🍪 Cookies & Sessions

Gérer l'État dans les Applications Web

Joseph AZAR
IUT Nord Franche-Comté
BUT2 - S3 INFO

Section I

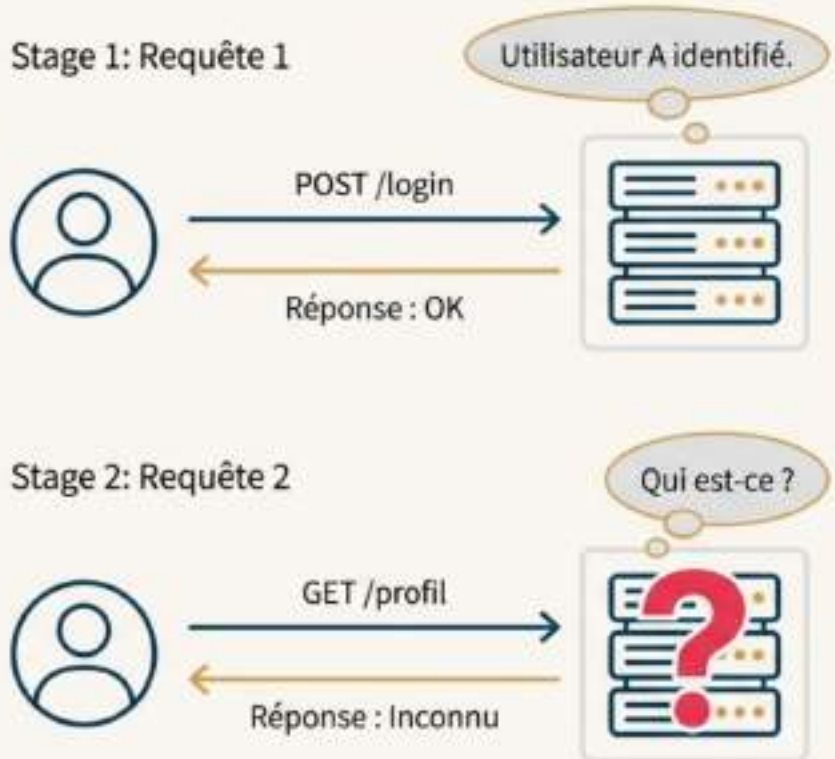
Introduction & Contexte

Le Problème Fondamental : Le Web est Amnésique

Le protocole HTTP, au cœur du web, est "sans état" (stateless). Chaque requête est traitée comme si elle était la première. Le serveur n'a aucune mémoire de vos interactions passées.

Exemple Concret :

1. Vous ajoutez un article à votre panier.
2. Vous cliquez pour voir un autre produit.
3. Votre panier est vide. Le serveur a oublié qui vous êtes.



🤔 Le Problème Fondamental : HTTP est Stateless

HTTP est sans état (stateless)
Chaque requête est traitée comme une transaction indépendante.
Le serveur n'a aucune mémoire des requêtes précédentes.

🤔 Le Problème Fondamental : HTTP est Stateless

HTTP est sans état (stateless)
Chaque requête est traitée comme une transaction indépendante.
Le serveur n'a aucune mémoire des requêtes précédentes.

Conséquence :

- Le serveur ne sait pas qui vous êtes
- Il ne se souvient pas de vos actions précédentes
- Impossible de maintenir une session de connexion

🤔 Le Problème Fondamental : HTTP est Stateless

HTTP est sans état (stateless)
Chaque requête est traitée comme une transaction indépendante.
Le serveur n'a aucune mémoire des requêtes précédentes.

Conséquence :

- Le serveur ne sait pas qui vous êtes
- Il ne se souvient pas de vos actions précédentes
- Impossible de maintenir une session de connexion

📖 Analogie : Le Bibliothécaire Amnésique

À chaque visite, le bibliothécaire vous demande votre carte. Il ne se souvient jamais de vous, même si vous venez juste de partir!

💡 La Solution : Cookies et Sessions

Comment maintenir l'identité de l'utilisateur ?
Nous avons besoin de mécanismes pour "se souvenir" de l'utilisateur entre les requêtes.

💡 La Solution : Cookies et Sessions

Comment maintenir l'identité de l'utilisateur ?
Nous avons besoin de mécanismes pour "se souvenir" de l'utilisateur entre les requêtes.

🍪 Cookies

Petit fichier stocké côté **client** (navigateur)

💾 Sessions

Données stockées côté **serveur** (mémoire, DB, Redis)

La Solution : Donner une Mémoire au Serveur

Pour surmonter l'amnésie de HTTP et maintenir l'identité d'un utilisateur au fil des requêtes (rester connecté, conserver un panier...), nous utilisons une combinaison de deux mécanismes :

Les Cookies

- Rôle : Le Transporteur d'Identité.
- Localisation : Côté client (dans le navigateur).

Les Sessions

- Rôle : Le Gardien de l'État.
- Localisation : Côté serveur.

Le cookie transporte une clé, la session est le coffre-fort que cette clé ouvre.

Section II

Les Cookies

🍪 Qu'est-ce qu'un Cookie ?

> **Définition :** Un cookie est un petit fichier de données (max 4 KB) envoyé par le serveur au navigateur du client.

🍪 Qu'est-ce qu'un Cookie ?

> **Définition :** Un cookie est un petit fichier de données (max 4 KB) envoyé par le serveur au navigateur du client.

Caractéristiques :

- 📦 Stocké sur le navigateur du client
- 🔄 Renvoyé automatiquement au serveur lors des requêtes suivantes
- 🏷️ Contient des paires clé-valeur
- ⏰ Peut avoir une date d'expiration

🍪 Qu'est-ce qu'un Cookie ?

> **Définition :** Un cookie est un petit fichier de données (max 4 KB) envoyé par le serveur au navigateur du client.

Caractéristiques :

- 📦 Stocké sur le navigateur du client
- 🔄 Renvoyé automatiquement au serveur lors des requêtes suivantes
- 🏷️ Contient des paires clé-valeur
- ⏰ Peut avoir une date d'expiration

> ⚠️ **Important :** Les cookies sont visibles et modifiables par l'utilisateur !

Le Cookie : Le Messager Côté Client

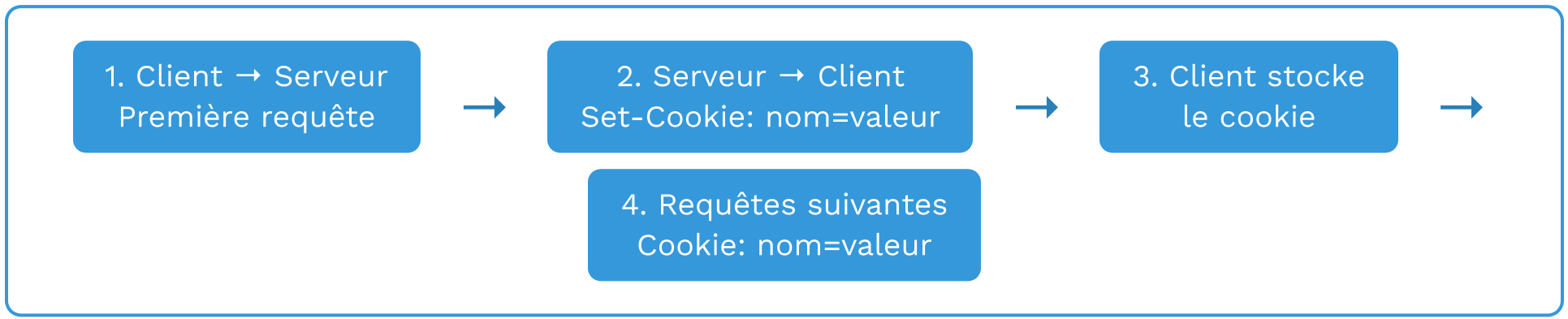
Définition :
Un cookie est un petit fichier texte que le serveur demande au navigateur de stocker.

Mécanisme :

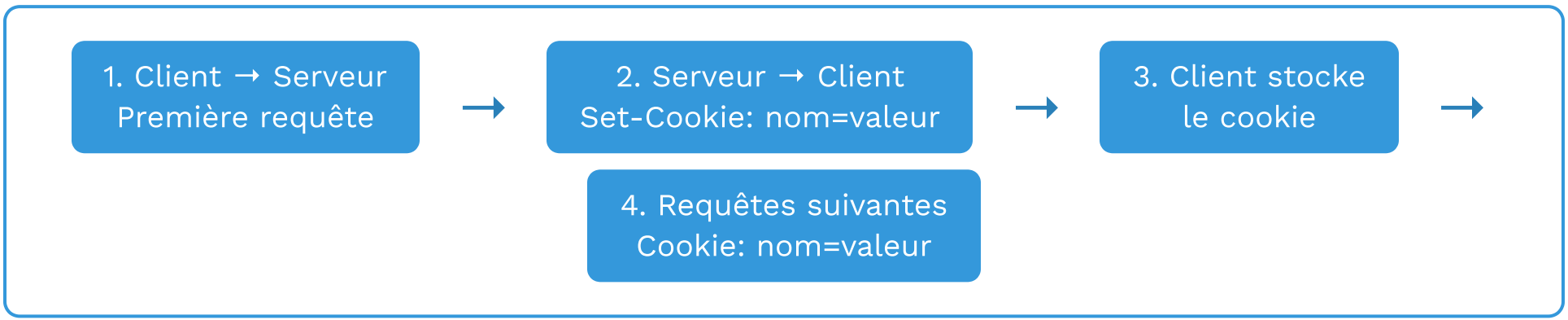
1. Le serveur envoie un en-tête `Set-Cookie` dans sa réponse.
2. Le navigateur stocke cette information.
3. À chaque requête future vers le même domaine, le navigateur renvoie automatiquement le cookie dans un en-tête `Cookie`.

Contenu typique : Il contient de petites informations, le plus souvent un **identifiant de session** unique et opaque (ex: `sessionid=aBcDeF12345`).

🔄 Comment Fonctionnent les Cookies ?



🔄 Comment Fonctionnent les Cookies ?




```
// Réponse HTTP du serveur
HTTP/1.1 200 OK
Set-Cookie: username=alice
Set-Cookie: theme=dark; Max-Age=3600
Content-Type: text/html

// Requête suivante du client
GET /profile HTTP/1.1
Cookie: username=alice; theme=dark
```

📦 Installation : cookie-parser

> **Étape 1 :** Installer le package cookie-parser

```
npm install cookie-parser
```

📦 Installation : cookie-parser

Étape 1 : Installer le package cookie-parser

```
npm install cookie-parser
```

Étape 2 : Configurer dans Express

```
const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();

// Middleware pour parser les cookies
app.use(cookieParser());

app.listen(3000, () => {
  console.log('Serveur démarré sur le port 3000');
});
```

🍪 Créer un Cookie

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Route pour créer un cookie simple
app.get('/setcookie', (req, res) => {
  // Créer un cookie avec nom et valeur
  res.cookie('username', 'alice');
  res.send('Cookie créé avec succès !');
});

app.listen(3000);
```

🍪 Créer un Cookie

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Route pour créer un cookie simple
app.get('/setcookie', (req, res) => {
  // Créer un cookie avec nom et valeur
  res.cookie('username', 'alice');
  res.send('Cookie créé avec succès !');
});

app.listen(3000);
```

Résultat : Le navigateur reçoit un cookie `username=alice` qui sera renvoyé automatiquement lors des prochaines requêtes.

Cookies avec Options

```
app.get('/setcookie', (req, res) => {
  res.cookie('sessionId', 'abc123', {
    maxAge: 3600000,      // Durée de vie en ms (1 heure)
    httpOnly: true,       // Accessible uniquement via HTTP (pas JS)
    secure: true,         // Envoyé uniquement en HTTPS
    sameSite: 'strict'    // Protection CSRF
  });
  res.send('Cookie sécurisé créé !');
});
```

Cookies avec Options

```
app.get('/setcookie', (req, res) => {
  res.cookie('sessionId', 'abc123', {
    maxAge: 3600000,      // Durée de vie en ms (1 heure)
    httpOnly: true,       // Accessible uniquement via HTTP (pas JS)
    secure: true,         // Envoyé uniquement en HTTPS
    sameSite: 'strict'    // Protection CSRF
  });
  res.send('Cookie sécurisé créé !');
});
```

Option	Description
`maxAge`	Durée de vie en millisecondes
`expires`	Date d'expiration précise
`httpOnly`	Non accessible via JavaScript (sécurité XSS)
`secure`	Envoyé uniquement via HTTPS
`sameSite`	Protection contre les attaques CSRF

📖 Lire un Cookie

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Route pour lire les cookies
app.get('/getcookie', (req, res) => {
  // Tous les cookies sont dans req.cookies
  console.log(req.cookies);

  // Accéder à un cookie spécifique
  const username = req.cookies.username;

  if (username) {
    res.send(`Bonjour ${username} !`);
  } else {
    res.send('Aucun cookie trouvé');
  }
});

app.listen(3000);
```

🗑️ Supprimer un Cookie

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Route pour supprimer un cookie
app.get('/deletecookie', (req, res) => {
  // Supprimer le cookie 'username'
  res.clearCookie('username');
  res.send('Cookie supprimé !');
});

// Supprimer avec options (si créé avec options)
app.get('/logout', (req, res) => {
  res.clearCookie('sessionId', {
    httpOnly: true,
    secure: true
  });
  res.send('Déconnecté !');
});

app.listen(3000);
```

🎯 Exemple Complet : Système de Préférences

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Page d'accueil
app.get('/', (req, res) => {
  const theme = req.cookies.theme || 'light';
  const language = req.cookies.language || 'fr';

  res.send(`
    <h1>Vos préférences</h1>
    <p>Thème : ${theme}</p>
    <p>Langue : ${language}</p>
    <a href="/preferences">Modifier</a>
  `);
});

// Sauvegarder les préférences
app.get('/preferences', (req, res) => {
  const { theme, lang } = req.query;

  if (theme) {
    res.cookie('theme', theme, { maxAge: 86400000 }); // 24h
  }
```

💾 Section III

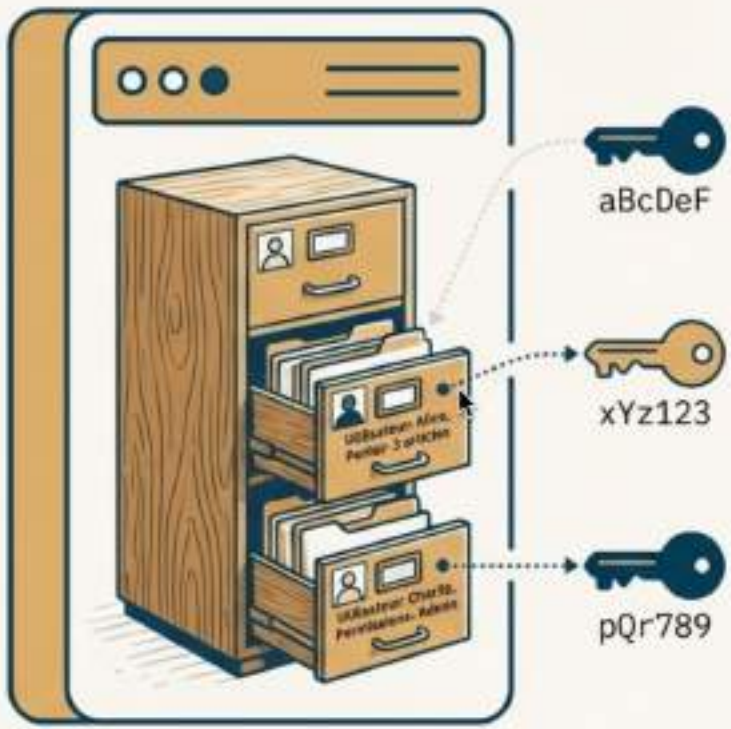
Les Sessions

La Session : La Mémoire Côté Serveur

Définition :
Une session est un espace de stockage côté serveur dédié à un utilisateur spécifique. Cet espace est identifié par un ID de session unique.

Objectif :
Stocker des informations d'état qui ne devraient pas transiter par le client ou qui sont trop volumineuses/sensibles pour un cookie.
Exemples : ID de l'utilisateur connecté, contenu du panier d'achat, permissions, préférences.

Analogie :
Si le cookie est la clé d'un casier, la session est le contenu du casier lui-même, gardé en sécurité derrière le comptoir.



💾 Qu'est-ce qu'une Session ?

> **Définition :** Une session est un mécanisme de stockage d'état côté serveur qui permet de conserver des informations sur l'utilisateur entre plusieurs requêtes.

💾 Qu'est-ce qu'une Session ?

> **Définition :** Une session est un mécanisme de stockage d'état côté serveur qui permet de conserver des informations sur l'utilisateur entre plusieurs requêtes.

Fonctionnement :

1. 1 Le serveur crée un ID de session unique
2. 2 Cet ID est stocké dans un cookie côté client
3. 3 Les données de session sont stockées côté serveur
4. 4 Le serveur récupère les données via l'ID

⚖️ Cookies vs Sessions

Caractéristique	Cookie	Session
Stockage	Côté client (navigateur)	Côté serveur
Capacité	~4 KB	Illimitée (dépend du serveur)
Sécurité	Visible/modifiable par l'utilisateur	Invisible pour l'utilisateur
Expiration	Définie manuellement	Expiration automatique (timeout)
Usage typique	Préférences, thème, langue	Authentification, panier, données utilisateur

Cookie vs. Session : La Synthèse des Différences

Cookie		Session	
Localisation	Côté client (navigateur)	**Localisation**	Côté serveur
Rôle	Transport (la clé)	**Rôle**	État (le contenu du casier)
Contenu	Données textuelles simples, souvent un identifiant	**Contenu**	Données complexes et sensibles
Taille	Limitée (~4 Ko)	**Taille**	Limitée par les ressources du serveur
Sécurité	Faible (visible et modifiable par le client)	**Sécurité**	Élevée (invisible pour le client)

🔗 Le Rôle du Cookie dans les Sessions

📖 Analogie : Le Club de Sport

Le **Cookie** est la clé de votre casier (juste un numéro).
La **Session** est le registre électronique à l'accueil (toutes vos informations).

🔗 Le Rôle du Cookie dans les Sessions

📖 Analogie : Le Club de Sport

Le **Cookie** est la clé de votre casier (juste un numéro).
La **Session** est le registre électronique à l'accueil (toutes vos informations).

🍪 **Cookie de session :** Contient uniquement l'ID

```
connect.sid=s%3AjG7yd...
```

💾 **Données de session côté serveur :**

```
{
  "sessionID": "jG7yd...",
  "username": "alice",
  "isAdmin": true,
  "cart": ["item1", "item2"]
}
```

Implémentation Pratique avec Express.js

Express, le framework web le plus populaire pour Node.js, simplifie la gestion des sessions grâce à un système de `middlewares`.

Les Outils Essentiels

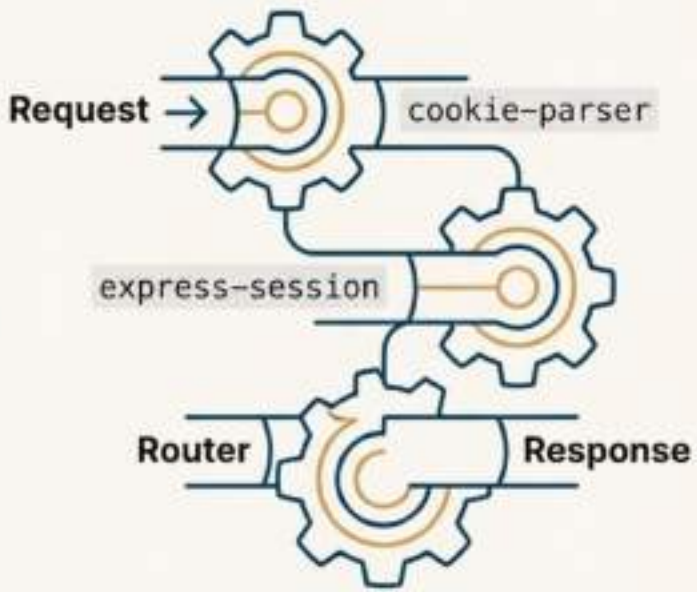
`cookie-parser`

Un middleware qui analyse les en-têtes `Cookie` des requêtes. Il remplit l'objet `req.cookies` avec les cookies reçus.

`express-session`

Le middleware principal. Il crée et gère les sessions, stocke les données de session sur le serveur, et attache `req.session` à la requête pour un accès facile.

← Express.js



Configuration de `express-session`

```
import session from 'express-session';

app.use(session({
  // Clé secrète pour signer le cookie. Essentiel à la sécurité.
  secret: 'ceci-est-un-secret-robuste',

  // Ne pas sauvegarder la session si elle n'est pas modifiée.
  resave: false,
  // Ne pas créer de session pour un utilisateur non identifié.
  saveUninitialized: false,

  // Configuration du cookie de session.
  cookie: {
    secure: true, // N'envoyer que sur HTTPS en production.
    httpOnly: true, // Empêche l'accès via JavaScript côté client.
    maxAge: 3600000 // Durée de vie en ms (1 heure).
  }
}));
```

Où sont stockées les sessions ?

En développement : Par défaut, en mémoire (`MemoryStore`).
Attention : les sessions sont perdues au redémarrage !

En production : Utilisez un 'store' persistant comme `connect-redis` ou `connect-mongo` pour la durabilité et la scalabilité.

📦 Installation : express-session

> **Étape 1 :** Installer express-session

```
npm install express-session
```

📦 Installation : express-session

Étape 1 : Installer express-session

```
npm install express-session
```

Étape 2 : Configurer dans Express

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

const app = express();

// Configuration de la session
app.use(session({
  secret: 'votre-clé-secrète-très-sécurisée',
  resave: false,
  saveUninitialized: true,
  cookie: {
    secure: false,    // true en production avec HTTPS
    maxAge: 3600000   // 1 heure
  }
}));

app.listen(3000);
```

⚙️ Options de Configuration

Option	Description
`secret`	Clé pour signer le cookie de session (obligatoire)
`resave`	Sauvegarder même si non modifiée (généralement `false`)
`saveUninitialized`	Sauvegarder les sessions vides (généralement `false`)
`cookie.secure`	HTTPS uniquement (`true` en production)
`cookie.maxAge`	Durée de vie du cookie de session

⚙️ Options de Configuration

Option	Description
`secret`	Clé pour signer le cookie de session (obligatoire)
`resave`	Sauvegarder même si non modifiée (généralement `false`)
`saveUninitialized`	Sauvegarder les sessions vides (généralement `false`)
`cookie.secure`	HTTPS uniquement (`true` en production)
`cookie.maxAge`	Durée de vie du cookie de session

⚠️ **Sécurité :** Utilisez un `secret` fort et unique, et activez `secure: true` en production !

💾 Créer et Lire une Session

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

app.use(session({
  secret: 'ma-clé-secrète',
  resave: false,
  saveUninitialized: false
}));

// Créer des données de session
app.get('/login', (req, res) => {
  // Stocker des informations dans la session
  req.session.username = 'alice';
  req.session.isAdmin = true;
  req.session.cart = ['item1', 'item2'];

  res.send('Session créée !');
});

// Lire les données de session
app.get('/profile', (req, res) => {
  if (req.session.username) {
    res.send(`Bienvenue ${req.session.username} !`);
  } else {
    res.send('Veuillez vous connecter');
```

🗑️ Détruire une Session

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

app.use(session({
  secret: 'ma-clé-secrète',
  resave: false,
  saveUninitialized: false
}));

// Détruire la session complète
app.get('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      return res.send('Erreur lors de la déconnexion');
    }
    res.send('Déconnecté avec succès !');
  });
});

// Supprimer une propriété spécifique
app.get('/clear-cart', (req, res) => {
  delete req.session.cart;
  res.send('Panier vidé !');
});
```

🔍 Comprendre req.session en Détail

> **Question :** D'où vient `req.session` ? Est-ce automatique ?

🔍 Comprendre req.session en Détail

Question : D'où vient `req.session` ? Est-ce automatique ?

Réponse : `req.session` est créé par le middleware `express-session`

```
// Ce middleware ajoute req.session à CHAQUE requête
app.use(session({
  secret: 'ma-clé',
  resave: false,
  saveUninitialized: false
}));

// Maintenant req.session est disponible dans toutes les routes !
```

📦 req.session : Qu'est-ce qui est inclus ?

🔧 Propriétés par Défaut

Ajoutées automatiquement par express-session :

- `req.session.id` - ID unique de session
- `req.session.cookie` - Infos du cookie
- `req.session.destroy()` - Méthode pour détruire
- `req.session.regenerate()` - Régénérer l'ID
- `req.session.save()` - Sauvegarder manuellement
- `req.session.reload()` - Recharger

✍️ Propriétés Manuelles

Ajoutées par VOUS (le développeur) :

- `req.session.username` ✍️ vous
- `req.session.isAuthenticated` ✍️ vous
- `req.session.isAdmin` ✍️ vous
- `req.session.cart` ✍️ vous
- `req.session.email` ✍️ vous
- **N'importe quelle propriété !**

📦 req.session : Qu'est-ce qui est inclus ?

🔧 Propriétés par Défaut

Ajoutées automatiquement par express-session :

- `req.session.id` - ID unique de session
- `req.session.cookie` - Infos du cookie
- `req.session.destroy()` - Méthode pour détruire
- `req.session.regenerate()` - Régénérer l'ID
- `req.session.save()` - Sauvegarder manuellement
- `req.session.reload()` - Recharger

✍️ Propriétés Manuelles

Ajoutées par VOUS (le développeur) :

- `req.session.username` ✍️ vous
- `req.session.isAuthenticated` ✍️ vous
- `req.session.isAdmin` ✍️ vous
- `req.session.cart` ✍️ vous
- `req.session.email` ✍️ vous
- **N'importe quelle propriété !**

💡 **Important :** `req.session` est un objet JavaScript vide au départ. Vous pouvez y ajouter n'importe quelle propriété que vous voulez !

👁️ Visualisation : req.session

Au départ (avant login) :

```
{
  id: "s3AjG7ydK...",        // ✅ Par défaut (express-session)
  cookie: {                  // ✅ Par défaut (express-session)
    maxAge: 3600000,
    secure: false,
    httpOnly: true
  },
  destroy: [Function],       // ✅ Par défaut (express-session)
  regenerate: [Function],    // ✅ Par défaut (express-session)
  save: [Function]           // ✅ Par défaut (express-session)
}
```

Après login (vous ajoutez des propriétés) :

```
{
  id: "s3AjG7ydK...",
  cookie: { ... },
  destroy: [Function],
  regenerate: [Function],
  save: [Function],
  username: "alice",         // ✍️ Ajouté par VOUS
  isAuthenticated: true,     // ✍️ Ajouté par VOUS
  isAdmin: false,            // ✍️ Ajouté par VOUS
  cart: ["item1", "item2"]   // ✍️ Ajouté par VOUS
}
```

✍️ Comment Ajouter vos Propriétés à req.session

```
// Étape 1 : Configurer le middleware (express-session)
app.use(session({
  secret: 'ma-clé',
  resave: false,
  saveUninitialized: false
}));

// Étape 2 : Dans vos routes, ajoutez vos propres propriétés
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Validation (simplifié)
  if (username === 'admin' && password === 'admin') {
    // ✍️ VOUS créez ces propriétés !
    req.session.username = username;           // Ajouté par vous
    req.session.isAuthenticated = true;        // Ajouté par vous
    req.session.isAdmin = true;                // Ajouté par vous
    req.session.loginTime = new Date();        // Ajouté par vous

    res.send('Connecté !');
  } else {
    res.send('Échec');
  }
});
```

✍️ Comment Ajouter vos Propriétés à req.session

```
// Étape 1 : Configurer le middleware (express-session)
app.use(session({
  secret: 'ma-clé',
  resave: false,
  saveUninitialized: false
}));

// Étape 2 : Dans vos routes, ajoutez vos propres propriétés
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Validation (simplifié)
  if (username === 'admin' && password === 'admin') {
    // ✍️ VOUS créez ces propriétés !
    req.session.username = username;           // Ajouté par vous
    req.session.isAuthenticated = true;        // Ajouté par vous
    req.session.isAdmin = true;                // Ajouté par vous
    req.session.loginTime = new Date();        // Ajouté par vous

    res.send('Connecté !');
  } else {
    res.send('Échec');
  }
});
```

✅ **Règle d'or :** Vous pouvez créer n'importe quelle propriété avec n'importe quel nom. C'est VOUS qui décidez !

🔐 Exemple Complet : Authentification

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

app.use(express.urlencoded({ extended: false }));
app.use(session({
  secret: 'super-secret-key',
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 3600000 } // 1 heure
}));

// Page de login
app.get('/login', (req, res) => {
  res.send(`
    <form method="POST" action="/login">
      <input name="username" placeholder="Username">
      <input name="password" type="password" placeholder="Password">
      <button>Login</button>
    </form>
  `);
});

// Traiter le login
app.post('/login', (req, res) => {
  const { username, password } = req.body;
```

🔐 Exemple Complet : Authentification (suite)

```
// Middleware de protection
function requireAuth(req, res, next) {
  if (req.session.isAuthenticated) {
    next(); // Utilisateur connecté, continuer
  } else {
    res.redirect('/login'); // Rediriger vers login
  }
}

// Route protégée
app.get('/dashboard', requireAuth, (req, res) => {
  res.send(`
    <h1>Dashboard</h1>
    <p>Bienvenue ${req.session.username} !</p>
    <a href="/logout">Logout</a>
  `);
});

// Logout
app.get('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      return res.send('Erreur');
    }
    res.redirect('/login');
  });
```

🗄 Section IV

Stockage des Sessions

🗄️ Options de Stockage des Sessions

Par défaut : Les sessions sont stockées en mémoire (MemoryStore).
⚠️ Non recommandé en production !

🗄️ Options de Stockage des Sessions

Par défaut : Les sessions sont stockées en mémoire (MemoryStore).
⚠️ Non recommandé en production !

Stockage	Avantages	Inconvénients	Usage
MemoryStore	Simple, rapide	Perdu au redémarrage, non scalable	Développement uniquement
FileStore	Persiste sur disque	Lent, non scalable	Petites applications
Redis	Rapide, scalable, distribué	Infrastructure supplémentaire	Production recommandée
MongoDB	Intégration DB existante	Plus lent que Redis	Si déjà utilisé

📁 Session FileStore

> **Installation :**

```
npm install session-file-store
```

📁 Session FileStore

> **Installation :**

```
npm install session-file-store
```

```
const express = require('express');
const session = require('express-session');
const FileStore = require('session-file-store')(session);

const app = express();

app.use(session({
  store: new FileStore({
    path: './sessions',      // Dossier pour stocker les sessions
    ttl: 3600,               // Durée de vie en secondes
    retries: 0
  }),
  secret: 'ma-clé-secrète',
  resave: false,
  saveUninitialized: false
}));

app.listen(3000);
```

⚡ Session Redis (aperçu)

Redis : Solution recommandée pour la production
Base de données en mémoire ultra-rapide, scalable et distribuée

```
npm install connect-redis redis
```

⚡ Session Redis (aperçu)

Redis : Solution recommandée pour la production
Base de données en mémoire ultra-rapide, scalable et distribuée

```
npm install connect-redis redis
```

```
const express = require('express');
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const redis = require('redis');
const redisClient = redis.createClient();

const app = express();

app.use(session({
  store: new RedisStore({ client: redisClient }),
  secret: 'ma-clé-secrète',
  resave: false,
  saveUninitialized: false
}));

app.listen(3000);
```

⚡ Session Redis (aperçu)

Redis : Solution recommandée pour la production
Base de données en mémoire ultra-rapide, scalable et distribuée

```
npm install connect-redis redis
```

```
const express = require('express');
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const redis = require('redis');
const redisClient = redis.createClient();

const app = express();

app.use(session({
  store: new RedisStore({ client: redisClient }),
  secret: 'ma-clé-secrète',
  resave: false,
  saveUninitialized: false
}));

app.listen(3000);
```

Note : Redis sera étudié plus en détail dans un cours dédié !

Section V

Sessions et Architecture REST

📊 Rappel : Le Principe REST et Statelessness

> **Contrainte REST :** Stateless (sans état)
> Chaque requête doit contenir toutes les informations nécessaires pour être traitée.

Rappel : Le Principe REST et Statelessness

Contrainte REST : Stateless (sans état)
Chaque requête doit contenir toutes les informations nécessaires pour être traitée.

Problème : Les sessions violent la contrainte stateless !
Le serveur doit maintenir un état (la session) entre les requêtes.

Rappel : Le Principe REST et Statelessness

Contrainte REST : Stateless (sans état)
Chaque requête doit contenir toutes les informations nécessaires pour être traitée.

Problème : Les sessions violent la contrainte stateless !
Le serveur doit maintenir un état (la session) entre les requêtes.

Pourquoi c'est un problème ?

- Le serveur doit interroger un store (mémoire, Redis, DB)
- Dépendance à l'état serveur
- Complexité en architecture distribuée

🤔 Quand Utiliser les Sessions ?

✅ Utilisez les Sessions

- Applications web traditionnelles (MVC)
- Rendu côté serveur (SSR)
- Besoin de contrôle serveur total
- Invalidation immédiate requise
- Applications monolithiques

❌ Évitez les Sessions

- APIs REST pures
- Microservices
- Applications distribuées
- Clients mobiles / SPAs
- Besoin de scalabilité horizontale

👍 Avantages des Sessions

1. Contrôle Serveur Total

Le serveur peut invalider une session instantanément (déconnexion forcée, suspension de compte).

👍 Avantages des Sessions

1. Contrôle Serveur Total

Le serveur peut invalider une session instantanément (déconnexion forcée, suspension de compte).

2. Sécurité

Les données sensibles restent côté serveur, seul l'ID de session est envoyé au client.

👍 Avantages des Sessions

1. Contrôle Serveur Total

Le serveur peut invalider une session instantanément (déconnexion forcée, suspension de compte).

2. Sécurité

Les données sensibles restent côté serveur, seul l'ID de session est envoyé au client.

3. Simplicité

Plus simple à implémenter pour des applications web classiques.

👍 Avantages des Sessions

1. Contrôle Serveur Total

Le serveur peut invalider une session instantanément (déconnexion forcée, suspension de compte).

2. Sécurité

Les données sensibles restent côté serveur, seul l'ID de session est envoyé au client.

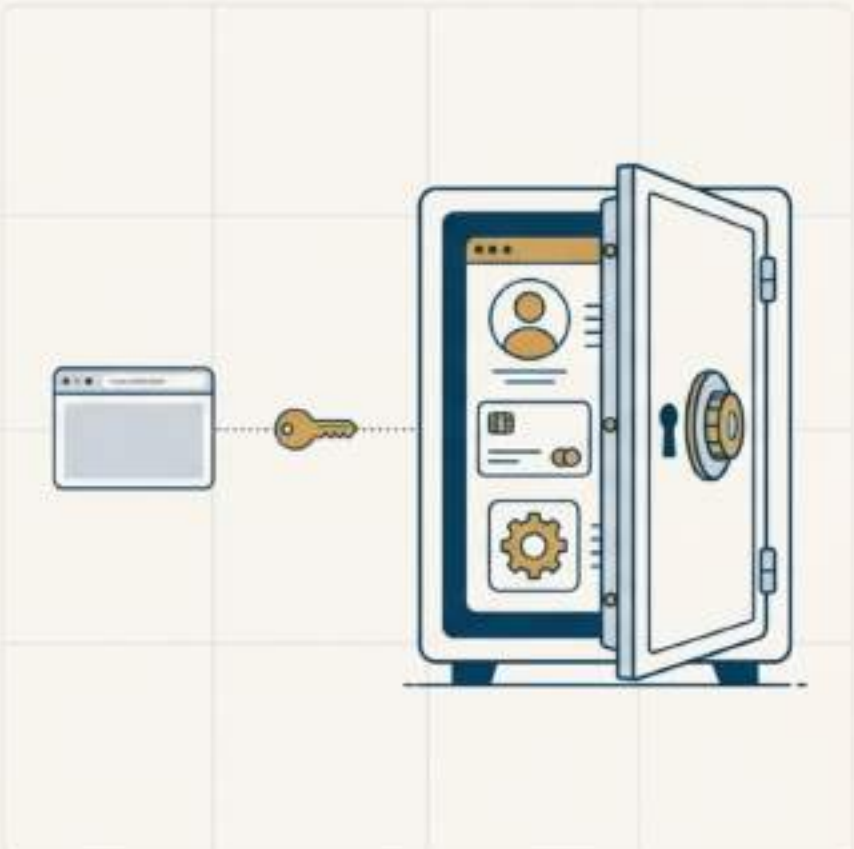
3. Simplicité

Plus simple à implémenter pour des applications web classiques.

Note : Pour les APIs REST pures, les tokens (comme JWT) sont préférables. Nous les verrons dans un cours ultérieur !

Les Avantages des Sessions Côté Serveur

- **Sécurité Accrue:** Les données sensibles (ID utilisateur, permissions, etc.) ne quittent jamais le serveur. Seul un identifiant opaque est exposé au client.
- **Contrôle Centralisé:** Le serveur a une autorité totale sur la durée de vie de la session. Il peut invalider une session à tout moment (ex: déconnexion forcée d'un utilisateur).
- **Flexibilité du Stockage:** Permet de stocker des données plus volumineuses et complexes qu'un simple cookie.
- **Solution Éprouvée:** Une méthode simple et robuste pour les applications web 'traditionnelles' où le serveur rend les vues HTML.

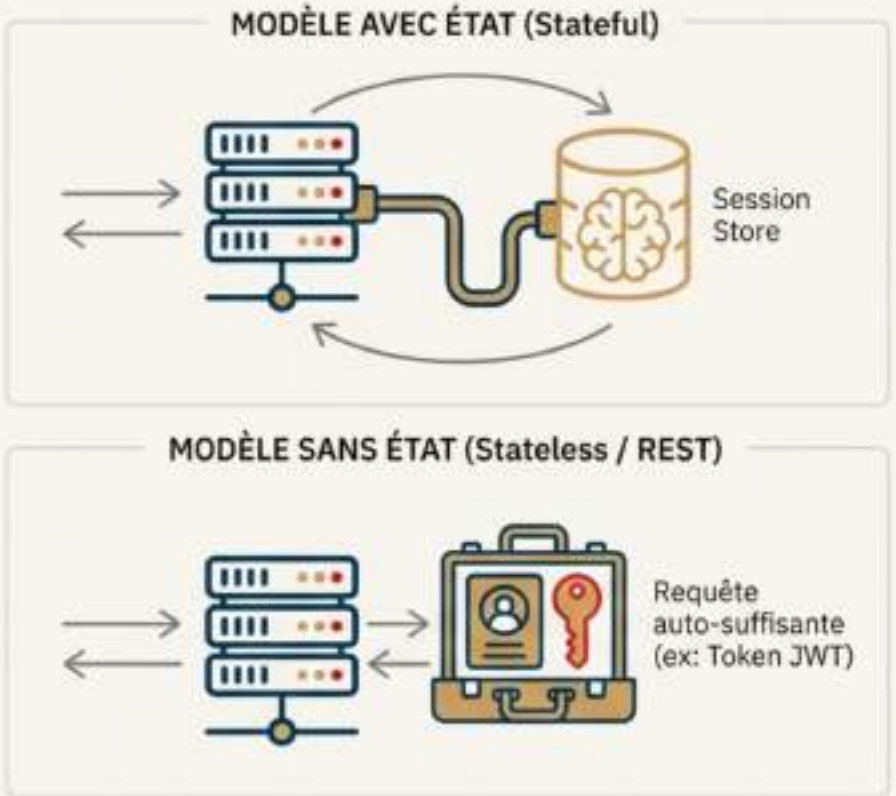


Le Défi des APIs REST : Le Retour du "Sans État"

Rappel sur REST : REST (Representational State Transfer) n'est pas un protocole, mais un style d'architecture.

La Contrainte Fondamentale de REST : *Stateless* (Sans État).

- Chaque requête du client doit contenir **toutes** les informations nécessaires pour que le serveur puisse la traiter.
- Le serveur ne doit conserver **aucun contexte** ou état du client entre les requêtes.



Le Conflit : Pourquoi les Sessions Violent le Principe REST

Le Point de Friction :

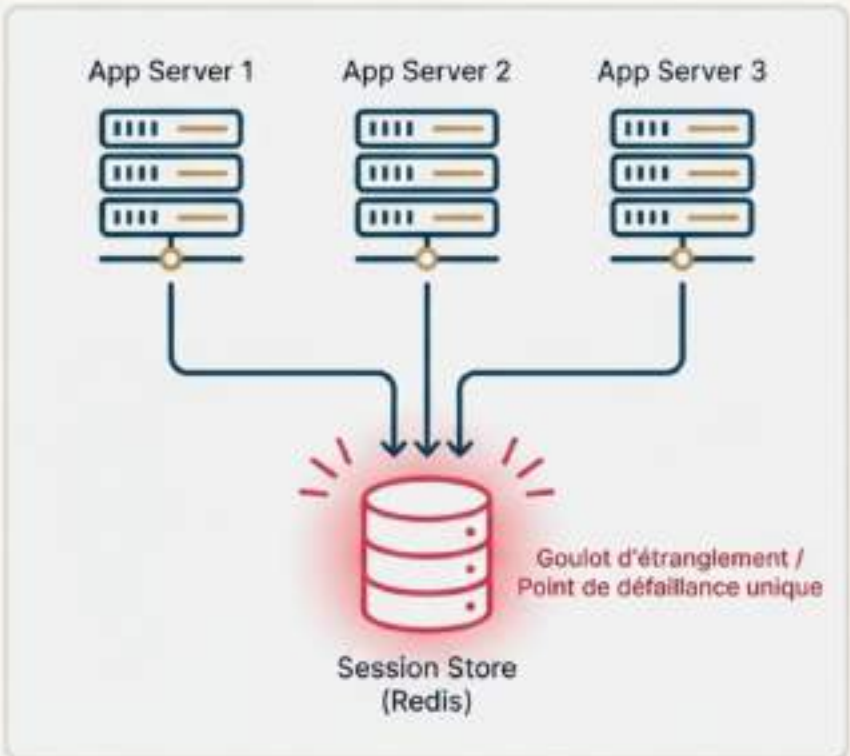
1. Lire l'ID de session depuis le cookie.
2. **Consulter un `Session Store` externe** (Redis, base de données...).

La Conséquence :

Le serveur dépend d'un état externe pour traiter la requête. Il n'est plus `stateless`. La requête n'est plus autonome.

Le Problème de Scalabilité :

- Dans une architecture distribuée (plusieurs serveurs, microservices), tous les services doivent accéder à un **`Session Store` centralisé et partagé.**
- Cela introduit de la complexité, un point de défaillance unique (Single Point of Failure), et une latence supplémentaire.



✅ Section VI

Bonnes Pratiques & Sécurité

🔒 Sécurité des Sessions

1. Utilisez HTTPS en production

```
cookie: {
  secure: true,  // ⚠️ Obligatoire en production !
  httpOnly: true,
  sameSite: 'strict'
}
```

🔒 Sécurité des Sessions

1. Utilisez HTTPS en production

```
cookie: {
  secure: true,  // ⚠️ Obligatoire en production !
  httpOnly: true,
  sameSite: 'strict'
}
```

2. Secret fort et unique

```
// ❌ MAUVAIS
secret: '1234'

// ✅ BON
secret: process.env.SESSION_SECRET  // Variable d'environnement
```

🔒 Sécurité des Sessions

1. Utilisez HTTPS en production

```
cookie: {
  secure: true,  // ⚠️ Obligatoire en production !
  httpOnly: true,
  sameSite: 'strict'
}
```

2. Secret fort et unique

```
// ❌ MAUVAIS
secret: '1234'

// ✅ BON
secret: process.env.SESSION_SECRET  // Variable d'environnement
```

3. Timeout de session

```
cookie: {
  maxAge: 1800000  // 30 minutes d'inactivité
}
```

✅ Bonnes Pratiques

1. Ne stockez pas d'informations sensibles dans les cookies

```
// ❌ MAUVAIS - données sensibles dans cookie
res.cookie('user', JSON.stringify({ password: '123', card: '4532' }));

// ✅ BON - seulement dans la session côté serveur
req.session.userDetails = { email: 'alice@example.com' };
```

✅ Bonnes Pratiques

1. Ne stockez pas d'informations sensibles dans les cookies

```
// ❌ MAUVAIS - données sensibles dans cookie
res.cookie('user', JSON.stringify({ password: '123', card: '4532' }));

// ✅ BON - seulement dans la session côté serveur
req.session.userDetails = { email: 'alice@example.com' };
```

2. Régénérez l'ID de session après login

```
app.post('/login', (req, res) => {
  // Après validation des credentials
  req.session.regenerate((err) => {
    req.session.username = 'alice';
    res.redirect('/dashboard');
  });
});
```

✅ Bonnes Pratiques (suite)

3. Vérifiez toujours la session

```
function requireAuth(req, res, next) {
  if (!req.session || !req.session.username) {
    return res.redirect('/login');
  }
  next();
}
```

✅ Bonnes Pratiques (suite)

3. Vérifiez toujours la session

```
function requireAuth(req, res, next) {
  if (!req.session || !req.session.username) {
    return res.redirect('/login');
  }
  next();
}
```

4. Utilisez un store adapté en production

```
// ❌ MAUVAIS en production
// Par défaut : MemoryStore (perdu au redémarrage)

// ✅ BON en production
store: new RedisStore({ client: redisClient })
```

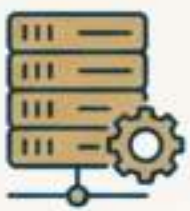
Alors, Sessions ou Pas Sessions ? Le Contexte Décide.

Utiliser les Sessions pour...

Applications Web Traditionnelles (Monolithiques) : Le serveur génère le HTML et gère l'état de l'interface.

Exemple : Un blog WordPress, un forum classique, un site e-commerce avec rendu côté serveur.

Avantages: Simple, sécurisé, et parfaitement intégré aux frameworks comme Express.

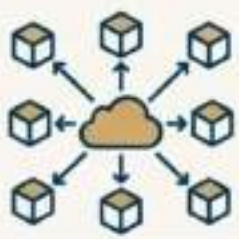


Éviter les Sessions pour...

APIs REST strictes : Destinées à des clients multiples (Single Page Apps, applications mobiles, autres services).

Architectures Microservices : Pour garantir l'indépendance et la scalabilité de chaque service.

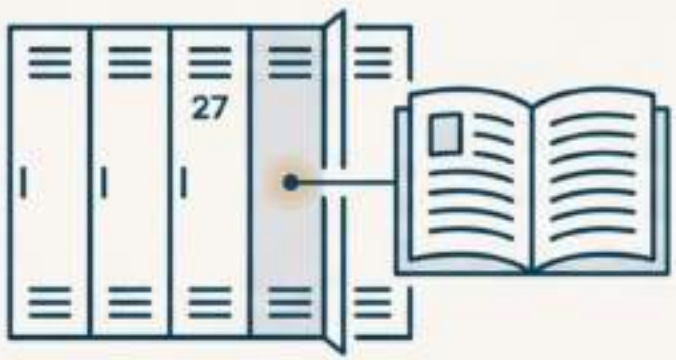
L'Alternative `Stateless` : Les JSON Web Tokens (JWT). Le token, signé et auto-suffisant, est envoyé à chaque requête et contient les informations de l'utilisateur.



L'Analogie du Casier : Le Résumé Visuel

La Clé du Casier

- Petit et simple, ne contient que l'identifiant (le numéro 27).
- Détenu par vous (le client).
- Inutile sans le registre du club.



Le Casier et son Registre

- Contient toutes vos informations (votre nom, vos affaires...).
- Stocké et sécurisé par le club (le serveur).
- La clé permet d'y accéder.

Vos Points Clés à Retenir

Cookie = Transport.

C'est le véhicule qui porte l'identifiant. Il est simple et côté côté client.

Session = État.

C'est la mémoire sécurisée côté serveur. Elle est riche et contrôlée par le serveur.

Le Contexte Architectural est Roi.

Le Contexte Architectural est Roi

Le choix entre une approche stateful (sessions) et stateless (tokens) n'est pas technique, mais architectural. Il dépend de votre application : un monolithe traditionnel ou une API distribuée.

Comprendre le pourquoi de ces choix est aussi crucial que de connaître le comment.

📝 Récapitulatif

🍪 Cookies

- Stockés côté client
- Max 4 KB
- Visibles par l'utilisateur
- Préférences, langue, thème
- `res.cookie()`
- `req.cookies`

💾 Sessions

- Stockées côté serveur
- Capacité illimitée
- Invisibles pour l'utilisateur
- Authentification, panier
- `req.session`
- Store : Redis, FileStore, DB

📝 Récapitulatif

🍪 Cookies

- Stockés côté client
- Max 4 KB
- Visibles par l'utilisateur
- Préférences, langue, thème
- `res.cookie()`
- `req.cookies`

💾 Sessions

- Stockées côté serveur
- Capacité illimitée
- Invisibles pour l'utilisateur
- Authentification, panier
- `req.session`
- Store : Redis, FileStore, DB

> 🔑 **Points Clés :**
>
> - HTTP est stateless, cookies et sessions permettent de maintenir l'état
> - Les sessions violent REST mais sont utiles pour les apps web traditionnelles
> - Sécurité : HTTPS, httpOnly, secret fort, timeout

Questions ?

Joseph AZAR
joseph.azar@univ-fcomte.fr
IUT Nord Franche-Comté

📚 Ressources :
express-session | cookie-parser