AWS サーバーレス多層アーキテクチャ

Amazon API Gateway と AWS Lambda の使用

2015 年 11 月

注意

本書は情報提供のみを目的としています。本書の発行時点における AWS の現行製品と慣行を表したものであり、それらは予告なく変更されることがあります。お客様は本書の情報、および AWS 製品またはサービスの利用について、独自の評価に基づき判断する責任を負います。いずれの AWS 製品またはサービスも、明示または黙示を問わずいかなる保証も伴うことなく、「現状のまま」提供されます。本書のいかなる内容も、AWS、その関係者、サプライヤ、またはライセンサーからの保証、表明、契約的責任、条件や確約を意味するものではありません。お客様に対する AWS の責任は、AWS 契約により規定されます。本書は、AWS とお客様の間で行われるいかなる契約の一部でもなく、そのような契約の内容を変更するものでもありません。

目次

要約

このホワイトペーパーでは、アマゾン ウェブ サービス (AWS) のイノベーションによって、マイクロサービス、モバイルバックエンド、公開ウェブサイトなどのポピュラーなパターンに関する多層アーキテクチャの設計がどのように変わるかを説明します。アーキテクトと開発者は、Amazon API Gateway や AWS Lambda などの実装パターンを使用できるようになりました。これにより、多層アプリケーションの作成と運用管理を行うために必要な開発サイクルおよび運用サイクルを短縮することができます。

はじめに

多層アプリケーション (3 層、n 層など) は何十年もの間、基礎となるアーキテクチャパターンでした。多層パターンは、アプリケーションコンポーネントを分離および拡張でき、(多くは別々のチームによって) 別々に管理および保守できるようにするためのガイドラインとして適しています。多層アプリケーションは、ウェブサービスを使用するためのサービス指向アーキテクチャ (SOA) アプローチで構築されることが少なくありません。このアプローチでは、ネットワークが各層の間の境界になります。ただし、アプリケーションの一部としての新しいウェブサービス層の作成には、未分化要素が多数あります。多層ウェブアプリケーション内に記述されたコードの大半は、パターン自体の直接的な結果です。1 つの層を別の層に統合するコード、層と層の間をやりとりするために各層で使用される API とデータモデルを定義したコード、層と層の統合ポイントが望ましくない形で公開されないようにするためのセキュリティ関連コードなどはその例です。

Amazon API Gateway[1] は、API を作成および管理するためのサービスであり、AWS Lambda[2] は、任意のコード関数を実行するためのサービスです。これらを一緒に使用することで、堅牢な多層アプリケーションを簡単に作成することができます。

Amazon API Gateway を AWS Lambda と統合することで、ユーザー定義の HTTPS リクエストを通じてユーザー定義のコード関数を直接トリガーすることができます。要求されたリクエストのボリュームに関係なく、API Gateway も Lambda も、アプリケーションのニーズを正確にサポートできるように、自動的にスケーリングされます。これらの組み合わせにより、アプリケーション用の層を作成し、アプリケーションにとって重要なコードを記述することができます。この層では、多層アーキテクチャの実装に関する、他のさまざまな未分化要素 (高可用性のための設計、クライアント SDK の記述、サーバー/オペレーティング システム (OS) 管理、スケーリング、クライアント認証メカニズムの実装など) は重要視されません。

最近、AWS は、お客様の Amazon Virtual Private Cloud (Amazon VPC)[3] 内で実行される Lambda 関数を作成できることを発表しました。この機能では API Gateway と Lambda の組み合わせによるメリットが拡張され、ネットワークプライバシーが要求されるさまざまなユースケースに対応できるようになりました。たとえば、ウェブサービスに、機密情報が格納されるリレーショナルデータベースを統合する必要があるような場合です。Lambda と Amazon VPC の統合により、間接的に Amazon API Gateway の機能も拡張されました。Amazon VPC の一部としてプライベートかつ安全に保たれるバックエンドの前面で、インターネットアクセスが可能な HTTPS API を開発者が独自に定義することができるのです。この強力なパターンのメリットは、多層アーキテクチャの各層にわたって

見ることができます。このホワイトペーパーでは、多層アーキテクチャの最もポピュラーな例である **3 層**ウェブアプリケーションに焦点を合わせています。ただし、この多層パターンは、典型的な 3 層ウェブアプリケーションを超えて適用することができます。

3 層アーキテクチャの概要

3 層アーキテクチャは、ユーザー向けアプリケーションでポピュラーなパターンです。このアーキテクチャを構成する 3 層は、**プレゼンテーション層**、**ロジック層**、および**データ層**です。プレゼンテーション層は、ユーザーが直接対話するコンポーネント (ウェブページ、モバイルアプリ UI など) を表します。ロジック層には、プレゼンテーション層におけるユーザーの操作を解釈し、アプリケーションの動作を駆動する機能に変換するために必要なコードが含まれます。データ層は、アプリケーションに関連するデータを保持するストレージメディア (データベース、オブジェクトストア、キャッシュ、ファイルシステムなど) から成っています。図 1 は、単純な 3 層アプリケーションの例を示しています。

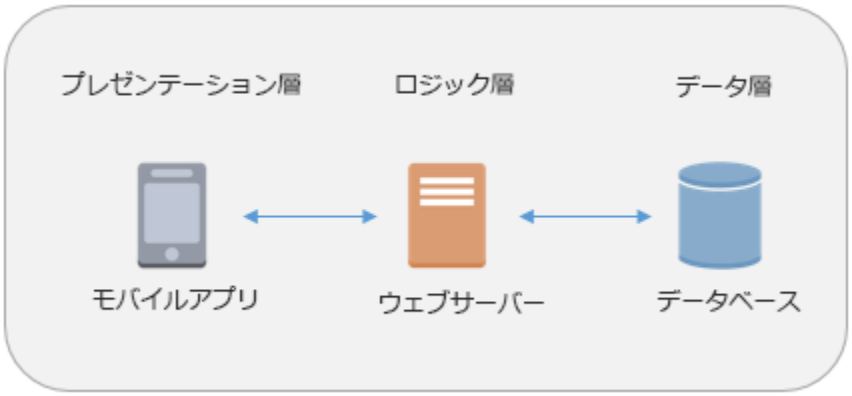

図 1: 単純な 3 層アプリケーション用のアーキテクチャパターン

一般的な 3 層アーキテクチャパターンの詳細について学ぶことができる多数の優れたリソースがオンラインで公開されています。このホワイトペーパーでは、Amazon API Gateway と AWS Lambda を使用する、このアーキテクチャの特定実装パターンに焦点を合わせています。

サーバーレスロジック層

3 層アーキテクチャのロジック層は、アプリケーションの頭脳を表します。このため、Amazon API Gateway と AWS Lambda を統合してロジック層を形成することは、非常に画期的です。これら 2 つのサービスの機能により、可用性、拡張性、安全性に優れた本稼働アプリケーションをサーバーレスで構築することができます。アプリケーションは何千ものサーバーを使用することもできますが、このパターンを活用することで、サーバーの管理が一切不要になります。さらに、これらのマネージドサービスを一緒に使用することで、次のようなメリットが生まれます。

- オペレーティング システムの選定、保護、保守、管理が必要ありません。
- サーバーのサイズ調整、監視、スケールアウトが必要ありません。
- オーバープロビジョニングによるコストのリスクがありません。
- アンダープロビジョニングによるパフォーマンスのリスクがありません。

さらに、各サービス内には、多層アーキテクチャパターンにプラスになる機能があります。

Amazon API Gateway

Amazon API Gateway は、API を定義、展開、および保守するための、全面的に管理されたサービスです。クライアントと API との統合には、標準的な HTTPS リクエストを使用できます。サービス指向の多層アーキテクチャに対する適用性は明白です。それだけではなく、ロジック層での大きな強みとなる機能と性質もあります。

AWS Lambda との統合

Amazon API Gateway を使用することで、アプリケーションは、単純な方法 (HTTPS リクエスト) で AWS Lambda のイノベーションを直接的に活用することができます。API Gateway は、AWS Lambda で作成した関数とプレゼンテーション層をつなぐブリッジになります。API を使用してクライアントとサーバーの関係を定義した後、クライアントの HTTPS リクエストの内容が Lambda 関数に渡され、実行されます。この内容には、リクエストのメタデータ、リクエストヘッダー、リクエスト本文が含まれています。

全世界で安定している API パフォーマンス

Amazon API Gateway の各展開には、Amazon CloudFront[4] の配布も含まれています。Amazon CloudFront は、開発した API に統合するクライアントの接続ポイントとして、Amazon が提供するエッジロケーションのグローバルネットワークを使用したコンテンツ配信ウェブサービスです。これは、API の合計応答時間レイテンシーの短縮に役立ちます。Amazon CloudFront を使用すると、世界中にある複数エッジロケーションの使用を通じて、分散サービス妨害 (DDoS) 攻撃への対処も可能になります。詳細については、ホワイトペーパー『AWS Best Practices for Combatting DDoS Attacks (DDoS 攻撃に対処するための AWS ベストプラクティス)[5]』をお読みください。

Amazon API Gateway を使用してオプションのインメモリキャッシュに応答を格納すると、特定の API リクエストのパフォーマンスを高めることができます。この方法では、繰り返し発生する API リクエストのパフォーマンスにメリットがあるだけではなく、バックエンドの実行回数を減らすことができるため、全体的なコストを節約できる可能性もあります。

イノベーションの促進

新しいアプリケーションの構築に必要な開発作業は、投資です。プロジェクトを開始するには、そのことに正当な根拠が必要になります。開発のタスクと時間に必要な投資金額を縮小すると、もっと自由に試行錯誤し、新しいことを取り入れる余裕が生まれます。

多層ウェブサービスをベースにした多くのアプリケーションの場合、プレゼンテーション層は容易に多数のユーザー用 (別々のモバイル デバイス、ウェブブラウザなど) に断片化されます。また、これらのユーザーは地理的な結びつきもありません。ただし、分離型のロジック層は、ユーザーによって物理的に断片化されるわけではありません。ユーザーはすべて、ロジック層を使用する同じインフラストラクチャに依存するため、インフラストラクチャの重要性が大きくなります。最初にロジック層を実装する場合の手抜き ("運用当初に数値の測定は必要ありません" "初期使用率は低いため、スケーリング方法は後で考えましょう" など) が、新しいアプリケーションを早く提供するためのメカニズムとして提案されることが少なくありません。本稼働中のアプリケーションに変更を展開する必要が生じると、このようなことが技術的な負債や運用上のリスクになる場合があります。Amazon API Gateway を使用する場合は、すでにサービスが実装されているため、コストを削減し、早く提供を開始することができます。

アプリケーションの寿命は全体的に、不明であるか短命であるとわかっているかのどちらかです。このため、新しい多層アプリケーションの検討用資料を作成することは難しい場合があります。Amazon API Gateway から提供される管理された機能がスタート地点で既に含まれていれば、その作業が容易になります。また、インフラストラクチャコストがかかるのは、開発した API がリクエストを受け取り始めてからです。詳細については、Amazon API Gateway の料金を参照してください。[6]

迅速に反復して機敏性を維持

新しいアプリケーションでは、ユーザーベースの定義 (サイズ、利用パターンなど) が十分でない場合があります。このため、ユーザーベースが形になっても、ロジック層では機敏性を維持する必要があります。アプリケーションとビジネスは、早期導入されたお客様からの変わり続ける要望に対応し、変化する必要があります。Amazon API Gateway を使用すると、API を開始から展開に進めるために必要な開発サイクル数を減らすことができます。Amazon API Gateway では、Mock 統合[7] を作成できるため、API 応答を直接 API Gateway から生成できます。この API Gateway に対してバックエンドロジック全体を開発しながら、並行してクライアントアプリケーションを開発することができます。この利点は API の最初の開発時だけでなく、ビジネス上の決定としてユーザーに応じてアプリケーション (および既存の API) を迅速に方向転換することになった場合にも当てはまります。API Gateway と AWS Lambda ではバージョニングが可能であるため、独立した API/関数バージョンとして新しい機能をリリースする場合も、既存の機能やクライアント依存関係はそのままにしておくことができます。

セキュリティ

3 層で構成される公開ウェブアプリケーションのロジック層をウェブサービスとして実装すると、すぐにセキュリティのトピックが浮上します。アプリケーションでは、許可されたクライアントのみが (ネットワークで公開される) ロジック層にアクセスできるようにする必要があります。Amazon API Gateway では、バックエンドの安全性を確信できる方法を通じてセキュリティのトピックに取り組んでいます。アクセス制御については、クライアントアプリケーションに静的な API キー文字列を提供する方法に頼らないでください。クライアントによって抽出され、別の場所で使用される可能性があります。Amazon API Gateway によってロジック層を保護できる方法がいくつかあるため、これらを利用することができます。

- 転送中の暗号化を可能にするために、API へのすべてのリクエストには HTTPS を使用することができます。
- 信頼関係が Amazon API Gateway 内の特定 API と AWS Lambda 内の特定関数の間のみに限定されるように、AWS Lambda 関数はアクセスを制限することができます。Lambda 関数の起動では、その関数を公開している API を使用する以外に方法はありません。
- Amazon API Gateway を使用すると、API に統合するためのクライアント SDK を生成することができます。この SDK では、API で認証が求められる場合にリクエストの署名も管理します。クライアント側で認証に使用される API 認証情報は、直接 AWS Lambda 関数に渡されるため、作成した独自コード内で必要に応じてさらに認証を行うことができます。

- API の一部として作成した各リソース/メソッドの組み合わせには、独自の特定 Amazon Resource Name (ARN) が付与されます。この ARN は、AWS Identity and Access Management (IAM)[8] ポリシー内で参照することができます。
  - これは、他の AWS 所有の API と同様に、独自の API も最重要視されることを意味します。IAM ポリシーはきめ細かく指定することができます。IAM ポリシーでは、Amazon API Gateway を使用して作成した API の特定のリソース/メソッドを個別に参照できます。
  - アプリケーションコードのコンテキスト外で作成した IAM ポリシーでは、API アクセスが強制されます。つまり、このようなアクセスレベルを認識または強制するためのコードを記述する必要はありません。コードが存在しないので、コードに含まれるバグや悪用を心配する必要もありません。
  - AWS 署名バージョン 4 (SigV4)[9] を使用した認証と IAM ポリシーで API アクセスのためのクライアント認証を行うと、必要に応じて、同じ認証情報で他の AWS サービスやリソース (Amazon S3 バケットや Amazon DynamoDB テーブルなど) についてもアクセスを制限または許可できます。

AWS Lambda

AWS Lambda は、本質的には、サポートされている任意の言語 (2015 年 11 月の時点では Node、JVM ベース、Python) で記述された任意のコードに対し、イベントへの応答としてトリガーされることを許可しています。このイベントには、AWS によって利用可能になる、プログラムによるトリガーの 1 つを使用できます。これは、**イベントソース**と呼ばれるものです (現在サポートされているイベントソースについては、こちらを参照してください[10])。AWS Lambda に関する多くのポピュラーなユースケースは、イベント駆動型データ処理ワークフローに従って展開されます。例としては、Amazon Simple Storage Service (Amazon S3)[11] に格納されているファイルの処理や、Amazon Kinesis[12] からのデータレコードのストリーミングなどがあります。

Amazon API Gateway との組み合わせで使用する場合、AWS Lambda 関数は、典型的なウェブサービスのコンテキストに存在することができ、HTTPS リクエストによって直接トリガーすることができます。Amazon API Gateway には、ロジック層の正面玄関のような役割がありますが、ロジックはこれらの API の背後で実行する必要があります。ここで AWS Lambda が必要になります。

ビジネスロジックの適用

AWS Lambda では、イベントによるトリガー時に実行される、**ハンドラー**と呼ばれるコード関数を記述できます。たとえば、API への HTTPS リクエストなどのイベントが発生したときにトリガーされるハンドラーを記述することができます。Lambda では、必要な粒度レベル (1 つは API 単位、1 つは API メソッド単位、など) でモジュール式のハンドラーを作成し、個々に更新、呼び出し、変更を行うことができます。ハンドラーは、自身が持つ他の依存関係 (独自のコードでアップロードした他の関数、ライブラリ、ネイティブバイナリ、外部のウェブ

サービスなど) にも自由に到達できます。Lambda では、必要な依存関係をすべて、関数定義に (作成時に) パッケージ化できます。関数を作成する場合は、展開パッケージ内のどのメソッドをハンドラーとして使用するかを指定します。同じ展開パッケージを複数の Lambda 関数定義に再利用することもできます。この場合、同じ展開パッケージ内の各 Lambda 関数に、それぞれ異なるハンドラーを指定することもできます。サーバーレス多層アーキテクチャパターンでは、Amazon API Gateway で作成した個々の API が、必要なビジネスロジックを実行する Lambda 関数 (および対応するハンドラー) に統合されます。

Amazon VPC との統合

ロジック層の中核にある AWS Lambda は、データ層と直接統合されるコンポーネントになります。データ層にはビジネスまたはユーザーの機密情報が含まれていることがあるため、データ層は厳重に保護する必要があります。Lambda 関数から統合できる AWS サービスの場合は、IAM ポリシーを使用してアクセス制御を管理できます。これに該当するサービスは、Amazon S3、Amazon DynamoDB、Amazon Kinesis、Amazon Simple Queue Service (Amazon SQS)、Amazon Simple Notification Service (Amazon SNS)、その他の AWS Lambda 関数などです。ただし、リレーショナルデータベースなど、独自のアクセス制御が使用されるコンポーネントもあります。このようなコンポーネントの場合は、プライベートなネットワーキング環境である Amazon Virtual Private Cloud (Amazon VPC)[13] に展開することで、より高いセキュリティを確保できます。

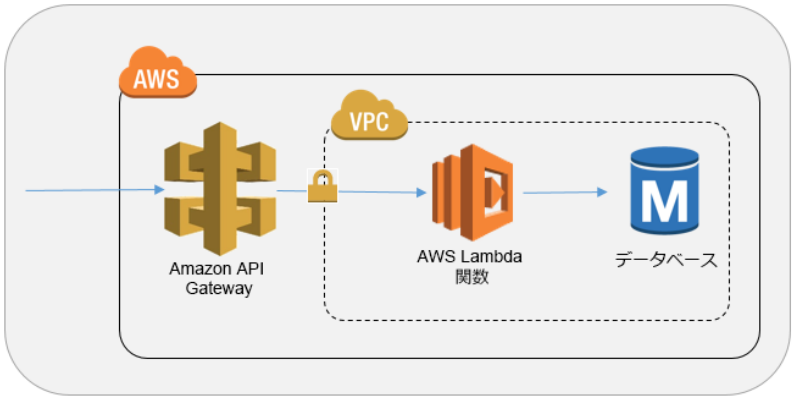

図 2: VPC を使用するアーキテクチャパターン

VPC を使用するということは、ビジネスロジックが依存しているデータベースやその他のストレージメディアに対して、インターネット経由のアクセスを不可能にできるということです。また、VPC を使用すると、インターネットでデータを操作する方法は、独自に定義した API と独自に作成した Lambda コード関数を経由する以外にありません。

セキュリティ

Lambda 関数を実行するには、イベントまたはサービスによってトリガーする必要があり、その動作が IAM ポリシーで許可されている必要があります。独自に定義した API Gateway リクエストから呼び出されない限り、一切実行できない Lambda 関数を作成することもできます。作成したコードは、作成した API で定義される、有効なユースケースの一部として処理されます。

各 Lambda 関数には IAM ロール (IAM 信頼関係を通じて付与される権限) が設定されます。この IAM ロールでは、Lambda 関数が操作できる他の AWS サービス/リソース (Amazon DynamoDB テーブルや Amazon S3 バケットなど) が定義されます。関数からアクセスできるサービスは、関数自体とは別の場所で定義および制御されます。小さなことですが、効果は強力です。これにより、作成し

たコードで AWS 認証情報を格納または取得する必要がなくなります。つまり、API キーをハードコーディングする必要も、API キーを取得してメモリ内に格納するためのコードを記述する必要もないということです。Lambda 関数から、IAM ロールで呼び出しが許可されているサービスへの呼び出しを可能にするかどうかは、サービス自体で管理されます。

データ層

ロジック層として AWS Lambda を使用する場合は、多数のデータストレージオプションから選んでデータ層に使用できます。これらのオプションは、2 つの大きなカテゴリとして、Amazon VPC でホストされているデータストアと IAM 対応のデータストアに分けることができます。AWS Lambda は、どちらにも安全に統合することができます。

Amazon VPC でホストされているデータストア

AWS Lambda を Amazon VPC と統合すると、プライベートかつ安全にさまざまなデータストレージテクノロジと統合するための機能が有効になります。

- Amazon RDS[14]

  Amazon Relational Database Service (Amazon RDS) によって利用可能になるいずれのエンジンを使用することもできます。Amazon RDS には、Lambda で記述したコードから直接接続できます。この点は Lambda 以外での操作と同様ですが、データベース認証情報の暗号化のために AWS Key Management Service (AWS KMS) と簡単に統合できるというメリットがある点は異なります。

- Amazon ElastiCache[15]

  管理されたインメモリキャッシュと Lambda 関数を統合すると、アプリケーションのパフォーマンスを引き上げることができます。

- Amazon RedShift[16]

  エンタープライズデータウェアハウスへの照会を安全に行うための関数を作成できます。レポートまたはダッシュボードを構築する場合や、アドホッククエリの結果を取得する場合に利用できます。

- Amazon Elastic Compute Cloud (Amazon EC2)[17] でホストされるプライベートウェブサービス

  VPC 内のプライベートな環境でウェブサービスを実行する既存のアプリケーションがある場合は、Lambda 関数から、論理的にプライベートな VPC ネットワークで HTTP リクエストを送信できます。

IAM 対応のデータストア

AWS Lambda は IAM と統合されているため、IAM を使用すると、任意の AWS サービスを安全に統合し、AWS API を使用して直接活用することができます。

- Amazon DynamoDB[18]

  Amazon DynamoDB は、無限に拡張できる AWS の NoSQL データベースです。規模に関係なく数ミリ秒のパフォーマンスでデータレコード (このホワイトペーパーの作成時点では 400 KB 以下) を取得するには、Amazon DynamoDB の使用をご検討ください。Amazon DynamoDB のきめ細かなアクセス制御を使用すると、DynamoDB 内の特定データを照会する際に、最小権限の使用というベストプラクティスを Lambda 関数に適用できます。

- Amazon S3[19]

  Amazon Simple Storage Service (Amazon S3) を利用すると、インターネット規模のオブジェクトストレージを使用できるようになります。Amazon S3 は、オブジェクトに対する 99.999999999% の耐久性を実現できるように設計されています。安くて耐久性の高いストレージがアプリケーションに必要な場合は、Amazon S3 の使用をご検討ください。これに加えて、Amazon S3 の設計は、1 年に最大 99.99% のオブジェクト可用性を実現します。可用性の高いストレージがアプリケーションに必要な場合も、使用をご検討ください。Amazon S3 に格納されているオブジェクト (ファイル、イメージ、ログ、任意のバイナリデータ) には、直接 HTTP でアクセスできます。Lambda 関数は、仮想プライベートエンドポイントを通じて安全に Amazon S3 と通信できます。S3 内のデータは、Lambda 関数に関連付けられている IAM ポリシーのみに制限できます。

- Amazon Elasticsearch Service[20]

  Amazon Elasticsearch Service (Amazon ES) は、ポピュラーな検索/分析エンジンである Elasticsearch のマネージドバージョンです。Amazon ES は、クラスターのマネージドプロビジョニング、障害検出、ノードの置き換えを処理します。Amazon ES API へのアクセスは、IAM ポリシーを使用して制限できます。

プレゼンテーション層

Amazon API Gateway を使用すると、プレゼンテーション層のさまざまな可能性が生まれます。HTTPS 通信が可能なクライアントであれば、インターネットアクセスが可能な HTTPS API を利用できます。アプリケーションのプレゼンテーション層用として使用を検討できる一般的な例を次に示します。

- モバイルアプリ: Amazon API Gateway および AWS Lambda を通じてカスタムのビジネスロジックに統合できることに加え、ユーザー ID の作成と管理には、メカニズムとして Amazon Cognito[21] を使用できます。
- 静的なウェブサイトコンテンツ (Amazon S3 でホストされているファイルなど): Amazon API Gateway の API を CORS (Cross-Origin Resource Sharing) 対応にすることができます。これにより、ウェブブラウザから、静的なウェブページ内で直接 API を呼び出せるようになります。
- その他の HTTPS 対応クライアントデバイス: 接続されている多くのデバイスは、HTTPS を使用した通信に対応しています。これは、Amazon API Gateway を使用して作成した API とクライアントが通信する方法として、何も特別なことではなく、純粋な HTTPS です。特殊なクライアントソフトウェアやライセンスは必要ありません。

アーキテクチャパターン例

ロジック層を形成する接着剤として Amazon API Gateway と AWS Lambda を使用すると、以下に示すポピュラーなアーキテクチャパターンを実装できます。それぞれの例では、ユーザーが独自のインフラストラクチャを管理する必要のない AWS サービスのみが使用されています。

モバイルバックエンド

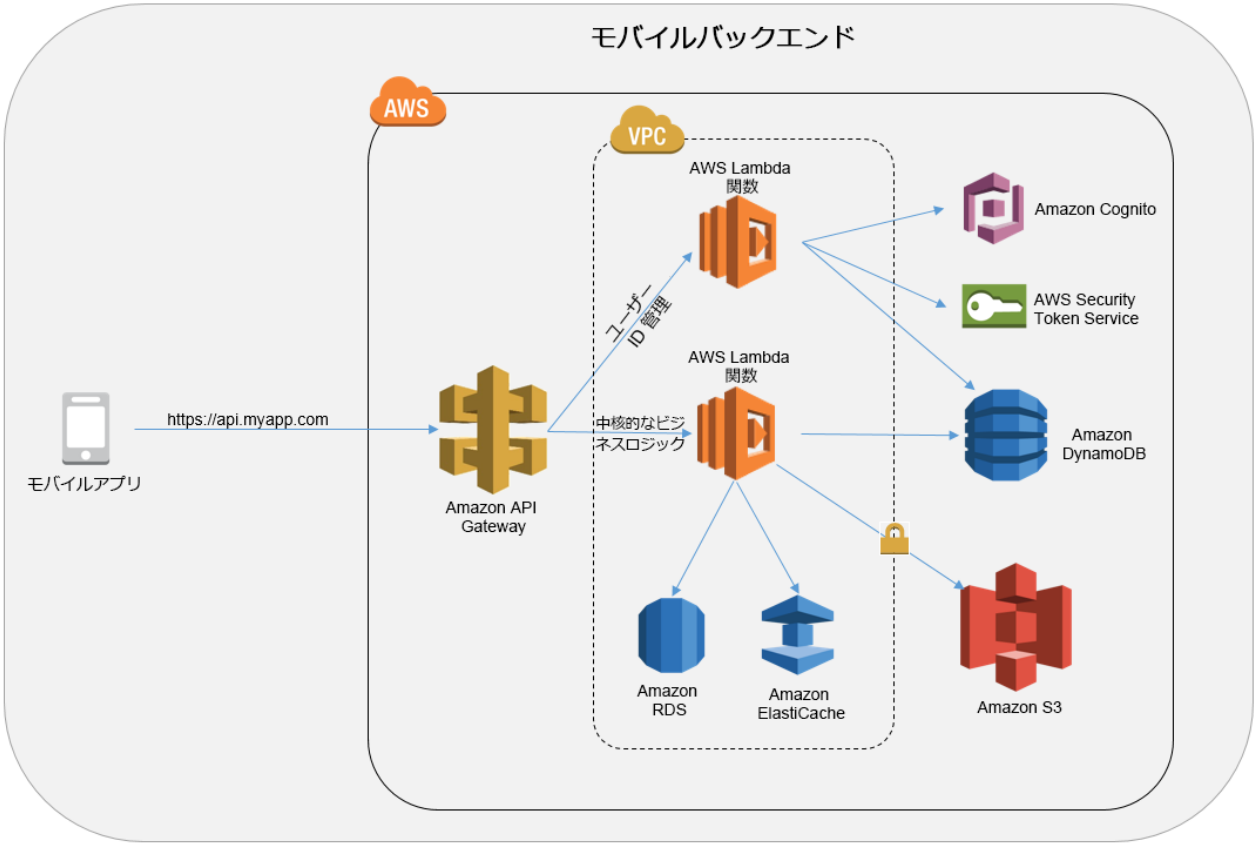

図 3: モバイルバックエンドのためのアーキテクチャパターン

- **プレゼンテーション層:** 各ユーザーのスマートフォンで実行されるモバイルアプリケーション。
- **ロジック層:** Amazon API Gateway と AWS Lambda。ロジック層は、各 Amazon API Gateway API の一部として作成された Amazon CloudFront ディストリビューションによって全世界に配布されます。Lambda 関数としては、Amazon Cognito で管理される、ユーザー/デバイスの ID 管理と認証に固有のものがいくつか考えられます。Amazon Cognito は一時的なユーザーアクセス認証情報用に IAM と統合されます。ポピュラーなサードパーティの ID プロバイダーとも統合できます。その他の Lambda 関数で、モバイルバックエンド用に中核的なビジネスロジックを定義できます。
- **データ層:** 必要に応じてさまざまなデータストレージサービスを活用でき、このホワイトペーパーに記載されているオプションを利用できます。

Amazon S3 でホストされているウェブサイト

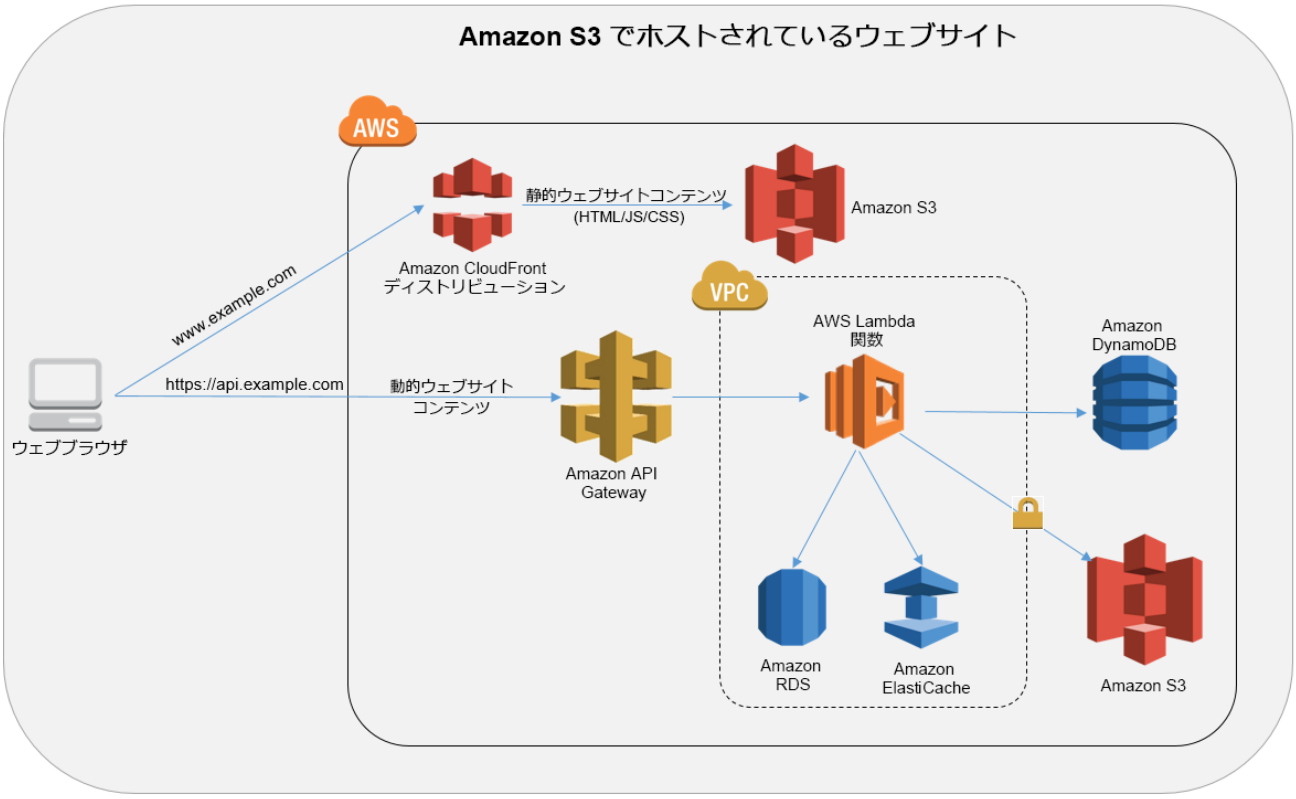

図 4: Amazon S3 でホストされている静的なウェブサイト用のアーキテクチャパターン

- **プレゼンテーション層:** 静的なウェブサイトコンテンツは Amazon S3 でホストされ、Amazon CloudFront によって配布されます。静的なウェブサイトコンテンツを Amazon S3 でホストすると、サーバーベースのインフラストラクチャでコンテンツをホストする場合より優れたコスト効率を実現することができます。ただし、ウェブサイトに豊富な機能を含めるには、静的なコンテンツと動的なバックエンドの統合が必要になる場合があります。
- **ロジック層:** Amazon API Gateway と AWS Lambda。Amazon S3 でホストされている静的なウェブコンテンツは、CORS 対応が可能な Amazon API Gateway と直接統合できます。

- **データ層:** 必要に応じてさまざまなデータストレージサービスを活用でき、このホワイトペーパーに記載されているオプションを利用できます。

マイクロサービス環境

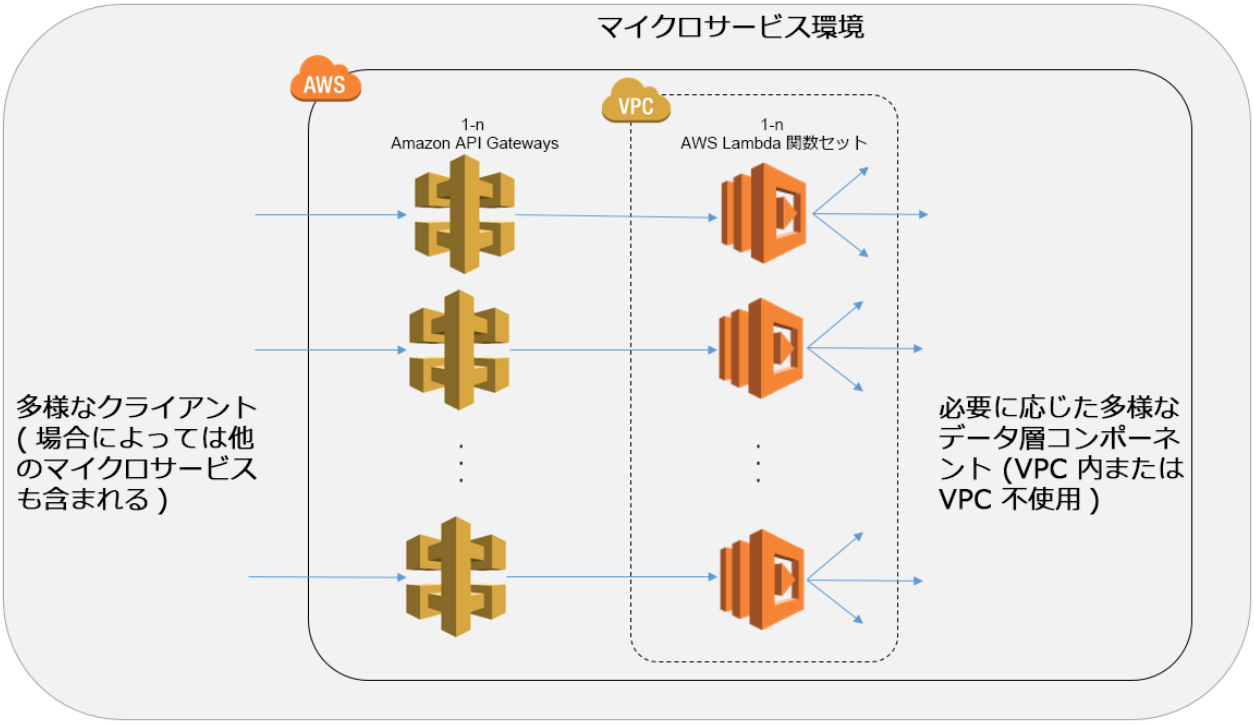

図 5: マイクロサービス環境のためのアーキテクチャパターン

マイクロサービスアーキテクチャパターンは、このホワイトペーパーで説明した典型的な 3 層アーキテクチャには該当しません。マイクロサービスアーキテクチャでは、ソフトウェアコンポーネントの分離が大量に行われるため、全体的に多層アーキテクチャの利点が増幅されます。マイクロサービスを構築するために必要になるのは、Amazon API Gateway で作成した API と、AWS Lambda で実行される後続の関数のみです。チームでこれらのサービスを自由に使用して、必要な粒度レベルまで環境を分離し、断片化することができます。

一般的には、マイクロサービス環境を使用すると、各マイクロサービスを新しく作成するときに毎回生じるオーバーヘッド、サーバーの密度/使用率の最適化に関する問題、複数バージョンの複数マイクロサービスを同時に実行する複雑さ、多数の別々のサービスと統合するためのクライアント側コード要件の拡散などの問題につながることがあります。

ただし、AWS サーバーレスパターンを使用してマイクロサービスを作成した場合は、これらの問題は解決が簡単になり、場合によっては問題が完全消滅することもあります。AWS マイクロサービスパターンでは、後続の各マイクロサービスを作成するための障壁が小さくなります (Amazon API Gateway によって、既存 API の複製も可能です)。このパターンでは、サーバーの使用率の最適化が意味を成しません。API Gateway と Lambda の両方によって、シンプルなバージョニングが可能になります。Amazon API Gateway では、統合のオーバーヘッドを縮小するために、プログラムで生成されたクライアント SDK が多数のポピュラーな言語で提供されています。

まとめ

多層アーキテクチャパターンでは、保守、分離、スケーリングが容易なアプリケーションコンポーネントを作成するというベストプラクティスが推奨されています。ロジック層を作成し、Amazon API Gateway 経由で統合して、AWS Lambda 内でコンピューティングを行っている場合は、目標を達成するための労力を縮小する一方で、その目標に近づいています。これらのサービスを組み合わせると、クライアント用に HTTPS API フロントエンドと、ビジネスロジックを実行するために VPC 内の安全な環境を実現できます。これにより、多数のポピュラーなシナリオを利用して、典型的なサーバーベースのインフラストラクチャを自分で管理する代わりに、これらのマネージドサービスを使用することができます。

寄稿者

本書の執筆に当たり、次の人物および組織が寄稿しました。

Andrew Baird (AWS ソリューションアーキテクト)

Stefano Buliani (AWS Mobile、テクノロジー、シニアプロダクトマネージャー)

Vyom Nagrani (AWS Mobile、シニアプロダクトマネージャー)

Ajay Nair (AWS Mobile、シニアプロダクトマネージャー)

注

[1] http://aws.amazon.com/api-gateway/

[2] http://aws.amazon.com/lambda/

[3] https://aws.amazon.com/vpc/

[4] https://aws.amazon.com/cloudfront/

[5] https://d0.awsstatic.com/whitepapers/DDoS_White_Paper_June2015.pdf

[6] https://aws.amazon.com/api-gateway/pricing/

[7] http://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-mock-integration.html

[8] http://aws.amazon.com/iam/

[9] http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html

[10] http://docs.aws.amazon.com/lambda/latest/dg/intro-core-components.html#intro-core-components-event-sources

[11] https://aws.amazon.com/s3/

[12] https://aws.amazon.com/kinesis/

[13] https://aws.amazon.com/vpc/

[14] https://aws.amazon.com/rds/

[15] https://aws.amazon.com/elasticache/

[16] https://aws.amazon.com/redshift/

[17] https://aws.amazon.com/ec2/

[18] https://aws.amazon.com/dynamodb/

[19] https://aws.amazon.com/s3/storage-classes/

[20] https://aws.amazon.com/elasticsearch-service/

[21] https://aws.amazon.com/cognito/