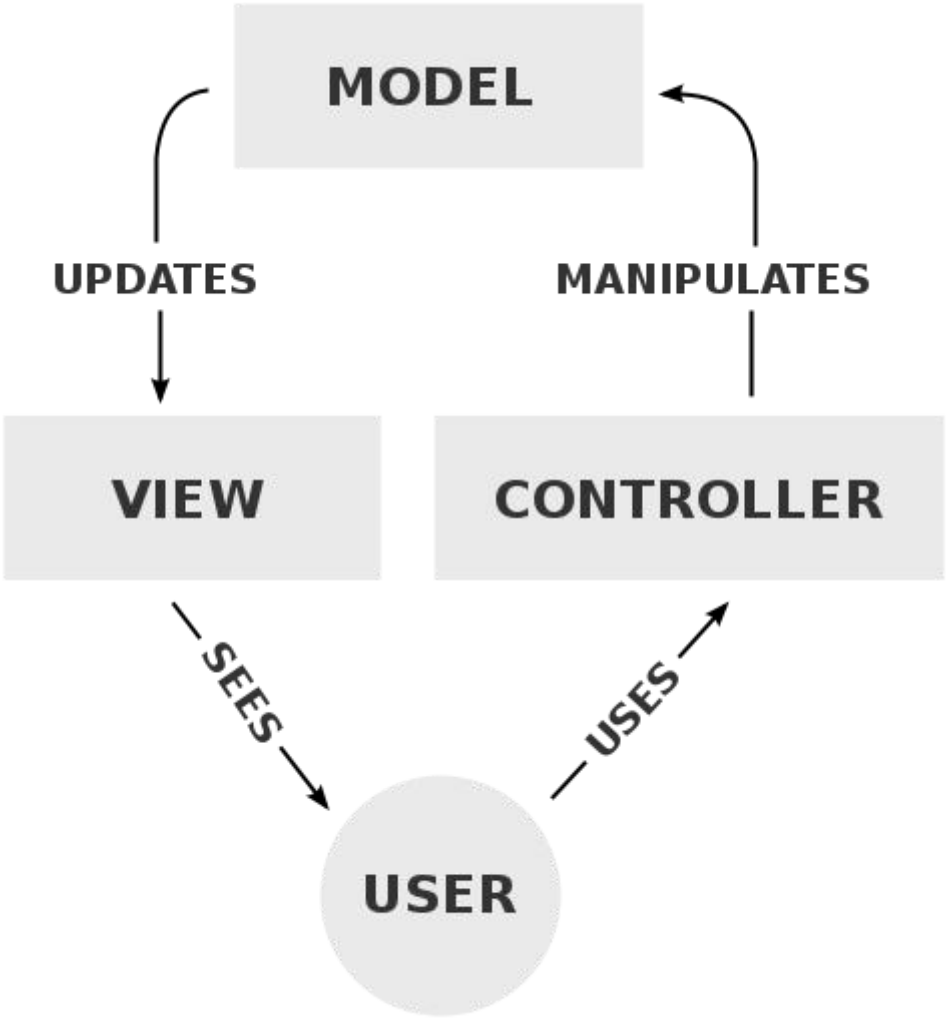
01. MVCとは

ドメイン駆動設計が考案される以前、MVCの考え方が主流であった。



◆ MVCからドメイン駆動設計への発展

しかし、特にModelの役割が抽象的過ぎたため、開発規模が大きくなるにつれて、Modelに役割を集中させ過ぎてしまうことがあった。それから、ドメイン駆動設計が登場したことによって、MVCは発展し、M・V・Cそれぞれの役割がより具体的で精密になった。

02. ドメイン駆動設計とは

◆ドメインエキスパートとユビキタス言語

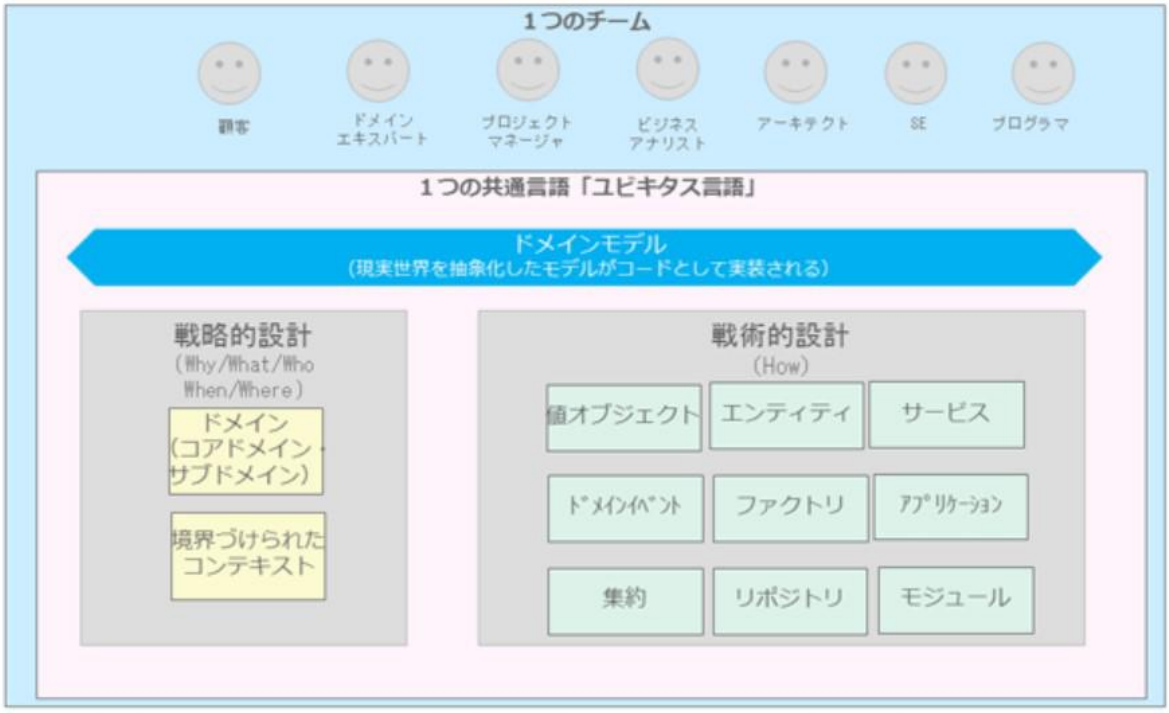
ドメインエキスパート（現実世界の業務内容に詳しく、また実際にシステムを使う人）と、エンジニアが話し合いながら、設計していく。設計の時、ドメインエキスパートとエンジニアの話し合いに齟齬が生まれぬように、ユビキタス言語（業務内容について共通の用語）を設定しておく。

◆ 戦略的設計

1. ドメインエキスパートと話し合い、現実世界の業務内容に含まれる『名詞』と『振舞』に着目。
2. 『名詞』と『振舞』を要素として、EntityやValueObjectを設計。
3. 設計されたEntityやValueObjectを用いて、ドメインモデリング（オブジェクト間の関連付け）を行う。

◆ 戦術的設計

戦略的設計を基に、各オブジェクトとオブジェクト間の関連性を実装していく。



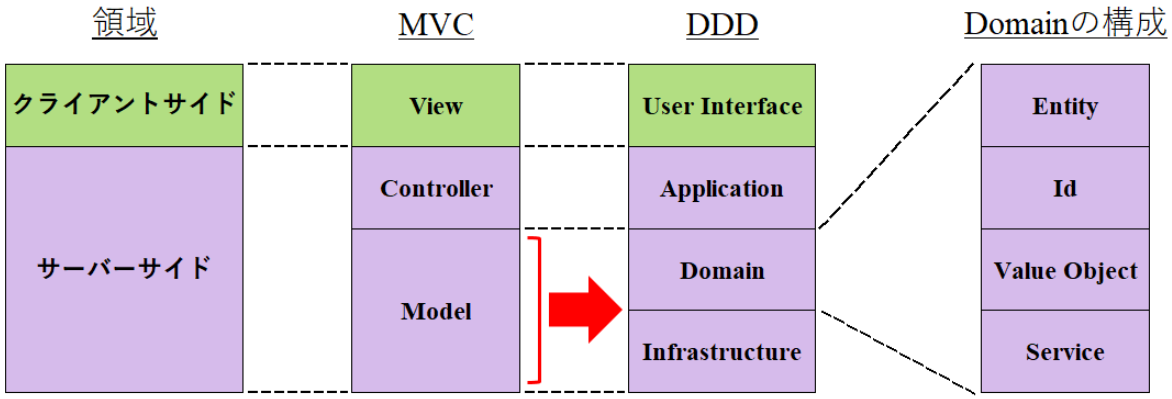
◆ ドメイン駆動設計の派生型

現在までに、ドメイン駆動設計の派生型がいくつか提唱されている。

- **Layered architecture**
- **Hexagonal architecture**
- **Onion architecture**

03. Layeredアーキテクチャ型ドメイン駆動設計

◆ 責務の分担方法

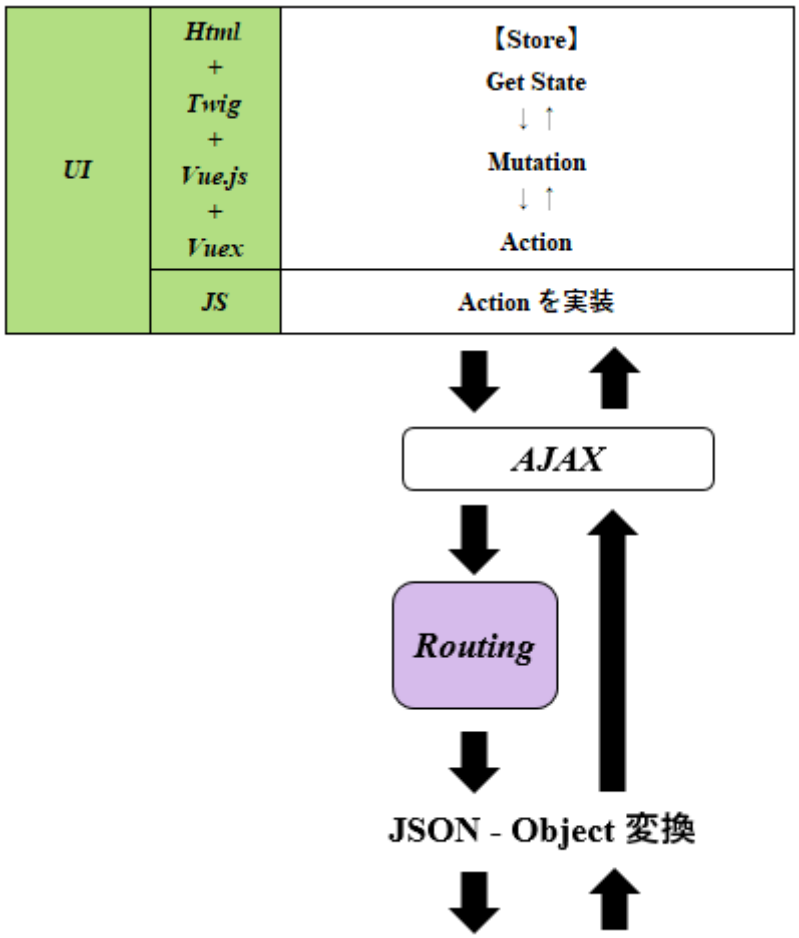


Layeredアーキテクチャ型ドメイン駆動設計において、MVCは、以下の4層に再編成できる。

- **User Interface層**
- **Application層**
- **Domain層（ビジネロジックをコード化）**
- **Infrastructure層（データベースとマッピング）**

04. アーキテクチャにおける処理の流れの全体像

処理フローは、『(Vuex) ⇄ (AJAX) ⇄ (DDD) ⇄ (DB) 』で実装される。クリックで解説ページへジャンプ。



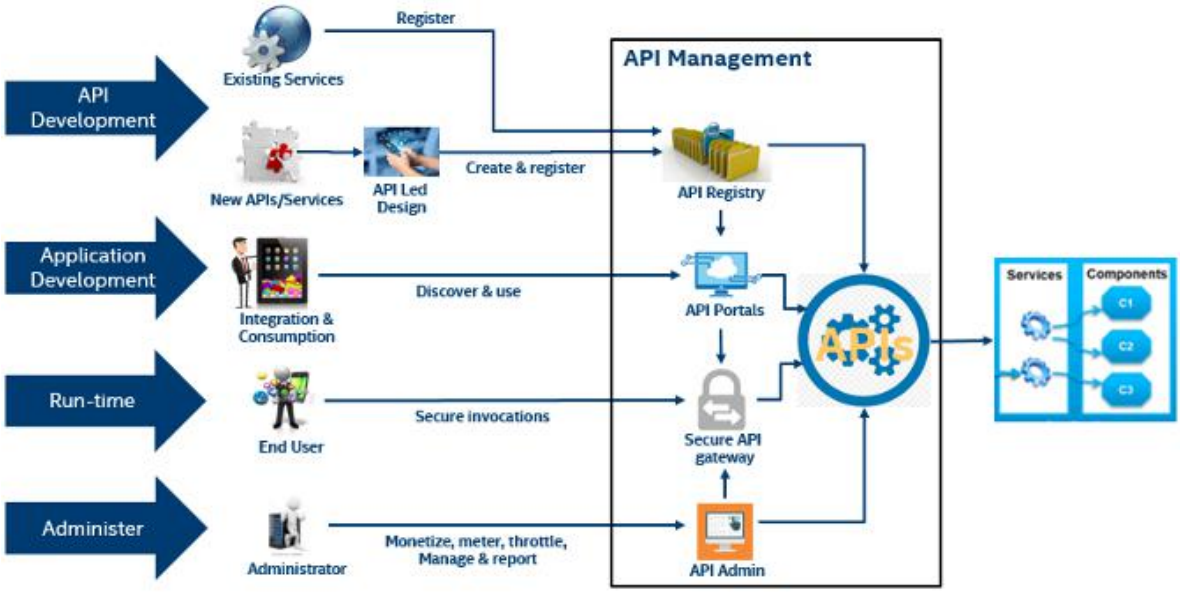
Application		itemController # itemRepository を実行
Domain	*Entity*	itemRepository
	Id	
	Value Object	
	Service	
Infrastructure		itemRepository を実装 # データベース接続 # SQLの動的生成によってデータ取得

整形

Database	itemTable

05. API ⇆ サーバサイド

◆ APIの概念



> In computer programming, an application programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication among various components.

> コンピュータプログラミングでは、アプリケーションプログラミングインターフェイス（API）は、サブルーティン定義、通信プロトコル、ソフトウェアを構築するための一連の方法。 一般的に、様々なコンポーネント間で明確に定義された一連の通信方法のことを言う。

【具体例】

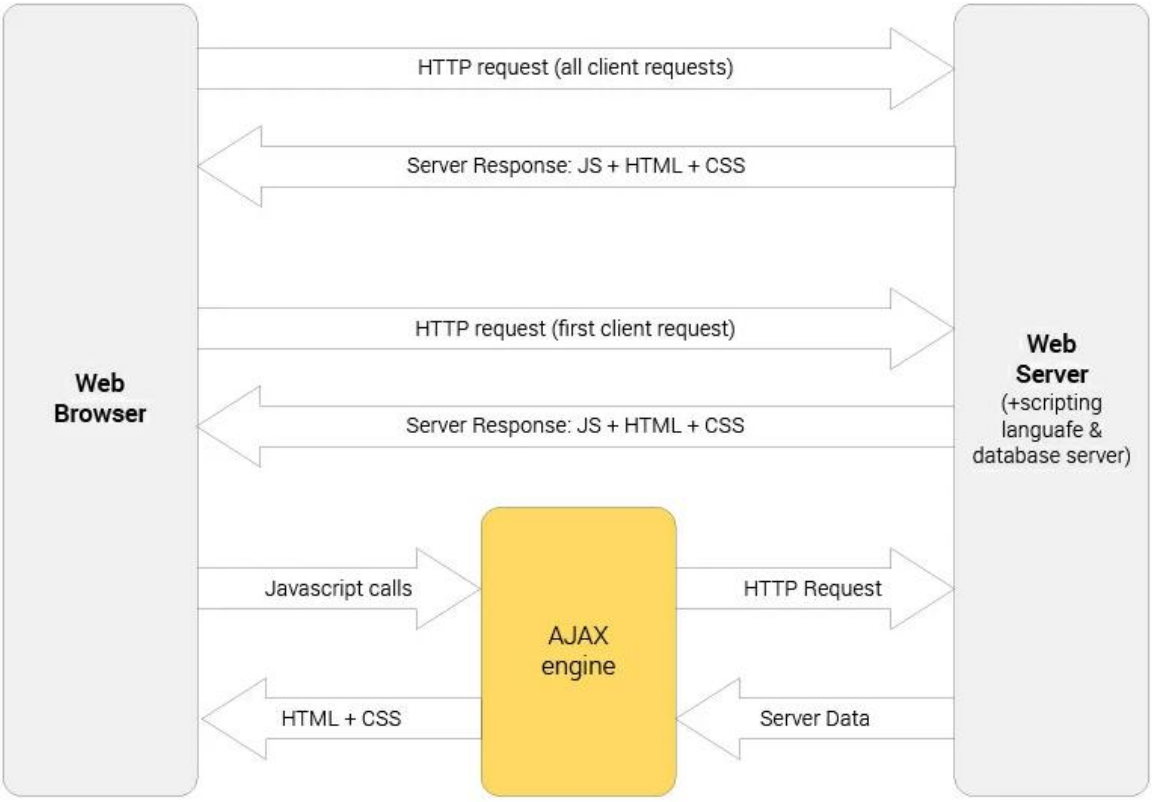
- **AjaxにふくまれるXMLHttpRequestとDOM**

```
```

- **JSON API**

```
    /**
     * @var \DateTime
     *
     * #マッピングするテーブルを指定
     * @ORM\Column(name="date", type="datetime")
     *
     * #Expose()でJSON形式に変換することを宣言
     * @JSON\Expose()
     *
     * #JSONに出力するときのフォーマット
     * @JSON\Type("DateTime<'Y-m-d'>")
     */
    private $date;
```

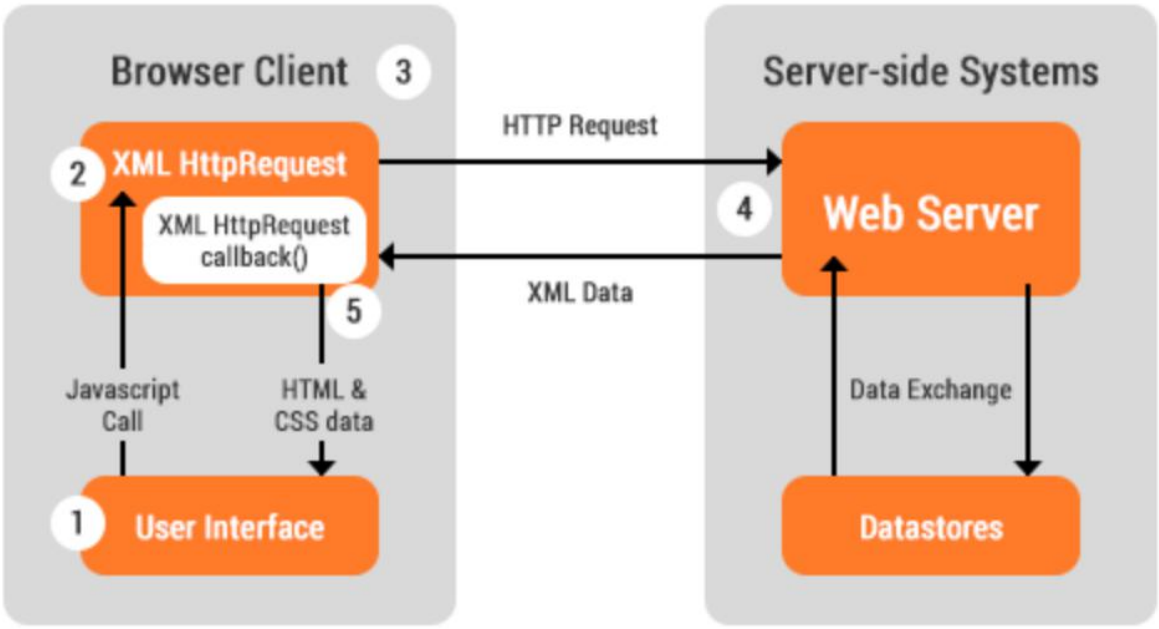
◆ AJAXを用いたAPI



AJAX（Asynchronous JavaScript + XML）は、以下の４つから構成される。

- **JavaScript**：
- **XML HttpRequest**
  オブジェクト形式で書かれ、ブラウザとサーバー間を繋ぐAPI。
- **DOM**：
  xmlやhtmlをツリー構造で表現することによって、ブラウザとサーバー間を繋ぐAPI。

- **XML**：



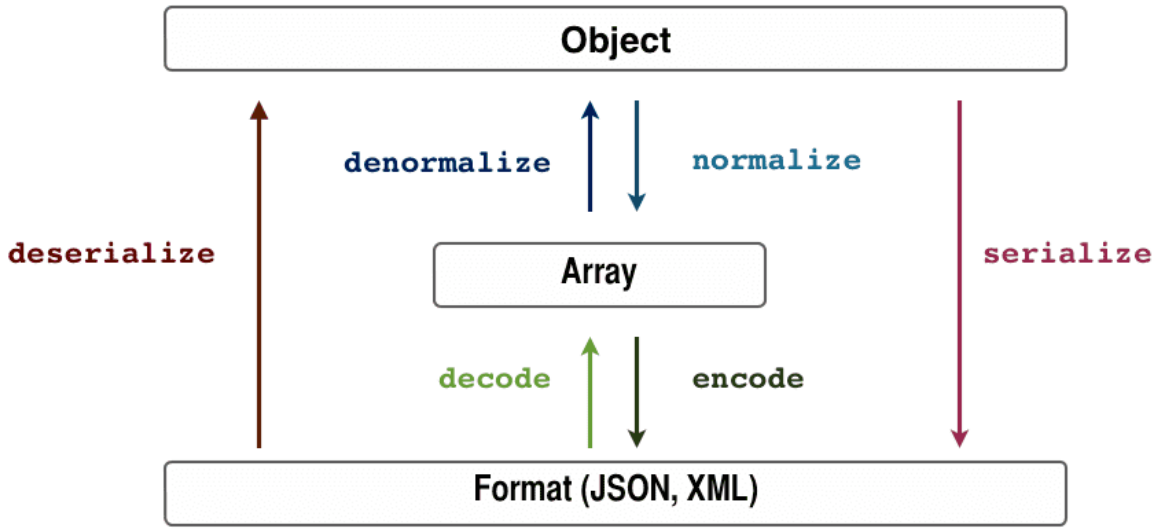
1. ページ上で任意のイベントが発生（ボタンクリックなど）
2. JavaScript + XMLHttpRequestでサーバーに対して、ルーティングを基にコントローラにリクエストを送信
3. サーバーで受け取った情報を処理。サーバーの処理中もクライアントは操作を継続可能（これぞ非同期通信）。
4. コントローラは処理結果をJSONやXMLなどの形式でレスポンスを送信。
5. レスポンスを受けて、DOMでページを更新。

【実装例】

```
  static find(criteria, b, c) {

    //jQueryのAJAXメソッド発動
    return $.ajax({
      type:        'GET', #HTTPリクエストとしてGETメソッドを指定
      url:         '/xxx/xxx-url', #コントローラを指定
      contentType: 'application/json',
      dataType:    'json', #リクエストにJSON形式を指定
      data: (() => {
        const query = {
          criteria: criteria,
          b: b,
          c: c,
        };

        // 検索条件を設定
        if (criteria.id) {
          query.id = criteria.id;
        }

        return query;
      })(),
    })
      .then((data) => {
        return {
          dataの配列
        };
```

```
    });
}
```

◆ Object ⇄ JSON の変換



【実装例】

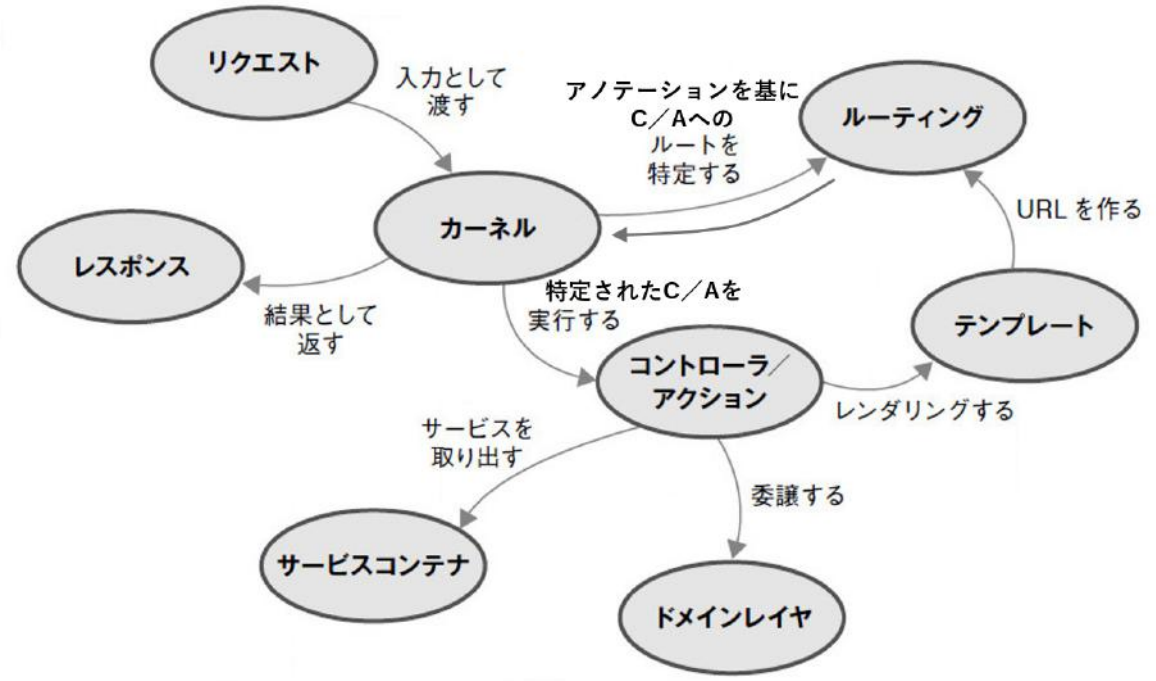
```
#【Object】
{foo: [1, 4, 7, 10], bar: "baz"}
```

⇓ ⇑

```
#【JSON】
'{"foo":[1,4,7,10],"bar":"baz"}'
```

06. 制御プログラム（カーネル）との

図2-9　アプリケーションの構成要素とその関係



◆ カーネルに必要なオブジェクト

1. Requestオブジェクト：グローバル変数から収集した情報やHTTPリクエストのヘッダ情報やコンテンツ情報を保持
2. カーネルオブジェクトの`handle()`：送られてきたURLを基にしたコントローラ／アクションへのルートの特定、特定されたコントローラ／アクションの実行、テンプレートのレンダリング
3. Responseオブジェクト：HTTPレスポンスのヘッダ情報やコンテンツ情報などの情報を保持

◆ オブジェクトから取り出されたメソッドの役割

1. カーネルが、クラアントからのHTTPリクエストをリクエストオブジェクトとして受け取る。
2. カーネルが、送られてきたURLとルート定義を基に、リクエストに対応するコントローラアクションを探し、実行させる。その後、テンプレートがURLを生成。
3. カーネルが、その結果をレスポンスオブジェクトとしてクライアントに返す。
   このカーネルを、特別に『HTTPカーネル』と呼ぶ。

【app.phpの実装例】

```
$kernel = new AppKernel('dev', true);
if (PHP_VERSION_ID < 70000) {
    $kernel->loadClassCache();
}
$request = Request::createFromGlobals();  #（1）
$response = $kernel->handle($request); #（2）
$response->send(); #（3）
$kernel->terminate($request, $response);
```

【Kernel.phpの実装例】
handle()が定義されているファイル。ここで定義されたhandle()が、C/Aへのルートの特定、特定されたC/Aの実行、テンプレートのレンダリングを行う。

```
/**
* {@inheritdoc}
*/
public function handle(Request $request, $type =
HttpKernelInterface::MASTER_REQUEST, $catch = true)
{
  $this->boot();
  ++$this->requestStackSize;
  $this->resetServices = true;

  try {
  return $this->getHttpKernel()->handle($request, $type, $catch);
  } finally {
  --$this->requestStackSize;
  }
}
```

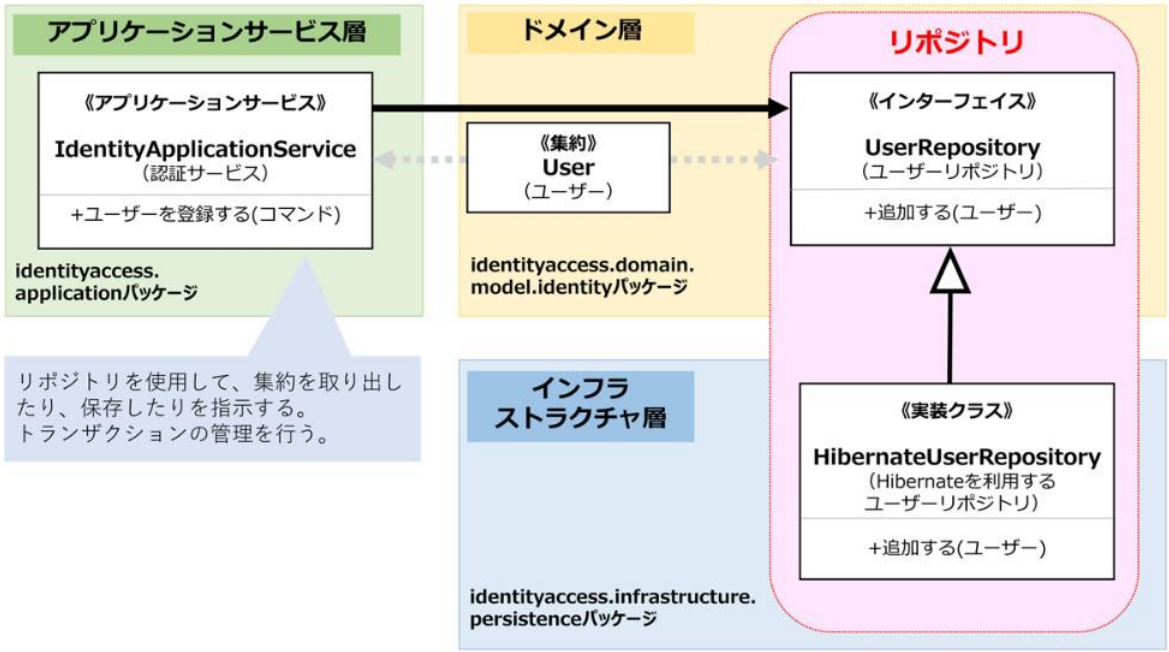
07. Dependency Inversion Principle（依存性逆転の原則）

◆ DIPとは

1. 上位のモジュールは、下位のモジュールに依存してはならない。どちらのモジュールも『抽象』に依存すべきである。
2. 『抽象』は実装の詳細に依存してはならない。実装の詳細が「抽象」に依存すべきである。

◆ DIPに基づくドメイン駆動設計

Repositoryの抽象クラスは、ドメイン層に配置する。そして、Repositoryの実装クラスはInfrastructure層に配置する。抽象クラスで抽象メソッドを記述することによって、実装クラスでの実装が強制される。つまり、実装クラスは抽象クラスに依存している。依存性逆転の原則に基づくことによって、ドメイン層への影響なく、Repositoryの交換が可能。



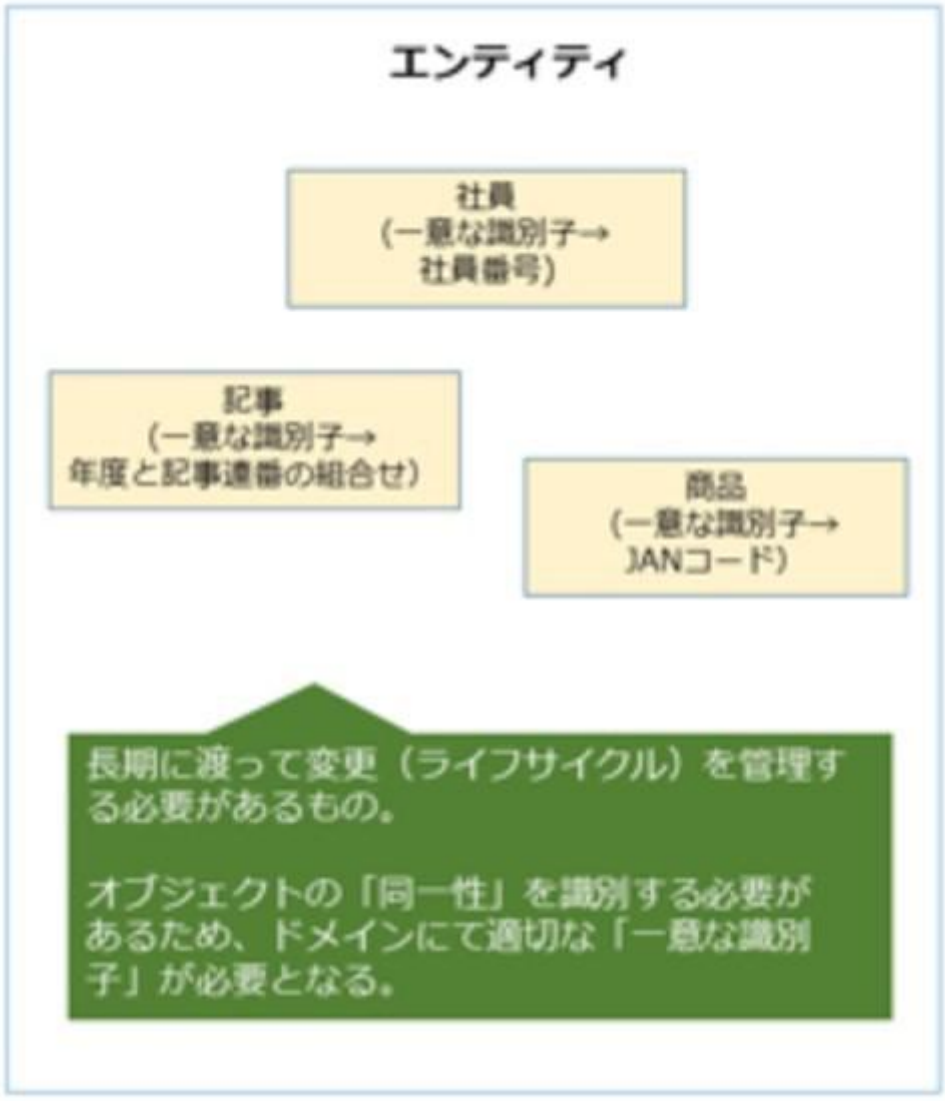
08. Entity

◆ Entityとは

オブジェクトをEntityとしてモデリング／実装したいのならば、以下に条件を満たす必要がある。

（ユビキタス言語の例）顧客、注文など

1. 状態を変化させる必要があるプロパティをもつ。
2. オブジェクトにアイデンティティがあり、他のオブジェクトと同じ属性をもっていても、区別される。



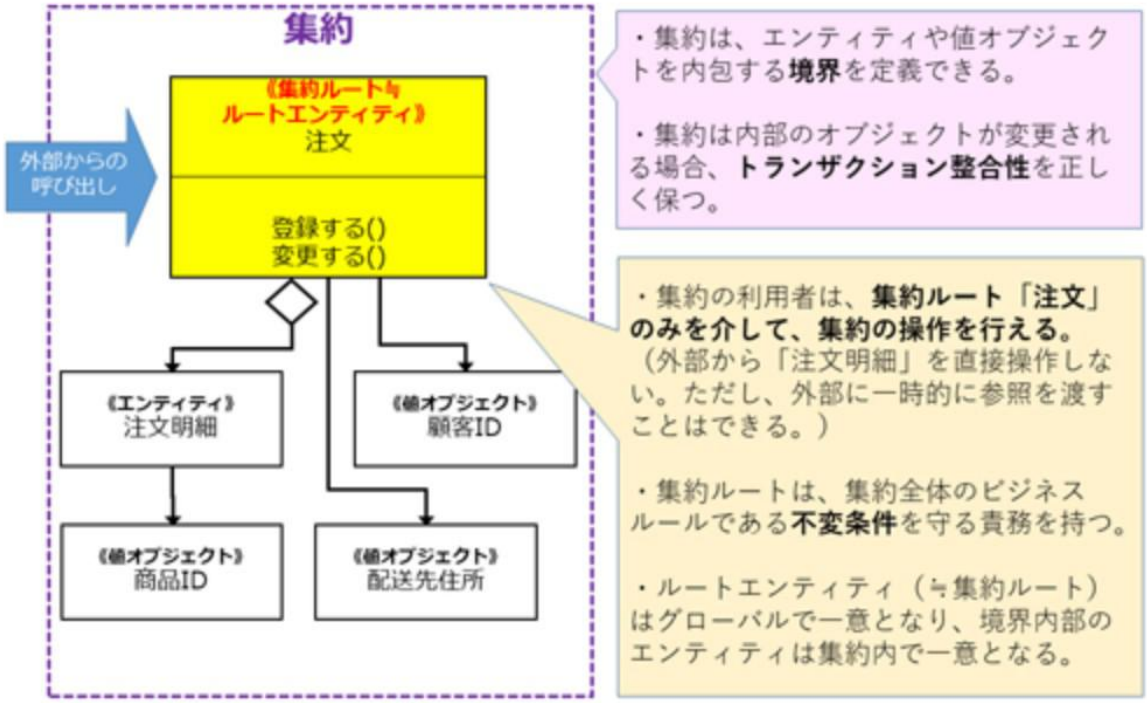
【実装例】

```
class DogToyEntity
{
    // おもちゃ種別VO
    private $toyType;

    // おもちゃ商品名
    private $toyName;

    // 数量
```

```
    private $number;

    // 価格VO
    private $priceVO;

    // 色VO
    private $colorVO;


    // Setterを実装
    public function __construct
    (
        ToyType $toyType,
        String $toyName,
        Int $number,
        priceVO $priceVO,
        ColorVO $colorVO
    )
    {
        $this->toyType = $toyType,
        $this->toyName = $toyName,
        $this->number = $number,
        $this->priceVO = $priceVO,
        $this->colorVO = $colorVO
    }


    // 自身と下位VOを用いて、集約を構成
    public static function aggregateDogToyEntity(Array $fetchedData)
    {
        return new DogToyEntity
        (
            new ColorVO($fetchedData['dog_toy_type']),
            $fetchedData['dog_toy_name'],
            $fetchedData['number'],
            new PriceVO($fetchedData['dog_toy_price']),
            new ColorVO($fetchedData['color_value'])
        );
    }


    // Getterを実装
    public function toyNameWithColor()
    {
        if($this->toyName && $this->colorVO->colorName())
        return sprintf
        (
            '%s（%s）',
            $this->toyName,
            $this->colorVO->colorName()
        )
    }

}
```

◆ RouteEntityとは

EntityやValueObjectからなる集約の中で、最終的にアプリケーション層へレスポンスされる集約を、『RouteEntity』という。



【実装例】

```
class ToyOrderEntity
{
    // 犬用おもちゃ
    private $dogToyEntity;

    // 猫用おもちゃ
    private $catToyEntity;

    // Setterを実装
    public function __construct
    (
        DogToyEntity $dogToyEntity,
        CatToyEntity $catToyEntity
    )
    {
        $this->dogToyEntity = $dogToyEntity;
        $this->catToyEntity = $catToyEntity;
    }

    // 自身と下位Entityを用いて、『RouteEntity』の集約を構成
    public static function aggregateToyOrderEntity
    (
        Array $fetchedData
    ): ToyOrderEntity
    {
        return new toyOrderEntity
        (
            DogToyEntity::aggregateDogToyEntity($fetchedData),
            CatToyEntity::aggregateCatToyEntity($fetchedData)
        );
    }
```

```
}
```

09. ValueObject

オブジェクトをValueObjectとしてモデリング／実装したいのならば、以下に条件を満たす必要がある。

【実装例】

```php
/**
 * 支払情報オブジェクト
 */
class PaymentInfoVO
{
    // 予め実装したImmutableObjectトレイトを用いて、プロパティの不変性を実現
    use ImmutableObject;

    /**
     * 支払いID
     *
```

```
     * @var PaymentId
     */
    private $PaymentId;

    /**
     * 支払い方法
     *
     * @var PaymentType
     */
    private $PaymentType;

    /**
     * 連絡先メールアドレス
     *
     * @var string|null
     */
    private $contactMail;

    /**
     * 金額
     *
     * @var Money
     */
    private $price;
```

◆ （1）ドメイン層の計測、定量化、説明

金額、数字、電話番号、文字列、日付、氏名、色など、一意に識別する必要がないユビキタス言語をオブジェクトモデルで表現する時に、ValueObjectとしてモデリング／実装すべき。例えば、電話番号を表すために、Int型ではなく、PhoneNumber型で定義する。

◆ （2）プロパティの不変性

生成されたインスタンスのプロパティは変更されない。オブジェクトにSetterを持たせずに、`__construct()`のみでしか値を設定できないように実装すれば、『Immutable』なオブジェクトを実現できる。

【実装例】

```
class Test02 {

    private $property02;

    // コンストラクタで$property02に値を設定
    public function __construct
    (
        $property02
    )
    {
        $this->property02 = $property02;
    }

}
```

- **『Immutable』を実現できる理由**

Test01クラスインスタンスの`$property01`に値を設定するためには、インスタンスからSetterを呼び出す。Setterは何度でも呼び出せ、その度にプロパティの値を上書きできてしまう。

```
$test01 = new Test01

$test01->setProperty01("プロパティ01の値")

$test01->setProperty01("新しいプロパティ01の値")
```

一方で、Test02クラスインスタンスの`$property02`に値を設定するためには、インスタンスを作り直さなければならない。つまり、以前に作ったインスタンスの`$property02`の値は上書きできない。Setterを持たせずに、`__construct()`だけを持たせれば、『Immutable』なオブジェクトとなる。

```
$test02 = new Test02("プロパティ02の値")

$test02 = new Test02("新しいプロパティ02の値")
```

◆ （3）概念的な統一体

```

```

◆ （4）オブジェクトの交換可能性

オブジェクトが新しくインスタンス化された場合、以前に同一オブジェクトから生成されたインスタンスから新しく置き換える必要がある。

◆ （5）オブジェクト間の等価性

全てのプロパティの値が他のVOと同じ場合、同一のVOと見なされる。

◆ （6）メソッドによってオブジェクトの状態が変わらない

【実装例1】

```
// （1）ドメイン層の氏名を扱うVO
class NameVO
{
   // （2）予め実装したImmutableObjectトレイトを用いて、プロパティの不変性を実現
    use ImmutableObject;

    // 苗字プロパティ
    private $lastName;
```

```
    // 名前プロパティ
    private $firstName;

    // （6） メソッドによってオブジェクトの状態が変わらない
    public function fullName(): String
    {
        return $this->lastName . $this->firstName;
    }

    //
    protected static function computedPropertyNames()
    {
        return [
            'fullName'
        ];
    }
}
```

【実装例2】

同様に、Immutableトレイトを基に、VOを生成する。

```
// ドメイン層の金額を扱うVO
class Money
{

}
```

◆ EntityとValueObjectのどちらとして、モデリング／実装すべきなのか？

EntityとValueObjectのどちらとして、オブジェクトモデルをモデリング／実装すべきなのかについて考える。そもそも、大前提として、『オブジェクトモデルはできるだけ不変にすべき』というベストプラクティスがあり、その結果、ValueObjectというが生まれたと考えられる。実は、ValueObjectを使わずに全てEntityとしてモデリング／実装することは可能である。しかし、不変にしてもよいところも可変になり、可読性や信頼性を下げてしまう可能性がある。

10. TypeCodeObject

◆ TypeCodeObjectとは

一意に識別する必要がないユビキタス言語の中でも、特に『区分』や『種類』などは、ValueObjectとしてではなく、TypeCodeObjectとしてモデリング／実装する。VOまたはEnumによって実装する。

◆ EnumによるTypeCodeObjectの実装

【実装例1】

```
class ColorVO extends Enum
{
    const RED = '1'
    const BLUE = '2'

    // 『self::定数名』で、定義の値へアクセスする。
    private $defs = [
        self::RED => ['colorname' => 'レッド'];
        self::BLUE => ['colorname' => 'ブルー'];
    ];

    // 色値プロパティ
    private $colorValue

    // 色名プロパティ。
    private $colorName;


    // インスタンス化の時に、『色の区分値』を受け取る。
    public function __construct
    (
        String $value
    )
    {
        // $kbnValueに応じて、色名をcolornameプロパティにセットする。
        $this->colorValue = $value;
        $this->colorname = $this->defs[$value]['colorName'];
    }


    // constructによってセットされた色値を返すメソッド。
    public function colorValue() :Int
    {
        return $this->colorValue;
    }


    // constructによってセットされた色名を返すメソッド。
    public function colorName() :String
    {
        return $this->colorName;
    }

}
```

11. Id

- **実装例**

12. Service

他の３区分に分類できないもの（例：Id-Aを生成するId-B）。

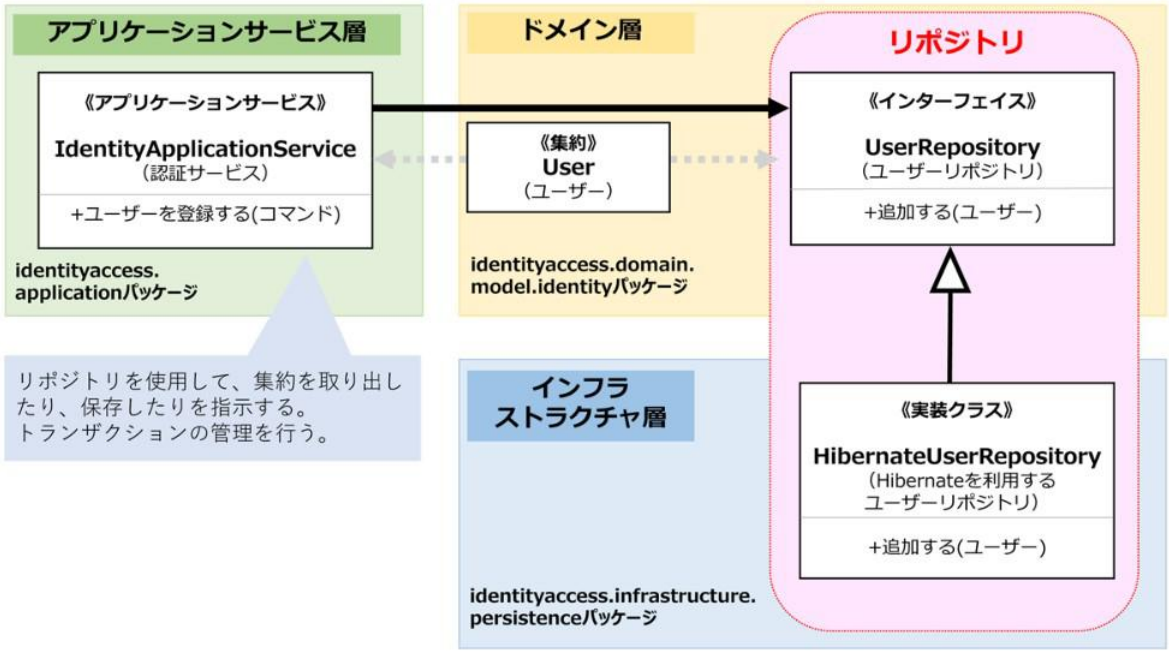
13. Repository

◆ Repositoryとは

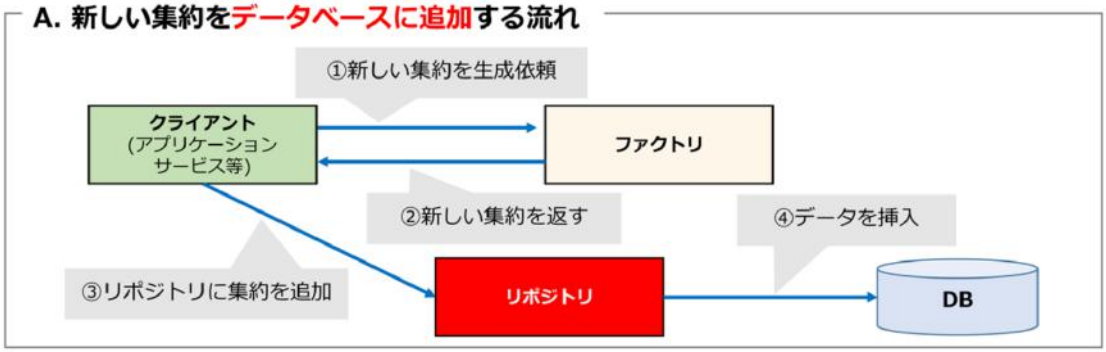
構成したい集約とRepositoryは、一対一の関係になる。例えば、OrderのRouteEntityからなる集約を構成するRepositoryは、OrderRepositoryと名付ける。

◆ 抽象クラス

DIPに基づくドメイン駆動設計の場合、Repositoryの抽象クラスを配置する。



◆ 集約構成とデータ挿入

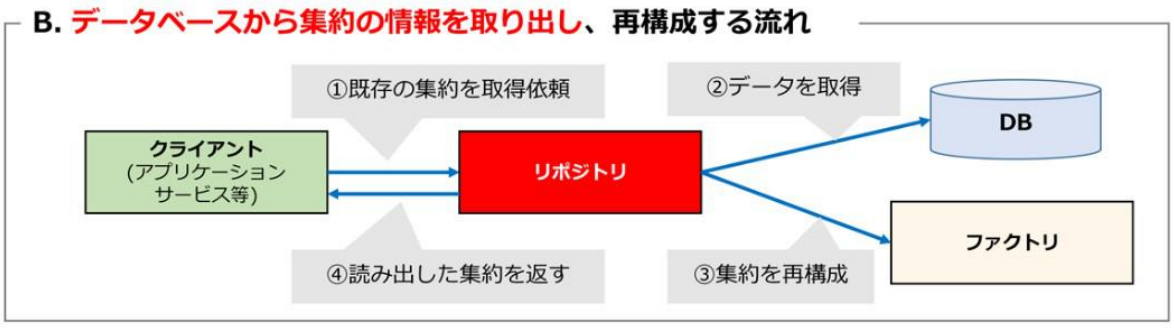
1. アプリケーション層から集約がリクエストされる。
2. ファクトリの呼び出しによって、EntityやValueObjectから構成される集約を新しく構成
3. データベースに追加する。
4. 

【実装例】

```
// 集約を新しく構成するRepository
class ToyOrderRepository
{
    // データベースへの挿入の前処理として、カラム名を値に持つ連想配列で定義
    $data = [
        'type' =>
        'name' =>
        'number' =>
        'price' => ($order->priceVO) ? $order->priceVO->amount() : 0,
        'color_value' => ($order->colorVO) ? $order->colorVO->value() : null,
    ]

    return $data;
}
```

◆ データ取得と集約再構成

1. データベースからデータを取得。
2. ファクトリの呼び出しによって、EntityやValueObjectから構成される集約を加工し、再構成する。
3. 再構成された集約をアプリケーション層にレスポンス。



【実装例】

```
// データ取得と集約再構成を行うRepository
class ToyOrderRepository
{

    // データベースからデータを取得
    public function fetchDataSet()
    {
        $select = [
            'dog_toy.type AS dog_toy_type',
            'dog_toy.name AS dog_toy_name',
            'dog_toy.number AS number',
            'dog_toy.price AS dog_toy_price,
            'color.color_value AS color_value'
        ];
        $query = $this->getFecthQuery($select);
        return $query->getConnection()->executeQuery()->fetchAll();
    }
```

```
    // 『RouteEntity』の配列をアプリケーション層へレスポンス
    public function arrayedToyOrderEntities(): ToyOrderEntities
    {
        $toyOrderEntities = [];
        foreach($this->fetchDataSet() as $fetchedData){
            $toyOrderEntities[] =
ToyOrderEntity::aggregateToyOrderEntity($fetchedData)
        }
        return $toyOrderEntities;
    }

}
```

14. Factory

◆ Factoryとは

責務として、構成した集約関係を加工して新sたな集約を再構成する。

- **実装例**

```
class Factory
{
    private $factory

    public function __construct
    (
        Factory $factory
    )
    {
        $this->$factory = $factory
    }

    public function ToyInstance()
    {
        isset($this->factory){
            //なんらかの集約処理;
        }
    }
}
```

15. Controller

責務として、ドメイン層の抽象メソッドを用いて、Use case（使用事例）を実装する。

【具体例】

オンラインショッピングにおけるUse case

オンラインショッピングのユースケース図