The power of Web-standards

Deno Fest - 2023.10.20 @Tokyo Akihabara

Yusuke Wada

自己紹介

- Yusuke Wada - 和田裕介
- Developer Advocate @Cloudflare
- Creator of Hono
- Co-founder of Bokete
- Web framework developer
- https://github.com/yusukebe

今日話すこと

1. Web-standardsとHono（イントロ）
2. Web-standardsのすごいところ
3. スタンダードではないもの
4. Service Worker Magic

Web-standardsのすごさをいかした
Denoとかで動くアプリをつくれるようになる？

1. Web-standardsとHono

Web-standards

- JavaScriptの環境で共通で使われるつつあるWeb系のAPI
- もちろんDenoも！
- ブラウザのAPIをサーバーサイドでも使えるようにしようとしている
- WinterCGでディスカッションされている
  - Web-interoperable Runtimes Community Group
  - ブラウザ以外の環境
  - 「Web相互運用可能」を目指す

https://wintercg.org/ より
議論に参加してるのはこれだけではない

fetch

```
const res = await fetch('https://deno.com')
console.log(res.status) // 200
```

Request/URL/Response

```
Deno.serve((req) => {
  const url = new URL(req.url)
  return new Response(`Path is ${url.pathname}`)
})
```

その他

- Headers
- URLSearchParams
- FormData
- Blob
- ReadableStream
- TextDecoder
- atob()/btoa()
- …

最小限で親しみやすいAPI

Hono

- Web-standardsのAPIのみを使ったWebフレームワーク
- 外部ライブラリへの依存は0
- 2021年12月から開発して約2年
- 当初はCloudflare Workersで動かすためにつくられた
- その後、Fastly Compute@Edge、Deno、Bun、Node.jsの順番で対応
- その経験から、Web-standardsについて語る

Hono - The Web Framework built on Web Standards
https://hono.dev/

2. Web-standardsのすごいところ

Web-standardsのすごいところ

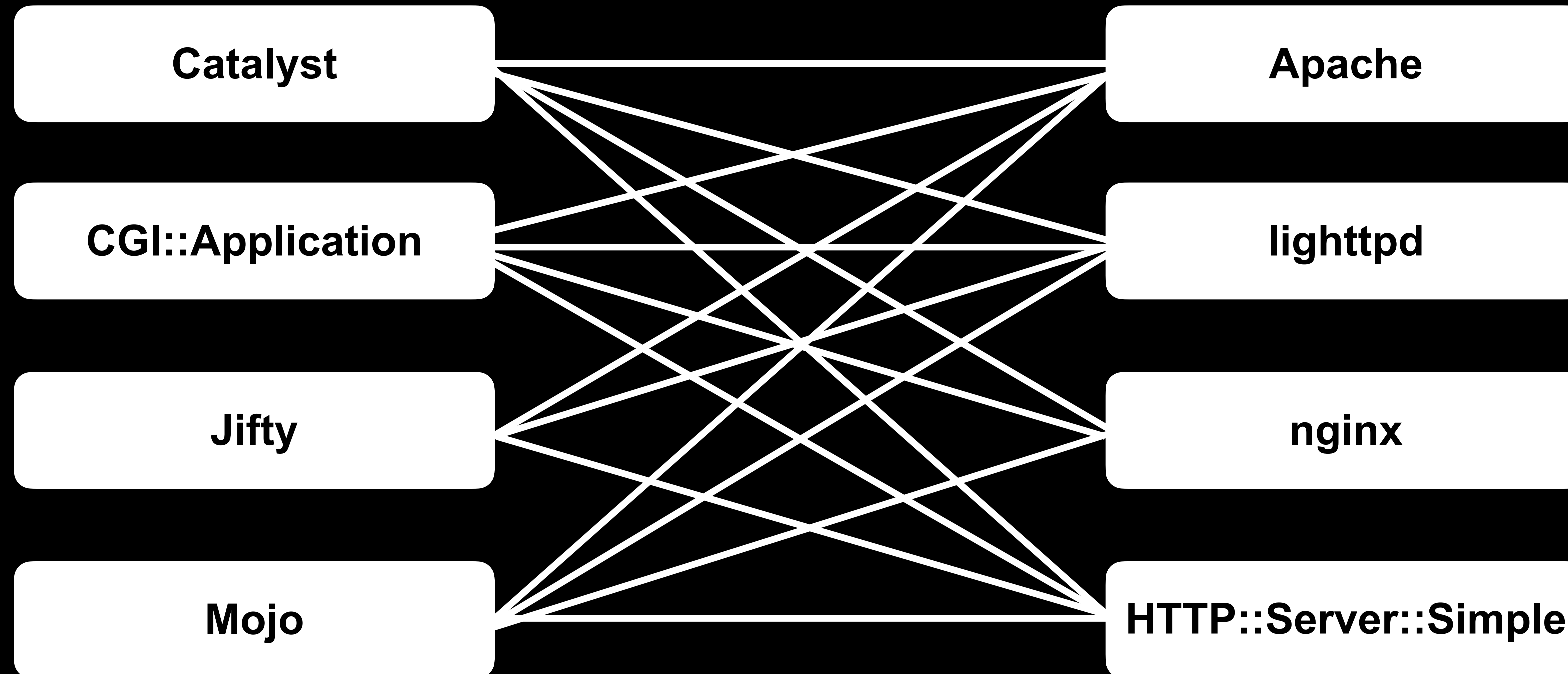
1. 他言語で苦労していたものがすでにある - 今回はとばします…
2. ほとんどのランタイムで実装されてる
3. テストがしやすい

1. 各言語で苦労していたものがすでにある

- 例: Perl - PSGI
  - 2009年のYAPC::Asiaで初めて語られる
  - フレームワークとWebサーバー間の約束事
  - PythonでいうWSGI、RubyのRack

とばします…

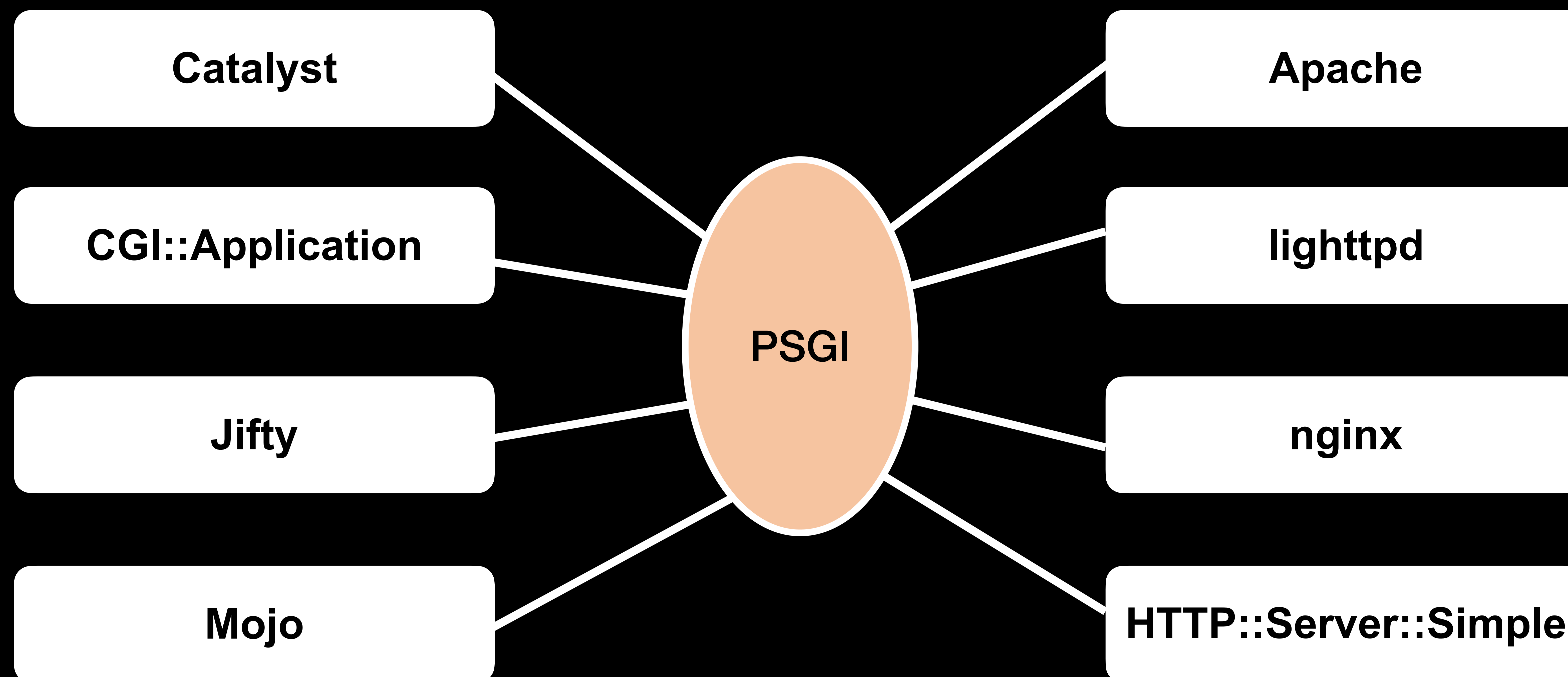
フレームワークとWebサーバ



とばします…

Perl Hackers Hub #1 -WEB+DB PRESS 2010.02.24
https://gihyo.jp/dev/serial/01/perl-hackers-hub/000101

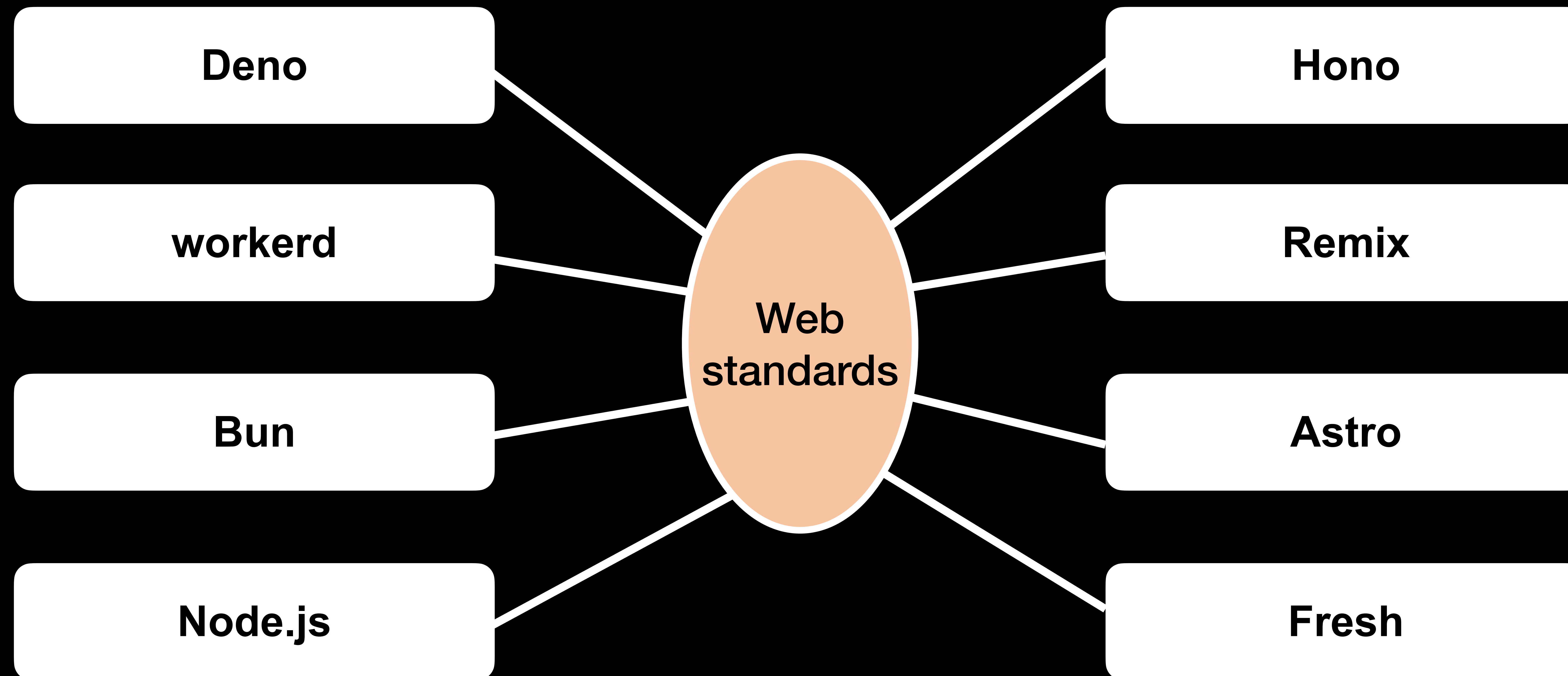
PSGIによるインターフェース共通化



とばします…

Perl Hackers Hub #1 -WEB+DB PRESS 2010.02.24
https://gihyo.jp/dev/serial/01/perl-hackers-hub/000101

Web-standards API



とばします…

Plack

- PSGIの実装のひとつ
- Webアプリ/フレームワークを作るためのキット
- シンプルなAPI
- それがもう標準で使える

とばします…

```perl
use Plack::Request;

my $app = sub {
    my $env = shift;
    my $req = Plack::Request->new($env);
    my $path = $req->path_info;

    my $res = $req->new_response(200);
    $res->body("Path is $path");

    return $res->finalize;
}
```

Plack::Requestによる実装

とばします…

```
Deno.serve((req) => {
  const url = new URL(req.url)
  return new Response(`Path is ${url.pathname}`)
})
```

ほぼ同じAPIがランタイム標準である

とばします…

いい時代になった

- 今になっては当たり前？
- 過去に各言語（例: Perl、Ruby、Python）が苦労してきたものがすでにある
- 付け加えると、その分APIが洗練されている

とばします…

2. ほとんどのランタイムで実装されている

- 本当にHonoはどこでも動く
  1. Cloudflare Workers
  2. Fastly Compute@Edge
  3. Deno
  4. Bun
  5. Lagon
  6. Node.js
  7. Verbal
  8. AWS Lambda
  9. Lambda@Edge

```
yusuke $ npm create hono@latest

create-hono version 0.3.1
✔ Target directory … my-app
? Which template do you want to use? › - Use arrow-keys. Return to submit.
    aws-lambda
    bun
    cloudflare-pages
    cloudflare-workers
❯   deno
    fastly
    lagon
    lambda-edge
    netlify
  ↓ nextjs
```

create-hono では12のテンプレートが使える

ImportとエントリポイントGが違うだけ

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

export default app
```

import

エントリポイント

Cloudflare Workers / Bun

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

export default app
```

Fastly Compute@Edge

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

app.fire()
```

Lagon

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

export const handler = app.fetch
```

Deno

```
import { Hono } from 'npm:hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

Deno.serve(app.fetch)
```

Node.js

```
import { serve } from '@hono/node-server'
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

serve(app)
```

AWS Lambda

```
import { handle } from 'hono/aws-lambda'
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

export const handler = handle(app)
```

Next.js

```
import { handle } from '@hono/node-server/vercel'
import { Hono } from 'hono'

const app = new Hono()

app.get('/hello', (c) => {
  return c.text('Hello')
})

export default handle(app)
```

どこでも動くのすごい

- 他にも
  - Lambda@Edge
  - Netlify
  - Vercel
  - などで動く

3. テストがしやすい

- 書きやすい
  - Request/Responseを書いてすべてのケースをカバーできる
  - サーバーを立ち上げなくてもe2eっぽい
- 相互運用可能

Honoのテスト

/helloにGETリクエストするRequestオブジェクト
をアプリに渡している

```
test('GET /hello is ok', async () => {
  const res = await app.request('/hello')
  expect(res.status).toBe(200)
  expect(await res.text()).toBe('hello')
})
```

Responseオブジェクト

Honoのテスト（12,000行）はすべてこんな感じ

例: ルーティング、リクエスト、レスポンスの扱い、ミドルウェアなどのテスト

相互運用可能

- HonoのテストはCloudflareの環境を使っていた
- それを外した = Node.jsへの移行
- やったことは以下だけ
    - node:cryptoをcryptoへマッピング
    - Cache APIのモック
    - MD5のテストをスキップ
        - workerdにはあるがWebCryptoスタンダードではない
    - ランタイムキーをworkerdからnodeへ変更

```
vitest.config.ts
@@ -6,7 +6,7 @@ export default defineConfig({
 6  6      globals: true,
 7  7      include: ['**/src/**/(*.)+(spec|test).+(ts|tsx|js)'],
 8  8      exclude: [...configDefaults.exclude, '**/sandbox/**'],
 9     -    environment: 'miniflare',
    9  +    setupFiles: ['./src/test-utils/setup-vitest.ts'],
10 10      coverage: {
11 11        provider: 'v8',
12 12        reporter: ['text'],
```

https://github.com/honojs/hono/pull/1558

以上、Web-standardsのすごいところでした

1. 他言語で苦労していたものがすでにある
2. ほとんどのランタイムで実装されてる
3. テストがしやすい
4. （APIが洗練されている）
5. （APIがミニマムだから覚えることが少ない）
6. （= 開発体験がよい）

3. スタンダードではないもの

スタンダードではないものを把握しなくてはいけない

- マルチランタイム対応で問題になるのは2つ
  1. 環境変数
  2. ファイルシステム

1. 環境変数

- いわゆるシステムの環境変数だけではない（例: Cloudflare Workers）
- Cloudflare: `c.env.TOKEN`
- Deno: `Deno.env`
- Bun: `process.env` or `Bun.env`
- Node.js `process.env`
- Fastly — そもそもユーザーが値をセットするものではない
  - `FASTLY_HOSTNAME`, `FASTLY_POP`, `FASTLY_REGION`…
  - 以前、Edge DictionaryをつかってTOKENを扱ったことがある
  - `ConfigStore`というのがあるらしい

Adapterヘルパーの `env()`

- 各ランタイム依存を吸収して、透過的に環境変数にアクセスできる

```
import { env } from 'hono/adapter'

app.get('/auth', (c) => {
  const { TOKEN } = env<{ TOKEN: string }>(c)
  // ...
})
```

`env()` の実装

- 「ランタイムキー」をキーに値を取得する

```
const runtimeEnvHandlers: Record<string, () => T> = {
  bun: () => globalEnv,
  node: () => globalEnv,
  'edge-light': () => globalEnv,
  lagon: () => globalEnv,
  deno: () => {
    return Deno.env.toObject() as T
  },
  workerd: () => c.env,
  fastly: () => ({} as T),
  other: () => ({} as T)
}
```

ランタイムキー

- WinterCGが「Runtime Keys Specification」をつくっている
- https://runtime-keys.proposal.wintercg.org/
  - node
  - deno
  - bun
  - workerd - Cloudflare Workers
  - fastly
  - edge-light - Vercel Edge Functions
  - lagon
  - …

`getRuntimeKey()`

```
app.get('/', (c) => {
  if (getRuntimeKey() === 'workerd') {
    return c.text('You are on Cloudflare')
  } else if (getRuntimeKey() === 'bun') {
    return c.text('You are on Bun')
  }
  ...
})
```

`getRuntimeKey()` の実装

```
export const getRuntimeKey = () => {
  const global = globalThis as any
  if (global?.Deno !== undefined) return 'deno'
  if (global?.Bun !== undefined) return 'bun'
  if (typeof global?.WebSocketPair === 'function') return 'workerd'
  if (typeof global?.EdgeRuntime === 'string') return 'edge-light'
  if (global?.fastly !== undefined) return 'fastly'
  if (global?.__lagon__ !== undefined) return 'lagon'
  if (global?.process?.release?.name === 'node') return 'node'
}
```

2. ファイルシステム

- 各ランタイムごとにファイルシステムへのアクセス方法が違う
  - Deno - `Deno.open()`
  - Bun - `Bun.file()`
- そもそもファイルシステムがない…
  - Cloudflare/Vercel/AWS…
- ファイルシステムじゃないストレージ
  - Cloudflare KV / R2
  - Vercel Blob
  - Amazon S3

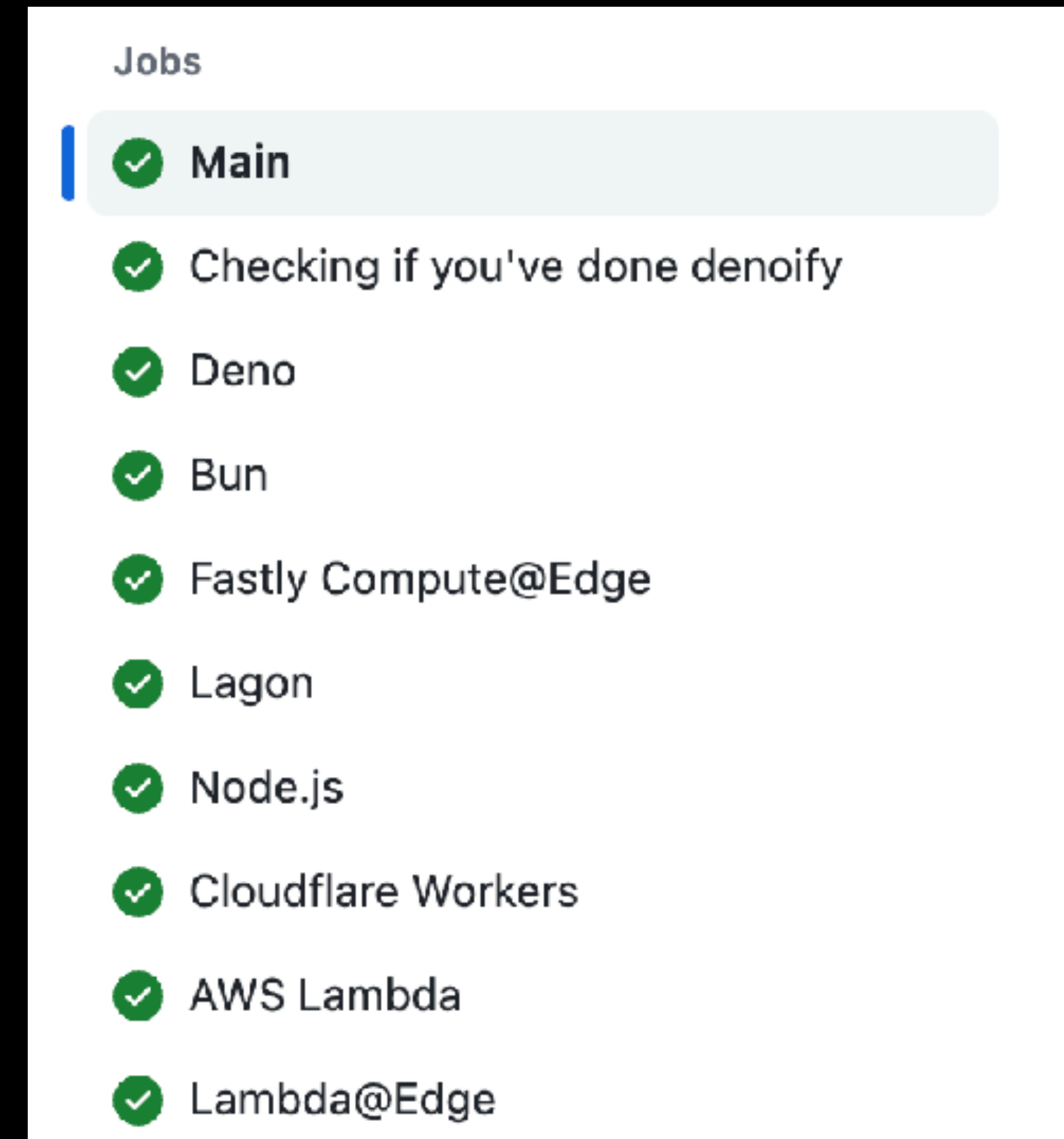
各ランタイムごとに`serveStatic()`を実装する

- ランタイム/プラットフォームごとのAdapterで`serveStatic()`を実装
  - Deno
  - Bun
  - Node.js
  - Cloudflare Workers

```
import { Hono } from 'https://deno.land/x/hono@v3.8.1/mod.ts'
import { serveStatic } from 'https://deno.land/x/hono@v3.8.1/middleware.ts'

const app = new Hono()

app.use('/static/*', serveStatic({ root: './' }))

Deno.serve(app.fetch)
```

テストをする

- ランタイムで差がでるところだけCIでテスト
  - Basic認証 - Crypto系（最近は平気）
  - `env()`
  - `getRuntimeKey()`
  - `serveStatic()`



HonoのCI
Mainの他に8つのランタイムのテストが走る

スタンダードなはずだけど同じじゃないもの

- スタンダードっぽいけどミニマムコモンには入ってないもの
  - https://common-min-api.proposal.wintercg.org/
  - Cache API - Cloudflare、Denoのみ
  - URLPattern - Bunでは実装しない
- 実装がミスってる
  - Bun - Requestオブジェクトをclone()してjson()すると起きるバグ
- 独自実装
  - Cloudflare - Requestインスタンスのcf
  - Reactのfetch？

とはいえ意識すればOK

- 環境変数とファイルシステム
- 「メイン」はしっかりテスト
- ランタイム依存はそこだけCIで保証する

4. Service Worker Magic

Honoはブラウザでも動く

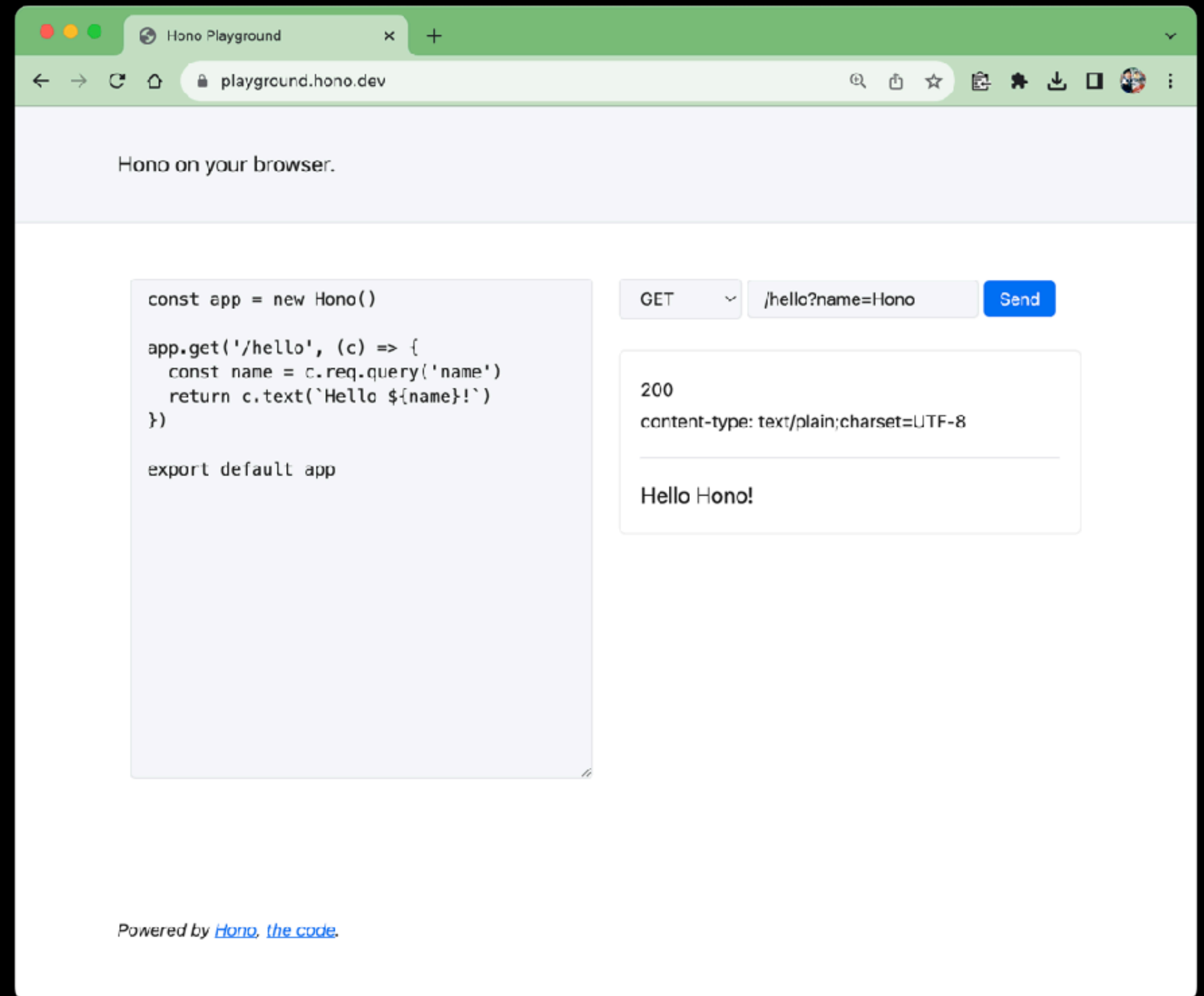
```
1 <html>
2   <head>
3     <script type="module">
4       import { Hono } from 'https://esm.sh/hono@3.8.1'
5       const app = new Hono()
6       app.get('*', (c) => c.text(`You are accessing ${c.req.path}`))
7       const res = await app.request(location.href)
8       document.getElementById('result').innerText = await res.text()
9     </script>
10   </head>
11   <body>
12     <div id="result"></div>
13   </body>
14 </html>
```

当たり前だけど面白い！

14行のHTMLの中でHonoを動かせる！

Playground

- 昨日作った
- スクリプト埋め込みHTMLだけ
- 83行でそれなりに動く
- https://playground.hono.dev/



そして、HonoはService Workerでも動く

- Service Worker = ブラウザ内で動くワーカー、オフラインキャッシュなど
- それを利用した「マジック」
- サーバーが自分自身と同じプログラムを配信して、それをブラウザがロードして、どちらでも同じコードが実行され、サーバーだけではなくブラウザからもレスポンスを返す

1. サーバーはCloudflare Workers、ブラウザはService Workerのプログラムを指す
2. サーバーのプログラムはsw.js、ブラウザで動くプログラムもsw.js
3. 全く同じ内容かつ同じリソースを参照して、同じように動く
4. サーバーsw.jsが自分自身のコードsw.jsを/sw.jsというパスで配信する
5. /にアクセスするとsw.jsがService Workerとして登録される
6. /sw/*をService Workerのスコープにする
7. /server/helloにアクセスするとサーバーからレスポンスが返る
8. /sw/helloにアクセスするとService Workerがインターセプトして、ブラウザからレスポンスが返る

デモ

localhost:8787

DevTools - localhost:8787/

Elements Console Network Performance Memory

Filter All levels 1 Issue: 1

Service Worker Magic

- From Server
- From Service Worker

This is /
Hello! from Server!

Registering Service Worker...

Server and Browser(Service Worker) code are same!

```
1:43:58 GET /hono.js 200
1:43:58 GET /hono.serve-static.js 200
1:43:58 GET /hono.logger.js 200
1:44:18 GET / 200
1:44:18 GET /sw.js 200
1:44:18 GET /hono.js 200
1:44:18 GET /hono.serve-static.js 200
1:44:18 GET /hono.logger.js 200

B to open a browser, D to open Devtools, S to turn on (experimental) sharing, L to turn on local mode, X to exit

[2] 0:node* 1:zsh 2:gh-                    [2022-02-11 01:45]
```

https://user-images.githubusercontent.com/10682/153455595-77fea6e5-93d7-4698-8d75-85896edd995b.gif

以上

1. Web-standardsとHono
2. Web-standardsのすごいところ
3. スタンダードではないもの
4. Service Worker Magic

まとめ

- Web-standardsのすごさ
  - 相互運用可能、テストがしやすい、DXがよい
- スタンダードじゃないもの
  - 環境変数、ファイルシステム、その他

Web-standardsを理解しておくと
そのすごさを活かして
Denoやその他のランタイムで動くアプリケーションを相互運用可能に作れる！