1 Database Design

1.1 Description

A website that gets users to post requests for commissions. They would need a database for all their data of requests, users' information, and commissions. For each request, they would need genres, allocated price range, deadline, and who posted the request. The database needs to keep track of all requests that are being posted. The users will have to input their information when registering an account. Users will also have to enter their roles when logging in and their card details when making a payment for their commission.

1.2 Business Reporting Requirements

The information the users of your application will want to be able to view are the following:

1. Users (including admin, moderators and ordinary) need to be able to create, read, update, and delete: art requests, profiles, commissions, and categories.
2. Users will need to be able to find all requests, profiles, and commissions ordered by their art categories
3. Users may want to find a request by a specific price range i.e. from high to low, or vice versa.
4. Users need to find all requests using a list of art genres i.e. painting, digital painting, traditional, pixel arts etc.
5. Clients need to find the profile for a specific genre.
6. Clients need to find the profiles using a artist's name.
7. Artists may need to find the list of art requests.
8. Users need to find festivals by location and the location needs to be displayed on a Google Map
9. User may need to find festivals by city
10. Users need to find clients within art requests by a similar field of genre.

1.3 Textual Representation of Data-Set

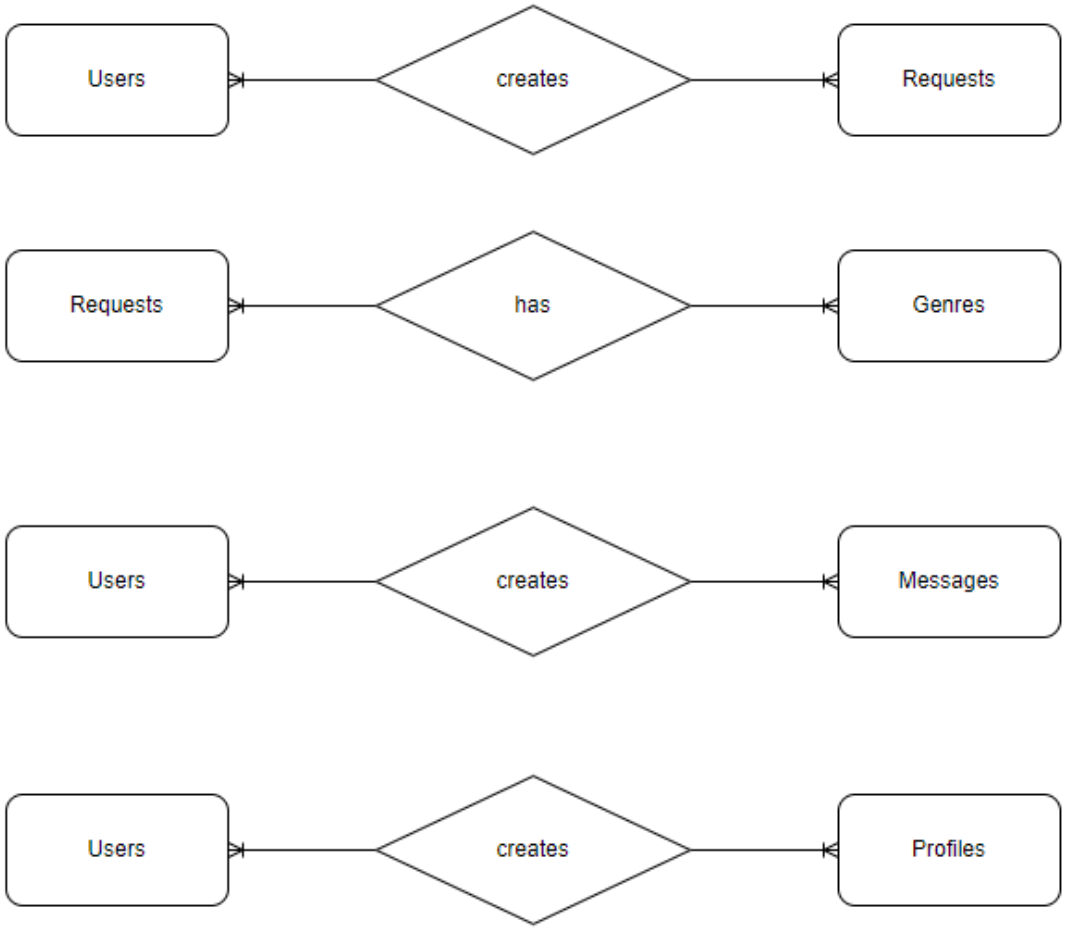
Substitute in here the tables for your database

REQUEST (id, title, commission_type, theme, descriptions, start_date, end_date, start_price, end_price, user_id)
USER (id, username, email_address, password, user_role_identifier_id, user_image_id)
GENRE (id, title, description)
USER_IMAGE (id, filename)
ARTIST (id, username, price_start, price_end, user_image_id, status, image_showcase_id, genre_id)
MESSAGE (id, messages, user_id, date_stamp)

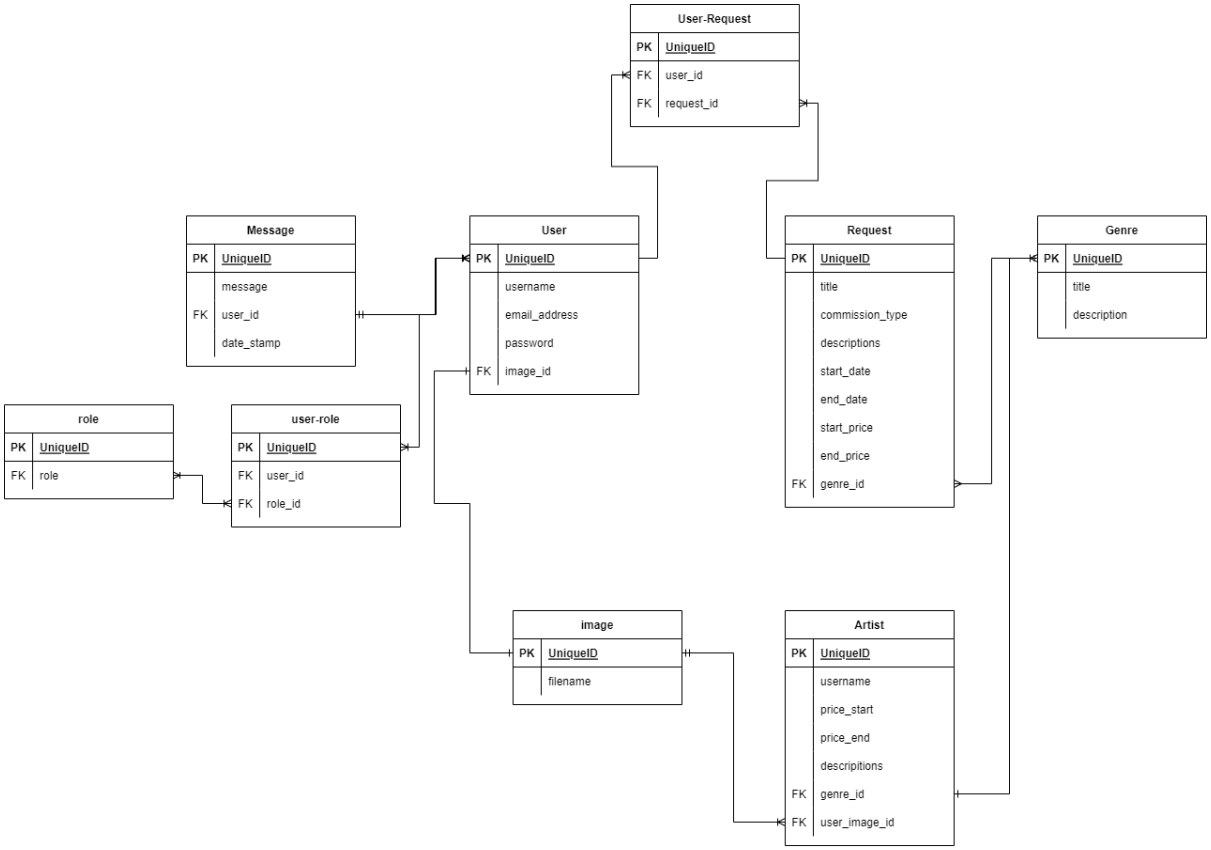
1.4 Business Rules

- A User has many **Requests**.
- A **Request** belongs to one **User**.
- A **Request** has many **Genres**.
- A **User** has many **Messages**.
- A **Message** belongs to one **User**.

1.5 Entity Relationship Diagram



1.6 Tables


User-Request
PK UniqueID
FK user_id
FK request_id
Message
PK UniqueID
message
FK user_id
date_stamp
User
PK UniqueID
username
email_address
password
FK image_id
Request
PK UniqueID
title
commission_type
descriptions
start_date
end_date
start_price
end_price
FK genre_id
Genre
PK UniqueID
title
description
role
PK UniqueID
FK role
user-role
PK UniqueID
FK user_id
FK role_id
image
PK UniqueID
filename
Artist
PK UniqueID
username
price_start
price_end
descripitions
FK genre_id
FK user_image_id

1.7 Database Dictionary

Table name	Column name	Description	Data type	Required	PK or FK	Ref Table for FK
User	Id	Primary key	Int	Yes	PK	
	Username	User's name	Varchar(30)	Yes		
	Email_address	User's email	Varchar(30)	Yes		
	Password	User's password	Varchar(50)	Yes		
		User's role	Varchar(50)	Yes		
	User_role_id	User' profile	Int	Yes	FK	User_role
	Image_id	image		No	FK	User_Image
Request	Id	Primary key	Int	Yes	PK	
	Title	Request's title	Varchar(30)	Yes		
	Commission_type	Traditional/Digital	Varchar	Yes		
	Descriptions	Detailed description	Varchar	Yes		
	Start_date	when request's begins	Date	Yes		
	End_date	request's deadline	Date	Yes		
	Start_price	request's start price	Float	Yes		
	End_price	and end price	Float	Yes		
	Genre_id	art genre	Int	Yes	FK	Genre
Artist	Id	Primary key	Int	Yes	PK	
	Username_id	Artist's username	Int	Yes	FK	user
	Price_start	commission's start	Int	Yes		
	Price_end	and end price	Int	Yes		
				Yes		

	Descriptions Genre_id User_image_id	Detailed description Art genre Artist's work preview	Char Int int	Yes Yes Yes	 FK FK	 Genre image
Message	Id message User_id Date_stamp	Primary key User's message Which user Time when message sent	Int Chart Int Date	Yes Yes Yes Yes	 FK	 User
User-Role	Id User_id Role_id	Primary key Id of the user Id of the role	Int Int Int	Yes Yes Yes	PK FK FK	 User Role
Role	Id role	Primary key Role name	Int Int	Yes Yes	PK	
Genre	Id Title Description	Primary key Genre name Description of the genre	Int Varchar Varchar	Yes Yes Yes	PK	
Image	Id Filename	Primary key File asset location	Int Varchar	Yes Yes	PK	

2 System Design/ Architecture Overview

2.1 Introduction

Figure 1 Introduction to Laravel Framework

The web framework we are tasked to use in this Continuous Assessment (CA) is called Laravel. This is a web application which has been built around being a starter kit for flexibility and mobility of the website.

Laravel contains a full-stack applications that could be used to link data management system like MySQL to PHP without needing to write the essential lines of code to intervene the communication of languages, bridging the two quickly using a shorter line of codes, creating a clean and beauty environment and outputting amazing functionality with the palm of our hands as it provides services such as database, queuing requests i.e. sending emails and performing tasks in a short spans of time, web-socketing and broadcasting events in real-time, and finally authenticate these sessions with security preventing painful authentication for APIs and mobile applications.

Alternatively, there is also another web application framework to use for the CA called CodeIgniter, but it is reported to be low-quality solution in the lights of Laravel, thus the existing of this framework has become quite unpopulated and untrendy by the majority of programmers.

2.2 Model View Controller

The general process of a model-view-controller design pattern can be divided into three major components such as:

- **The model** – which is the main component of the structure, and its tasks are to precisely manage the data, logic, and rules of the website, keeping it in working

orders. This component will perform tasks in the backend of the application and when done, it will send these queries to the view component.

- **The view** – a component that receive data from the backend or the model and turn those findings into information, and remarkably display it on the webpage.
- **The controller** – another important component that allows developers to parse inputs and convert them into variable commands i.e. calculations, data from the database for the model and/or the view components.

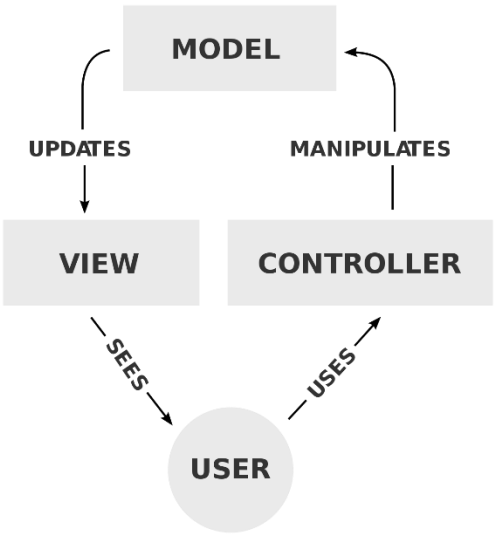


Figure 2 Model View Controller Design Pattern

2.3 User Authentication

The **user authentication** secures each page in the website application which prevents users who does not have the required permissions to access said pages. It allows for the finer method to guard users from getting into places where they are not meant to be in.

When a user wants to access pages that they might be able to view, they need to log in or register to the site in order to do so. This is done through the help of the database management system which logged the data of users' information including their usernames and passwords, and their assigned role(s) that they will automatically be assigned to giving/taking away permissions to view the page(s).

To further securing the account of users for their safety in the case of horrible scenario that may potentially occurred, the Hash method is used, in order to encrypt the password of the

users that are stored in the database. This allow for a greater security measurement and prevent hacker from stealer the clients' secret credentials.

2.4 Routing

In order for Laravel to understand which page to display to the user(s), we need to address where to send the user to, and which **blade.php** or page to load and display to the user(s). This can be done by using the **Route()** command which takes a parameter of the what URI you want to link the view to, so that it would be able to return that page to the front-end user. During this process, the middleware will come in handy in authenticating the user's permits to access the page.

2.5 Templating

The templating engine has been provided by Laravel which further allowed the riddance of hideous syntax writing and easing the process of templating a PHP page. This engine is called **Blade**, wherein it provides us with its very own structure of coding such that you would only have to use conditional statements and loops to extract data from the database or the classes responsively and powerfully by using simple and precise commands/syntax. For instance, if you would like to echo the value of a variable, you could simply do so by encapsulate the variable inside the curly brackets i.e. {{ $variable }}, instead of having to write the code in PHP format like <?php echo $variable; ?>, which could be painstakingly bothersome every time you would like to call variables. It provides a shorter yet effective method to code web application.

Conditional statements are also made accessible and precise with only having to write @if(condition) at the start and @endif at the end to iterate through the codes. The modern

method such as this made it responsively for programmers to code and finish jobs in an easier manner than if one were to using the PHP language's syntax.

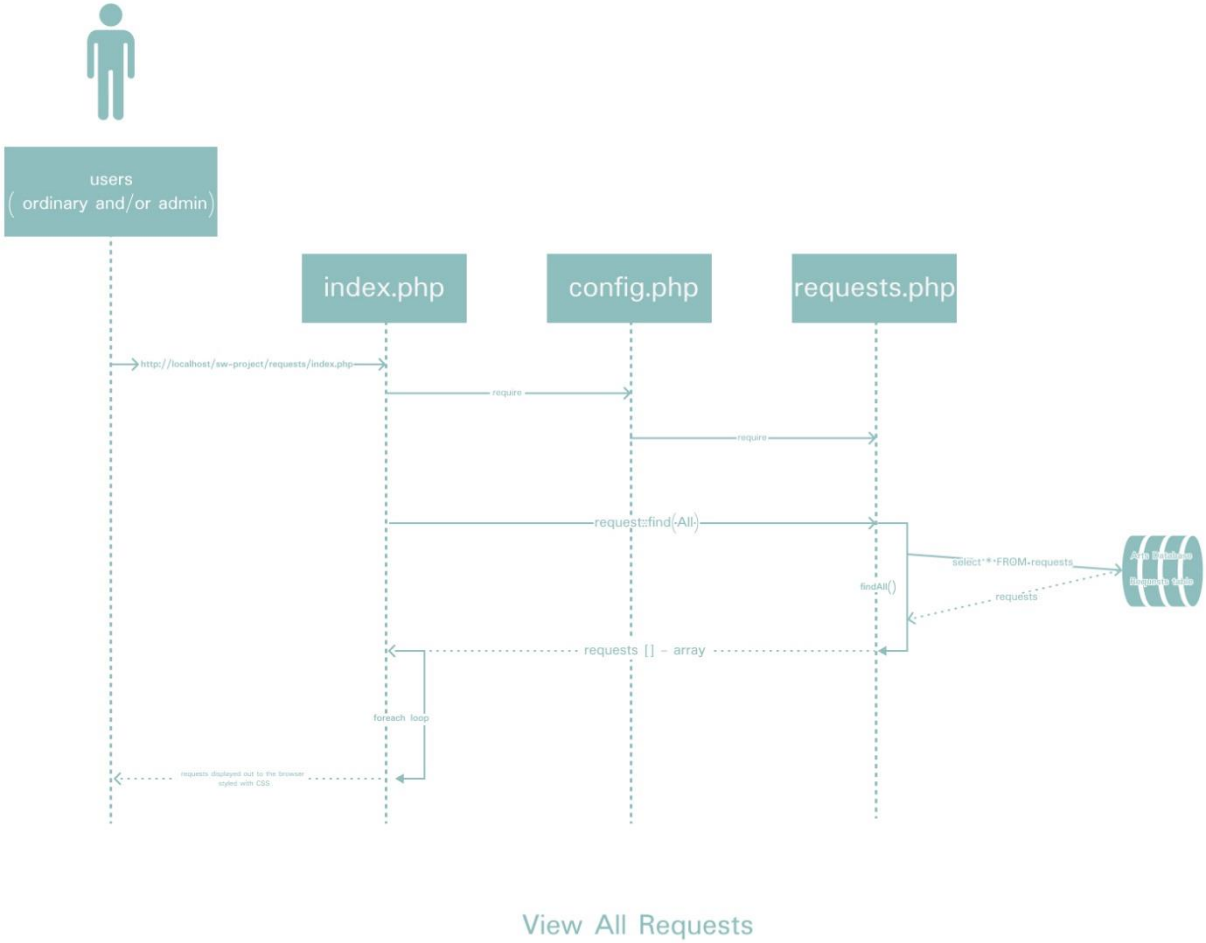


Figure 3 Template of View All Requests Blade

This is a template to represent the process of routing when a user accesses the request page of my website. The client-side interface would access the data which have been done in the server-side process in the background to retrieve these findings from the database, encapsulate it in a variable of array and return it to the index.php.

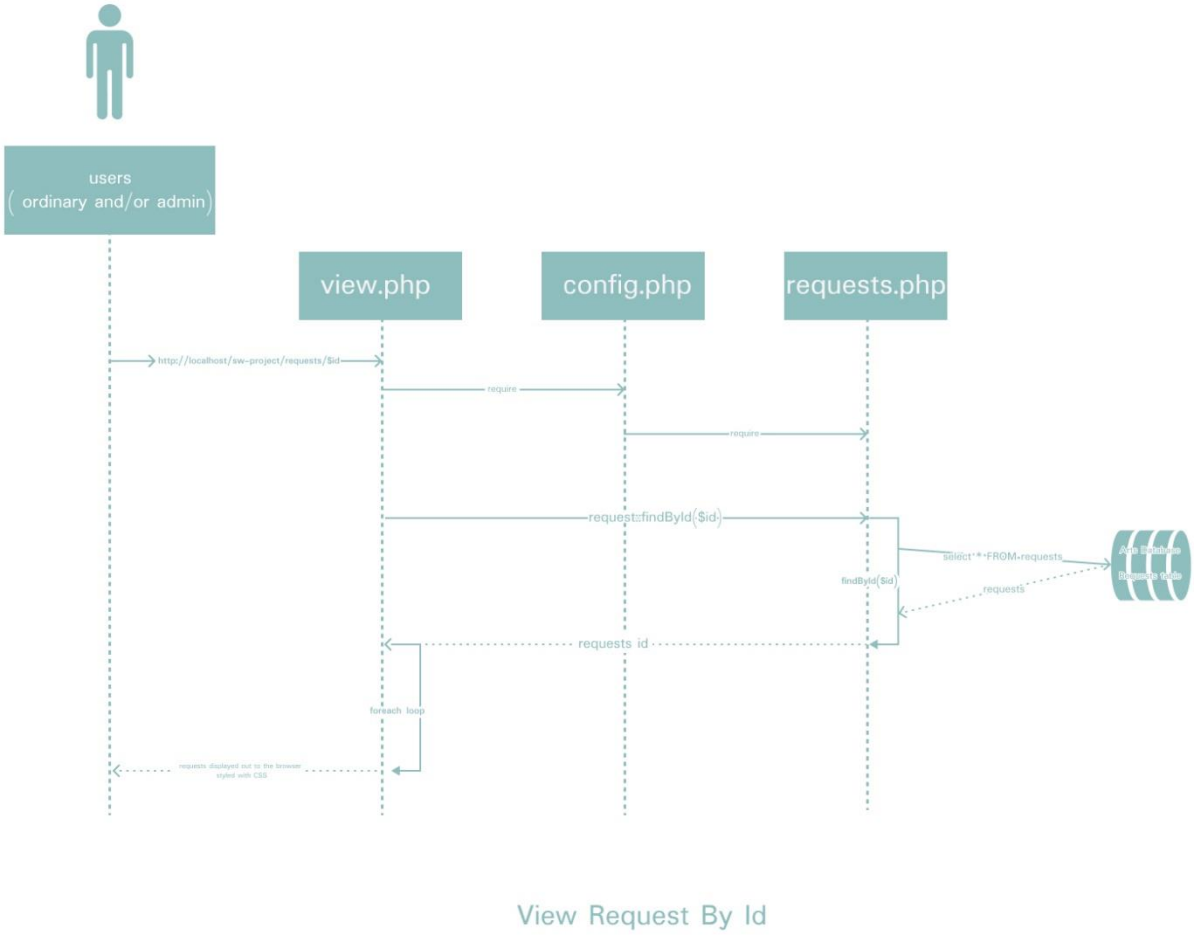
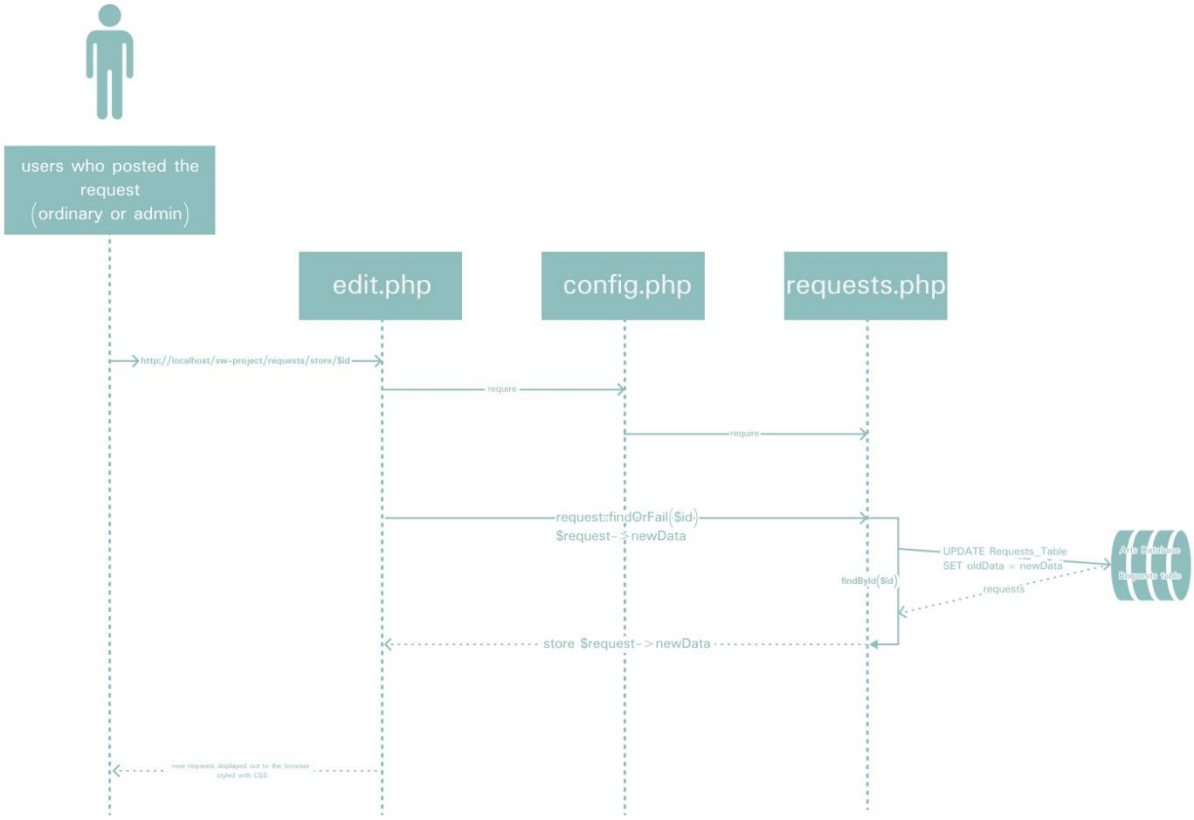


Figure 4 Template of View Requests By Id Blade

When the user clicked on the request(s), they will be routed to the view.php which takes a argument or id of the request from the index.php and extract the findings from the database then display it to the user in a stylised html code.



Edit Request By Id

And if the user who has posted the said request wished to edit their requests, the new updated data would be push through the method in the controller, where the data would be instantaneously get updated on the database. Ultimately, these sample templates would be reoccurring in most of other templates in my web application.