Resumo teórico

Neste documento consta um resumo teórico de todo o material da disciplina. Use-o como complemento nos estudos, sendo esta uma visão mais ampla e o material original uma visão mais prática dos mesmos conceitos.

1. Introdução ao Node.js

1.1 Conceitos básicos de aplicações web

O que é uma aplicação web?

> *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.*

1.1.1 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).

1.1.2 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. 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*.
- **Cabeçalhos (Headers):** são atributos da requisição que podem ser usados para diversos fins.
- **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.

1.1.3 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.

1.1.4 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.

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.

1.1.5 Back-end (lado do servidor)

Composto por um *servidor web*, capaz de aceitar as requisições HTTP, e normalmente por um *framework web*. É aqui que entra o Node.js e todo o seu ecossistema.

1.1.6 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?

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.).

1.2 Node.js

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.

1.3 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`.

1.4 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+).

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

1.5 Sistema de arquivos, Promises e Async

O Node.js fornece uma API para manipulação de sistema de arquivos, sendo que a versão padrão efetua as operações de maneira síncrona. 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 carregamento 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).

Não é difícil perceber que a codificação assíncrona fica rapidamente complexa de entender. A necessidade de encadear três operações assíncronas ou mais em um código real (na prática podem ocorrer muito mais) é comum, e a legibilidade do código se denigre muito rápido nessas situações. 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.

A legibilidade do código fica 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.

Note que só o fato de demarcar uma função como `async` já faz ela retornar uma promessa.

1.6 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?

Para esse tipo de situação existe o módulo `cluster`, capaz de criar processos filhos que podem acabar sendo agendados em núcleos diferentes do processador.

1.7 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. Existem também módulos estáveis para HTTP/2 e HTTPS.

2. Express

2.1 Problemas enfrentados com o uso do módulo HTTP nativo do Node.js

Durante o desenvolvimento de uma API HTTP usando o módulo nativo do Node.js, algumas coisas claramente se mostraram difíceis de resolver, produzindo código frágil e mal organizado, como por exemplo:

- *Processamento do corpo da requisição:* foi necessário tratar o recebimento dos dados em "chunks". A implementação que foi feita (concatenação de strings) só funciona com dados textuais, não funcionaria para upload de arquivos, por exemplo. Se a entrada for JSON, a conversão para um objeto JavaScript e validação dos dados de entrada também foi feita manualmente, o que não é o ideal.
- *Roteamento da requisição:* foi usada uma grande carga de manipulação de strings para rotear as chamadas. Isso rapidamente deixa o código poluído e difícil de entender.
- *Tratamento de casos de erro:* como o módulo HTTP não oferece nada com relação a tratamento de erros, foi necessário codificar com cuidado para garantir que toda requisição tivesse uma resposta, mesmo em casos de exceção. Isso fica ainda pior se considerar o fato de que o processamento costuma ser assíncrono (baseado em callbacks/Promises).

Além disso, algumas coisas nem chegaram a ser abordadas, como:

- *Disponibilização de arquivos estáticos (HTML/CSS/JavaScript de front-end):* não é a única maneira, mas uma forma bem simples de disponibilizar sua aplicação para os usuários é através do próprio servidor de back-end.
- *Upload/download de arquivos*
- *Segurança:* como trabalhar a autenticação dos usuários da aplicação? Quais os mecanismos disponíveis e quando usar cada um deles?

2.2 Node Package Manager (npm)

É um gestor de pacotes inicialmente projetado para o ecossistema Node, mas que hoje atende a praticamente todos os projetos que usam JavaScript.

2.3 Semantic Versioning (SemVer)

Todo pacote Node que é publicado em um repositório (como o npmjs.com, por exemplo) precisa ter um nome e uma versão. Essa versão precisa ser compatível com o conceito de versão semântica (SemVer). De forma resumida, versionamento semântico é composto por três números:

```
1.2.3
```

O primeiro é a versão `MAJOR`. O segundo, a `MINOR`, e o terceiro, `PATCH`. A versão `MAJOR` só deve mudar quando a biblioteca alterar sua *interface externa* de modo *incompatível* com as versões anteriores, por exemplo: alteração de assinatura de método de modo que chamadas existentes deixem de funcionar, remoção completa de um método. A `MINOR` é incrementada quando houver *adição de funcionalidade*, sem quebrar código existente. Por fim, a versão `PATCH` só é incrementada quando existirem apenas correções de bugs, melhorias de performance e coisas do tipo em funcionalidades existentes.

2.4 Express Generator

É uma ferramenta de geração de código disponível para usuários de Express. Ela é capaz de gerar um esqueleto de projeto Node, opcionalmente adicionando uma camada de visão (motor de template HTML).

2.5 Middlewares

Um Middleware, dentro do Express, nada mais é uma função que será chamada para todas as requisições (dentro do escopo onde o middleware foi instalado), capaz de enriquecer os dados da requisição/resposta, finalizar uma resposta automaticamente (tratamento de erros, segurança, etc), dentre outras. Uma forma de ver é comparar com o conceito de `plug-in`.

Por mais que seja possível e até incentivado o desenvolvimento de middlewares customizados, o mais comum é que pacotes de funcionalidades externos sejam adicionados dessa forma, como é o caso do middleware `express.static` que foi apresentado na seção anterior.

Alguns middlewares úteis oferecidos nativamente no Express são: `json` e `static`. Vários outros podem ser adicionados via NPM, como é o caso do `formidable`, `jwt`, `cookie-parser` etc.

2.6 Routers

Routers são uma espécie de mini aplicação Express, e seu intuito principal é permitir a modularização do código do projeto. Todo Router, internamente, é tratado como um middleware pela aplicação Express.

2.7 Validação de dados de entrada

Uma das características discutidas no início deste tópico foi a validação de dados de entrada. O framework Express não possui nada nativo com relação a isso (exceto a conversão automática de `json`, que já vem sendo usada nas seções anteriores), mas a comunidade criou várias bibliotecas que permitem adicionar esse tipo de validação no seu projeto. Uma delas é a `express-validator`.

2.8 Autenticação e autorização

Boa parte das aplicações não triviais possui a necessidade de autenticar seus usuários, e autorizar o acesso a funcionalidades e dados. A autenticação pode ser feita, por exemplo, através de um formulário que solicita usuário e senha, e a autorização permitindo apenas que o usuário visualize e altere os dados que ele próprio cadastrou. Um exemplo mais complicado de autenticação seria por

exemplo integrar com o Google Accounts, e um exemplo de autorização seria permitir que um usuário compartilhasse dados com outros, que passariam a ter acesso àqueles dados e talvez até a modificá-los.

Existem muitas técnicas para implementar autorização em chamadas de API, mas normalmente elas giram em torno de passar no header de requisição `Authorization` um valor, seja ele qual for, capaz de provar para o servidor que quem submeteu aquela requisição é quem diz ser.

No ecossistema Express, uma implementação de autenticação bem popular é o uso de Json Web Tokens (JWT) através das bibliotecas `express-jwt` (middleware de autenticação) e `jsonwebtoken` (geração de tokens).

2.9 Trabalhando com uploads

Express não oferece suporte nativo para processamento de requisições com conteúdo binário, mas esse suporte pode ser facilmente adicionado através de middlewares de terceiro, como o `express-fileupload`.

3. Knex

3.1 Da escolha do tipo de armazenamento de dados

Quando se analisa as possíveis escolhas para persistência de dados em um projeto, o modelo mais simples e difundido é o uso de uma base de dados relacional. O fato de possuir um esquema robusto, ser flexível, ter muito conhecimento difundido na Internet e escalar apropriadamente para a grande maioria dos cenários a qual é subjulgada faz desse tipo de base de dados a escolha padrão. É quando o cenário é específico demais, seja pelo volume de informações, modelo muito variado, características de acesso recursivas dentre outras peculiaridades que outros modelos como o baseado em documentos, chave-valor ou grafos surgem para atender a demanda.

Com o tempo, não é raro que projetos passem a usar mais de um tipo de base de dados simultaneamente.

3.2 Armazenamento de hashes de senha

Quando se discute sobre armazenamento de senhas, é senso comum *não armazená-las* como texto plano, pois isso é uma falha gravíssima de segurança. O que se discute com menos frequência é a técnica de *espalhamento/hash* usada para proteger o valor antes de armazená-lo, e que tipo de dado utilizar na base.

Com relação às possíveis técnicas, uma combinação bastante aceita é a seguinte: use uma função espalhadora forte, como a SHA-256, *repetidamente* sobre o valor de entrada *concatenado* com uma palavra segredo, chamada de *salt*.

Isso produz um vetor de 32 bytes que pode ser armazenado no banco, mas como armazená-lo? Dentre os possíveis formatos, três se destacam:

- Uso de uma coluna de texto de 44 posições e armazenar o valor codificado como base64.
- Uso de uma coluna de texto de 64 posições e armazenar o valor codificado como hexadecimal.
- Uso de uma coluna de vetor de bytes de 32 posições e armazenar o valor sem codificação.

Normalmente se usa uma das primeiras duas opções, pois manipular cadeias de caracteres costuma ser mais conveniente e o ganho de performance/armazenamento não é tão expressivo, mas a terceira é a que mais respeita o tipo fundamental do dado.

3.3 Driver, Query Builder ou ORM?

3.3.1 Driver baixo nível

Para scripts simples, projetos muito pequenos ou que exijam máxima atenção com performance, a melhor opção pode ser usar uma biblioteca que simplesmente encapsula a conexão com o banco de dados, não ajudando em muito além disso.

3.3.2 ORM

Com a popularização da programação orientada a objetos, naturalmente surgiram ideias para facilitar o uso de bases de dados relacionais neste meio. Uma das vertentes mais agressivas propõe o maior nível de abstração possível, e é chamada de ORM (mapeamento objeto-relacional). A ideia do ORM é configurar previamente um conjunto de classes (ou equivalente) e atributos de modo a representarem o modelo de tabelas/colunas do banco de dados em um grafo de objetos. Com base nessa configuração, a ferramenta de ORM é capaz de fornecer APIs de consulta e manipulação dos dados de maneira simplificada.

A primeira vista essa parece a melhor opção, mas como o mapeamento entre orientação a objetos e modelo de dados relacional é complexo e incompleto, o resultado quase sempre é o sacrifício de performance para o ganho de simplicidade e produtividade no desenvolvimento. Portanto o uso dessa técnica deve ser feito com cuidado, e ela nem de longe exime o desenvolvedor de possuir um conhecimento aprofundado em modelos relacionais.

3.3.3 Query Builder

A terceira opção para comunicação com o banco de dados é um intermediário entre as outras duas. Ao invés de escrever os comandos SQL manualmente, como na primeira opção, ou configurar uma camada de abstração orientada a objetos, como na segunda, existem bibliotecas que ajudam a montar dinamicamente os comandos SQL sem remover a essência relacional do processo. Esse é o fundamento por tras dos Query Builders, como o Knex.

3.4 Evolução do modelo

Suponha que sua aplicação está em produção, sendo utilizada por vários usuários. Em determinado momento você decide evoluí-la, adicionando uma funcionalidade nova. Isso vai significar, em algum momento, a adição de novas tabelas no banco de dados. Como gerenciar esse tipo de evolução? "Lembrar" de executar scripts SQL é suscetível a erros entre ambientes, e a última coisa que você deseja é diferenças entre os ambientes de desenvolvimento/testes e produção.

Para resolver este ponto existe o conceito de "migrações de modelo de dados". Basicamente cada "migração" é um script (que pode ser SQL, ou código escrito em alguma biblioteca de interface com o banco como o próprio Knex) que atualiza o banco de dados para que ele fique compatível com a versão mais recente do código fonte. Essa migração pode ser disparada via CLI ou por dentro do código da própria aplicação. O importante é que ela faça parte do processo de integração/entrega contínua sem exigir nenhum tipo de intervenção humana.

Vale observar que alguns projetos decidem *não* oferecer suporte para rollback de migrações (o método `down`). Garantir a qualidade desses módulos é custoso e eles praticamente nunca são usados, portanto o argumento principal é não implementá-los e atuar nos rollbacks manualmente quando algum imprevisto ocorrer nos ambientes produtivos.

3.5 UUID como chave primária

É possível utilizar colunas do tipo `uuid` ao invés de inteiros com auto-incremento na definição de chaves-primárias das tabelas. Essa é uma alternativa interessante para identificação única global de registros, amigável com arquiteturas compostas por várias fontes de dados distintas (o UUID pode ser gerado em qualquer lugar e distribuído para todas as fontes de dados envolvidas).

3.6 Pool de conexões

Independentemente da estratégia adotada para se comunicar com o banco de dados, é interessante otimizar o seu pool de conexões, a fim de garantir uma boa performance e aproveitamento de recursos do servidor. Por padrão o Knex configura um mínimo de 2 conexões e um máximo de 10, mas isso pode ser facilmente modificado.

Pools muito pequenos podem causar gargalos, e pools muito grandes podem usar mais do banco de dados do que lhe é necessário. Encontrar o balanço correto é fruto de monitoramento e análise constante do ambiente em execução.

4. Stateless vs. Stateful

Uma aplicação convencional normalmente usa o conceito de "sessão do usuário" para registrar informações como sua identificação, seu carrinho de compras etc. Essa "sessão do usuário" normalmente é implementada da seguinte forma:

- Assim que uma requisição de um usuário desconhecido chega, um Cookie de resposta com um identificador aleatório é gerado e enviado.
- Todas as futuras requisições daquela mesma sessão vão enviar esse Cookie, permitindo que o servidor correlacione as requisições como oriundas do mesmo utilizador.
- Qualquer tipo de informação pode ser armazenado individualmente para cada sessão, o que ocorre normalmente em memória, mas pode ser aplicado o mesmo conceito para qualquer solução de armazenamento.

Imagine que sua aplicação necessite escalar horizontalmente, ou está adotando uma topologia de microsserviços com times autônomos. Como garantir que um mesmo usuário veja os mesmos dados de sessão nesse caso? Existem duas possíveis soluções, o uso de *Sticky session* ou repositórios de sessão persistentes.

4.1 Sticky Session

Uma das maneiras mais fáceis de garantir que um usuário sempre visualize os mesmos dados é fazer com que, uma vez atendido por um servidor, ele sempre seja atendido pelo mesmo servidor, durante todo o tempo de sua sessão. Virtualmente todas as soluções de balanceamento de carga/proxy reverso vão suportar algo nesse sentido.

O prejuízo dessa solução é que a resiliência da aplicação é afetada, pois mesmo tendo 100 servidores, a queda de qualquer um deles significa que todos os usuários que estavam sendo atendidos por aquele servidor em específico vão enfrentar problemas.

4.2 Sessão persistente

Ao invés de usar um repositório em memória, é possível usar um repositório mais persistente, como por exemplo armazenamento em um banco de dados.

O prejuízo dessa solução é que esse repositório passa a ser um gargalo, visto que todas as requisições passam a consultá-lo para carregar e armazenar os dados daquele usuário, sendo que nem todos os

seus serviços de back-end precisam disso e quando precisam, dificilmente precisam da sessão completa.

4.3 Stateless

Com base nesses argumentos, sugere-se dar preferência para o desenvolvimento de APIs stateless, que não usam sessão de modo algum, armazenando as informações de uso diretamente em banco de dados ou soluções similares. Combinada com uma estratégia de autenticação também stateless como JWT, essa arquitetura providencia performance, escalabilidade e separação de responsabilidades até mesmo em arquiteturas baseadas em microsserviços.

5. Cache

O que é Cache? De acordo com a Wikipedia:

> *Na área da computação, cache é um dispositivo de acesso rápido, interno a um sistema, que serve de intermediário entre um operador de um processo e o dispositivo de armazenamento ao qual esse operador acede. A principal vantagem na utilização de um cache consiste em evitar o acesso ao dispositivo de armazenamento - que pode ser demorado -, armazenando os dados em meios de acesso mais rápidos.*

No âmbito de uma aplicação web, esse conceito pode ser aplicado em várias etapas, as quais serão discutidas individualmente neste material. São elas:

- Armazenamento em memória client-side do navegador do usuário.
- Uso de cabeçalhos de manipulação de cache suportados pela própria especificação HTTP. Este pode ser entendido como cache HTTP client-side.
- Na camada de modelo (ou qualquer outra camada do back-end), através do uso de *memoização*.

5.1 Memória client-side (navegador)

Imagine que sua aplicação tenha um processamento considerável para mostrar uma lista de informações na página inicial. Não há a necessidade de atualizar esses dados a cada vez que o usuário acessa essa tela, logo uma estratégia de cache pode poupar muitos recursos dos seus servidores.

5.2 Cabeçalhos HTTP (também client-side)

Além da utilização de estruturas JavaScript, é possível utilizar o suporte existente no próprio protocolo HTTP para cache. Ao inspecionar os cabeçalhos de resposta de uma requisição que usa esse mecanismo, perceberá que o cabeçalho `Cache-Control` é responsável por este papel.

5.3 Memoização server-side

Uma das maneiras mais fáceis de explicar o conceito de memoização é, sem dúvida, a aplicação dela no cálculo do enésimo número da sequência de Fibonacci. Considere a seguinte implementação JavaScript:

```
console.log(fib(43));

function fib(n) {
    if (n <= 1) {
```

```
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Na máquina onde esse código foi escrito, ele demora em média 4s para calcular o 43º número da sequência. Por qual motivo existe essa demora? Repare que o algoritmo calcula várias vezes o mesmo valor, em uma estrutura de árvore:

```
fib(5)
  fib(4)
    fib(3)
      fib(2)
        fib(1)
        fib(0)
      fib(1)
    fib(2)
      fib(1)
      fib(0)
  fib(3)
    fib(2)
      fib(1)
      fib(0)
    fib(1)
```

Veja quantas vezes o `fib(2)` foi calculado, por exemplo. É fácil ver que conforme o número aumenta, essa situação só piora.

Mas onde a memoização entra nesse cenário? Imagine investir um pouco de *espaço em memória* para *memorizar* os cálculos e *reutilizá-los* quando solicitados novamente no futuro. Veja:

```
const memory = {};
console.log(fib(1000));

function fib(n) {
    if (!memory[n]) {
        if (n <= 1) {
            memory[n] = n;
        } else {
            memory[n] = fib(n - 1) + fib(n - 2);
        }
    }
    return memory[n];
}
```

Note que agora o limitador passa a ser o tamanho da pilha de chamadas, e não mais o tempo de processamento.

Além disso, é importante observar que a memoização só pode ser utilizada se a função for *pura*, ou seja, retorna sempre o mesmo resultado dados os mesmos parâmetros.