

# Do Desenvolvimento à Produção

O Guia Definitivo para Boas Práticas em CAP Node.js

# Uma Jornada da Fundação à Operação

Este guia não é uma lista de regras, mas um roteiro para construir aplicações CAP Node.js excepcionais. Vamos percorrer o ciclo de vida do desenvolvimento, transformando boas práticas em um método de trabalho para criar software mais estável, seguro e sustentável.



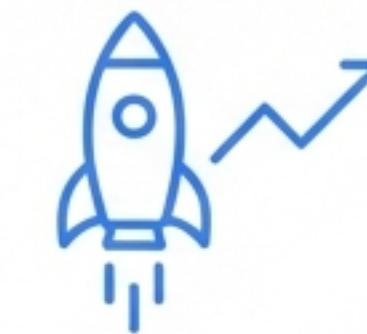
## A Fundação

Preparando o terreno  
para o sucesso.



## A Construção

Escrevendo código  
robusto e seguro.

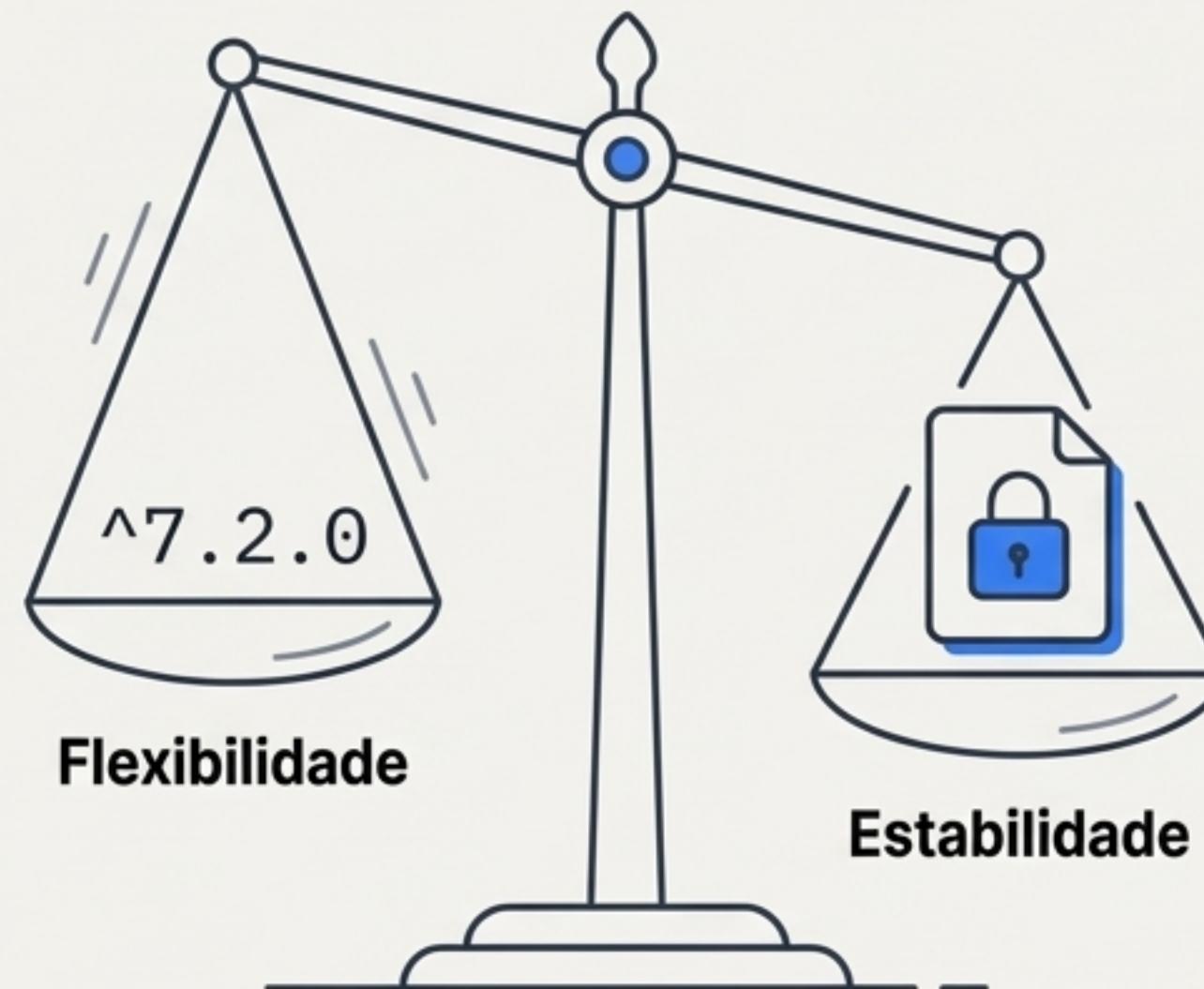


## A Operação

Garantindo performance e  
estabilidade em produção.

# O Princípio das Dependências: Flexibilidade vs. Estabilidade

O gerenciamento eficaz de dependências é um equilíbrio. Durante o desenvolvimento e para pacotes de reuso, precisamos de flexibilidade para receber as últimas correções e funcionalidades. Em produção, precisamos de estabilidade absoluta para garantir que o que foi testado é o que é implantado.



- **Desenvolvimento:** Use ranges abertos (com `^`) para receber atualizações de patch e minor automaticamente.
- **Produção:** Use um arquivo de lock (package-lock.json) para 'congelar' todas as dependências, garantindo implantações consistentes e seguras.

# Estratégia para Desenvolvimento e Reuso: Mantenha Ranges Abertos

Para pacotes `@sap` e open-source, usar a notação de circunflexo (^) garante que seu projeto receba as últimas features e correções importantes, além de otimizar os pacotes com o `dedupe` do NPM.

```
// Good practice for development & reuse packages
"dependencies": {
  "@sap/cds": "^7.8.0",
  "@sap/some-reuse-package": "^1.1.0",
  "express": "^4.17.0"
}
```

## Por que isso é crítico para pacotes de reuso?

- **Evita Duplicação:** Projetos consumidores não recebem versões duplicadas de pacotes.
- **Garante Correções:** Consumidores recebem fixes importantes sem que você precise lançar uma nova versão.
- **Permite Reuso de Modelos CDS:** Essencial para pacotes como `@sap/cds/common`.



**Dica de Mestre:** Antes de publicar um pacote de reuso, sempre execute `npm update` e teste exaustivamente, de preferência de forma automatizada em seu pipeline de CI/CD.

# A Chave para Produção: Congele Dependências com `package-lock.json`

Ao preparar uma aplicação para consumidores finais, a previsibilidade é tudo. O `package-lock.json` garante que o ambiente de produção espelhe exatamente o ambiente que foi testado e verificado contra vulnerabilidades.

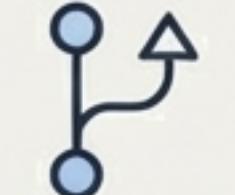
## Processo de Release



1. Habilitar o lockfile:  
`npm config set package-lock true`



2. Atualizar para as últimas versões seguras:  
`npm update`



3. Adicionar ao controle de versão:  
`git add package-lock.json`



4. Executar todos os testes e verificações de vulnerabilidade.

## Resultado

O arquivo `package-lock.json` é implantado com sua aplicação. Instalações subsequentes (por cloud deployers, build packs) usarão sempre as mesmas versões que você validou. Isso impede que novas vulnerabilidades surjam em produção sem sua intervenção.

# Mitigando Riscos: Pacotes Open Source e Atualizações de Versão

Cada dependência é uma superfície de ataque potencial e uma fonte de manutenção. Uma gestão proativa é essencial.

## Minimize o Uso de Pacotes Open Source

- Projetos "end-of-chain" são responsáveis por verificar vulnerabilidades em TODAS as dependências, diretas e transitivas.
- Reduzir o número de pacotes diminui a carga de verificação e o risco.
- Considere tornar features que dependem de pacotes de terceiros opcionais.

## Adote Novas Versões Major Rapidamente

- SDKs como o CAP lançam features e melhorias significativas em ciclos de 6-12 meses.
- Fixes críticos são portados para majors recentes por um período de tempo limitado. Para garantir o recebimento contínuo de correções, mantenha seus projetos ativamente nas versões major mais recentes.

## Ferramentas Úteis

Automatize o processo de atualização com ferramentas como **Renovate** ou o **Dependabot** do GitHub.

# Segurança Começa no Bootstrap

**Contexto:** O CAP adota uma abordagem minimalista para não impor dependências desnecessárias. Isso significa que middlewares de segurança populares, como o `helmet`, não são montados automaticamente. A responsabilidade é do desenvolvedor.

**Ação:** Use o mecanismo de bootstrapping do CAP (`cds.on('bootstrap', ...)`) para montar middlewares de segurança essenciais do Express.

Para um guia completo, consulte a documentação 'Production Best Practices: Security' do Express.

```
// local ./server.js

const cds = require('@sap/cds')
const helmet = require('helmet')

cds.on('bootstrap', app => {
    // Mount security middleware
    app.use(helmet())
})

module.exports = cds.server // > delegate
to default server.js
```

# Protegendo o Conteúdo e as Ações do Usuário



## Content Security Policy (CSP)

**O que é:** Um pilar na segurança de aplicações web que ajuda a prevenir ataques de cross-site scripting (XSS) e injeção de dados.

**Implementação:** O `helmet` fornece uma política padrão. Você pode customizá-la facilmente.

```
cds.on('bootstrap', app => {
  app.use(helmet({
    contentSecurityPolicy: {
      directives: {
        ...helmet.contentSecurityPolicy.getDefaultDirectives(),
        // ... suas diretivas customizadas aqui
      }
    }
  })
})
```



## Cross-Site Request Forgery (CSRF)

**O que é:** Um ataque que força um usuário final a executar ações indesejadas em uma aplicação web na qual ele está autenticado.

**A Solução Recomendada:** Utilizar o App Router. Ele é configurado por padrão para exigir um token CSRF para todas as rotas protegidas e métodos HTTP (exceto HEAD e GET). Para desenvolvedores SAPUI5, o tratamento do token é transparente.

# CSRF: Implementação Manual e Escalabilidade

Em cenários sem o App Router ou que requerem controle customizado, a proteção CSRF pode ser implementada manualmente.

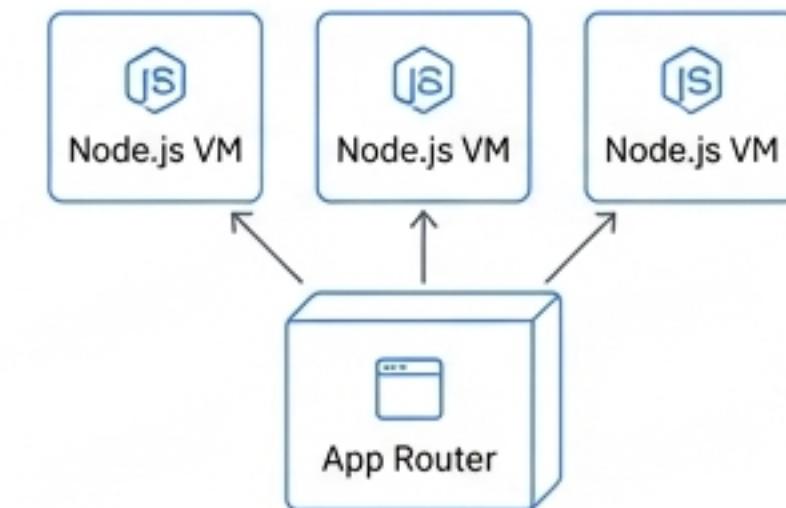
```
// 1. Setup do middleware
const csrfProtection = csrf({ cookie: true });

// 2. Handler para fornecer o token (requisição HEAD)
.head('/<service endpoint>', csrfProtection, (req, res) => {
  res.set({
    'X-CSRF-Token': req.csrfToken(),
    'Cache-Control': 'no-store, no-cache, must-revalidate'
  }).send();
});

// 3. Proteger a rota (ex: POST)
.post('/<service endpoint>/batch', csrfProtection, ...);

// 4. Tratamento de erro para token inválido
.use((err, req, res, next) => {
  if (err.code !== 'EBADCSRFTOKEN') return next(err);
  res.status(403).set('X-CSRF-Token', 'required').send();
});
```

## Dica de Mestre: Escalabilidade Horizontal



Ao escalar VMs Node.js horizontalmente, utilize o tratamento CSRF do App Router. Isso garante consistência entre as instâncias e evita falhas de correspondência de token que poderiam ocorrer se cada VM gerenciasse CSRF de forma independente.

# Gerenciando Acessos de Outras Origens (CORS)

O Cross-Origin Resource Sharing (CORS) permite que um servidor informe ao navegador quais outras origens são confiáveis para solicitar recursos. Em desenvolvimento, o CAP permite todas as origens por padrão. Em produção, isso deve ser restrito.

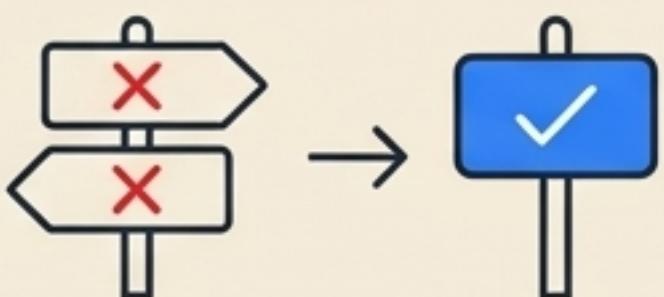
## Implementação Customizada

```
const ORIGINS = { 'https://seu-domínio.com': 1 };

cds.on('bootstrap', app => app.use((req, res, next) => {
  if (req.headers.origin in ORIGINS) {
    res.set('access-control-allow-origin', req.headers.origin);
    if (req.method === 'OPTIONS') { // Preflight request
      return res.set('access-control-allow-methods', '...').send();
    }
  }
  next();
}));
```

## Alternativa Recomendada

O [App Router](#) possui suporte completo a CORS. Centralize a configuração nele.



### Dica de Mestre: Ponto Único de Configuração

Evite configurar CORS tanto no App Router quanto no servidor CAP. Configurações duplicadas podem levar a cenários de depuração confusos. Centralizar as regras em um único local diminui a complexidade e, consequentemente, melhora a segurança.

# A Filosofia de Tratamento de Erros: 'Let It Crash'

Um bom tratamento de erros é vital para a performance e a produtividade. O primeiro passo é entender a natureza do erro.

## Dois Tipos de Erros



### Erros de Programação

**Causa:** Bugs no código (ex: `cannot read 'foo' of undefined`).

**Ação:** Devem ser corrigidos.



### Erros Operacionais

**Causa:** Eventos externos durante a operação (ex: sistema remoto indisponível).

**Ação:** Devem ser tratados.

## Princípio "Let It Crash" (para Erros de Programação)

- **Falhe Ruidosamente:** Não esconda erros nem continue silenciosamente. Logue o erro de forma clara.
- **Não Programe Defensivamente:** Concentre-se na sua lógica de negócio. Use `try/catch` apenas quando souber que um erro operacional pode ocorrer.

Após um erro inesperado, você não pode garantir o estado da aplicação. Manter um processo em execução, especialmente em ambientes multitenant, arrisca a integridade e a segurança dos dados. Deixar o processo travar e ser reiniciado por um orquestrador é a abordagem mais segura.

# Prática de Erros: Não Esconda a Origem

Quando um erro ocorre, sua origem deve ser rastreável. Capturar um erro e lançar um novo, perdendo o stack trace original, torna a depuração exponencialmente mais difícil.

## A Prática Correta

Se precisar adicionar contexto a um erro, modifique o objeto de erro original e relance-o.

### Incorreto (Esconde a origem)



```
try {  
    // algo falha  
} catch (e) {  
    // Perde o stack trace original  
    throw new Error('Oh no! ' + e.message);  
}
```

### Correto (Preserva a origem)



```
try {  
    // algo falha  
} catch (e) {  
    // Adiciona informação, mas preserva o erro original  
    e.message = 'Oh no! ' + e.message;  
    e.additionalInfo = 'This is just an example.';  
    throw e; // Relança o mesmo objeto  
}
```

# Timestamps: Garanta Consistência com req.timestamp

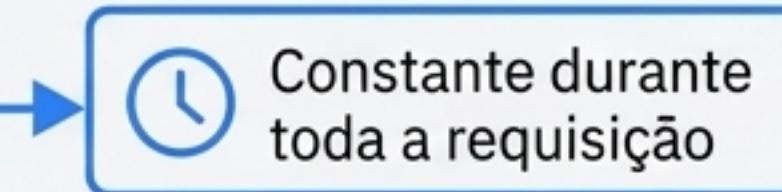
## O Desafio

Ao lidar com timestamps (ex: `createdAt`, `modifiedAt`), é crucial garantir que o mesmo valor de tempo seja usado em toda a transação ou requisição para manter a integridade dos dados.

## A Solução CAP

O objeto `req` da requisição contém a propriedade `timestamp`. Este valor é inicializado no início da requisição (`new Date()` ) e permanece constante até o seu fim.

```
// Em um handler de serviço
srv.before("UPDATE", "EntityName", (req) => {
  // Use o mesmo timestamp para todos os campos de data
  const now = req.timestamp;
  req.data.modifiedAt = now;
  // Se estivéssemos em um CREATE: req.data.createdAt = now;
});
```



Simplicidade e consistência garantida sem a necessidade de gerenciar objetos `Date` manualmente em diferentes partes do seu código. É comparável ao `CURRENT\_TIMESTAMP` do SQL.

# Monitorando a Saúde da Sua Aplicação

Para identificar problemas proativamente, todas as aplicações em produção devem expor um endpoint de verificação de disponibilidade (health check).

## Funcionalidade Nativa do CAP

A partir do @sap/cds^7.8, o runtime Node.js fornece um endpoint em /health por padrão.

Ele não requer autenticação e retorna um status 200 com o corpo: { "status": "UP" }.

## Customização

Se for necessário um health check mais específico, você pode sobrescrever o comportamento padrão durante o bootstrap.

```
cds.on('bootstrap', app =>
  app.get('/health', (_, res) => {
    // Sua lógica de verificação customizada aqui
    res.status(200).send(`I'm fine, thanks.`);
  }));
}
```



Verificações de saúde mais complexas (ex: checar a conexão com o banco de dados) **devem usar autenticação** para prevenir ataques de Negação de Serviço (DoS)!

# O Blueprint de uma Aplicação CAP Profissional

## A Fundação

Começa com um gerenciamento de dependências consciente, que equilibra a agilidade do desenvolvimento com a rocha sólida da estabilidade em produção.



## A Construção

Continua com um pilar de segurança em cada camada da aplicação e uma filosofia de tratamento de erros que valoriza a robustez e a depuração eficaz.

## A Operação

Culmina em uma aplicação estável, monitorada e pronta para o mundo real, garantindo confiabilidade e performance.

**Estas não são apenas regras, são os princípios de um software profissional e sustentável no ecossistema CAP. Construa com excelência.**