

JS RPCを理解する

Cloudflare Meet-up Osaka #4

May 17, 2024. Yusuke Wada

自己紹介

- Yusuke Wada
- Web framework developer
- Developer Advocate @Cloudflare
- Bokete co-founder
- Creator of Hono
- <https://github.com/yusukebe>



<https://hono.dev/>

自己紹介キーワード

Emerging Technology and Incubation

- 部署に当たる
- いわゆる研究開発ではない
- Workersを中心とした開発者プラットフォームを担当する

Developer Relations

- ETIの中のひとつのチーム
- 開発者プラットフォームとそれを使う開発者の関係について
- 現在10名程度
- Developer Advocate、フォーラムマネージャー、Discordマネージャー、Educator、エンジニア

Developer Advocate

- 開発者と自社のプロダクトをつなぐ
- 現在和田を含めて3人
- YouTubeなどビデオ講座の作成
- デモアプリの作成
- チュートリアル執筆
- イベント開催
- イベント登壇など

Hono

- 和田が作っているWebフレームワーク
- Cloudflare Workersを含め様々なJavaScriptランタイムで動作する
- Developer Advocateの一環で作成
- Cloudflareの社内プロダクトやチュートリアルで使われている

超前置き

Cloudflare 開発者プラットフォーム

Cloudflare 開発者プラットフォーム



開発者向け
製品

コンピューティング



Workers
Bundled



Cron
Triggers



Workers
Unbound

データ ストレージ



Workers
KV



Durable
Objects



D1
データ
ベース



Hyperdrive



Vectorize
(ベクトル
DB)



R2 オブ
ジェクト
ストレージ



サードパーティ
接続



Queues

開発者向けサービス



Pages



Stream



プラット
フォーム



Images



Workers
分析エンジン



可観測性
ツール



プラット
フォーム

Cloudflare ネットワーク
インフラストラクチャ



グローバル エッジ：300以上の都市、インターネットユーザの95%から50ミリ秒以内、13000 の相互接続、248 Tbps の容量、China Network



構成要素：SSL/TLS、mTLS、権威/再帰 DNS、DNSSEC、DNS over HTTP、L4-7 over Wireguard



コンプライアンス/プライバシー：ISO、SOC、PCI、GDPR 準拠、ログと分析、Data Localization Suite

アジェンダ

1. 前置き
2. まずBindingsを理解する
3. Service Bindings
4. JS RPC
5. 例
6. Smart Placementとの未来
7. マーケットプレイス？

1. 前置き

**この発表はCloudflare Workersの新機能
「JS RPC」について理解するためのものです
(わりと概念的な話です)**

RPCとは？

RPCとは

- **Remote Procedure Call**
- **遠隔（リモート）にあるコンピュータの関数（プロシージャ）を実行（コール）する手法**
- **例**
 - **gRPC**
 - **JSON-RPC**
 - **XML-RPC**

ようは（Worker）の呼び出し方についての話をします

Kenton氏のブログ

- 2024年4月 Developer Weekで発表された2つのブログ記事
- 「Why Workers environment variables contain live objects」
- 「We've added JavaScript-native RPC to Cloudflare Workers」
- AI製品のアップデートやD1のGAに隠れたテクニカルな記事
- これから紹介するJS RPCを理解する鍵となる

“Why Workers environment variables contain live objects”

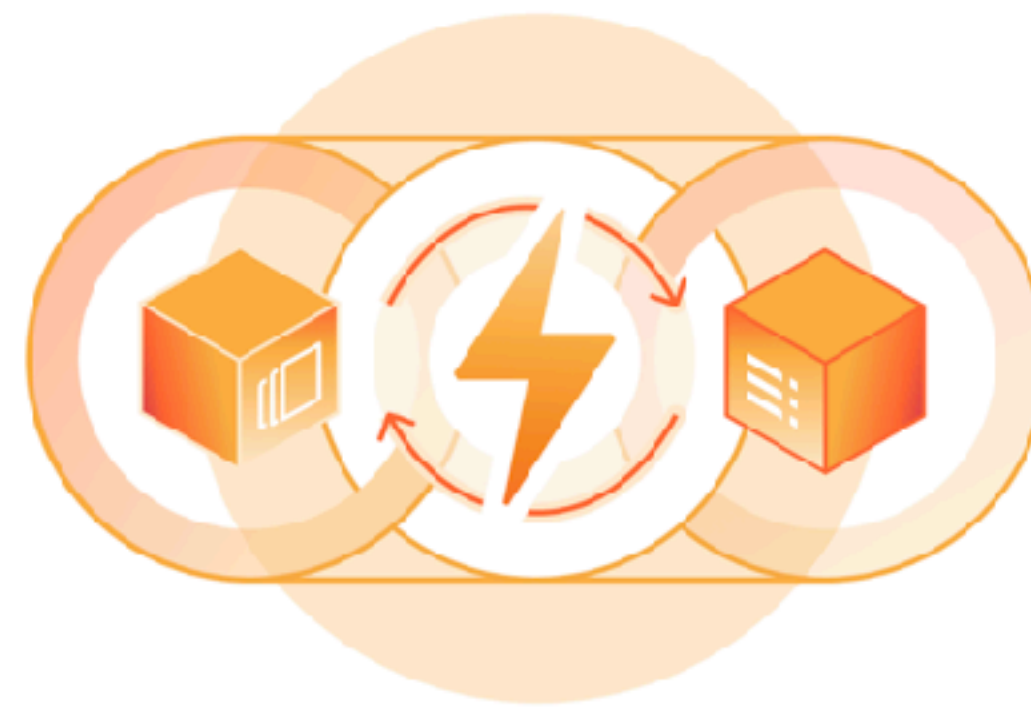
Why Workers environment variables contain live objects

04/01/2024



Kenton Varda

16 min read



<https://blog.cloudflare.com/workers-environment-live-object-bindings>

“We've added JavaScript-native RPC to Cloudflare Workers”

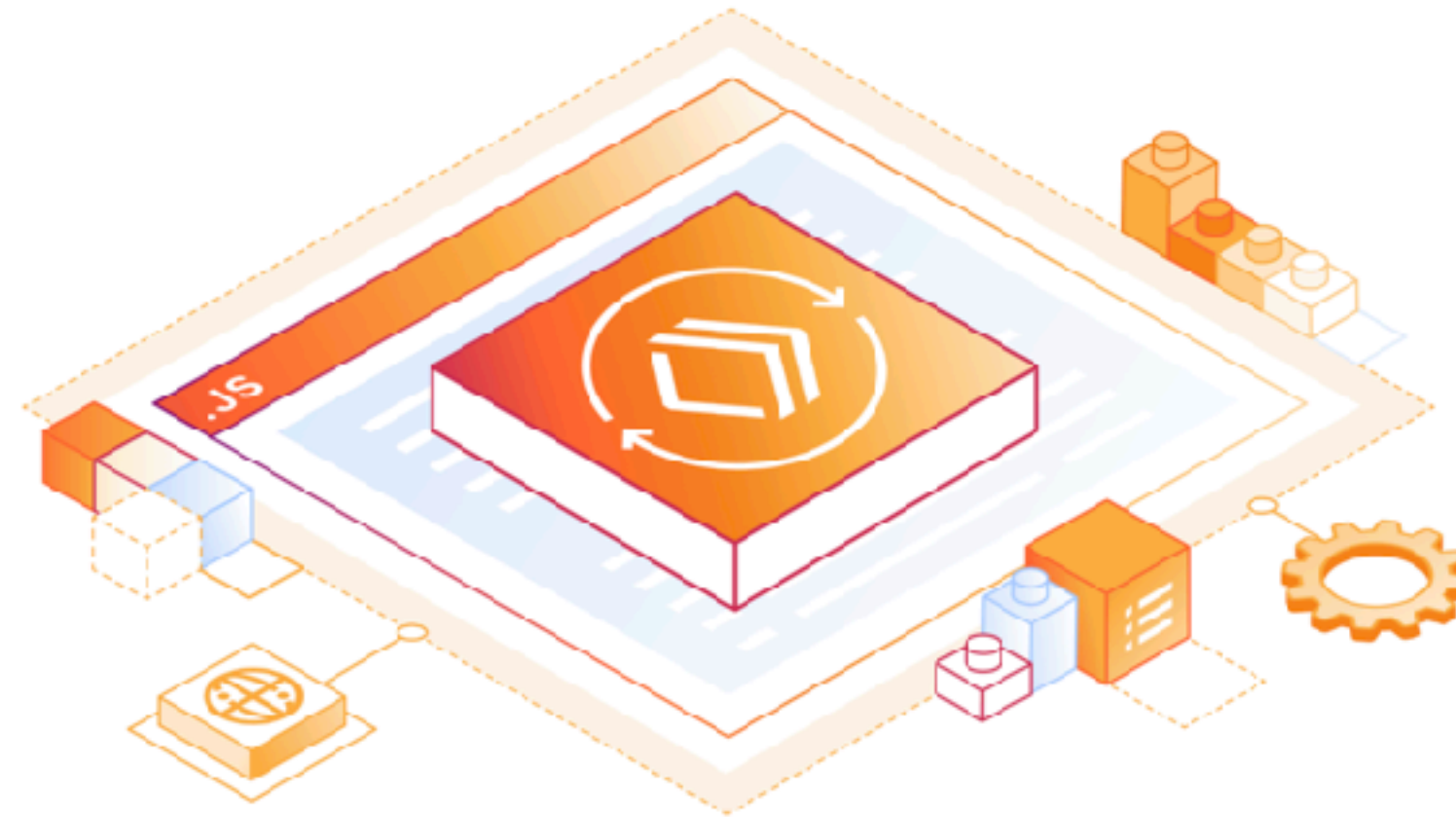
We've added JavaScript-native RPC to Cloudflare Workers

04/05/2024



Kenton Varda

13 min read



<https://blog.cloudflare.com/javascript-native-rpc>

これから話すことはこのブ
ログ記事に基づく

2. まず Bindings を理解する

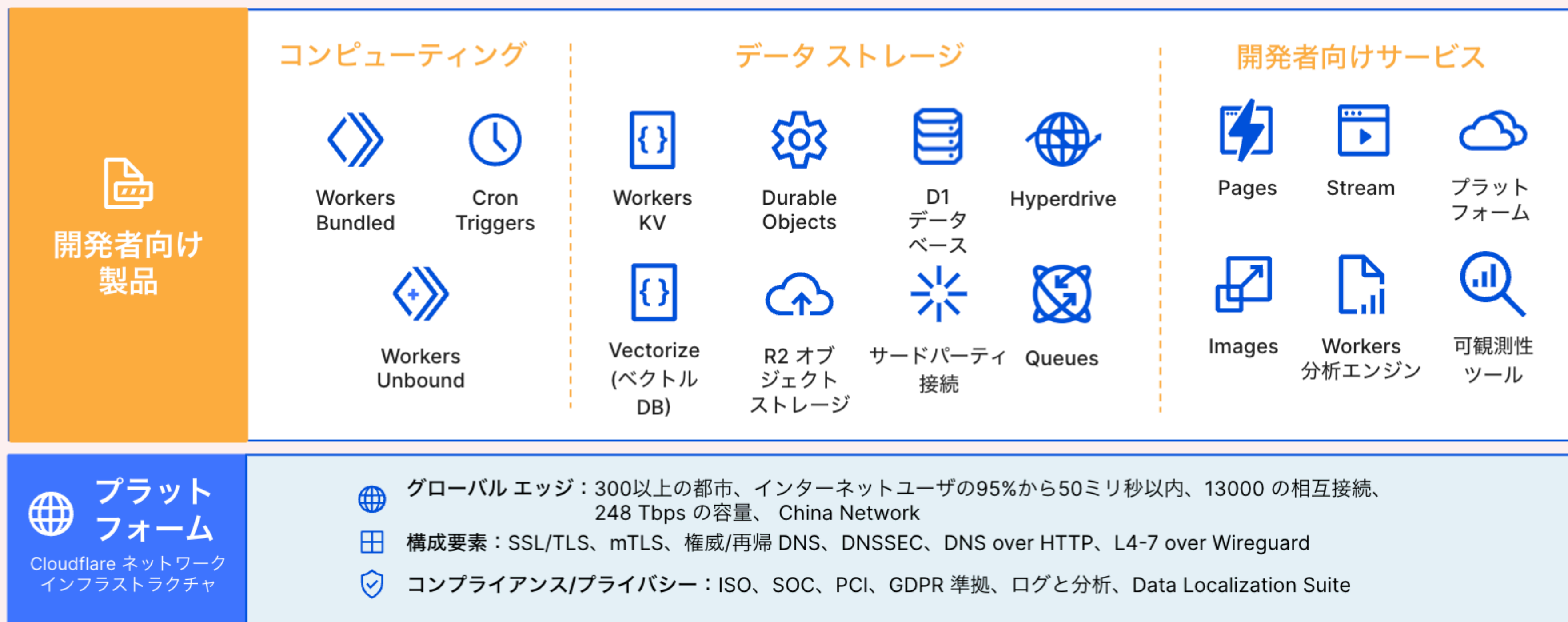
Bindingsとは？

- CloudflareのプラットフォームへWorkersからつなぐための概念と手段
- 例
 - AI、Analytics Engine、Browser Rendering、D1、Environment Variables、Hyperdrive、KV、Queues、R2、Rate Limiting、Vectorize

各種プロダクト

Cloudflare 開発者プラットフォーム

Cloudflare 開発者プラットフォーム



KVを使う場合の実装



```
export default {  
  async fetch(request, env, ctx) {  
    return new Response(await env.MY_KV.get('content'))  
  }  
}
```

env.MY_KV

KVオブジェクトそのものになっている

例: `get()` / `put()` / `delete()` が使える



```
export default {  
  async fetch(request, env, ctx) {  
    return new Response(await env.MY_KV.get('content'))  
  }  
}
```

他に書くことはwrangler.tomlの設定だけ



wrangler.toml

```
[[kv_namespaces]]
```

コード内でどのような名前呼び出すか？

```
binding = "MY_KV"
```

```
id = "KVデータベースを識別するID"
```

作ったKVのID

もし、 Bindingsを使わな
かったらどうなるか？



SDKをimport

```
// SDKのクライアントになるのかな??  
import { KV } from 'cloudflare:kv'
```

外部APIへ接続

```
export default {  
  async fetch(request, env, ctx) {  
    // データベースにつなぐのにシークレットキーが必要??  
    // それらは環境変数から来るのか??  
    const myKv = KV.connect('my-kv-namespace', env.MY_KV_AUTHKEY)  
    return new Response(await myKv.get('content'))  
  }  
}
```

環境変数からキーを取得

取得したオブジェクトで操作

2つの問題

1. セキュリティ - キーの管理、キーの処理を書かなくてはならない
2. 開発者体験 - SDKを使わないといけない、`connect`の処理は必要

2つを解決しているのがBindingsである



1. 認証キーを書く必要がない

```
export default {  
  async fetch(request, env, ctx) {  
    return new Response(await env.MY_KV.get('content'))  
  }  
}
```

2. SDKでオブジェクトを作らず直接JSのメソッドで呼べる

3. Service Bindingsとは？

Worker AからBをつなぐ

Worker A

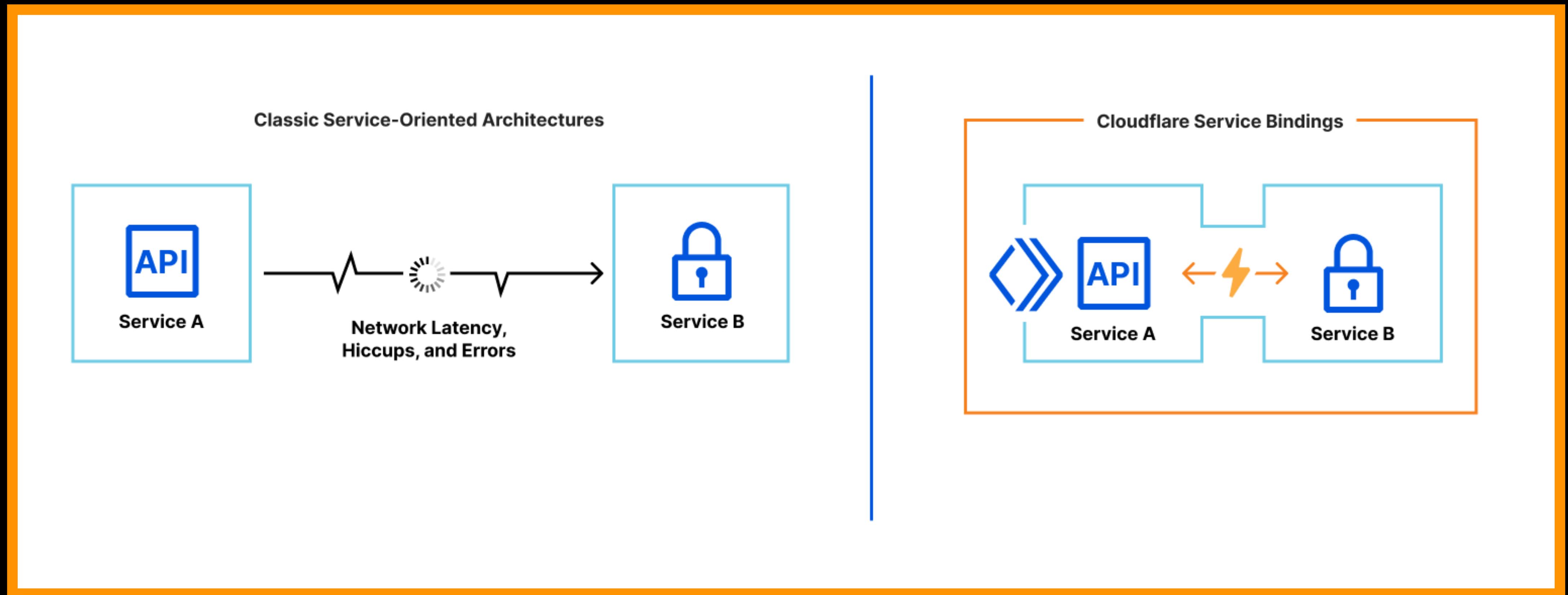


内部通信なので速い

Worker B

Cloudflareネットワーク

公式の図



<https://developers.cloudflare.com/workers/runtime-apis/bindings/service-bindings/>

実装

- インターナルなfetchをかける

```
export default {
  async fetch(request, env, ctx) {
    const authResponse = await env.AUTH_SERVICE.fetch('https://auth/getUser', {
      headers: request.headers,
    })
    const userInfo = await authResponse.json()
    return new Response('Hello, ' + userInfo.name)
  },
}
```

Service Bindingsのユースケース

- ファイル容量制限を避ける
- 責務の分割
- マイクロフロントエンド

Cloudflare Workers and micro-frontends: made for one another

10/20/2022



Peter Bacon Darwin



James Culveyhouse



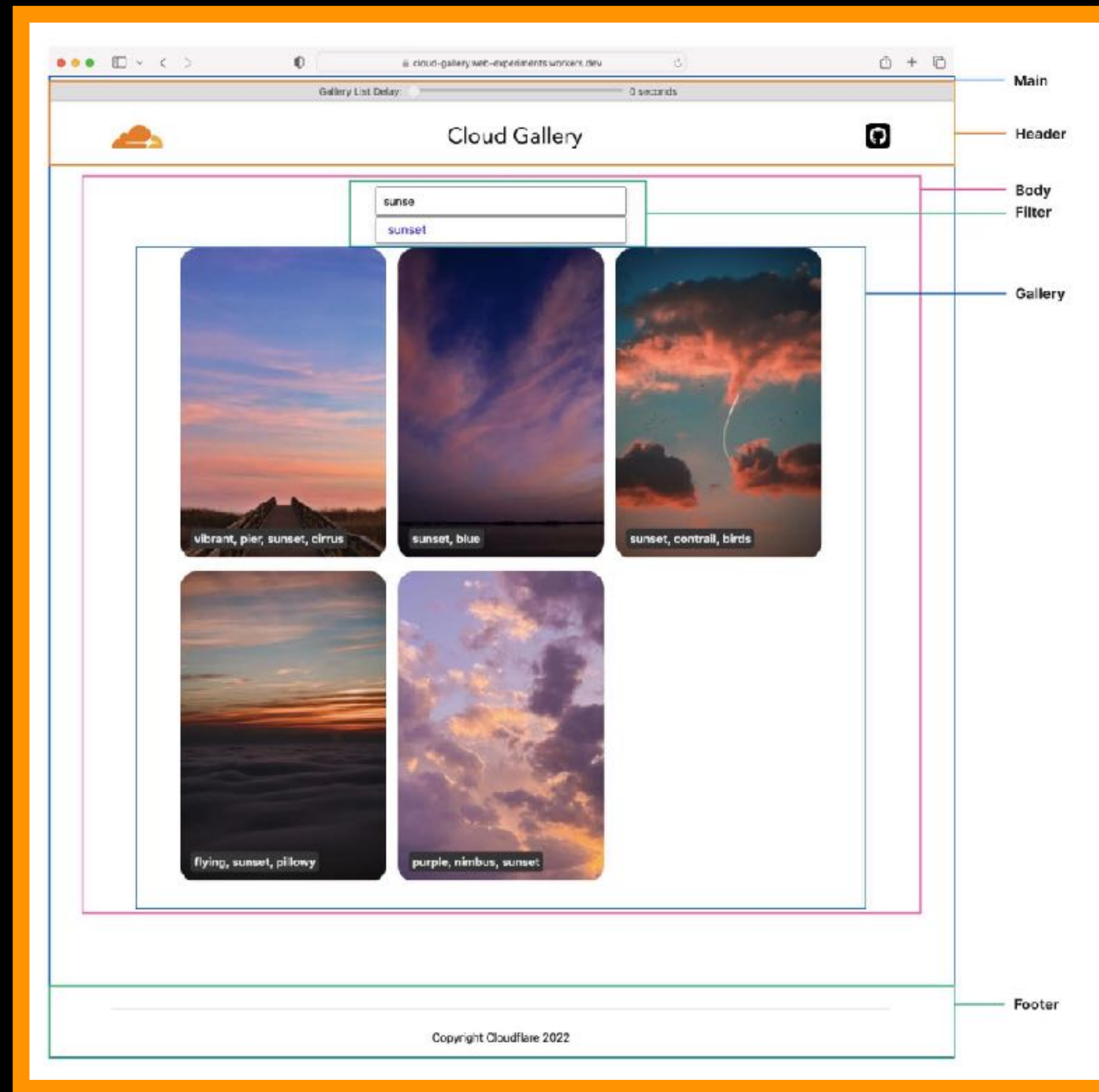
Igor Minar

12 min read

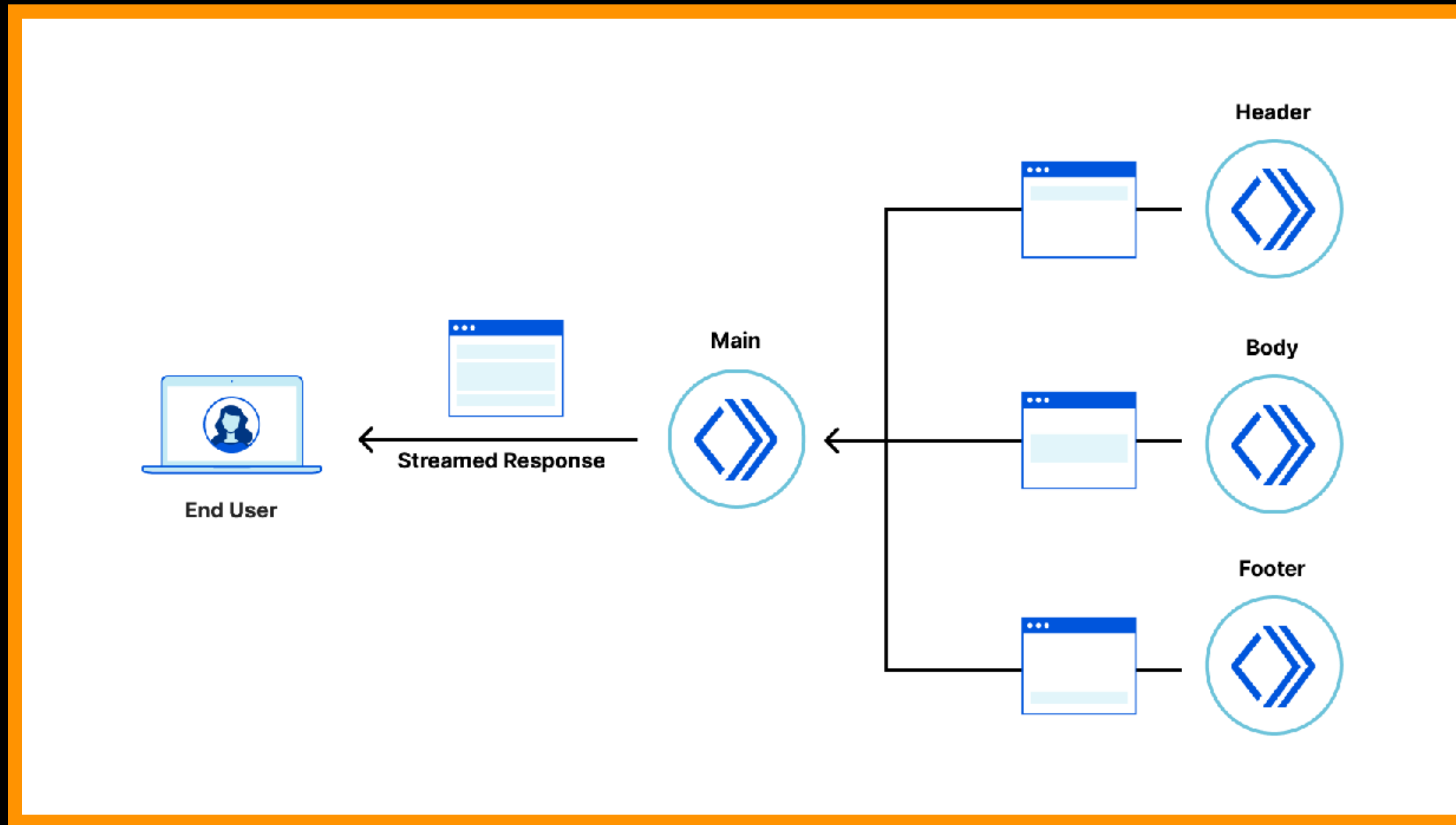
This post is also available in [简体中文](#) and [繁體中文](#).

<https://blog.cloudflare.com/better-micro-frontends>

Webページをパーツに分ける



それぞれのパーツをWorkerにしてServiceBindingsでつなぐ



よい。が、実はそんなに使
われていない

使い勝手が悪い…？

4. JS RPC

Kentonのブログ記事と共に公開

[Products](#) > [Workers](#) > [Runtime APIs](#) > [Remote-procedure call \(RPC\)](#)

Remote-procedure call (RPC)

Workers provide a built-in, JavaScript-native [RPC \(Remote Procedure Call\)](#) [↗](#) system, allowing you to:

- Define public methods on your Worker that can be called by other Workers on the same Cloudflare account, via [Service Bindings](#)
- Define public methods on [Durable Objects](#) that can be called by other workers on the same Cloudflare account that declare a binding to it.

The RPC system is designed to feel as similar as possible to calling a JavaScript function in the same Worker. In most cases, you should be able to write code in the same way you would if everything was in a single Worker.

<https://developers.cloudflare.com/workers/runtime-apis/rpc/>

JS RPCを理解するための2つのアプローチ

1. Bindings - WorkerをカスタムBindingsとして自分でつくれる
2. Service Bindings - Service Bindingsが簡単に書ける

この2つを満たすのがJS RPC

Worker A

Bindingとして呼び出せる



Service Bindingsで繋いでる

Worker B

「電卓」の場合

1. Bindings - 「電卓」という Binding（Workerで実装される）を作る
2. Service Bindings - 「電卓」という Workerを簡単に呼び出せる

cnv. CALC.add(1, 2)

Worker A

Bindingとして呼び出せる



Service Bindingsで繋いでる

電卓

return a+b

実装方法

電卓の実装 - 呼び出し可能なWorkerを作る



TS projects/calc/src/index.ts

```
import { WorkerEntrypoint } from 'cloudflare:workers'

export default class Calc extends WorkerEntrypoint {
  async add(a: number, b: number) {
    return a + b
  }
}
```



WorkerEntrypointをextendsしたクラスをexportするだけ

呼び出ししてみよう

1. wrangler.tomlに定義を追加する



wrangler.toml

```
[[services]]
```

コード内でどのような名前呼び出すか？

```
binding = "CALC"
```

```
service = "calc"
```

呼び出すWorkerの名前

2. Workerから呼び出す



TS projects/server/src/index.ts



```
export default {  
  async fetch(request, env) {  
    const result = await env.CALC.add(1, 2)  
    return new Response(`Result is ${result.toString()}`)  
  }  
} satisfies ExportedHandler<Env>
```

つまり…

1. CalcというサービスをCALCというBindingとして利用する
2. 呼び出すWorkerとCalc間はService Bindingsで接続される

さらに、JavaScriptネイティブな呼び出しができる

=> 「JS Native RPC」

TypeScriptの型がちゃんとつく

```
import type Calc from 'calc'

interface Env {
  CALC: Service<Calc>
}

export default {
  async fetch(request, env) {
    // ...
  }
} satisfies ExportedHandler<Env>
```

JS RPCを使う
と嬉しいこと

JS RPCだとBindingsの2つを享受できる

1. セキュリティへの考慮 - 呼び出しにはシークレットが必要ない
2. 開発者体験 - Bindingとして特にJS/TSネイティブに呼べる

5. 例

ブログサービス

- DBを扱うサービスとそれ呼び出すUI

`cnv.BLOG.createPost()`

Web

Bindingとして呼び出せる



Service Bindingsで繋いでる

Blog

`db.execute(`INSERT INTO Posts ...`)`

コード例

TS projects/web/src/index.ts

```
import { zValidator } from '@hono/zod-validator'
import { z } from 'zod'

// ...

const app = new Hono<Env>()

const schema = z.object({
  title: z.string(),
  body: z.string()
})

app.post('/', zValidator('form', schema), async (c) => {
  const { title, body } = c.req.valid('form')
  await c.env.BLOG.createPost({
    title,
    body
  })
  return c.redirect('/blog')
})

export default app
```

TS projects/blog/src/index.ts

```
import { WorkerEntrypoint } from 'cloudflare:workers'
import { drizzle } from 'drizzle-orm/d1'
import { posts } from '../schema'

interface Env {
  DB: D1Database
}

export default class Blog extends WorkerEntrypoint<Env> {
  async createPost({ title, body }: { title: string; body: string }) {
    const db = drizzle(this.env.DB)
    const uuid = crypto.randomUUID()
    return await db
      .insert(posts)
      .values({
        id: uuid,
        title,
        body
      })
      .returning()
  }
}
```

6. Smart Placementとの未来

実はこのJS RPC

2つのメリットに加えてもうひとつアツいのが...

Smart Placement

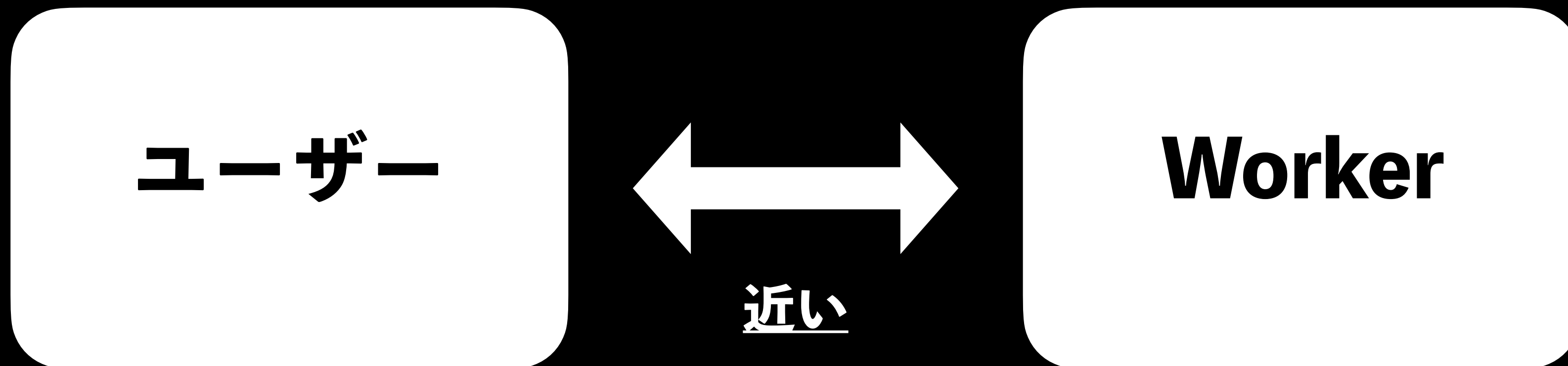
と一緒に使う件

Smart Placementとは？

- ユーザー側よりバックエンド（オリジン）にWorkerが近い方が効率的なことがある
- Workerの場所をよしなにしてくれる

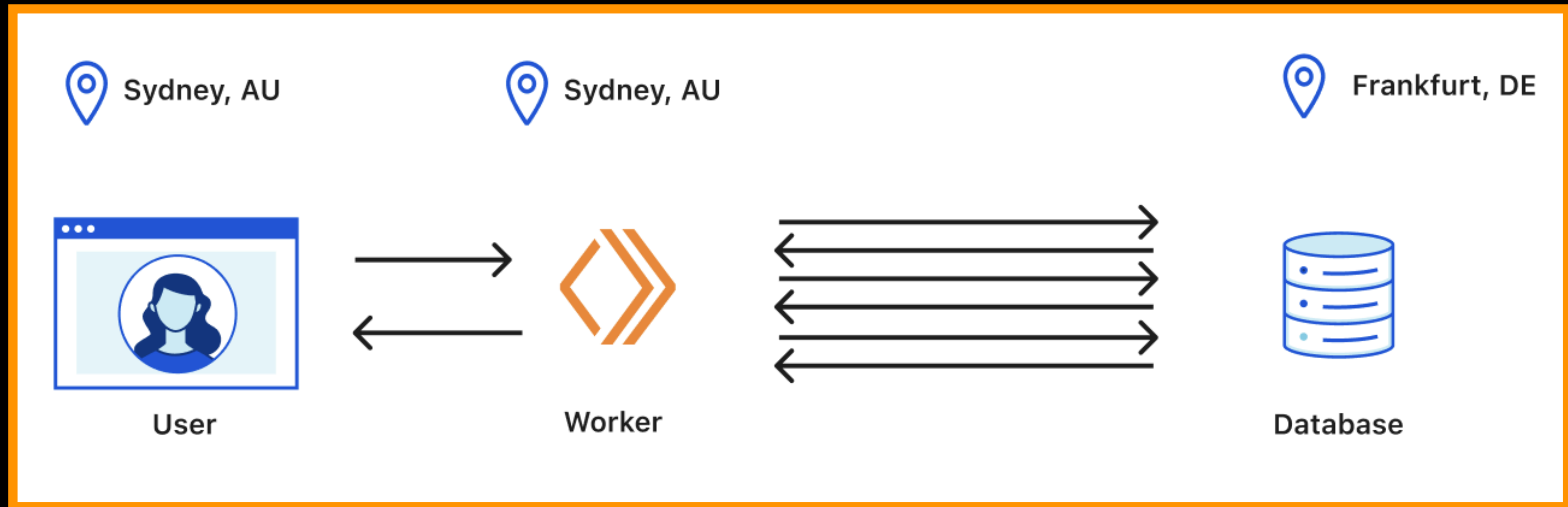
いわゆる「エッジ」のよさ

- ユーザーとWorkerが近い



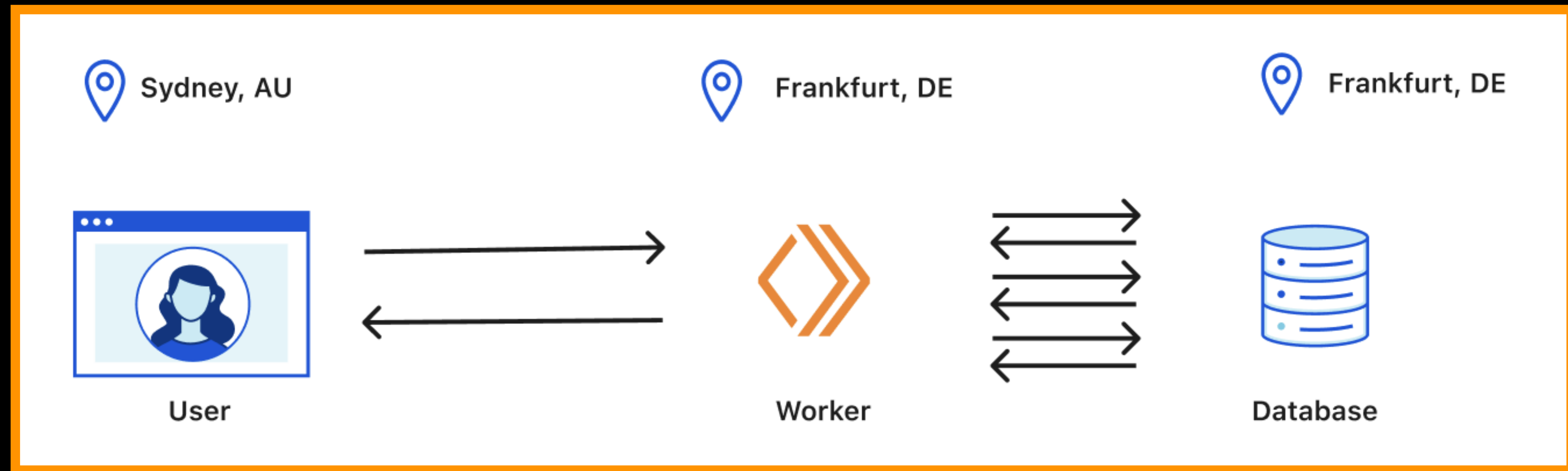
Workerがユーザーに近い けどデータベースからは遠い

- データベースとのやりとりが多い場合は非効率



Workerをデータベースの近くに置こう

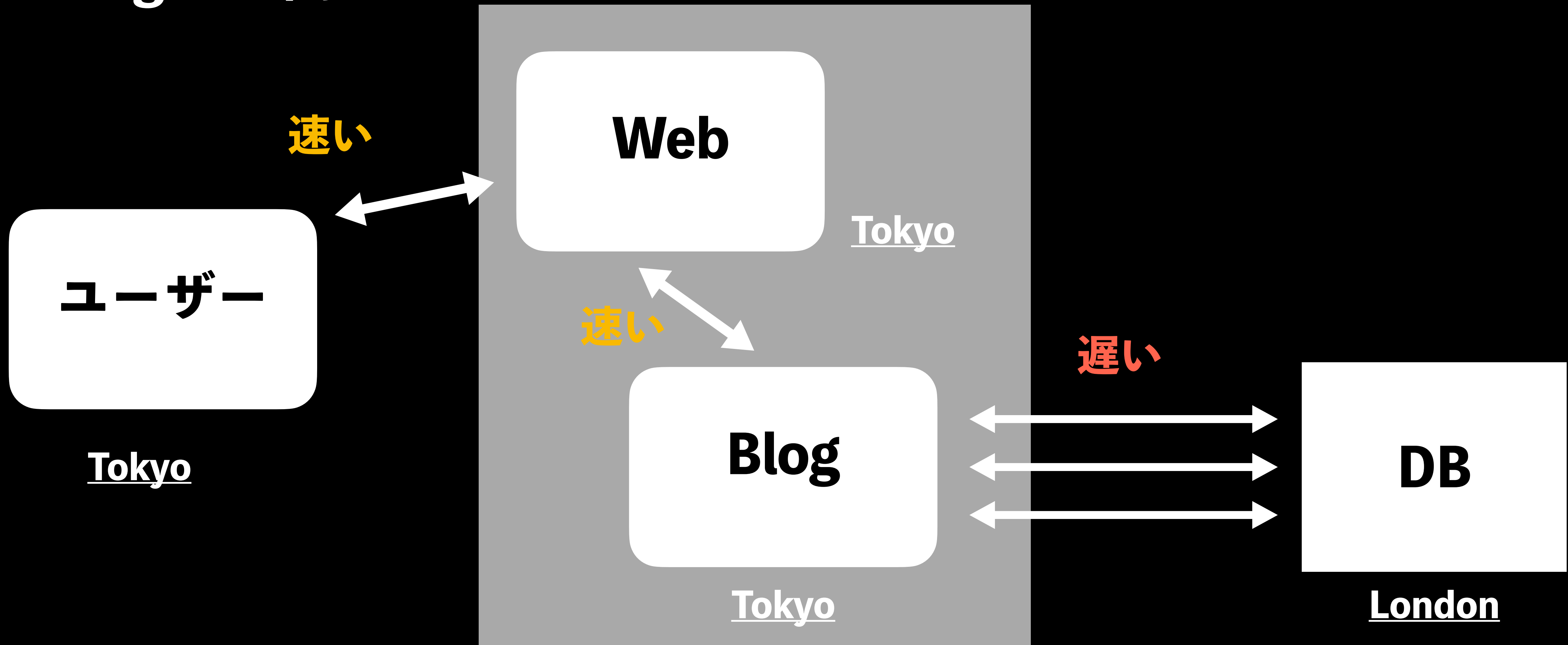
- ユーザーから遠いけど、データベースとのやりとりが多い場合は効率的



エントリーポイントごと
にロケーションが最適
化されれば嬉しい

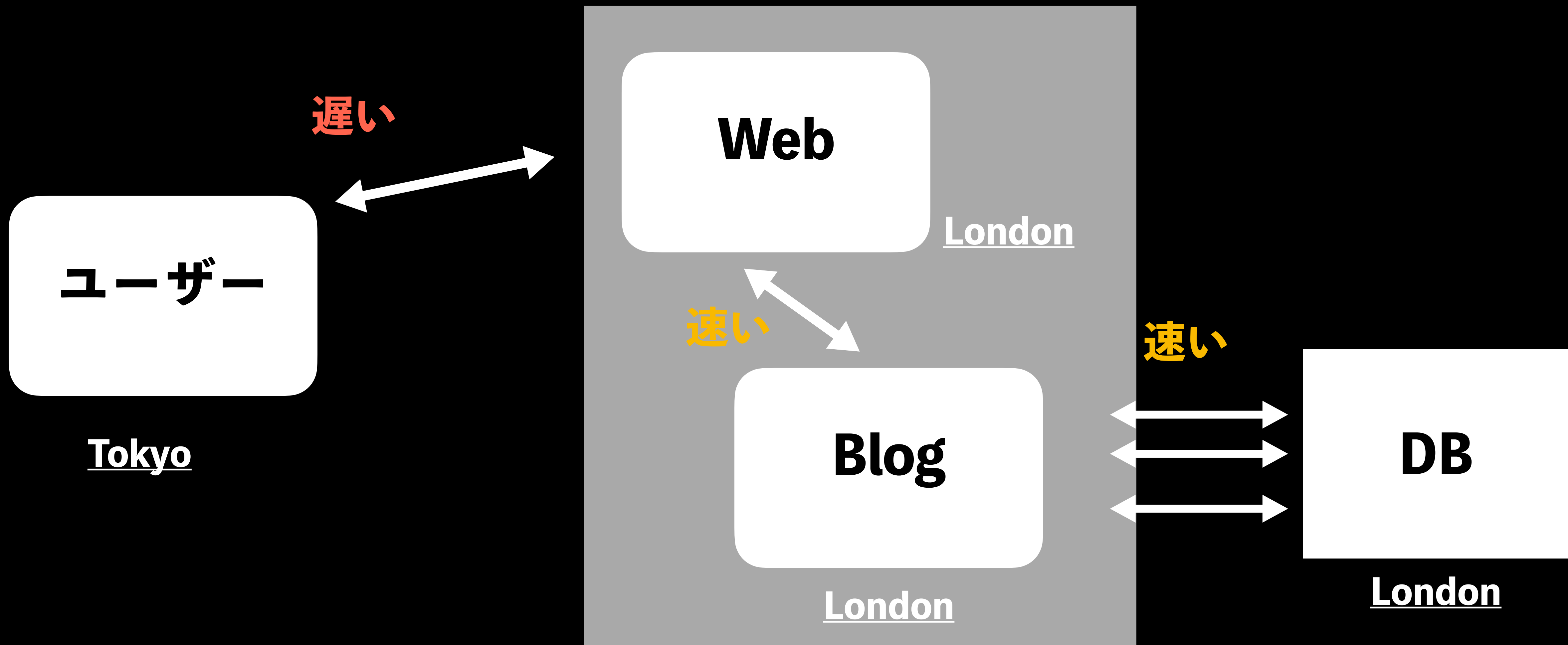
ブログサービスの場合 - Tokyo編

- Blog - DB間が遅い



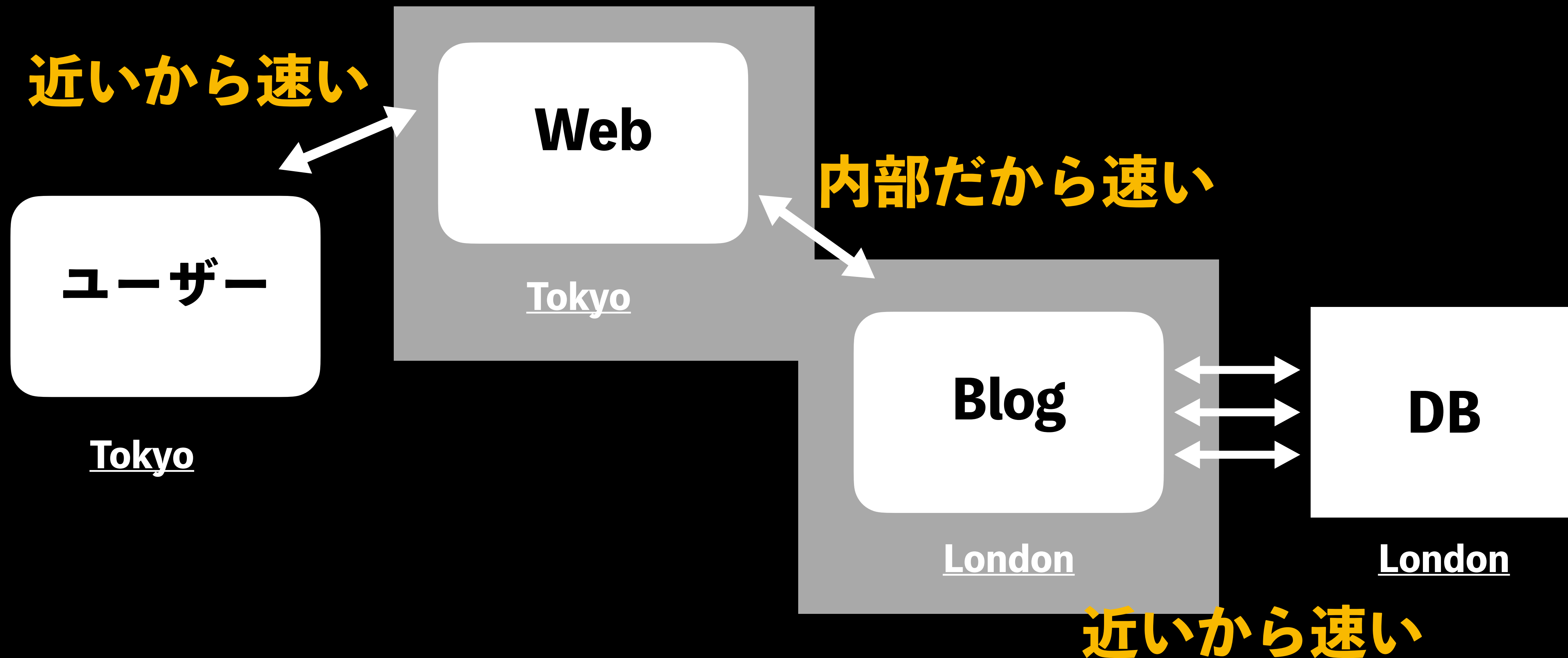
ブログサービスの場合 - London編

- ユーザー - Web間が遅い



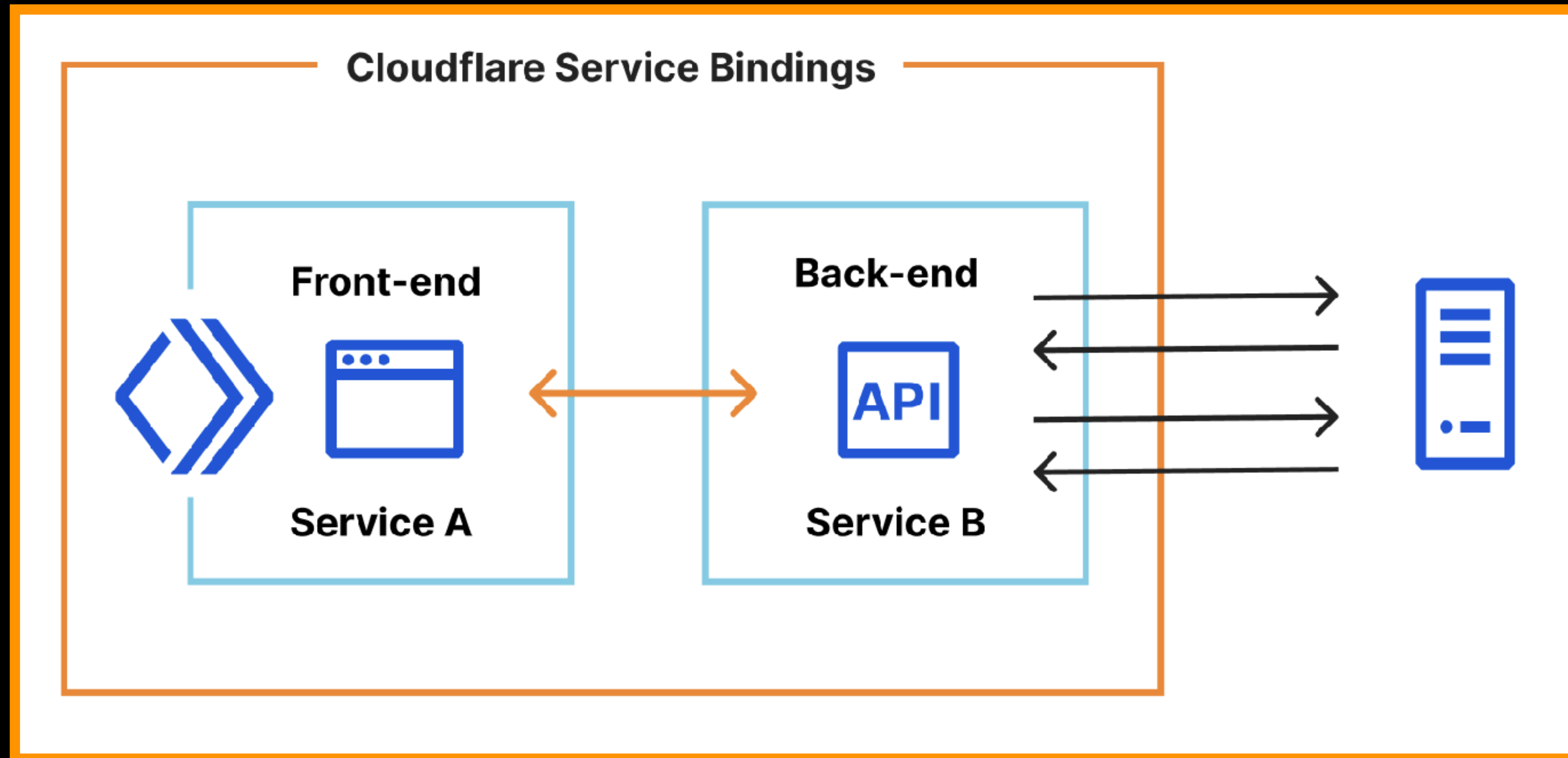
ブログサービスの場合 - 理想

- DBがユーザーから遠い場合、BlogだけDBに近いと良い



Service Bindingsの例

- もうすでに対応はしている



理想的な書き方



ts blog/src/index.ts

```
// DBがあるAPACの近くで実行される
export class BlogService extends WorkerEntrypoint {
  DB = this.env.APAC_DB

  async creatPost(post: Post) {
    DB.query('...')
    // ...
  }
}
```

APACにあるDBを取得する

Service Bindingで接続されたBLOGを呼び出す

```
// 呼び出すWorkerではそれを意識しなくていい
app.post('/posts', validator, async (c) => {
  const data = c.req.valid('json')
  c.env.BLOG.creatPost(data)
  // ...
})
```

7. マーケットプレイス？

Bindingsマーケットプレイスができるかも？

- ユーザーが作ったカスタムBindingsが流通する
- 現在は作ったアカウント内でしか使えない
- 例
 - DiscordボットBinding
 - SlackボットBinding
 - Stripe決済Binding

おさらい

JS RPCで嬉しいこと

- セキュリティへの考慮
- 開発者体験の向上
- Smart Placementと組み合わせられる
- マーケットプレイスで夢が広がる

遊んでみてくだ

さい

Thanks!!