Webアプリケーション開発基礎

2025年度エンジニア内定者研修

目次

- 講義の目的とゴール
- Webアプリケーションの構成要素
  - 前置き
  - ブラウザが表示するHTMLはどこから取得できるのか
  - 外部データソースを利用した動的なレンダリング
- バックエンドとフロントエンド
  - バックエンド
  - フロントエンド
- 開発で必要な知識
  - アーキテクチャ
  - テスト
  - CI/CD

講義の目的とゴール

講義の目的とゴール

- Webサイト（Webアプリケーション）がどのようにしてユーザーに届けられているのかを知る
- Webアプリケーションの基本的な考え方を知る
- フロントエンドとバックエンドについて理解する
- チーム開発に必要な知識を知る

Webアプリケーションの構成要素

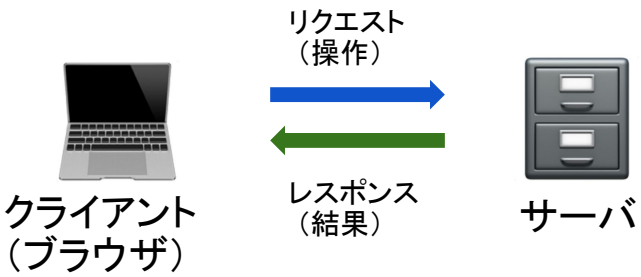
前置き

Webアプリケーションとは？

「Webサイト」と「Webアプリケーション」の違いとは？🧐

【Webサイト】

【Webアプリケーション】



Webページを構成する三大要素

Webページの「骨格」

見出し・段落・リストリンクなどページの構造を定義する役割

Webページの「見た目」

文字の色や大きさ・レイアウト・背景色などを装飾する役割

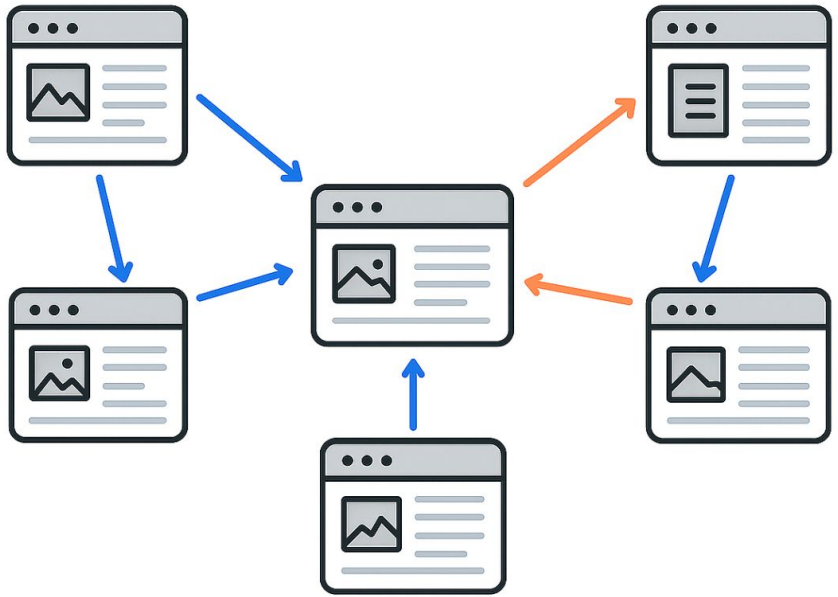
Webページの「動き・機能」

操作に応じた表示変更やサーバーと通信するなどページに動きを与える役割

HTMLとは？

HTML（HyperText Markup Language）：**ハイパーテキスト**を記述するためのマークアップ言語

→ 複数のテキストなどを相互に関連付けたシステム



HTMLの基本的な構造

HTMLの基本構造

DOCTYPE宣言

HTMLの種類を宣言

<!DOCTYPE html>

html要素

HTMLの最上位要素

head要素

メタ情報（タイトル等）

<meta charset “UTF-8”>
<title>HTML</title>

body要素

本文を記述

<h1>見出し</h1>

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset "UTF-8">
    <title>HTML</title>
  </head>
  <body>
    <h1>見出し</h1>
    <p>段落</p>
  </body>
</html>
```

ブラウザがHTMLをどのように理解するか

HTMLの基本構造

DOCTYPE宣言

HTMLの種類を宣言

<!DOCTYPE html>

html要素

HTMLの最上位要素

head要素

メタ情報（タイトル等）

<meta charset “UTF-8”>
<title>HTML</title>

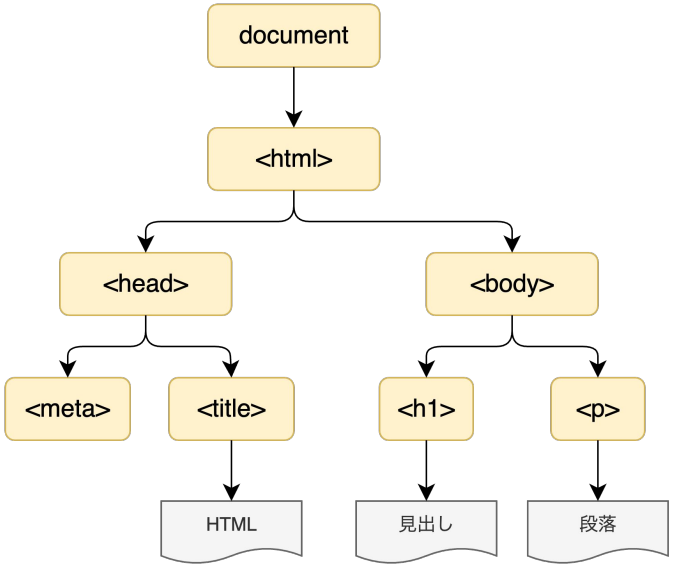
body要素

本文を記述

<h1>見出し</h1>

ブラウザが変換

DOMツリーとして理解（Document Object Model）



CSSとは？

CSS（Cascading Style Sheets）：DOMツリーの各要素に対して色・サイズなどの**スタイルを適用**

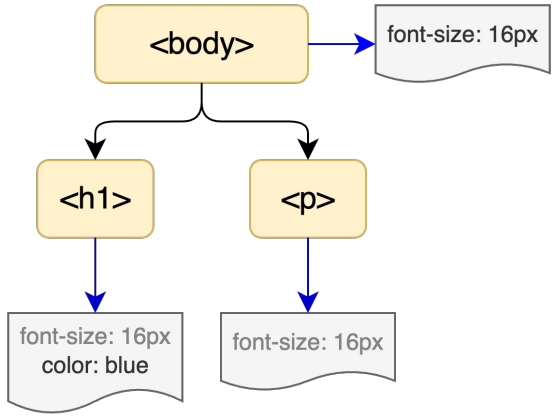
```
body {
  font-size: 16px;
}
h1 {
  color: blue;
}
```

基本ルール

セレクター { プロパティ : 値 ; }

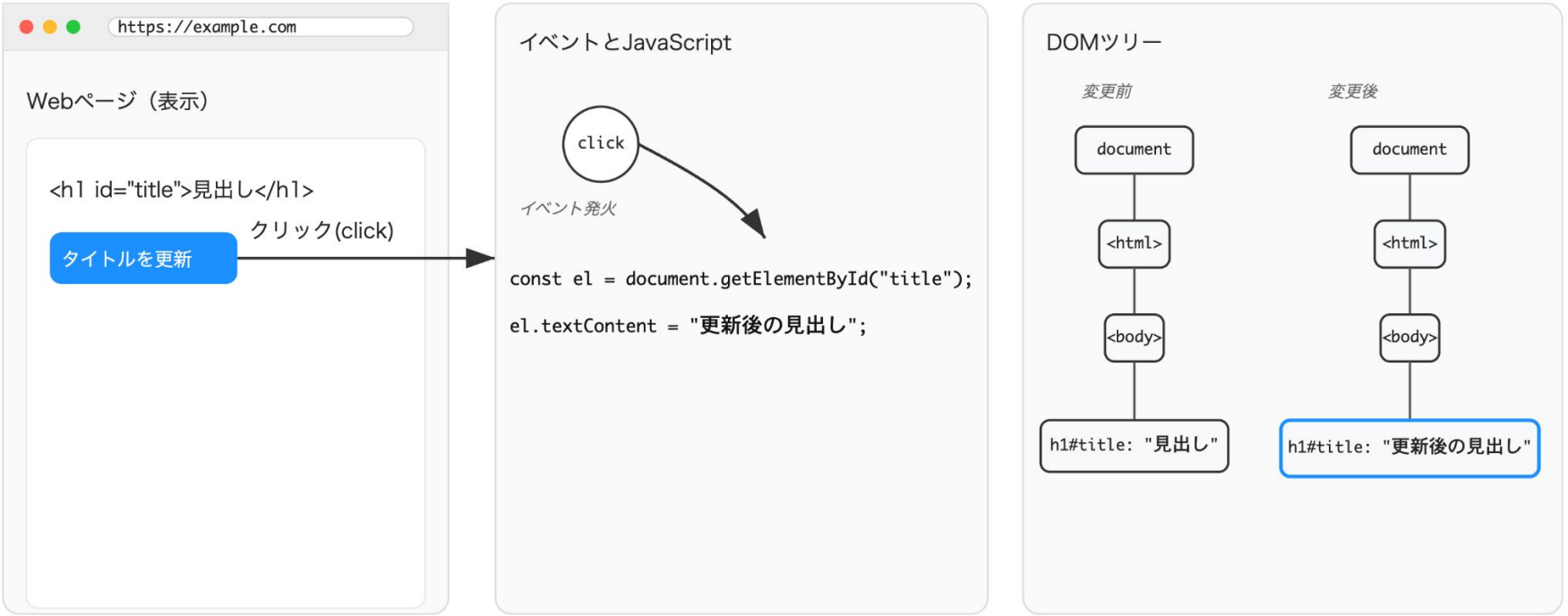
（どこの）　（何を）　（どうする）

CSSOMツリーとして理解



JavaScriptとは？

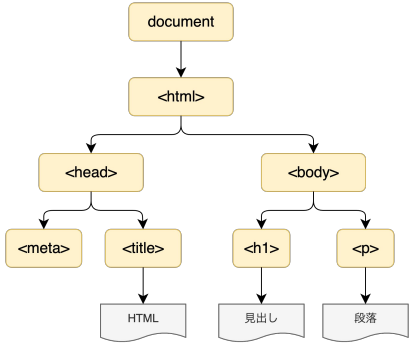
JavaScript：クリックや入力をきっかけに**DOMツリーを直接変更**（操作）することが可能



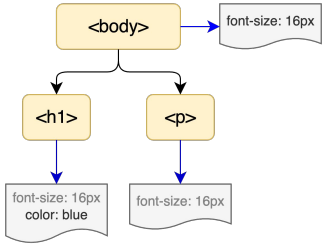
ブラウザの表示の仕組み:レンダリングツリーの構築

DOMツリーとCSSOMツリーからレンダリングツリーを形成(各表示要素の計算に使用)

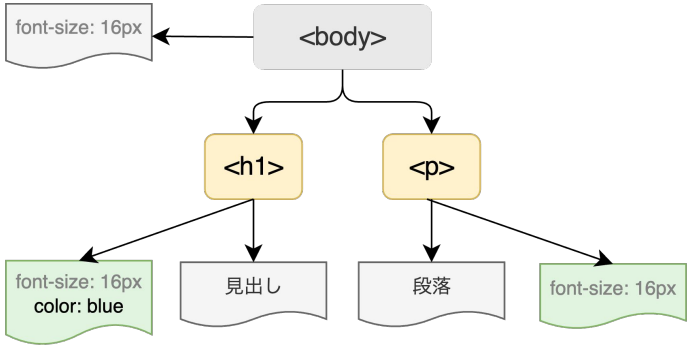
DOMツリー



CSSOMツリー



レンダリングツリー

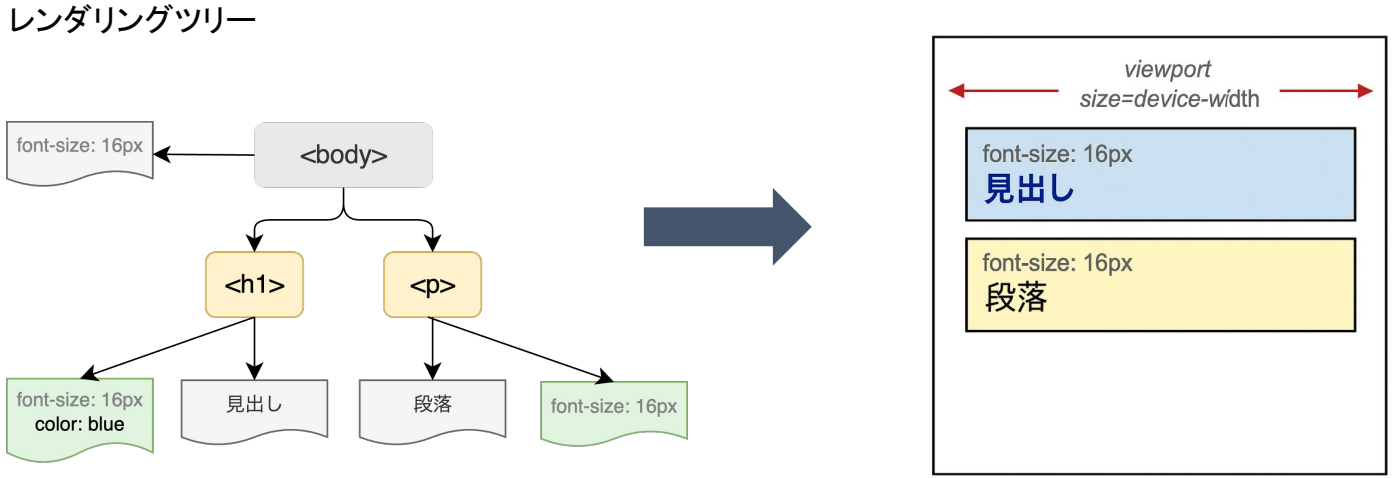


画面に表示されない要素は含まれない(ex. meta, head)

ブラウザの表示の仕組み:レイアウト

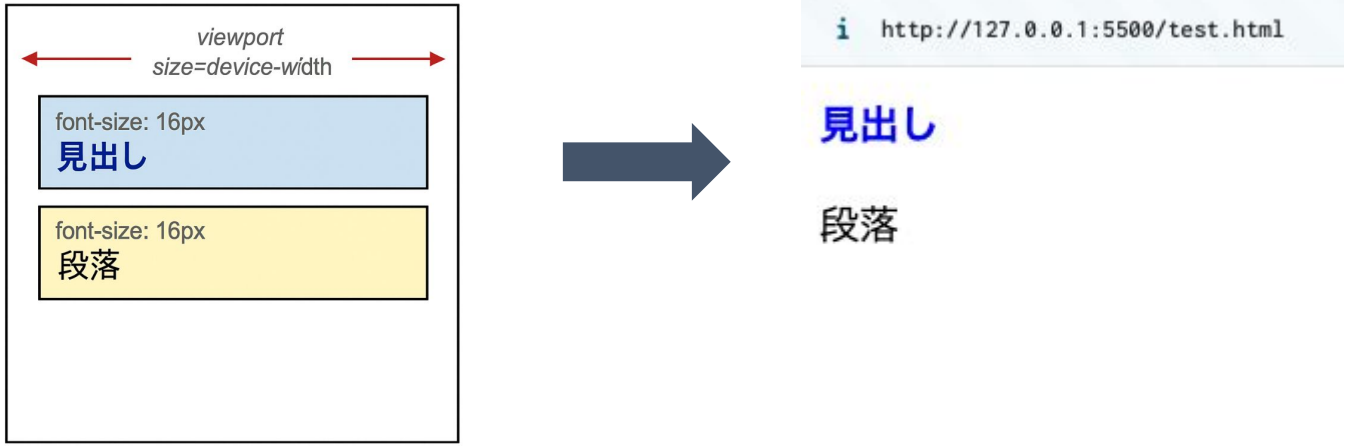
レンダリングツリーの各ノードの位置と大きさを計算

ex.「見出しを画面上部に配置して、その下に本文を置く」といった座標とサイズの計算をする



ブラウザの表示の仕組み：ペイント

- レイアウトで決まった情報を元にピクセルを塗る処理を行う（文字色・背景色などを描画）
- その後、レイヤー合成をされて最終的に画面に表示される



ブラウザが表示するHTMLはどこから取得できるのか

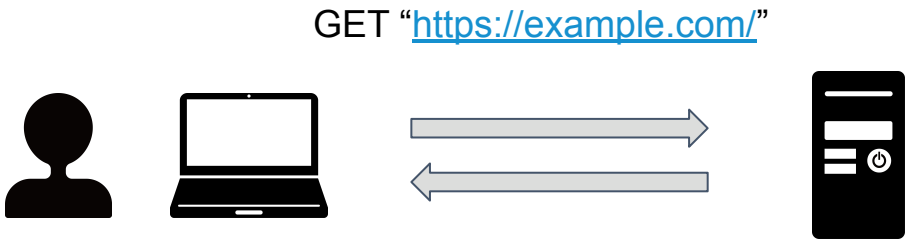
ブラウザが表示するHTMLはどこから取得できるのか

ブラウザで https://delishkitchen.tv/ にアクセスすると、以下の画面がブラウザで表示される

Webサイトを訪問すると、ブラウザが HTML を元にページを描画する
しかし、肝心の HTML のソースは手元にはない・・・
＝＞**「どこか」**から HTML をブラウザまで届ける必要がある 🧐

「サーバー」が HTML をブラウザに渡す

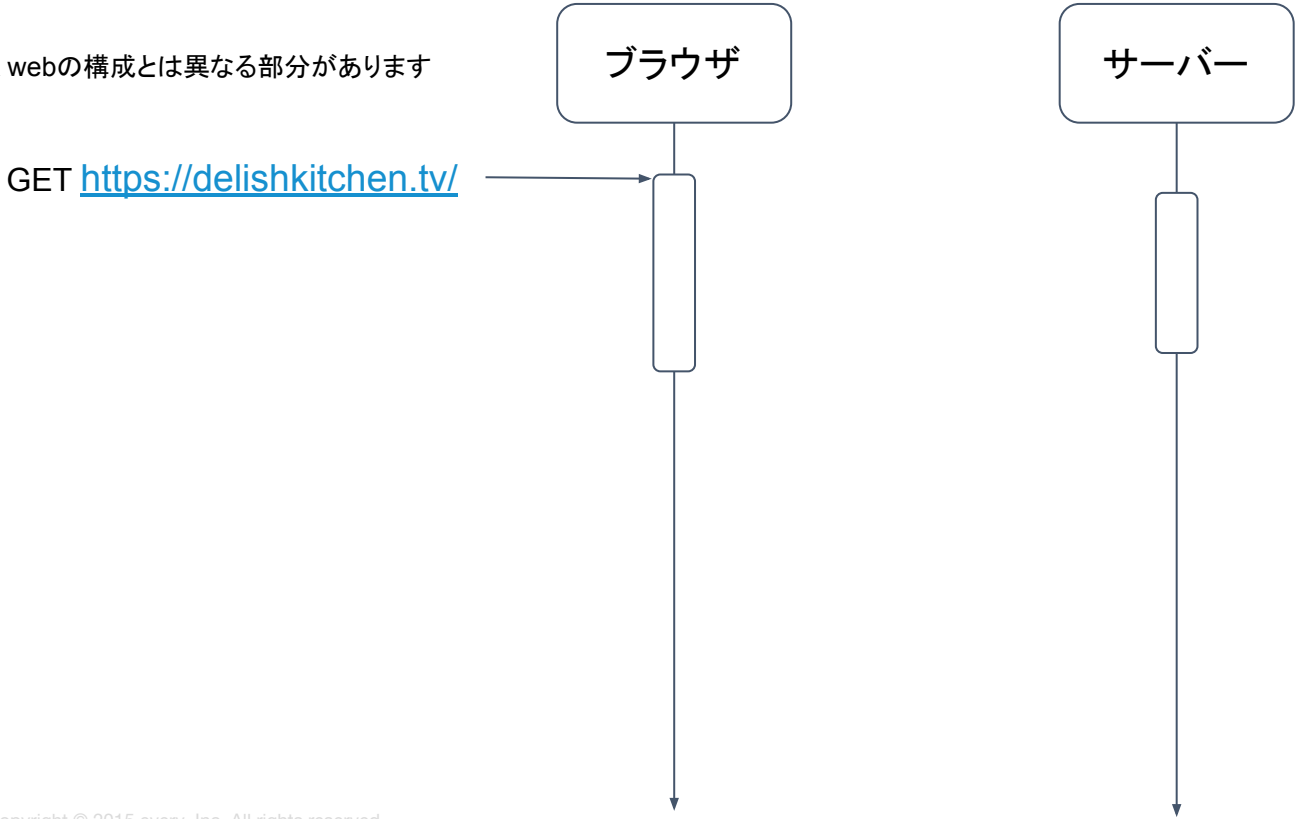
ブラウザ（クライアント）は、HTML を取得するためのリクエストを送信する
→サーバーがクライアントからのリクエストを受け取って、レスポンス (HTML)をブラウザに返す
（クライアント／サーバーシステムの一種）



第2章で登場したHTTPを用いて通信

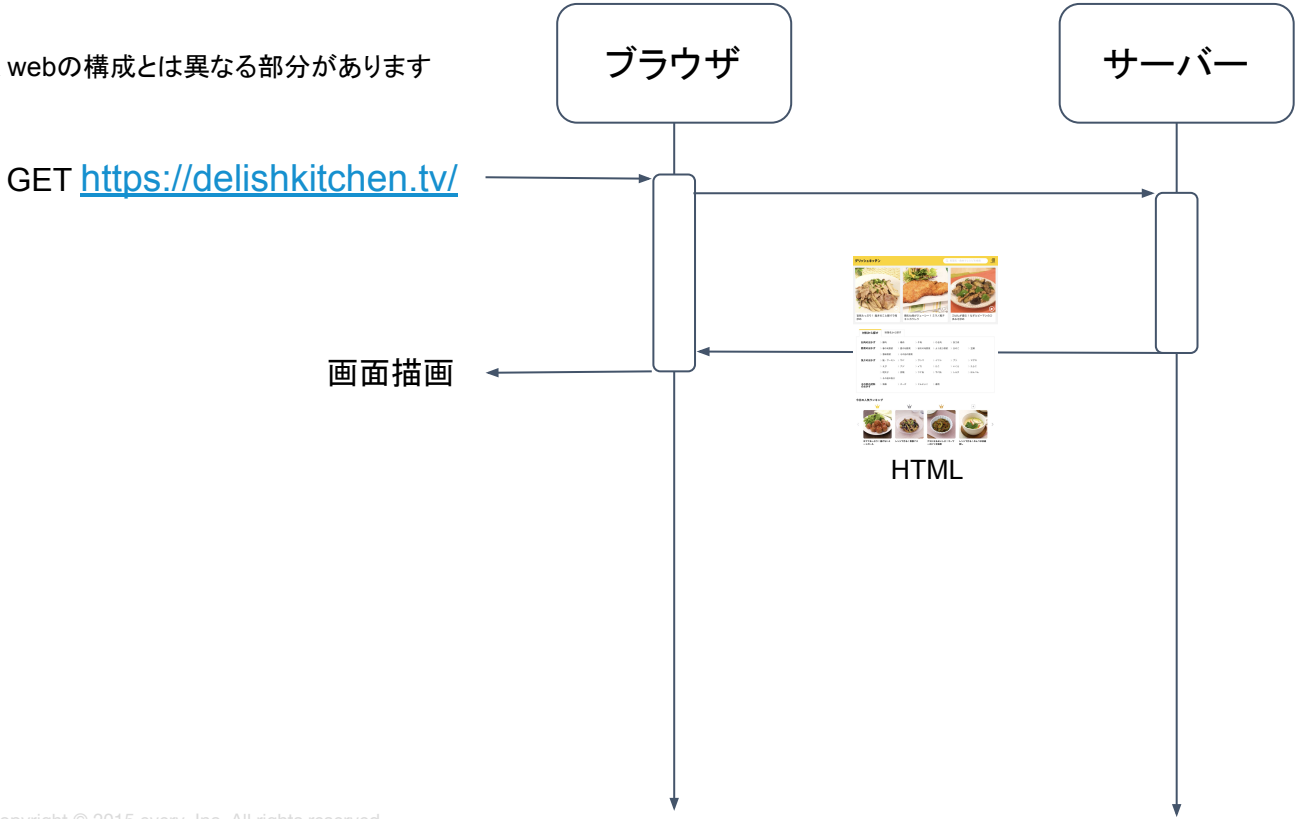
サーバーが HTML を返す方法①
リクエストを受け取るたびに静的 HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



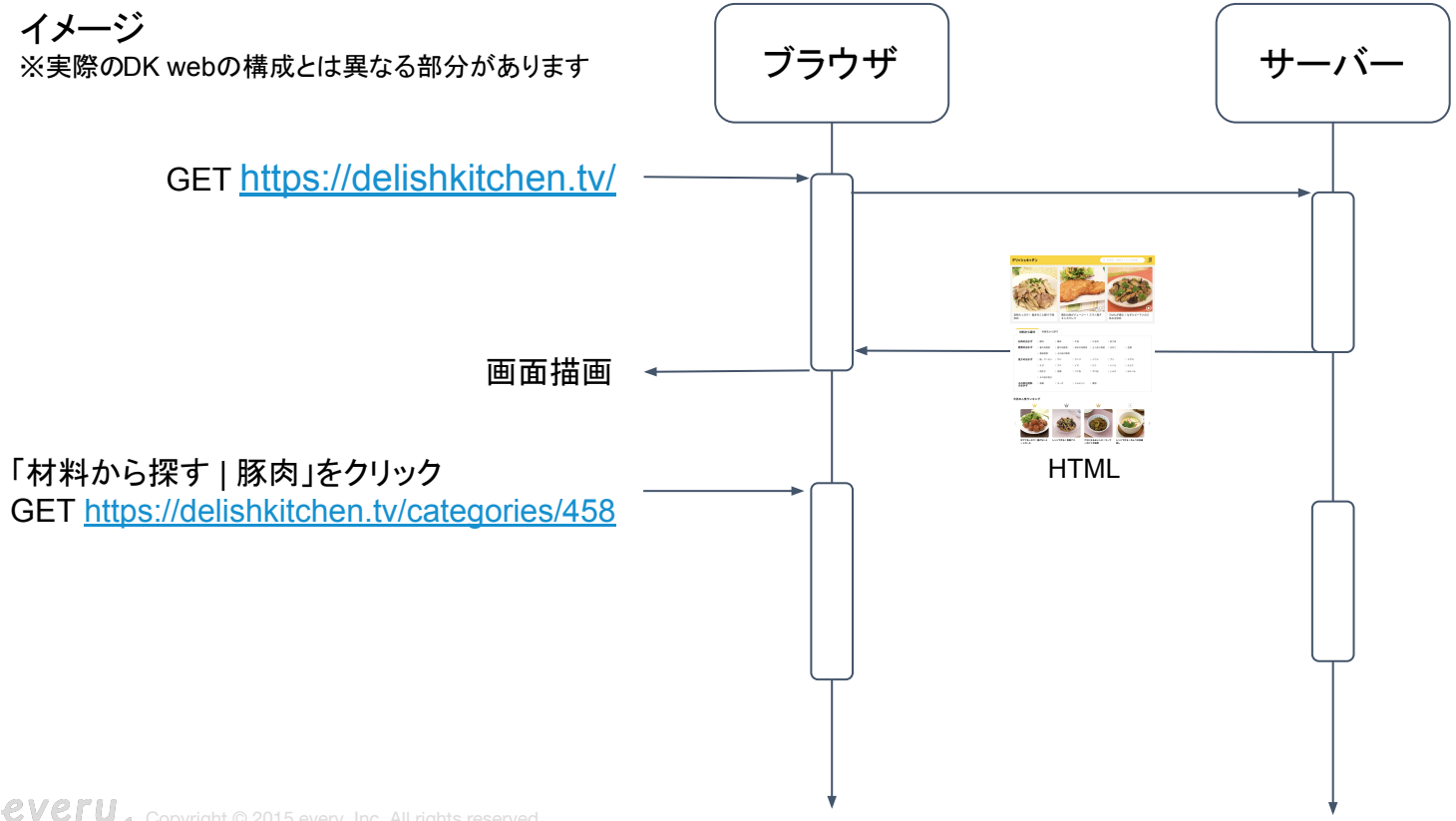
サーバーが HTML を返す方法①
リクエストを受け取るたびに静的 HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



サーバーが HTML を返す方法①
リクエストを受け取るたびに静的 HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります

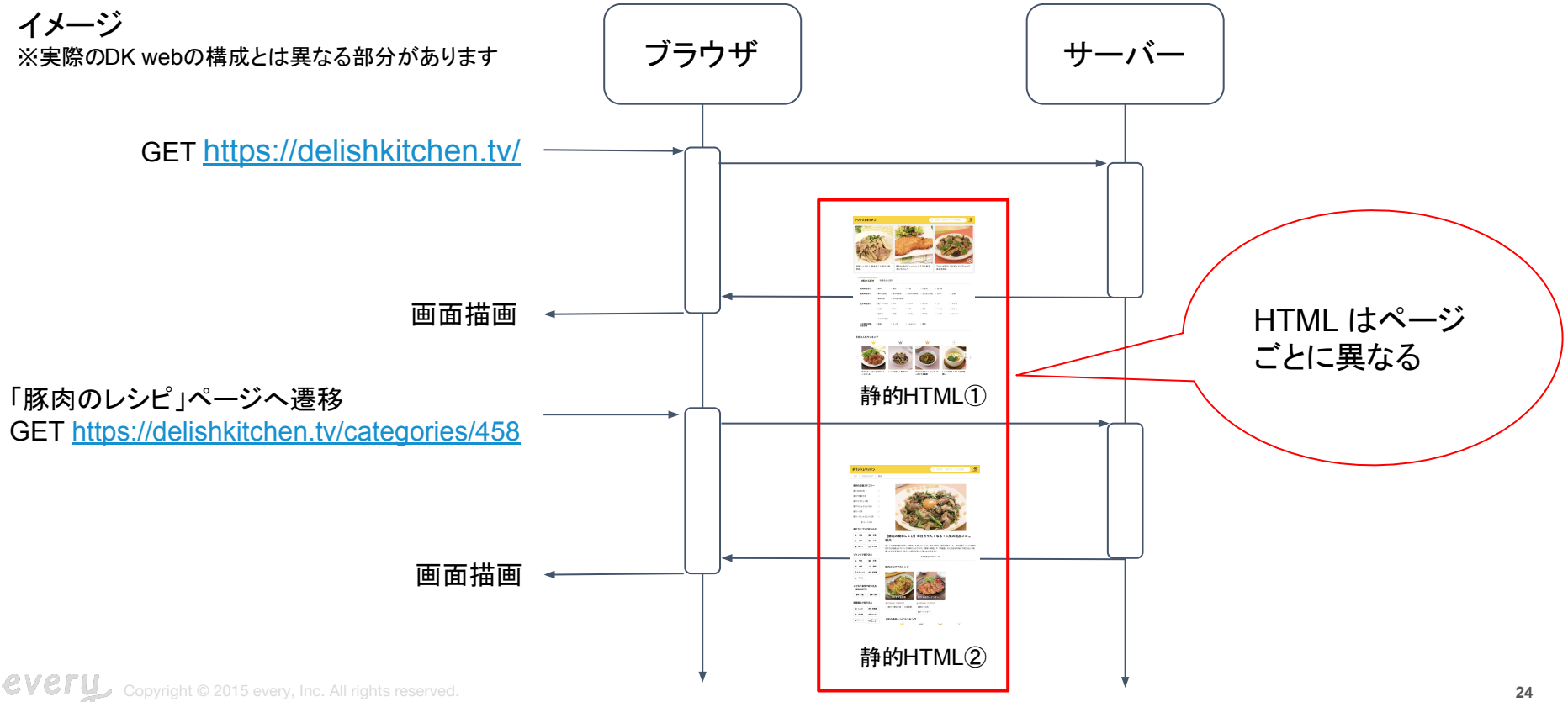


サーバーが HTML を返す方法①
リクエストを受け取るたびに静的 HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります

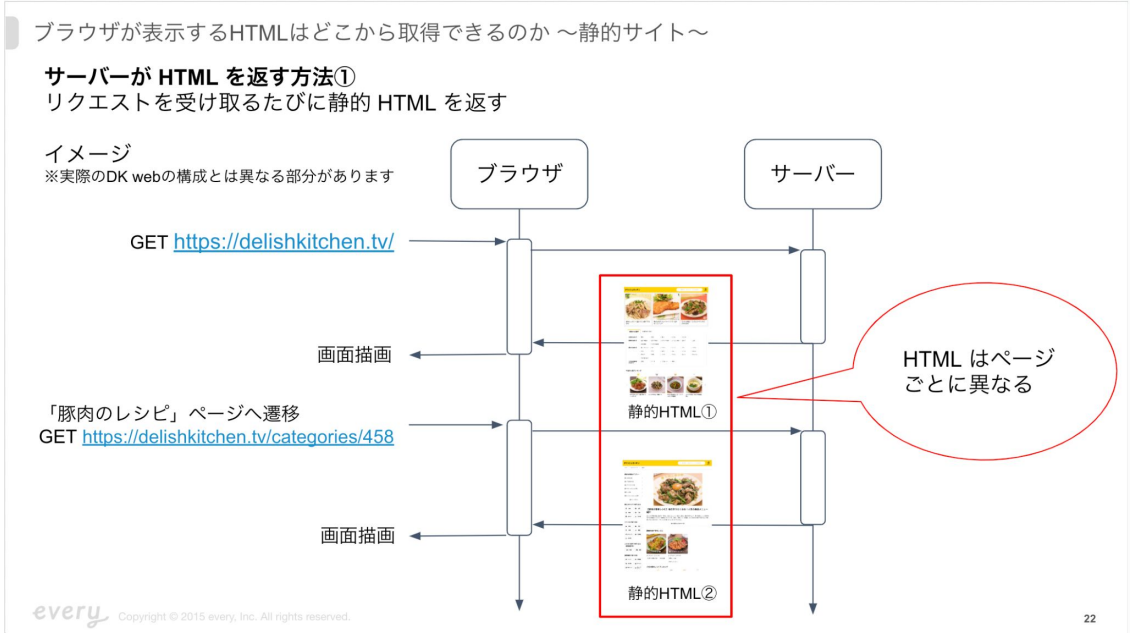
サーバーが HTML を返す方法①
リクエストを受け取るたびに静的 HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



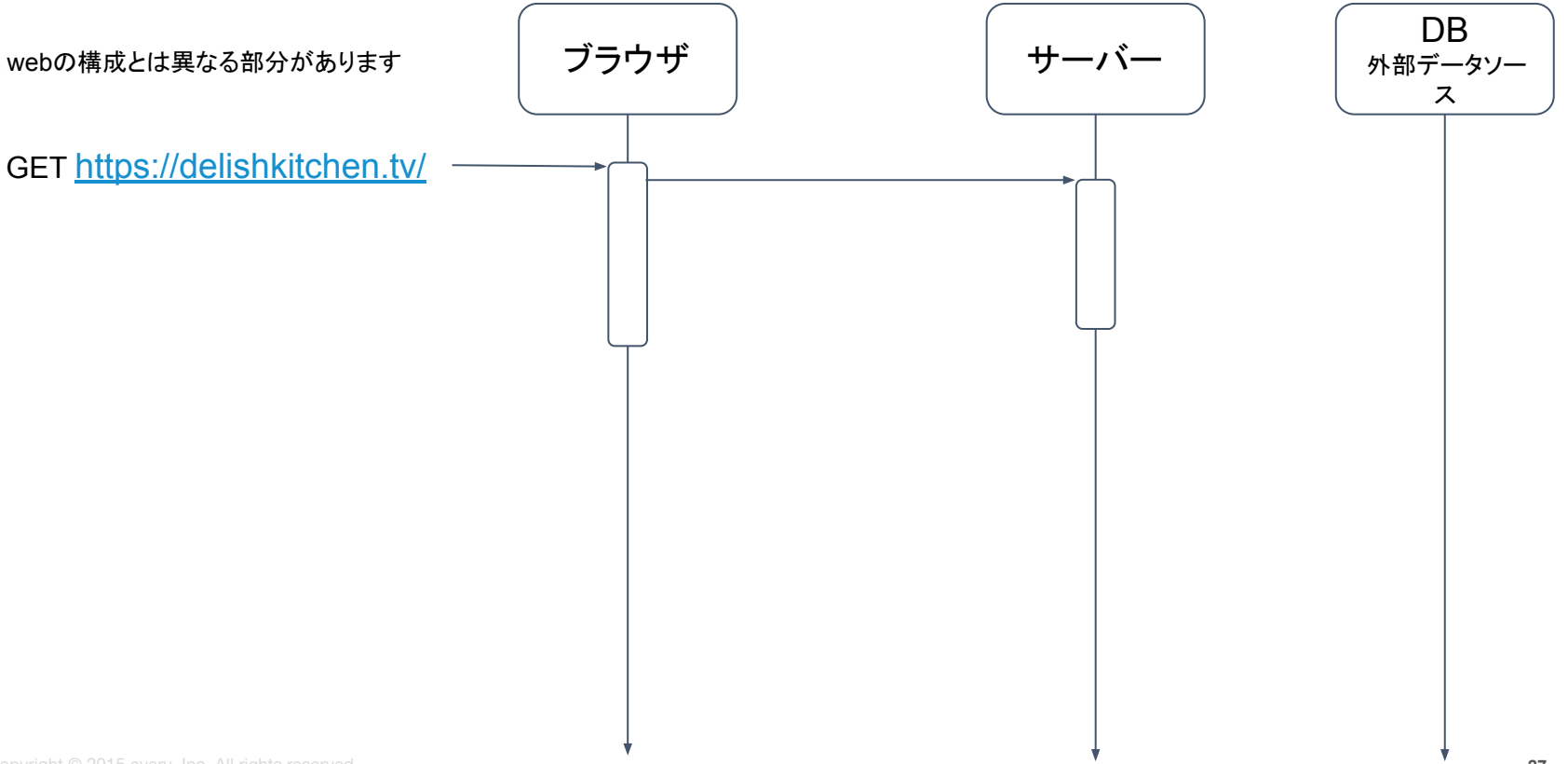
外部データソースを利用した動的なレンダリング

サーバーが静的な HTML を返す方式は、自由度が低くなってしまう



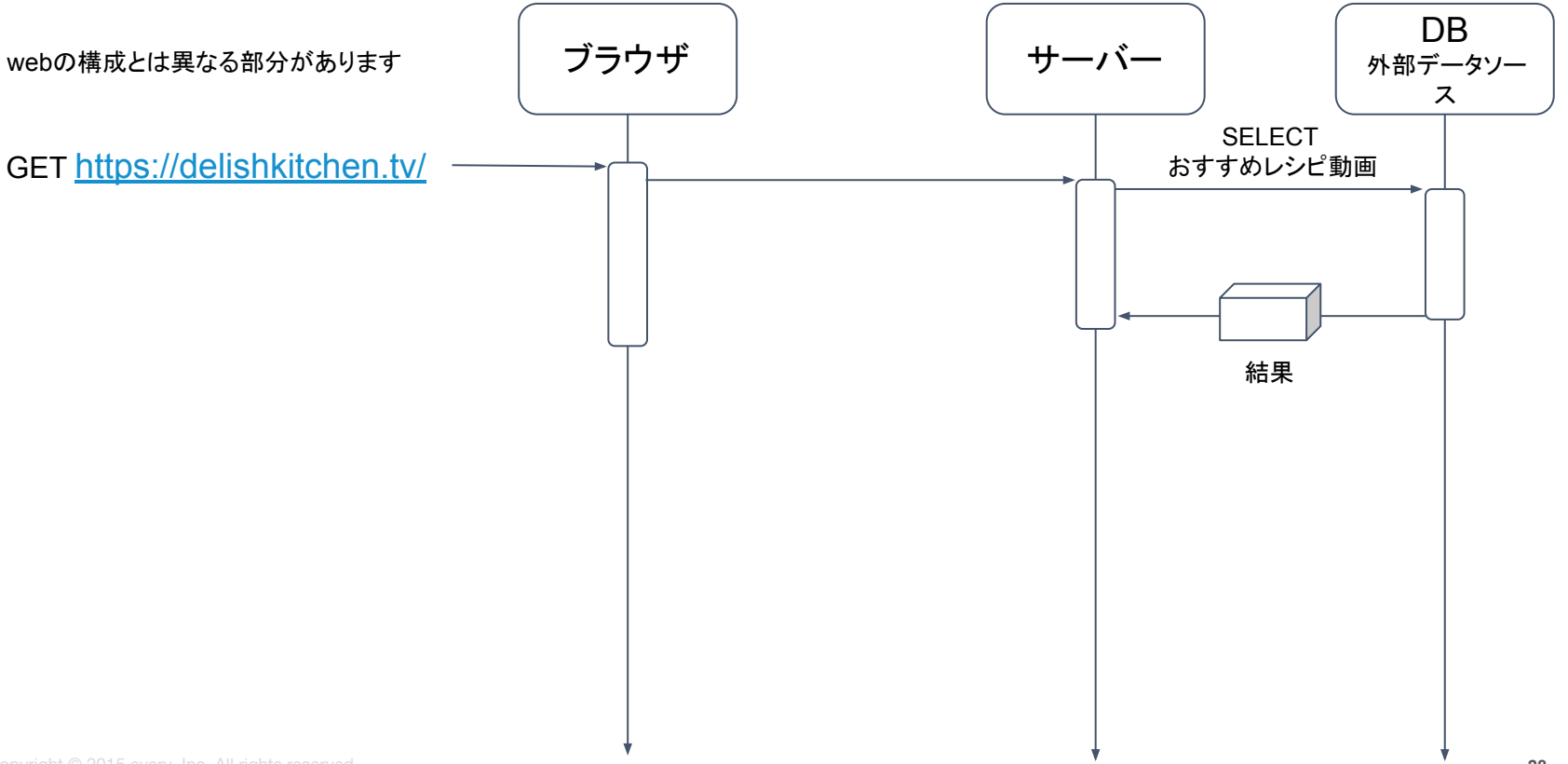
サーバーが HTML を返す方法②
リクエストを受け取るたびに動的にレンダリングした HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



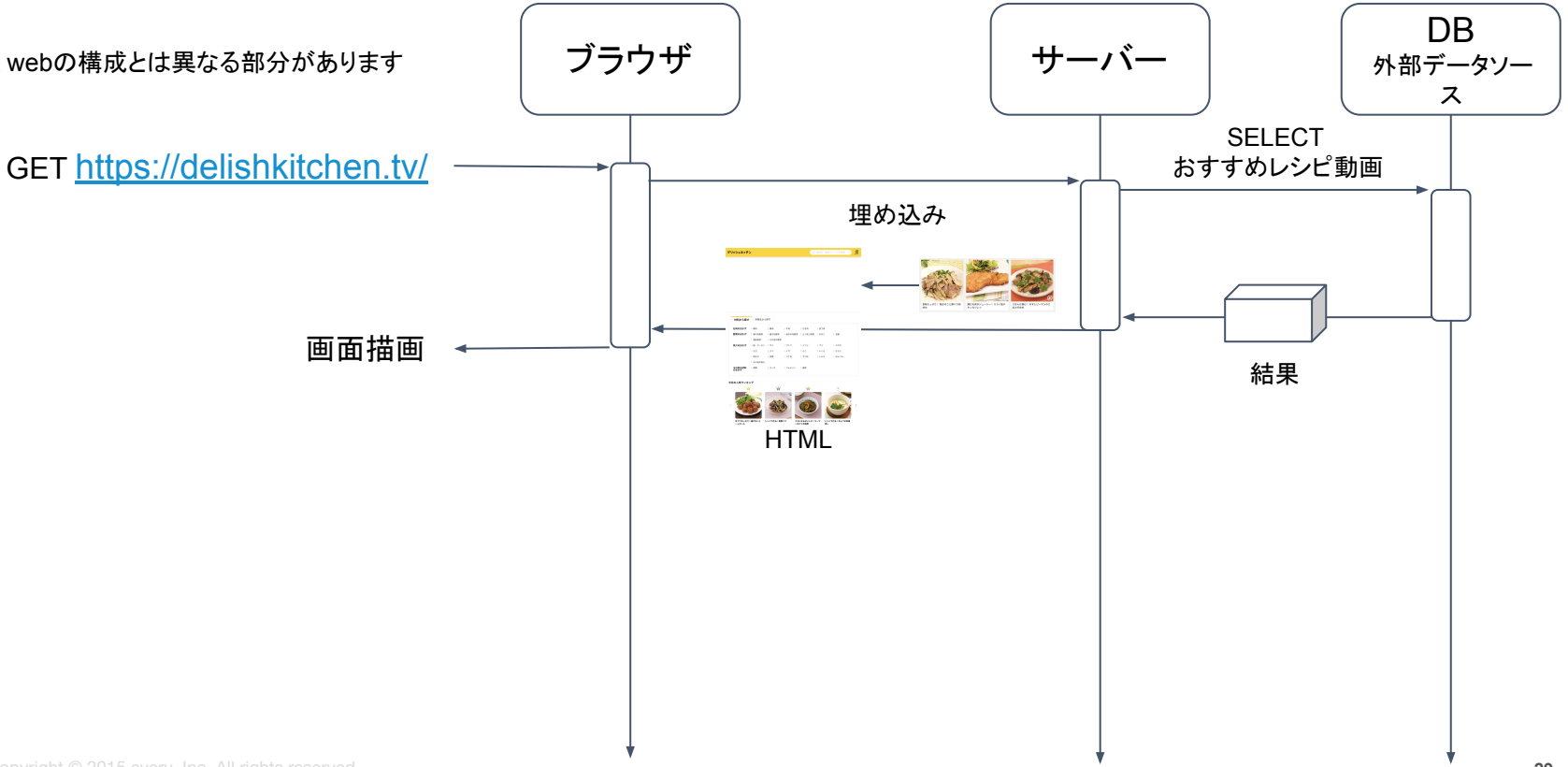
サーバーが HTML を返す方法②
リクエストを受け取るたびに動的にレンダリングした HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



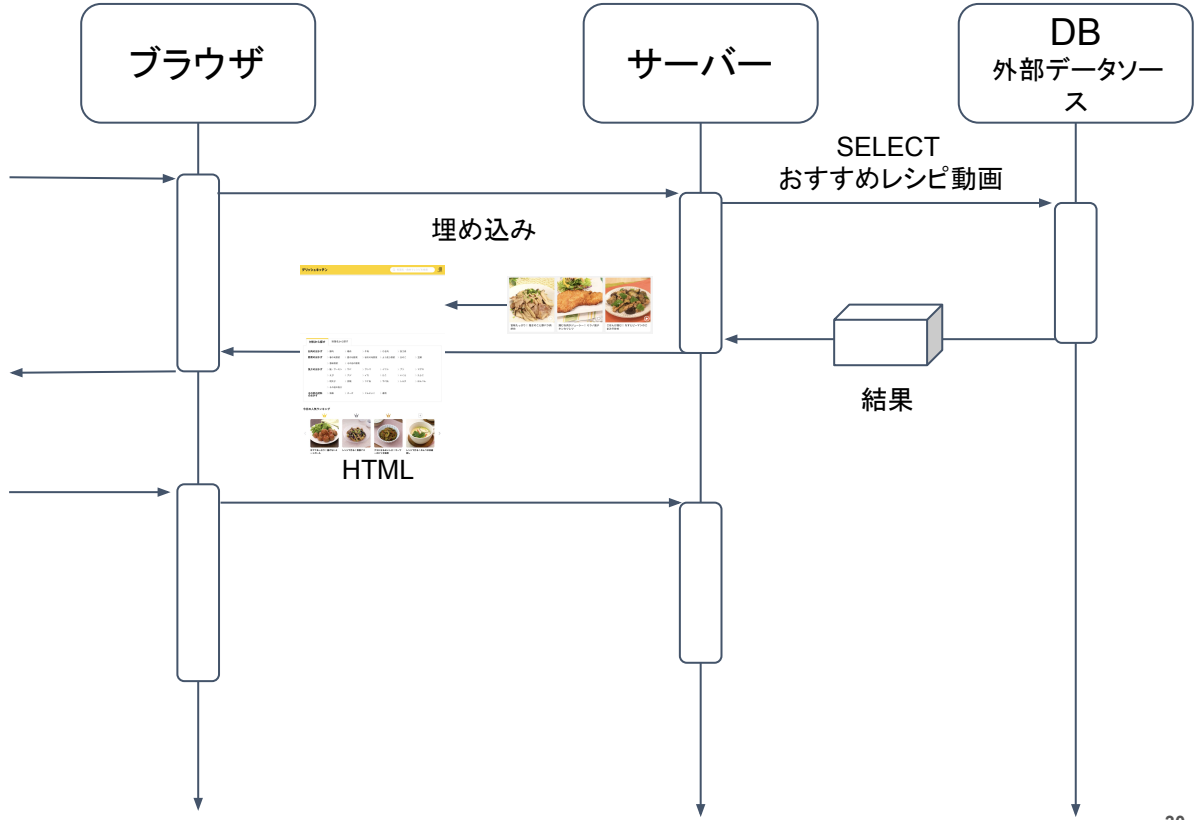
サーバーが HTML を返す方法②
リクエストを受け取るたびに動的にレンダリングした HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



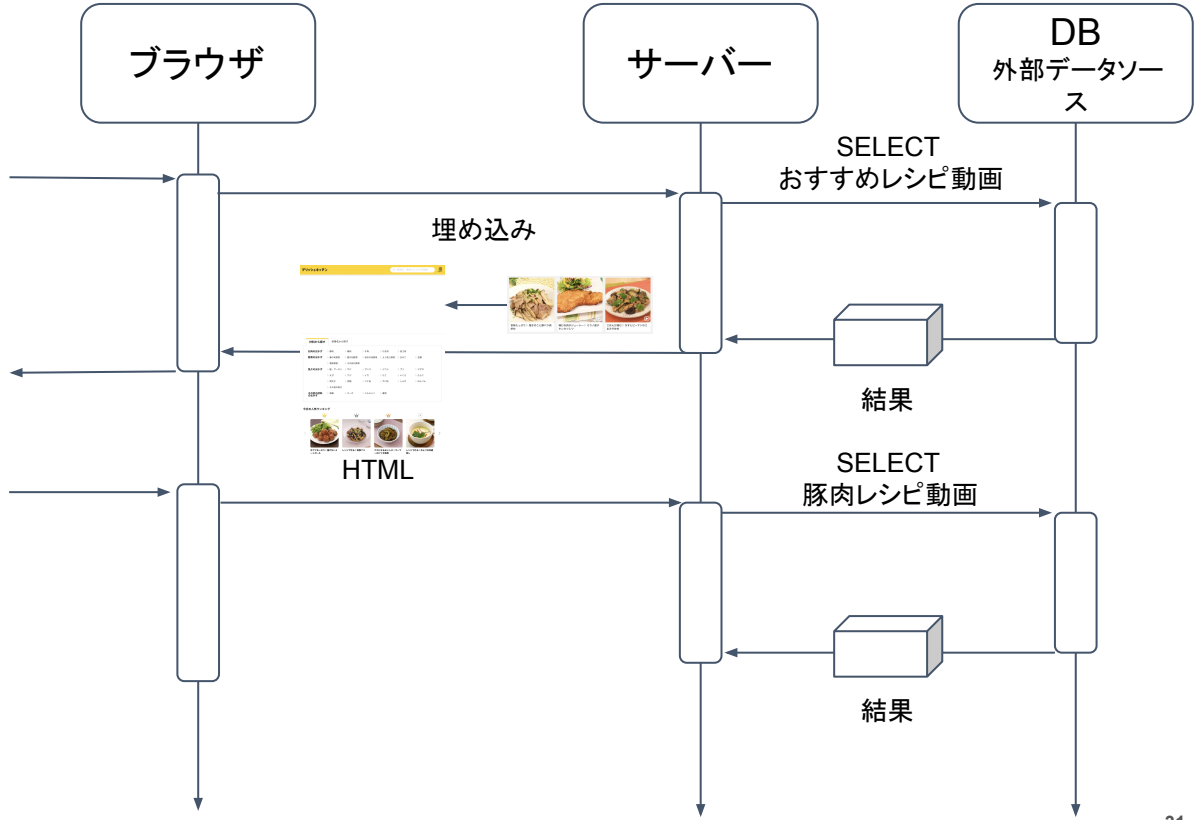
サーバーが HTML を返す方法②
リクエストを受け取るたびに動的にレンダリングした HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



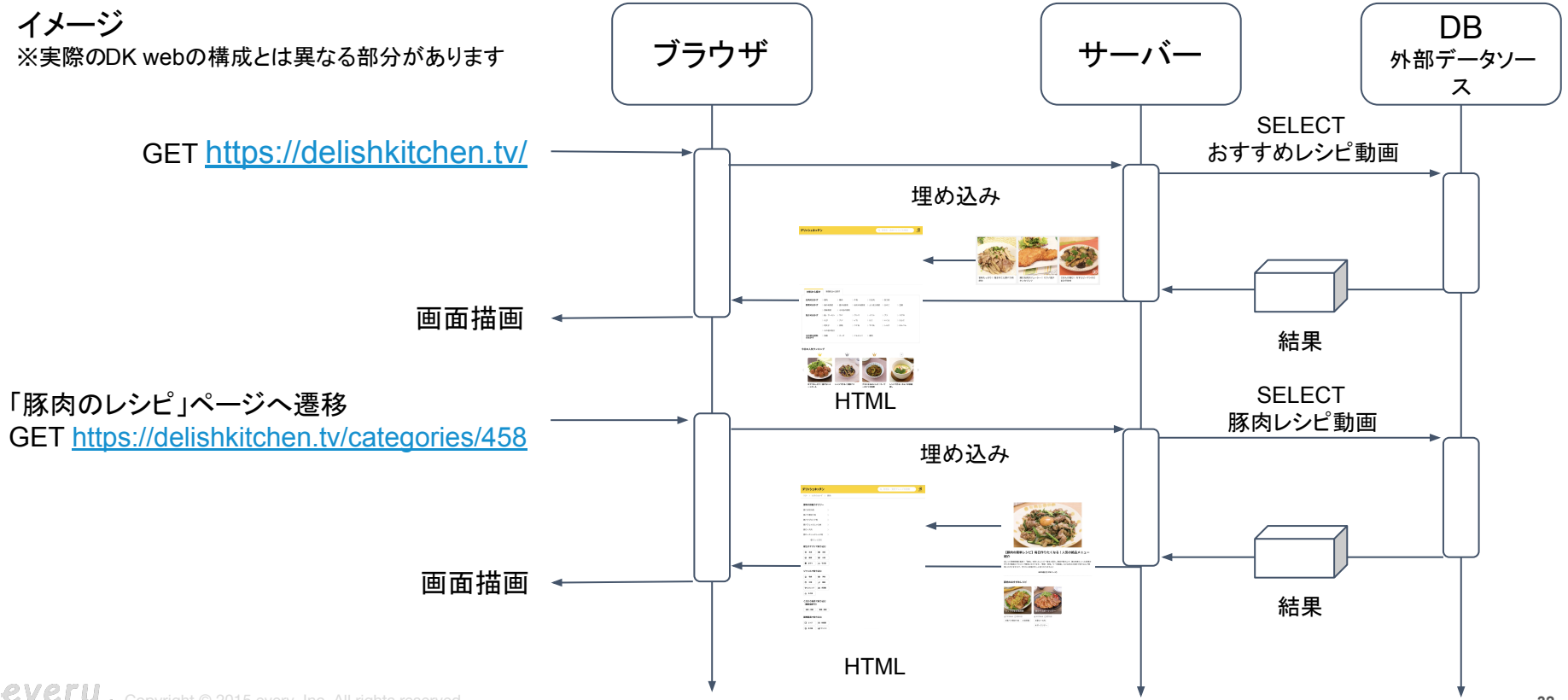
サーバーが HTML を返す方法②
リクエストを受け取るたびに動的にレンダリングした HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります



サーバーが HTML を返す方法②
リクエストを受け取るたびに動的にレンダリングした HTML を返す

イメージ
※実際のDK webの構成とは異なる部分があります

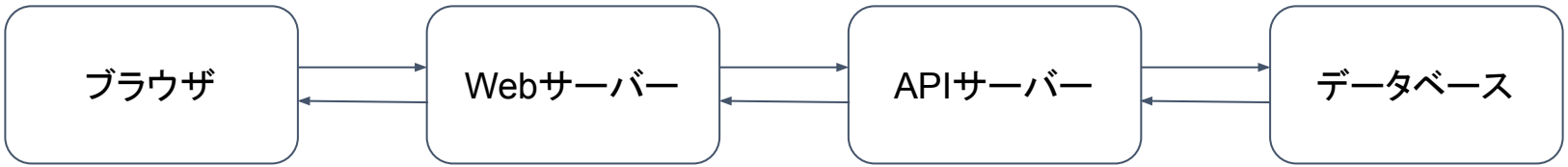


バックエンドとフロントエンド

主な登場人物

多くのWebアプリケーションはだいたい次のような構成になっている

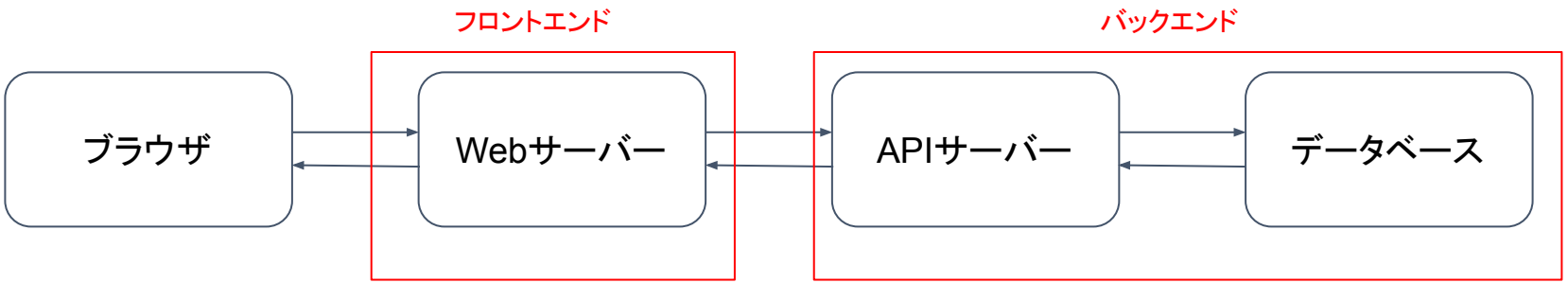
- ブラウザ
- Webサーバー
- APIサーバー
- データベース



主な登場人物

多くのWebアプリケーションはだいたい次のような構成になっている

- ブラウザ
- Webサーバー
- APIサーバー
- データベース



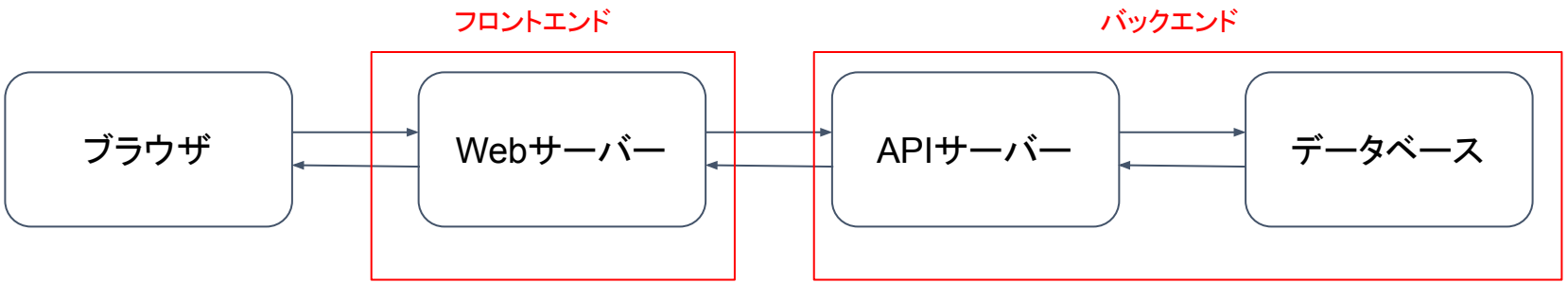
それぞれの役割

フロントエンド

- 見た目の部分
- UI/UX

バックエンド

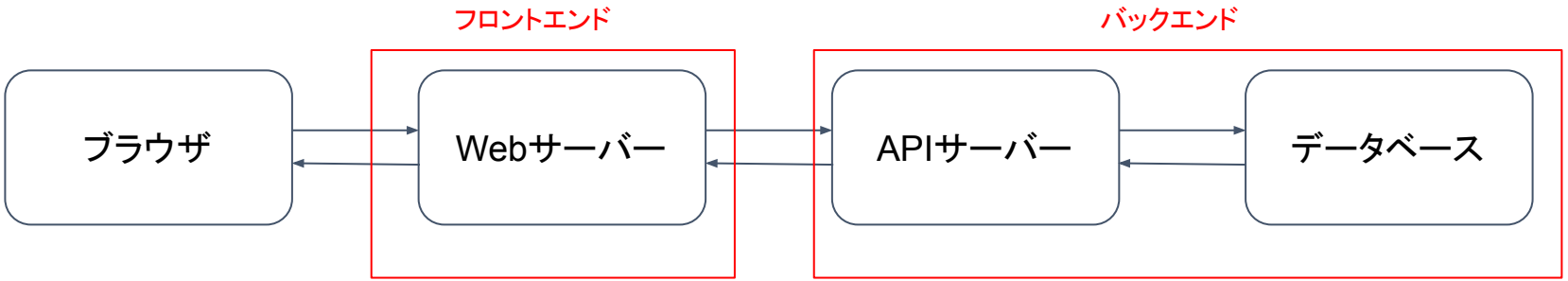
- 必要なデータの保存、整形、返却



なぜ分けるのか？

- 責務が違う
  - フロントエンド：見た目に集中
  - バックエンド：データの処理に集中
- 柔軟性
  - 元々Pythonで書かれていたAPIサーバーをGoで書き換える、でもWebサーバーはそのまま
  - Web用に作られていたAPIサーバーをモバイルでも使えるようにする

この辺りは「アーキテクチャ」の説明でより詳しく説明します



フロントエンド

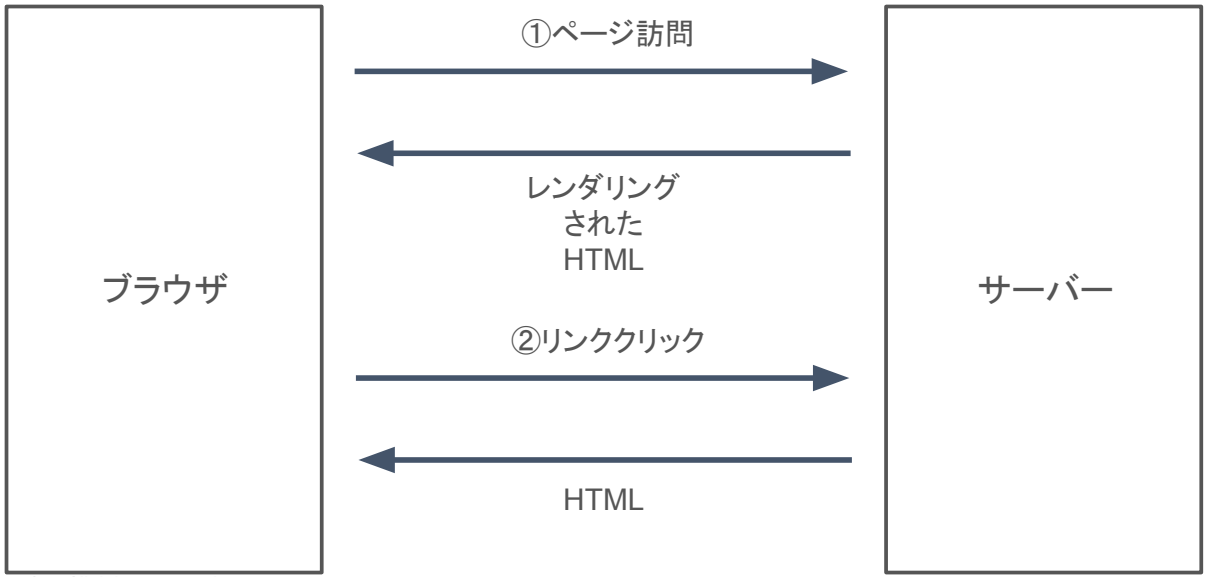
目次

- MPA, SPA
- SSR, CSR
- 使われている言語、フレームワーク
- DOM
- 非同期処理
- APIとの通信

MPAとSPA

MPA(Multiple Page Application)

- 複数のページからなる
- ページ遷移のたびにサーバーへリクエストを送る
- サーバーは完成したHTMLをレンダリングして返す
- 2000年代前半以前のWebアプリケーションでよく見られた



MPAのメリデメ

メリット

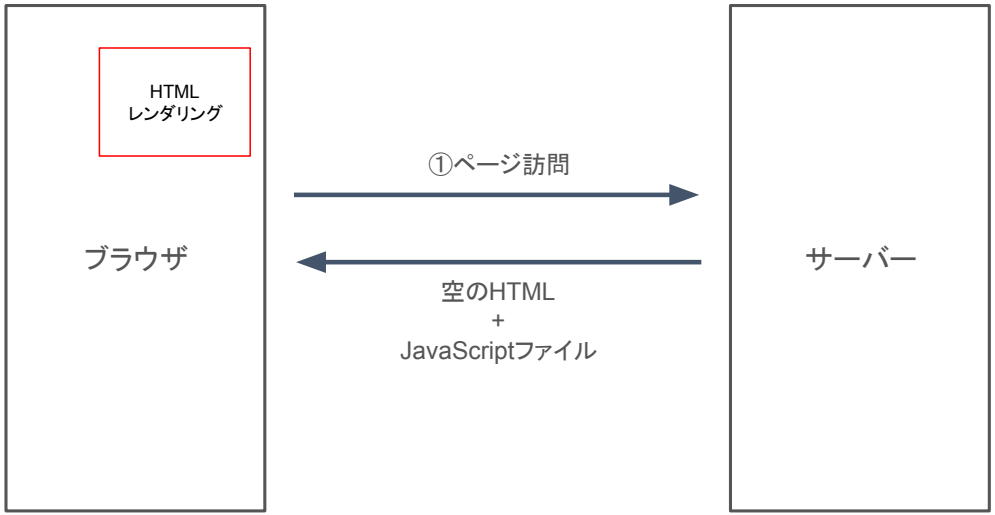
- 初回ロードが軽い
- SEOに強い

デメリット

- ページ遷移のたびにフルリロードを行う → UXが良くない
- CSSやJavaScriptなどの共通リソースを毎回読み込む必要があり、パフォーマンス面でオーバーヘッドが生じる

SPA(Single Page Application)

- JavaScriptを使って動的に更新
- ルーティングはブラウザ側で行われる
- History API
  - ブラウザのセッション履歴を管理する Web API (Web APIについては後述)
  - リロードせずとも、ブラウザの「戻る」「進む」ボタンが使える
- レンダリングもJavaScriptで動的に行う
- 2010年代前半



SPAのメリデメ

メリット

- ページ遷移が高速
-
- バックエンドとフロントエンドが分けやすい（UIとリソースの分離）

デメリット

- SEOに弱い
- 初回ロードが重くなりやすい
- JavaScriptがクライアント側で肥大化しがち→パフォーマンスの悪化

CSRとSSR

CSR (Server Side Rendering)

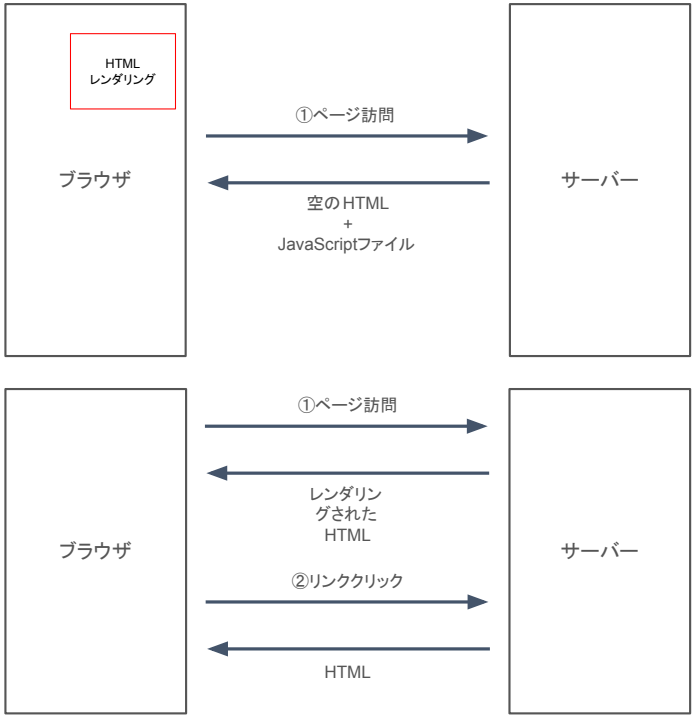
- HTMLとJavaScriptがブラウザに送られてレンダリング
- 典型的なSPAはこっち

SSR (Client Side Rendering)

- サーバーでレンダリング済みのHTMLを返す
- 典型的なMPAはこっち

SPAも初期表示だけ SSRを採用すれば、デメリットを補える！

最近のフレームワークは大体CSR, SSRどちらにも対応している



使われている言語・ライブラリ・フレームワークなど

主な言語・ライブラリ・フレームワーク

言語

- HTML
- CSS
- JavaScript
- TypeScript (JavaScriptに型システムを入れたもの)

ライブラリ

- React
- Vue

フレームワーク（どちらもSSRに対応）

- Nextjs
- Nuxtjs

Node.js

- JavaScriptのランタイム環境
- Google Chromeと同じV8エンジンを採用
- **JavaScriptをサーバーサイドで実行可能**
- fsモジュールなどでファイルの読み書きもできる（サーバーならでは）

npm

- Node.jsのパッケージ管理ツール

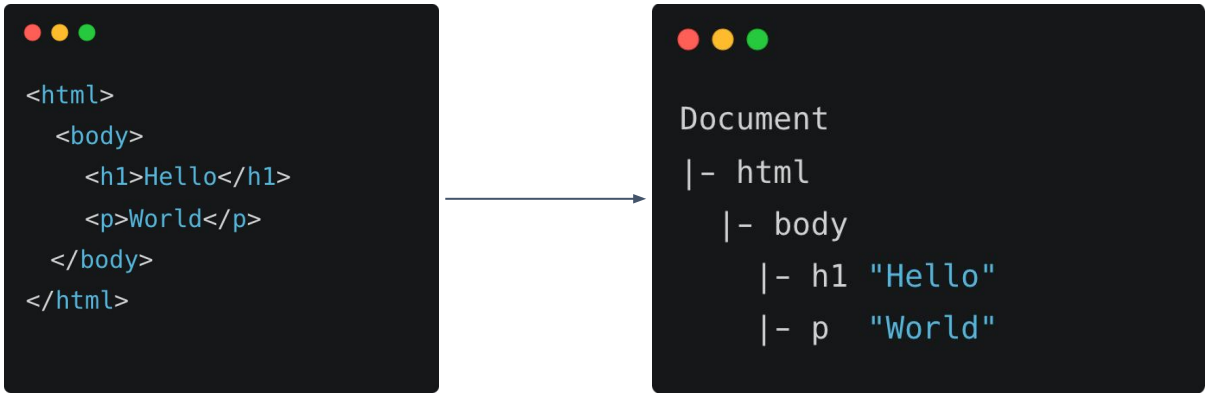
ランタイム環境は他にもBunやDeno、パッケージ管理ツールはyarnやpnpmなど色々ある

DOM

DOMとは

- DOM (Document Object Model)
  - HTMLやXMLドキュメントの構造、操作を定義したもの
- DOMツリー
  - DOMから作られる木構造のオブジェクト

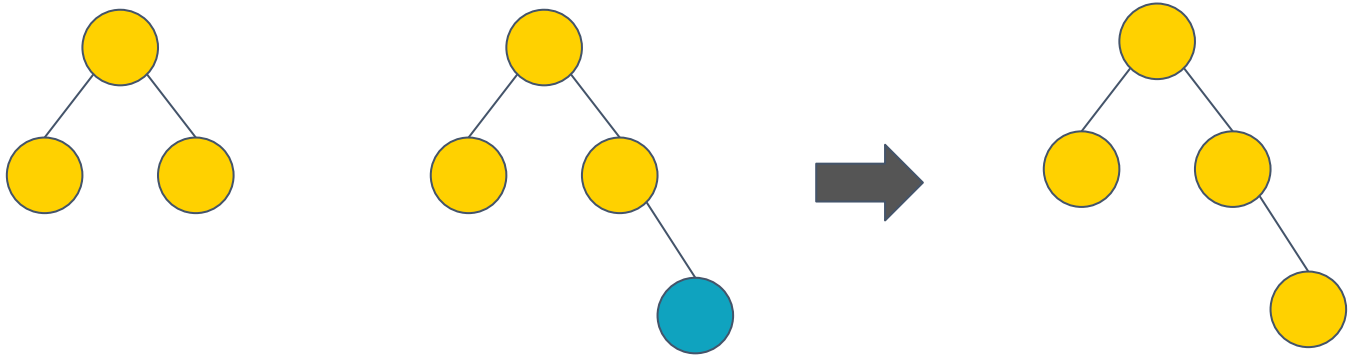
JavaScriptはこのDOMを操作して画面の更新を行う



仮想DOM

- 実際のDOMの軽量コピーをメモリ上に持つ仕組み
- VueやReactなどで用いられる
- 差分を計算し最小限の変更のみを実際のDOMに適用、同期する

→ JavaScriptとDOMの中間レイヤーとして仮想DOMを挟むことで、**実DOMを効率的に操作できる！**



メモリ上で差分を計算

実際のDOMに反映

仮想DOMが出てきた背景

- jQuery時代
  - .addClass()や.append()など、命令的にDOMを直接いじる
  - 「こうして、ああして」の処理を書いていくので、状態が追いづらい

- ReactやVueなど、宣言的なフレームワークの登場
  - 「今の状態はこう」と宣言的にUIを記述できる
  - 宣言したものを全て再描画していたらコストが高すぎる
    →差分の計算の必要性が高まり、仮想 DOMが導入された

JavaScriptの非同期処理

非同期処理とは

↔ 同期処理：複数のタスクを実行する際に順に実行される

ファイル操作やAPIリクエストのような重いタスクを実行する際に、処理をバックグラウンドに移すことでタスクの処理を止めることなく別のタスクを実行できる

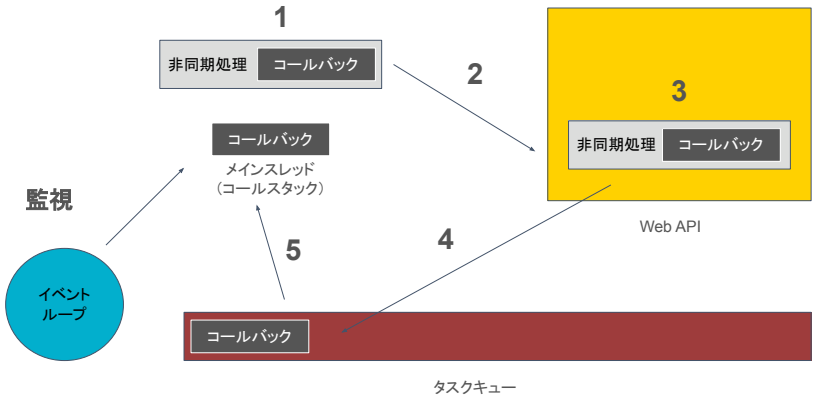
→ **JavaScriptエンジンはシングルスレッド**
これのみでは非同期処理は実現できないので他の仕組みを利用している！

ブラウザでの非同期処理

Web APIを用いることで非同期処理を実現している

Web APIの例

- fetch api
- storage api
- DOM



ブラウザは複数のスレッドを持っており、メインスレッドとブラウザのスレッドに分けられる

1. メインスレッドから非同期処理を呼び出す
2. Web APIが呼び出される
3. Web APIがブラウザのスレッドで実行される
4. 完了後コールバックがタスクキューに追加される
5. メインスレッドが空くとコールバックがメインに移動し実行される

非同期処理は何が偉いのか？

- 再読み込みをせずに画面の状態を変えられる
- 例：Google Mapの移動
- 今まで散々上のことについて喋ってきたが、当時はすごいことだった

参考：

Ajax (Asynchronous JavaScript and XML)

- JavaScriptを使った非同期処理の技術
- 現在はあまり使われていない
- XMLHttpRequestを使ったHTTP通信
- 初期のGoogle Mapに使われた
- 直接DOM変更を行う

APIとの通信

APIとの通信

動的コンテンツや負荷の高い処理、外部システムの利用などF/Eで完結できないことが多い

- レシピデータやユーザーデータなどの動的なコンテンツはDBに保存されている
- 画像の加工やDBから取得したデータの操作など負荷の高い処理はB/Eが行う
- 認証認可などの複雑な処理は自作したくない

APIとの通信

動的コンテンツや負荷の高い処理、外部システムの利用などF/Eで完結できないことが多い

- レシピデータやユーザーデータなどの動的なコンテンツはDBに保存されている
- 画像の加工やDBから取得したデータの操作など負荷の高い処理はB/Eが行う
- 認証認可などの複雑な処理は自作したくない

→ **APIを呼び出すことで上記の問題を解決する ！**

API通信フロー

リクエストの作成・送信

- url, method, parameterなどを指定してリクエストを送信する

レスポンスの受信

- status codeやbody(jsonなど)を受け取る

データの利用

- レスポンスを元に状態やDOMの更新を行う

これらの処理はfetchやaxiosなどの**非同期通信**を用いる

バックエンド

目次

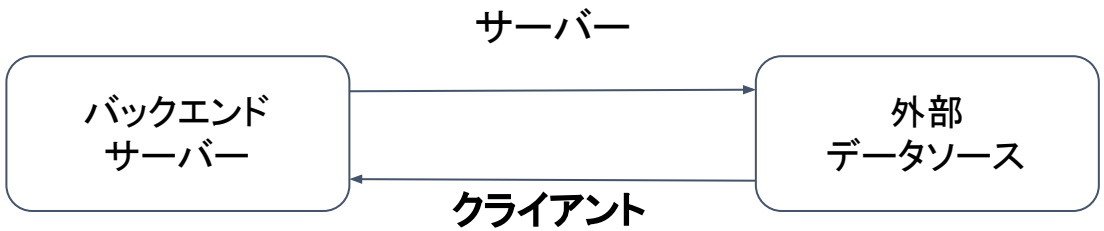
- データの処理
- ビジネスロジック
- API
- 認証・認可
- 使われている言語、フレームワーク

データの処理

基本的な役割

- 外部ソースからデータを取得しフロントエンドに返す
- フロントエンドから来たデータを外部ソースに永続化したり送ったりする

基本的にはこの2つだけ



外部データソースから見るとバックエンドサーバーがクライアント

外部ソースの例

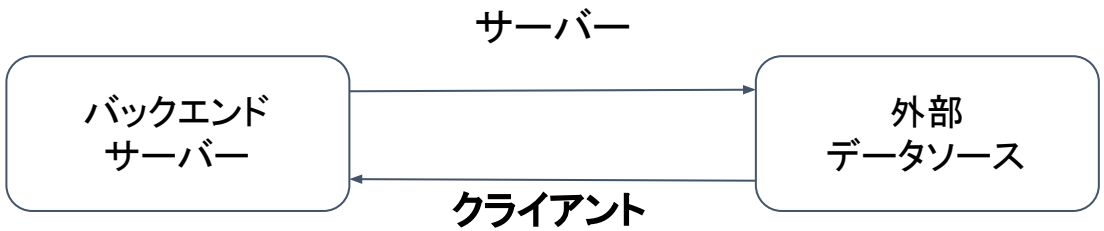
- データベース
  - RDB
    - MySQL
    - PostgreSQL
  - NoSQL
    - DynamoDB
    - MongoDB
- キャッシュ
  - オンメモリキャッシュ
  - リモートキャッシュ
    - Redis
    - Amazon ElastiCache
- 外部API
  - 認証や課金用のAPI
  - 企業が提供するAPI
- クラウドストレージ
  - Amazon S3
  - Google Cloud Storage

ビジネスロジック

基本的な役割

- 外部からデータを取得しフロントエンドに返す
- フロントエンドから来たデータを外部に永続化したり送ったりする

基本的にはこの2つだけ

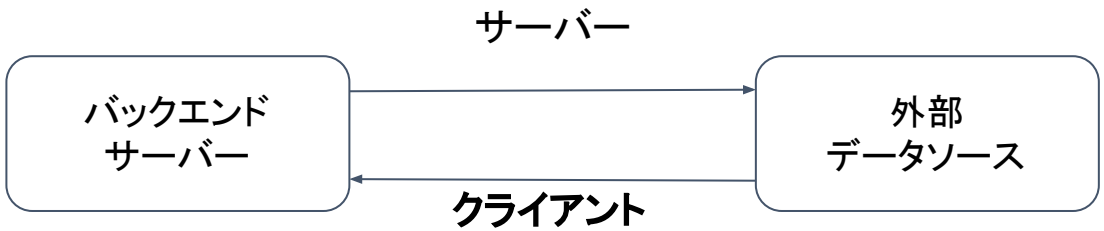


外部データソースから見るとバックエンドサーバーがクライアント

基本的な役割

- 外部からデータを取得し（整形して）フロントエンドに返す
- フロントエンドから来たデータを（整形して）外部に永続化したり送ったりする

基本的にはこの2つだけ



外部データソースから見るとバックエンドサーバーがクライアント

ビジネスロジック

（雑に言うと）アプリケーション特有の整形、バックエンドサーバーの中核の役割

- ECサイト：注文金額が5000円以上なら送料を無料にする
- SNS：フォローしていない人の投稿は表示しない
- 予約サイト：前日になったらキャンセルできない

など。これはフロントエンドに書かない

レスポンスのための整形

- 日付のフォーマット
- DBには1がiOS、2がAndroidとして保存されているのを、stringに変換する

などはビジネスロジックではない

が、少なくとも後者はバックエンドサーバーで実装すべきこと

API

API (Application Programming Interface)

- プログラムにとってのインターフェース
- Webアプリケーションで使われるほとんどのAPIは以下のどちらかの形式のデータを返す
  - JSON
  - XML

```json
{
   "user": {
     "name": "山田太郎",
     "age": 28,
     "email": "taro.yamada@example.com"
   }
 }
```

```xml
<user>
    <name>山田太郎</name>
    <age>28</age>
    <email>taro.yamada@example.com</email>
</user>
```

HTTPリクエスト

ヘッダ

- Content-Type：レスポンスの種類
- Authorization：認証情報
- Cache-Control, Expires：キャッシュに関する情報
- Last Modified：最後に更新された日時

リクエスト種類

- GET：取得
- POST：作成
- PUT：（全体）更新
- PATCH：（一部）更新
- DELETE：削除

REST API

- よく使われるAPIの設計手法
- 次の6つの条件を満たす
  - クライアント／サーバー (Client-Server)
  - ステートレス (Stateless)
  - キャッシュ (Cache)
  - 統一インターフェース (Uniform Interface)
  - 階層システム (Layered System)
  - オンデマンドのコード (Code-On-Demand)

例:

/users/1 → ユーザー情報

/users/1/orders → 注文履歴

GraphQL

- 比較的新しい設計手法
- Facebookが開発したOSS
- 必要なデータをクエリで指定できる
- 一度に複数のデータを取得できる

例:

query { user(id:1){ name, orders } } → クエリで複数のデータを取得

認証・認可

認証・認可

- 認証（Authentication）
  - ユーザーが本人であることを証明する仕組み
- 認可（Authorization）
  - ユーザーに対して、アクセスできるリソースや操作を制御する仕組み

似てるけど別物

例：特定のCIDRからだけアクセスを許可する（認証はしてないけど認可はしている）

OAuth2.0

- 認可のための仕組み
- アクセストークンを使って、自分の認証情報を第三者のアプリケーションに渡すことなく、捜査の許可をする
- 代表的なフローはAuthorization Code Flow (複雑なので各自調べてください)

例：あるアプリがGoogleカレンダーに予定を書き込みたいとき、ユーザーがGoogleログインして承認すると、アクセストークンを使ってカレンダーへの書き込みを行う

OpenID Connect

- OAuthの拡張仕様
- アクセストークンに加えてIDトークン(JWT)も渡すことで、認可と同時に認証も行う

例：SSO（シングルサインオン）

アーキテクチャ

アーキテクチャとは

→システムを構築するための設計思想や構造

アーキテクチャの目的

- システム構造の明確化
- システムの保守性、拡張性の向上
- 再利用性の向上

システム構造の明確化

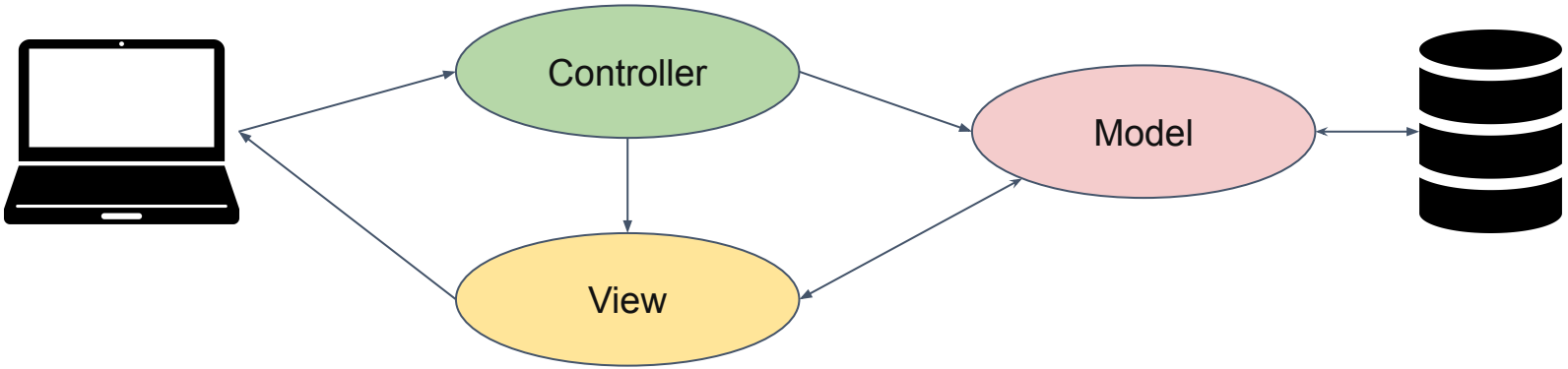
システム全体の見通しを良くし、開発者間で共通の理解を持つことができる

→ システム全体を把握せずとも、特定の部分を効率的に開発したり修正したりすることができる

システム構造の明確化

例. MVCを用いた場合

- Model : ビジネスロジックを管理する。データベースや外部APIからデータ取得・保存を行う
- View : ユーザーに情報を表示する部分。画面のレイアウトなどのインターフェースを提供する
- Controller : ユーザー入力を受け取り、ModelやViewに伝える。

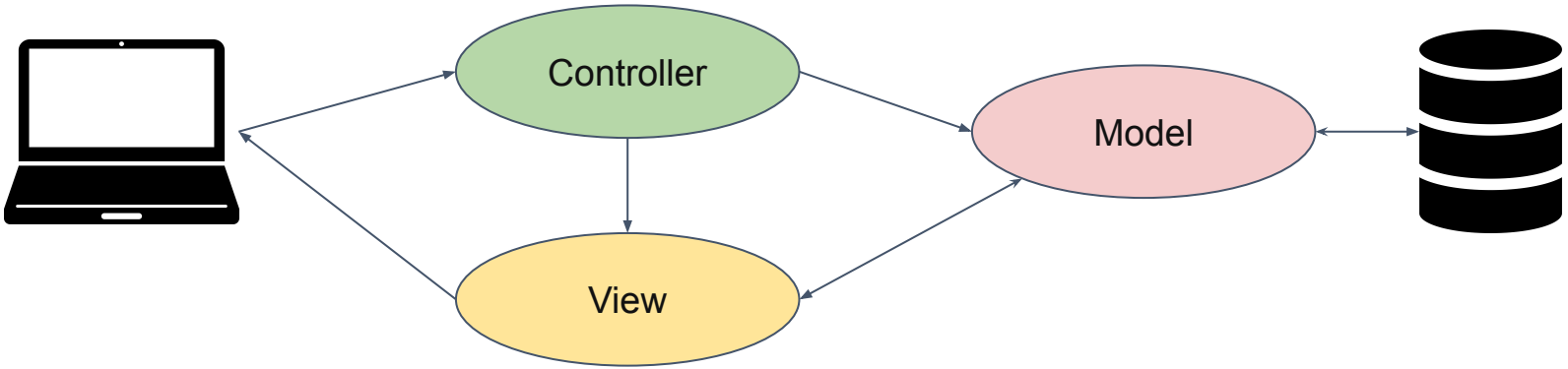


システム構造の明確化

Q.「入力フォームの位置を変えてほしい」どこを修正すれば良い？

A. View

→ ModelやControllerなどでどのような実装がされているか完全に把握する必要がない！



保守性・拡張性の向上

保守性 : システムの修正や更新する際の容易さ

拡張性 : システムに新しい機能を追加する際の容易さ

保守性・拡張性の向上

保守性：システムの修正や更新する際の容易さ

- コードの分割やテスト可能な設計を行うことで向上する
  - 責務を明確にし最小限にすること(SRP Single Responsibility Principle)
  - 重複したコードを減らす(DRY Don’t Repeat Yourself)

保守性・拡張性の向上

拡張性：システムに新しい機能を追加する際の容易さ

- 設計の柔軟性が高く、変化に対応しやすいことが求められる
  - オープン・クローズドの原則(OCP)に基づいた設計
  - 将来の変更を想定した構造化
    - インターフェースの汎用化
    - 例 REST APIなど

再利用性の向上

開発したコンポーネントやモジュールを他のプロジェクトや機能で再利用できる

- コスト削減：開発やテストの手間を軽減
- 一貫性: 同じロジックを利用することでエラーやバグの可能性の減少

再利用性の向上

例：UIコンポーネントのモジュール化

- ボタンやモーダル、フォームなどを再利用可能なコンポーネントとして設計

```
// Button.js
import React from 'react';
import './Button.css';

const Button = ({ children, onClick }) => {
  return (
    <button className="resusable-button" onClick={onClick}>
        {children}
    </button>
  );
};

export default Button;
```

アーキテクチャの種類

B/E

アーキテクチャの分類

層ベースのアーキテクチャ

- Layered Architecture
- Clean Architecture
- Hexagonal Achitecture

ドメイン中心アーキテクチャ

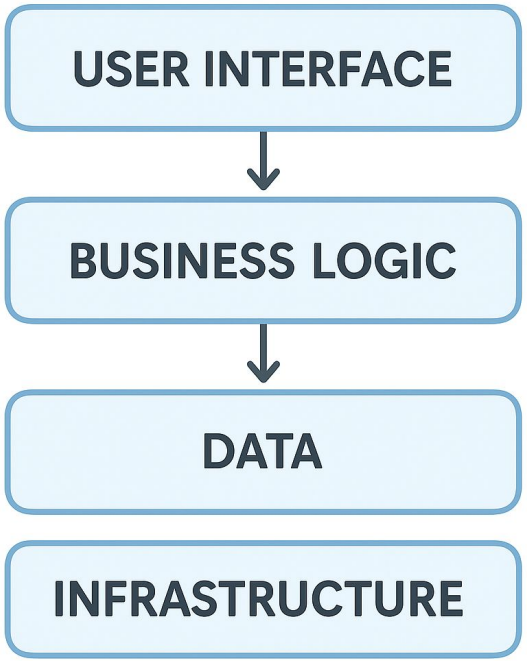
- DDD

分散システムアーキテクチャ

- Microservice Architecture
- Event-Driven Architecture

Layered Architecture

- 役割ごとに層として分離
- 層ごとに依存関係が明確に
- 小~中規模のシステムで利用される



上の層は下の層に依存し、下の層は上を知らない

Layered Architecture: クイズ1

シナリオ：レシピデータをMySQLに保存する

問題：「ライブラリを使用してSQL INSERT文を実行する」処理はどの層に実装すべきですか？

A. User Interface層（フォーム送信処理の一部として）

B. Business Logic層（レシピ作成ロジックとして）

C. Data層（データアクセス抽象化として）

D. Infrastructure層（SQL実行の技術的実装として）

Layered Architecture: クイズ2

シナリオ：レシピ検索結果を返す実装を行うが、以下のルールを適用させたい

- ユーザーがプレミアム会員の場合、お気に入りレシピを上位3件に表示
- 通常会員の場合、お気に入りレシピに「★」マークを付けるだけ
- 残りは人気順でソート

問題：この「お気に入りレシピの表示順制御」ロジックはどの層に実装すべきですか？

A. User Interface層（画面表示のレイアウト制御として）

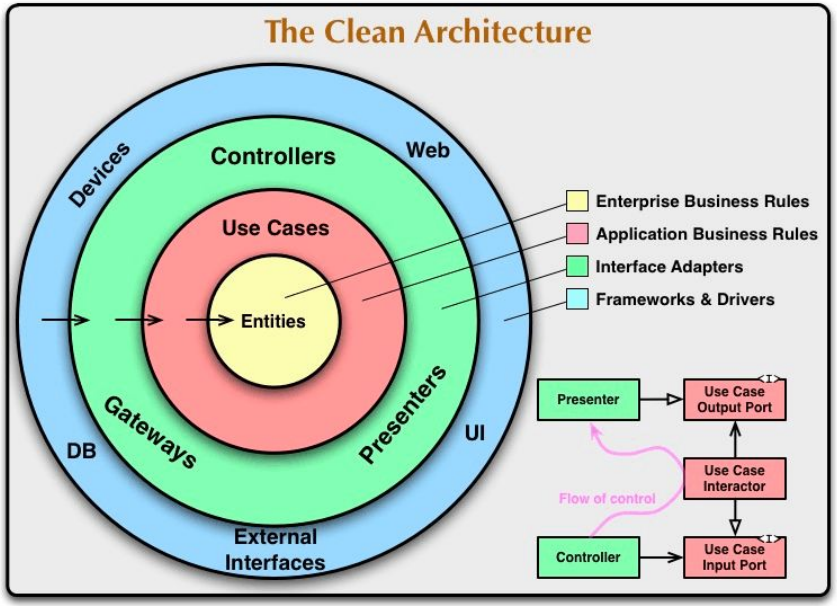
Ⓑ Business Logic層（ビジネスルールの実装として）

C. Data層（データ取得時のソート処理として）

D. Infrastructure層（キャッシュシステムの一部として）

Clean Architecture

- Entities(ビジネスロジック)を中心に設計
- それぞれの層の依存関係を外→内に明確化
- フレームワークやインフラ(DB等)に依存しない状態にすることで変更や拡張に強い



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

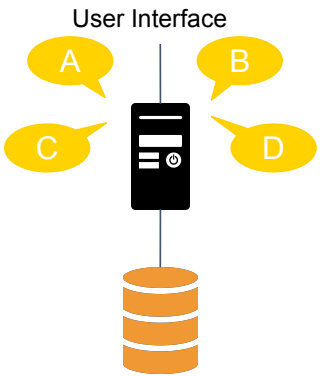
DDD

- ドメイン駆動設計(Domain-Driven Design)
- ドメインモデルに重点を置き、ビジネスロジックを中心に設計
- 複雑なビジネスロジックに対して効果的

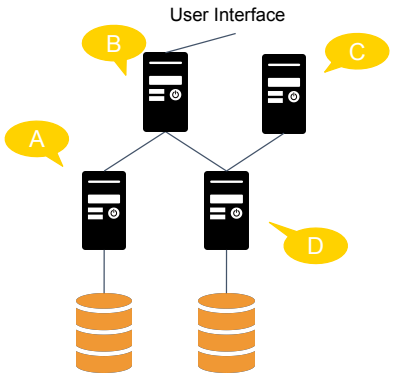
Microservice Architecture

- アプリケーションを、独自の責任範囲を持つ独立した小さな部分に分割
  - 例. 全社共有の認証機能を認証サービスとして独立させる
- 大規模システムやスケーラブルなシステムで利用
- 各サービスの管理コストが増加してしまうデメリットもある

従来の設計



マイクロサービスアーキテクチャ



F/E

アーキテクチャの分類

コンポーネント中心アーキテクチャ

Atomic Design

機能中心アーキテクチャ

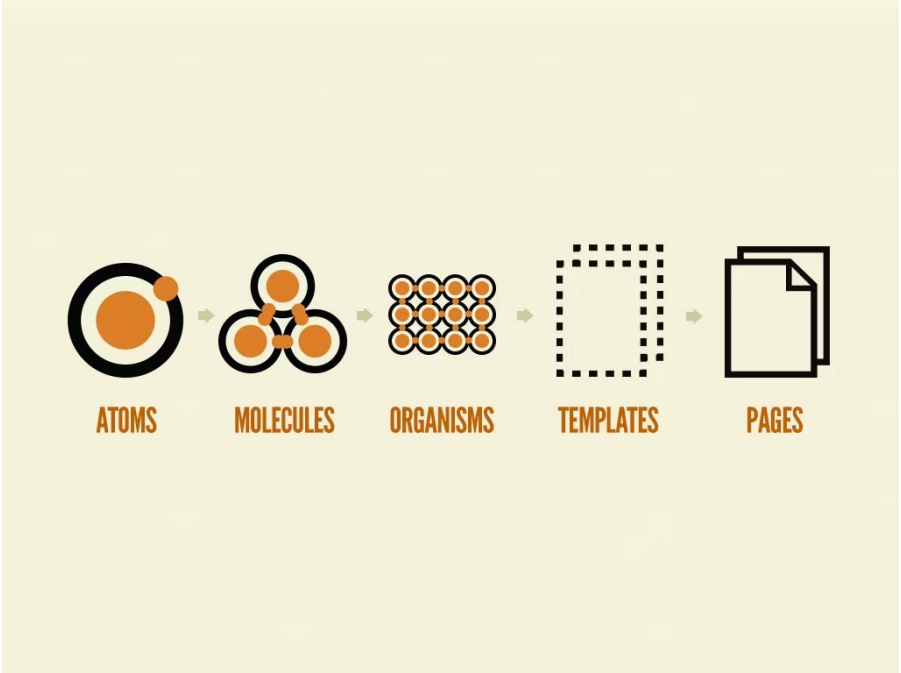
Feature-Based Architecture

層ベースのアーキテクチャ

Layered Architecture

Atomic Design

- UIコンポーネントを右図のように分割
- それぞれのコンポーネントの組み合わせで上位のコンポーネントを表現
- スタイルの共通化や再利用性が向上

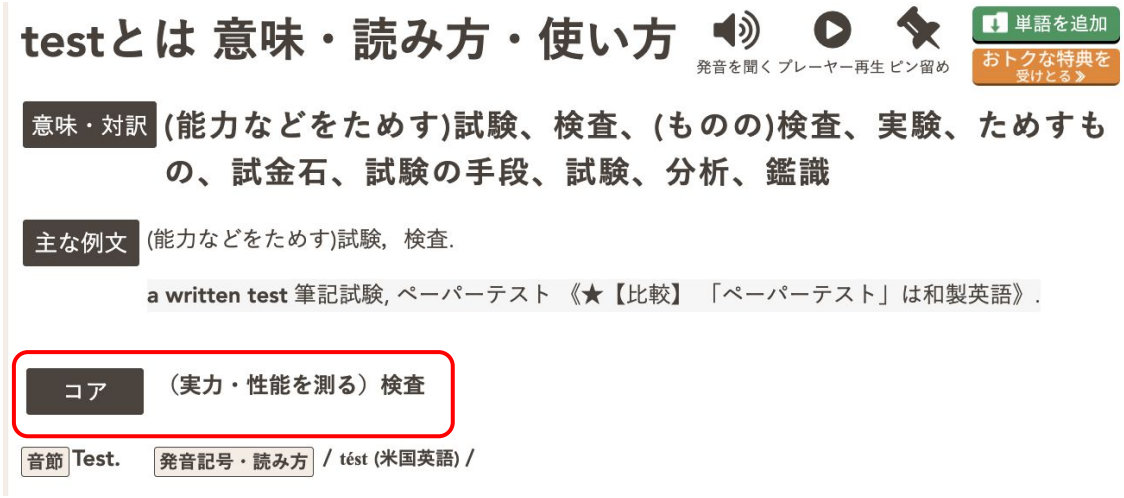


https://atomicdesign.bradfrost.com/

Atomic Design

テスト

テスト(test)とは？



https://ejje.weblio.jp/content/test#goog_rewarded

テスト(test)の本質は「（実力・性能を測る）検査」をすること

ソフトウェアエンジニアリングにおけるテスト (test)とは？

ソフトウェアテストは、**欠陥を発見し**、

ソフトウェアアーティファクトの**品質を評価する**ための一連の活動

テストでは**指定されている要件をシステムが満たすかどうか**の確認（検証）に加えて、

ユーザーやその他のステークホルダーのニーズを

運用環境でシステムが満たしていること（妥当性）を確認

JSTQB テスト技術者資格制度 Foundation Level シラバス Version 2023V4.0.J02 | 1.1 テストとは何か？ より抜粋

テストのスコープ

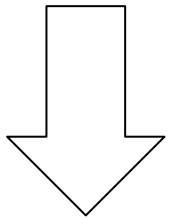
- **静的テスト**（プログラムの実行を伴わないテスト）
  - 静的解析
    - タイプミスや型エラーがないかなどを確認する
    - コードレビュー
    - etc.
- **動的テスト**（プログラムの実行を伴うテスト）
  - ユニットテスト
  - 結合テスト
  - システムテスト
  - E2Eテスト
  - 受け入れテスト

テスト手法

- **ブラックボックステスト**
  - 仕様に基づく手法。内部構造は気にせず振る舞いをテストする。
  - 利用者視点のテスト
    - 境界値分析
    - 同値分割法
    - etc.
- **ホワイトボックステスト**
  - 構造に基づく手法。内部構造や処理を流れをテストする。
  - 作成者視点のテスト
    - ステートメントテスト
    - ブランチテスト
    - etc.

テストが無いとどうなるか？

- プロダクトに混入したバグに気づけない
- 仕様・要件を十分に満たしていないことに気づけない
- etc.



- ユーザーからの不満を買ってしまう
- プロダクト・企業への不信感を募らせてしまう
- etc.

テストが無いとどうなるか？　case.1

```
func CalcDiv(operator string, a, b int) int {
    return a / b
}
```

b = 0 のとき、ゼロ除算でランタイムエラー

テストが無いとどうなるか？　case.1

```go
package tax

import "math"

// バグ: 本当は切り捨てにすべきなのに、四捨五入してしまっている
func AddTax(price int) int {
    tax := float64(price) * 0.08
    return price + int(math.Round(tax))
}
```

仕様：税額は「円未満切り捨て」
実装：四捨五入（`math.Round`）しており、1円多く請求するケースがある

（107 x 1.08 = 115.56 → 115円が実際は正しいが、上記コードでは116円になり多く請求してしまう）

テストが無いとどうなるか？　case.2

```
type (
    IUser interface {
        Update(c echo.Context) error
    }
    User struct {
        userService service.IUser
        concern     concern.ICurrentUser
    }
)
func NewUser() IUser {
    return &User{
        userService: service.NewUser(),
    }
}
func (h *User) Update(c echo.Context) error {
    currentUser, err := h.concern.Get(httputils.GetUUID(c))
    ...
```

User オブジェクトのプロパティconcern が
初期化されていない
→ panic になる

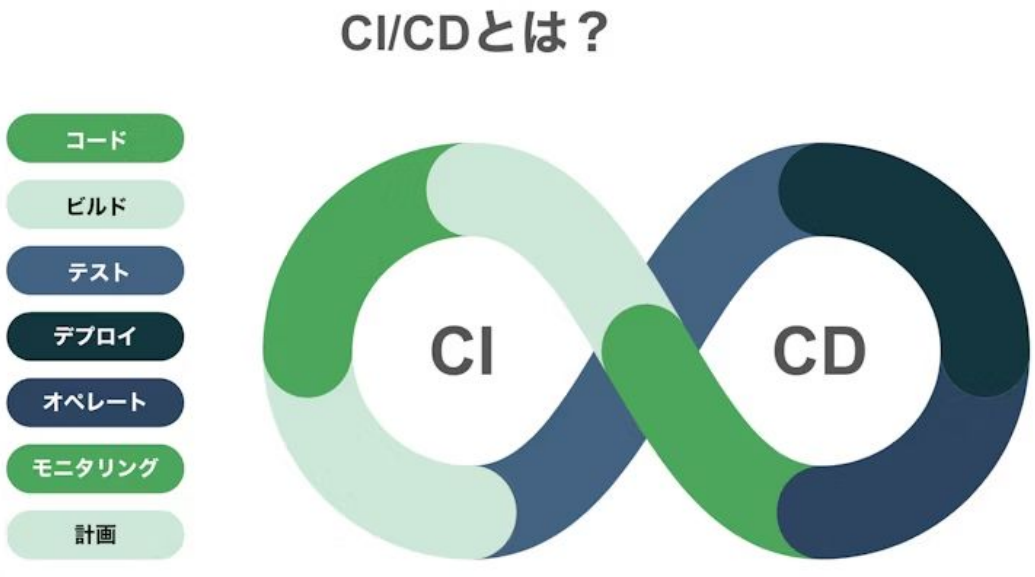
テストは大事！

- 自分のミスを防ぐため
- プログラムの振る舞いを明確にするため
- バグを早期発見するため
- etc.

CI/CD

CI/CDとは？

- CI (Continuous Integration, 継続的インテグレーション)
- CD (Continuous Delivery/Deploy, 継続的デリバリー／デプロイ)



https://circleci.com/ja/blog/what-is-ci-cd/

CI/CDの目的・意義

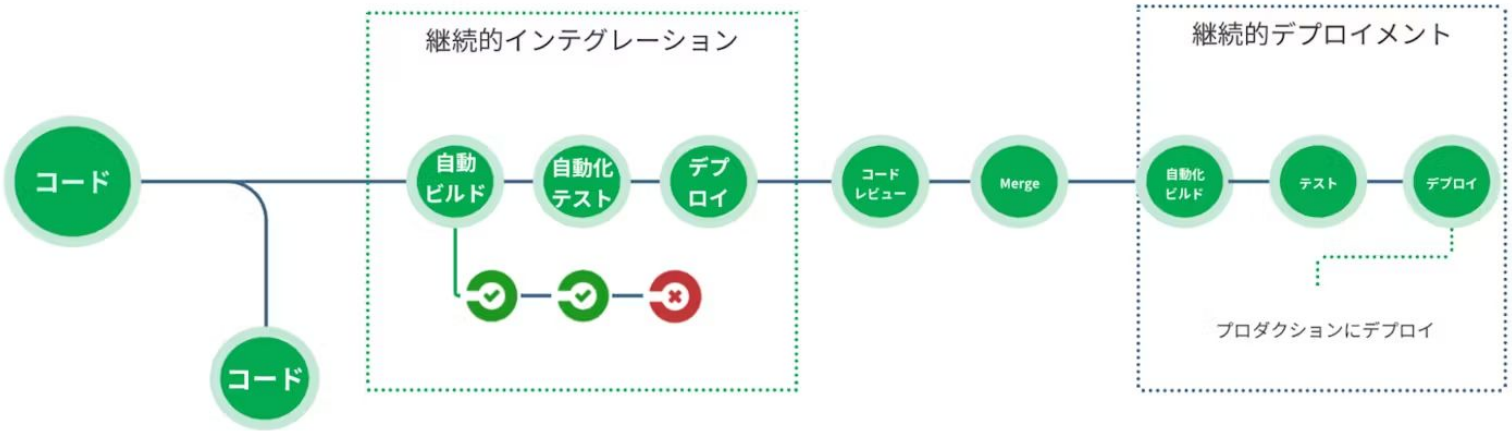
- CI (Continuous **Integration**, 継続的インテグレーション)
  - 自動で作業成果物をテストしてリポジトリに共有する
  - 開発者のコード作成に関与する
- CD (Continuous **Deploy**, 継続的デプロイ)
  - 自動で作業成果物を運用システムに反映させる
  - 完成したコードに関与する

CI/CDのメリット

- コードの共有・テスト／デプロイを自動化できる
- 開発サイクルを高速化できる
- コード品質を一定に保つことができる

CI/CDフロー

開発 → CI → レビュー → マージ → ビルド・デプロイ

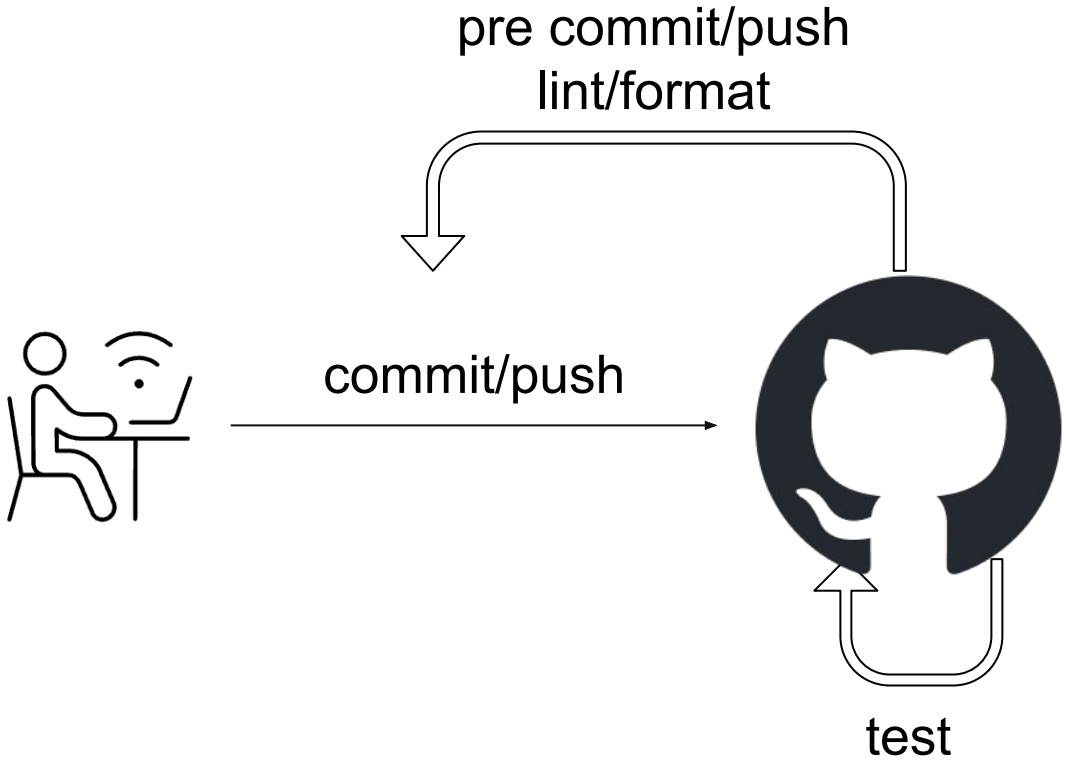


https://circleci.com/ja/blog/what-is-ci-cd/

CIで使っている解析ツール例

- lint
  - ソースコードの静的解析
- format
  - ソースコードのフォーマッティング
- test
  - ソースコードのテスト

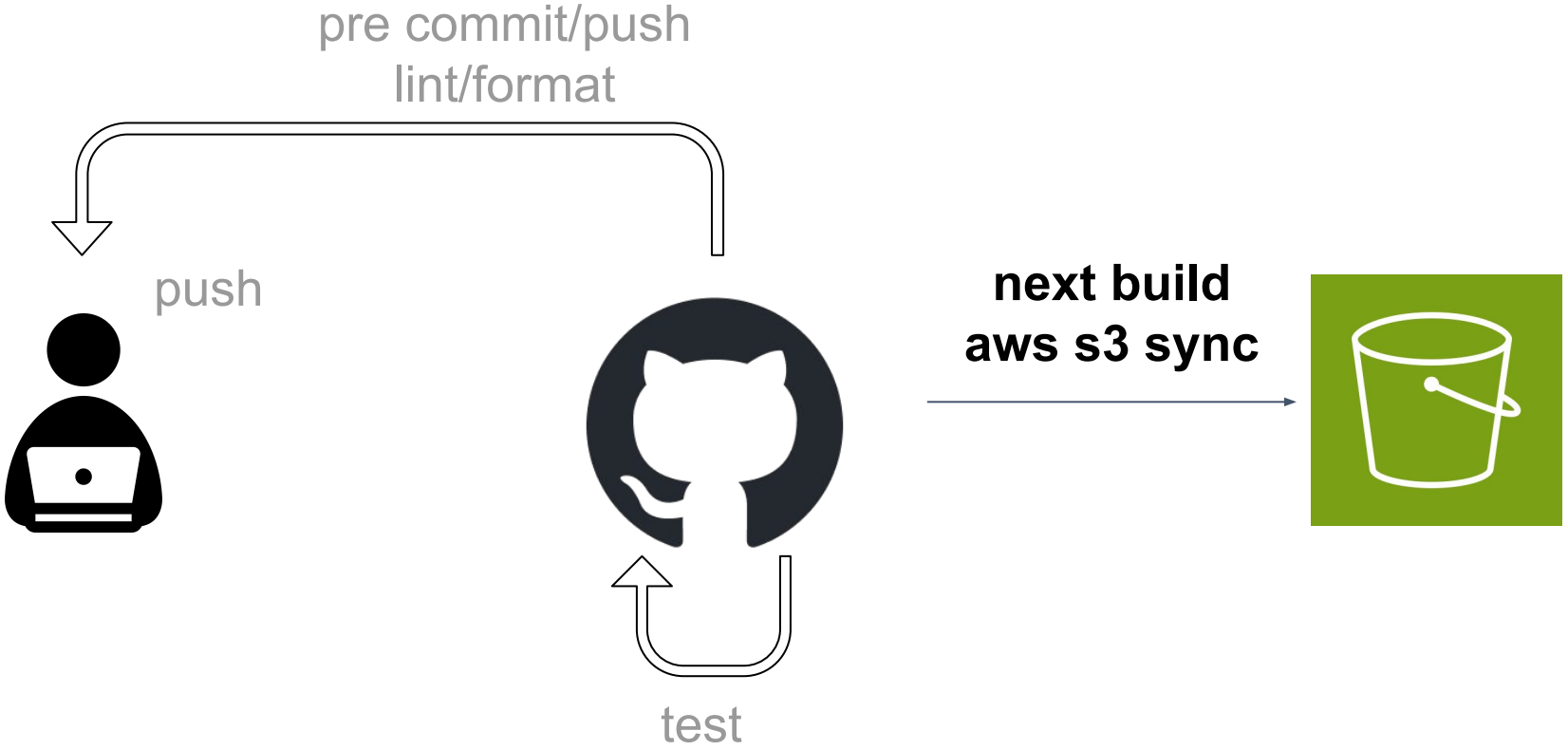
CIのユースケース〜 lint/format/test〜



CDのユースケース

- デプロイ
  - EC2のデプロイ
  - ECSサービス／タスクのデプロイ
  - Lambda Functions のデプロイ
  - フロントエンドのビルド成果物を S3 にプッシュする
  - etc.

CDのユースケース〜ビルド成果物を S3にプッシュ〜



CI/CDツールの紹介

- エブリーで利用されているもの
  - GitHub Actions
  - Circle CI
  - CodeBuild(AWSのCI/CD自動化サービス)
- その他
  - Jenkins
  - etc.

参考・引用

- 開発者向けのウェブ技術 | MDN
- レンダリングの仕組み | Vue.js
- TypeScript誕生の背景 | TypeScript入門『サバイバル TypeScript』
- Actions | GitHub
- AWS CodeBuild（継続的スケーリングによるコードのビルドとテスト） | AWS
- Jenkins
- Clean Coder Blog
- Atomic Design by Brad Frost
- CI/CD とは？開発に欠かせない理由と必要性を解説
- AWS シンプルアイコン - AWS アーキテクチャーセンター | AWS
- Dissecting layered architecture - DEV Community
- マイクロサービス アーキテクチャとは | Google Cloud
- マイクロサービスとは？ そのメリットを簡単に解説（初心者・非エンジニア向け） - NCDC株式会社