/usuarios', (req, res) => {
  const { error, value } = schema.validate(req.body);

  if (error) {
    return res.status(400).send(error.details[0].message);
  }

  const { nome, email, senha, page, limit } = value;

  connection.query(
    'INSERT INTO usuarios (nome, email, senha) VALUES (?, ?, ?)',
    [nome, email, senha],
    (err) => {
      if (err) {
        return res.status(500).send(err);
      }
```

```
    const offset = (page - 1) * limit;

    connection.query(
      'SELECT * FROM usuarios LIMIT ? OFFSET ?',
      [limit, offset],
      (err, results) => {
        if (err) {
          return res.status(500).send(err);
        }

        res.status(201).json({
          mensagem: 'Usuário criado com sucesso!',
          pagina: page,
          limite: limit,
          usuarios: results
        });
      }
    );
  }
);
});
```

5. **Crie um endpoint que permita atualizar registros pelo ID.**

```
const Joi = require('joi');


// Schema de validação
const schema = Joi.object({
  nome: Joi.string().min(3).required(),
  email: Joi.string().email().required(),
  senha: Joi.string().min(6).required()
});
```

```
// Endpoint PUT para atualizar usuário por ID
app.put('/usuarios/:id', (req, res) => {
  const { error } = schema.validate(req.body);

  if (error) {
    return res.status(400).send(error.details[0].message);
  }

  const { nome, email, senha } = req.body;
  const { id } = req.params;

  connection.query(
    'UPDATE usuarios SET nome = ?, email = ?, senha = ? WHERE id = ?',
    [nome, email, senha, id],
    (err, result) => {
      if (err) {
        return res.status(500).send(err);
      }

      if (result.affectedRows === 0) {
        return res.status(404).send('Usuário não encontrado.');
      }
```

```
    res.send('Usuário atualizado com sucesso!');
  }
 );
});
```

6. Implemente um endpoint para excluir registros com base no ID.

```
app.delete('/usuarios/:id', (req, res) => {
 const { id } = req.params;

 connection.query('DELETE FROM usuarios WHERE id = ?', [id], (err, result)
=> {
  if (err) {
   return res.status(500).send(err);
  }

  if (result.affectedRows === 0) {
   return res.status(404).send('Usuário não encontrado.');
  }

  res.send('Usuário excluído com sucesso!');
 });
});
```

7. Configure um filtro para buscar registros pelo nome em uma tabela.

```
app.get('/usuarios', (req, res) => {
 const { nome } = req.query;

 let sql = 'SELECT * FROM usuarios';
 let params = [];
```

```
  if (nome) {
    sql += ' WHERE nome LIKE ?';
    params.push(`%${nome}%`);  // filtro parcial com % para qualquer posição
  }

  connection.query(sql, params, (err, results) => {
    if (err) {
      return res.status(500).send(err);
    }
    res.json(results);
  });
});
```

8. Documente sua API com Swagger, incluindo os endpoints criados.

swagger.js

```
const swaggerJsdoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');

// Configuração do Swagger
const options = {
  definition: {
    openapi: "3.0.0",
    info: {
```

```
    title: "API de Usuários",
    version: "1.0.0",
    description: "API para gerenciamento de usuários com CRUD"
  },
  servers: [
    {
      url: "http://localhost:3000",
      description: "Servidor local"
    }
  ]
 },
 apis: ["./index.js"] // arquivos onde estão os comentários da API
};

const specs = swaggerJsdoc(options);

function setupSwagger(app) {
 app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(specs));
}

module.exports = setupSwagger;
```

index.js

```
/**
```

```
* @swagger
* components:
*   schemas:
*     Usuario:
*       type: object
*       required:
*         - nome
*         - email
*         - senha
*       properties:
*         id:
*           type: integer
*           description: ID do usuário
*         nome:
*           type: string
*           description: Nome do usuário
*         email:
*           type: string
*           description: Email do usuário
*         senha:
*           type: string
*           description: Senha do usuário
*       example:
*         id: 1
```

```
 *         nome: João Silva
 *         email: joao@email.com
 *         senha: 123456
 */

/**
 * @swagger
 * /usuarios:
 *   get:
 *     summary: Lista todos os usuários
 *     responses:
 *       200:
 *         description: Lista de usuários
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/Usuario'
 */
app.get('/usuarios', (req, res) => {
  // implementação...
});
```

```
/**
 * @swagger
 * /usuarios:
 *   post:
 *     summary: Cria um novo usuário
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Usuario'
 *     responses:
 *       201:
 *         description: Usuário criado com sucesso
 *       400:
 *         description: Dados inválidos
 */
app.post('/usuarios', (req, res) => {
  // implementação...
});

/**
 * @swagger
 * /usuarios/{id}:
```

```
*   put:
*     summary: Atualiza um usuário pelo ID
*     parameters:
*       - in: path
*         name: id
*         required: true
*         schema:
*           type: integer
*         description: ID do usuário
*     requestBody:
*       required: true
*       content:
*         application/json:
*           schema:
*             $ref: '#/components/schemas/Usuario'
*     responses:
*       200:
*         description: Usuário atualizado com sucesso
*       400:
*         description: Dados inválidos
*       404:
*         description: Usuário não encontrado
*/
app.put('/usuarios/:id', (req, res) => {
```

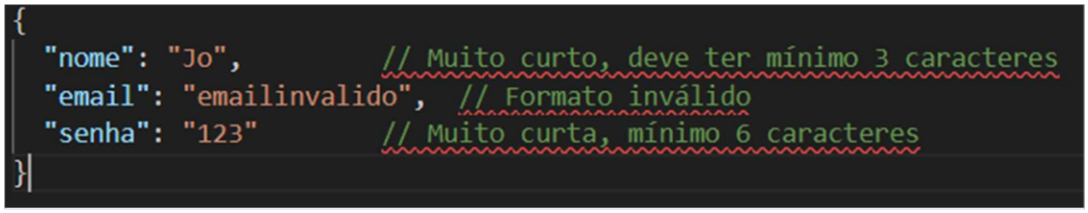
```
  // implementação...
});

/**
 * @swagger
 * /usuarios/{id}:
 *   delete:
 *     summary: Exclui um usuário pelo ID
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *         description: ID do usuário
 *     responses:
 *       200:
 *         description: Usuário excluído com sucesso
 *       404:
 *         description: Usuário não encontrado
 */
app.delete('/usuarios/:id', (req, res) => {
  // implementação...
});
```

9. **Teste a validação de dados no backend enviando requisições com dados inválidos.**

Para testar a validação de dados no backend, implementei um endpoint na rota POST /usuarios que recebe três atributos obrigatórios no corpo da requisição: nome, email e senha. Utilizei a biblioteca Joi para realizar a validação desses dados, definindo um schema que exige que o nome tenha no mínimo 3 caracteres, o email esteja em um formato válido e a senha tenha ao menos 6 caracteres. O schema foi definido conforme o exemplo a seguir: const schema = Joi.object({ nome: Joi.string().min(3).required(), email: Joi.string().email().required(), senha: Joi.string().min(6).required() }). Dentro do endpoint, utilizei const { error } = schema.validate(req.body) para verificar se os dados enviados estão em conformidade com as regras estabelecidas. Caso haja algum erro, a resposta retorna com status 400 e a mensagem de erro detalhada.

Para testar a validação, utilizei a ferramenta Postman enviando requisições com diferentes combinações de dados inválidos. Um dos testes consistiu em enviar um corpo JSON com nome muito curto ("Jo"), email em formato incorreto ("emailinvalido") e senha com apenas três caracteres ("123"). O corpo da requisição enviado foi: { "nome": "Jo", "email": "emailinvalido", "senha": "123" }. Como esperado, o backend retornou status 400 e uma mensagem de erro indicando que o nome deve ter no mínimo 3 caracteres, o email deve ser válido e a senha deve ter pelo menos 6 caracteres. Também testei o envio de campos nulos ou ausentes, como no exemplo: { "nome": null, "email": 12345, "senha": true }, o que gerou respostas coerentes com o tipo e obrigatoriedade dos dados esperados. Além disso, utilizei o comando curl via terminal para simular requisições rápidas, como em curl -X POST http://localhost:3000/usuarios -H "Content-Type: application/json" -d '{"nome":"Jo","email":"invalido","senha":"123"}'. Esses testes demonstraram que a camada de validação com Joi está funcionando corretamente, impedindo que dados fora do padrão sejam armazenados no banco de dados. Com isso, a aplicação garante maior robustez e segurança ao tratar os dados logo na entrada, evitando inconsistências e potenciais falhas em etapas posteriores do sistema.



```
{
  "nome": "Maria"
  // email e senha ausentes
}
```

```
{
  "nome": null,
  "email": 12345,
  "senha": true
}
```

```
curl -X POST http://localhost:3000/usuarios \
  -H "Content-Type: application/json" \
  -d '{"nome":"Jo","email":"inválido","senha":"123"}'
```