クラウドコンピューティング
第10回 クラウドサービスの概念と用語

3年、秋学期、選択; 旧「オペレーティングシステム」; ;脚注はELや定期試験の範囲ではありません

ビジネスの要諦

1. (手段を問わず)利益をあげることが目的
2. つまり、売れる機会があるなら、その機会を逃すな！
    - 機会を逃すこと -> ビジネス用語の**「機会損失」**
3. インターネットは24時間いつでも世界中の人が使える。それゆえ
    - サービスの中断(サーバやネットワーク障害)は許されない
    - 人気が出て、ユーザ数が増加した時に、(ほぼリアルタイムで)増加に対応できることは必須

(脚注1) まずは、ビジネスについて、そもそも論を語ります。 とりあえず近代資本主義の是非については論じない

(脚注2) Q: Amazonのサーバが1分おちたら、どれくらい売上を逃すのでしょうか？ A: 単純化して考えますが、 Amazon Japanの場合、年間売上が約4兆円なので、1分あたり数百万円になります。 全世界では100兆円ちかいので1分あたり1億円くらい逃すという見積り

クラウドとは？(AWSの定義からスタート)

> …さまざまなITリソースをオンデマンドで利用することができるサービスの総称です。必要なときに必要な量のリソースへ簡単にアクセスすることができ、ご利用料金は実際につかった分のお支払いのみといった重量課金が一般的です。(AWSのウエブより引用)

- **「必要なときに」「必要な量」**がもっとも重要なクラウドの特徴です
  - 例
    - 3日間だけサーバ1000台ほしい,4日目からは10台かもしれない(例: ソーシャルゲーム)
    - 半日だけスーパーコンピュータが使いたい(128vCPU, メモリ2TBとか)
  - 21世紀のクラウドサービスでは上の条件が大事
    (この条件を満たさない)似たようなサービスなら、昔からあります

(脚注1) 引用元は https://aws.amazon.com/jp/cloud/

(脚注2) クラウドサービスのためには、**サーバとネットワークの大量の在庫が必要です。** (物理サーバでは大変なので)たいていは**仮想化されているサーバやネットワークとして提供**します。 また、大量のサーバの操作が必要なので、プログラムから簡単に利用できる(APIを提供する)ことも重要です。 これにより遠隔かつ自動で**サーバ数の増減**が可能になります。 これがIaC (Infrastructure as Code)

必要な時に、必要な量を(1): www.pyはステートレス

- ショッピングカートを作ることを考えます。 出発点は、いままでどおりwww.py
- www.pyは、ユーザが入力した情報を受け取り、それを表示できます
    - ショッピングなので、 ユーザの**カートにある「商品」と「(買いたい)数量」という情報をサーバに送る**必要があります。 これは今のwww.pyで出来ます。 FORM文とCGIの話です、OK?
- サイトを利用するユーザが増えてきたら(たとえば数万人いるとしたら)EC2一台では裁けないでしょう。 www.pyを動かすEC2を複数うごかします。 EC2(一号機)のEC2コピー群を起動すれば十分です。 www.pyは表示しかしてないので、このサーバ群は、うまく動きます
- しかしながら、**ユーザが入力した情報はwww.pyやEC2が再起動したら失われます【重要】**。 このようなwww.pyの性質を**ステートレス**と呼びます
    - ステートレス = 入力した情報を元に、何らかのルールにそって、出力するだけ。状態情報を持たない

(脚注1) EC2のコピー方法は演習しません。どうせ2個なので:-) ふつうは、 EC2をテンプレート化してコピーしていくか、 インフラを操作するプログラムで起動していきます(IaC)。 AWS Cloud FormationとかTerraformとか使います

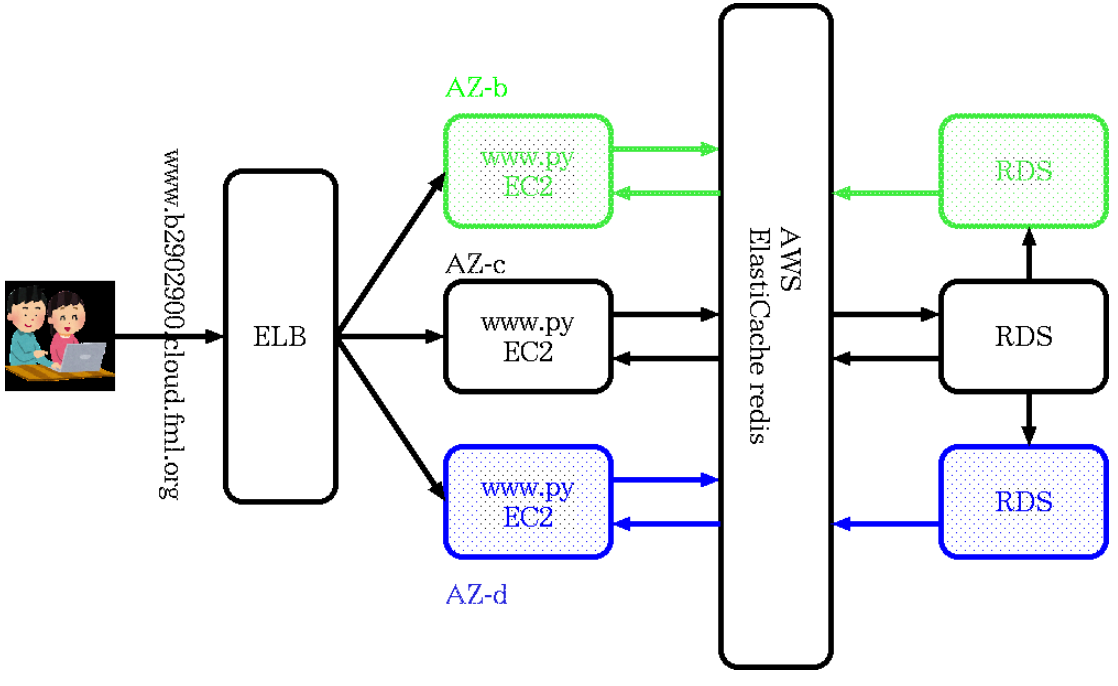
必要な時に、必要な量を(2): ストレージ、ロードバランサー

- 障害時を考え、入力された情報を、どこかに保存する必要があります (ストレージ)
    - キャッシュサービスやRDBMSサービス(AWS RDS)を利用します。 これらはAWSのサービスなので、冗長化の構築作業もメインテナンスなどの面倒なこと一切を**AWSまかせ**に出来ることが重要です
    - 例：購入情報（ショッピングサイトが失ってはいけない重要な情報）
- 多数のEC2群が起動している時、どのEC2にアクセスを誘導するべきでしょうか？
  そして10台EC2があるなら10台が同程度の負荷であることが望ましいですよね？
    - **ロードバランサー**(AWS ELB)というサービスで、 **負荷をバランスしながらアクセスを振り分け**られます。 **自動的にEC2を増減**したり、 **障害を起こしたEC2を運用から外す**といったことも可能です

(脚注1) 改善するには？という話をしています (脚注2) はじめから冗長構成のサービスもあれば、冗長構成がオプションのサービスもあります (脚注3) RDSならAuroraを買うべきで、Aurora買う金がないならAWSなんか使うな！です (脚注4) **AWSまかせに出来て簡単 = エンジニアのスキルは上がらない**ことを気にとめておくべきですね。 あなたは10年後20年後どうやって食べていくつもりかな？
(脚注5) Q: ユーザは、どのEC2にアクセスするの？ A: 設計しだいでしょう。 まぁ何も考えてないと1号機にだけアクセス集中でしょうか。 ちなみに、この問題に対してDNSという単語が最初に浮かんだ人は春学期の内容を理解できています

理想的な構成図(再掲)

- 第01回より図を再掲します
- 今月は、これの部分的な構築と解説をしていきます



(脚注) ひととおり書いてみると、 (1)ユーザからのアクセスをELBで振り分けて、 (2)異なるAZにあるEC2のどれかのwww.pyにアクセスします (3)www.pyはキャッシュサービスにカートの情報等を(一時)保存 (4)購入情報などの重要な情報はRDSサービスに情報を保存

用語: クラウドの種類: EC2はIaaS、Office365はSaaS

用語	意味	事例
SaaS	S=サービス、ユーザが直接つかうサービス	ウエブメール(Office365、photon.chitose.ac.jp)
PaaS	P=プラットホーム	AWS RDS
IaaS	I=(IT)インフラストラクチャ	AWS EC2

- XaaSはX as a Serivceの頭文字。Xの部分は様々で、なんでもXaaSと表現するのが流行して久しい(恥)
  - IaaSで使うLinuxもので二大有名どころが**Debian** GNU/Linux系と**Redhat系**です
  - みなさんが普段つかっているメール(photon.chitose.ac.jp)がSaaSです
- 本科目では、 いままで、 IaaS(EC2)とIaaS上のアプリケーション(www.py、いわゆるWeb APIサーバ)だけを扱ってきました。 このあとはPaaSとの連携と冗長化構成をやっていきます。 SaaSの構築は扱いません

(脚注1) AWSには、Linux以外のUnixもあるし、商用OSも利用できますが、詳細は省略

(脚注2) いろいろな Linux ディストリビューション(Unix クローンOS)があります。 「Unix クローン」とは「オリジナルUNIXの動作を真似するOSをゼロから作ったもの」です。 数万個のソフトウエアを寄せ集めたものを**ディストリビューション**と呼びます。 長いので、**ディストロ(distro)**と略して呼ぶ人も多いです。 標準インストールのOSは、distro中の最重要な千数百のソフトウエアから構成されています (脚注3) Debian系とは、**Debian(本家)、Ubuntu**、ラズパイのOSなどです。 フリーソフトウエアの理念に近い側と言っても良い。 Redhat系とは、Redhat(本家)、Fedora、CentOS、Miracle、Alma、Rockyなど...

クラウドコンピューティング
第11回 データの永続化

3年、秋学期、選択; 旧「オペレーティングシステム」; ;脚注はELや定期試験の範囲ではありません

【復習】

- 「必要なときに」「必要な量」のITインフラやサービスを購入できることがクラウドの特徴
    - ユーザ数に応じてWeb APIサーバを必要なだけ何台でも起動させ(られ)ます
    - 「ソーシャルゲーム」や「BLACk FRAIDAYなどの特売イベント」はクラウドが活かせる好例
- システム設計と運用では、きちんと障害対策を考えているか？が重要です
- Web APIサーバをステートレスに作ると障害時にはデータが失われます
    - ショッピングを例にとると、ショッピングカートの情報や購入履歴が消えてしまいます

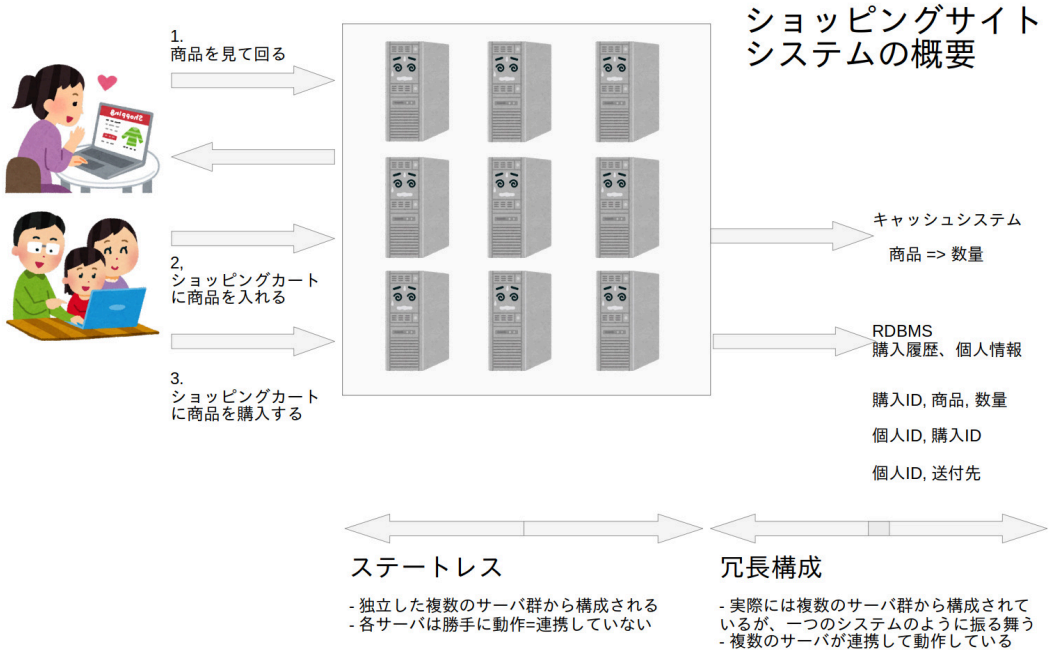
(脚注) どれだけ障害対策を考えられているか？で、エンジニアの腕が分かるというものです

ショッピングサイトの全体像

1. ショッピングサイトにアクセスし、商品をいろいろ見て回ります

2. ショッピングカートに商品を入れます。 キャッシュシステムにカートの内容を保存します

3. 商品を購入した場合は、キャッシュではなく、きちんと履歴をRDBMS(いわゆるSQLサーバ)に(原則、永久)保存します

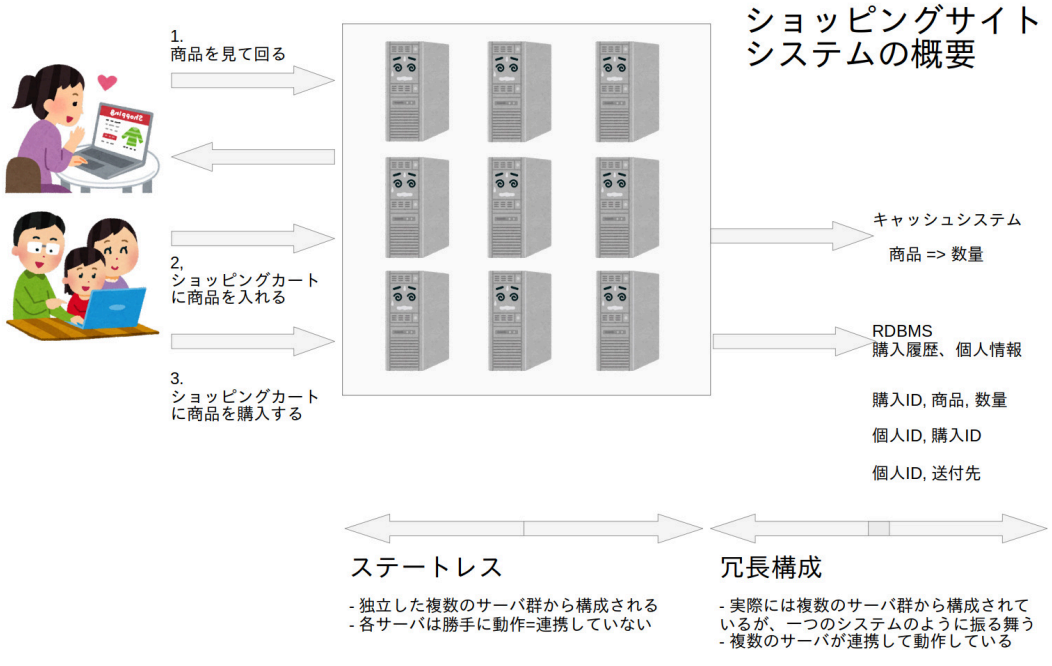


(脚注1) 1.は、「同じコンテンツを持ったサーバ群のどれか一つ」にユーザを誘導すればよいことに注意してください

(脚注2) 2.キャッシュシステムも3.RDBMSも障害に備えて冗長化されているシステムという想定です

ステートレスなWeb APIサーバ群(図の1.)

- Web APIサーバ = 本科目ではwww.pyのこと
(図中央のサーバ群に相当する)
- ショッピングで見てまわるときは、コンテンツを見るだけ(READ ONLY)ですよね？ (図の1.)
- 一式EC2の上にコンテンツとwww.pyを構築して、EC2をコピーすれば100台でも起動できます
(どのサーバに誘導するのか？は次回のテーマ)

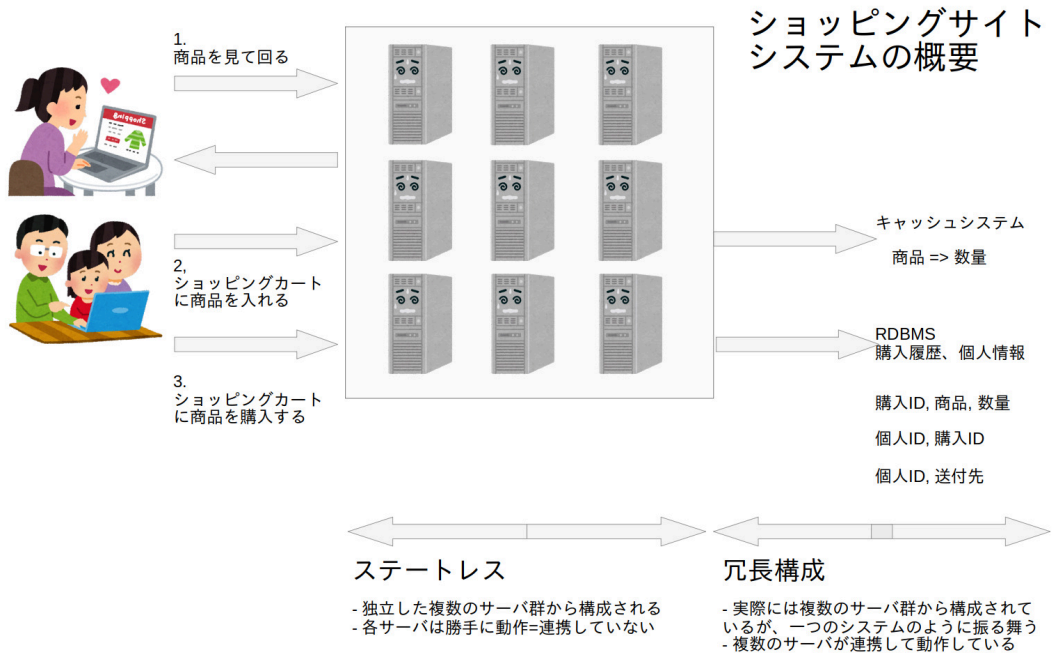


(脚注1) (本題ではないので演習ではやりませんが) 実際には、もっと多数の商品と、その商品の説明や画像のコンテンツを持っているはずです。 Amazonみたいに商品が1000万点とかあるなら違う作り方が必要ですが、 最終課題で考えてみると良いテーマでしょう

(脚注2) 商品が数十〜数百ならWeb APIサーバ全体をコピーしてよいと思います。 ちなみに、 アプリのなかに画像など(assets)をまるごと入れたシングルバイナリを作るやり方もあります。 例えば、Go言語なら go-bindata や go-assets を使って、そういうシングルバイナリを作ります。 コンテナなら、なおさら、このやり方がよいよね？

キャッシュシステム(図の2.)

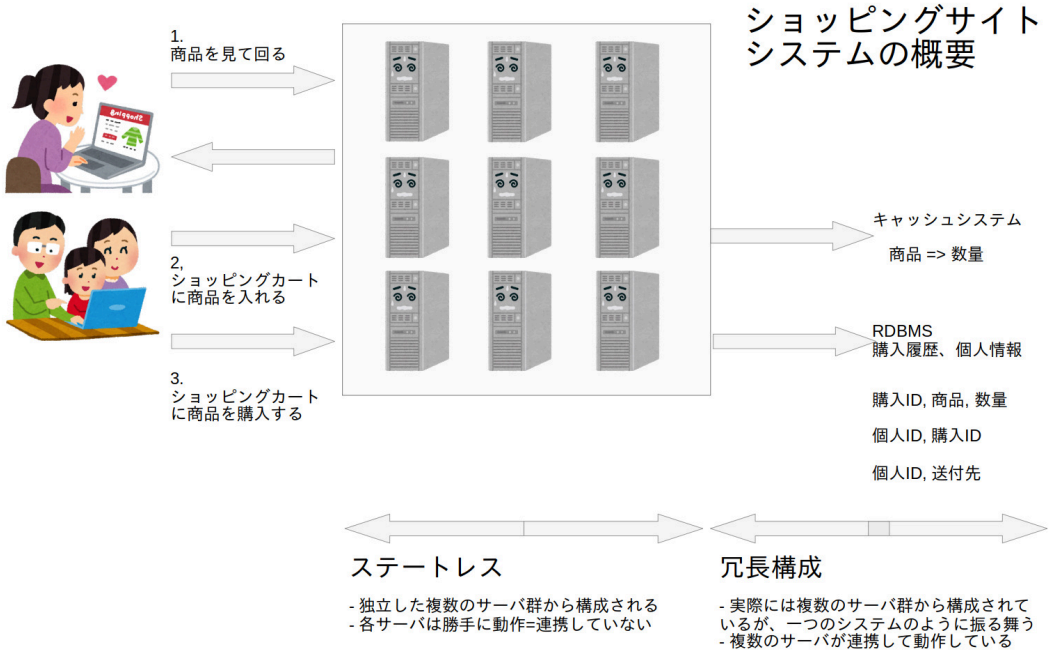
- `{商品 => 数量}`という値の組を保存できれば十分
- なぜなら、ショッピングカートなら、 顧客番号と商品と数量を保存しておけば、再現できますよね？ あと、購入前なら、最悪いちから商品を選び直してもらうのもありでしょう
- 性能重視。やることが単純なのでキャッシュシステムは高速な動作が期待できます。 また、安定した性能が出せるように、古い情報は消してデータ量を抑制したりもします



(脚注1) www.pyやEC2が落ちたり再起動したときは、 ショッピングカートは一からやりなおしという作り方もありえます。 このへんは設計と運用の問題です。 でもショッピングカートをやり直させるのは、ビジネスチャンスを逃すことになるので、あまりやらないとは思いますけど ... (そういうサイトも実在はしますね) (脚注2) キーバリューストアやハッシュ、連想配列などシステムによって色々な呼び方がありますけど、 ようするに KEY => VALUE の組を扱うシステムです

RDBMS(リレーショナルデータベース管理システム,図の3.)

- キャッシュは性能重視で、あくまでも一時保存に使います
- 購入履歴は、原則として消しません。そういったものはRDBMSに書きこみます
- 何も考えずにRDBMSを構築すると、たんにサーバを一台つくって終わりになりますが、 これでは障害時にデータが失われます。 きちんと冗長構成のRDBMSを構築する必要がありますが、 けっこう面倒な構築なので、AWSなら素直にAWS RDSを購入しろ！という話になります



(脚注1) 前ページで述べたように、 ある程度時間が経つとキャッシュは消えていく想定なので、 サイト会員のカートはRDBMSにも保存するとか、 キャッシュとRDBMSの使い分けが必要でしょう。 やはり、これも設計と運用の話になります (脚注2) RDSは選択肢が6種類あるのですが、もちろんAuroraが推奨です。 高価だけど高いだけの価値があります (注: AWS Academyでは利用できません)

クラウドコンピューティング

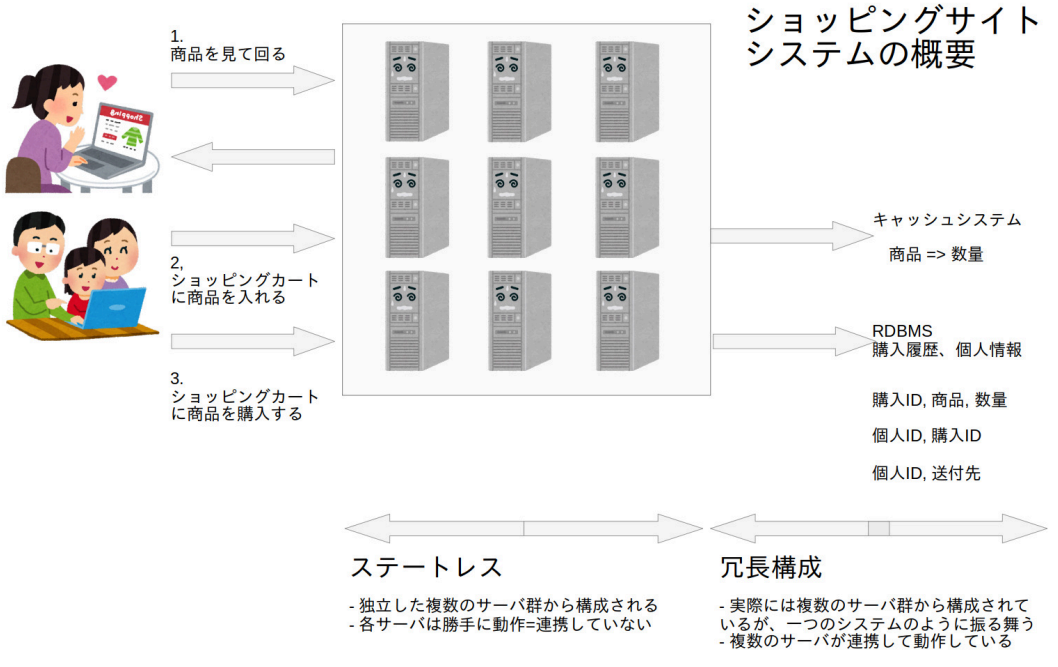
第12回 ロードバランサー

3年、秋学期、選択; 旧「オペレーティングシステム」; ;脚注はELや定期試験の範囲ではありません

【復習】ショッピングシステムの全体像

- Web APIサーバ = 本科目ではwww.pyのこと
  (図中央のサーバ群に相当する)

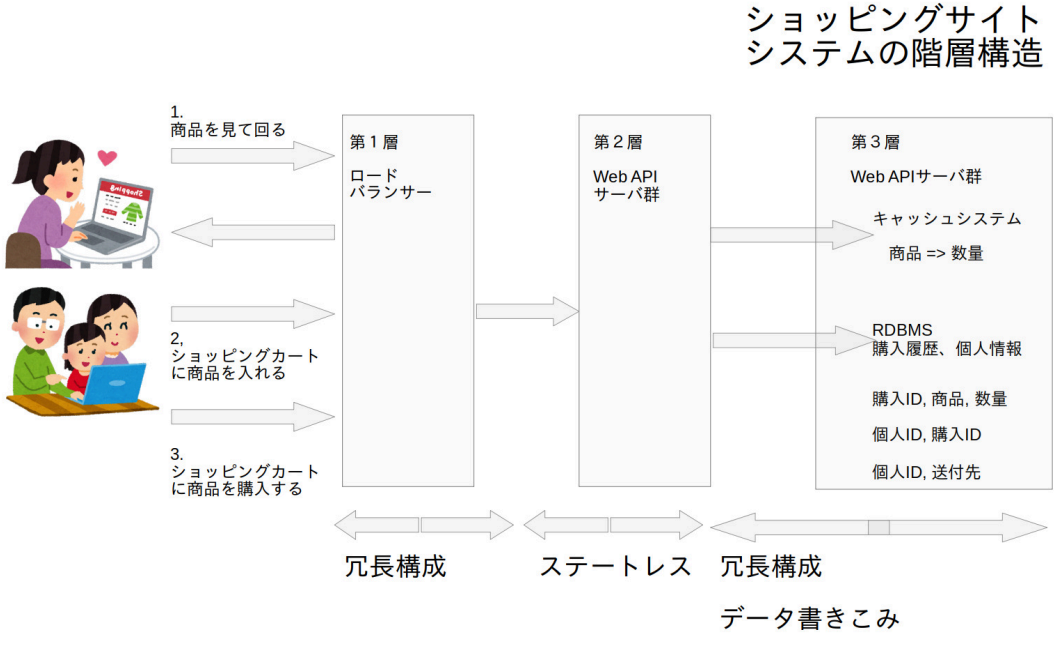
- 一式EC2の上にコンテンツとwww.pyを構築して、EC2をコピーすれば何台でも起動できます
  - 100台だろうが1万台だろうがドンとこい
  - **どのサーバに誘導するのか？が今回のテーマ**
  - 「ロードバランサー」が、その担当



(脚注) もっとも1万台も起動したら裏側の書きこみ部分が保つのか？は、別途、検討が必要です

ショッピングシステムの階層構造

- 階層モデルとして考えられます (左->右: 1->3)
- 第1階層「ロードバランサー」はネットワークの担当です。 トラフィック(通信)を第2階層のどこか(Web APIサーバの一つ)に振り分けます
- 第2階層「Web APIサーバ」が処理の本体、ビジネスロジックの実装部分。 スケールできるようにステートレスなアプリを作ります
- 第3階層はデータを確実に読み書きする担当



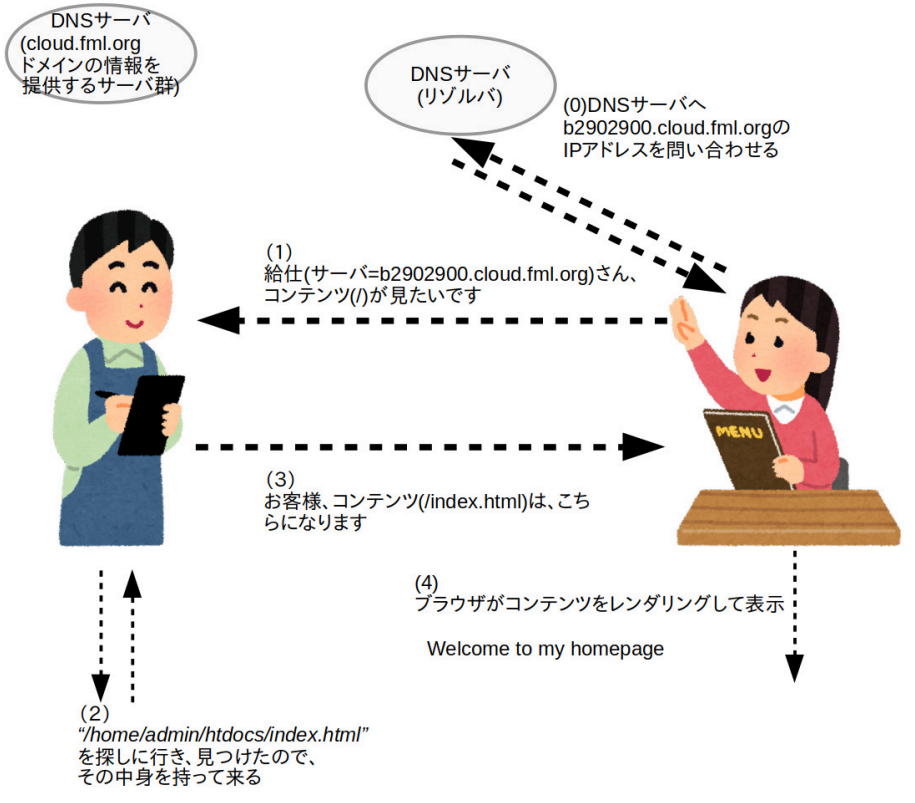
(脚注) 同じ「階層」という用語ですが、 ファイルシステムとは考えていることが異なります。 ビルの1階2階のように階層として表現できるところだけは同じですが ... (a)"ファイルシステム"は**階層分類**を表現していました。 あの**木構造は「分類の大小関係」という暗黙の概念を含んでいます。** そして同じ階層で横に並ぶモノは、 (大小関係では同レベルですが)分類は異なるモノです。

(b)本スライドは単なる階層で、大小関係はありません。 同じ階層には同じモノが並んでいて数量が増減します

【復習、一年分】HTTPアクセスの全体像

ブラウザにURLを与えクリックした時の動作

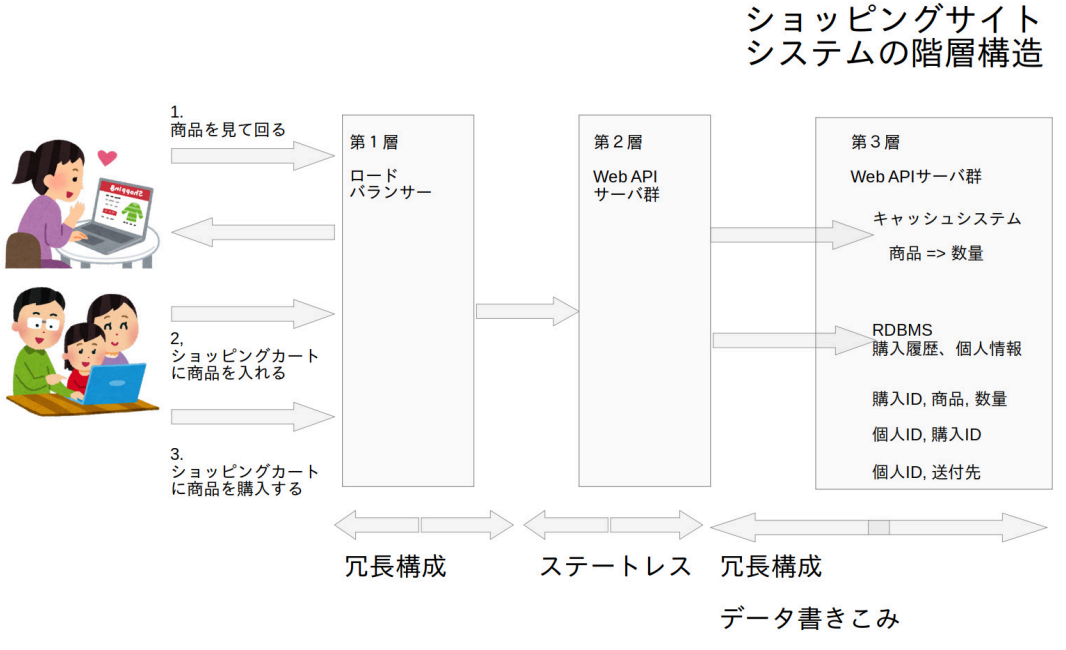
- (0) DNSに問い合わせて、URLのサーバ名のIPアドレスを調べる
- (1) サーバのIPアドレスへTCPコネクションを貼り、 HTTPプロトコルを使い、コンテンツを要求
- (2) 複雑な処理が必要な場合は、裏側で別のサーバに処理を依頼する (これがWeb APIサーバ)
- (3) コンテンツのダウンロードを行う
- (4) ブラウザは、ダウンロードしたコンテンツをレンダリングし、画面上に表示する



(脚注1) コンピュータネットワークの第9回「DNS」の復習です。 (脚注2) いわゆるサーバクライアントモデル。 サーバとはサービスを提供する側のことなので、 この図では、給仕さんがサーバ(この図はHTTPの説明なので「WWWサーバ」)に相当します

ロードバランサーの動作

- URLのサーバ名はロードバランサーに設定
  - 例: www.google.co.jp
  - 例: lb.b2902900.cloud.fml.org
- ブラウザは(、DNSを引き)、ロードバランサー(のIPアドレス)に対してHTTPアクセス
- ロードバランサー(第1階層)が、 Web APIサーバ群(第2階層)のどれかにHTTPを振り分けます

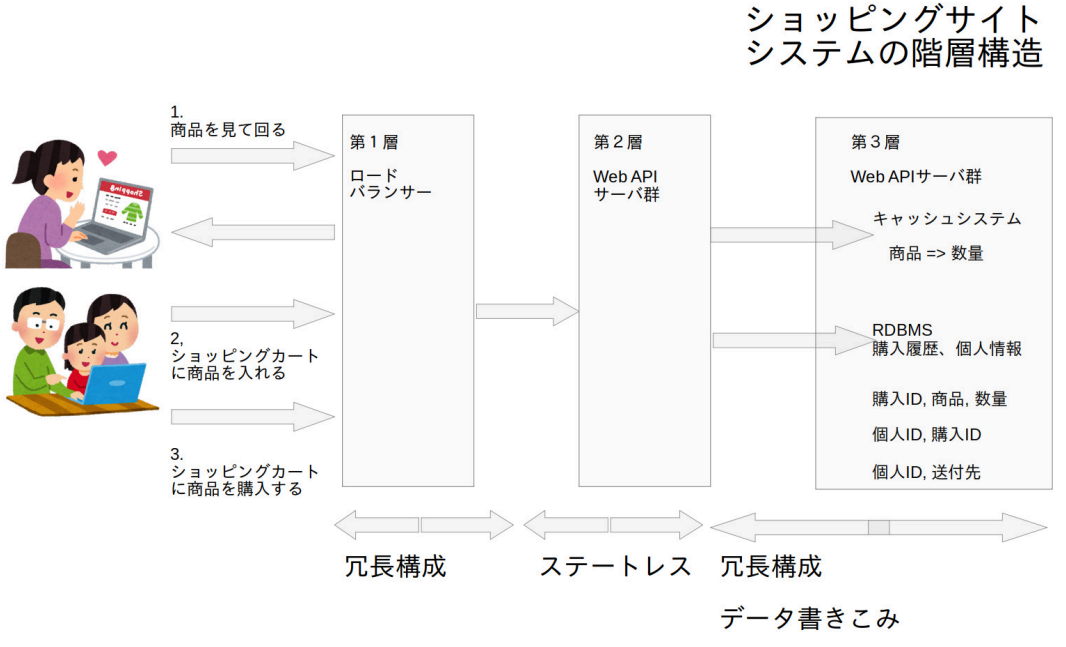


(脚注1) だからELBの設定をしたら「ELBのコンソールでDNS名を調べて申請してください」ということになるわけです

(脚注2) ロードバランサーの返すIPアドレスが1個だったり複数個だったりしますが、 そのへんのネットワーク実装の詳細は省略

AWS ELBの特徴、ELBで出来ること

- 高可用性
  - ロードバランサーが障害を起こしたら全システムがダウンするので、**複数のサーバ群が協調して動作しています(分散システム)**
  - ネットワーク障害に備えて、デフォルトが**マルチAZ構成(異なるAZ上にサーバ群を構築)**
- モニタリング
  - 振り分け先の負荷や障害を監視します
- 自動スケーリング
  - 負荷に応じて自動的にELBのサーバ群も増減
  - ELBとAWS Auto Scalingが連動することで、負荷を見ながらWeb APIサーバを増減可(振り分け先も増減するEC2に自動追従)



(脚注1) その他の特徴については公式を参照 -> https://aws.amazon.com/jp/elasticloadbalancing/

ロードバランサーも製品によって出来ることが異なるので、 あくまでもAWS ELBの例です

(脚注2) ちなみにELBも、実体はELBイメージを動かすEC2群のはずで、このEC2群が増減しています