University of Technology, Jamaica

Faculty: Engineering and Computing

School: School of Computing and Information and Technology

Academic Year: 2018/19

Semester: 2

Module: Advance Programming

Research project on John Shop User Manual

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tutor: Mr. Gilroy Gordon

Table of Contents

ERD1

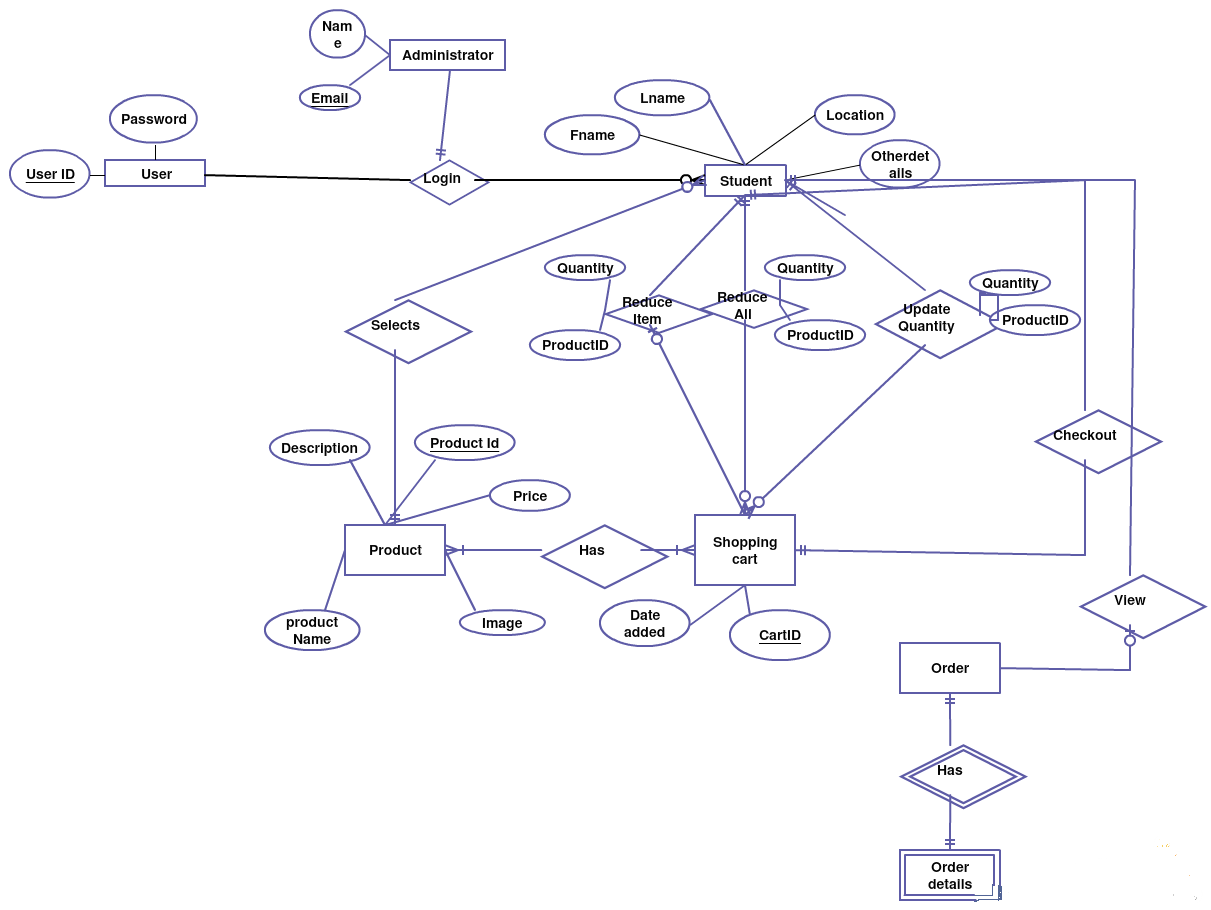
Class Diagram2

SOLID Principles-4-6

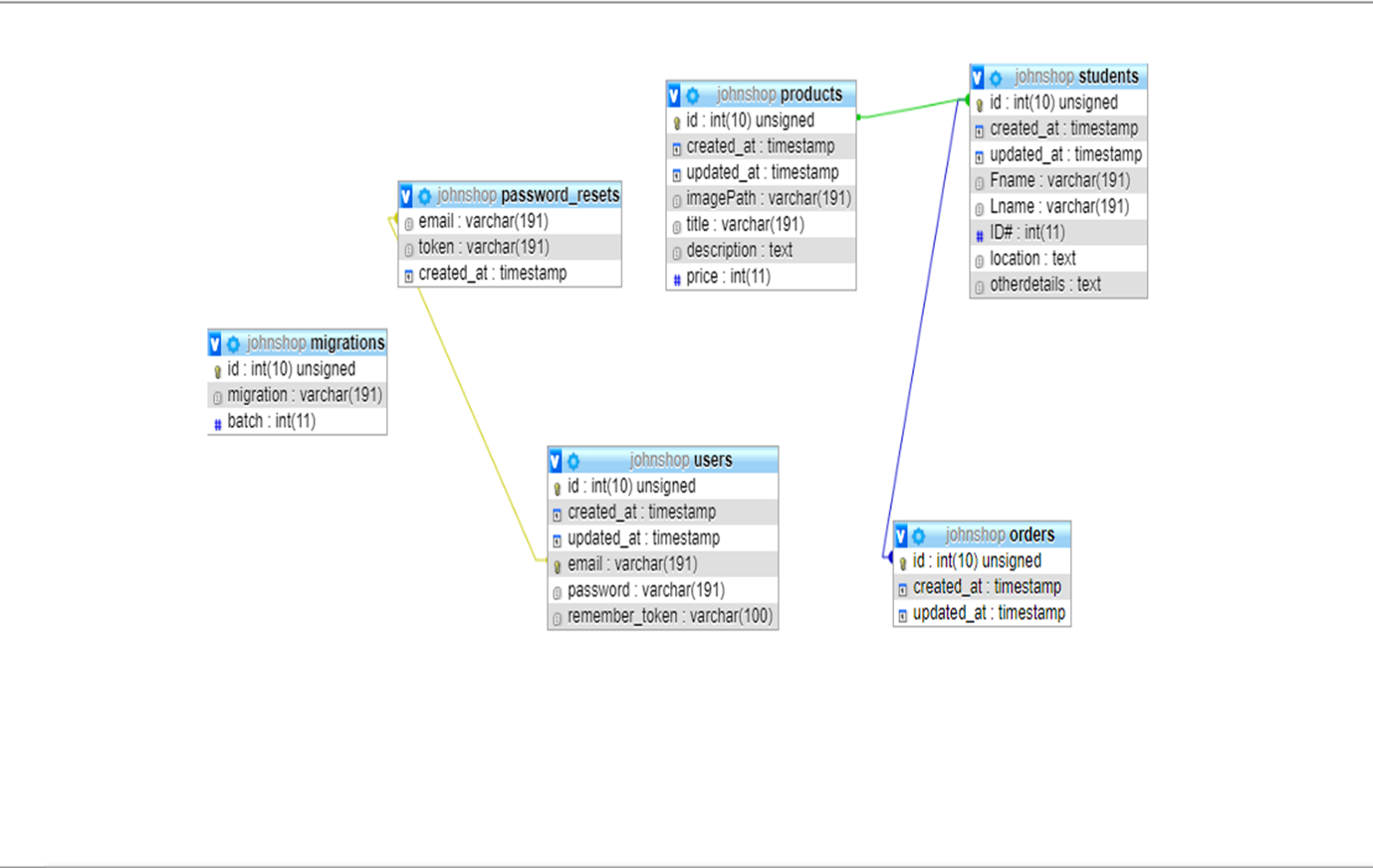
Design Patterns7-9

Reference10

ERD Diagram



Class Diagram



SOLID PRINCIPLES

class Authenticate extends Middleware

{

/\*\*

\* Get the path the user should be redirected to when they are not authenticated.

\*

\* @param \Illuminate\Http\Request $request

\* @return string

\*/

protected function redirectTo($request)

{

return route('login');

}

}

Fig 1.

In this figure we can see that the SOLID Principle, Single Responsibility is taking place. Where this class and methods only has one responsibility which is to authenticate a request and redirect it. Single responsibility was incorporated in my project from the SOLID principles where for a user’s to login functionality had only one single responsibility.

class DatabaseSeeder extends Seeder

{

/\*\*

\* Seed the application's database.

\*

\* @return void

\*/

public function run()

{

// $this->call(UsersTableSeeder::class);

$this->call(ProductTableSeeder::class);

}

}

Fig 2

In this figure SOLID Principles were again seen for OPEN/CLOSED where it is open for extension and closed for modification. It is also inherits and extends to add more functionality.

public function getAddToCart(Request $request ,$id)

{

$product = $this->productRepo->get($id);

$oldCart = Session::has('cart') ? Session::get('cart') : null;

$cart = new Cart ($oldCart);

$cart->add($product, $product->id);

$request->session()->put('cart', $cart);

return redirect()->route('product.index1');

}

Fig 3’

Liskov Subsitution has been used for objects to be replaced without altering the correctness of the program.

class ProductRepository implements IProductRepository{

public function get($id){ //where the function get only has one responsibility

return Product::find($id);

}

}

Fig 4

Use of Interface segregation as a solid principle as it helps in separation of behavior.

Design Patterns

interface IProductRepository{

public function get($id);

}

Fig 5

class ProductRepository implements IProductRepository{

public function get($id){ //where the function get only has one responsibility

return Product::find($id);

}

}

Fig 6

Singleton Pattern

Figure 6 Illustrate the Singleton Pattern within the program as it allows us to utilize only one instance “$id”of the function especially for access to the database.

Factory Pattern

Figure 5 and 6 both has the factory pattern design as the creation of the object in IProductRepository interface is then delegated to the ProductRepository class. It allows for changes to be made only to the factory function and not to the classes. Factory Pattern was not incorporated as much because it isn’t as feasible with laravel php and with this project because of amount classes and superclasses.

class ProductController extends Controller

{

/\*\*

\* @instance of App\Contracts\Repos\IProductRepository

\*/

private $productRepo;

public function \_\_construct(

IProductRepository $productRepo

){

$this->$productRepo = $productRepo;

}

Fig 7

Repository Pattern

Repository pattern was also incorporated in the project and is illustrated in figure 7 where it takes the IProductRepository as a parameter and return the products. It minimizes the duplication and makes unit testing easier.

Model View Controller Pattern

This pattern was incorporated in the project throughout because of the nature of the project because it requires a lot of usage from the client side as to the back end of the program and such incorporating a mvc framework worked really well in terms of linking the models and view together from the controllers functionality.

Source Control Tools

The tools use for the source control was the tool github. It was feasible for the project as it adds continuous integration to the project.

Package Management Tool

The framework use was laravel which utilizes its main language php and there the package management tool use from the installation was composer. It helps in keeping the code up to date and helps in importing code bases.

class ExampleTest extends TestCase

{

/\*\*

\* A basic test example.

\*

\* @return void

\*/

public function testBasicTest()

{

$this->assertTrue(true);

}

}

Fig 8.

Unit Testing

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box and a phpunit.xml file is already set up for your application. The framework also ships with convenient helper methods that allow you to expressively test your applications

References

* Media, T. (n.d.). https://www.youtube.com/user/TechGuyWeb.
* Miller, D. (n.d.). Retrieved from https://startbootstrap.com/templates/.
* OTWELL, T. (n.d.). Retrieved from <https://laravel.com/docs/5.8>.
* Shvets, A. (2019). Design Patterns. Retrieved from <https://refactoring.guru/design-patterns>
* Dixon,D.(n.d)<https://www.youtube.com/playlist?list=PLhxWTTvNWTRQBjnUf5VoqE2ZQo93RofFG>
* Dixon, D. (n.d.). https://github.com/daniel470/Johnshop/tree/master/Design. Retrieved from <https://github.com>.