

John Shop Research Project Report

Kedishia Soares 1607294

Javon Moatt 1300168

Advanced Programming

School of Computing and Information Technology

University of Technology, Jamaica

## Table of Contents

Topic	Page #
SOLID Principles:	3-6
Single Responsibility Principle	3-4
Open-Close Principle	4
Liskov Substitution Principle	5
Interface Segregation Principle	5
Dependency Inversion Principle	6
Repository Pattern	7
Model-View-Controller Pattern	8-10
Singleton Pattern	11
Factory Pattern	12
Code Generation Tool	13
Source Control Management Tool	14
Package Management Tool	15
Unit Testing and Test Automation	16
Continuous Integration Using a Continuous Integration Server	17
References	18-19
Appendices	20-22

## SOLID Principles

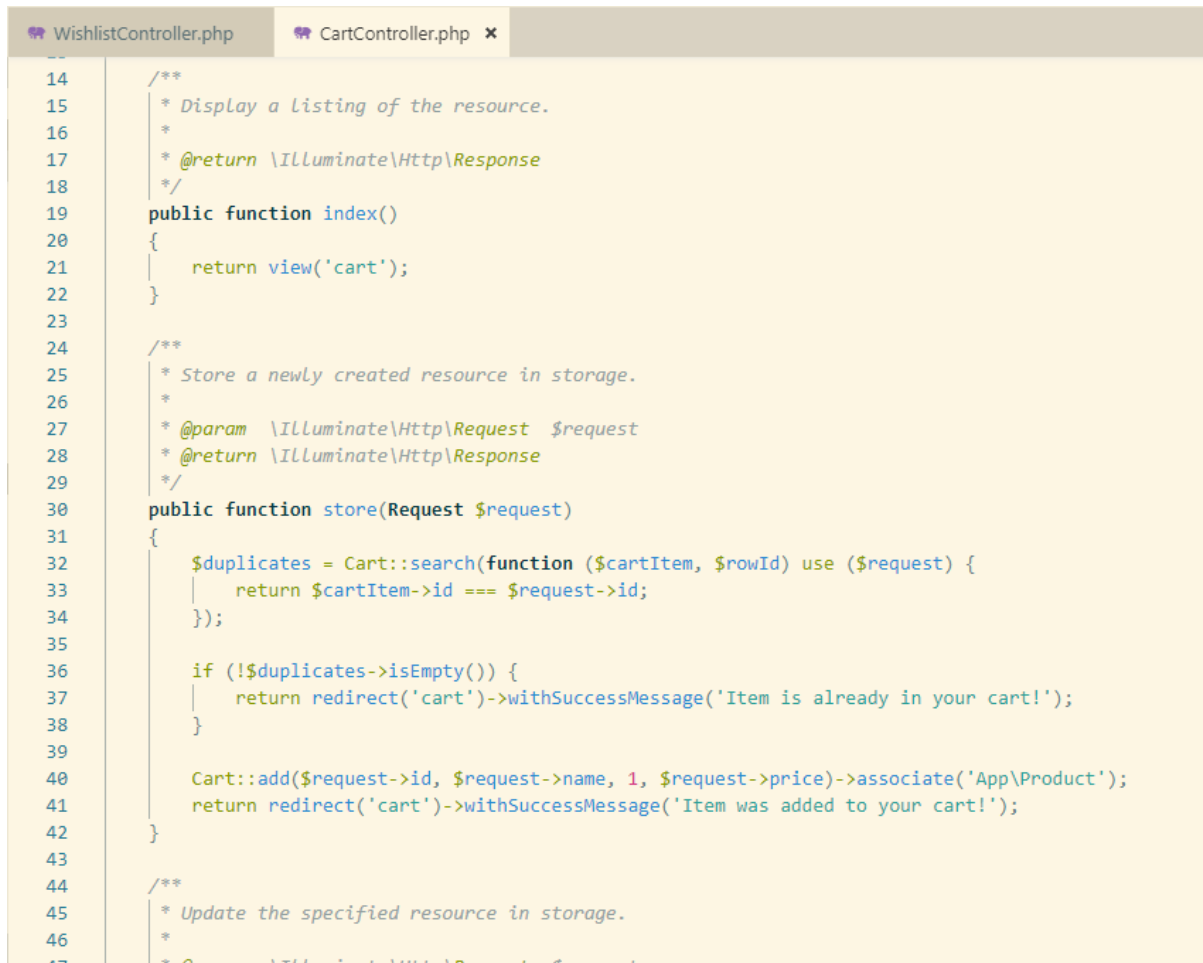
The SOLID principles are a set of software design principles used in Object Oriented Programming which aids in the flexibility and maintainability of code. SOLID is an acronym for:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principles
- Interface Segregation Principle
- Dependency Inversion Principle

### Single Responsibility

The Single Responsibility principle allows for separation of concerns as it speaks to ensuring that each module is only given one responsibility (Gordon, 2019). For example, in the creation of a calculator software, there must be a module for each function of the calculator, namely addition, subtraction, multiplication, and division, etc.

This is demonstrated in this application through the use of controllers and methods. Each function of the website is divided into controllers. Each controller is then given the responsibility to control a single aspect of the website. Controllers are further divided into methods which control a specific task of the web application. This ensures that no method is given multiple responsibilities.



```

14  /**
15   * Display a listing of the resource.
16   *
17   * @return \Illuminate\Http\Response
18   */
19  public function index()
20  {
21      return view('cart');
22  }
23
24  /**
25   * Store a newly created resource in storage.
26   *
27   * @param \Illuminate\Http\Request $request
28   * @return \Illuminate\Http\Response
29   */
30  public function store(Request $request)
31  {
32      $duplicates = Cart::search(function ($cartItem, $rowId) use ($request) {
33          return $cartItem->id === $request->id;
34      });
35
36      if (!$duplicates->isEmpty()) {
37          return redirect('cart')->withSuccessMessage('Item is already in your cart!');
38      }
39
40      Cart::add($request->id, $request->name, 1, $request->price)->associate('App\Product');
41      return redirect('cart')->withSuccessMessage('Item was added to your cart!');
42  }
43
44  /**
45   * Update the specified resource in storage.
46   *
47   * @param \Illuminate\Http\Request $request

```

### Open/Closed Principle

This principle states that all software entities must be open for extension but closed for modification. According to Martin (2000), this can be explained as being able to write code that allows you to change what the module does without altering the source code. This is often achieved through the use of polymorphism; codes should utilize inheritance and the implementation of interfaces (Hoiberg, n.d.).

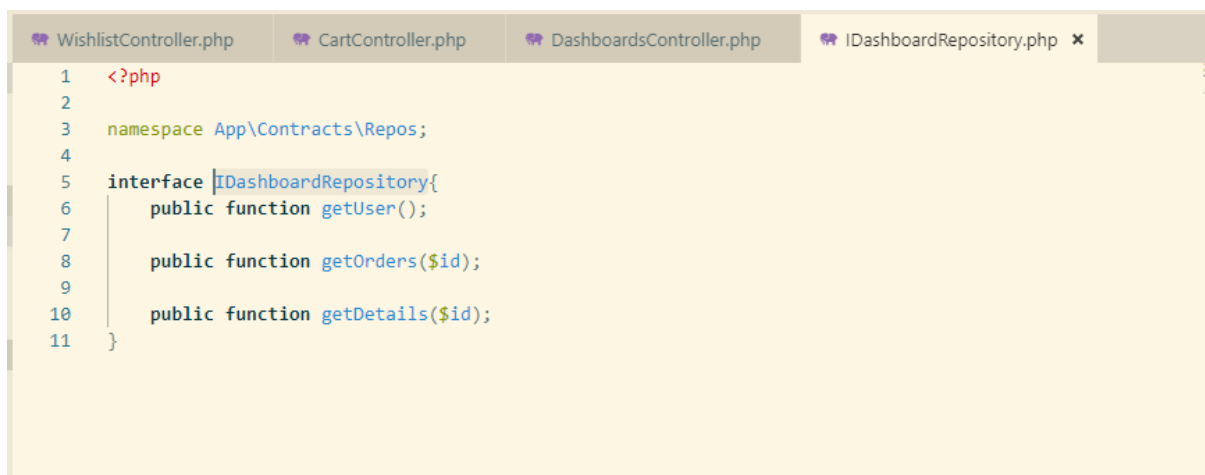
One example of this throughout the John Shop web application comes in the form of the main view. The code has a main view called a template and all other views inherit from this view. This not only preserves time and space, as only the specific content for each page is needed, but it also accommodates modification if a similar view is needed or extra requirements need to be added to one view.

## Liskov Substitution Principle

The Liskov Substitution principle states that the subclasses or objects in a program should be substitutable for their base class without changing the correctness of the program (Hoiberg, n.d.). In other words a derived class should be interchangeable for a base class (Gordon, 2019).

## Interface Segregation Principle

The Interface Segregation principle states that it is a better practice to create several specific interfaces rather than one general interface as this prevents the child class from inheriting any behaviour that is unrelated. Additionally, it also ensures that classes are not depending on methods that they will never use.



The screenshot shows a code editor with four tabs: WishlistController.php, CartController.php, DashboardsController.php, and IDashboardRepository.php. The IDashboardRepository.php tab is active, displaying the following PHP code:

```
1  <?php
2
3  namespace App\Contracts\Repos;
4
5  interface IDashboardRepository{
6      public function getUser();
7
8      public function getOrders($id);
9
10     public function getDetails($id);
11 }
```

## Dependency Inversion Principle

The Dependency Inversion principle states that both high-level modules and low-level modules should depend on abstraction instead of a high-level module depending on a low-level module (Hoiberg, n.d.). A high-level module is a module that does not focus on detailed implementation, it is the interface or abstraction that will be directly consumed by the presentation layer. On the other hand, a low-level module is a sub-system which aids the high-level module.

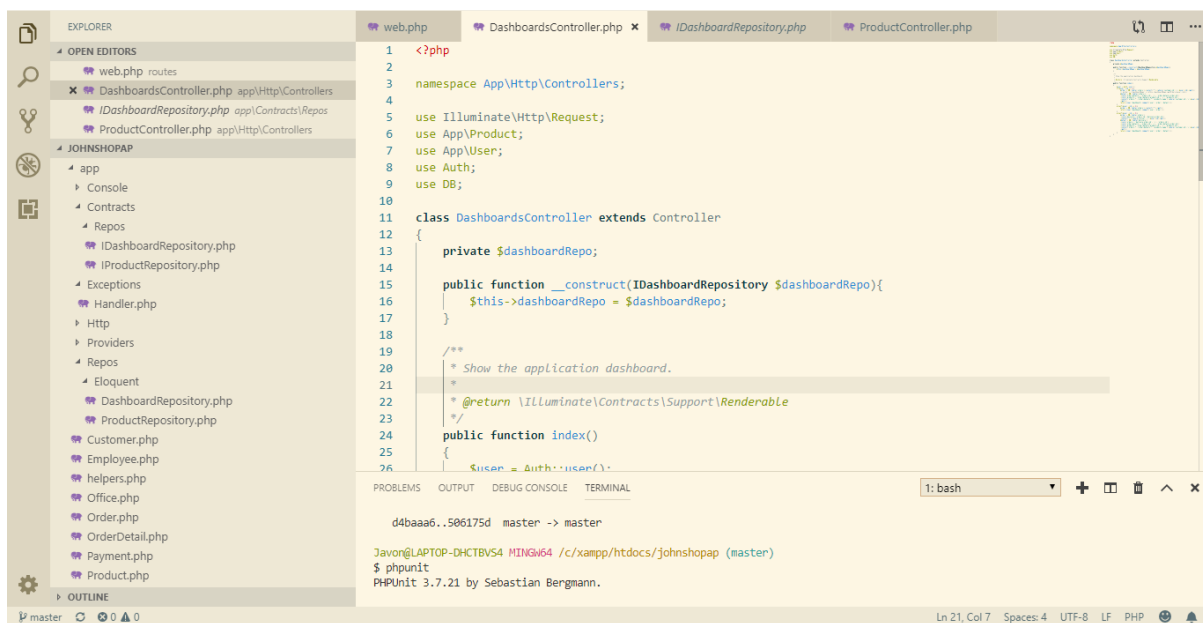
This was demonstrated in the John Shop web application as the application utilizes one main controller which then extends to all other controllers.



```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6  use App\Product;
7  use App\User;
8  use Auth;
9  use DB;
10
11 class DashboardsController extends Controller
12 {
13     private $dashboardRepo;
14
15     public function __construct(IDashboardRepository $dashboardRepo){
16         $this->dashboardRepo = $dashboardRepo;
17     }
18
19 }
```

## Repository Pattern

According to Martin Fowler (2003), [A Repository] mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects. The repository pattern is an intermediary between two layers. This is often used when there is a need for modification of data before the data passes to the next stage (Kumar, 2015). The use of the repository pattern is also used to reduce coupling (Gordon, 2019). This is achieved through the creation of an abstraction data layer (Bergman, 2017).



### Model-View-Controller Design Pattern

The Model-View-Controller (MVC) design pattern is a pattern, commonly used for applications with modern user interfaces, which separates three application concerns:

- Model – the way in which the application accesses data (MVC, 2008).
- View – the way in which the user interacts with the application and is able to visualize data (the user interface) (Gordon, 2019).
- Controller – the way in which interaction between the view and the model is handled, the controller manages data entering the model object and updates the view as data changes while keeping the model and the view separate (Design Patterns - MVC Pattern, n.d.).

Laravel is largely based on the Model-View-Controller design pattern. In the John Shop web application a model, Product, was created to access the data stored in the database. The corresponding controller, ProductController, is tasked with retrieving and displaying products so that the user is able to interact with the products available can be achieved. In the resources/views directory, which stores views, the corresponding view, product.blade.php, stored in HTML, present the user interface.



```
Product.php x product.blade.php ProductController.php
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Product extends Model
8 {
9     //
10 }
11
```

```
Product.php product.blade.php ProductController.php x
14
15 public function __construct(
16     IProductRepository $productRepo){
17     $this->productRepo = $productRepo;
18 }
19
20 /**
21  * Display a listing of the resource.
22  *
23  * @return \Illuminate\Http\Response
24  */
25 public function index()
26 {
27     $products = Product::all();
28     return view('shop')->with('products', $products);
29 }
30
31
32 /**
33  * Display the specified resource.
34  *
35  * @param string $slug
36  * @return \Illuminate\Http\Response
37  */
38 public function show($slug)
39 {
```



```
1 @extends('layouts.app')
2
3 @section('content')
4
5     <div class="container">
6         <p><a href="{{ url('/shop') }}">Shop</a> / {{ $product->name }}</p>
7         <h1>{{ $product->name }}</h1>
8
9         <hr>
10
11        <div class="row">
12            <div class="col-md-8">
13                <div class="thumbnail">
14                    
15                </div>
16            </div>
17
18            <div class="col-md-4">
19                <h3>{{ $product->price }}</h3>
20                {{ $product->description }}
21                <hr>
22                <form action="{{ url('/cart') }}" method="POST" class="side-by-side">
23                    {!! csrf_field() !!}
24                    <input type="hidden" name="id" value="{{ $product->id }}">
25                    <input type="hidden" name="name" value="{{ $product->name }}">
```

## Singleton Pattern

The singleton pattern is a design pattern which stipulates that only one instance of a resource or class is to be created for use by several objects (Gordon, 2019).

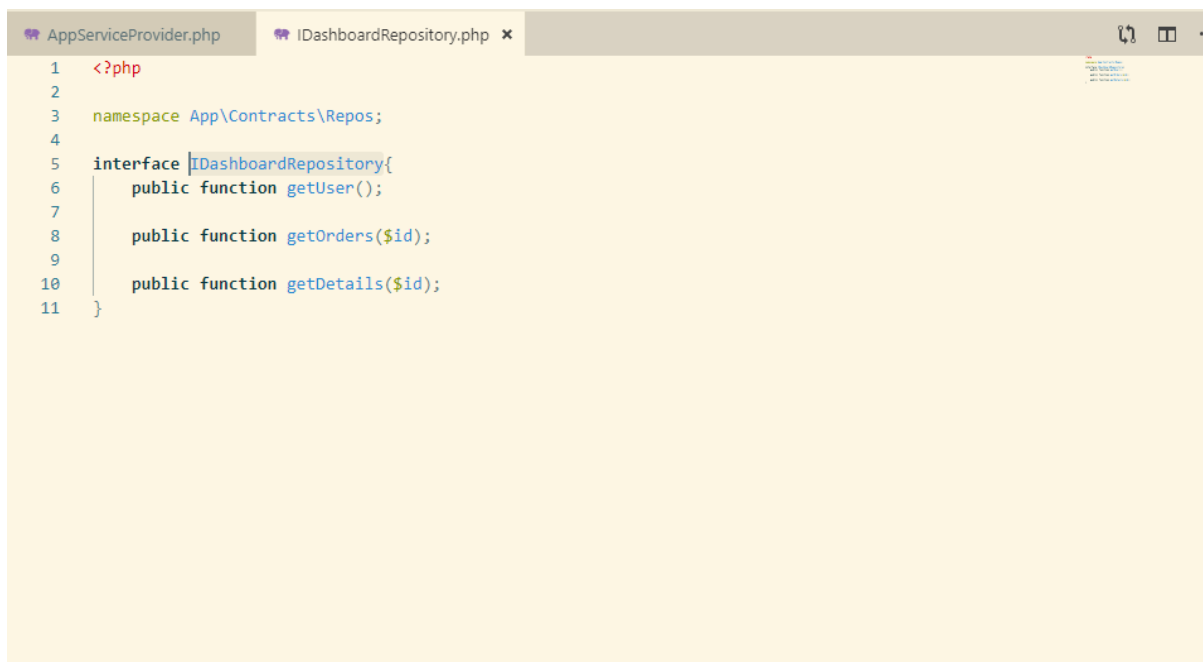
This is demonstrated in the John Shop web application through the implementation of the singleton method. The ServiceProvider is binding the IDashboard repository to the DashboardRepository. Only one instance of the interface is created for each user who logs in to the web application during the lifespan of the application.



```
20
21
22  /**
23   * Bootstrap any application services.
24   * @return void
25   */
26  public function boot()
27  {
28      //
29  }
30
31  /**
32   * All of the container singletons that should be registered.
33   * @var array
34   */
35  public $singletons = [
36      IProductRepository::class => ProductRepository::class,
37      IDashboardRepository::class => DashboardRepository::class,
38  ];
39
40 }
41
```

## Factory Pattern

The factory pattern is a design pattern which speaks to the creation of objects (Gordon, 2019). The factory pattern ensures that the creation logic of an object is not exposed to the client and defers the creation of new objects to a common interface.



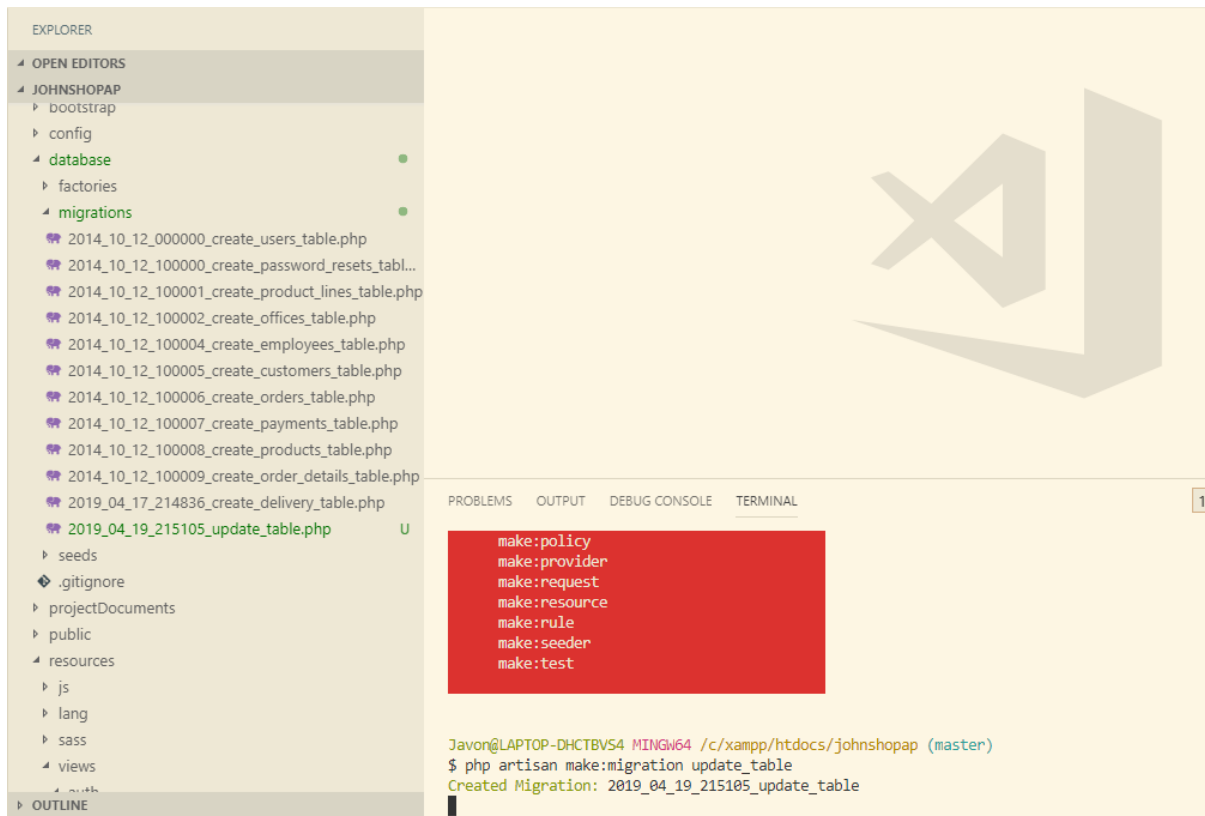
The screenshot shows a code editor with two tabs: 'AppServiceProvider.php' and 'IDashboardRepository.php'. The 'IDashboardRepository.php' tab is active, displaying the following PHP code:

```
1  <?php
2
3  namespace App\Contracts\Repos;
4
5  interface IDashboardRepository{
6      public function getUser();
7
8      public function getOrders($id);
9
10     public function getDetails($id);
11 }
```

## Code Generation Tool

A code generation tool is a tool that makes coding easier by generating code through the use of other programs (Keret & Zviling). It is most commonly defined as a tool which is used to convert source code to machine language, i.e. the compiler (Techopedia, n.d.).

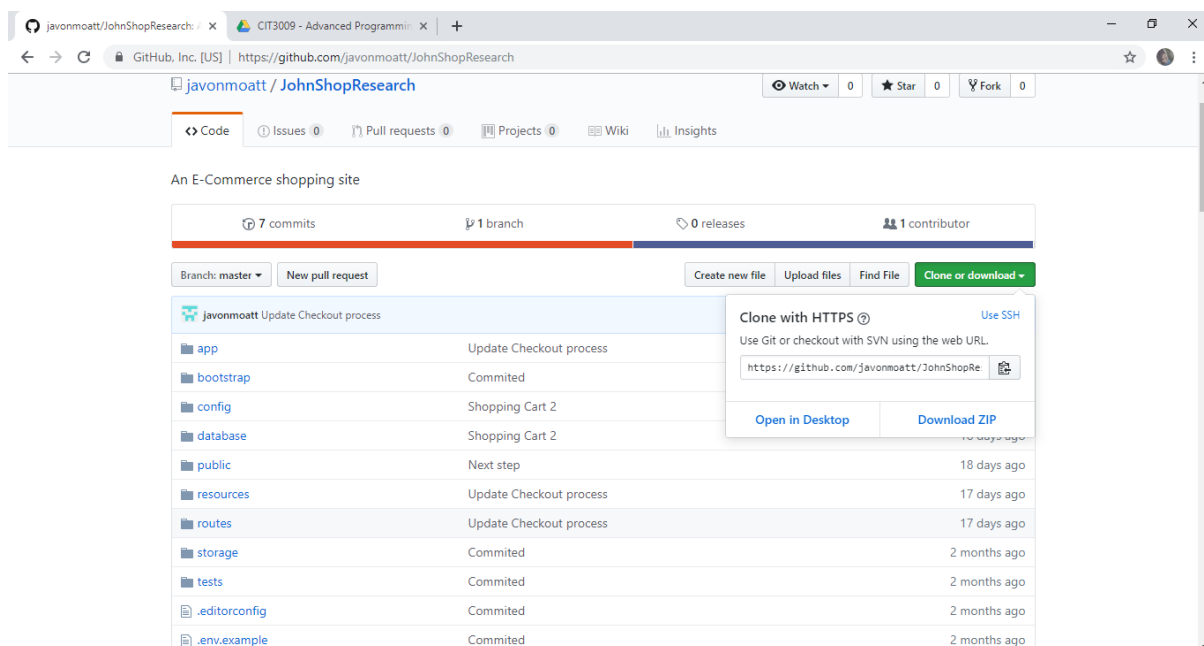
The compiler utilized in this project was Artisan, as pictured below.



## Source Control Management Tool

A source control management tool is any tool which showcases and allows one, who is a member of a project team, to track all the changes made to a code and by whom. Moreover, it allows a team to easily collaborate and offers a centralized management portal, allowing for continuous delivery (What are Source Control Management Tools?, n.d.).

This project utilized Github as a source control management tool. Github was used so that the team could work together on the project by being able to track the changes each person made, thus creating a collaborative, yet transparent effort.



### Package Management Tool

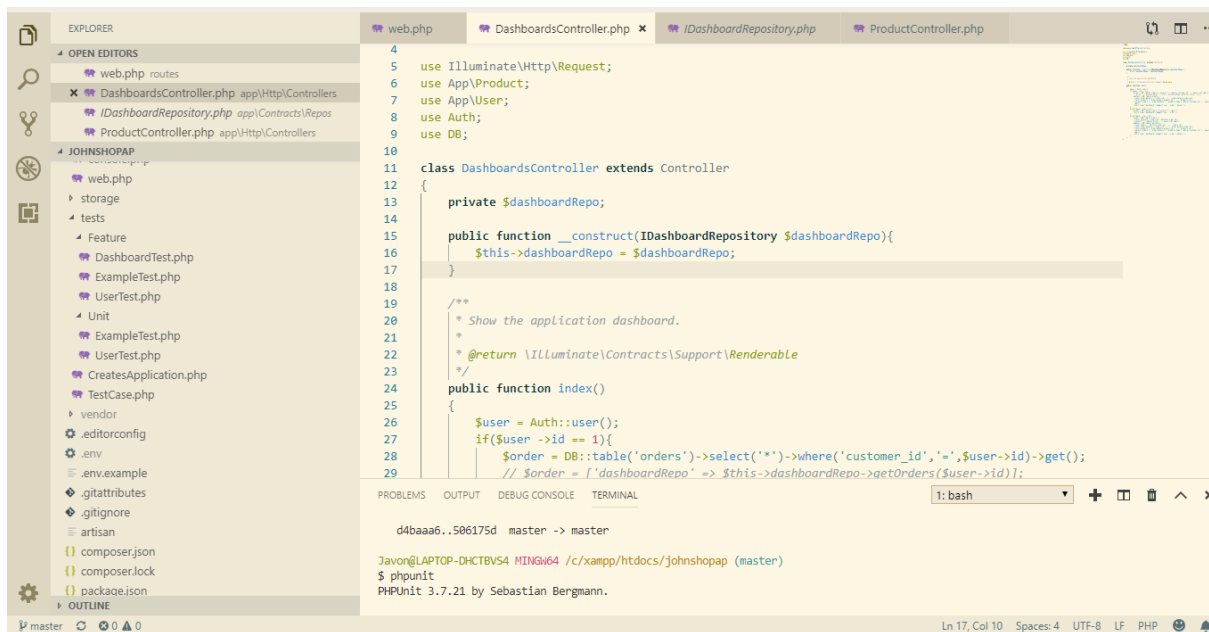
A package management tool is a tool or collection of tools which allows users to remove, configure, upgrade, or install packages according to their needs (Red Hat, Inc, 2005) (Lindley, 2018).

The package management tool used in this project was Ninte. This was used to install the relevant packages used to create and run the John Shop web application.

## Unit Testing and Test Automation

Unit Testing refers to software testing which tests components or units of software to confirm that the component works as it should. For the purpose of Object-Oriented Programming a unit or component refers to a method (Unit Testing, n.d.).

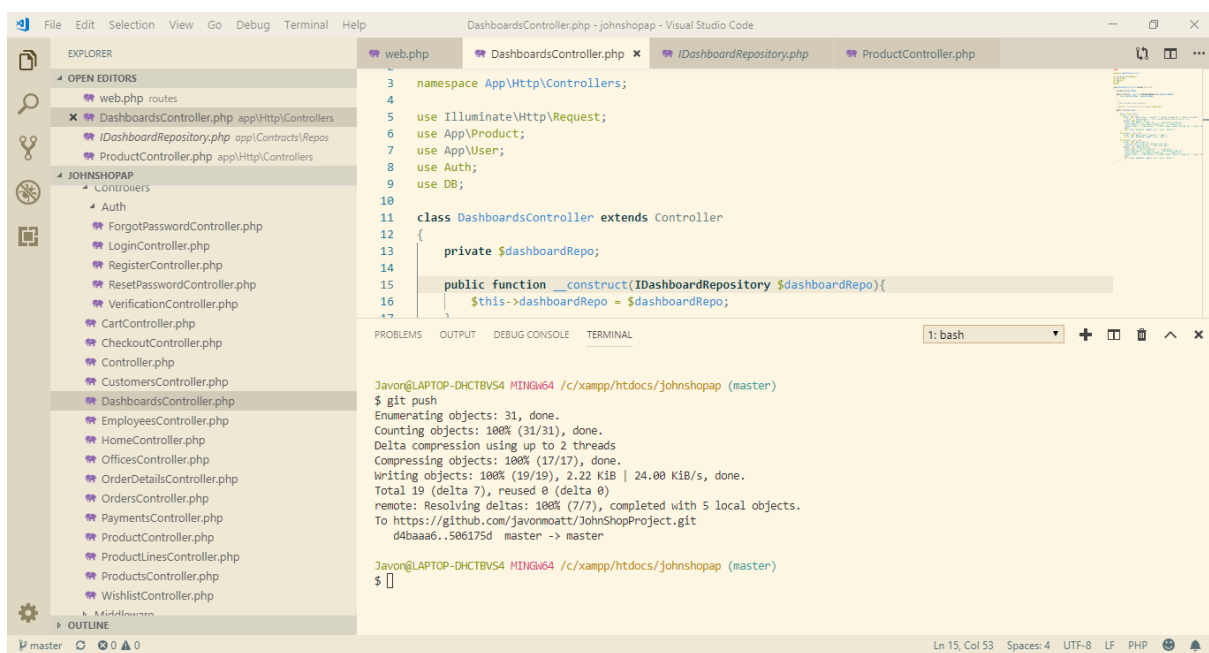
Test Automation refers to a method of software testing which utilizes software tools to control the execution of tests. This is especially useful when the tests which need to be performed are too tedious when done manually (Automated Testing, n.d.).





## Continuous Integration Using a Continuous Integration Server

Continuous integration is a practice used by developers that involves the integration of code into a shared repository. This is followed by an automated build so that errors can be detected easily (Continuous Integration, n.d.).



```
File Edit Selection View Go Debug Terminal Help DashboardsController.php - johnshopap - Visual Studio Code

EXPLORER
  OPEN EDITORS
    web.php routes
    DashboardsController.php app\Http\Controllers
    IDashboardRepository.php app\Contracts/Repos
    ProductController.php app\Http\Controllers
  JOHNSHOPAP
    Controllers
      Auth
        ForgotPasswordController.php
        LoginController.php
        RegisterController.php
        ResetPasswordController.php
        VerificationController.php
      CartController.php
      CheckoutController.php
      Controller.php
      CustomersController.php
      DashboardsController.php
      EmployeesController.php
      HomeController.php
      OfficesController.php
      OrderDetailsController.php
      OrdersController.php
      PaymentsController.php
      ProductController.php
      ProductLinesController.php
      ProductsController.php
      WishlistController.php
    Middlewares

web.php
routes
DashboardsController.php
IDashboardRepository.php
ProductController.php

3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6 use App\Product;
7 use App\User;
8 use Auth;
9 use DB;
10
11 class DashboardsController extends Controller
12 {
13     private $dashboardRepo;
14
15     public function __construct(IDashboardRepository $dashboardRepo){
16         $this->dashboardRepo = $dashboardRepo;
17     }
18 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash

Javong@LAPTOP-DHCTBV54 MINGW64 /c:/xampp/htdocs/johnshopap (master)
$ git push
Enumerating objects: 31, done.
Counting objects: 100% (31/31), done.
Delta compression using up to 2 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (19/19), 2.22 KiB | 24.00 KiB/s, done.
Total 19 (delta 7), reused 0 (delta 0)
remote: Resolving deltas: 100% (7/7), completed with 5 local objects.
To https://github.com/javonmoatt/JohnShopProject.git
d4baaa6..506175d master -> master

Javong@LAPTOP-DHCTBV54 MINGW64 /c:/xampp/htdocs/johnshopap (master)
$
```

## References

*APA Sample Paper - APA Style*. (2018, March 20). Retrieved from APA Style:

<https://www.apastyle.org/manual/related/sample-experiment-paper-1.pdf>

*Automated Testing*. (n.d.). Retrieved from Techopedia:

<https://www.techopedia.com/definition/17785/automated-testing>

Bergman, P.-E. (2017, April 20). *Repository Design Pattern*. Retrieved from Medium:

<https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30>

*Continuous Integration*. (n.d.). Retrieved from ThoughtWorks:

<https://www.thoughtworks.com/continuous-integration>

*Design Patterns - MVC Pattern*. (n.d.). Retrieved from Tutorialspoint:

[https://www.tutorialspoint.com/design\\_pattern/mvc\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm)

Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. Boston, MA: Pearson Education, Inc.

Gordon, G. (2019, January). Lecture 1 - Continuous Integration. Kingston, Jamaica.

Gordon, G. (2019, January). Lecture 5 - Design Patterns. Kingston, Jamaica.

Hoiberg, S. L. (n.d.). *SOLID Principles: Explanation and examples*. Retrieved from ITNext:

<https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4>

Keret, G., & Zviling, M. (n.d.). Code Generation Tools.

Kumar, M. (2015, October 12). *Repository Pattern and Generic Repository Pattern*.

Retrieved from C# Corner: <https://www.c->

sharpcorner.com/UploadFile/8a67c0/repository-pattern-and-generic-repository-pattern/

Lindley, C. (2018). *Front-End Developer Handbook 2018*. GitBook.

Martin, R. C. (2000). *Design Principles and Design Patterns*.

MVC. (2008, March 2018). Retrieved from TechTerms: <https://techterms.com/definition/mvc>

Red Hat, Inc. (2005). *Chapter 8. Package Management Tool* . Retrieved from Red Hat

EnterpriseLinux 4 - MIT: <https://assessment-tools.ca.com/tools/continuous-delivery-tools/en/source-control-management-tools>

*Techopedia*. (n.d.). Retrieved from Code Generator:

<https://www.techopedia.com/definition/17062/code-generator>

*Unit Testing*. (n.d.). Retrieved from Software Testing Fundamentals:

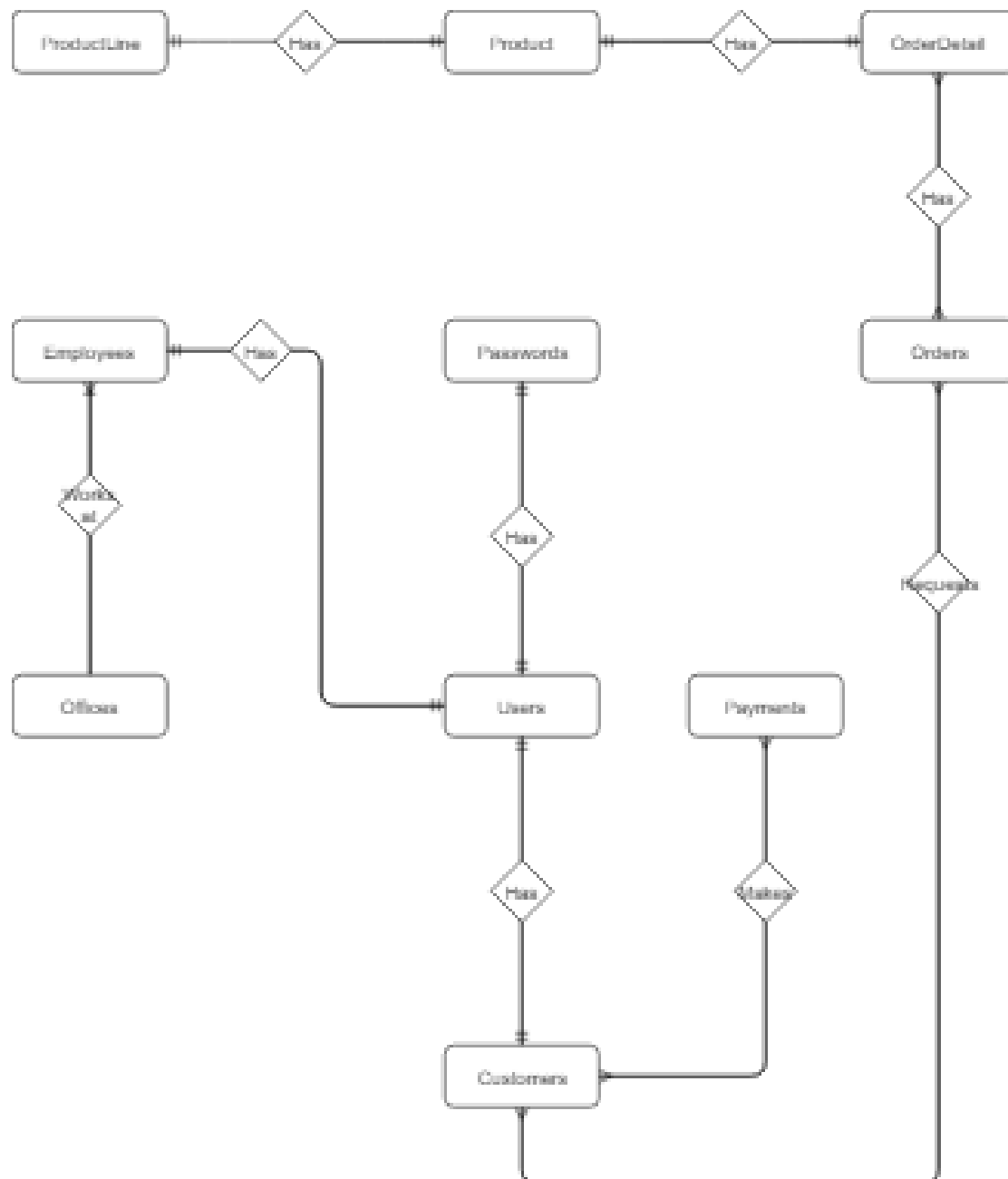
<http://softwaretestingfundamentals.com/unit-testing/>

*What are Source Control Management Tools?* (n.d.). Retrieved from CA Technologies:

<https://assessment-tools.ca.com/tools/continuous-delivery-tools/en/source-control-management-tools>

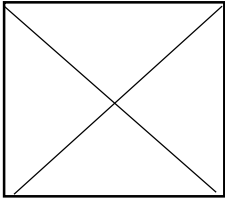
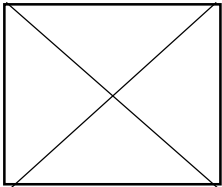
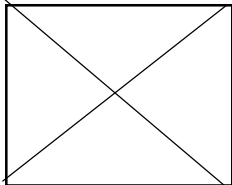
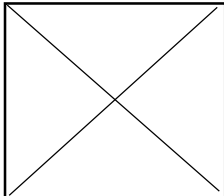
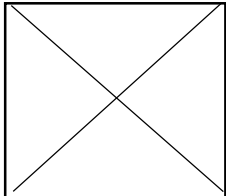
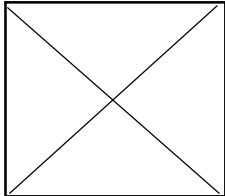
## Appendices

## Entity Relationship Diagram



## Wire Frame

/shop

<div>JOHN SHOP</div>		
		
		

## Login

JOHN SHOP	
<div>USERNAME</div> <div>PASSWORD</div>	

## Shopping cart

