Engineer

マイクロサービス

個別にデプロイできる小さなサービスに分ける設計

どのようなものか

マイクロサービスは、一つのアプリケーションを小さなサービスに分ける設計様式です。各サービスは、別のプロセスとして動く独立したプログラムで、ほかと別に配備できます。2014年に最も知られる説明を書いたJames LewisとMartin Fowlerは、「独立して配備できるサービスの集合」としてアプリケーションを作ることと要約します。共通しがちな特徴として、HTTPのリソースAPIなど軽い仕組みでの通信、業務機能を軸とする構成、完全自動化した配備、最小限の中央管理、サービスごとにプログラミング言語と保存先を選ぶ自由を挙げます。正式な定義ではなく、その名前を持つシステムの大半に見られる特徴だとも、慎重に付け加えます。

言葉は、考え方より新しいものです。同じ記事の脚注は、2011年5月にVenice近郊のソフトウェア設計者の集まりで用語を議論し、同じ人々が2012年5月に「microservices」に決めたと記録します。NetflixのAdrian Cockcroftは、それ以前から、細粒度のSOAと呼んでいました。LewisとFowlerも、新しい設計だとは主張しません。その起源を、少なくともUnixの基礎となった原則にまで見いだしています。

通常、比較する相手はモノリスです。サーバー側のアプリケーションを、一単位としてビルドし、配備します。一つのリクエストのロジックは、すべて一プロセスで動きます。注意して作れば、全体を一人の開発者のノートPCで動かして試せます。ロードバランサーの後ろでコピーを増やせば、規模も増やせます。難しさは後から来ます。小さな変更でも全体の再ビルドと再配備が必要で、何年も良いモジュール構成を保つのは難しく、容量の必要な部分だけでは増やせず、すべてを一緒に増やすしかありません。

LewisとFowlerは、部品を、単独で置き換え、更新できるソフトウェアの単位と定義します。ライブラリーは、プログラムにリンクし、メモリー内で呼びます。サービスは、プロセス外の部品で、Webリクエストや遠隔手続き呼び出しを使

い、ネットワーク越しに到達します。結合の強さに違いが出ます。大半の言語では、他のコードがライブラリー内部に入り込むのを防ぐのは、文書と規律だけです。サービスには、明示した遠隔呼び出しでしか到達できず、インターフェースを迂回しにくくなります。ただし、一つのサービスでも、アプリケーションと専用データベースのように、複数のプロセスを含む場合があります。

「マイクロ」の意味は、見た目ほど多くありません。LewisとFowlerは、名前が大きさに注意を集め、どこまでをマイクロと呼ぶかの議論を生むと指摘します。知っていた最大のサービスは、Amazonのピザ二枚で足りるチーム、つまり多くても十二人という目安に従いました。小さい方では、六人のチームが六つのサービスを担当していました。

概念を判断に生かす

「マイクロサービスとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 データの保存先、処理のルール、他のシステムとの接続方法など、利用者の目に見えない部分をつくります。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 ETL/ELTパイプライン6本を設計・運用 — 6 3種類のDBエンジンにまたがるETLパイプライン

ETL/ELTパイプライン6本を設計・運用 お問い合わせ →

なぜ重要なのか

最初の利点は、自分たちの予定でリリースできることです。分割すれば、多くの変更では一サービスだけを再配備でき、他チームを待たずにリリースできます。LewisとFowlerは、絶対ではないと明記します。サービスのインターフェースを変えるなら、呼び出すサービスとの調整は、依然必要です。

二つ目は、ソフトウェアを、組織に逆らわせず、組織に沿わせられることです。Melvin Conwayは1968年のDatamationで、システムを設計する組織は、自身の伝達構造を写した設計しか作れないと論じました。LewisとFowlerは、大きいアプリケーションを、UI、サーバー側ロジック、データベースのチームという技術層で分ける、通常の方法に当てはめます。単純な変更さえ複数チームの仕事になり、ロジックは、チームが手を入れられる場所に落ち着きます。マイクロサービスは逆にします。注文、在庫、支払いなど、一つの業務機能を、UIから保存先まで一サービスが扱い、その縦断部分に必要な技能を持つ、機能横断のチームが所有します。プロジェクト終了時に引き渡さず、存続期間全体に責任を持つべきです。Amazonの規則では、作った人が、本番でも運用します。

三つ目は、ロジックをサービスに残すことです。LewisとFowlerは、賢いエンドポイントと単純な経路と要約します。サービスは、リソースAPIへのHTTPリクエストか、配送以外は何もしない軽いメッセージバスで話し、判断は両端のサービスで行います。配送、変換、業務規則の適用を、ネットワークの両端ではなく中央で行う、エンタープライズサービスバスから、意図的に離れる選択です。

データも同じ線で分けます。各サービスが自分のデータベースを管理し、適すれば、違うデータベースシステムを選べます。Eric Evansのドメイン駆動設計が根拠を示します。大きいシステムには、複数のモデルがあり、それぞれ、チームの組織、担当するアプリケーションの部分、コードベースやデータベーススキーマという物理的なものまで含め、境界を明示したコンテキストが必要です。販売側の顧客像と、サポート側の顧客像は、一つのテーブルに揃えなくても、違っていて構いません。

代償も定義の一部で、LewisとFowlerは、はっきり述べます。遠隔呼び出しは、メモリー内の呼び出しよりはるかにコストが高いため、インターフェースは、より粗い単位にする必要があります。サービスの境界をまたいでコードを移すのは難しく、インターフェースの変更を調整し、後方互換性の層を加え、テストも複雑になります。分散トランザクションは実装しにくいため、使わずに調整し、整合性が最終的にしか得られない可能性を受け入れ、補償操作で間違いを直します。どの呼び出しも失敗し得ます。きれいにつながらなければ、複雑さは消えず、見えにくい接続部分へ移っただけです。LewisとFowlerは、まずモノリスを作り、明確なモジュールにし、問題になってから分けるという、妥当と考える意見も紹介します。ただし、プロセス内で良いインターフェースが、サービス間でも良いことは少ないと付け加えます。

実際の活用例

最後のチケットを、ちょうど一回だけ売るノートは、一つのデータベースでは足りなくなってから、サービスを使います。その際、設計を保つ規則を示します。各商品の在庫の所有者は、在庫サービス一つだけで、ほかは判断しません。そのサービスは、テナントと商品で分割し、最大のテナントには、専用のセルを与えます。同期する必要があるのは、在庫の確保だけです。支払い、注文、領収書は別のシステムで、またぐトランザクションはありません。二相コミットは、一参加者が遅くても止まり、決済提供者も参加しないので拒否します。代わりに、一連のローカルな手順と、各手順を取り消す手順からなるサガを使います。

各手順は、知らせる変更と同じトランザクションで、メッセージをアウトボックスのテーブルに書きます。PostgreSQLでは、`WITH` 内のデータ変更文から、結果を挿入に渡せるため、在庫サービスは、一つの文で一単位を確保し、メッセージ

も記録できます。

```
-- 在庫サービスと、その専用データベース
WITH held AS (
    UPDATE inventory
    SET available = available - 1
    WHERE tenant_id = 't-7' AND item_id = 'item-881'
      AND available >= 1
    RETURNING item_id
)
INSERT INTO outbox (order_id, item_id, event, units)
SELECT 'A-17', item_id, 'StockReserved', 1 FROM held;
-- 売り切れなら一致する行がなく、メッセージも書かれない
```

変更データキャプチャーが、後からアウトボックスの行を公開します。メッセージが少なくとも一回届くため、各受信側は、注文をキーとする冪等な処理にします。この設計例では、`inventory` テーブルに触るのは在庫サービスだけです。他は、メッセージか、在庫サービスへの問い合わせで在庫を知ります。

同じノートは、分割を仕事のどの順に行うかも示します。先に行う一覧では、在庫に専用のサービスと所有者を置くのは四番目です。計測、書き込みを一つの条件付き文にすること、待合室、ゲート、売り切れのキャッシュ応答で負荷を落とすことの後です。安い修正を先にします。

明確な層で作った海事プラットフォームの事例は、NextMarinerにおける創業者の設計上の貢献を記録し、選択のもう一方を示します。要件が大きく変わる若い製品で、業務規則の変更のたびに、フロントエンド、API、データベースを同時に触らないことを目指しました。各層の仕事を一つにした三層です。表示用のNext.jsフロントエンド、ルーティング、検証、セキュリティー上の絞り込み、処理の調整を行う、意図的に薄いGo API、業務ロジックをすべて置くPostgreSQL関数です。境界を文書にし、レビューで守りました。LewisとFowlerは、一つのアプリケーションのモジュール境界を保つには、多くの規律が必要で、サービスの明示的な分離の方が、チームの境界を明確に保ちやすいと述べます。このプラットフォームは、業務ロジックを別々のサービスに分けずに、その規律を保ちました。

問題が起きるところ

同期呼び出しの連鎖が、停止の影響を掛け合わせます。 一つのリクエストへの応答で、最初のサービスが二つ目を、それが三つ目を呼ぶなら、三つとも動く間だけ成功し、一つずつ順に待ちます。LewisとFowlerは、停止時間の乗算効果と

呼び、Guardianの、利用者リクエスト一回につき同期呼び出し一回という規則を挙げます。ノートのように、待つのは確保だけ、その後はすべてメッセージとして、同期経路を短く保ちます。残る各呼び出しには、タイムアウトとサーキットブレーカーを置きます。LewisとFowlerがRelease It!で挙げるパターンで、壊れたサービスを、待ち続けずに切り離します。

常に一緒に変わるサービスです。 不適切な境界で分けると、すべての機能が三サービスに触れ、歩調を合わせてリリースするしかない、単独ではリリースできないサービスになります。LewisとFowlerの判定は、二サービスをいつも同時に変えるなら、統合すべきだというものです。関連する失敗は、メソッド呼び出しを一対一で遠隔呼び出しに変えることで、ネットワーク上で頻繁に話し、遅くなります。業務機能と境界のあるコンテキストで分け、インターフェースの単位を粗くし、呼び出し先の変更に耐えるよう設計します。提供側が成長しても、受け手を壊さずに済みます。LewisとFowlerは、インターフェースのバージョンを分けることを最後の手段とします。

二サービスが、一つのデータベースを共有します。 注文サービスが在庫テーブルを直接読み書きすれば、スキーマが、誰も所有しないインターフェースになります。どちらのチームも単独で変えられず、在庫には、在庫サービスの検査に見えない二つ目の書き手が生まれます。ノートの在庫のように、各データに一所有者を設け、他のサービスには、問い合わせかメッセージを使わせます。更新がサービスをまたぐなら、分散トランザクションではなく、補償手順付きのサガを使います。

誰もリクエストを追えません。 複数サービスを通るリクエストは、経緯の断片を、それぞれのログに残します。LewisとFowlerは、リアルタイムの監視を中心に置き、データベースへの毎秒のリクエストといった技術の数値と、毎分の注文という業務の数値を並べて見ます。互いのイベントに反応するサービスが生む創発的な動作も、早く見つける必要があると警告します。すべてのリクエストをトレースします。OpenTelemetryでは、一つのリクエストの経路を、同じトレースIDを持ち、親を指すスパンから作ります。コンテキストの伝播で、違うサービスのスパンを一つのトレースにつなげます。呼び出しとメッセージのすべてに、アウトボックス経由も含め、トレースのコンテキストを渡します。なければ、最初の非同期の受け渡しでトレースが途切れます。

参考資料

- マイクロサービス — James LewisとMartin Fowlerの2014年の記事です。九つの特徴、名前の歴史、大きさと同期呼び出しの補足、慎重さの理由です。

- 委員会はどう発明するか — Datamationに載ったMelvin Conwayの1968年の論文です。設計した組織の伝達構造を、システムの構造が写します。
- ドメイン駆動設計参照資料 — 境界のあるコンテキストと、それらの関係の地図を含む、Eric Evansのパターンの要約です。
- OpenTelemetryのトレース — スパン、トレースID、親スパン、コンテキストの伝播という、サービスをまたぐ一リクエストを見えるようにする部品です。
- PostgreSQLのWITHクエリー — 一つの文で行を変え、別の場所に記録できる、`WITH` 内のデータ変更文です。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「マイクロサービスとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/microservices/