サーバとの通信

ウェブ実装で必要になる通信の知識

この資料の目標

Webアプリを支えるサーバとの通信の
しくみを、ざっくりつかむ！

この資料で扱うこと

1. **HTTP** とは
2. **URL** の構成
3. **HTTPメソッド**（GET / POST）
4. **HTTP** の中身（ヘッダ）
5. **CORS** とは

CHAPTER 1

HTTPとは

HTTP は、Webでよく使われる通信のルール（プロトコル）

HTTP ＝ Hypertext Transfer Protocol

HTTPとは

もとはブラウザ⇄サーバ間の通信用。**WebAPIの呼び出し**などにも使われる。

いまの主流は HTTPS

HTTP

password

中身がまる見え

HTTPS

x8#@k2!

暗号化されて読めない

HTTPS は、HTTPの通信を**暗号化**したもの。盗み見や改ざんを防ぐ。

SSL と TLS

- **SSL**：古い暗号化の規格（いまは使われない）
- **TLS**：SSL の後継。実際に使われているのはこちら
- 慣習で、TLS のことを**「SSL」と呼ぶ**ことも多い

つまり **HTTPS ＝ HTTP を TLS で暗号化**したもの。
この資料は HTTP を前提に解説。詳しくは **HTTPS / SSL / TLS** で調べてみよう。

HTTPは「ステートレス」

通信は1回ごとに独立していて、
前回の続きとしては扱われない

ステート（状態）＋ **レス**（ない）。
サーバーは前の通信を覚えていない。たとえば「ログイン成功」も保持しない。

でも「セッションレス」ではない

通信にセッション情報を乗せるので、ログイン状態は保てる

だから HTTP は **ステートレスだが、セッションレスではない。**

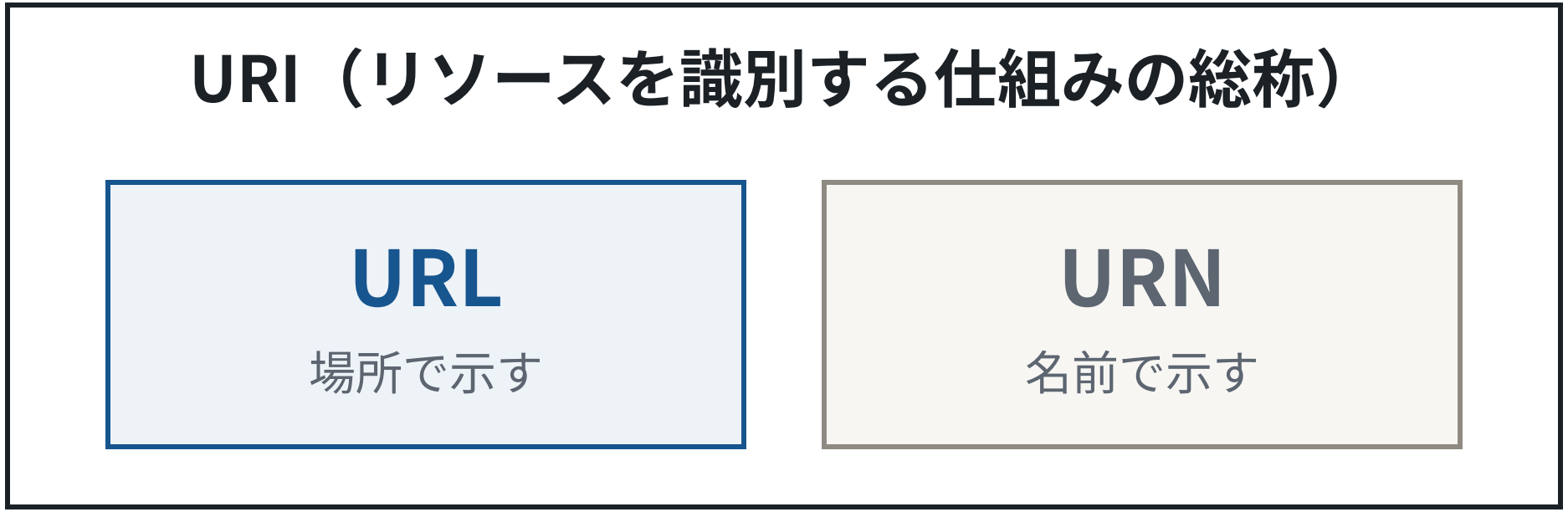
CHAPTER 2

URLの構成

URL は、インターネット上の
リソースへのあて先

URL ＝ Uniform Resource Locator

URL と URI



URI はリソースを識別する仕組みの総称。**URL はその一種**で、実務ではほぼ URL を使う。

URLの構成

```
http://example.com:8080/hoge/index.html?name=Taro#profile
```

- **スキーム**
  通信のプロトコル ・ http
- **ドメイン**
  接続先のアドレス ・ example.com
- **ポート**
  サーバのポート番号 ・ 8080（省略可）
- **パス**
  サーバ内の場所 ・ /hoge/index.html
- **クエリパラメータ**
  URLに乗せて渡すデータ ・ name=Taro
- **アンカーリンク**
  ページ内の特定の場所 ・ #profile

オリジン

スキーム・ドメイン・ポートの3つをまとめて **オリジン** と呼ぶ。

オーソリティ

`//` の後ろ、パスの前までの部分。ホストの前に**認証用のユーザー情報**を含めることもある。
オリジンとは違い、**スキームを含まず・ユーザー情報を含む。**

CHAPTER 3

HTTPメソッド

メソッド ＝ 通信のアクション

仕様上のメソッド

GET / POST / PUT / DELETE など

今回は、よく使う **GET** と **POST** の2つを見る。

GET と POST

GET

データを取得する

- クエリパラメータで送る
- URLに付くので人に見える
- 秘密の情報は送らない

べき等・安全

POST

データを更新する

- ペイロードをbodyで送る
- URLには出ない
- ただし暗号化はしない

べき等でない・安全でない

GET とは

- サーバから**データを取得**する
- クエリパラメータを URL に含められる
- URLは人の目に触れやすい → **個人情報は送らない**

HTTPメソッドとして**べき等・安全**。
ここでの「安全」は、サーバ側へ**更新を要求しない**という意味。

POST とは

- サーバの**データを更新**する
- データは **body に含めて**送る（＝ペイロード）
- URLには出ないが、**暗号化はしない**

HTTPメソッドとして**べき等でなく・安全でない**。
個人情報を扱うなら HTTPS で通信ごと暗号化する。

TIPS：べき等とは

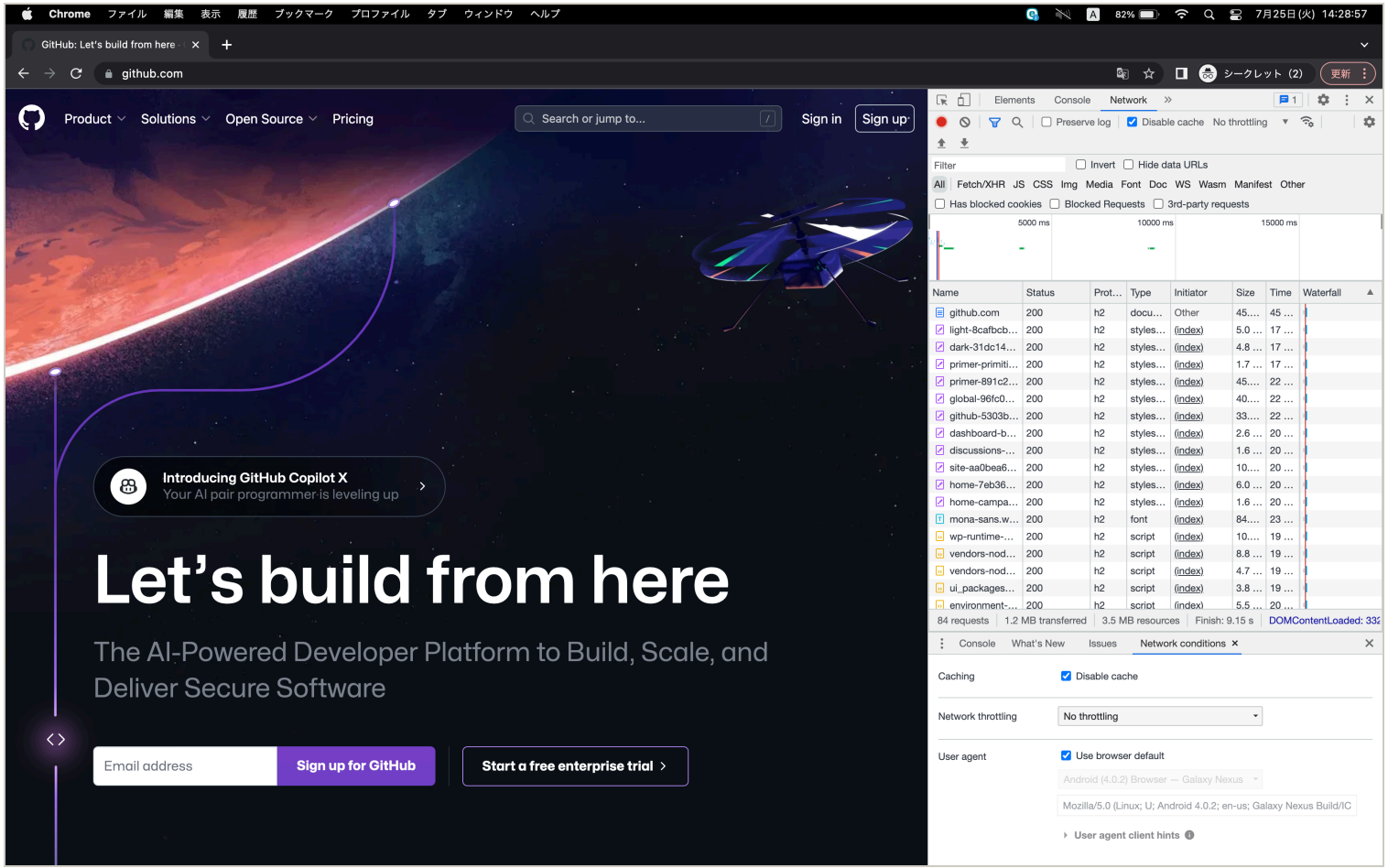
同じリクエストを何回処理しても、
サーバのデータが同じ状態になること

ここまでは**メソッドの定義**の話。
GET/POSTを使ったAPIのべき等性・安全性は、**サーバ/クライアントの実装しだい**。

実際にやってみる（GET）

1. 新しいタブで開発者ツールを開く（githubログイン中ならシークレットウィンドウ）
2. `Network` タブを開いて `github.com` にアクセス
3. 一覧の `github.com` の行がメソッド **GET** になっているか確認
4. リロードやロゴのクリックで内容が変わらないことを確認（**べき等**）
5. 検索窓に `jigintern` など入れて検索
6. `search` を探し、**GET** で呼ばれ、Request URL に入力文字列が含まれることを確認
7. 同じ文字列なら何度でも同じ結果になることを確認

Network タブで確認する



実際にやってみる（POST）

1. github にログインする
2. 一覧から `session` を探し、**POST** で呼ばれていることを確認
3. URLにはパスワードが入っていないのに、**Payload** タブを開くとパスワードが入っていることを確認

まとめ（HTTPメソッド）

- **GET**：クエリで送る／人に見える情報だけ／べき等な処理に
- **POST**：body で送る／秘密の情報に／更新する処理に（暗号化はしない）

メソッドは動作や性質が**定義**されているだけ。
定義どおりに動くよう、**サーバ側が実装する**必要がある。

CHAPTER 4

HTTPの中身

リクエストとレスポンスの構成

リクエスト

- メソッド（GET / POST）
- パス
- プロトコルバージョン
- リクエストヘッダ
- body（POSTのとき）

レスポンス

- プロトコルバージョン
- ステータスコード（200 など）
- ステータスメッセージ（OK など）
- レスポンスヘッダ
- レスポンスボディ（省略可）

主なリクエストヘッダ

ヘッダ	役割
user-agent	ブラウザ・OS の種類
cookie	サーバから預かった情報（セッションなど）
host	接続先のホスト名
accept	受け取れるデータ形式
accept-language	受け取れる言語
accept-encoding	受け取れる圧縮形式

主なレスポンスヘッダ

ヘッダ	役割
content-type	本文のデータ形式
Content-Encoding	本文の圧縮形式
set-cookie	ブラウザに保存させる情報
connection	接続の扱い方
server	サーバのソフトウェア名

ヘッダは基本 **ブラウザが自動で付与**。慣れるまで自作・付与はしなくてよい。

実際に見てみる

GETの中身

- ログイン済みの `github.com` を開く（リロード）
- Nameが `github.com` の行をクリック
- リクエスト/レスポンスヘッダ、レスポンスボディを確認

POSTの中身

- github にログイン（済みなら一度サインアウト → 再ログイン）
- Nameが `session` の行をクリック
- ヘッダやペイロードを確認

まとめ（HTTPの中身）

- リクエストもレスポンスも、**本体以外の情報**をたくさん持つ
- **ヘッダ**から通信の情報を得られる
- ヘッダは任意に足せるが、**基本はやらなくてよい**

CHAPTER 5

CORSとは

CORS は、オリジンをまたいで
データをやり取りするしくみ

CORS ＝ Cross-Origin Resource Sharing

同一オリジンポリシー

同一オリジンポリシー

異なるオリジン間の送受信は、標準ではブロックされる

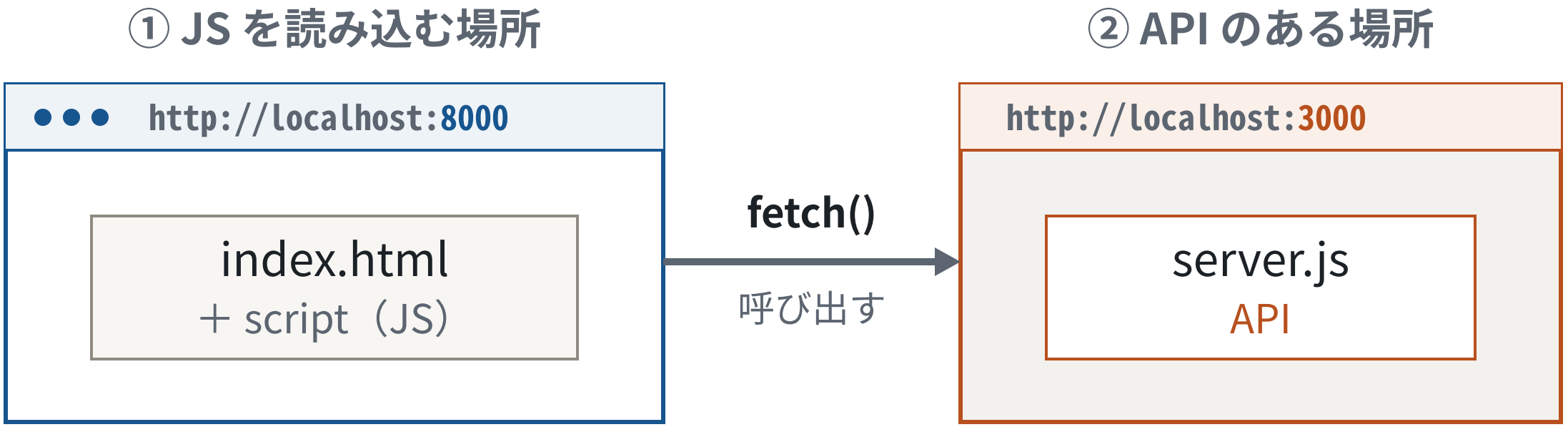
CSRF や XSS といった攻撃を防ぐため、異なるオリジン間の送受信は標準でブロックされる。

なぜ CORS が必要？

HTML/JS を取ってくるオリジンと、
叩くAPIのオリジンが違うことがある

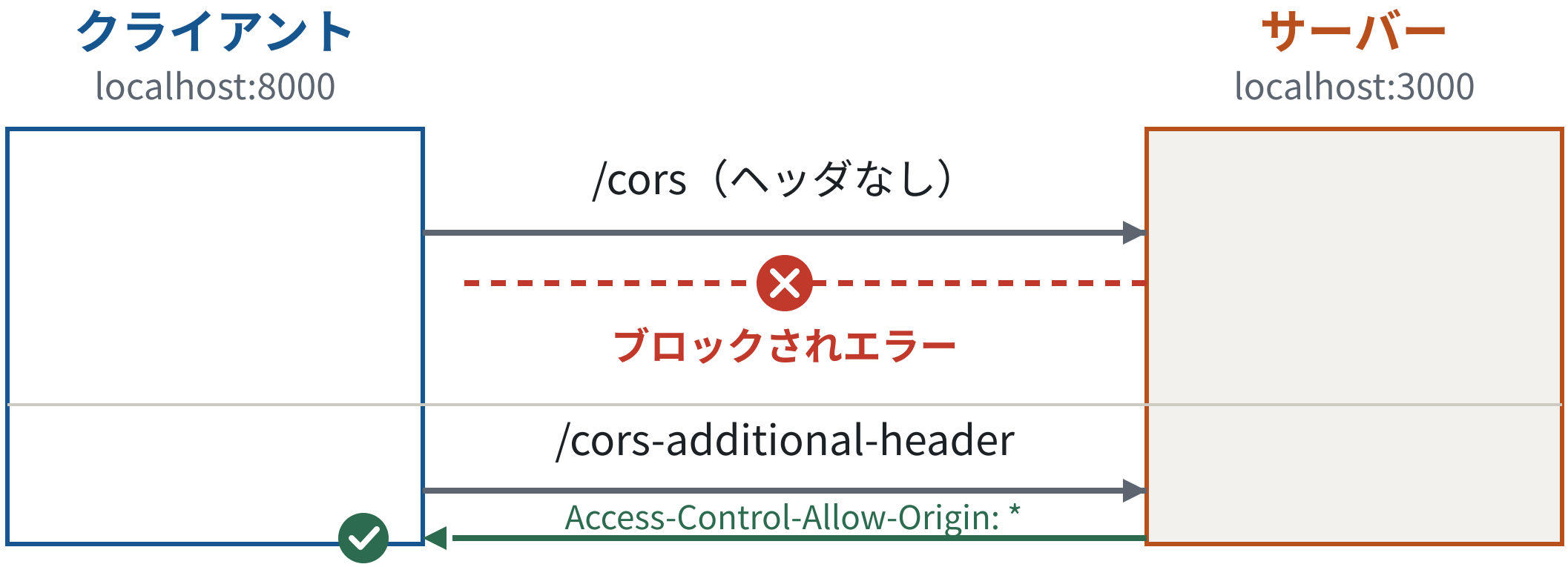
そこで、**安全と認められたオリジン間**だけ、またいで通信できるようにしたのが CORS。

この教材でのしくみ



8000 のページから、その JS が **3000** の API を呼ぶ。ポートが違う＝**別オリジン**なので CORS が必要。

CORSリクエスト



リクエストに `origin`、レスポンスに `Access-Control-Allow-Origin` を付け、両者が一致すれば受け取れる。

プリフライトを起こさない条件（単純リクエスト）

メソッド

GET / HEAD / POST

ヘッダ

Accept / Accept-Language / Content-Language / Content-Type / Range

Content-Type

x-www-form-urlencoded / multipart/form-data / text/plain

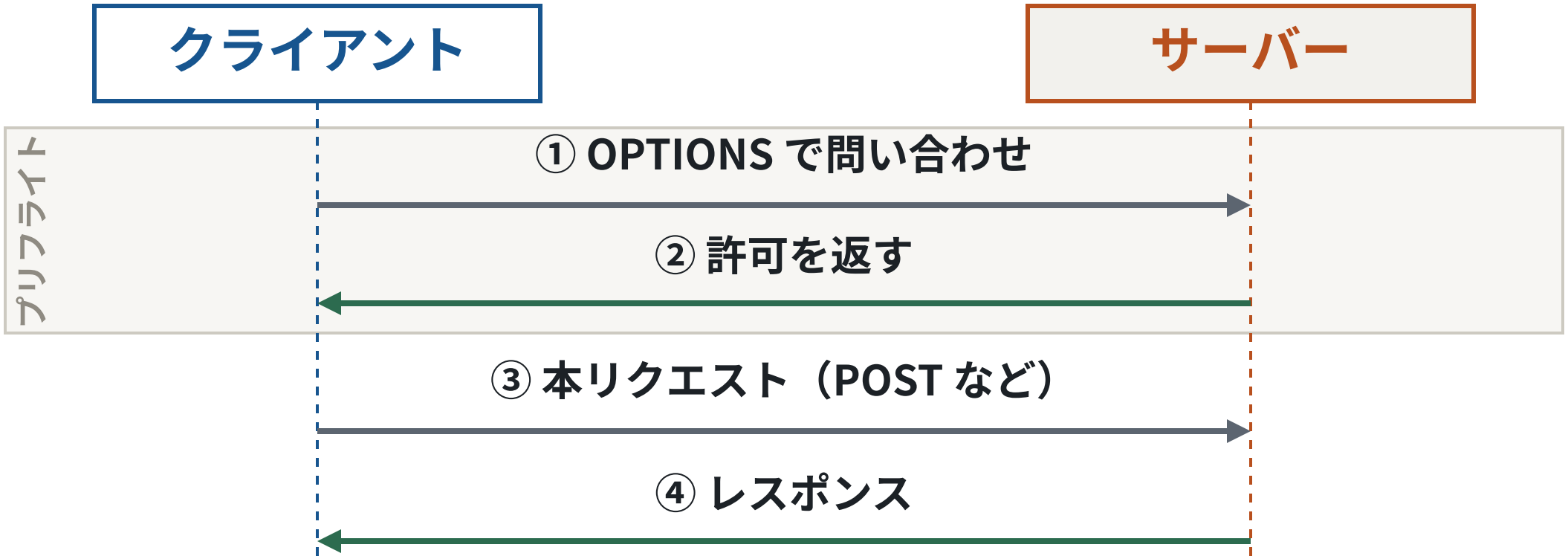
加えて、アップロード監視のイベントリスナーや `ReadableStream` を使っていないこと。

実際に見てみる（単純リクエスト）

1. `/transfer-protocol` に移動
2. `deno run server.js` を実行（サーバは **3000番**）
3. 別ターミナルで `deno run client.js` を実行（クライアントは **8000番**）
4. `http://localhost:8000` を開き、開発者ツールを開く
5. `try cors` ボタン → Network / Console を確認（**エラー**）
6. `try cors (simple request)` ボタン → 正常に受け取れることを確認

`/cors` はヘッダなし、`/cors-additional-header` は `Access-Control-Allow-Origin: *` を付けて返している。

プリフライトリクエスト



条件を満たせない（主にユーザのデータに影響する）通信では、先に `OPTION` で安全性を確認してから本リクエストを送る。

実際に見てみる（プリフライト）

1. `/transfer-protocol` に移動
2. `deno run server.js` を実行
3. 別ターミナルで `deno run client.js` を実行
4. `http://localhost:8000` を開き、開発者ツールを開く
5. `try cors (prefright request)` ボタン → Network / Console を確認

POST で `Content-Type: application/json` を持つため単純リクエストにならず、先にプリフライトで必要な許可を確認している。

まとめ（CORS）

- オリジンをまたぐアクセスを許すなら **CORSの設定**を行う
- 条件を満たせないと **プリフライト**（安全確認の通信）が起きる
- **外部のWebAPI** を呼ぶときは CORS の設定が必要になることがある

FINALE

実装で意識すること

まとめ

HTTPメソッド
更新・個人情報なら POST、取得だけなら GET

ヘッダ
基本はブラウザが自動で付与。自分で足さなくてよい

CORS
外部APIを叩くときに設定。fetch なら mode を cors に

まずは GET / POST から。
残りは実装に慣れてきてから使えばよい

実装のときは

- メソッドは、更新・個人情報なら **POST**、取得だけなら **GET**
- サーバ側は、原則そのメソッドの**仕様を守って**実装する
- ヘッダは基本ノータッチ。**ブラウザ任せ**でよい

自作サーバは別オリジンから叩かれないので、サーバ側のCORSはあまり気にしなくてよい。
クライアントで外部APIを叩くときは、`fetch` の **mode を cors** にすればおおむね動く。

おわりに

もっと深く知りたい人は、参考文献で自学してみよう ◎

参考文献

- HTTPの概要（MDN）
- HTTPメソッドについて（MDN）
- GETとPOSTの違いについて（Qiita）
- CORSについて（MDN）
- CORSを絶対に理解する（Zenn）