# Stateless vs. Stateful

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

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

É possível usar essa abordagem com Node e Express, usando o pacote `express-session`.

Imagine uma API com os seguintes endpoints:

```
POST /carrinho/itens
GET /carrinho/itens
```

Só que ao invés de persistir os itens direto no banco de dados, você optou por usar uma estratégia de API stateful e armazenar os itens usando o conceito de sessão. Para implementar este exemplo, comece criando um novo projeto e instalando as dependências necessárias:

```
npm init
npm i express express-session
```

Implemente agora o arquivo `index.js`:

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


const app = express();


app.use(express.json());
app.use(session({
    secret: 'umsegredo',
    resave: false,
    saveUninitialized: false
}));


app.get('/carrinho/itens', (req, res) => {
    const itens = req.session.itens || [];
    res.send(itens);
});


app.post('/carrinho/itens', (req, res) => {
    if (!req.session.itens) {
```

```
        req.session.itens = [];
    }
    req.session.itens.push(req.body);
    res.status(201).send();
});


app.listen(3000);
```

Execute a API e analise o resultado. Note que entre a chamada do POST e do GET, o Cookie `connect.sid` deve ser mantido, caso contrário o express-session irá entender que são usuários diferentes. Normalmente o cliente HTTP usado para testes fará isso por você, como é o caso do Postman e do próprio navegador, mas nem sempre, como é o caso do cUrl.

Essa abordagem apresenta um problema, no entanto. Imagine que sua aplicação necessite escalar horizontalmente, ou está adotando uma topologia de microsserviços com times autônomos. Como garantir que um mesmo usuário veja os mesmos dados de sessão nesse caso? Existem duas possíveis soluções, o uso de *Sticky session* ou repositórios de sessão persistentes.

## Sticky Session

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

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

## Repositório de sessão persistente

Ao invés de usar um repositório em memória, é possível (na realidade é exigido, pois o próprio express-session diz que o armazenamento em memória não foi projetado para ser usado em produção) plugar no `express-session` um repositório mais persistente, como por exemplo armazenamento em um banco de dados.

O prejuízo dessa solução é que esse repositório passa a ser um gargalo, visto que todas as requisições passam a consultá-lo para carregar e armazenar os dados daquele usuário, sendo que nem todos os seus serviços de back-end precisam disso e quando precisam, dificilmente precisam da sessão completa.

## Stateless

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