Le Dialogue Client-Serveur : Anatomie d'une Requête Web

Cette infographie illustre le processus d'une communication web standard. Elle montre comment un client formule une demande (requête HTTP) pour obtenir des informations, et comment le serveur analyse cette demande, la traite (souvent en interagissant avec une base de données) et renvole une réponse structurée.

Client

Étape 1 : La Requête du Client

Le client construit et envoie une requête HTTP.

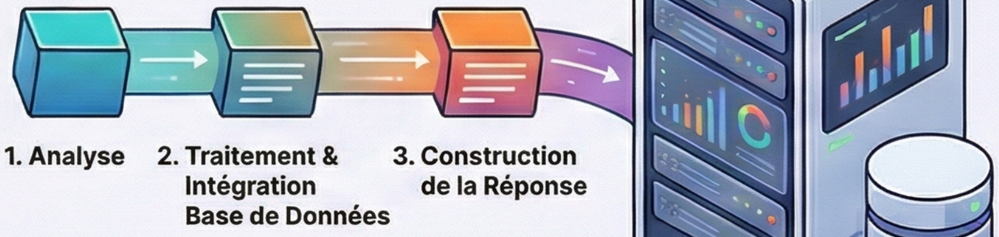
Les en-têtes expliquent au serveur **QUI** parle, **QUOI** faire et **COMMENT** répondre. ils contiennent des métadonnées comme le type de contenu, l'authentification et l'origine.

LES DONNÉES PEUVENT ÊTRE ENVOYÉES DE DIFFÉRENTES MANIÈRES

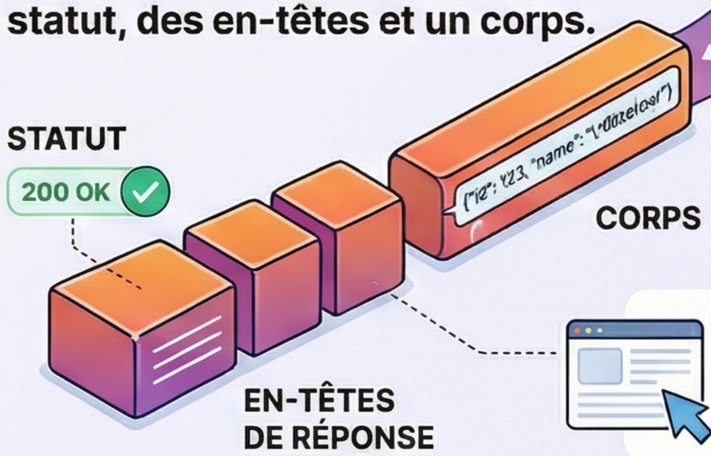
Étape 2 : La Réponse du Serveur

Serveur

Le serveur analyse la requête, la traite et construit une réponse.



La réponse HTTP contient un statut, des en-têtes et un corps.



Les en-têtes de réponse guident le navigateur. ils expliquent comment interpréter, sécuriser et mettre en cache le contenu reçu.

Les headers HTTP dans une requête web

Comprendre les métadonnées échangées entre client et serveur

1 Client HTTP (Navigateur)

Construit et envoie une requête HTTP

Requête envoyée

Requête HTTP

🔷 Ligne de requête

```
GET /api/users HTTP/1.1
```

Action demandée au serveur : Méthode + Ressource + Version HTTP

🔶 Headers HTTP ⭐

```
Host: api.example.com
User-Agent: Mozilla/5.0
Accept: application/json
Content-Type: application/json
Authorization: Bearer TOKEN
Origin: https://site-client.com
```

- **Host** → site ciblé
- **User-Agent** → client utilisé
- **Accept** → format attendu
- **Content-Type** → format envoyé
- **Authorization** → identité / droits
- **Origin** → sécurité (CORS)

Les headers sont des métadonnées, pas des données métier

🔷 Body (optionnel)

```
{
  "username": "alice",
  "password": "•••••"
}
```

Données envoyées (POST / PUT)

🔍 Zoom pédagogique

❓ À quoi servent les headers ?

- 🧭 Identifier le client
- 📦 Décrire les données
- 🔒 Sécuriser l'accès
- 🌍 Gérer les origines - Adapter la réponse

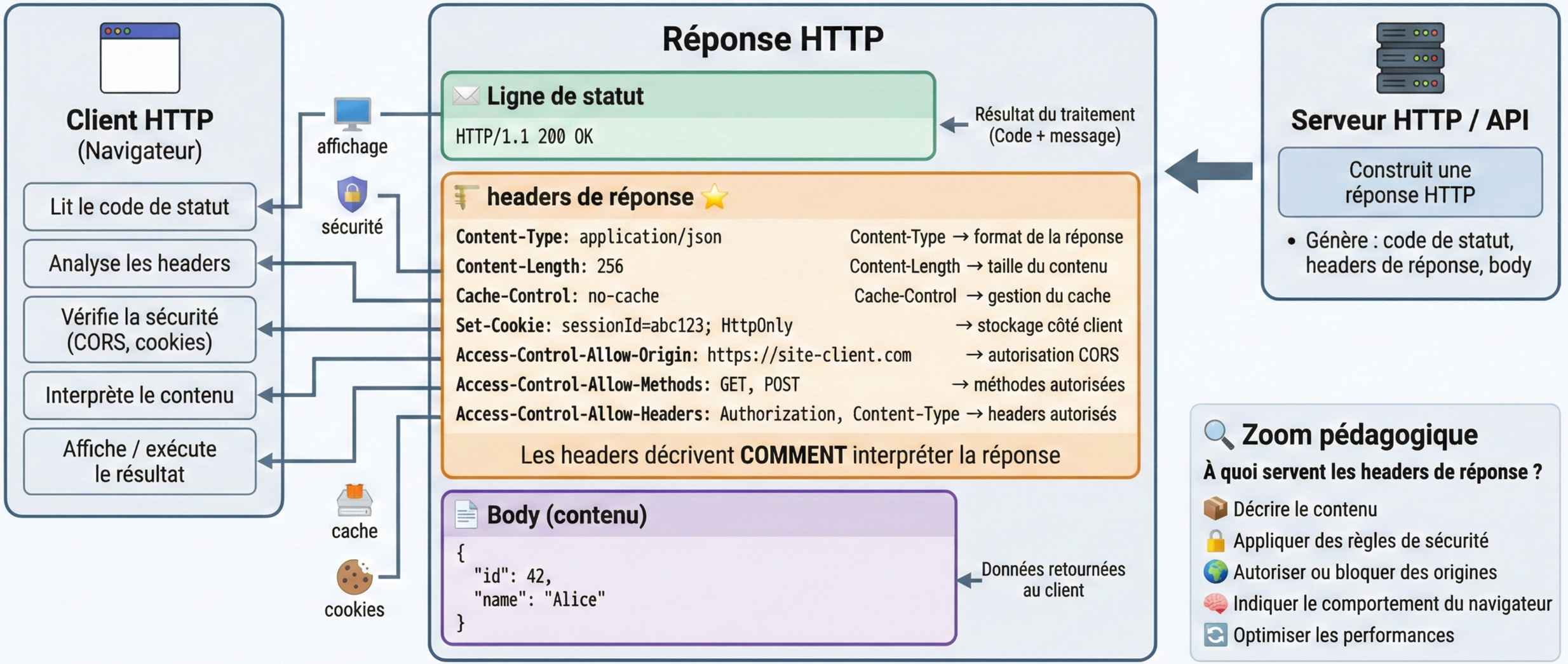
3 Serveur HTTP / API

- Lit la ligne de requête
- Analyse les headers
  - Authentification
  - Format de réponse
  - Routage
- Vérifie la sécurité
- Traite la requête
- Construit la réponse HTTP

🧠 **Message clé** : Les headers HTTP expliquent la requête au serveur pour qu'il sache **QUI** parle, **QUOI** faire et **COMMENT** répondre.

Les headers de réponse HTTP

Comprendre les métadonnées envoyées par le serveur



🔍 Zoom pédagogique

À quoi servent les headers de réponse ?

- 📦 Décrire le contenu
- 🔒 Appliquer des règles de sécurité
- 🌍 Autoriser ou bloquer des origines
- 🧠 Indiquer le comportement du navigateur
- 🔄 Optimiser les performances

🔑 Les headers de réponse HTTP expliquent au navigateur **comment traiter et sécuriser** la réponse du serveur.

L'Anatomie d'une Conversation Web

Étape 1 : La Demande

Comment le client (navigateur) formule et envoie une requête HTTP.

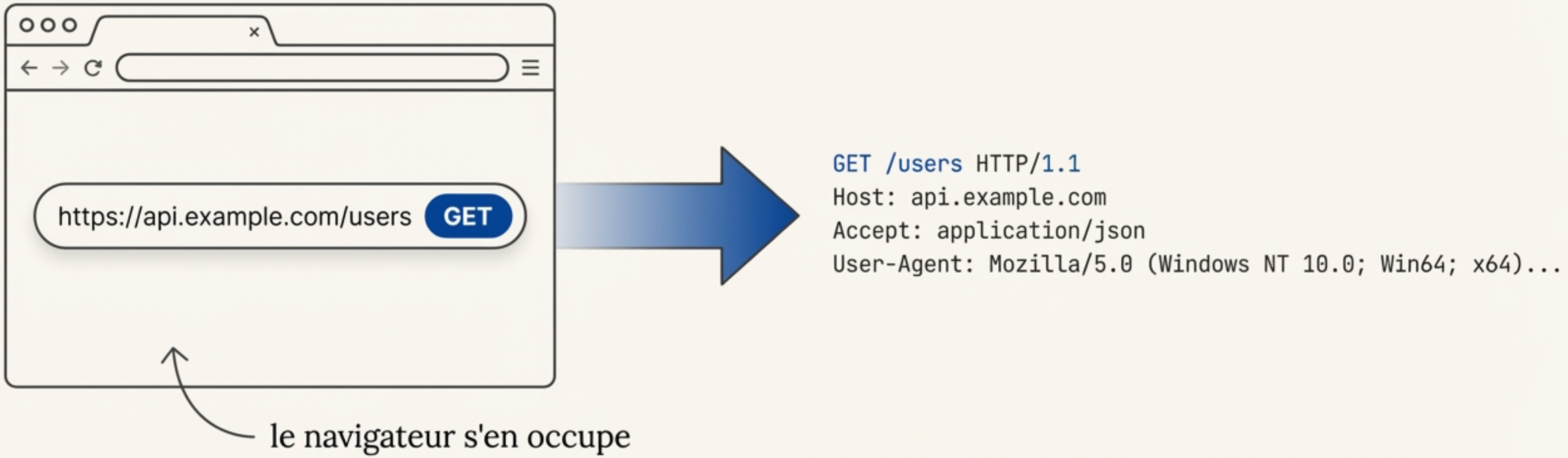
Étape 2 : La Logique du Serveur

Le traitement de la requête par une API Node.js avec le framework Express.

Étape 3 : La Réponse

Comment le serveur construit et renvoie une réponse, et comment le client l’interprète.

Le Point de Départ : Le Client Construit la Requête



Chaque action dans votre navigateur initie la création d'un message structuré destiné au serveur.

Décortiquer la Requête HTTP : Le Langage du Client

```
GET /api/users HTTP/1.1
```

Action demandée au serveur (Méthode + Ressource + Version HTTP)

```
Host: api.example.com
User-Agent: Mozilla/5.0
Accept: application/json
Origin: https://site-client.com
```

Le "passeport" de la requête : d'où vient-elle ?

```
{
  "scope": "user_data"
}
```

Données envoyées (typiquement pour POST / PUT)

Le Mur de Sécurité du Web : La Politique de Même Origine

Par défaut, les navigateurs appliquent une règle de sécurité stricte : un script sur une page web ne peut faire de requêtes qu'à des ressources de la **même origine** (même protocole, même domaine, même port).

Une page sur `https://monsite.com` est bloquée si elle essaie d'appeler une API sur `https://api.partenaire.com`.

Mais le web moderne dépend de ces interactions ! Comment permettre ces communications de manière sécurisée ?

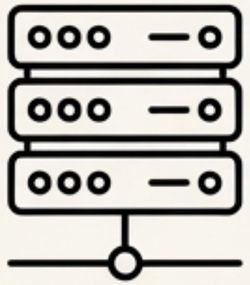
La Solution : CORS (Cross-Origin Resource Sharing)

CORS est un mécanisme basé sur des en-têtes HTTP qui permet à un serveur d'indiquer des origines (domaines) autres que la sienne, à partir desquelles un navigateur devrait être autorisé à charger des ressources.

C'est la clé d'accès qui permet de franchir légalement la 'frontière' entre les domaines.

Le Triangle CORS : La Négociation en 4 Étapes

1. Le Chargement
L'utilisateur charge le code HTML/JS depuis le Serveur A (monsite.com). C'est la situation de départ, de même origine.

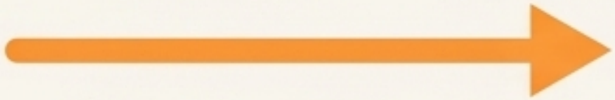


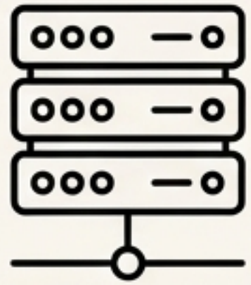
Serveur Web "Site A" (monsite.com)

Client (Navigateur)

2. La Tentative d'Accès
Le JS tente de faire une requête à une API sur le Serveur B. Le navigateur attache le "passeport": l'en-tête

```
Origin: https://monsite.com
```





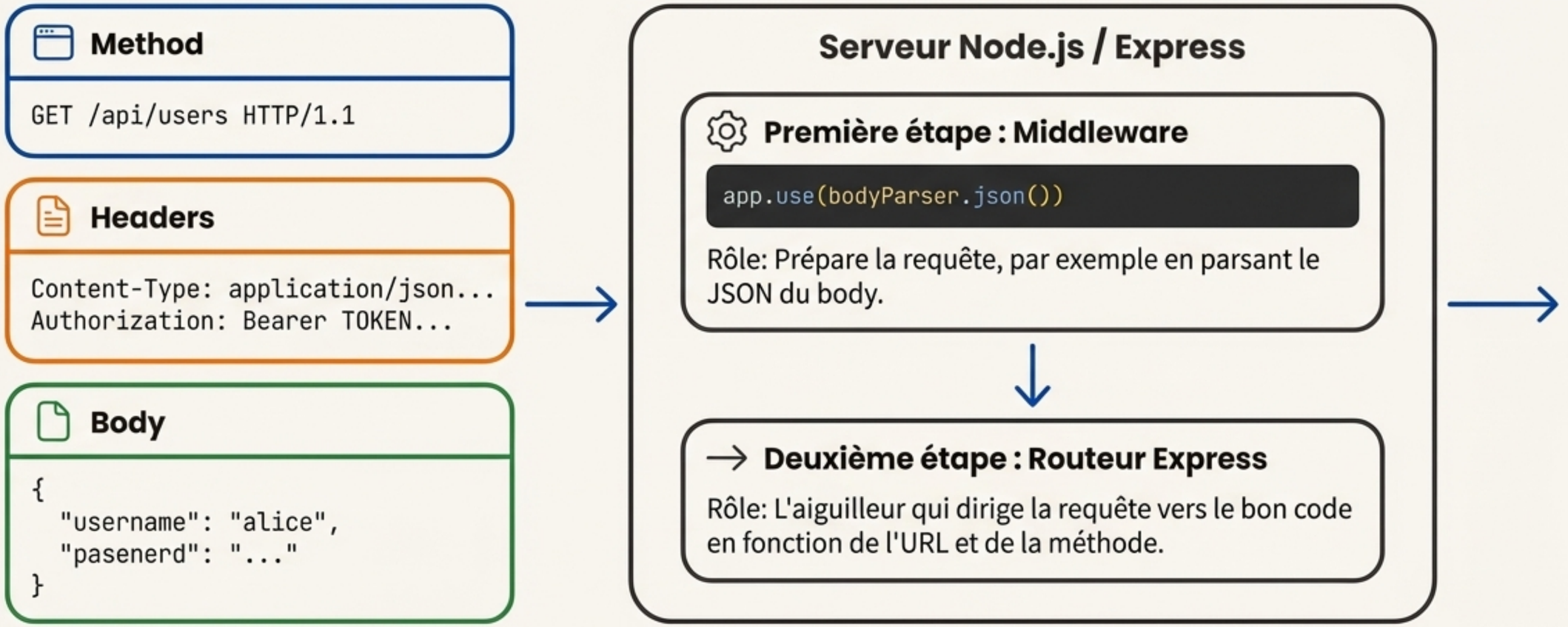
Serveur API "Site B" (api.google.com)

3. L'Autorisation
Le Serveur B reçoit la requête, examine l'en-tête 'Origin' et, s'il l'autorise, renvoie la réponse avec le 'visa": l'en-tête Access-Control-Allow-Origin:

```
Access-Control-Allow-Origin: https://monsite.com
```

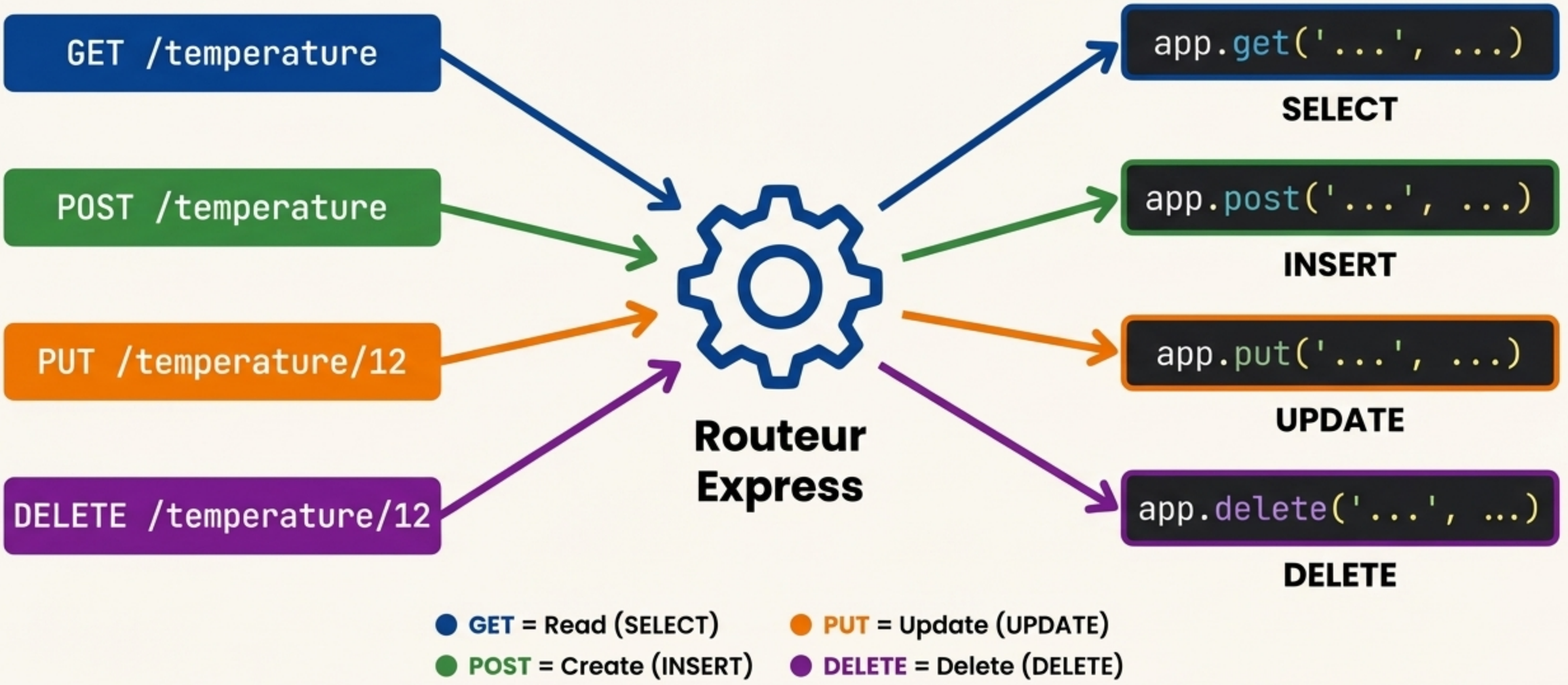
4. Le Contrôle de Sécurité
Le navigateur reçoit la réponse. Il agit comme un agent de sécurité : il compare l'origine du site (monsite.com) avec le 'visa' reçu. Si ça correspond, les données sont transmises au JS. Succès ! Sinon, c'est une erreur CORS.

L'Arrivée au Cœur du Système : Le Serveur Node.js / Express



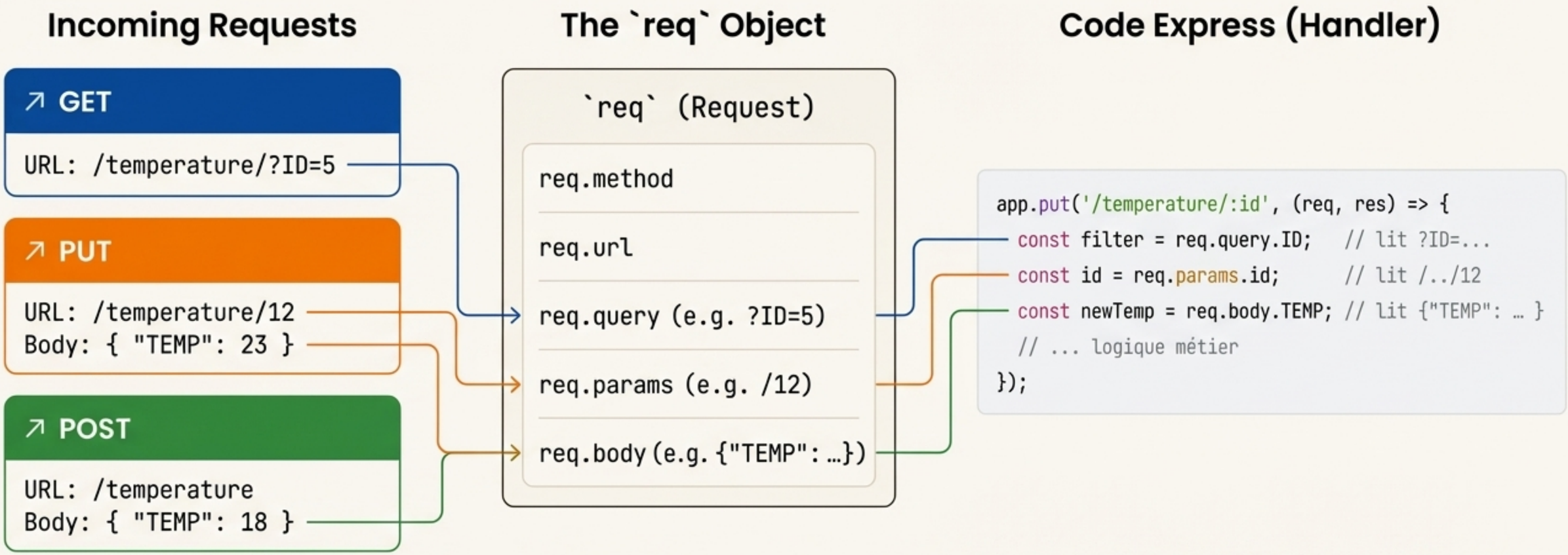
Le serveur n'est pas un monolithe ; c'est une chaîne de traitement organisée.

Le Routeur Express : L'Aiguilleur des Requêtes



Le routeur est le cerveau de l'API, il associe une intention (méthode + URL) à une action (code).

Lire la Requête : L'Objet `req` et ses Propriétés



- `req.params` → paramètres dans l'URL (ex: `/:id`)
- `req.query` → paramètres après '?'
- `req.body` → données JSON envoyées (POST/PUT)

Le Retour du Voyage : Le Serveur Construit la Réponse

Action du serveur

- Génère un **Code de statut** (ex: 200 OK, 404 Not Found)
- Prépare des **Headers de réponse** (Instructions pour le client)
- Construit le **Body** (contenu)

Construit une réponse HTTP

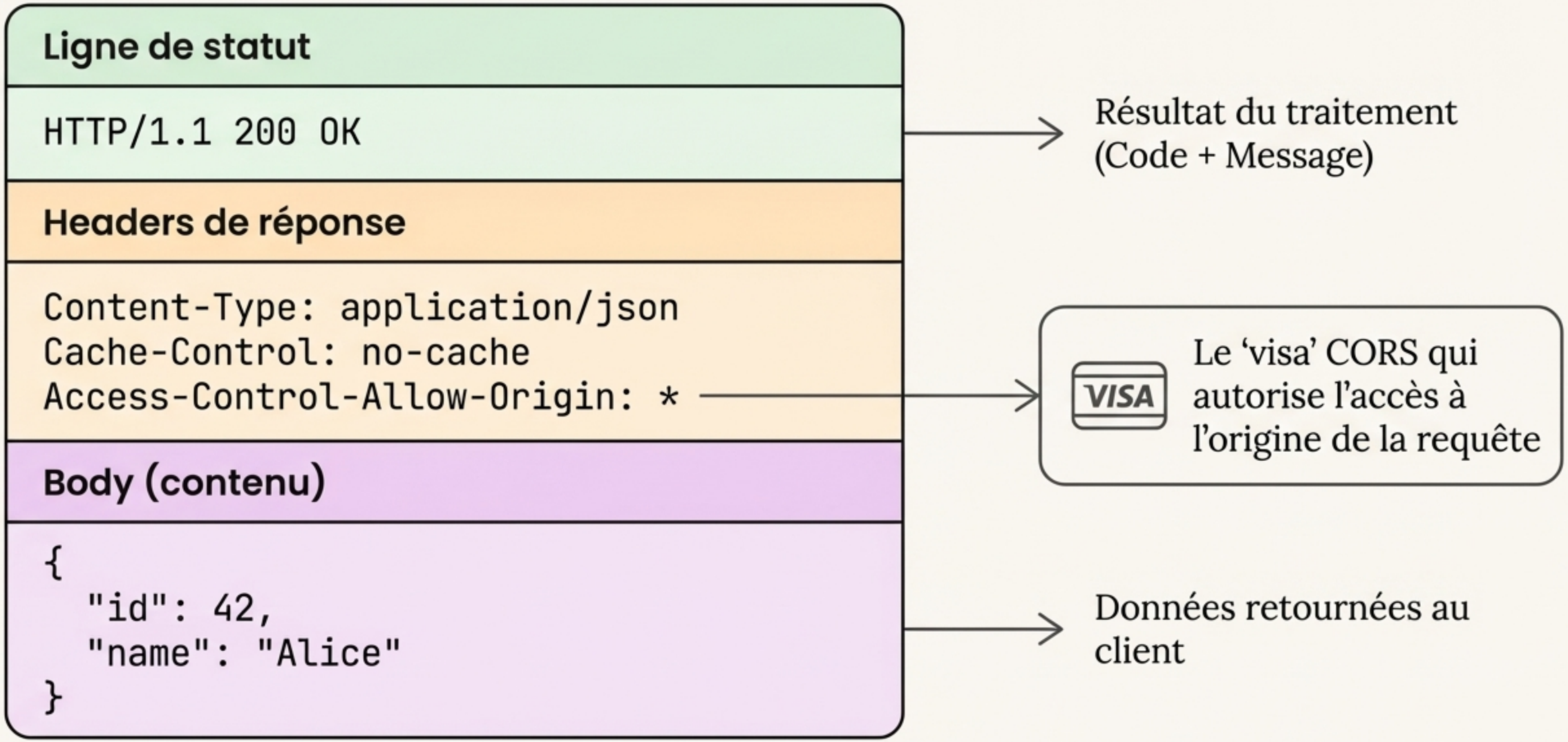
Réponse HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Tue, 13 Apr 2023 1:16:49 GMT

{
  "data": "..."
}
```

La réponse est aussi structurée que la requête, fournissant un retour clair et des instructions précises au client.

Décortiquer la Réponse HTTP : Le Langage du Serveur



Fin du Voyage : Le Client Interprète la Réponse

Réponse HTTP

Ligne de statut

```
HTTP/1.1 200 OK
```

Headers de réponse

```
Content-Type: application/json
Cache-Control: no-cache
Access-Control-Allow-Origin: *
```

Body (contenu)

```
{
  "id": 42,
  "name": "Alice"
}
```

Client HTTP

1. Lit le code de statut : Succès (200) ? Erreur (404, 500) ?

2. Analyse les headers : Y a-t-il des règles CORS à respecter ? Faut-il mettre en cache ?

3. Interprète le contenu : Si c'est du JSON, le parser pour l'utiliser dans l'application.

4. Affiche / exécute le résultat : Mettre à jour l'interface utilisateur avec les nouvelles données.

En Pratique : Activer CORS sur un Serveur Express

Installation

Pour gérer les en-têtes CORS facilement, on utilise le package `cors`.

```
npm install cors
```

Configuration de Base (Tout Autoriser)

Pour activer CORS pour toutes les routes et toutes les origines, ajoutez-le comme middleware.

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

// Middleware CORS qui autorise tout
app.use(cors());

// Vos routes ici...
app.listen(3000);
```

Configuration Avancée : Sécuriser Votre API en Production

En production, il est dangereux de tout autoriser. Configurez CORS pour n'accepter que les origines de confiance.

Autoriser une Origine Spécifique

N'autorisez que votre application frontend.

```
app.use(cors({
  origin: 'https://mon-app-frontend.com'
}));
```

Activer CORS pour une Route Spécifique

Vous pouvez aussi l'activer au cas par cas pour chaque route.

```
app.get('/api/donnees-publiques', cors(), (req, res) =>
  // Cette route est accessible en cross-origin
});
```

Configurer CORS précisément est une étape essentielle pour sécuriser une API et contrôler qui peut utiliser vos données.