**Node.js e Express**

Node.js é um ambiente de execução de javascript para back-end fora de um web browser.

Pode ser usado como um servidor (seja de arquivos, de host, etc...) e também como API e/ou gerenciador de rotas.

Node é performático, assíncrono e eficiente, já que praticamente nunca bloqueia processos, já que não realiza I/Os diretamente (operações de dados entre disco e RAM).

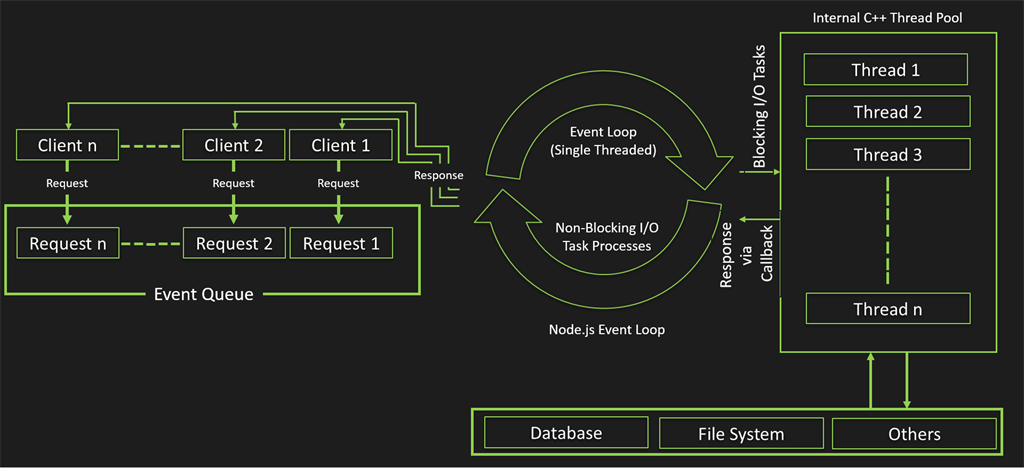
Diferente de outras aplicações para servidores, o Node executa em thread única, sendo gerenciada pelo seu próprio Event Loop. Isso permite muito mais conexões e requisições simultâneas, mas em contrapartida muitas requisições de computação pesada ao mesmo tempo, na mesma thread, podem derrubar o serviço node.

Quando instalamos o Node.js, junto vem o NPM (Node Package Manager), que gerencia pacotes de plugins e bibliotecas que podem ser adicionadas ao nosso projeto Node.js. Um dos mais usados é o Express.

Node tem a grande vantagem de usar javascript, o que permite o trafego de dados, via JSON, ser simples e sem nenhuma necessidade de conversão.

**Vantagens:** Leve, ter o NPM, evita bloqueios de I/O, escalonável, fácil integração com javascript client-side.

**Event Loop do Node.js:**



Quando o server recebe um request, ele é colocado no Event Queue, assim que é “chamado” para o event loop, o processo será servido, e em caso de ser necessário operações de I/O, esse processo é enviado para uma Pool interna de C++, ela consegue executar diversas operações ao mesmo tempo em diversos threads. Assim, o event loop e o node ficam livres para executar non-I/O e colocar mais requisições no loop, sem trancar a execução.

Quando essas operações terminam no pool C++, seus callbacks são retornados para o event loop, que por sua vez, devolve para o client que fez a requisição.

**Exemplo simples de código para subir um server Node**

*const http = require('http');*   
*const hostname = '127.0.0.1';*  
*const port = 3000;*  
  
*const server = http.createServer((req, res) => {*   
 *res.statusCode = 200; res.setHeader('Content-Type', 'text/plain');*   
 *res.end('Hello World');*  
*});*

*server.listen(port, hostname, () => {*   
 *console.log(`Server running at http://${hostname}:${port}/`);*  
*});*

Na primeira linha requisitamos a biblioteca responsavel por protocolos http  
abaixo setamos o host para acessar o servidor e a porta. Então criamos o server, que irá retornar “Hello World”. Em **server.listen**, é onde realmente “ligamos” o servidor.

**Express**

É um framework que auxilia e inclui uma robusta biblioteca de métodos para gerenciamento de rotas e tráfego de dados via métodos de HTTP(S) (get, post, etc...).

Para usar o express é simples

*const express = require('express')*  
*const app = express()*  
*const port = 3000*

*app.get('/', (req, res) => {*  
 *res.send('Hello World!')*  
*})*

*app.listen(port, () => {*  
 *console.log(`Example app listening at http://localhost:${port}`)*  
*})*

Fazemos o require dele (deve ser instalado previamente via NPM), criamos a instancia ‘app’.  
Setamos um método de escuta “GET”, no endpoint root (‘/’) que envia como resposta uma string.  
Iniciamos nosso server usando express com app.listen, na porta que especificarmos.

**Request & Response**

Request e Response são parâmetros padrões dos callbacks de requisição, normalmente representados nas variáveis *(req, res).*   
O *Request* representa a chamada HTTP e contém propriedades como query string, o body da chamada, os headers de requisição, etc...  
Algumas propriedades relevantes:  
***req.body*** -> Contém uma lista de chave-valor que foi definido no body da requisição (normalmente é undefined, mas se usar ‘body-parser’, virá com as chave-valor)  
**req.cookies** -> Se definido na requisição, irá contém os cookies da chamada.  
**req.ip** -> contém o ip de onde veio a requisição  
**req.params** -> Muito útil, contém os parâmetros setados em uma rota, tal como /usuario/:id, irá conter o id passado na URL, aqui  
**req.query** -> Contém todas propriedades em forma de objeto, que foram setadas na query string `...?usuario=1` , terá um {usuario: 1}  
**req.protocol** -> informação se chamada veio de HTTP ou HTTPS  
**req.is([type])** -> método retorna true/false se o content-type da chamada é conforme o parâmetro passado.  
**req.get([field])** -> método que retorna o valor do header solicitado como parâmetro

O *Response*, por sua vez, é usado para definir parâmetros de retorno da chamada para o client-side. Os principais métodos usados para definir o retorno da chamada são:   
**res.cookie(cookieKey, value)** -> seta o valor de um cookie (em string ou JSON)  
**res.download([filePath])** -> retorna um arquivo para download, como um anexo e força o browser a abrir a caixa de diálogo de download.  
**res.end()** -> finaliza a requisição  
**res.json([jsonBody])** -> envia retorno para o client passando um JSON como parâmetro  
**res.send([data])** -> envia dados de retorno para o client que fez a requisição  
**res.set([field, value])** -> Usado para setar valores de headers no HTTP de retorno.  
**res.status([statusCode])** -> Usado para setar o cód. Status da chamada (200, 404, 500...)

**Métodos HTTP**

GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS e TRACE, são os métodos HTTP existentes, e são usados para fazer requisições do client para o server-side.  
**GET** -> Usado para solicitar dados, como o nome diz, get, pegar. Uma chamada GET costuma passar parâmetros pela URL. A requisição/request não possui um body, mas o retorno/response deve conter.  
**HEAD** -> Idêntico ao GET, porém o retorno não contem um body, somente informações de header.  
**POST** -> Envia dados para o server, o tipo de dados é definido pelo content-type. Utilizar o método POST tem como ideais: Postar dados de um form; publicar mensagens ou lista de emails; Criar novo usuário via form; Envio de dados para processamento e retorno de dados processados.   
Post não é idempotente, ou seja, postar os mesmos dados, irá criar duas requisições diferentes, e por exemplo, criar dois usuários idênticos.  
**PUT** -> Tem semelhança com o POST, exceto que o POST normalmente é usado para criar algo novo, e o PUT pode ser usado com ideia de criar OU alterar se já existente. É idempotente, se enviarmos os mesmos dados, por exemplo, para criar usuário, a primeira requisição irá criar, a segunda, alterar.  
**DELETE** -> Usado parar requisitar que o servidor delete algum recurso existente.

**HTTP e HTTPS**

Estes protocolos definem questões de segurança da navegação e transferência de dados. O HTTPS utiliza encriptação SSL, que para ser válido necessita de um certificado. Normalmente utilizam portas diferentes, o HTTP tende usar a port 80, e HTTPS a port 443. Um dominío que queira utilizar HTTPS, precisa ser validado como seguro.

**REST vs. RESTful**

REST é a arquitetura de software que utiliza os protocolos e tecnologias da web para transferir dados e realizar serviços. RESTful é o “adjetivo” dado a APIs ou sistemas que usam arquitetura REST. De acordo com a definição, RESTful seriam apenas as aplicações que utilizam de forma completa os métodos HTTP (GET, PUT, POST, DELETE, etc...) não apenas GET e POST.