Introdução ao Node.js

Antes de discutir sobre Node.js, vale relembrar a estrutura de uma aplicação web.

Mas o que é uma aplicação web? Para os efeitos dessa disciplina, a seguinte definição (texto próprio) será utilizada:

> *Uma aplicação web é um software que disponibiliza sua funcionalidade através de conexões de rede, normalmente consumidas por navegadores web através da Internet. Outros clientes (como aplicações em dispositivos móveis) e topologias de rede que não dependem da Internet também são possíveis e não a descaracterizam.*

Com base nessa definição, é possível revisitar alguns conceitos chaves para qualquer aplicação web. São eles:

Protocolo HTTP

É o protocolo mais conhecido e difundido na construção de aplicações web. Baseado no protocolo TCP/IP, define uma abstração para que clientes (dentre eles, navegadores web) efetuem *requisições* e recebam *respostas*.

Durante sua concepção, o protocolo HTTP (junto com a linguagem HTML) tinha um propósito bem diferente do atual: permitir o acesso e modificação de documentos de texto com formatações simples e *links* navegáveis. Para atender a este objetivo, bastava permitir uma comunicação unilateral (partindo do cliente).

Requisição HTTP

Sempre que um cliente web (ex: navegador) precisa de alguma informação de um servidor (ex: uma página HTML, uma folha de estilo, uma imagem, etc.) ele a solicita através de uma requisição HTTP[1]. Essa requisição é composta das seguintes partes:

- **Método:** define o objetivo da requisição. Os principais métodos HTTP são: GET/POST/PUT/PATCH/DELETE.
- **Caminho:** é a parte que fica *depois* do endereço do servidor, incluindo a *query string*. Como exemplo, considere a URL abaixo:

> *http://www.minhaaplicacao.com/clientes?nome=Maria*

A parte antes do `://`, `http`, é o protocolo de comunicação utilizado para iniciar a conversa com o servidor. `www.minhaaplicacao.com` é um nome completo de domínio (*Fully-Qualified Domain Name*, FQDN), passado pelo sistema de DNS resultando em um endereço IP. Opcionalmente uma porta pode ser definida, caso contrário a porta padrão do protocolo de comunicação é usada (no caso do protocolo HTTP essa porta padrão é a 80). O cliente usa o IP e a porta para estabelecer uma conexão TCP, e enfim chegamos ao *Caminho* da requisição, que é tudo que ficou *depois* do FQDN e da opcional porta. No caso acima, esse caminho é `/clientes?nome=Maria`.

- **Cabeçalhos (Headers):** são atributos da requisição que podem ser usados para diversos fins, como formatos aceitos, autorização/autenticação, idiomas suportados, etc. Os *cookies*, por exemplo, nada mais são que um *header* especialmente processado pelos servidores e navegadores, para dar a impressão de estado persistente entre várias conexões.

- **Corpo:** dependendo do método da requisição, um *corpo* pode ser enviado junto com ela. Esse corpo pode ser o conteúdo de um formulário preenchido pelo usuário, um arquivo selecionado para *upload*, um objeto no formato JSON, dentre outras coisas.

Durante o processamento da requisição HTTP, o servidor produz uma *resposta*, assim composta:

- **Status Code:** um valor numérico que indica de maneira geral o resultado daquela requisição. Dividida em 6 principais grupos, de acordo com a *centena* do código: 1xx (informativo), 2xx (sucesso), 3xx (redirecionamento), 4xx (erro do cliente), 5xx (erro no servidor). Alguns exemplos clássicos: 404 (página não encontrada), 403 (não autorizado), 301 (redirecionamento permanente), 200 (sucesso).
- **Cabeçalhos:** a resposta também contém cabeçalhos, assim como a requisição. Um exemplo é o cabeçalho `Content-Type`, que descreve o formato do corpo da resposta.
- **Corpo:** quando aplicável, uma resposta pode conter um corpo, seja ele uma imagem, uma página HTML, um objeto JSON, um PDF para download, ou qualquer outra coisa.

[1] Atualmente existem outras formas de se efetuar essa comunicação, como WebSockets, mas para fins de simplificar a revisão essas outras tecnologias foram desconsideradas.

Front-end (lado do cliente)

É possível observar que ao desenvolver uma aplicação web, é necessário se preocupar com pelo menos dois componentes: o navegador, responsável pela interação com o usuário, e o servidor, responsável por processar as requisições.

Como funciona então o desenvolvimento no lado do cliente? Não é regra, mas normalmente se espera o uso de um *navegador web* (Chrome, Firefox, Edge, Safari, etc.). Esses navegadores estão preparados para mostrar páginas HTML, que por sua vez podem depender de folhas de estilo CSS e código JavaScript.

JavaScript no navegador web

Originalmente, a linguagem JavaScript foi concebida com o intuito de permitir a implementação de dinamizações simples nas páginas web, como validações em campos de formulário, estilização dinâmica, etc. Não se pensava no uso do JavaScript como principal linguagem em uma aplicação (como é comum hoje em dia), muito menos no uso dele como linguagem no lado servidor (como essa disciplina inteira propõe e defende).

Note que o fato de usar JavaScript no servidor, através de uma ferramenta como o Node.js, não muda em nada o fato de existirem requisições HTTP no meio do caminho. Uma boa regra pra se ter em mente é: para o código JavaScript que está sendo executado no navegador/cliente, é impossível discernir se o servidor está sendo desenvolvido em JavaScript, Java, Python ou qualquer outra linguagem. O navegador e o servidor conversam entre si usando HTTP, o Node.js não muda isso (e nem tem essa intenção).

Back-end (lado do servidor)

Composto por um *servidor web*, capaz de aceitar as requisições HTTP, e normalmente por um *framework web*, responsável por facilitar o desenvolvimento do software que entende as requisições e gera as respostas apropriadas, conversando com quaisquer fontes de dados (ex: banco de dados relacional) ou serviços externos (ex: API para envio de email) necessários para isso. É aqui que entra o Node.js e todo o seu ecossistema.

Modelo de execução do JavaScript

Os navegadores executam JavaScript de modo Single Thread e seguindo uma arquitetura baseada em eventos. Mas o que exatamente isso significa?

Um processo no sistema operacional pode iniciar uma ou mais *threads*. De modo simplificado, cada *thread* é uma sequência de execução, podendo ou não ser executada paralelamente às outras (em máquinas com mais de um núcleo). Mesmo com um único núcleo, as *threads* normalmente são *escalonadas* pelo sistema operacional de modo a darem a impressão de paralelismo através de execução concorrente.

O desenvolvimento *multi-thread* traz consigo um conjunto de desafios, principalmente quando há o compartilhamento de *recursos* (periféricos, sistema de arquivos, etc.) e *estado* (memória).

Na concepção do JavaScript, foi definido que a execução se daria em uma única thread (pelo menos do ponto de vista do desenvolvedor. Nada impede, e isso de fato ocorre, que motores JavaScript executem em múltiplas threads, desde que isso fique abstraído do desenvolvedor final). Mas isso soa bem engessado. Como as aplicações conseguem atingir níveis de responsividade e dinamismo sem a capacidade de executar código de forma concorrente/paralela?

É verdade que não há como executar código JavaScript de forma *paralela*, mas a *concorrência* pode ser atingida sim, através de um conceito chamado *Event Loop*.

Todo motor de JavaScript implementa um loop de eventos, basicamente composto por uma fila de tarefas. Todo código JavaScript é uma tarefa chamada por esse loop, seja essa tarefa criada por um evento de usuário (clique em um botão) ou evento de navegador (término do carregamento de uma página, término de uma requisição AJAX, término do tempo de espera de temporizadores, etc.). Considere o seguinte exemplo:

```
<!DOCTYPE html>
<html>
<body onload="digaOi()">
  <script>
    function digaOi() {
      console.log('Oi!');
    }
    setTimeout(function () {
      console.log('5s depois...');
    }, 5000);
    console.log('fim da tag script');
  </script>
  <button onclick="digaOi()">Diga oi!</button>
</body>
</html>
```

O loop de eventos inicia processando a tag `script`, que define a função `digaOi`, registra uma outra tarefa para ser executada depois de 5 segundos (note que isso só é possível pois funções são *cidadãs de primeira classe* no JavaScript, o método `setTimeout` está recebendo uma função como parâmetro), e por fim escreve um texto no console do navegador. Assim que a página termina de ser processada, o próprio navegador executa o script contido no atributo `onload` do elemento `body`. E também, sempre que o usuário clica no botão "Diga oi!", uma tarefa é registrada no loop de eventos, responsável por executar o script do atributo `onclick` deste botão.

Note que nada aqui foi executado de modo paralelo, mas a ideia de concorrência é fortemente presente graças ao uso de um loop de eventos.

Requisições AJAX

Esse modelo de execução gerou um padrão de codificação muito difundido em aplicações web, na parte do front-end: a execução de requisições HTTP iniciadas pelo próprio JavaScript, de modo assíncrono. Quando uma resposta é recebida, um *callback* é disparado, ou em outras palavras, o navegador coloca uma tarefa no loop de eventos para executar uma função passando a resposta da requisição, seja ela positiva ou não. Veja o exemplo:

```
<!DOCTYPE html>
<html>
<body>
  <script>
    function tratarErro(erro) {
      document.getElementById('resultado').innerText = `Erro: ${erro}`;
    }
    function buscar() {
      fetch('https://rickandmortyapi.com/api/character/1').then(resp => {
        if (resp.ok) {
          resp.json().then(dados => {
            document.getElementById('resultado').innerText = `Nome: ${dados.name}`;
          }, tratarErro);
        } else {
          tratarErro(resp.statusText);
        }
      }, tratarErro);
    }
  </script>
  <button onclick="buscar()">Buscar</button>
  <span id="resultado">
  </span>
</body>
</html>
```

Node.js

Agora que a revisão foi feita, é possível olhar para o Node.js e entender onde ele se propõe a trabalhar. Considere a definição no próprio site da ferramenta:

> *Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.*

Basicamente o que isso quer dizer é: uma camada sobre um motor de execução de JavaScript muito poderoso (V8), permitindo o desenvolvimento de aplicações que não executam no contexto de um navegador.

Instalação

O primeiro passo para o aprendizado do Node.js é a instalação da ferramenta. Como isso varia muito dependendo do sistema operacional, a única coisa que será feita aqui é o apontamento para a página de download: https://nodejs.org/en/download/. Note que virtualmente todas as distribuições Linux possuem o Node.js em seu gestor de pacotes.

Para verificar se você já possui uma instalação, ou se a que acabou de fazer deu certo, abra um terminal e execute o seguinte comando:

```
node -v
```

Se o comando retornar um número de versão, sem falha, significa que deu certo. Garanta que você possui a versão 12 ou superior para evitar problemas no decorrer da disciplina.

Node Version Manager (NVM)

Do mesmo modo que ocorre em outras linguagens, pode ser que você se encontre em uma situação onde precise ter várias versões do Node na máquina, uma para cada projeto em que está trabalhando. Neste tipo de situação é comum ouvir o termo "ambiente virtual" (virtual environment). No caso do Node, uma das ferramentas que auxilia na gestão de ambientes virtuais é o `nvm` (https://github.com/nvm-sh/nvm).

Note que o SO Windows não é suportado por essa ferramenta, então a instalação raíz deverá ser usada durante a disciplina, ou deverá ser efetuada a instalação de uma alternativa como o `nvm-windows` (https://github.com/coreybutler/nvm-windows).

Para instalar a distribuição mais recente, use o comando `nvm install node`. Para instalar a distribuição LTS mais recente, use `nvm install lts/*`. Para listar as instalações disponíveis na máquina, use `nvm ls`. Para selecionar uma instalação específica, use `nvm use <versão>`.

Primeiro script Node.js

Para executar um script usando o ambiente de execução do Node.js, basta abrir um terminal e usar o seguinte comando:

```
node arquivo.js
```

Onde `arquivo.js` é o script que deseja executar. Faça um teste com o seguinte código de exemplo:

```
console.log(1 + 2);
```

Salve o texto acima em um arquivo e execute-o. A saída deve ser algo parecido com isso:

```
$ node arquivo.js
3
```

Note que, no decorrer deste material, linhas iniciadas com `$` significam um comando digitado no terminal, enquanto as outras linhas significam a saída desses comandos.

Modifique agora o arquivo com o código abaixo, um pouco mais complicado, incluindo leitura de dados do usuário:

```
const readline = require('readline');

const reader = readline.createInterface(process.stdin, process.stdout);
reader.question('Seu nome: ', nome => {
    console.log(`Olá, ${nome}!`);
});
reader.addListener("line", linha => {
    if (linha.toLowerCase() === 'sair') {
        reader.close();
```

```
    }
});
```

Pontos interessantes para observar no código acima:

- Importação de módulo usando a função `require`. O pacote `readline` é um pacote nativo do Node.js que permite interação facilitada com o usuário através da linha de comando.
- Uso do método `question`, passando um `callback`, para implementar a interação com o usuário.
- Uso do método `addListener`, para verificar se o usuário digitou o termo `sair`, e em caso positivo desligar o leitor, efetivamente encerrando o processo.

Por qual motivo esse último passo foi necessário? Como discutido nas seções anteriores, o Node.js é construído sobre o motor de execução V8, que por sua vez implementa um looping de eventos. A partir do momento em que o leitor de dados da interface é criado, ele impede que a aplicação encerre após o término do script, pois ele é uma fonte de eventos desse loop. Em outras palavras, enquanto o leitor não for fechado manualmente, ou o método `process.exit(status)` for chamado, o processo não será encerrado e segurará o terminal do usuário.

Debugging

Uma ferramenta muito importante para se ter enquanto desenvolvedor de software é um bom método de *debugging*. No caso do Node.js, existem várias disponíveis:

- Manualmente, usando `node inspect` ao invés de simplesmente `node` para executar o script. Obviamente essa opção é péssima em termos de usabilidade.
- Acessar a URL `chrome://inspect` no navegador Chrome e anexar a um processo `node` iniciado com a flag `--inspect`, ex: `node --inspect arquivo.js`.
- Iniciar o processo por dentro de uma IDE com suporte ao modo de debug. O Visual Studio Code é um exemplo: habilite a configuração *Debug > Node: Auto Attach*, abra um terminal integrado, e inicie o script com a flag `--inspect` normalmente.

Módulos

Virtualmente todas as linguagens de programação oferecem alguma maneira de divisão de código em *partes*. Algumas chamam-as de *pacotes*, outras de *módulos*, etc.

O JavaScript puro não oferecia esse conceito, mas conforme a sua evolução essa capacidade foi incorporada, e hoje faz parte da especificação da linguagem (ES6+) [1].

Note que também existe um gestor de dependências/pacotes para Node.js, chamado `npm`, mas este não será apresentado por enquanto.

Considere o seguinte código:

```
const a = 3;
const b = 10;
console.log(a + b);
```

`a + b` aqui representa uma potencial operação de negócio, que eventualmente será reutilizada em outros pontos do projeto. Quando isso acontecer, como fazer para evitar a duplicação do código? Comece criando um outro arquivo, chamado `soma.js`, com o seguinte código:

```
module.exports = function (a, b) {
  return a + b;
};
```

E ajuste o arquivo anterior:

```
const soma = require('./soma');

const a = 3;
const b = 10;
console.log(soma(a, b));
```

Repare que desse modo, um único objeto (seja ele um valor, uma função, uma classe, etc.) é exportado pelo módulo. Também é possível exportar vários elementos. Crie um arquivo `operacoes.js` com o seguinte código:

```
module.exports.PI = 3.1415;

module.exports.soma = function (a, b) {
    return a + b;
};

module.exports.sub = function (a, b) {
    return a - b;
};

module.exports.mult = function (a, b) {
    return a * b;
};

module.exports.div = function (a, b) {
    return a / b;
};
```

E adicione o seguinte código no script original:

```
const operacoes = require('./operacoes');
console.log(operacoes.PI * 2);
console.log(operacoes.soma(10, 3));
console.log(operacoes.sub(10, 3));
console.log(operacoes.mult(10, 3));
console.log(operacoes.div(10, 3));
```

Para os familiarizados com Angular e TypeScript (que permitem o uso da sintaxe de módulos do ES6), o equivalente em um projeto com essas tecnologias seria:

```
// soma.js
export default function (a, b) {
  return a + b;
}
```

```
// operacoes.js
export const PI = 3.1415;
```

```
export function soma(a, b) {
  return a + b;
}
export function sub(a, b) {
  return a - b;
}
[...]
```

```
// script.js
import soma from './soma.js';
import * as operacoes from './operacoes.js';
import { PI, soma, sub } from './operacoes.js'; // preferível
[...]
```

Infelizmente a sintaxe de módulos do ES6 ainda é experimental no Node.js, portanto seu uso ainda não é apropriado [2].

[1] https://medium.com/sungthecoder/javascript-module-module-loader-module-bundler-es6-module-confused-yet-6343510e7bde.

[2] https://nodejs.org/api/esm.html#esm_introduction

Sistema de arquivos

Uma das melhores maneiras de aprender uma nova linguagem é aplicando conceitos já conhecidos em outras, e comparar as soluções obtidas.

Um desses conceitos bem comuns é a manipulação do sistema de arquivos. O Node.js fornece uma API para isso [1], evidentemente. Veja o exemplo abaixo de leitura de um arquivo:

```
const fs = require('fs');

const data = fs.readFileSync('lorem.txt');
console.log(data);
```

Repare que o retorno é um objeto do tipo `Buffer`. Isso ocorre pois o Node.js não assume que o arquivo contém texto. É possível obter texto (`string`) de duas formas:

1. Chamando o método `toString` passando a codificação correta:

   ```
   console.log(data.toString('UTF-8'));
   ```

2. Passando a codificação diretamente na chamada do método `readFileSync`:

   ```
   const data = fs.readFileSync('lorem.txt', {
      encoding: 'UTF-8'
   });
   console.log(data);
   ```

Note que essas operações são síncronas, e como já discutido, o Node.js executa em uma única thread sobre um loop de eventos. Basicamente o que isso significa é que se o arquivo lido for muito grande, o processo (sua aplicação) inteira vai ficar parada aguardando o carregando dos dados.

Isso é definitivamente uma situação indesejada, e é esperado de todo programador Node.js o conhecimento das APIs assíncronas (elas existem para todos os lados) equivalentes e um bom

julgamento na hora de escolher qual usar (a API assíncrona prioriza a performance, enquanto a API síncrona prioriza a simplicidade do código). Veja como fica o exemplo acima usando a API assíncrona:

```
const options = { encoding: 'UTF-8' };
const callback = (err, data) => {
    if (err) throw err;
    console.log(data);
};
fs.readFile('lorem.txt', options, callback);
console.log('A leitura ainda não ocorreu');
```

Assim que o método `readFile` é chamado, a execução continua, e quando os dados estiverem prontos (ou um erro for identificado) o loop de eventos vai chamar a função passada como callback para que o processamento continue.

Não é difícil perceber que a codificação assíncrona fica rapidamente complexa de entender. Considere o seguinte código, que primeiro lê o conteúdo de um arquivo, depois pergunta uma informação ao usuário, depois adiciona essa informação no arquivo:

```
const fs = require('fs');
const readline = require('readline');

const reader = readline.createInterface(process.stdin, process.stdout);

fs.readFile('dados.txt', { encoding: 'UTF-8' }, (err, data) => {
    if (err) throw err;
    console.log(data);
    reader.question('Linha: ', linha => {
        fs.appendFile('dados.txt', linha + '\n', err => {
            if (err) throw err;
            reader.close();
        });
    });
});
```

Não é difícil imaginar a necessidade de encadear três operações assíncronas em um código real (na prática podem ocorrer muito mais) e é visível que a legibilidade do código se denigre muito rápido. Extrair os callbacks não ajuda muito, pois granulariza demais o código, o que também dificulta a leitura.

Com o objetivo de melhorar isso, surgiu no JavaScript o conceito de promessas (*promises*). A ideia principal é, ao chamar uma operação assíncrona, ao invés de passar um callback como parâmetro, receber um objeto como retorno, e ser capaz de inscrever neste objeto funções que serão chamadas no futuro com o resultado da operação. Veja o exemplo acima, reescrito para usar a versão da API de sistema de arquivos baseada em Promises:

```
const fs = require('fs');
const readline = require('readline');

const reader = readline.createInterface(process.stdin, process.stdout);

fs.promises.readFile('dados.txt', { encoding: 'UTF-8' })
    .then(data => {
```

```
        console.log(data);
        return new Promise((resolve, _) => {
            reader.question('Linha: ', linha => resolve(linha));
        });
    })
    .then(linha => fs.promises.appendFile('dados.txt', linha + '\n'))
    .then(() => reader.close());
```

E como fica o tratamento de erros? Da forma como está implementado no exemplo, o comportamento é exatamente o mesmo da versão anterior, visto que o erro era apenas relançado assim que identificado. Para fazer algo diferente com ele, no entanto, é necessário usar a função `catch` da promessa, ou passar um segundo callback como parâmetro para as chamadas `then`:

```
fs.promises.readFile('dados.txt', { encoding: 'UTF-8' })
    .then(data => {
        console.log(data);
        return new Promise((resolve, _) => {
            reader.question('Linha: ', linha => resolve(linha));
        });
    })
    .then(linha => fs.promises.appendFile('dados.txt', linha + '\n'))
    .then(() => reader.close())
    .catch(err => {
        console.log(`Erro: ${err}`);
        reader.close();
    });
```

A legibilidade do código ficou muito melhor, mas ainda é possível melhorar. Com o intuito de aproximar ainda mais a sintaxe de código assíncrono da sintaxe de código síncrono, duas novas palavras-chave surgiram na linguagem: `async` e `await`. `async` é adicionado em funções, e diz para o Node.js que essa função pode usar a palavra-chave `await` dentro dela. Já a `await` é usada logo antes de chamadas que retornam promessas, e basicamente faz com que a execução interrompa (sem bloquear o loop de eventos) até que a resposta chegue e ela possa prosseguir de onde parou. Veja o exemplo reescrito com o uso dessas palavras-chave:

```
const fs = require('fs');
const readline = require('readline');

const reader = readline.createInterface(process.stdin, process.stdout);

async function main() {
    try {
        const data = await fs.promises.readFile('dados.txt', { encoding: 'UTF-8' });
        console.log(data);
        const linha = await new Promise((resolve, _) => {
            reader.question('Linha: ', linha => resolve(linha));
        });
        await fs.promises.appendFile('dados.txt', linha + '\n');
    } catch (err) {
        console.log(`Erro: ${err}`);
    } finally {
        reader.close();
    }
```

```
}

console.log(main());
```

Agora sim a implementação está bem legível, sem comprometer a performance. Note que só o fato de demarcar a função como `async` já faz ela retornar uma promessa, o que faz sentido se analisar as consequências disso.

[1] https://nodejs.org/api/fs.html

Múltiplos processos

Hoje em dia dificilmente se fala em computadores com um único núcleo de processamento. Com base nisso, como garantir que sua aplicação, principalmente se ela for um servidor, está adequadamente aproveitando todos os recursos da máquina? Com a programação multi-thread, a solução é garantir que o trabalho de processamento fique adequadamente distribuído em threads cujo número seja no mínimo igual à quantidade de processadores disponíveis. Mas e no Node.js?

Considere o seguinte exemplo, e monitore a execução desse código (usando algo como `htop` ou o Gerenciador de Tarefas):

```
const startInMillis = new Date().getTime();
let soma = 0;
for (let i = 0; i < 10000000000; i++) {
    soma += i;
}
console.log(soma);
const endInMillis = new Date().getTime();
const durationInMillis = endInMillis - startInMillis;
console.log(durationInMillis);
```

Note que o processamento fica todo em um único núcleo. Como aproveitar todo o recurso disponível nestes casos? Para esse tipo de situação existe o módulo `cluster`, capaz de criar processos filhos que podem acabar sendo agendados em núcleos diferentes do processador. Veja o exemplo reescrito:

```
const cluster = require('cluster');

if (cluster.isWorker) {

    cluster.worker.on('message', message => {
        let soma = 0;
        for (let i = message.inicio; i < message.fim; i++) {
            soma += i;
        }
        console.log(`Worker: ${soma}`);
        cluster.worker.send(soma);
        cluster.worker.disconnect();
    });

} else {

    let recebidos = 0;
    let soma = 0;
    cluster.on('message', (_, message) => {
```

```
        soma += message;
        recebidos++;
        if (recebidos == NUMERO_DE_PROCESSOS) {
            console.log(`Master: ${soma}`);
        }
    });

    const workers = [];
    const NUMERO_DE_PROCESSOS = 4;
    for (let i = 0; i < NUMERO_DE_PROCESSOS; i++) {
        workers.push(cluster.fork());
    }

    const TOTAL_PARCELAS = 100000000000;
    const PARCELAS_POR_PROCESSO = TOTAL_PARCELAS / NUMERO_DE_PROCESSOS;
    for (let i = 0; i < NUMERO_DE_PROCESSOS; i++) {
        workers[i].send({
            inicio: PARCELAS_POR_PROCESSO * i,
            fim: PARCELAS_POR_PROCESSO * (i + 1)
        });
    }

}
```

Note que dificilmente o desenvolvedor usa esses módulos diretamente, mas é importante entender como os frameworks utilizados fazem o que fazem, isso ajuda na depuração de problemas e na concepção de soluções para problemas não tão comuns.

Servindo HTTP

Ser capaz de escrever utilitários de linha de comando é útil, mas não é nem de longe o único uso do Node.js. Servir como back-end HTTP é um dos principais motivos de sua concepção, e existe suporte nativo para isso [1]. Existem também módulos estáveis para HTTP/2 [2] e HTTPS [3].

Suponha que você deseja implementar uma API de dados para manipulação de tarefas. Cada tarefa possui um identificador, uma data de criação, uma descrição, uma data prevista para conclusão e uma data real de conclusão. As seguintes operações devem ser suportadas pela API:

- Listar as tarefas (apenas o identificador, a descrição e um indicativo de conclusão) filtrando (opcionalmente) por qualquer parte da descrição.
- Buscar os dados completos de uma tarefa específica.
- Cadastrar uma tarefa.
- Alterar a descrição de uma tarefa existente.
- Alterar a data prevista de conclusão de uma tarefa.
- Concluir uma tarefa.
- Reabrir uma tarefa.

Traduzindo para o modelo REST, uma das formas de implementar essa API é através do seguinte conjunto de requisições:

- `GET /tarefas[?filtro=]`. Resposta: `[{ id: number, descricao: string, concluida: boolean }]`.
- `GET /tarefas/{id}`. Resposta: `{ descricao: string, criacao: Date, previsao: Date, conclusao?: Date }`.

- `POST /tarefas`. Corpo: `{ descricao: string, previsao: Date }`.
- `PUT /tarefas/{id}/descricao`. Corpo: `string`.
- `PUT /tarefas/{id}/previsao`. Corpo: `Date`.
- `POST /tarefas/{id}/finalizar`
- `POST /tarefas/{id}/reabrir`

Note que também seria possível implementar um único `PUT` expondo todos os atributos, mas isso transfere muita inteligência de negócio para o front-end, e normalmente não é isso que se deseja.

Antes de iniciar o desenvolvimento da API, escreva o seguinte módulo JavaScript, que fará o papel de repositório de dados. Como ainda não existe uma base de dados disponível, os dados serão voláteis, ou seja, serão perdidos entre execuções da aplicação. Coloque o código em um arquivo chamado `tarefas/tarefas-modelo.js`:

```
let ID = 1;
const REGISTROS = [];


class Tarefa {
  constructor(descricao, previsao) {
    this.id = ID;
    ID++;
    this.descricao = descricao;
    this.previsao = previsao;
    this.criacao = new Date();
    this.conclusao = null;
  }
};
module.exports.Tarefa = Tarefa;


REGISTROS.push(new Tarefa('Terminar essa API', null));
REGISTROS.push(new Tarefa('Ir embora', new Date()));


module.exports.listar = function (filtro) {
  if (!filtro) {
    return REGISTROS;
  } else {
    const lower = filtro.toLowerCase();
    return REGISTROS.filter(x => x.descricao.toLowerCase().indexOf(lower) >= 0);
  }
};


function buscarPorId (id) {
  return REGISTROS.filter(x => x.id == id)[0];
}
module.exports.buscarPorId = buscarPorId;


module.exports.cadastrar = function (tarefa) {
  REGISTROS.push(tarefa);
```

```
};


module.exports.alterarDescricao = function (id, descricao) {
  buscarPorId(id).descricao = descricao;
};


module.exports.alterarPrevisao = function (id, previsao) {
  buscarPorId(id).previsao = previsao;
};


module.exports.concluir = function (id) {
  buscarPorId(id).conclusao = new Date();
};


module.exports.reabrir = function (id) {
  buscarPorId(id).conclusao = null;
};


module.exports.excluir = function (id) {
  const registro = buscarPorId(id);
  const idx = REGISTROS.indexOf(registro);
  REGISTROS.splice(idx, 1);
}
```

Note que esses métodos não estão efetuando nenhum tipo de validação nos dados, eles servem apenas para fins ilustrativos.

Teste esse módulo usando o REPL do Node. Para isso, bbasta executar o comando `node`, importar o módulo via `require` e usar os métodos dele, ex:

```
$ node
> const tarefas = require('./tarefas/tarefas-modelo');
> tarefas.buscar();
> tarefas.excluir(1);
```

Com o módulo de acesso aos dados concluído, é possível prosseguir para o desenvolvimento da API HTTP que expõe as funcionalidades desse módulo. Crie um arquivo chamado `main.js`, com o seguinte código:

```
const http = require('http');


const server = http.createServer((req, res) => {
    let data = "";
    req.on('data', chunk => {
        data += chunk;
    });
    req.on('end', () => {
```

```
        console.log(data);
        res.write(`Olá, ${data}!`);
        res.end();
    });
});
server.listen(8080);
```

Teste usando qualquer ferramenta de execução de chamadas HTTP, como o cUrl ou o Postman.

Prossiga agora para os *endpoints* de listagem, detalhamento e cadastro. Pensando na modularização do código, crie um arquivo `tarefas/tarefas-api.js`, com o seguinte conteúdo:

```
const tarefas = require('./tarefas-modelo');


function get(req, res, data, path, queryParams) {
    res.write('get');
}


function getById(req, res, data, path, queryParams) {
    res.write('getById');
}


function post(req, res, data, path, queryParams) {
    res.write('post');
}


module.exports.processar = function (req, res, data, path, queryParams) {
    const method = req.method;
    if (method == 'GET' && path.startsWith('/tarefas/')) {
        getById(req, res, data, path, queryParams);
    } else if (method == 'GET' && path === '/tarefas') {
        get(req, res, data, path, queryParams);
    } else if (method == 'POST' && path === '/tarefas') {
        post(req, res, data, path, queryParams);
    } else {
        res.statusCode = 404;
    }
}
```

E ajuste o arquivo `main.js` dessa maneira:

```
const http = require('http');
const querystring = require('querystring');
const tarefas = require('./tarefas/tarefas-api');


const server = http.createServer((req, res) => {
    let data = "";
    req.on('data', chunk => {
```

```
        data += chunk;
    });
    req.on('end', () => {
        const url = req.url.split('?');
        const path = url[0];
        const queryParams = url.length > 1 ? querystring.parse(url[1]) : {};
        if (path.startsWith('/tarefas')) {
            tarefas.processar(req, res, data, path, queryParams);
        } else {
            res.statusCode = 404;
        }
        res.write('\n'+data);
        res.write('\n'+path);
        res.write('\n'+JSON.stringify(queryParams));
        res.end();
    });
});
server.listen(8080);
```

Teste o resultado usando cUrl ou Postman. Uma vez entendido o funcionamento, termine a implementação dos métodos no módulo `tarefas-api.js`:

```
const tarefas = require('./tarefas-modelo');


function get(req, res, data, path, queryParams) {
    const filtro = queryParams['filtro'];
    const registros = tarefas.listar(filtro);
    res.write(JSON.stringify(registros.map(x => {
        return {
            id: x.id,
            descricao: x.descricao,
            concluida: x.conclusao != null
        };
    })));
}


function getById(req, res, data, path, queryParams) {
    const partes = path.split('/');
    const id = parseInt(partes[2]);
    const registro = tarefas.buscarPorId(id);
    res.write(JSON.stringify({
        descricao: registro.descricao,
        criacao: registro.criacao,
        conclusao: registro.conclusao,
        previsao: registro.previsao
    }));
}


function post(req, res, data, path, queryParams) {
```

```
    const obj = JSON.parse(data);
    const descricao = obj['descricao'];
    const previsao = obj['previsao'] ?
        new Date(obj['previsao']) : null;
    const registro = new tarefas.Tarefa(descricao, previsao);
    tarefas.cadastrar(registro);
}


module.exports.processar = function (req, res, data, path, queryParams) {
    const method = req.method;
    if (method == 'GET' && path.startsWith('/tarefas/')) {
        getById(req, res, data, path, queryParams);
    } else if (method == 'GET' && path === '/tarefas') {
        get(req, res, data, path, queryParams);
    } else if (method == 'POST' && path === '/tarefas') {
        post(req, res, data, path, queryParams);
    } else {
        res.statusCode = 404;
    }
}
```

Antes de testar, remova as escritas de teste deixadas no arquivo `main.js`, deixando-o dessa forma:

```
const http = require('http');
const querystring = require('querystring');
const tarefas = require('./tarefas/tarefas-api');


const server = http.createServer((req, res) => {
    let data = "";
    req.on('data', chunk => {
        data += chunk;
    });
    req.on('end', () => {
        const url = req.url.split('?');
        const path = url[0];
        const queryParams = url.length > 1 ? querystring.parse(url[1]) : {};
        if (path.startsWith('/tarefas')) {
            tarefas.processar(req, res, data, path, queryParams);
        } else {
            res.statusCode = 404;
        }
        res.end();
    });
});
server.listen(8080);
```

Finalize agora os outros 5 endpoints, modificando o arquivo `tarefas-api.js`:

```
const tarefas = require('./tarefas-modelo');
```

```
function extrairId(path) {
    const partes = path.split('/');
    return parseInt(partes[2]);
}


function get(req, res, data, path, queryParams) {
    const filtro = queryParams['filtro'];
    const registros = tarefas.listar(filtro);
    res.write(JSON.stringify(registros.map(x => {
        return {
            id: x.id,
            descricao: x.descricao,
            concluida: x.conclusao != null
        };
    })));
}


function getById(req, res, data, path, queryParams) {
    const id = extrairId(path);
    const registro = tarefas.buscarPorId(id);
    res.write(JSON.stringify({
        descricao: registro.descricao,
        criacao: registro.criacao,
        conclusao: registro.conclusao,
        previsao: registro.previsao
    }));
}


function post(req, res, data, path, queryParams) {
    const obj = JSON.parse(data);
    const descricao = obj['descricao'];
    const previsao = obj['previsao'] ?
        new Date(obj['previsao']) : null;
    const registro = new tarefas.Tarefa(descricao, previsao);
    tarefas.cadastrar(registro);
}


function del(req, res, data, path, queryParams) {
    const id = extrairId(path);
    tarefas.excluir(id);
}


function finalizar(req, res, data, path, queryParams) {
    const id = extrairId(path);
    tarefas.concluir(id);
}
```

```
function reabrir(req, res, data, path, queryParams) {
    const id = extrairId(path);
    tarefas.reabrir(id);
}


function alterarDescricao(req, res, data, path, queryParams) {
    const id = extrairId(path);
    const descricao = data;
    tarefas.alterarDescricao(id, descricao);
}


function alterarPrevisao(req, res, data, path, queryParams) {
    const id = extrairId(path);
    const previsao = new Date(data);
    tarefas.alterarPrevisao(id, previsao);
}


module.exports.processar = function (req, res, data, path, queryParams) {
    const method = req.method;
    if (method == 'GET' && path.startsWith('/tarefas/')) {
        getById(req, res, data, path, queryParams);
    } else if (method == 'GET' && path === '/tarefas') {
        get(req, res, data, path, queryParams);
    } else if (method == 'POST' && path === '/tarefas') {
        post(req, res, data, path, queryParams);
    } else if (method == 'DELETE' && path.startsWith('/tarefas/')) {
        del(req, res, data, path, queryParams);
    } else if (method == 'POST' && path.endsWith('/finalizar')) {
        finalizar(req, res, data, path, queryParams);
    } else if (method == 'POST' && path.endsWith('/reabrir')) {
        reabrir(req, res, data, path, queryParams);
    } else if (method == 'PUT' && path.endsWith('/descricao')) {
        alterarDescricao(req, res, data, path, queryParams);
    } else if (method == 'PUT' && path.endsWith('/previsao')) {
        alterarPrevisao(req, res, data, path, queryParams);
    } else {
        res.statusCode = 404;
    }
}
```

Teste novamente. Note que qualquer erro causa indisponibilidade completa da aplicação. É possível melhorar isso no arquivo `main.js`, envolvendo todo o processo interno em um `try/catch`:

```
req.on('end', () => {
    try {
        const url = req.url.split('?');
        const path = url[0];
        const queryParams = url.length > 1 ? querystring.parse(url[1]) : {};
```

```
        if (path.startsWith('/tarefas')) {
            tarefas.processar(req, res, data, path, queryParams);
        } else {
            res.statusCode = 404;
        }
    } catch (err) {
        console.error(err);
        res.statusCode = 500;
    } finally {
        res.end();
    }
});
```

Note que isso só funciona pois o processamento interno foi todo desenvolvido de forma síncrona. Na prática isso não iria acontecer, e além desse `try/catch` uma estrutura de promessas teria que ser usada para capturar todos os possíveis casos de tabela.

Uma outra pequena melhoria possível é a escrita do Header de resposta `Content-Type: application/json` , isso ajuda a ferramentas como o Postman (e as bibliotecas de HTTP dos frameworks web) a entenderem como processar a resposta. Isso pode ser feito de modo genérico direto no `main.js` :

```
req.on('end', () => {
    try {
        res.setHeader('Content-Type', 'application/json');
        const url = req.url.split('?');
        const path = url[0];
```

[1] https://nodejs.org/api/http.html

[2] https://nodejs.org/api/http2.html

[3] https://nodejs.org/api/https.html

Consumindo a API usando um front-end Angular

Para finalizar o tópico, construir um front-end simples, usando Angular, que consome a API construída parece uma boa ideia. Comece garantindo que possui o `@angular/cli` disponível na instalação do Node.js:

```
$ ng --version
```

Caso não tenha (provavelmente esse será o caso por conta da instalação do `nvm`), instale-o com o seguinte comando:

```
$ npm install -g @angular/cli
```

Inicie um novo projeto com o comando abaixo (não há a necessidade de adicionar roteamento para este exemplo). Dê o nome `tarefas` ao projeto:

```
$ ng new
```

Teste se tudo funcionou com o comando abaixo:

```
$ cd tarefas/
$ ng serve
```

E acesse `http://localhost:4200/` no navegador. Abra um outro terminal e deixe a API executando.

Modifique o arquivo `app.component.html` dessa forma:

```
<table>
  <thead>
    <tr>
      <th>Tarefa</th>
      <th>Previsão</th>
      <th>Concluída?</th>
      <th>Ações</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let tarefa of tarefas">
      <td>{{tarefa.descricao}}</td>
      <td>{{tarefa.previsao | date:"dd/MM/yyyy HH:mm"}}</td>
      <td>{{tarefa.concluida ? 'Sim' : 'Não'}}</td>
      <td>
        <button (click)="concluir(tarefa)" *ngIf="!tarefa.concluida">Concluir</button>
        <button (click)="reabrir(tarefa)" *ngIf="tarefa.concluida">Reabrir</button>
        <button (click)="remover(tarefa)">Excluir</button>
      </td>
    </tr>
  </tbody>
</table>

<form [formGroup]="formGroup" (ngSubmit)="cadastrar()">
  <p>
    Descrição:
    <input formControlName="descricao">
  </p>
  <p>
    Previsão:
    <input type="date" formControlName="previsao">
  </p>
  <p>
    <button>Cadastrar</button>
  </p>
</form>
```

Instale também o `@angular/forms` no projeto, usando o comando abaixo:

```
$ npm install @angular/forms
```

E importe o módulo de formulários reativos na classe `AppModule`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
```

```
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Agora implemente os atributos para a tabela e o formulário na classe `AppComponent`, por enquanto sem nenhuma comunicação com a API:

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';


interface Tarefa {
  id: number;
  descricao: string;
  previsao?: Date;
  concluida: boolean;
}


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  tarefas: Tarefa[] = [
    {
      id: 1,
      concluida: false,
      descricao: 'Tarefa 1',
      previsao: new Date()
    }
  ];

  descricao?: string;
  previsao?: Date;

  formGroup: FormGroup;
```

```
  constructor(private builder: FormBuilder) {
  }

  ngOnInit() {
    this.formGroup = this.builder.group({
      descricao: ['', Validators.required],
      previsao: [null]
    });
  }

}
```

Execute a aplicação e garanta que ela está funcionando (exceto os cliques nos botões).

Para implementar as integrações com a API de dados, comece instalando o módulo `@angular/http`:

```
$ npm install @angular/http
```

Depois importe o módulo na classe `AppModule`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Importe agora o `HttpClient` na classe `AppComponent` e efetue a implementação da chamada de API para buscar tarefas:

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { HttpClient } from '@angular/common/http';


interface Tarefa {
  id: number;
  descricao: string;
  previsao?: Date;
```

```
  concluida: boolean;
}


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  tarefas: Tarefa[] = [];

  formGroup: FormGroup;

  constructor(private builder: FormBuilder,
              private http: HttpClient) {
  }

  ngOnInit() {
    this.formGroup = this.builder.group({
      descricao: ['', Validators.required],
      previsao: [null]
    });
    this.http.get('http://localhost:8080/tarefas').subscribe(tarefas => {
      this.tarefas = <Tarefa[]>tarefas;
    });
  }

}
```

Ao testar, o navegador não vai permitir a finalização da chamada, devido a um mecanismo de segurança chamado CORS (Cross-Origin Resource Sharing). Existem muitas maneiras de lidar com CORS, a mais fácil (e insegura) delas é adicionar um header em todas as respostas, permitindo que qualquer host efetue a chamada. Faça isso no arquivo `main.js` (note que isso é feito no back-end, e não no front-end):

```
req.on('data', chunk => {
    data += chunk;
});
req.on('end', () => {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Headers', '*');
    res.setHeader('Access-Control-Allow-Methods', 'OPTIONS, POST, GET, DELETE, PUT');
    if (req.method == 'OPTIONS') {
        res.end();
        return;
    }
    try {
        res.setHeader('Content-Type', 'application/json');
        const url = req.url.split('?');
        const path = url[0];
```

Agora termine a implementação dos outros comportamentos do gestor de tarefas, na classe `AppComponent`:

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { HttpClient } from '@angular/common/http';


interface Tarefa {
  id: number;
  descricao: string;
  previsao?: Date;
  concluida: boolean;
}


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  tarefas: Tarefa[] = [];

  formGroup: FormGroup;

  constructor(private builder: FormBuilder,
              private http: HttpClient) {
  }

  ngOnInit() {
    this.formGroup = this.builder.group({
      descricao: ['', Validators.required],
      previsao: [null]
    });
    this.carregar();
  }

  private carregar() {
    this.http.get('http://localhost:8080/tarefas').subscribe(tarefas => {
      this.tarefas = <Tarefa[]>tarefas;
    });
  }

  concluir(tarefa: Tarefa) {
    this.http.post(`http://localhost:8080/tarefas/${tarefa.id}/finalizar`,
{}).subscribe(() => {
      tarefa.concluida = true;
    });
  }
```

```
  reabrir(tarefa: Tarefa) {
    this.http.post(`http://localhost:8080/tarefas/${tarefa.id}/reabrir`,
{}).subscribe(() => {
      tarefa.concluida = false;
    });
  }

  remover(tarefa: Tarefa) {
    this.http.delete(`http://localhost:8080/tarefas/${tarefa.id}`).subscribe(() => {
      this.tarefas.splice(this.tarefas.indexOf(tarefa), 1);
    });
  }

  cadastrar() {
    const dados = this.formGroup.value;
    this.http.post(`http://localhost:8080/tarefas`, dados).subscribe(() => {
      this.formGroup.setValue({
        descricao: "",
        previsao: null
      });
      this.carregar();
    });
  }

}
```

Nota: em um projeto real seriam utilizadas bibliotecas de componentes (como a `@angular/material`), uma componentização mais granular (ao invés de implementar tudo no `AppComponent`) e camadas de serviços para isolar as chamadas HTTP dos componentes. No entanto, nada disso é essencial para passar a mensagem que se espera nesse exemplo.