ユーザー その2

ユーザーの概要

ユーザーでできる事

商品一覧・商品検索
商品詳細・店舗詳細
商品をカートに保存
購入(Stripe API)
購入確認のメール

基本設計リンク

URL設計、テーブル設計、機能設計
(Googleスプレッドシート)
https://docs.google.com/spreadsheets/d/1YlDqTKH2v2-n97kb2GNhWrcMGnJD84JMqTuzD_poMqo/edit?usp=sharing

ER図(draw.io)
https://drive.google.com/file/d/18sEk5LC-jJum-NU9JKNZibGRVX81aWE1/view?usp=sharing

商品一覧
ローカルスコープ

商品一覧を表示する条件

表示する条件
Shop ・・ is_selling = true
Product ・・ is_selling = true
Stockの合計 ・・ 1以上

商品一覧クエリ 現状1

select内でsumを使うため
クエリビルダのDB::rawで対応

```
$stocks = DB::table('t_stocks')
        ->select('product_id',
        DB::raw('sum(quantity) as quantity'))
        ->groupBy('product_id')
        ->having('quantity', '>', 1);
```

商品一覧クエリ 現状2

前ページの $stocksをサブクエリとして設定
products、shops、stocksをjoin句で紐付けて
where句で is_sellingがtrue かの条件指定

```
$products = DB::table('products')
        ->joinSub($stocks, 'stock', function($join){
            $join->on('products.id', '=', 'stock.product_id');
        })
        ->join('shops', 'products.shop_id', '=', 'shops.id')
        ->where('shops.is_selling', true)
        ->where('products.is_selling', true)
        ->get();
```

商品一覧クエリ 現状3

Eloquent->クエリビルダに変更したためselectで指定

```
$products = DB::table(‘products')
        略
        ->join('secondary_categories', 'products.secondary_category_id', '=',
'secondary_categories.id')
        ->join('images as image1', 'products.image1', '=', 'image1.id')
        ->join('images as image2', 'products.image2', '=', 'image2.id')
        ->join('images as image3', 'products.image3', '=', 'image3.id')
        ->join('images as image4', 'products.image4', '=', 'image4.id')
        略 (前ページのwhere句)
        ->select('products.id as id', 'products.name as name', 'products.price'
        ,'products.sort_order as sort_order'
        ,'products.information', 'secondary_categories.name as category'
        ,'image1.filename as filename')
        ->get();
```

商品一覧 ローカルスコープ

何度も使うクエリは1箇所にまとめておきたい
コントローラをできるだけシンプルにしたい

モデル Product

```
Public function scopeAvailableItems($query)
{
  $stocks = 略;

  return $query->joinSub(略 〜:
   ※getは書かない
}
```

商品一覧 ローカルスコープ

コントローラ側

```
$products = Product::availableItems()->get()
```

1行で収まる

商品詳細
コンストラクタ

URLに直接入力した場合を想定

```
$this->middleware(function ($request, $next) {
    $id = $request->route()->parameter('item');
    if(!is_null($id)){
    $itemId = Product::availableItems()-
>where('products.id', $id)->exists();
        if(!$itemId){
            abort(404); // 404画面表示
        }
    }
    return $next($request);
});
```

表示順

表示順

ECサイトには必須といってもいい機能

おすすめ順、高い順、安い順、新しい順、古い順・・・

大規模なECサイトになるほど
上位表示にお金がかかる
(広告枠 1週間で◯万円など)

表示順 定数

```
const ORDER_RECOMMEND = '0';
const ORDER_HIGHER = '1';
const ORDER_LOWER = '2';
const ORDER_LATER = '3';
const ORDER_OLDER = '4';

const SORT_ORDER = [
  'recommend' => self::ORDER_RECOMMEND,
  'higherPrice' => self::ORDER_HIGHER,
  'lowerPrice' => self::ORDER_LOWER,
  'later' => self::ORDER_LATER,
  'older' => self::ORDER_OLDER
];
```

表示順 ローカルスコープ

```
public function scopeSortOrder($query, $sortOrder){
        if($sortOrder === null || $sortOrder === \Constant::SORT_ORDER[‘recommend']){
           return $query->orderBy('sort_order', 'asc') ;
        }
        if($sortOrder === \Constant::SORT_ORDER['higherPrice']){
            return $query->orderBy('price', 'desc') ;
        }
        if($sortOrder === \Constant::SORT_ORDER['lowerPrice']){
        return $query->orderBy('price', 'asc') ;
        }
        if($sortOrder === \Constant::SORT_ORDER['later']){
        return $query->orderBy('products.created_at', 'desc') ;
        }
        if($sortOrder === \Constant::SORT_ORDER['older']){
        return $query->orderBy('products.created_at', 'asc') ;
        }
    }
```

表示順 コントローラ

ItemController

```
public function index(Request $request)
{
$products = Product::availableItems()
        ->sortOrder($request->sort)
        ->get();
略
}
```

表示順 ビュー その1

```
<form method="get" action="{{ route('user.items.index')}}">

<span class="text-sm">表示順</span><br>
<select id="sort" name="sort">
  <option value="{{ \Constant::SORT_ORDER['recommend']}}"
  @if(\Request::get('sort') === \Constant::SORT_ORDER['recommend'] )
  selected
  @endif>おすすめ順
  </option>
  略
</select>
</form>
```

表示順 ビュー その2

```
<script>
   const select = document.getElementById('sort')
   select.addEventListener('change', function(){
        this.form.submit()
    })
</script>
```

表示件数

表示件数 ビュー その1

20件、50件、100件の3択

```
<span class="text-sm">表示件数</span><br>
<select id="pagination" name="pagination">
  <option value="20"
  @if(\Request::get('pagination') === '20')
  selected
  @endif>20件
  </option>
   略
</select>
```

表示件数 ビュー その2

paginationでgetパラメータの引き継ぎ

```
{{  $products->appends([
  'sort' => \Request::get('sort'),
  'pagination' => \Request::get('pagination'),
  ])->links() }}

<script>
const paginate = document.getElementById('pagination')
paginate.addEventListener('change', function(){
        this.form.submit()
    })
</script>
```

表示件数 コントローラ

ItemController

```
$products = Product::availableItems()
  ->sortOrder($request->sort)
  ->paginate($request->pagination);
```

修正

```
->paginate($request->pagination ?? '20');
```

検索フォーム

検索フォーム

カテゴリ + キーワード
以前作成した表示順・表示件数も
まとめてgetパラメータで受け渡す

<form action=“get”>
 カテゴリー
 キーワード
 検索ボタン
 表示順
 表示件数
</form>

検索フォーム ビュー側

```
<h2></h2>
<form>
 <div class=“lg:flex lg:justify-around”>
   <div class=“lg:flex items-center”>
     <select>カテゴリー</select>
     <div class=“flex”>
       <div><input placeholder=“キーワードを入力”></div>
       <div><button>検索する</button></div>
     </div>
   </div>
   <div class=“flex”>
     略 (表示順・表示件数)
   </div>
 </div>
</form>
```

カテゴリー ビュー側

```
<select name="category">
  <option value="0" @if(\Request::get('category') === ‘0’) selected
@endif >全て</option>
  @foreach($categories as $category)
    <optgroup label="{{ $category->name }}">
    @foreach($category->secondary as $secondary)
      <option value="{{ $secondary->id}}"
@if(\Request::get('category') == $secondary->id) selected @endif >
        {{ $secondary->name }}
      </option>
    @endforeach
@endforeach
</select>
```

カテゴリー ローカルスコープ

モデル側

```
public function scopeSelectCategory($query, $categoryId)
    {
        if($categoryId !== '0')
        {
            return $query->where('secondary_category_id', $categoryId);
        } else {
            return;
        }
    }
```

コントローラ側

```
$products = Product::availableItems()
        ->selectCategory($request->category ?? '0')
        ->sortOrder($request->sort)
         略
```

キーワード ローカルスコープ

モデル側

```
public function scopeSearchKeyword($query, $keyword)
    {
        if(!is_null($keyword))
        {
            $spaceConvert = mb_convert_kana($keyword,'s'); //全角スペースを半角に
            $keywords = preg_split('/[\s]+/', $spaceConvert,-1,PREG_SPLIT_NO_EMPTY); //空白で区切る
            foreach($keywords as $word) //単語をループで回す
            {
                $query->where('products.name','like','%'.$word.'%');
            }
            return $query;
        } else {
            return;
        }
    }
```

コントローラ側　->searchKeyword($request->keyword)

メール関連

メール

商品を購入時、ユーザーオーナーそれぞれに通知
ログインパスワード忘れ時の連絡など

レンタルサーバーや
パブリッククラウド(AWSなど)側にメールサーバーが存在

環境によって設定が変わる(.envを編集)
MAIL_DRIVER, MAIL_HOST
MAIL_PORT, MAIL_USERNAME,
MAIL_PASSWORD など

mailtrap.io

メール送信のテスト用サイト
https://mailtrap.io/

Googleアカウントで登録

Laravel設定用のコードを.envにコピー

php artisan config:cache
でキャッシュを削除

メールの設定ファイル

Config/mail.php

メーラー設定 初期値はsmtp
(simple mail transfer protocol)

グローバルFromアドレスは.envファイルに追記

Mailableクラス テストメール生成

php artisan make:mail TestMail

App/Mail/TestMail.php が生成される

```
public function build()
 {
    return $this
   ->subject(‘テスト送信完了') //タイトル
    ->view(‘emails.test'); //本文
 }
```

テストメール

ビュー側 views/emails/test.blade.php
メール本文です。
メール本文です。

テストメール

コントローラ側
テストとして User/ItemController@index
に追記

```
use Illuminate\Support\Facades\Mail;
use App\Mail\TestMail;

Mail::to('test@example.com') //受信者の指定
->send(new TestMail()); //Mailableクラス
```

非同期処理
(キュー & ジョブ & ワーカー)

非同期処理

メール送信には時間がかかる

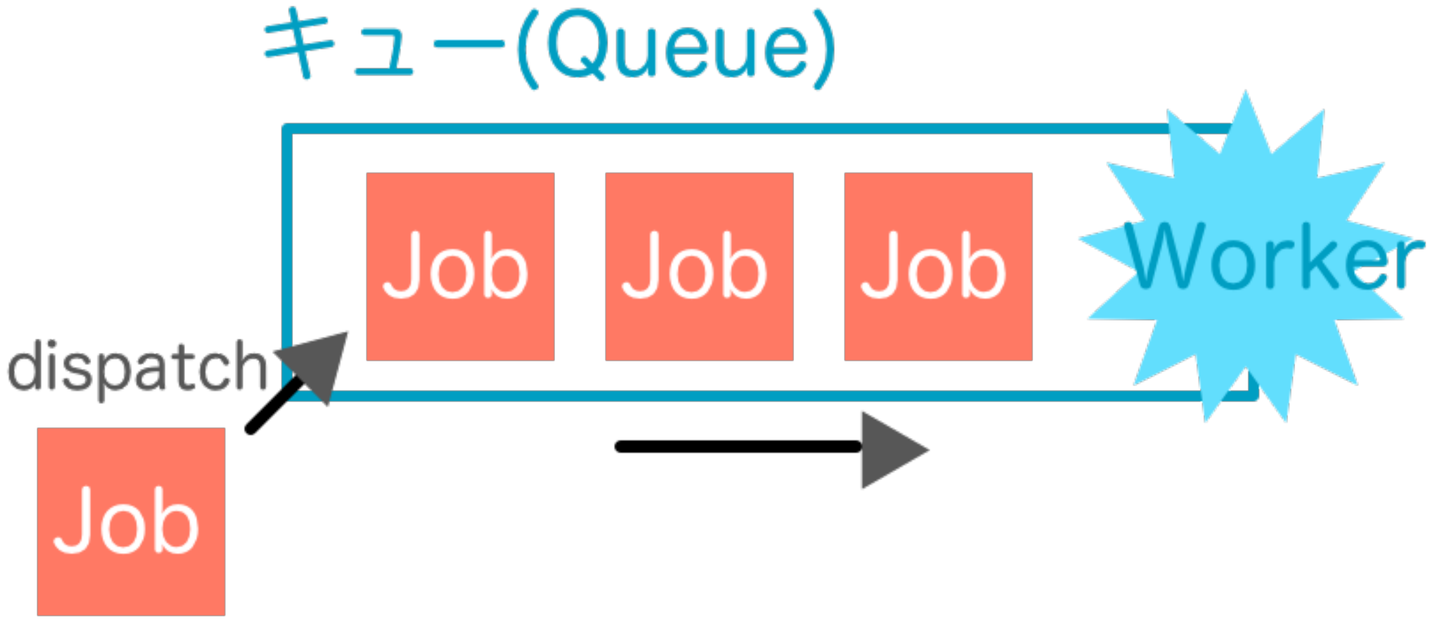
同期処理・・送信してから画面更新(3～4秒)

非同期処理・・画面上は更新しつつ(0,1秒～)、裏側でメール送信

->キューで対応

非同期処理 簡易図

Queue (キュー)・・待ち行列
Job (ジョブ)・・1つ1つの処理
Worker (ワーカー)・・処理をする人



キューの設定

マイグレーション failed_jobs_table

設定ファイル config/queue.php

sync・・同期
(他に database, redis, beanstalkd, sqsなど)
今回は databaseに変更

.envファイルを下記に書き換える
QUEUE_CONNECTION=database

php artisan config:cache でキャッシュ削除

キューの保存場所がDBになる

ジョブ用のテーブル生成

ジョブを保存するテーブルを生成
php artisan queue:table
(これで jobsというテーブルが生成される)

php artisan migrate
(DB内にテーブル生成)
キューを使うとこのテーブルに
未実行のジョブが溜まっていく

ジョブクラスの作成

ジョブの作成
Php artisan make:job SendThanksMail

App\Jobs\SendThanksMail.php
の中にメール送信設定を追加

```
use Illuminate\Support\Facades\Mail;
use App\Mail\TestMail;

public function handle()
{
  Mail::to('test@example.com')->send(new TestMail());
}
```

ジョブのdispatch->キュー

User/ItemController@index

```
use App\Jobs\SendThanksMail;

// キューにジョブを入れて処理(非同期)
SendThanksMail::dispatch();
```

これで非同期処理になる
表側(画面)はすぐに更新される
裏側でキューにジョブが入っていく
(phpMyAdminでjobsテーブルを見てみる)

ワーカーの起動

ワーカー(Worker 処理をする人)

php artisan queue:work

キューに入ったジョブを裏側で処理してくれる

ワーカープロセスが常に動いている必要がある
・・監視しておく必要がある

本番環境(Linux)の場合は supervisor で監視設定必要

改めての
メール設定

商品購入後の流れ

1. カート情報を取得
2. カートから商品を削除
3. ユーザー向けのメール(Job)
4. オーナー向けのメール(複数Job)

カート情報から下記を取得
(商品情報、在庫、オーナー(名前・メールアドレス)
-> コード量が増えるので
App\Services\CartService@getItemsIncart を作成

CartServiceファイル

```
<?php
namespace App\Services;
use App\Models\Product;
use App\Models\Cart;

class CartService
{
  public static function getItemsInCart($items)
  {
    $products = []; //空の配列を準備

    foreach($items as $item){ // カート内の商品を一つずつ処理
     略(次ページ)
    }
      return $products; // 新しい配列を返す
   }
}
```

カート情報を元に配列を作成

```
foreach($items as $item){
 $p = Product::findOrFail($item->product_id);
 $owner = $p->shop->owner->select('name', 'email')->first()->toArray();//オーナー情報
 $values = array_values($owner); //連想配列の値を取得
 $keys = ['ownerName', 'email'];
 $ownerInfo = array_combine($keys, $values); // オーナー情報のキーを変更
 $product = Product::where('id', $item->product_id)
                ->select('id', 'name', ‘price')->get()->toArray(); // 商品情報の配列
 $quantity = Cart::where('product_id', $item->product_id)
                ->select(‘quantity')->get()->toArray(); // 在庫数の配列
 $result = array_merge($product[0], $ownerInfo, $quantity[0]); // 配列の結合
 array_push($products, $result); //配列に追加
}
```

CartController

checkoutメソッドに一旦追加

```
use App\Services\CartService;
略
public function checkout()
{
  $items = Cart::where('user_id', $user->id)->get();
  $products = CartService::getItemsInCart($itemsInCart);
  略
}
```

ユーザー向けにメール送信

ユーザー向け
 メールクラス ThanksMail
 ジョブクラス SendThanksMail

メール生成
php artisan make:mail ThanksMail

ジョブ
php artisan make:job SendThanksMail (生成済み)

ユーザー情報、商品情報(変数)を受け渡す

Controller

```
$user = 略;

Job($user,
$products);
```

Job

```
Class{

public $user;
Public $products,

public function __construct($user,
$products)
{
  $this->user = $user;
  $this->products = $products;
}

public function handle()

  Mail::to($this->user)
  ->send(new ThanksMail($this->user,
  $this->products));
}
```

Mailable

```
Class{

public $user;
Public $products;

public function __construct($user,
 $products)
{
  $this->user = $user;
  $this->products = $products;
}

public function build()

  return $this->view('emails.thanks')
  ->subject();

}
```

Blade

```
{{ $user->name }}

@foreach($products
as $product)

{{ $product['name'] }}
略

@endforeach
```

オーナー向けにメール送信 その1

複数の商品・・オーナーが別
->それぞれにメール送信

メール php artisan make:mail OrderedMail
ジョブ php artisan make:job SendOrderedMail

オーナー向けにメール送信 その2

CartController

```
use App\Job\SendOrderedMail;
略

foreach($products as $product)
{
  SendOrderedMail::dispatch($product, $user);
}
```

コントローラ->ジョブ->メール->ブレードの流れは
ユーザー向けと同じ

動作確認できたら

CartController
checkoutメソッド内からSuccessメソッドに移動

```
pubilc function success()
{
$products = CartService::getItemsInCart($itemsInCart);
SendThanksMail::dispatch($products, $user);

        foreach($products as $product){
            SendOrderedMail::dispatch($product, $user);
        }

Cart::where('user_id', Auth::id())->delete();
略
}
```

その他

その他の対応

新規ユーザー登録後の
リダイレクト先を / に変更
RouteServiceProvider.php

README.mdに
stripe, mailtrap, queueワーカーを追記