

モダンなJavaScript/TypeScript実行環境「Deno」

Node.js 7つの後悔

- 後悔1: **Promise** を使わなかった
- 後悔2: **Security Sandbox** を活用しなかった
- 後悔3: **GYP** を使い続けてしまった
- 後悔4: **package.json**
- 後悔5: **node_modules**
- 後悔6: モジュール解決時の拡張子省略
- 後悔7: **index.js**

Devsumi 2022 Deno - 17 / 97

Deno とは
"改良版" Node.js

Devsumi 2022 Deno - 21 / 97



Devsumi 2022 Deno - 22 / 97

Deno の特徴

- ブラウザ互換性
- TypeScript サポート
- サンドボックスセキュリティ
- ビルトイン開発ツール

ブラウザ互換性

- Deno には可能な限りブラウザ互換 API を取り入れるというデザイン方針がある
- Node の開発が始まった時に比べてかなり多くのブラウザ API が定義されていて、いろいろな事がブラウザと同じ API で出来るようになっている
- ただし、Node.js も可能な場合は後からブラウザ互換 API を取り入れるという流れがあり、ややこしい状態になっている
 - 例. `url` と `URL`、`Buffer extends Uint8Array`

Devsumi 2022 Deno - 27 / 97

Deno に実装されている ブラウザ互換API の例

fetch API

```
const resp = await fetch("https://example.com");
const html = await resp.text();
console.log(html);
```

- 簡単に HTTP リクエストが出来る
- http client library などが不要

Deno に実装されている ブラウザ互換API の例

バイナリ処理 - TypedArray API (Uint8Array, etc)

```
const data = Uint8Array.from([0x66, 0x6f, 0x6f]);
const text = new TextDecoder().decode(data);
// => foo
```

- Node の場合は Buffer (独自クラス)
- ただし今は Node は TypedArray も持っている

Deno に実装されている ブラウザ互換API の例

URL パーサー

```
const url = new URL("https://example.com/?foo=bar");
console.log(url.hostname); // => example.com
console.log(url.searchParams.get("foo")); // => bar
```

- Node の場合は require("url") が昔からあるが、後に URL も実装されて両方ある状態

Deno に実装されている ブラウザ互換API の例

Web Storage

```
localStorage.setItem("key", data);
...
// プログラム再起動後
console.log(localStorage.getItem("key"));
// => さっき保存したデータが残っている
```

- エントリポイント毎に独立したストレージを持てる
- 内部では **SQLite** を使って保存している
- **Node** には無い機能

Deno に実装されている ブラウザ互換API の例

HTTP imports

```
import { serve }
  from "https://deno.land/std@0.126.0/http/server.ts";
serve((_req) => new Response("Hello, world"));
```

- **URL** 指定でモジュールを取得できる。
- この機能があるため **Deno** では **package.json** や **node_modules** が不要になっている
- **Node** でも最近実験的な実装が始まったが物議を醸している

Deno に実装されているブラウザAPI の例

- **PubSub - EventTarget API**
- **ストリーミング処理 - Web Stream API**
- **暗号 - Web Crypto API**
- **GPU - WebGPU**
- **http server - Request, Response API**

参考: [A list of every web API in Deno](#)

ブラウザ互換 API の良いところ

- ブラウザと共通して使えるコードを書ける
- ブラウザ API はとてもきちんと定義されている
 - 議論の質が高い ↗
 - 仕様書の質が高い
 - 自動テストがある
- 仕様策定プロセスがあるため、簡単に変わることはない

=> 安心して使える

GoogleやAppleの人が議論するほど

ブラウザ互換性 - 最近の進捗 - WPT

- 2021年1月 Web Platform テストを CI に導入
- Web Platform Test = ブラウザが共通で通している Web API のテストスイート
- コミット毎に Web 互換性をチェックしています

Path	Chrome 97 Canary #64 Feb 26, 2021	Edge 97 Microsoft 97.0.10563.62 Feb 26, 2021	Firefox 97 Canary #64 Feb 26, 2021	Safari 15.0 (pre-release) 15A5201.1018 Feb 26, 2021	Opera 77 Linux (pre-release) 77.0.4084.2024
BackgroundSync/	30/30	30/30	30/30	30/30	0/0
Clipboard/	1076/1082	1076/1082	1076/1082	1081/1082	1047/1108
IndexedDB/	2002/2008	2002/2008	2002/2008	2002/2002	0/0
WebComponents/	40307/40309	40307/40309	40307/40309	40307/40309	36930/40274
WebCodecs/	104/107	101/107	101/107	101/107	0/0
WebGL2/	10/10	0/0	10/10	10/10	0/0
WebGPU/	102/102	102/102	102/102	102/102	0/0
WebAuthn/	40/37	36/37	36/37	41/37	0/0
WebAssembly/	101/100	101/100	101/100	101/100	0/0
WebP/	0/0	0/0	0/0	0/0	0/0
WebVTT/	400/400	400/400	400/400	400/400	397/397
WebXR/	20/37	20/37	20/37	20/37	0/0
WebGPU/	204/208	204/208	204/208	20/20	0/0
WebGL/	61/70	61/70	61/70	61/70	0/0
WebGPU/	61/68	61/68	61/68	61/68	0/0
WebGPU/	100/100	100/100	100/100	100/100	0/0

Devsumi 2022 Deno - 43 / 97

Web 互換性 - 最近の進捗 - MDN

- 2021年8月 MDN への掲載が始まる

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	Deno
<code>TextEncoderStream</code>	71	79	No	No	56	14.1	71	71	No	90	14.5	10.0	1.11
<code>TextEncoderStream.prototype.constructor</code>	71	79	No	No	56	14.1	71	71	No	90	14.5	10.0	1.11
<code>encoding</code>	71	79	No	No	56	14.1	71	71	No	90	14.5	10.0	1.11
<code>readable</code>	71	79	No	No	56	14.1	71	71	No	90	14.5	10.0	1.11
<code>writable</code>	71	79	No	No	56	14.1	71	71	No	90	14.5	10.0	1.11

TypeScript サポート

- TypeScript をそのまま実行できる

```
// sample.ts
const res = await fetch("https://example.com")
console.log(res.body.text);
```

```
$ deno run sample.ts
Check file:///Users/kt3k/sample.ts
error: TS2531 [ERROR]: Object is possibly 'null'.
console.log(res.body.text);
~~~~~
```

↑ 実行時エラーではなく型エラー

デフォルトで型チェックが可能

TypeScript サポート deno lsp

- ネットワーク越しの TypeScript も型補完が可能

```
1 import { serve } from "https://deno.land/std@0.115.1/http/server.ts";
2
3
4 serve();
```

serve(handler: Handler, options?: ServeInit): Promise<void>

The handler for individual HTTP requests.
Serves HTTP requests with the given handler.
You can specify the `addr` option, which is the address to listen on, in the form "host:port". The default is "0.0.0.0:8000".
The below example serves with the port 8000.

```
import { serve } from
"https://deno.land/std@STD_VERSION/http/server.ts";
const addr = "http://localhost:8000";
```

TypeScript サポート 補足

- なお、ベストプラクティスと考えられている設定がデフォルトで入っているので、設定無しで TypeScript を使い始められます。
- デフォルトから外れたい場合は、自分でコンパイラオプションを書くこともできます。

サンドボックスセキュリティ

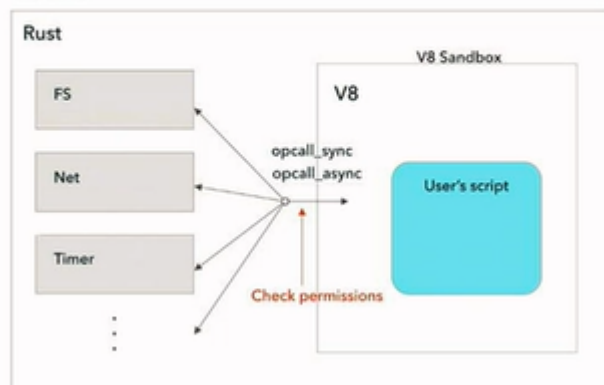
前提の話

- Deno は内部的に V8 エンジンを使っている。
- V8 は「信用できないコード」を動かす事を前提に設計されているため、サンドボックス化されている
- => V8 の外に影響を及ぼせないようになっている

サンドボックスセキュリティ: ユーザ作成の信用できないコードを動かす前提でつくられている

サンドボックスセキュリティ

Deno



Devsumi 2022 Deno - 54 / 97

サンドボックスセキュリティ

ファイルの読み取りを許可する場合 (全部許可)

```
deno run --allow-read program.ts
```

カレントディレクトリのみ読み込み許可

```
deno run --allow-read=. program.ts
```

サンドボックスセキュリティ

ファイルの書き込みを許可する場合 (全部許可)

```
deno run --allow-write program.ts
```

dist/ ディレクトリのみ書き込み許可

```
deno run --allow-write=dist/ program.ts
```


サンドボックスセキュリティ

ネットワークアクセスを許可する場合 (全部許可)

```
deno run --allow-net program.ts
```

特定のドメイン・ポートのみネットワークアクセス許可

```
deno run --allow-net=example.com:80 program.ts
```

=> 意図しない攻撃コード混入時に被害を防ぐ事が出来る

サンドボックスセキュリティ

その他のパーミッション

- `--allow-env` 環境変数の使用許可
- `--allow-run` 別プロセス実行の使用許可
- `--allow-ffi` 外部ネイティブ拡張使用許可
(Deno のセキュリティモデルを無視したコードが実行されるため、使用注意)
- `--allow-hrtime` 高精度タイマー使用許可・スベクター対策 (基本許可しない)
- `--allow-all, -A` 全部許可、開発時などに利用

サンドボックスセキュリティ余談

- ところで、npm では恒常的にセキュリティインシデントが起きている
- その大部分は、Deno の場合はセキュリティフラグを正しく使う事で回避出来る
- Node.js に今からこの機能を入れる事は現実的ではない
- Deno が Node.js を本質的に"改善"している機能と言える

開発する前にやるが多すぎる

最近の Node.js 開発の始め方

- TypeScript のインストール
- ESLint のインストール
- Prettier のインストール
- Jest のインストール
- テストカバレッジツールのインストール 😞
- バンドラーのインストール 😞

=> 最初からインストールしないといけないものが多い!

=> しかも時間が経つと「それはもう古い」になりがち!

Denoなら開発を始めるコストが低い

Deno のビルトイン開発ツール

- 例

- コードのフォーマット => `deno fmt`
- コードのリント => `deno lint`
- ユニットテストの実行 => `deno test`
- テストカバレッジ => `deno coverage`
- スクリプトのバンドル => `deno bundle`
- TypeScript => 本体に内包

=> Deno 本体さえあれば、開発に必要なツールが一通り揃っている!

Deno の採用例 - GitHub

次世代 Data Access API



Flat Data

Flat explores how to make it easy to work with data in git and GitHub. It builds on the "git screens" approach pioneered by Simon Willison to offer a simple pattern for bringing working datasets into your repositories and versioning them. Because developing against local datasets is faster and easier than working with data over the wire.

What is it used for

Bring working sets of data to your repositories

stars

👤 📄 📄

Who made it

👤 Stan Gost

👤 Amelia Waterberger

👤 Matt Rotherberg

👤 Yone Abarado

Deno の採用例 - Slack



Slack Introduces New Platform With Help From Deno

November 16, 2021 by
Aaron O'Mullan, Ryan Dahl

Today Slack has announced their next generation development platform. Slack chose Deno for its "secure by default" principles, its web standard APIs, and its first-class TypeScript support. Read more about it at <https://api.slack.com/future>.

Deno の採用例 - Slack

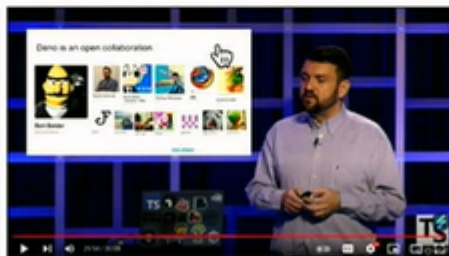
- Slack の新しい SDK は Deno ベース
- この件以降、各種スタートアップから Deno 社への問い合わせがかなり増えている

まとめ

- Deno は "改良版" Node.js を目指すプロジェクト
- Deno は Web 互換性、セキュリティ、TypeScript サポート、ビルトイン開発ツールなどが特徴的でそれらの機能はかなり充実・安定してきている
- Deno は GitHub、Slack などから採用が始まっている

入社するまでのコントリビューション 2018 - 2020

- 純粋に技術的に面白そうという理由で開発に参加
- この時点では会社は存在していなかった
- 外部コントリビュータとしては 3 番目ぐらいに contribute していた



2020年中盤 Deno Land Inc. 設立

- US の登記情報のようなサイトが検索で引っかかる
- Node.js contributor の一人が Deno Land Inc. に入ると issue 上でコメント
- ただし公式アナウンスは無し

個人にタスクをアサインすることはない

入社後

- まず、個人にタスクをアサインする事はないと告げられる。
- 各人が Deno にとって良いと思った事をする。
- と言っても、自分が得意な分野には限りがあるため、バリューが出せそうな領域を見つけて取り組む
- 取り組むタスク探しはいつも難しい問題

Deno での1年の振り返り

- 初めての英語環境での就業
- 初めての仕事としての OSS 開発
- 世界レベルで有名な人との協業

=> ものすごく挑戦があり、ほぼ良いことづくめの環境

=> ただし周りが非常に優秀なためプレッシャーもある

JavaScriptは.jsもTypeScriptは.tsとすればよい

Deno2.0からNode.js完全上位互換になる予定

Denoを学ぶには？

<https://examples.deno.land/>

Google Cloud Japanメンバーに聞いてみよう！ITエンジニアの人材育成と知識共有 ～ Googleのスケラブルなアプローチ

Google Cloud

Google に入社して感じたこと

- エンジニア同士の情報共有がものすごく積極的に行われている！
 - 社内教育コース、定期的な Tech-Talk
 - 社内ポータルや Google ドライブ での情報検索
 - 個人的に連絡すると、Google Meet で親切に教えてくれる



書籍「Software Engineering at Google」は気になる

Google Cloud



O'REILLY
Software Engineering at Google
Lessons Learned from Programming Our Site
Curated by Titus Winters, Tom Harebeck & Hyungho Wang

- I. Thesis
 - 1. What Is Software Engineering?
- II. Culture
 - 2. How to Work Well on Teams
 - 3. Knowledge Sharing
 - 4. Engineering for Equity
 - 5. How to Lead a Team
 - 6. Leading at Scale
 - 7. Measuring Engineering Productivity
- III. Processes
 - 8. Style Guides and Rules
 - 9. Code Review
 - 10. Documentation
 - 11. Testing Overview
 - 12. Unit Testing
 - 13. Test Doubles
 - 14. Larger Testing
 - 15. Deprecation
- IV. Tools
 - 16. Version Control and Branch Management
 - 17. Code Search
 - 18. Build Systems and Build Philosophy
 - 19. Critique: Google's Code Review Tool
 - 20. Static Analysis
 - 21. Dependency Management
 - 22. Large-Scale Changes
 - 23. Continuous Integration
 - 24. Continuous Delivery
 - 25. Compute as a Service
- V. Conclusion

<https://www.oreilly.com/library/view/software-engineering-at/9781492082781/>

5

Programming vs Software Engineering

- **Programming**
 - 「コードを書く」という開発者個人の活動
 - 開発者個人の能力・生産性が重要
- **Software Engineering**
 - ソフトウェアの開発・変更・メンテナンスを組織的に長く継続すること
 - チームとしての生産性を高める「ポリシー」が重要
 - システムのサイズが 10 倍になった時、エンジニアの労力が 10 倍になったとすれば、そのポリシーは間違っている

6

現実的ではないイメージ

The Genius Myth (伝説の天才エンジニア)

- 「天才エンジニア」が生まれる過程
 - 新しいアイデアを思いつく
 - 数ヶ月間、洞窟に隠れて、完成品を作り上げる
 - 完成品を世の中に発表して、世界を驚かせる
 - 富と名声を手に入れる

実際

The Genius Myth (伝説の天才エンジニア)

- ~~「天才エンジニア」が生まれる過程~~
 - ~~新しいアイデアを思いつく~~
 - ~~数ヶ月間、洞窟に隠れて、完成品を作り上げる~~
 - ~~完成品を世の中に発表して、世界を驚かせる~~
 - ~~富と名声を手に入れる~~
- 最初期のプロトタイプは、個人のアイデアから生まれたとしても、広く使われるサービスとして成功に導くには、エンジニアリング組織の力が必要
 - アイデアや知識はチームの中で早く・広く共有する方がよい
 - Fail early, Fail fast, Fail often

8

心理的安全性について認識しているあたり、Googleという組織がいかに理知的であるかがよくわかる気がする



Knowledge Sharing (知識共有)のアンチパターン

- 情報のサイロ化
 - 分野ごとの専門家はいても、全体を理解する人がいない
 - 「完全に理解した」か「何も知らない」かの2極化
- コピペ文化 / 技術の墓場
 - 中身を理解しないまま、既存のやり方を踏襲
 - 「やぶ蛇」を恐れて、やり方を変えようとしていない
- 心理的安全性の欠如
 - 「知らない事」=「劣っている事」と見なす組織文化
 - 「質問をためらう気持ち」は、個人ではなく、本当は、組織の問題

9



情報共有における心理的安全性の確保

- メンター制度
 - Noogler (New Googler) には、マネージャーやチームリーダー以外の (直接の利害関係が無い) メンターをアサイン
- メールリストや社内掲示板での QA 対応
 - 大袈裟な反応の禁止: 「え! そんな事も知らないの!」
- オフィスアワー
 - 特定分野のエキスパートが待機して、対面で自由に質問できる時間を作る

「データドリブンな判断」と心理的安全性

- データに基づいて客観的に判断することが大切
 - ただし、判断に必要なデータがすべて手に入るとは限らない
 - 現状で手に入るデータで、まずは、判断することが大切
- プロジェクトを進める中で、新しいデータが手に入る
 - 途中で「失敗」に気づく事もある
- 失敗によって得られたデータから、再度、判断をやり直す
 - 「失敗」=「新しいデータを得るためのプロセス」
 - Fail fast and Iterate (早く失敗して、改善を繰り返す)

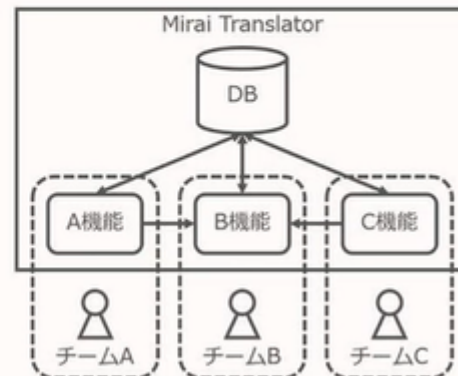
異なるAPIがひとつのDBを参照

技術負債を抱えたモノリス



各機能が密結合

- 同じDBの同じテーブルを参照
- 同期的に呼び出している箇所も存在

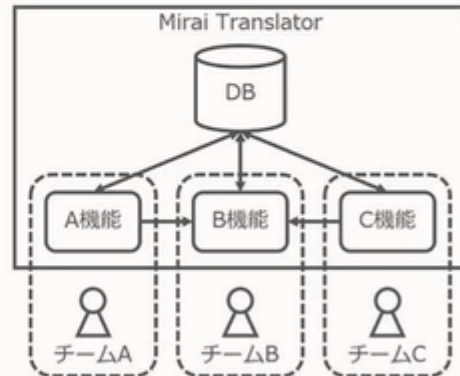


各機能が密結合

- 同じDBの同じテーブルを参照
- 同期的に呼び出している箇所も存在

デプロイ単位がシステム全体

- “A機能だけリリース”ができない
- 複数案件をまとめてリリースしがち
- リードタイムが一番遅い案件に引っ張られる



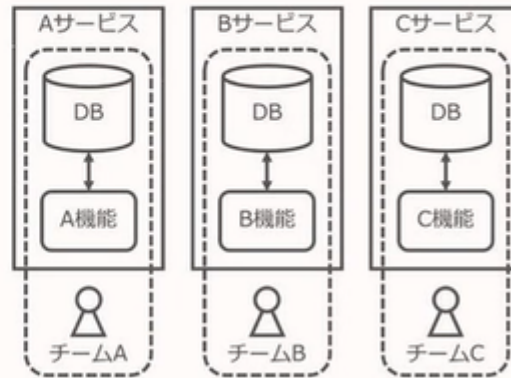
- 各機能が疎結合
- あるチームのリリース（デプロイ）を独立して実施できる

→その為に**マイクロサービス**を目指す

複数のサービスでシステムを構成する

利点

- サービス単位にデプロイできる
- サービス間の依存が発生しにくい



マイクロサービスを目指す理由

理由①：サービス単位のデプロイを可能にしたい

- サービス間で競合開発が発生しない状態を目指す

理由②：モノリスのリファクタリングによる解決は困難

- 不適切なデータモデリングに起因する複雑さが存在
- テーブル定義の改修は難易度が高い

理由①：サービス単位のデプロイを可能にしたい

- ・サービス間で競合開発が発生しない状態を目指す

理由②：モノリスのリファクタリングによる解決は困難

- ・不適切なデータモデリングに起因する複雑さが存在
- ・テーブル定義の改修は難易度が高い

理由③：トランザクションの必要箇所は少ない (ハズ)

- ・現状の仕様を鑑みるに、結果整合性でもよい箇所が多い (もちろん今後ビジネスサイドとの合意は必要)

結果整合性

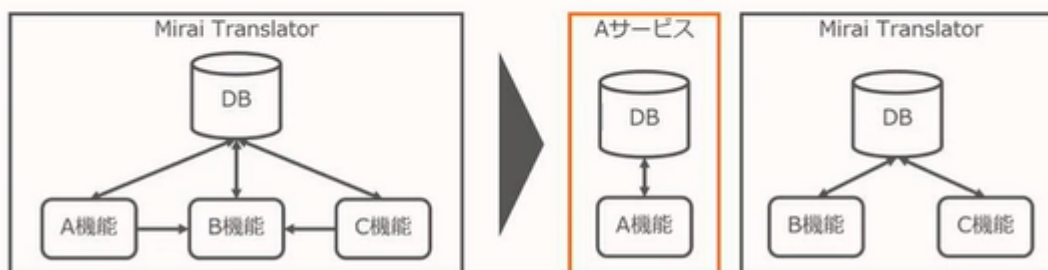
<https://ja.wikipedia.org/wiki/%E7%B5%90%E6%9E%9C%E6%95%B4%E5%90%88%E6%80%A7#:~:text=%E7%B5%90%E6%9E%9C%E6%95%B4%E5%90%88%E6%80%A7%EF%BC%88%E8%8B%B1%3A%20Eventual,%E4%BF%9D%E8%A8%BC%E3%81%99%E3%82%8B%E3%82%82%E3%81%AE%E3%81%A7%E3%81%82%E3%82%8B>

ストラングラーパターン(*)

一部分だけ置き換える～を繰り返す

➡採用

- ・小さい単位で置き換えられるためリスクが低い
- ・新規機能開発も止めなくてよい

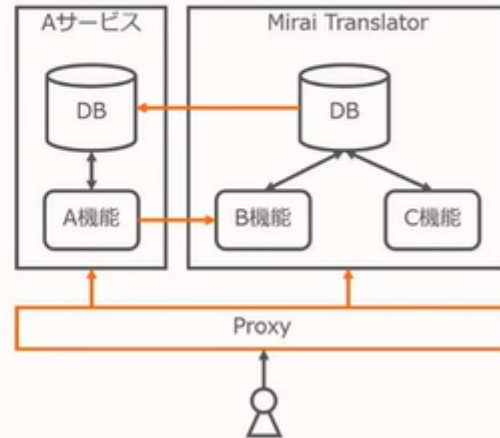


マイクロサービス化では通信は非同期性が大切。でないと、結局密結合になる

モノリスでは不要だった技術基盤が必要

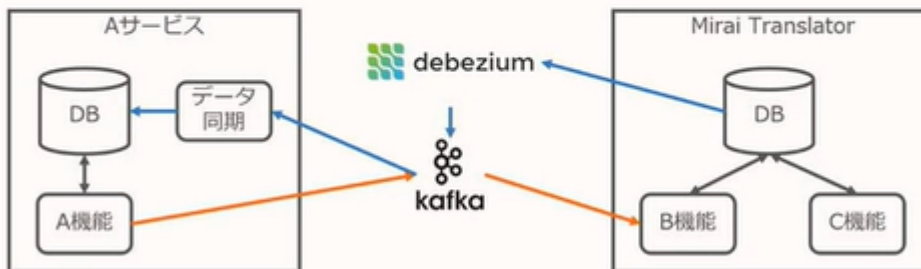
- サービス間の非同期なメッセージング基盤
- モノリスからサービスへのデータ同期基盤
- ユーザーリクエストをサービスとモノリスに振り分ける基盤

...etc



今作っている技術基盤

- Kafkaを用いた非同期メッセージング基盤
- Debeziumを用いたデータ同期基盤





- OSSの分散メッセージングプラットフォーム
 - “at least once” かつ メッセージ順序保証可
 - メッセージのスキーマは任意
 - 複数Publisher(Producer)による利用可能
 - 複数Subscriber(Consumer)による利用可能
 - メッセージを内部で永続化可能

GCPでいうところのCloud PubSub

マイクロサービス化には非同期メッセージングが重要

Debezium

DebeziumはDBに対するデータ操作をキャプチャしてイベントストリームに変換してくれる分散プラットフォーム。

<https://debezium.io/>

<https://rheb.hatenablog.com/entry/debezium-intro>

ユーザの声: debezium便利です。特に既存のシステムをあまりいじらなくて済むのありがたいです

今のところの成果



最小構成のデータ同期基盤を組み、負荷試験や復旧試験を実施

分かったこと

- メッセージの重複欠損なしに通信できている
- KafkaとDebeziumのパフォーマンスは十分
 - RDBの変更を100ms未満でメッセージ化

今後の課題

- むしろアプリケーション側がKafkaに追いついていない
 - Lambdaの並列度向上やアプリの並列化が必要
- 監視対象メトリクスが未精査
 - CloudWatchでの取得可不可、ほか監視ツールやサービスの選定

OSSでマイクロサービス化を目指すならこういった技術が有用か。

OSSにこだわらないなら、クラウドプラットフォームのリソースを利用してもよいのだろう。

取り組んだことまとめ



- SRE でデプロイの改善を行った
- ブランチ戦略の見直し、それによるデプロイ方法の改善
- デプロイ方法を改善する中で CI/CD ツールの置き換えなどでデプロイをより効率化する改善を行った

今後やりたいこと



- デプロイとリリースの分離
 - デプロイ：アプリケーションを新しく配置すること
 - リリース：エンドユーザーがデプロイしたアプリケーションを使える状態にすること
- アプリケーションのインフラをコンテナ化
 - よりデプロイしやすい構成にしてリリースサイクルを高速にする

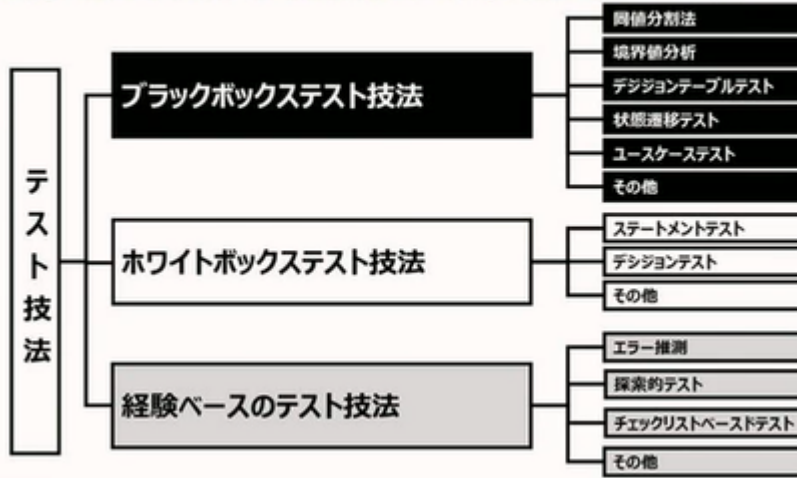
マイクロサービス化は年単位で実施していく

“GIHOZ”を活用したWeb APIテスト設計の勘所

テスト技法の種類

VERISERVE

➤ 例：JSTQB（ソフトウェアテストの国際資格である“ISTQB”の日本版）による分類

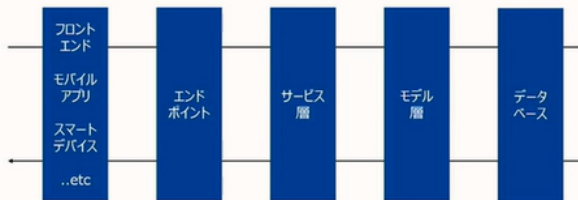


(参考)「E2Eテスト」

VERISERVE

➤ 「E2Eテスト」という言葉は、開発現場ごとに意味合いが異なるので、社外交流の際は要注意

➤ システムアーキテクチャに対するカバレッジのみが関心ごとの場合



E2E = End to End (頭からお尻まで)

エンジニアキャリアと組織のつくり方、これまでの10年と今後の10年

ベテランの声

すべてを理解するには、人知を超えているので、専門領域をつくるのがふつう。