Project No: SCE15-0738

NANYANG TECHNOLOGICAL UNIVERSITY

Justgivit: Digital Platform for Giving

Robert Limanto

School of Computer Science and Engineering

October 2016

Project No: SCE15-0738

NANYANG TECHNOLOGICAL UNIVERSITY

SCE15-0738
Justgivit: Digital Platform on Giving

Submitted in Partial Fulfilment of the Requirements
for the Degree of Bachelor of Engineering (Computer Engineering)
of the Nanyang Technological University
by

Robert Limanto
School of Computer Science and Engineering

October 2016

Project No: SCE15-0738

Abstract

This project tries to encourage people to extend the lives of their idle possessions by providing the innovative two-way interaction between the potential donator and receiver. Furthermore, this project includes the implementation of web and mobile application that should improve continuously using agile methodology. The application is already in the cloud and getting many constructive and positive feedbacks from the users

Acknowledgement

First, I would like to express my gratitude to Dr Owen Noel Newton Fernando for his continuous support and guidance in my final year project. It is a great opportunity to work under his supervision. He has been very resourceful and helpful each time I have a problem and continuously giving constructive feedback to ensure this FYP project successful.

Next, I would like to thank my family, my parents Mr. Yusdi Edy and Mrs. Iin Koemala and my two brothers for their unconditional love and support in my study here in NTU.

Besides that, I would like to thank all my friends especially my SCE and Indonesian friends. I will never forget about the time that we have spent together and joked about our modules and hobbies.

Table of Contents

List of Figures

List of Tables

Project No: SCE15-0738

1. Introduction

1.1. Background

Consumptive behavior in this fast-paced society has created a toll on our Mother Earth. Every year, humans worldwide produce incredibly 2.12 million tons of waste. A recent investigation showed that 99 percent of what we buy today would end up in the landfill after merely 6 months. [1] The vicious cycle continues where consumers keep replacing their functioning goods to catch up with the trend, creating the supposedly unnecessary production and recycling effort.

One of the reasons these short-lived goods are wasted before their marginal utility is fully benefited is because it is clearly more convenient to simply dispose of their belongings than undergo the complicated procedure of donating them. On the other hand, there is also no platform for the unprivileged people to request for the types of the goods that they desperately need.

1.2. Purpose of project

This Final Year Project aims to develop a platform that emphasizes on direct interaction between the goods donator and the potential receiver, requesting stuff, and connecting the needed stuff to a pool of relevant stakeholders. The author believes that this project can encourage people to extend the lives of their goods by donating to those who will really benefit from the donation.

1.3. Scope

The scope of this project is to create both the Web and Mobile Application in Android that can help users in crowdsourcing and give away their stuff for free. To achieve that, the author should design the platform from scratch, implement a scalable software architecture, ensure the application is applicable and perform well, as well as continuously analyzing the data obtained from the users. In addition, the author must consider feedback from users in each version and utilize them to improve the next version of the project.

This project improves continuously from one version to another version. In each version, the author carries specific targets to be done before the deadline. This deadline may vary from 1 week to 8 weeks depending on the tasks. The duration of the starting time of a version to the deadline of a version is often called as a sprint.

However, this project has not yet adopted test-driven development methodology that develops test cases to automate testing. Nevertheless, appropriate testing has been done prior to the deployment of the project using the Use Cases that has been specified in the Software Requirements.

1.4. Organization

This report is organized into six chapters and each chapter may have several sections. In the first chapter, the discussed topics are the motivation, scope, resource requirements, and the project schedule. In Chapter 2, the author introduces the software requirements for this project. In Chapter 3, the author explains about the design and implementation of the software. In Chapter 4, details about the software implementation using Notification as an example are analyzed. In Chapter 5, the author explains about the result of the implementation and the discussion of the data obtained from the users. In the last chapter, the author concludes and gives recommendations for the future works.

1.5. Resource Requirements

1.5.1. Hardware

1.5.1.1. Laptop with Windows OS

1.5.1.2. Mobile Devices with Android and IOS as the Operating System

1.5.2. Software

1.5.2.1. Laptop, Windows 8 & 10

1.5.2.2. Apache Server with PHP 5.6 and MySQL 5.6 installed

1.5.2.3. Amazon Web Services with EC2, Route53, and RDS

1.5.2.4. Version Control using GitHub Private Repository

1.5.2.5. Yii2 Framework

1.5.2.6. Gulp

1.6. Project Schedule

In this section, the author discusses briefly the expected and the actual schedule of this project. In this project, the Expected Project Schedule for Development phase should not be discussed during the initial phase of the project since the duration of each sprint duration can only be determined after the previous sprint is done. Therefore, the expected project schedule shown in the next section is determined not in the initial phase of this final year project.

1.6.1. Expected Project Schedule

Date period	**Plan**
04th December 2016 - 03rd February 2016	- Conduct market research on several ideas - Finalization on FYP topic
4th February 2016 – 22nd March 2016	- Finish 1st Prototype

23rd March 2016 – 22nd May 2016	- Finish 2nd prototype.
23rd May 2016 – 18th July 2016	- Finish and release 1st Alpha version
19th July 2016 – 22nd August 2016	- Finish and release 2nd Alpha version. - Work and Finish on Interim Report
23rd August 2016 – 11th September 2016	- Finish and release 1st Beta version.
12th September 2016 – 18th September 2016	- Start working on FYP report - Finish Mobile App for Android
19th September 2016 – 28th September 2016	- Work on FYP report - Finish and release 2nd Beta version.
29th September 2016 – 16th October 2016	- Finalize FYP Final Report
17th October 2016 – 11th November 2016	- Finalize on amended final report
12th November 2016– 4th December 2016	- Preparation for FYP oral presentation
5th December 2016	- FYP oral presentation

Table 1- Expected Project Schedule

1.6.2. Actual Project Schedule

Date period	**Plan**
04th December 2015 - 03rd February 2016	- Conduct market research on several ideas - Finalization on FYP topic

4th February 2016 – 22nd March 2016	- Finish 1st Prototype. This version includes Login system, create a post, post listing, and profile page.
23rd March 2016 – 22nd May 2016	- Finish 2nd prototype. This version adds new feature such as Login with Facebook system, improvement on UI and setting up Cloud system
23rd May 2016 – 18th July 2016	- Finish and release 1st Alpha version to the cloud.
19th July 2016 – 22nd August 2016	- Finish and release 2nd Alpha version. This version improves the responsive layout feature. - Work and Finish on Interim Report
23rd August 2016 – 11th September 2016	- Finish and release 1st Beta version. This version adds a notification system, searching feature, improving UI, and many other features.
12th September 2016 – 18th September 2016	- Start working on FYP report - Finish Mobile App for Android
19th September 2016 – 28th September 2016	- Work on FYP report - Finish and release 2nd Beta version. This version adds new features such as searching by location and improving layout for Firefox and Edge.

29th September 2016 – 16th October 2016	- Finalize FYP Final Report
17th October 2016 – 11th November 2016	- Finalize on amended Final Report
12th November 2016 – 4th December 2016	- Preparation for FYP oral presentation
5th December 2016	- FYP oral presentation

Table 2 - Actual Project Schedule

2. Literature review

2.1. Academic Research

Crowdsourcing was firstly introduced by Jeff Howe in 2006 which refers to "the act of taking a job traditionally performed by a designated agent (usually an employee) and outsourcing it to an undefined, generally large group of people in the form of an open call.[2] Since then, the idea of crowdsourcing has seen wide applications and implementations.

Crowdsourcing of data is ubiquitous and even provided for free by influential companies like Google, Facebook, Instagram and other social media platform. A search on a word in hashtag, a tagging tool in Instagram, could provide sufficient information regarding the words from around the world.

Next, we see the application of crowdsourcing shifting as an interactive platform to exchange ideas. For example, in 2012, team Impact from Virginia Commonwealth University started an idea crowdsourcing project for people from different professional backgrounds to develop a fresh idea to a solid, ready to be implemented idea. The highlight of the project is to provide interaction of the owner of the project with the public, which is one of the major key in a successful crowdsourcing project. Horvát et al showed that the relationship network of the owner could help in the success of a new crowdfunding project, another type of crowdsourcing. [3] Similarly, a study by S Hui et al proposed that the challenge of crowdfunding were to understand network capabilities, activate network connections, and expand network reach. [4]

Crowdsourcing is also applied in delivery of physical goods, a major leap from merely digital data. Libero and LifeDelivery are examples of the mobile app to crowdsource people who are willing to deliver stuff as requested for monetary rewards. [5][6] Moreover, Cvijikj et al shows that people are willing to use an app for acquiring and sharing crime related information. This study convinced the author that people generally are willing to use an app for a good cause, without receiving any or insignificant compensation. [7]

2.2. Application Review

The hype of the crowdfunding projects has started since 2010 when Kickstarter.com became popular by providing a platform for ideas or projects for raising money. As a result, this platform has helped 113,509 launched projects with the total amount pledged has reached $2,663,066,707. Due to the increasing popularity and impact given to the users, Kickstarter.com received the "Best Inventions of 2010" award and "Best Websites of 2011" award. [8]

The popularity of crowdfunding can also be seen from the thriving of Indiegogo.com. This Silicon Valley grown start-up helps individuals, groups, and non-profit organizations to raise money. According to their website, they are currently the largest crowdfunding platform in the world by having 15 million users visiting their platform each month. [9]

Unlike crowdfunding platforms, there are not many projects on crowdsourcing free goods. In the past, the hindrance might be due to online payment security and logistical management. Online money transferring nowadays is already ubiquitous. Meanwhile, the process of transferring physical things has not been as easy as transferring money. Another issue is people tend to simply discard their belongings rather than go into the hassle of donating for those in needs.

There have been trials on developing projects for crowdsourcing physical stuff in the market such as totallyfreestuffs.com. However, this project is not able to expand their market since the platform only provides one-way communication for the advertiser and very lacking of an interactive function.

3. Software Requirements

This chapter discusses the software requirements of this project. The software requirement includes Use Cases, Functional Requirements, and Non-functional Requirements. The Use Case described the high-level description of the software’s features whereas The Functional Requirement discusses all the features in details. Furthermore, the non-functional requirement describes the requirement of the system as a whole.

The author for testing before deploying the application uses the software requirement. This is done by checking each point in functional requirement and non-functional requirement whether it has been satisfied in the application.

3.1. Use Case Diagram

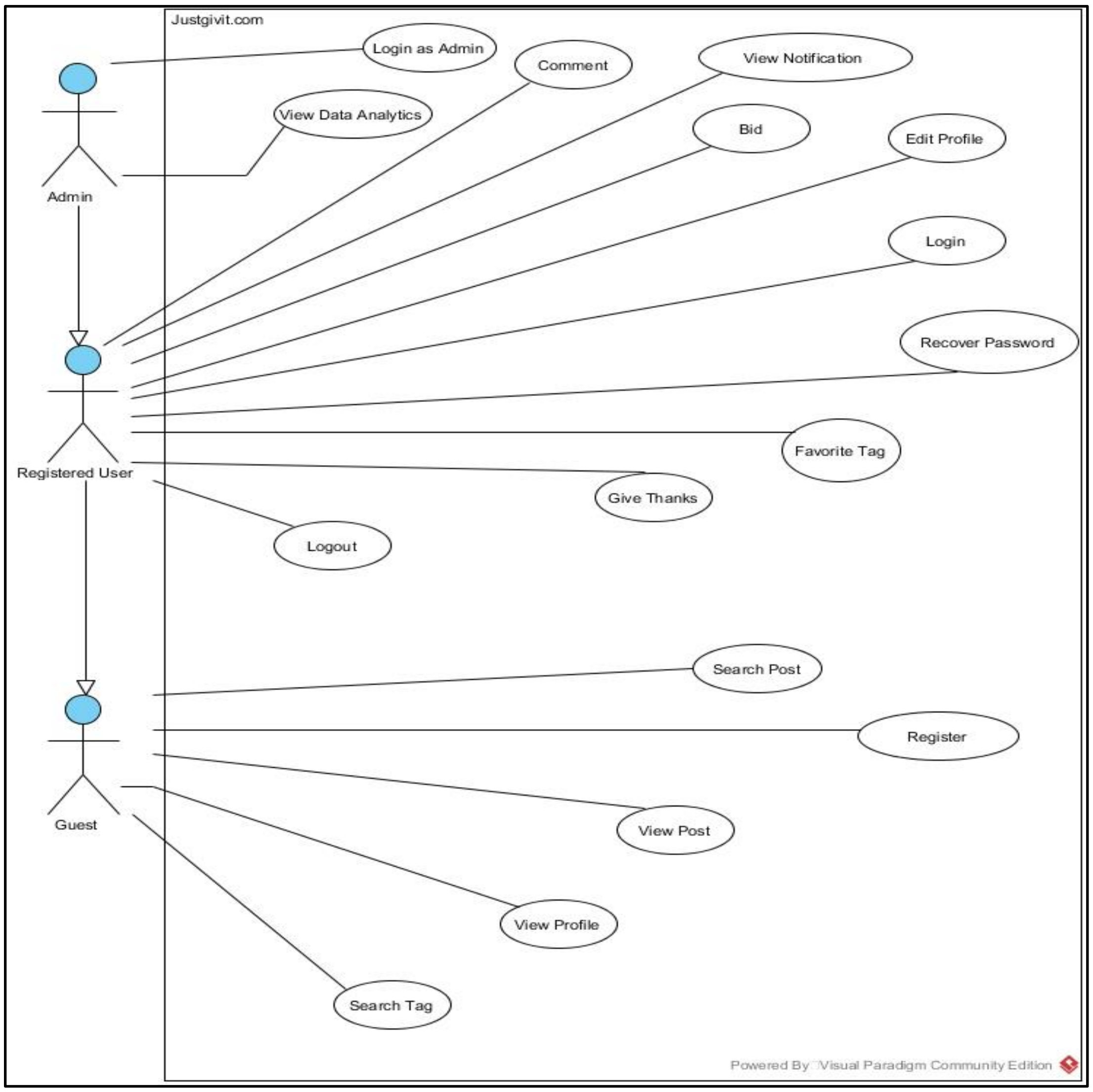


Figure 1- High-Level Use Case of this Project

2.2. Functional Requirements

Section ID	Requirement Description
2.2.1.	The system must be able to be logged in using valid email and password

2.2.1.1.	If the email address is not entered, the system responds by displaying the error message “Email address must not be empty”.
2.2.1.2.	If the password is not entered, the system responds by displaying the error message “Password must not be empty”.
2.2.1.3.	If the email entered is not in a valid format, the system responds by displaying the error message “Email is not a valid email address” If the email entered is not in a valid format, the system responds by displaying the error message “Email is not a valid email address”.
2.2.1.4.	If both the email and password are not valid, the system responds by displaying the error message “Incorrect username and password”.
2.2.1.5.	If the email and password are valid, the system will redirect to the home page.
2.2.2.	The system must be able to be logged in using the user’s Facebook Account when the user clicks the “Continue with Facebook” button.
2.2.3.	The system must be able to send a forgot password email to the email once the email and the captcha are valid.
2.2.3.1.	If the email is not entered, the system responds by displaying the error message “Email address must not be empty”.
2.2.3.2.	If the captcha is not entered, the system responds by displaying the error message “Captcha should not be empty”.
2.2.3.3.	If the email entered is invalid, the system responds by displaying the error message “Email is not a valid email address”.
2.2.3.4.	If the email has not been registered before, the system responds by displaying the error message “There is no user with such email”.
2.2.3.5.	If the captcha is not correct, the system responds by displaying the error message “Captcha is incorrect”.

2.2.3.6.	If both the email and captcha are correct, the system responds by displaying the message “We have sent you the email to reset your password, please check your spam folder if you cannot find it.” The system will then send the email to the specified email.
2.2.4.	The system must be able to change a user password with a valid new password.
2.2.4.1.	If password is not entered, the system responds by displaying the error message “Password cannot be blank”.
2.2.4.2.	If the password entered is less than six characters, the system responds by displaying the error message “Password should contain at least six characters”.
2.2.4.3.	If the password is valid, the system updates the password and redirects the page to the home page.
2.2.5	The system must receive first name, last name, email, and password input to perform a successful registration.
2.2.5.1.	If the first name is not entered, the system responds by displaying the error message, “the First name should not be empty”.
2.2.5.2.	If the email is not valid, the system responds by displaying the error message “Email is not valid”.
2.2.5.3.	If the password length is less than 6 characters, the system responds by displaying the error message “Password should be at least 6 characters”.
2.2.5.4.	If the email has already been registered, the system responds by displaying the error message, “The email address has been taken”.
2.2.5.5.	If all the data are valid, the system will register the account and redirect to the home page.

2.2.6.	The system must refresh the post list immediately when the user gives inputs in the search bar.
2.2.7.	The system must refresh the post list immediately when the user selects a country in the drop-down.
2.2.8.	The system must refresh the post list immediately when the user selects a tag in the tag navigation.
2.2.9.	The system must display the total new notifications in the notification icon on the menu bar.
2.2.10.	The system must display the latest notification items when the user clicks the notification icon.
2.2.10.1	The system must set the background color to blue if the notification item is not read.
2.2.10.2	The system must set the background color to white if the notification item is read.
2.2.11.	The system must redirect to the page with the URL specified in the notification items.
2.2.12.	The system must take the image, type, category, name, details, and quantity and pick up location to perform a successful creating of the post.
2.2.12.1.	If the image is empty, the system responds by displaying the error message, “Please put an image”.
2.2.12.2.	If neither both types are selected, the system responds by displaying the error message “Please select one of the choices above”.
2.2.12.3.	If the category is not entered, the system responds by displaying the error message “Please choose at least 1 tag”.
2.2.12.4.	If the title is not entered, the system responds by displaying the error message “Title cannot be empty”.

2.2.12.5.	If the description is not entered, the system responds by displaying the error message “Description cannot be empty”.
2.2.12.6.	If the quantity is less than 1, the system responds by displaying the error message “Quantity should be at least 1”.
2.2.12.7.	If the Pickup location is not entered, the system responds by displaying the error message, “Pick up location cannot be empty”.
2.2.12.8.	If the data are valid, the system must store the data and redirect the page to the post page.
2.2.13.	The system must take the proposal and the quantity to perform successfully proposal.
2.2.14.	The system must be able to responds thanks sent by the user.
2.2.15.	The system must be able to close the post when the owner of the post requests for it.
2.2.16.	The system must be able to delete the post when the owner of the post requests for it.
2.2.17.	The system must take the category, title, description and quantity to perform successful edit port when the owner of the post request for it.
2.2.18.	The system must take the comment’s message to perform successful comment.
2.2.19.	The system must set the user’s name on the comment box when his or her comment is being replied.
2.2.20.	The system must take the user’s new comment on performing successful edit comment.
2.2.21.	The system must be able to delete the comment when the user requests for it.

2.2.22.	The system must send an acceptance notification to the bidders when the owner of the post accepts the proposal.
2.2.23.	The system's bid must be able to be replied by the post owners and bidders.
2.2.24.	The system's bid must be able to be removed by the bidders.
2.2.25.	The system's profile name must be able to be edit by the profile's owner.
2.2.26.	The system's profile introduction must be able to be edit by the profile's owner.
2.2.27.	The system's profile country must be able to be edit by the profile's owner.
2.2.28.	The system's tag must be able to be searched from the tag navigation search bar when the user gives input to it.
2.2.29.	The system's tag items must be able to be starred by the log-in user.

Table 3- Functional Requirements

2.3. Non-functional Requirements

The non-functional requirements of this projects are:

1. Performance
    1.1. The program should be free from critical errors
2. Reliability
    2.1. The program and server must be available and ready at 99.999% of all time.
    2.2. The server should be able to respond to requests sent by clients at all time and at most 2 seconds.
3. Maintenance
    3.1. The program should able to retrieve the previous version upon program failure during deployment
    3.2. The program should able to trace bugs easily.

4. Software Design

4.1. Code Architecture and Design

In this chapter, the author discusses on how the software is designed and implemented based on the software requirements in Chapter 3. Firstly, the author describes on how the main systems in the application interact with each other. After that, the author describes the design and implementation of each system in details.

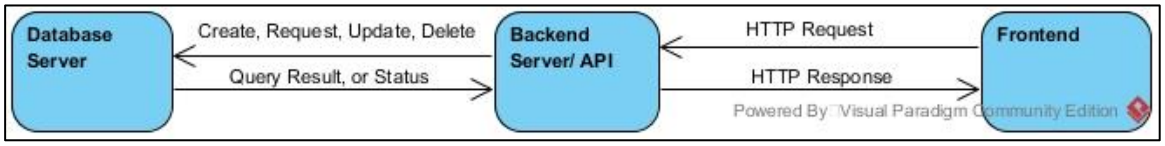


Figure 2- Rest Architecture of Web Application

The commonly used web architecture is usually called as **representational state transfer (REST).** In REST Architecture, communication between client and server is stateless. Stateless communication means that client can communicate in any way with the server without storing the client state on the server. As a result, the web architecture becomes scalable since it provides an abstraction of the server to the client.

In REST Architecture, the server usually contains many REST APIs (Application Programming Interface) that each of them has been assigned a unique Universal resource identifier (URI). Similar to the server, this API is stateless that means the client does not have to store the client state on the API. In addition, the client also does not have to know the structure of the API to use it. As a result, the client only has to decode the response object from the API.

In a Web application, the server is frequently called as Data Access Layer (Back-end) since it is mostly used to access the database. On the other hand, the client is called as the Presentation Layer (Front-end) since it brings user interface and interaction in the browser. Front-end and Back-end usually communicate through HTTP/S protocol in a way that Front-end sends the request to the Back-end API and Back-end API responses with HTML or JSON.

As mentioned before, the server in Web Application is normally referring to the back-end. However, the term **Server** in this report refers to the Virtual Server that is provided by Amazon Web Services (AWS) that is hosting the back-end in the cloud. On the other hand, back-end refers to the application developed from scratch by the author using PHP Yii2 Framework.

The mobile application was also developed using the Web-View technology. However, Web-View technology is actually another web application that is embedded into Native Mobile Application. Therefore, the mobile application will not be discussed in depth.

4.1.1. Back-end

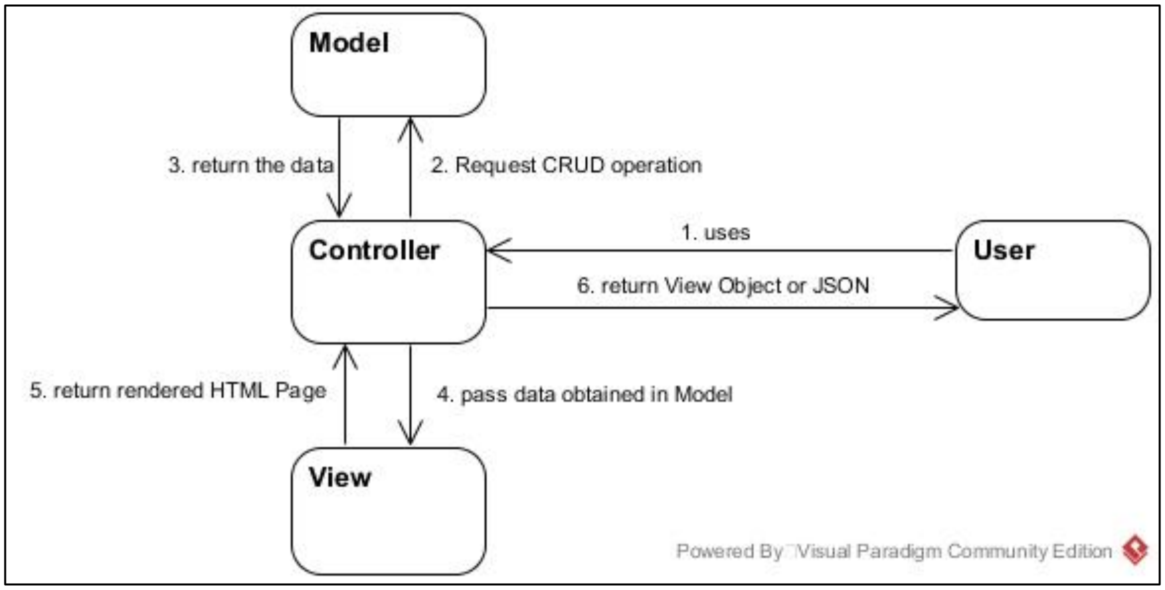

Model
3. return the data
2. Request CRUD operation
Controller
1. uses
6. return View Object or JSON
User
5. return rendered HTML Page
4. pass data obtained in Model
View
Powered By Visual Paradigm Community Edition

Figure 3- Back-end System with MVC Approach

The back-end system in this project is implemented using Yii2 Framework that uses MVC approach. In MVC approach, the back-end is divided into three major components, which are Model, View, and Controller.

In the Controller, Model object is created upon receiving the request. After that, Controller will put the received data into the created Model object. When Model receives the data, it performs a **Server Side Validation.** In Server Side Validation, the Model validates user information, types of data, data requirement and data integrity with the database. If the validation is successful, the model accesses the database and/or returns the operation status to the controller.

In this project, HTML is rendered in the back-end, which is usually called as **Server side rendering**. The rendering process is helped with the use of View Component that is just an HTML template. When an API is expected to return HTML, the controller will use the View Component to get the HTML. After that, the controller creates an HTTP response that may contain HTML or JSON that contains HTML. However, some APIs do not need to return HTML which means that using View Component is not necessary.

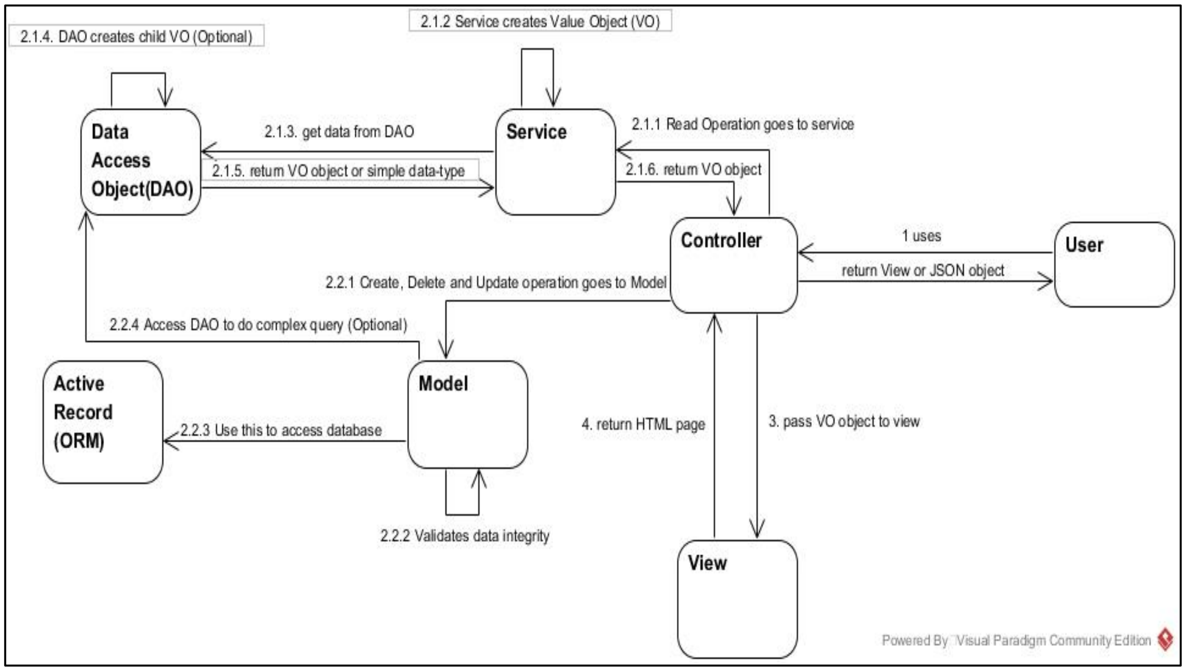

2.1.4. DAO creates child VO (Optional)
2.1.2 Service creates Value Object (VO)
Data Access Object(DAO)
2.1.3. get data from DAO
2.1.5. return VO object or simple data-type
Service
2.1.1 Read Operation goes to service
2.1.6. return VO object
Controller
1 uses
return View or JSON object
User
2.2.1 Create, Delete and Update operation goes to Model
2.2.4 Access DAO to do complex query (Optional)
Active Record (ORM)
Model
2.2.3 Use this to access database
4. return HTML page
3. pass VO object to view
2.2.2 Validates data integrity
View
Powered By Visual Paradigm Community Edition

Figure 4- Back-end system in this project

In MVC Approach, Model has many responsibilities such as validating data, accessing the database, building response object, and inputting data into View template. Therefore, the author decides to divide Model into several components, which are Value Object, Data Access Object, Service, and Model. In this new approach, Service is responsible for building Value/Response Object; Data Access Object is responsible for querying data from the database; Value Object helps Controller to change the View template into HTML; and Model is responsible for data validation before accessing the database. As a result, the readability and scalability of the code are improved significantly because it follows Single Responsibility Principle.

However, this approach requires the controller to decide on using a Service component and/or creating a Model object. When the response object contains HTML, the controller has to access service component to get required Value Object/s (VO). This VO is used to input data to the View component easily. On the other hand, Model component is utilized when data validation before accessing the database is required. Data validation often happens to the APIs that need to modify the database or to authorize users into the system.

In the next several sections, the author will discuss the implementation and design details of each component. Furthermore, the author also provides pseudocode to give an understanding of the design used in this project.

4.1.1.1. **Controller**

When API receives a request, the Controller is the first component to process it. Initially, Controller may perform a simple checking to the received data. For example, some APIs only can be accessed by registered user, therefore, the controller has to reject the request from Guest. After the simple checking is successful, the controller decides on using Service or Model depending on the API.

Another controller's task is to build and send response object to the front-end. This response object may be in the form of HTML, JSON without HTML or JSON with HTML depending on the operation status and the API type.

```
class PostController extends Controller
{
    private $service_factory;
    private $post_service;
    private $profile_service;
    public function init() {
            //initialize all fields
    }
    public function actionIndex(){//API implementation}
    public function actionThanks(){//API implementation}
}
```

Figure 5- Example of Controller Implementation

In Yii2 Framework, each Controller class has a parent class called *Controller* and the Controller's name must have "Controller" as a suffix, For example, *PostController*. Furthermore, the Controller class can create several APIs in the form of class methods (API method). This API method's name must have "action" as a prefix. For example, *PostController* has two API methods, *actionIndex* and *actionThanks*. The API URL is determined by the method's name and the controller's name. For example, the URL of *actionIndex* in *PostController* is "justgivit.com/post/index".

```
public function actionIndex() {
   if(!isset($_GET['id'])) {
      return $this->render('error');
   }
   $post_id = $_GET['id'];
   $vo = $this->post_service->getPostInfo(Yii::$app->user->getId(), $post_id);
   if($vo->getPostStatus() === 0) {
       return $this->render('delete');
   }
   return $this->render('index', ['post' => $vo]);
 }
```

Figure 6- actionIndex obtains VO from Service component

The *actionIndex* API sends HTML that gives user interface for item preview page. Because the response contains HTML, the service called *PostService* that will return the required VO. After that, the Controller uses the View together with the help of VO to render HTML and sends it to the client as a response.

```
public function actionThanks() {
   $data = array();
   if(!Yii::$app->user->isGuest && isset($_POST['stuff_id'])) {
      $model = new ThanksForm();
      $model->user_id = Yii::$app->user->getId();
      $model->stuff_id = $_POST['stuff_id'];
      if($model->thanks()) {
          $this->createNotificationPostThanks($_POST['stuff_id']);
          $data['status'] = 1;
          return json_encode($data);
      }
   }
   $data['status'] = 0;
   return json_encode($data);
}
```

Figure 7- actionThanks API uses model to validate the data

The *actionThanks* API is used to when a particular user sends Thank You to the post owner. However, the API has to ensure the data in the database is not duplicated. Therefore, the controller creates a model called *ThanksForm* to validate the data before accessing the database. Initially, the required data which are the user id and the post id are put into the *ThanksForm*. After that, the *thanks* method is called and returns an operation status. Finally, the controller creates JSON response that contains an operation status from the model.

4.1.1.2. View

Unlike other components, View component is actually an HTML template. The controller uses View and VO to create HTML. The VO is used to structure the data required by the template and input it into the template. This way is better compared in inputting the data to the template one by one.

```
<div class="tag-navigation-item" data-tag_id="<?= $tag->getTagId() ?>">
   <div class="tag-navigation-item-label">
      <?= $tag->getTagName() ?>
   </div>
   <div class="tag-navigation-item-button">
      <?= Html::button('<span class="glyphicon glyphicon-ok"></span>') ?>
      <?= Html::button('<span class="glyphicon glyphicon-star-empty"></span>') ?>
   </div>
</div>
```

Figure 8- Example of View implementation

The example above is the View that creates an HTML for *TagNavigationItem* component in the Front-end. The data required in this template is the tag name and the tag id which is structured in VO called *TagVo*.

4.1.1.3. Value Object (VO)

VO is an entity object used for embedding data into View components since it is more structured and supports type hinting. Moreover, VO object can be either created in Service or Data Access Object. The Service usually creates the **Main VO** that can contain several child VOs object. On the other hand, the data access object creates **Child VO** that is usually placed inside another VO object.

```
class ExampleVo {
    private $a; private $b; private$c ..... ; private $z;
    function __construct($a, $b, $c ..... $z) {
        //initialize
     }
}
```

Figure 9- Injecting data into VO via constructor is messy when there are too many data

Unlike other entity objects, VO only contains getters without setters. VO does not allow setters because the data inside the VO are not supposed to be changed after VO is created. However, the data still has to be injected into the VO class fields in some ways before creation. In Figure 9, the constructor of the VO is used to inject the data. However, using constructor may not be effective when there are many data needed to be injected inside the VO.

A better way is to use Builder factory pattern that creates a temporary object called the builder. Both VO and Builder actually have the same class field. However, Builder contains both getters and setters whereas VO only contains getters. Builder will initially store all the required data using setters and convert it later to VO.

```
class TagVoBuilder implements Builder {
   private $tag_name;
   private $tag_id;
   public function build() {
      return new TagVo($this);
   }
   public function getTagName() {
      return $this->tag_name;
   }
   public function getTagId() {
      return $this->tag_id;
   }
   public function setTagName($tag_name) {
      $this->tag_name = $tag_name;
   }
   public function setTagId($tag_id) {
      $this->tag_id = $tag_id;
   }
}
```

Figure 10- Tag Value Object Builder contains build method, setters, and getters

Figure 10 show that *TagVoBuilder* contains both getters and setters for all fields. After all necessary data are injected into the builder using its setters, the *build* method in the *TagVoBuilder* class is called to convert *TagVoBuilder* into the *TagVo* by passing *TagVoBuilder* to the *TagVo* constructor's parameter.

```
class TagVo implements Vo {
  private $tag_name;
  private $tag_id;
  function __construct(TagVoBuilder $builder) {
    $this->tag_id = $builder->getTagId();
    $this->tag_name = $builder->getTagName();
  }
  public function getTagId() {
    return $this->tag_id;
  }
  public function getTagName() {
    return $this->tag_name;
  }
}
```

Figure 11- In the constructor, Value object assigns the data from the builder to its own field

In *TagVo's* constructor, the data from *TagVoBuilder* is assigned into corresponding fields in *TagVo*. Eventually, Vo Object will contain the same data as Vo Object Builder.

4.1.1.4. **Data Access Object (DAO)**

The data access object provides an abstraction between database and back-end system which helps in hiding the details of the database. Generally, each DAO class has multiple methods which each of them correspond to one database operation. In this project, DAO is mostly used by the Service to return VO object or simple data and sometimes used by the Model to return simple data.

```
function getCountNotification($user_id) {
   $result =  \Yii::$app->db
      ->createCommand(self::COUNT_NEW_NOTIFICATION)
      ->bindValues([':user_id' => $user_id])
      ->queryScalar();
      return (int)$result;
}
```

Figure 12- Data Access Object code implementation

The method above is stored in DAO called *NotificationDao* and used by the *NotificationService* to return the total new notifications to the client. The SQL Statement of this database operation is stored in COUNT_NEW_NOTIFICATION constant of *NotificationDao* class.

4.1.1.5. Service

Service is used to get a VO or simple data which are obtained from the DAO. If Controller specifically requests for Main VO, Service will create VO builder and immediately build the builder after all the data has been put into it. Before building it, Service component has to access some DAOs to get necessary data/Child VO for the Main VO builder.

```
public function getHomeInfo($current_user_id) {
    $builder = new HomeVoBuilder();
    $builder->setPostList($this->home_dao->getAllGiveStuffs($current_user_id));
    $builder->setMostPopularTag($this->tag_dao->getMostPopularTag($current_user_id));
    $builder->setStarredTagList($this->tag_dao->getStarredTag($current_user_id));
    return $builder->build();
}
```

Figure 13- Implementation of Service Component

Figure 13 shows the method in *HomeService* to create a *HomeVo* for particular View template that builds a Home Page. This method initially creates *HomeVoBuilder* and accesses multiple DAOs to get three data that are post list, most popular tag, and starred tag by the current user. Finally, the builder is converted into VO by calling *build* method and returns it to the controller.

4.1.1.6. Model

Model is used to validate the data before accessing the data. This usually happens during database modification (Create, Update, and Delete operation) which often needs to check whether the data meet the requirement before modifying the database. However, Model also can be used in the operation that does not need to modify the database such as Login or Validating Email since they need complex validation.

```
public function actionSignup()
{
  if(isset($_POST['first_name']) && isset($_POST['last_name'])
          && isset($_POST['email']) && isset($_POST['password'])) {
     $model = new SignupForm();
     $model->email = $_POST['email'];
     $model->first_name = $_POST['first_name'];
     $model->last_name = $_POST['last_name'];
     $model->password = $_POST['password'];
     if ($user = $model->signup()) {
     }
  }
}
```

Figure 14- Implementation of Model

The browser calls the API above after the user successfully completes the user registration form. In this API, the *SignUpForm* model is created and the four data received by the API which are email, first name, last name, and password are put into this Model. After that, the sign-up method inside the *SignUpForm* Model is called.

```
class SignupForm extends Model
{
   public $email;
   public $password;
   public $first_name;
   public $last_name;
   public function rules()
   {
            //rules
   }
   public function signup()
   {
      if ($this->validate()) {
         //save user  data in the database
       }
   }
}
```

Figure 15- Model code Implementation

The *SignUpForm* has a parent class called Model provided by Yii2 Framework. This parent class forces the child to override the rule method. Furthermore, the parent class also has a method called *validate* that can be used in *SignUpForm*. If *validate* is called, the requirement that is specified in the *rules* method is checked. If the rules are satisfied, the *validate* method returns true. Otherwise, it returns false.

```
public function rules()
  {
      return [
          ['email', 'email'],
          [['first_name', 'email', 'password'], 'required'],
          [['first_name', 'last_name'], 'string', 'min' => 1],
          ['password', 'string', 'min' => 6],
       ];
  }
```

Figure 16- rules and update method SignUpForm

In Figure 16, the *rule* method shows that the last name field is not required while the other fields are required. Moreover, it also describes that email must be in the form of email while the first name, last name, and password must be in the form of a string. If these requirements are not met, the *validate* method will return false.

4.1.1.7. Back-end Performance Discussion

In this project, the back-end performance is optimized through balancing the use of a single big query statement and multiple small queries for the database and helping the Front-end to reduce the image size need to be downloaded.

Choosing the use of Single Big query and Multiple Small Queries in the database is tricky since the result may vary across multiple situations. In many cases, a Single big query is faster since the connection time to the database is only one time (Θ (1)). On the other hand, Multiple N queries can lead to poorer performance since the connection time of the database is N times (Θ (N)). However, if the query results have many duplicate columns, the big queries sometimes can lead to poorer performance because the database has to send a big chunk of data to the back-end. The author's strategy is to use single big query as long as there is no data duplication in the query result. Otherwise, the author uses multiple N queries.

The images that are stored in the system usually have unpredicted size and downloaded to the Front-end when it is requested. When the image size is too big, this method is not so effective since the downloading time increases proportionally with image size. To solve this, the back-end provides an API that allows front-end requests the image based on the width and height. If the image files with the specified height, width, and quality do not exist, the back-end API takes the original file and manipulate (cropping and resizing) the original file until it has the same width and height. After that, the image file is cached in the system and it is sent back to the client.

4.1.2. Front-end

In the Front-end, Justgivit.com adopts Component-based Approach that divides the entire HTML page into many reusable components. In many cases, the component can contain one or more components that form a parent-child relationship.

Creating one component requires a CSS File, a JavaScript Class, and Component's View. This View is actually the same with the View component in the back-end. In the parent-child relationship, the parent's view generates the child's view and Child's JavaScript class becomes the field of Parent's JavaScript class (See Figure 17 & 18).

```
var Site = function($root) {
    this.$root = $root;
    this.search_bar = null;
    this.init();
    this.initEvents();
};
Site.prototype.init = function() {
    this.search_bar = new SearchBar(this.$search_bar);
};
```

Figure 17- Search bar is a child component of Site, therefore, it becomes a field in the Site JavaScript class

```
<div class="site-index">
    <div class="site-post-area-header">
        <?= SearchBar::widget(['id' => 'site-search-bar'])?>
        <?= CountrySearch::widget(['id' => 'site-country-search']) ?>
    </div>
</div>
```

Figure 18- Site Component's view generates the view of its child component (Search Bar and Country Search)

4.1.2.1. Event Delegation in JavaScript

The interaction between component and user is usually done using Event Binding. In Event binding, a call-back function is registered to an event in one element during page initialization. When this element’s event is triggered, the subsequent call-back function is called.

```
this.$register_button.on( 'click', function(e) {
   this.submitRegisterForm();
}.bind(this));
```

Figure 19- A function is registered to a click event of register button element

However, registering a function to element's event can be tricky because some elements do not exist during page initialization. The solution of this problem is called Event Delegation. In Event Delegation, the call-back function is not registered to the target element' event but to the parent of target's element which exists during page initialization. This can be done because element's event is also triggered when its child with the same event is called.

```
$(document).on('click', '#' + this.id , function(e) {
      if(e.target && $(e.target).closest(".tag-navigation-item").length === 1) {
           //execute the function
      }
   }.bind(this));
```

Figure 20- Implementation of Event Delegation in TagNavigationItem

When a user searches for tags in the home page, the server sends *TagNavigationItem* components to the client. This means that these *TagNavigationItem* components only exist after the user searches for the tags. Therefore, the click event should not be bind to *TagNavigationItem* element but to the Document (root) element that exists during page initialization. When a click event is triggered in document element, it will check whether the event comes from *TagNavigationItem*. If it occurs from *TagNavigationItem* element, it will execute the function.

4.1.2.2. Component Communication in JavaScript

All components can communicate to each other by combining parent to child and child to parent communication through JavaScript class. Parent-Child communication means that the parent component wants the child component to do a certain action. On the other hand, Child-Parent communication means that the child component wants the parent component to do a certain action.

In Parent-Child communication, the communication is relatively simple since the parent component class has the child component class as a field. Therefore, parent component can access the child component's method to communicate.

On the other hand, Child-Parent communication can be done through the call-back mechanism. Firstly, the child component creates a custom event that is triggered when the user interacts with this child component. After that, the parent component passes the callback function to this event. This callback function will be called when this child component's event is triggered.

a. Implementation of child-parent communication

TagNavigationItem is a child component of the *TagNavigation* component. When the user clicks a TagNavigationItem element, the *TagNavigation* must know which TagNavigationItem is clicked to refresh the list of the post on the home page.

```
TagNavigationItem.prototype.initWidgetEvents = function() {
   this.tick_event_ = new
                        CustomEvent(TagNavigationItem.EVENTS.TAG_NAVIGATION_ITEM_TICK);
};
TagNavigationItem.prototype.triggerTickEvent = function(tag_id) {
   $("#" + this.id).trigger(TagNavigationItem.EVENTS.TAG_NAVIGATION_ITEM_TICK, [tag_id]);
};
```

Figure 21- TagNavigationItem creates an event called TAG_NAVIGATION_ITEM_TICK

During initialization, the tick event is created in *TagNavigationItem* component using *CustomEvent* class. To trigger this event, the class has to call *triggerTickEvent* method.

```
$(document).on(TagNavigationItem.EVENTS.TAG_NAVIGATION_ITEM_TICK,
                   "#" + this.id, this.tickCallback());
```

Figure 22- In TagNavigation, the call-back function is registered to TAG_NAVIGATION_ITEM_TICK event.

In the parent component, the parent will pass a callback function called *tickCallback* that will be called when the tick event is triggered in the child component.

b. Implementation of parent-child communication

```
Loading.prototype.hide = function() {

   this.$root.hide();

};

Loading.prototype.show = function() {

   this.$root.show();

};
```

Figure 23- Loading component two main methods

In parent-child communication, the author uses loading component as an example. This component is used in many places as a child component to show the waiting state of its parent. Moreover, loading component has two main methods called show and hide that shows and hides the loading bar.

4.1.2.3. The Design of HTML and CSS Code

Designing HTML and CSS code to be readable are essential since the codes from both languages are easy to mess up if certain rules are not followed.

One of the ways to improve the readability of the HTML code is not to mix the PHP and HTML code too much. This can be done by limiting the PHP code in the View Component. In other words, View component is only served as a template. The PHP code in the View component is only limited to the Value Object and if-else condition that sometimes needed.

In the HTML, the naming of the classes and ids of elements should always start with the file name. This helps in avoiding name duplication that can lead to unseen error in JavaScript and CSS.

4.1.2.4. Front-end Performance Discussion

In this project, the front-end's performance is optimized by reducing the size of files that should be downloaded by the browser. This could be achieved in several ways such as combining multiple JS and CSS files into a single JS and CSS file, minimizing the JavaScript, HTML, and CSS file, reducing the size of the image before downloading it and placing the JavaScript code after the HTML pages.

In this project, Closure compiler is used to minimize and combine JS files whereas Yui Compressor is used to minimize and combine CSS files. Furthermore, Gulp is used to automate both these tasks into a single line of command. This eases the process of deployment to the cloud since developers do not have to do the compressing of JS and CSS file one by one.

Loading Time	Before Optimization	After Optimization
1st Trial	5.00s	4.01s
2nd Trial	4.87s	3.92s
3rd Trial	5.02s	4.03s
4th Trial	4.95s	4.27s
Average	4.96s	4.06

Table 4 - The Loading time is faster by 0.9s after minifying and combining CSS and JS files tested in Localhost.

4.1.3. Database

In this project, Database follows the relational approach in which each table has a relation to another table through the foreign key. Furthermore, the database has been checked to be at least third normalized form that helps in ensuring that there is no column-duplication in the implementation.

The database has 24 tables that can be further divided into 7 major components, which were Location System, Image System, Login System, Posting System, Tagging System, Bidding System, and Notification System. In the next few subsections, all these systems will be described in details except Notification System. The notification system will be explained in the next chapter.

4.1.3.1. Location System

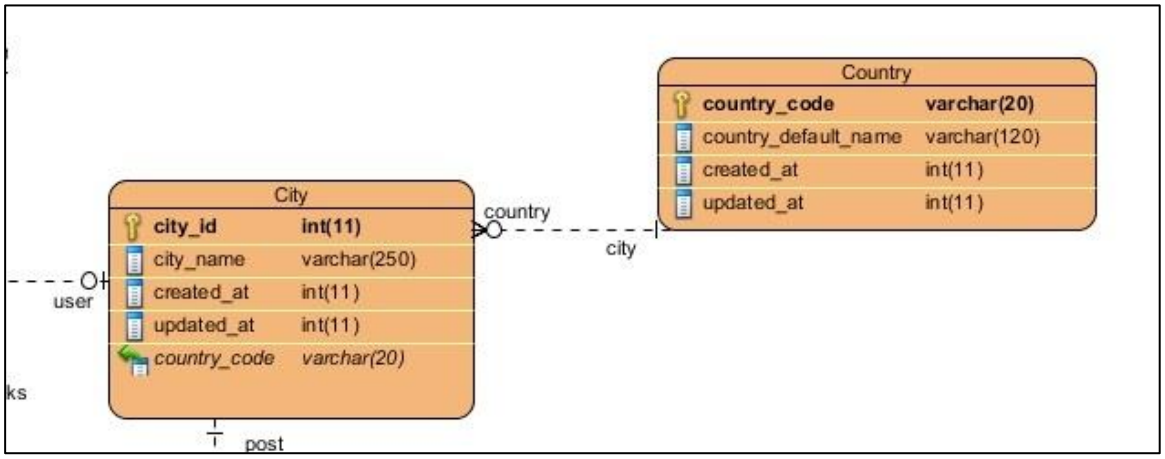


Figure 24: Location System in Database

This project defines City as the smallest location entity. Moreover, each city is associated with one country and one country can have multiple cities. Therefore, city and country have a many-to-one relationship.

4.1.3.2. Image System

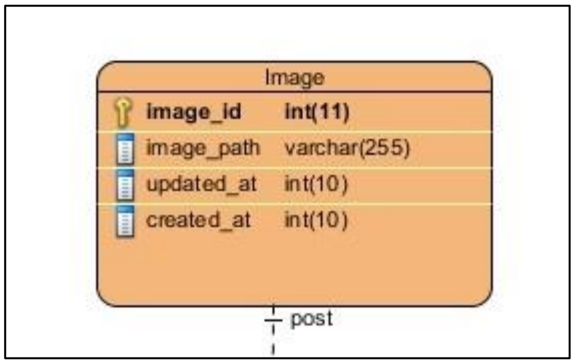


Figure 25- Image System in Database

Image System has only one table which is *Image* table. Furthermore, this table contains a path to the location and an image identity (Id). This Id is used to ease the joining process with another table.

4.1.3.3. User Login System

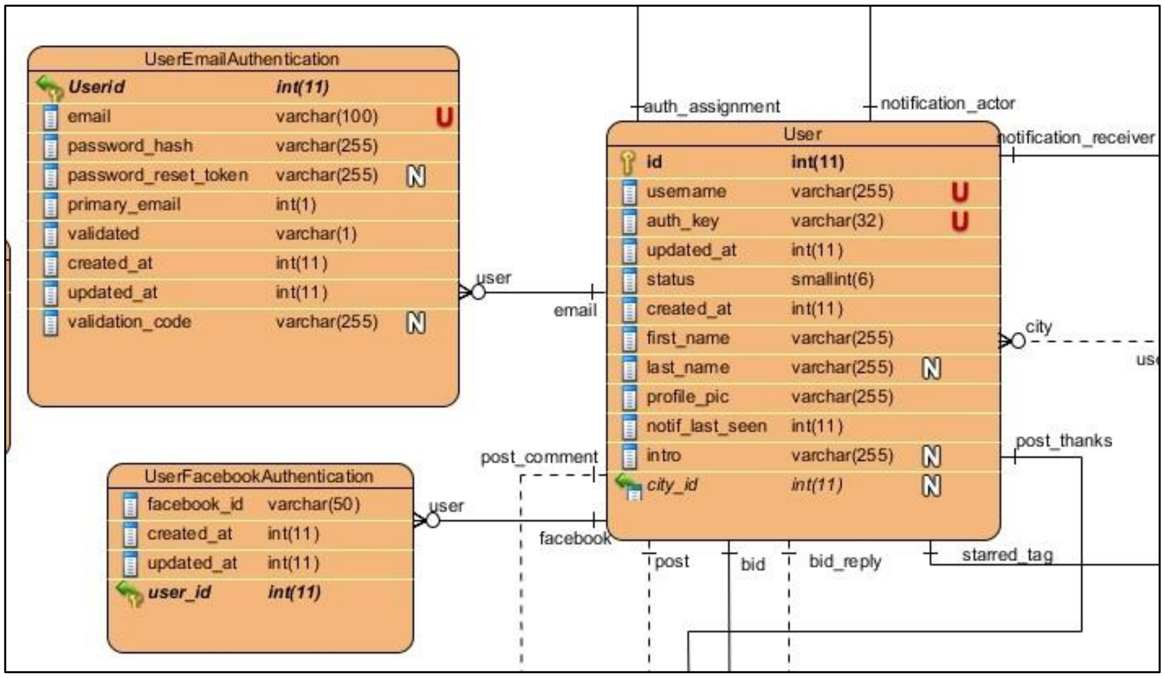


Figure 26- User Login System in Database

User Login System comprises of three tables, which are *User*, *UserEmailAuthentication*, and *UserFacebookAuthentication*. *User* table contains the primary information of the user. Each user can have multiple emails but each email is only associated with one user. Therefore, *UserEmailAuthentication* and *User* table have a many-to-one relationship. Furthermore, each user also can have multiple Facebook account but each Facebook account can only be associated with one user, therefore, *UserFacebookAuthentication* and *User* have a many-to-one relationship.

UserEmailAuthentication table contains several columns such as:

1. Hashed password: The password associated with the email for login. The password is hashed to ensure the privacy of the user
2. Password reset token: it contains 64 random characters when a user wanted to reset the password. Otherwise, this field is set to NULL
3. Primary Email: If this email is primary email to the user, this attribute is set to 1. Otherwise, it is zero. This primary email receives notification from the system.
4. Validated: the email has to be validated to ensure that the email belongs to the user.
5. Validation code: This attribute is used to authenticate the link to validate the email.

UserFacebookAuthentication table stores the Facebook id of the user. When the user clicks Login with Facebook, the Facebook system return a Facebook Id of the current user. The user that has this Facebook id in *UserFacebookAuthentication* is logged in.

4.1.3.4. Tagging System

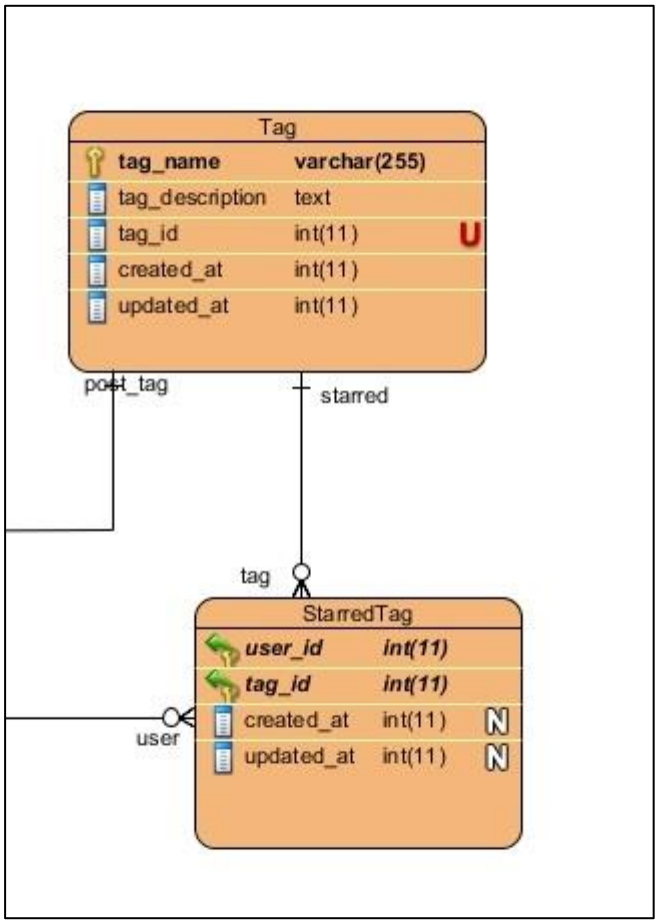


Figure 27- Tagging System in Database

Tag table has multiple columns such as:

1. Tag Name: The name of the tag
2. Tag Description: the description of the tag
3. The id of the tag: the id of the tag to ease joining with another table

Each user can star many tags and each tag can be starred by many users. This shows that *User* and *Tag* have a many-to-many relationship and the association is stored in *StarredTag* table. The *StarredTag* table stores the id of the user and the id of the tag.

4.1.3.5. Posting System

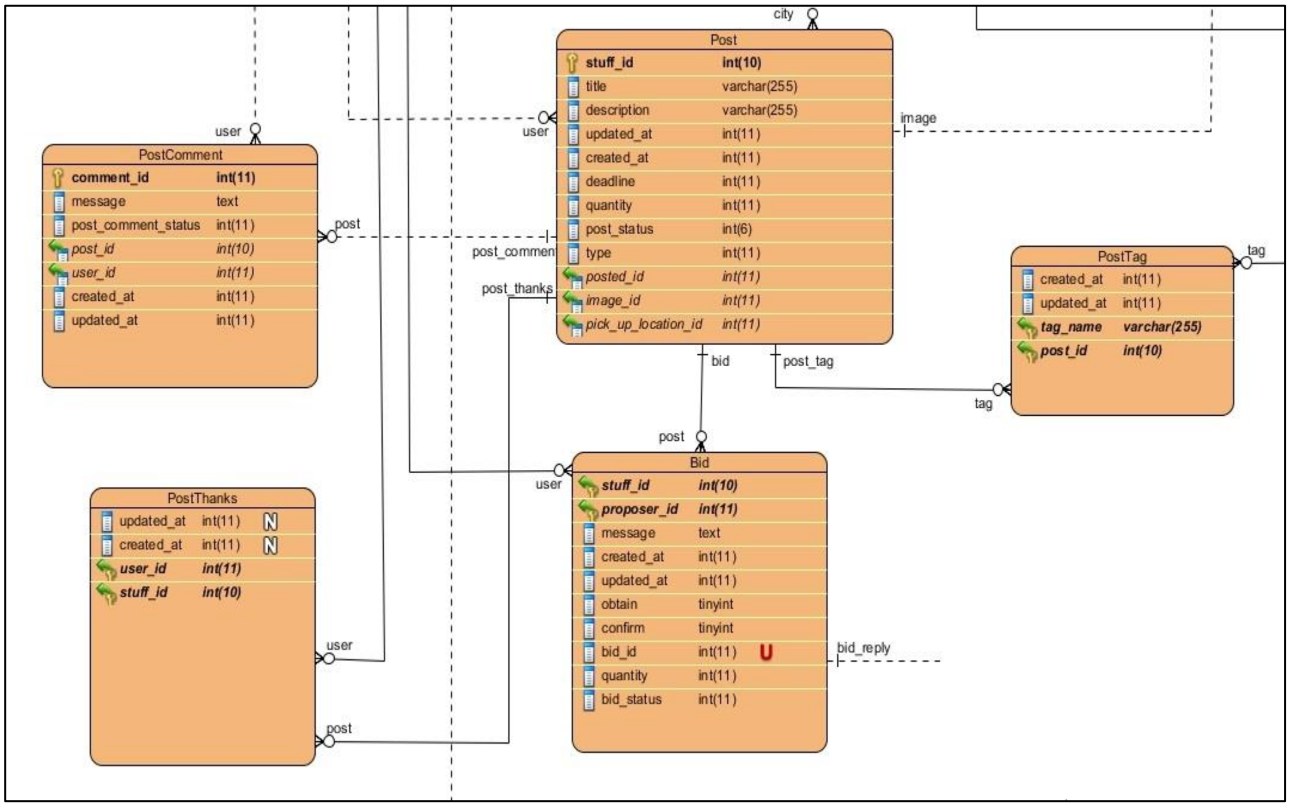


Figure 28- Posting System in Database

Posting Database System contains five tables which are *Post*, *PostTag*, *Bid*, *PostComment*, and *PostThanks* table.

Post Table contains multiple columns that are:

1. Stuff Id: The id of the post
2. Title: the title of the post
3. Description: the description of the post
4. Quantity: the quantity of the stuff asked/requested
5. Type: the type of the post (Ask/ Request)
6. Deadline: the duration of the post will remain opened
7. Image Id: the id in the image table that will give the path of the post image

8. Pick up location Id: the id of the city in which the owner preferred to pick up the stuff
9. Poster Id: the id of the user that creates the post
10. Post Status: the status of the post (closed, active, delete, or banned)

Each post can contain multiple comments and each comment is only associated with one post. Therefore, *PostComment* has a many-to-one relationship to *Post* table.

PostComment table contains multiple attributes, which are:

1. Comment Id: the id of the comment
2. message: the message of the comment
3. Comment Status: the status of the comment (active/deleted)
4. Post Id: the id of the post
5. User Id: the id of the user who comments on the post

Multiple users can give thanks to a post and each user can give multiple posts. Therefore, *User* and *Post* for thanks have a many-to-many relationship. *PostThanks* table is created to store the id of the user and the id of the post.

Each post can have multiple users to bid and each user can bid in multiple posts. Therefore, *User* and *Post* have a many-to-many relationship in bidding event. The Association is stored in *Bid* Table. Further explanation of bid table will be explained in the next section.

4.1.3.6. Bidding System

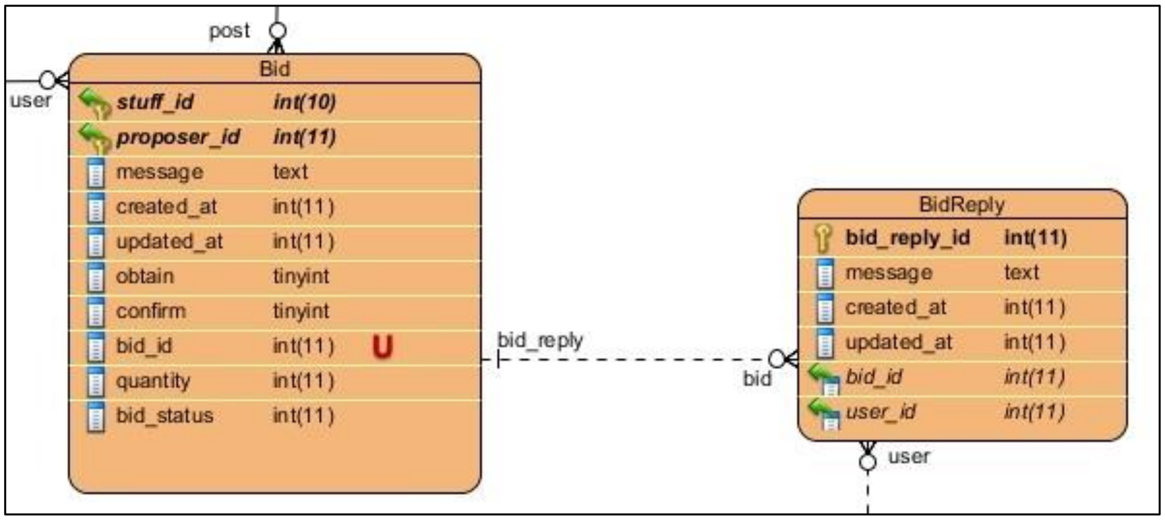


Figure 29- Bidding System in Database

Only Bidders and Owners can reply in the bid and this reply message is stored in *BidReply* table. Each bid can have multiple replies and each reply is associated with one bid, therefore, *BidReply* and *Bid* table have a many-to-one relationship.

Bid Table contains multiple attributes such as

1. Stuff Id: the id of the stuff being bid
2. Proposer Id: the id of the user who bid on this post
3. message: the proposal message to convince the owners
4. obtain: this attribute is set when the owner has approved the bidder's proposal
5. confirm: the bidder has confirmed that s/he received the stuff
6. quantity: the quantity of the items in the post being given/requested
7. Bid Id: the id of the bid to ease the join process

4.1.4. Server

The server plays an important part in Application since the Front-end, Database and Back-end are initially stored in the Server. This project is using cloud service called Amazon

Web Service to host and deploy the application. Currently, there are at least two servers that were used in AWS that are EC2 (Elastic Compute Cloud) to host the Application Service and RDS (Relational Database Service) to host the Database.

EC2 provides a virtual environment to control the server that is provided by Amazon. Furthermore, EC2 also enables a feature to resize the server capacity according to our needs, which led to effective cost for developers.

To access the virtual server, the author uses GIT Bash, which is a console-like command prompt that combines with Git feature. In the Virtual Server, Linux is used as an Operating System and Apache is used as the HTTP server.

On the other hand, RDS is a database server that also supports resizable capacity like EC2. In addition, it also provides support to launch one or more replicas that help users to scale the database.

In the next few sections, the author explains of some designs or implementations that are used on the server to improve the security and performance of the application.

4.1.4.1. Secured Connection through HTTPS

In HTTP protocol, the data that is sent from browser to server is not secured. This can lead to disclose User private information to the intruders. To solve this, the connection between browser and server has to be encrypted using HTTPS protocol.

In HTTPS protocol, the data is encrypted through asymmetric Public Key Infrastructure (PKI). In Asymmetric PKI, the private key is secured in the Server and the public key of the Server is sent to the trusted SSL Certificate Authority (CA). Before sending the data to the server, the client encrypts the data using a public key that is obtained from the CA. Because they are encrypted, the data are secured since it cannot be decrypted unless using the server’s private key.

4.1.4.2. GZIP Connection

GZIP connection has the same principle with ZIP technology in Operating System. The main difference is GZIP is used to compress data on the cloud communication while ZIP is used to compress data for inter-OS communication. GZIP will compress the data before sending them to the client and client will decompress it back to the original data. This helps in improving the performance since the size of the data is smaller.

4.2. UI Design

In this section, the author shows the result of the UI implementation based on the Software Requirement in Chapter 2.

4.2.1. Login

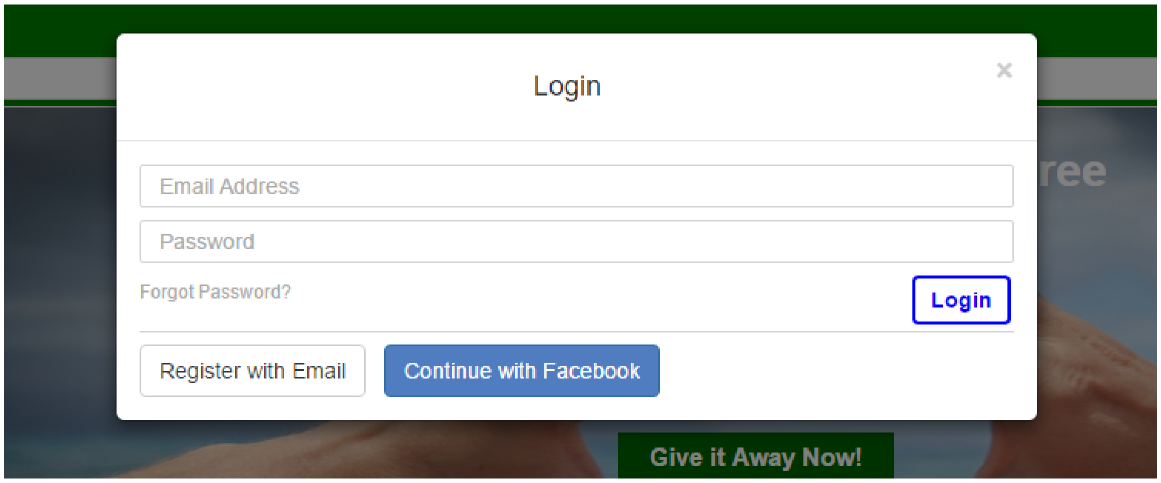
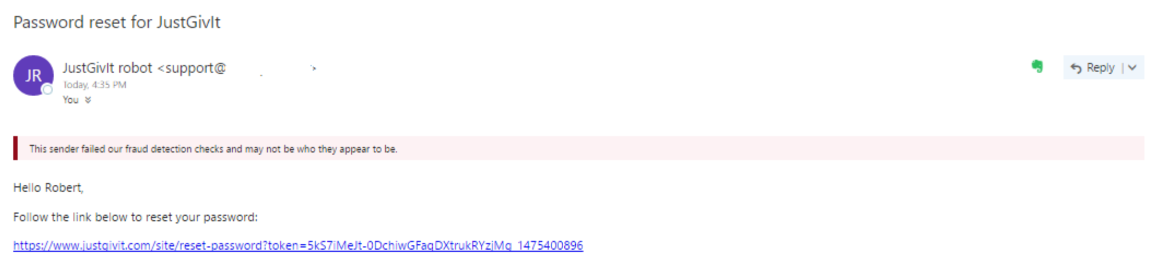


Figure 30- User Interface for Login

4.2.2. Forgot Password



Password reset for JustGivIt

JR JustGivIt robot <support@

Today, 4:35 PM

You

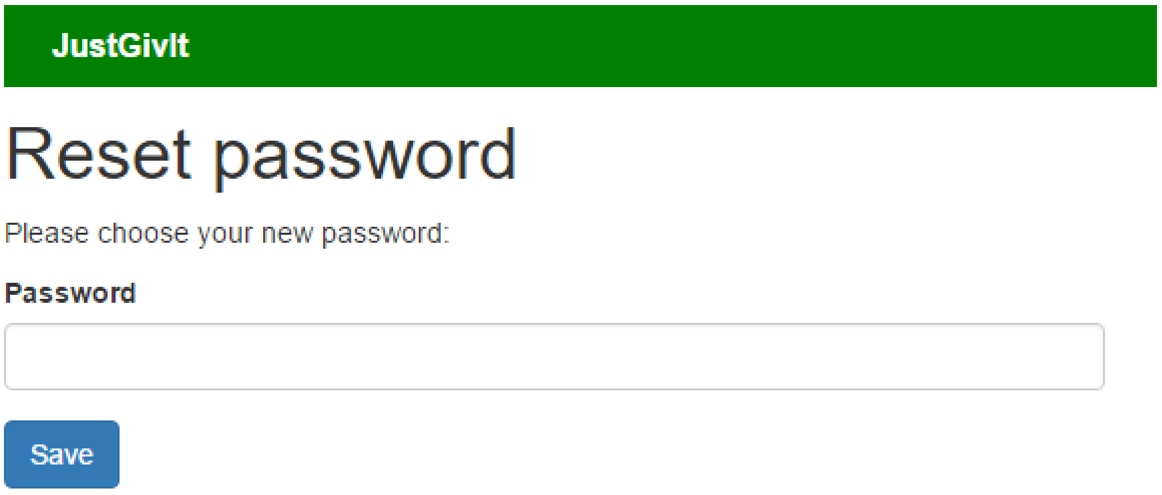
This sender failed our fraud detection checks and may not be who they appear to be.

Hello Robert,

Follow the link below to reset your password:

https://www.justgivit.com/site/reset-password?token=5kS7iMeJt-0DchiwGFaqDXtrukRYzjMq_1475400896

Figure 31- User Interface for Forgot Password Email



JustGivIt

Reset password

Please choose your new password:

Password

Figure 32- User Interface for changing Password

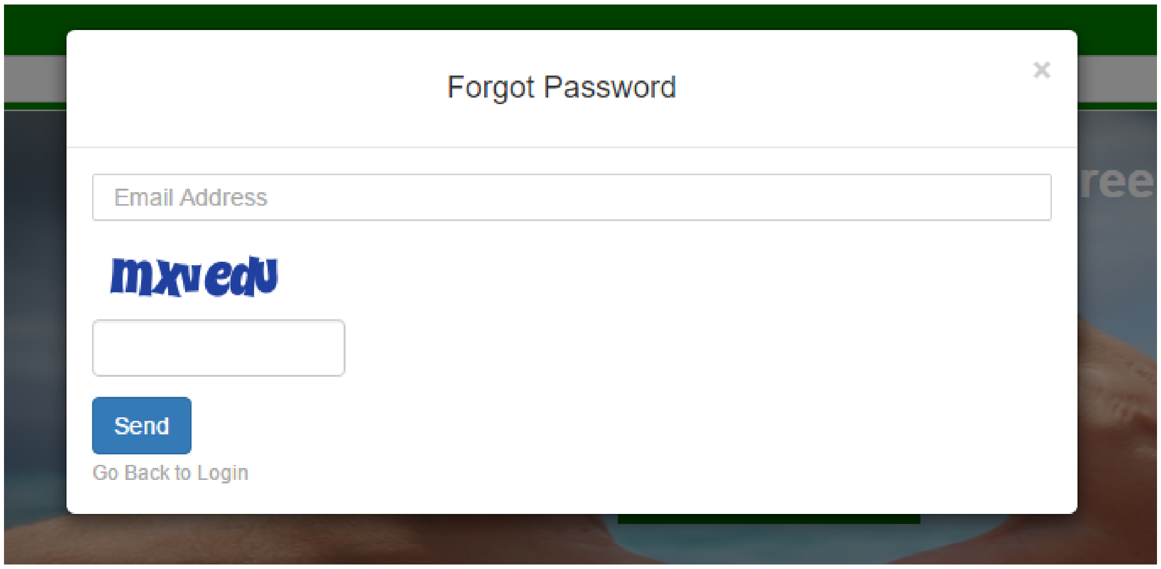


Figure 33- UI for Forgot Password Form

4.2.3. Registration

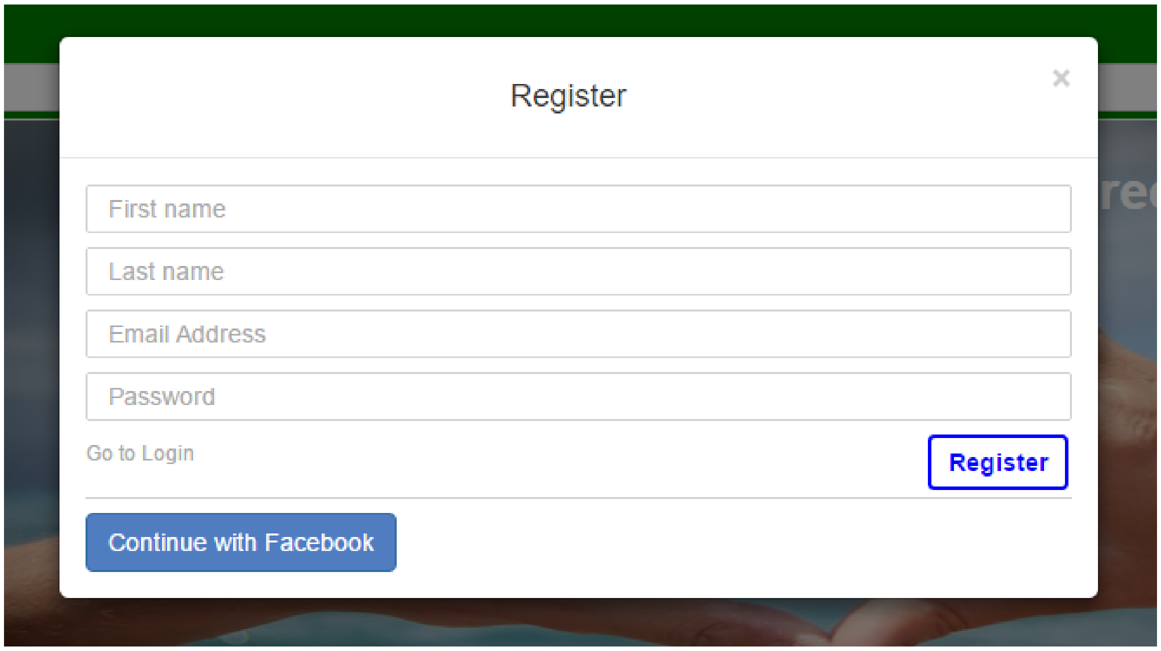


Figure 34- UI for Registration Form

4.2.4. Search

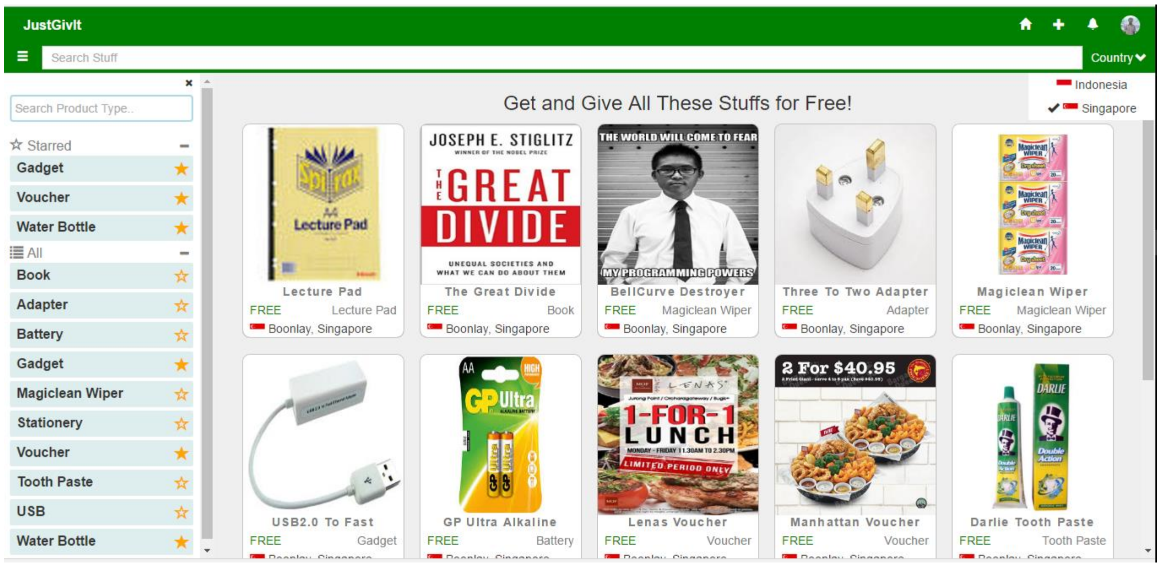
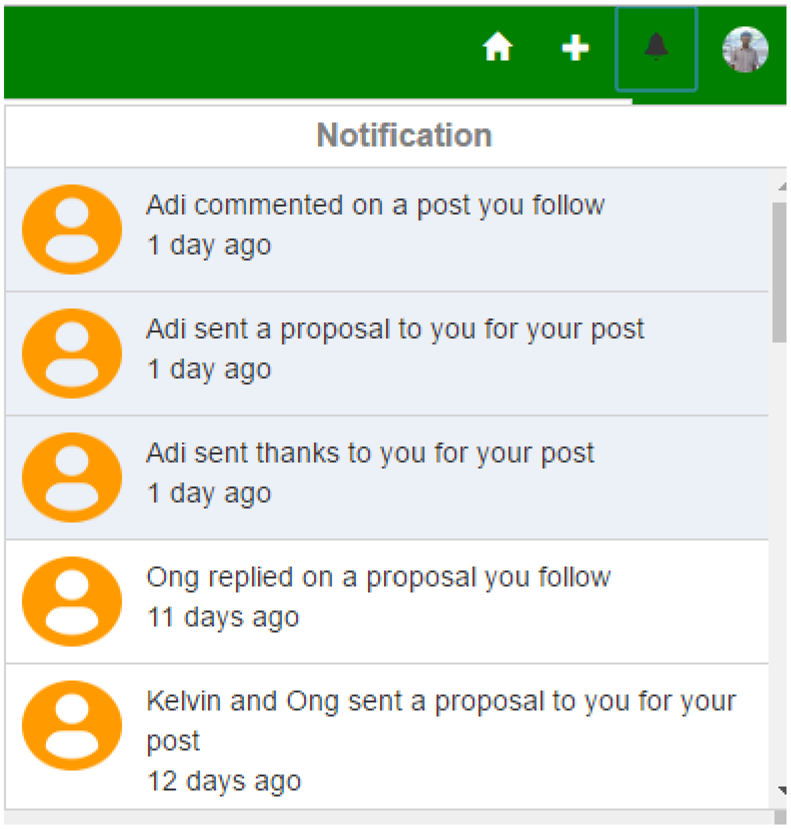


Figure 35- UI for Search bar and Country Search (Topside)

4.2.5. Notification



Notification

Adi commented on a post you follow
1 day ago

Adi sent a proposal to you for your post
1 day ago

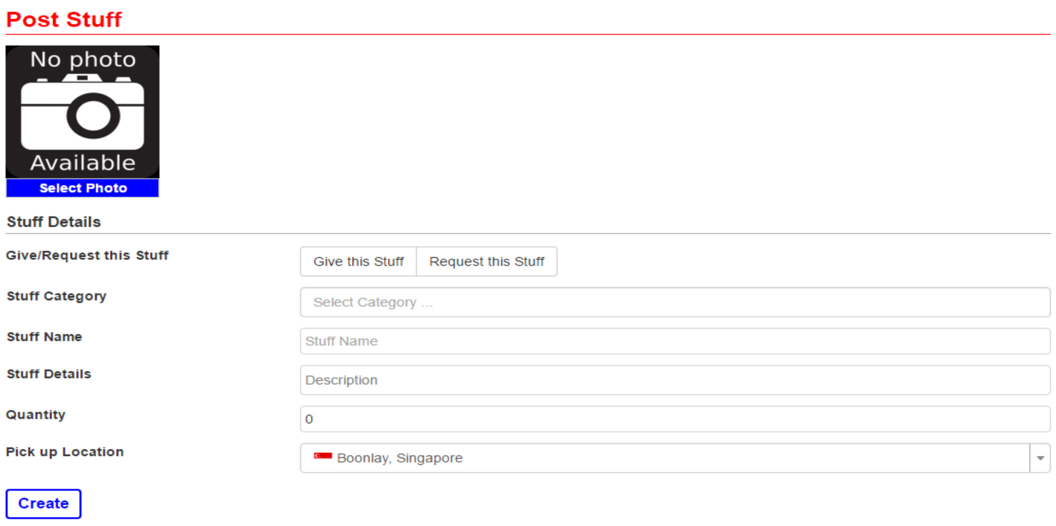
Adi sent thanks to you for your post
1 day ago

Ong replied on a proposal you follow
11 days ago

Kelvin and Ong sent a proposal to you for your post
12 days ago

Figure 36- User Interface for Notification List

4.2.6. Create Post



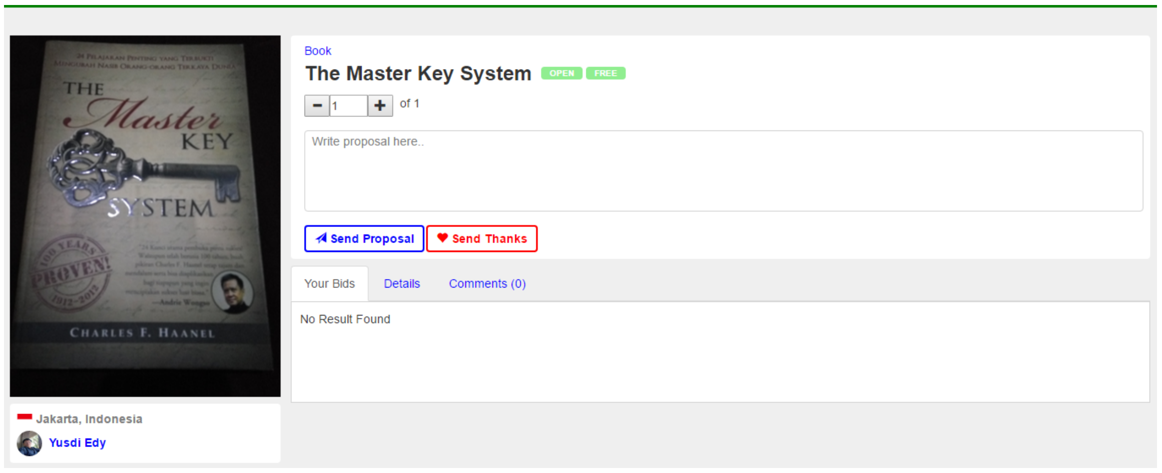
Post Stuff

Stuff Details

Give/Request this Stuff	Give this Stuff / Request this Stuff
Stuff Category	Select Category ...
Stuff Name	Stuff Name
Stuff Details	Description
Quantity	0
Pick up Location	Boonlay, Singapore

Figure 37- User Interface for Creating Post

4.2.7. Post



Book

The Master Key System OPEN FREE

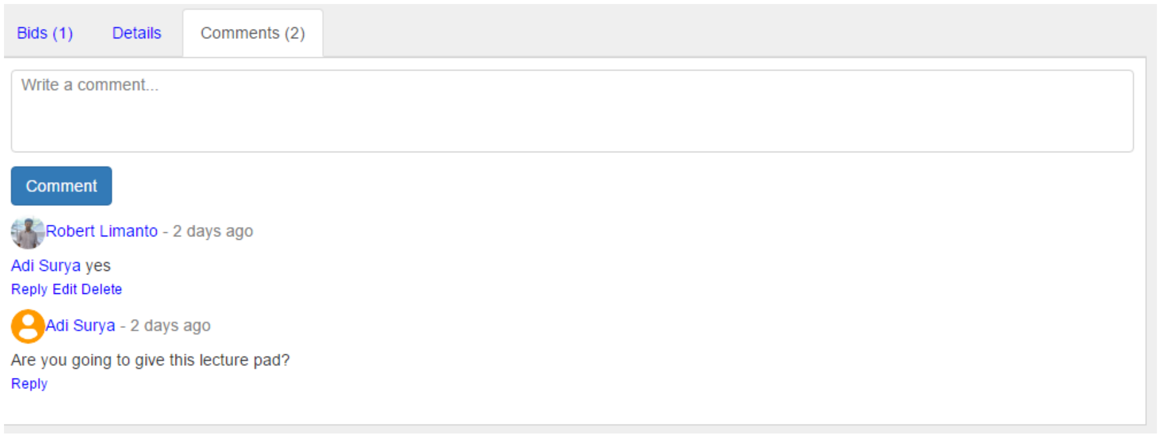
No Result Found

Jakarta, Indonesia

Yusdi Edy

Figure 38- User Interface for Post

4.2.8. Comment



Robert Limanto - 2 days ago

Adi Surya yes

Reply Edit Delete

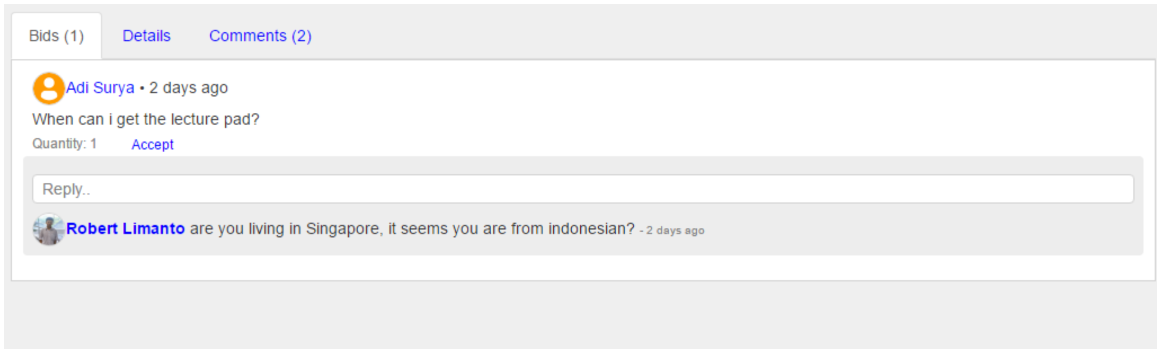
Adi Surya - 2 days ago

Are you going to give this lecture pad?

Reply

Figure 39- User Interface for Comment

4.2.9. Bid



Adi Surya • 2 days ago

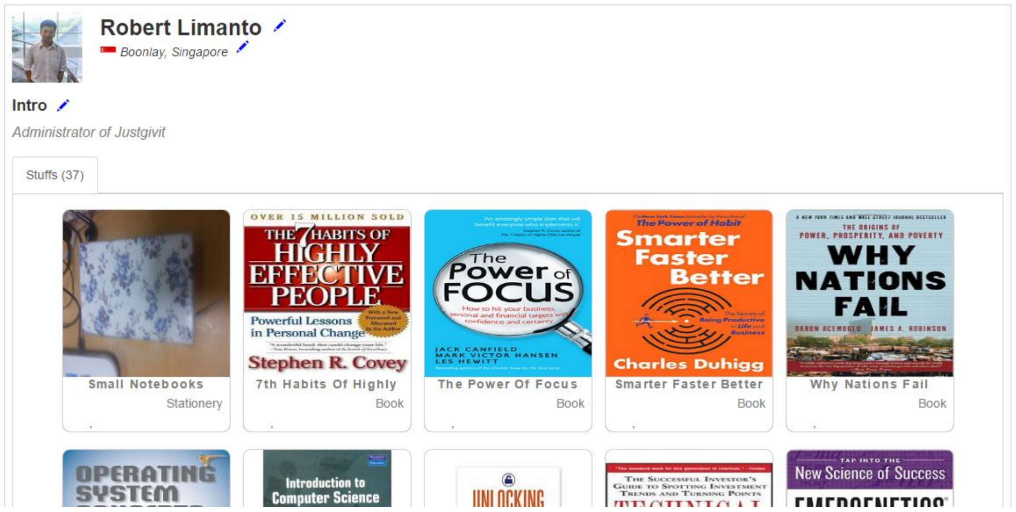
When can i get the lecture pad?

Quantity: 1 Accept

Robert Limanto are you living in Singapore, it seems you are from indonesian? - 2 days ago

Figure 40- User Interface for Bidding

4.2.10. Profile


Robert Limanto

Boonlay, Singapore

Intro

Administrator of Justgivit

Stuffs (37)

Small Notebooks	7th Habits Of Highly	The Power Of Focus	Smarter Faster Better	Why Nations Fail
Stationery	Book	Book	Book	Book

Figure 41- User Interface for Profile Page

4.2.11. Tag

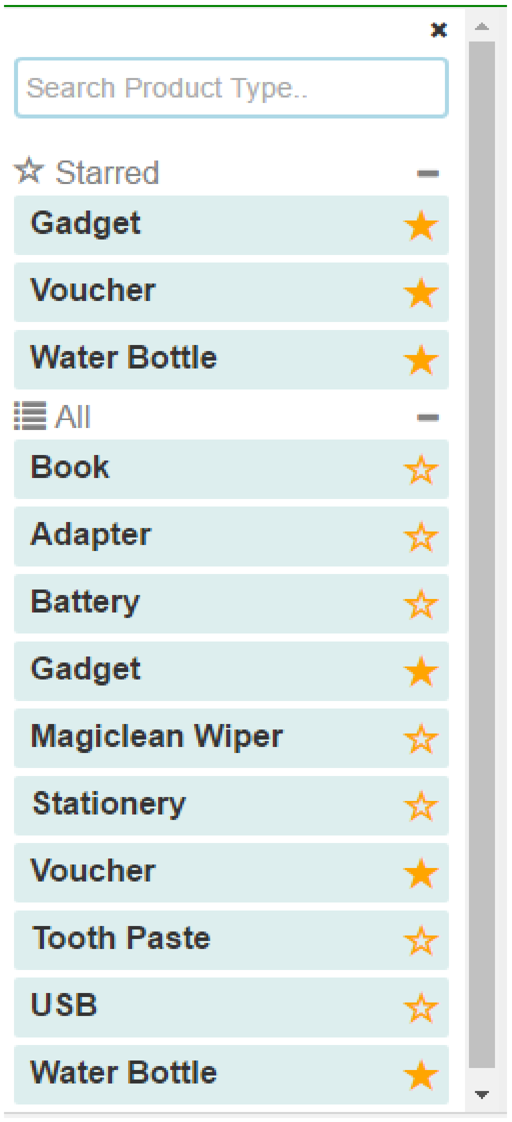


Figure 42- User Interface for Tag System

4.2.12. Responsive Layout

The width range of the layout that is supported in Justgivit.com started from 320px. This makes the web application can also be viewed from mobile and tab devices.

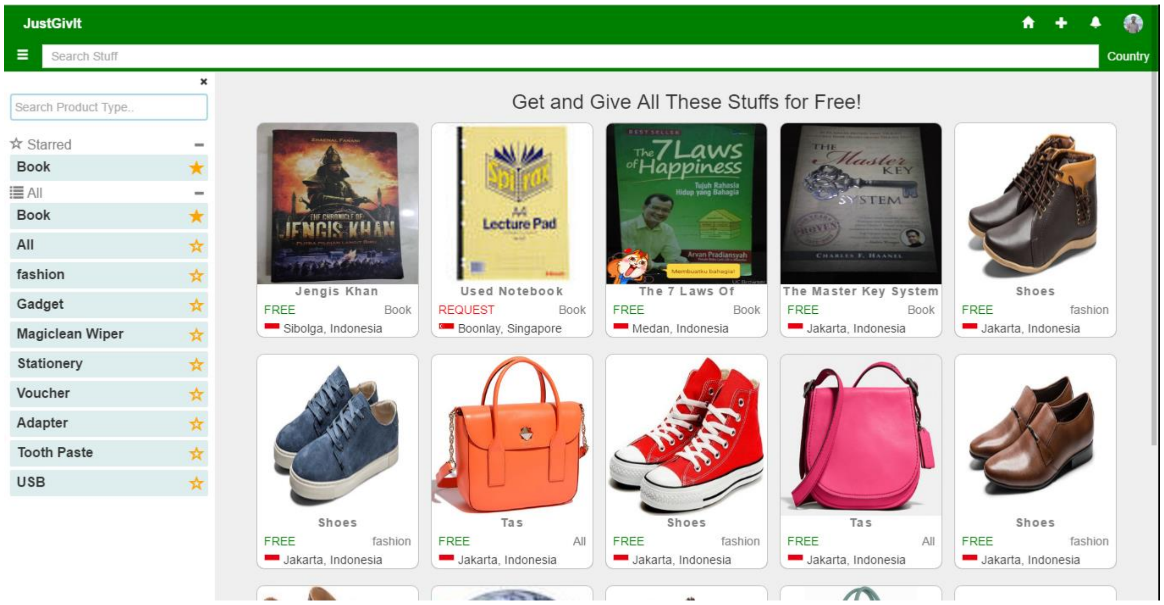


Figure 43 - User Interface for Desktop View

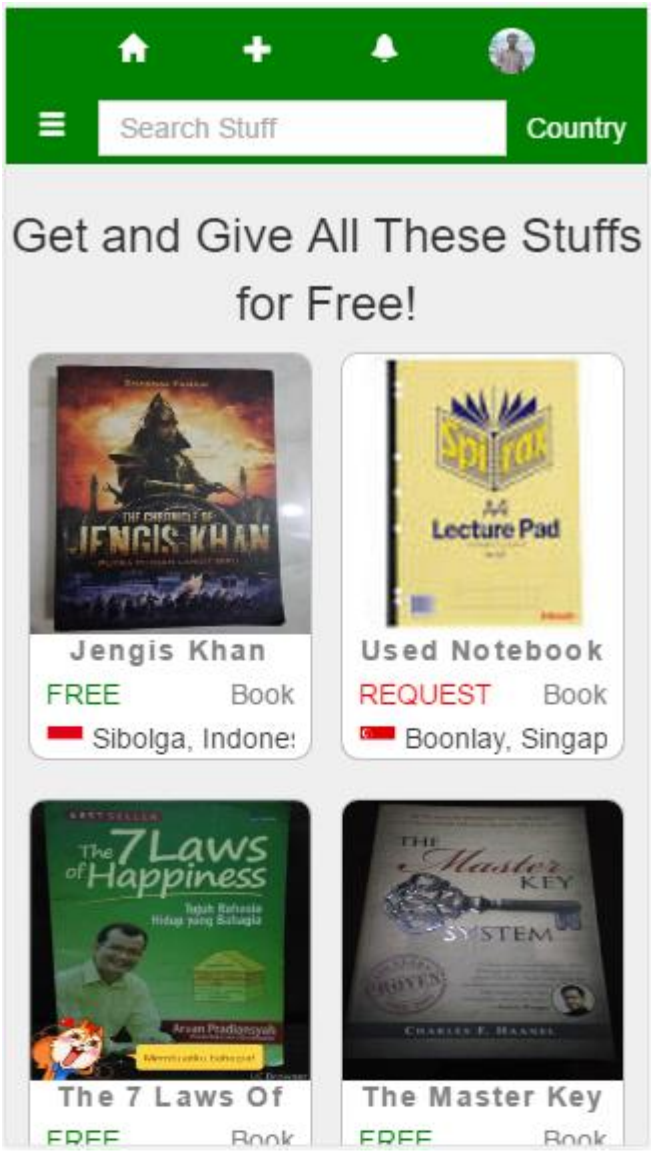


Figure 44- User Interface for Mobile View

5. Software Implementation

Notification is used to notify users about the progress of their posts that they follow. This includes the Thank You Notification sent to the post owners, notification when someone replies or comments on the post related to them and notification about whether the owner has accepted user's proposal.

This Notification system of this project is similar to that of Facebook, which allows multiple receivers and multiple actors at the same time. Each Notification in the Notification List usually carries a notification text and a URL. When the user clicks on the notification item or notification text, the system redirects to the URL according to the notification item.

In the next several sections, the author will discuss Database, Back-end, and Front-end of the Notification in details.

5.1. **Database**

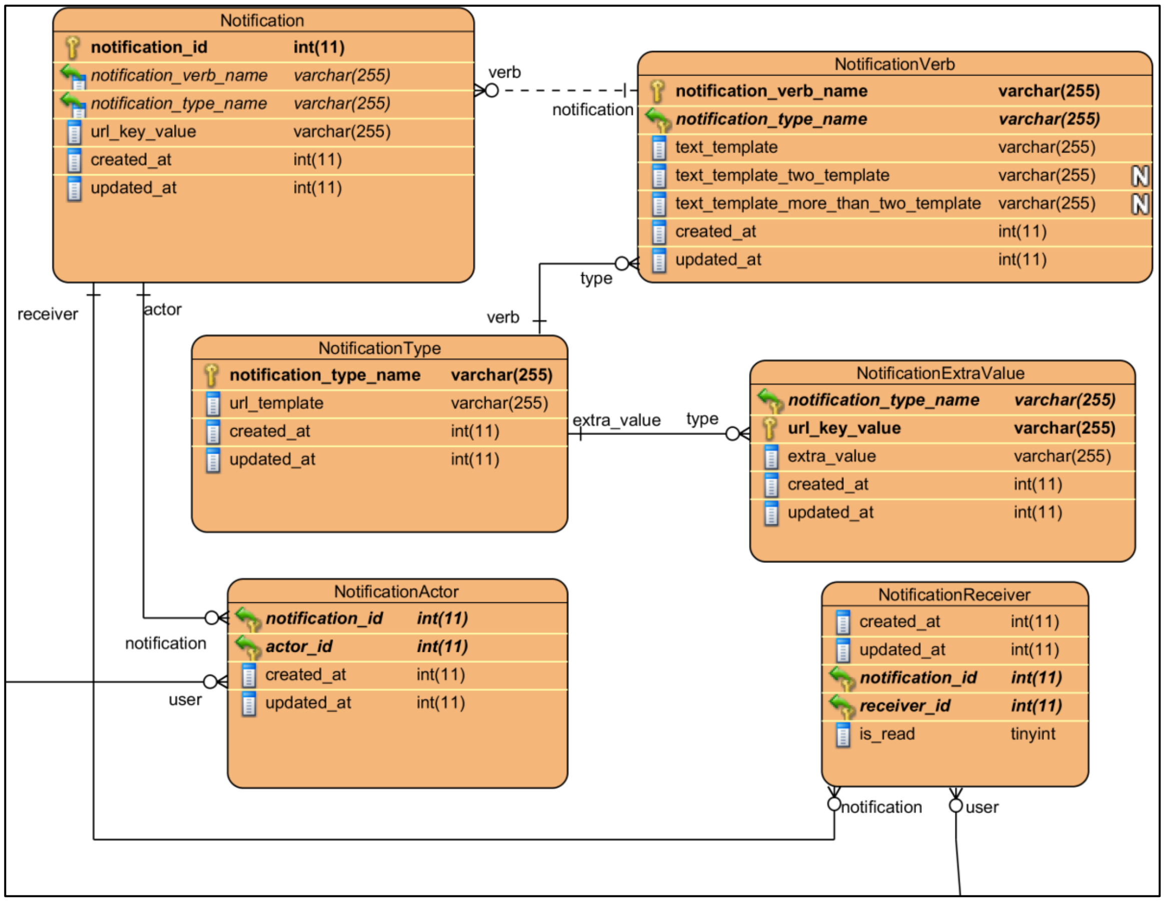


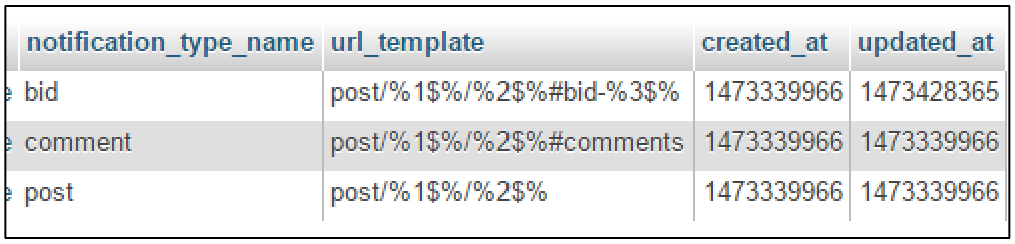
Figure 45- Notification System in Database

The Notification is made up of six tables, which are Notification, NotificationType, NotificationVerb, NotificationActor, NotificationReceiver, and NotificationExtraValue.

Each Notification can have multiple users as actors and each user can become an actor in multiple notifications. Therefore, the user that is an actor and notification has a many-to-many relationship and this relation is preserved in NotificationActor table.

Each notification can have multiple users as receivers and each user can become a receiver in multiple notifications. Therefore, the user as a receiver and notification have a many-to-many relationship and this relation is preserved in NotificationReceiver table.

Compared to NotificationActor table, NotificationReceiver table has is_read attribute that is set if the user has read/clicked the corresponding notification item in the front-end.

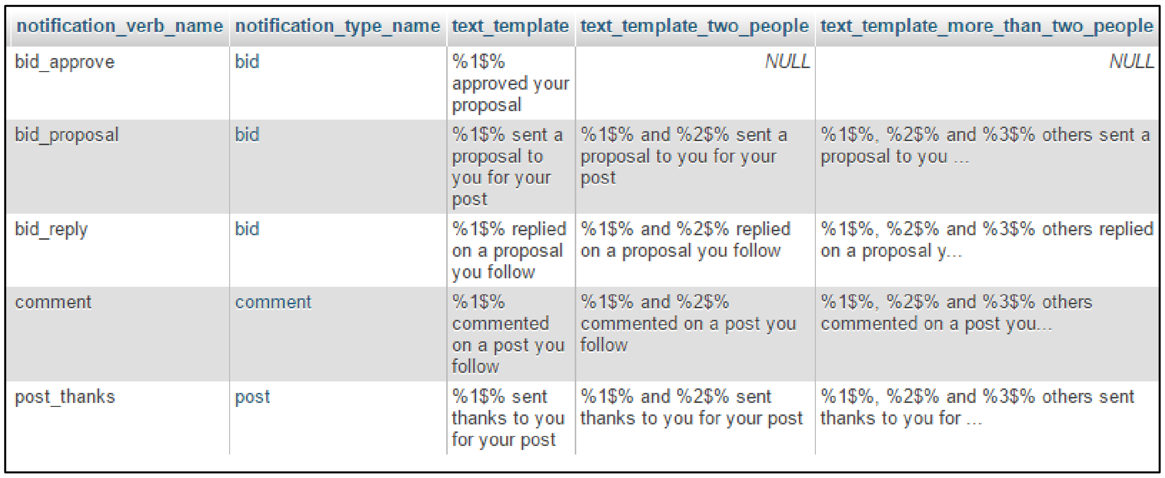


notification_type_name	url_template	created_at	updated_at
bid	post/%1$%/%2$%#bid-%3$%	1473339966	1473428365
comment	post/%1$%/%2$%#comments	1473339966	1473339966
post	post/%1$%/%2$%	1473339966	1473339966

Figure 46-Templating system in Notification database

In Notification system, Templating system is used since the value in the URL and notification text varies depending on the notification. %N$% notation shows the parameter that must be replaced in the back-end while N inside the notation shows the position of the parameter.

In Notification table, the notification_type_name column determines the URL of the notification item. This column is associated with NotificationType table that stores the URL template. Figure 11 shows there are three notification types currently in Justgivit. In NotificationType table, %1$% will be replaced by the key value column inside the Notification table, %2$% will be replaced by the extra value attribute inside the NotificationExtraValue table, and the replacement of %3$% depends on the implementation in the database.



notification_verb_name	notification_type_name	text_template	text_template_two_people	text_template_more_than_two_people
bid_approve	bid	%1$% approved your proposal	NULL	NULL
bid_proposal	bid	%1$% sent a proposal to you for your post	%1$% and %2$% sent a proposal to you for your post	%1$%, %2$% and %3$% others sent a proposal to you ...
bid_reply	bid	%1$% replied on a proposal you follow	%1$% and %2$% replied on a proposal you follow	%1$%, %2$% and %3$% others replied on a proposal y...
comment	comment	%1$% commented on a post you follow	%1$% and %2$% commented on a post you follow	%1$%, %2$% and %3$% others commented on a post you...
post_thanks	post	%1$% sent thanks to you for your post	%1$% and %2$% sent thanks to you for your post	%1$%, %2$% and %3$% others sent thanks to you for ...

Figure 47- Notification Verb table stores the text template for the notification

The notification_verb_name column determines the text of the notification. This column is associated with NotificationVerb table that stores the text template of the notification. In NotificationVerb's templating system, the notation %N$% will be replaced by the latest actors or the number of actors of the notification.

5.2. Back-end

In this section, the author will discuss the two APIs associated with the Notification Back-end. Lastly, the author will also explain how the notification database system is used using Thank You Notification as an example.

5.2.1. Get Total New Notifications

```
public function actionCountNewNotification() {
       $data = array();
       if(Yii::$app->user->isGuest) {
          $data['status'] = 0;
          return json_encode($data);
       }
       $count = $this->notification_service->countNewNotification(Yii::$app->user->getId());
       $data['count'] = $count;
       $data['status'] = 1;
       return json_encode($data);
   }
```

Figure 48- API to get the total new notifications

The figure above shows the controller of the API used to get the total new notifications. In this API, NotificationService and NotificationDao are accessed subsequently to get the result.

In NotificationDao, the associated SQL statement is executed to return the total new notifications. This SQL statement works by counting how many notifications are newer than the update_last_seen column in the User table. The notif_last_seen column in User table stores the timestamp when the last time the notification icon in the menu bar is clicked. In Appendix 8.1, the author has put the SQL statement for this API.

5.2.2. Get Latest Notification List

```
public function actionRetrieveNotification() {
      $data = array();
      if(Yii::$app->user->isGuest) {
         $data['status'] = 0;
      } else {
         $notifications = $this->notification_service->getNotification(Yii::$app->user->getId());
         $views = '';
         foreach($notifications as $notification) {
            $views .= NotificationItem::widget(
                  ['id' => 'notification-item-' . $notification->getNotificationId(),
                   'notification' => $notification]);
         }
         $data['status'] = 1;
         $data['views'] = $views;
      }
      return json_encode($data);
   }
```

Figure 49- API to get the notification list

This API procedure is similar to API in section 4.2.1 in which that *NotificationService* and *NotificationDao* is accessed subsequently. However, in this section, the service returns a list of Vos that are converted into HTML by using *NotificationItem* View template.

In *NotificationDao*, the associated SQL statement is executed to get an array of latest notifications. Firstly, This SQL works by finding the notification that has the current user as a receiver. After that, this SQL will find the last actor of the notification obtained in the previous step. Finally, these notifications are sorted in descending order by the time the last actor becomes the actor in the notification. In Appendix 8.2, the author has put the SQL statement for this API.

5.2.3. Thank You Notification

Previously, we have discussed how the two APIs related to Notification System are used. In this section, we will discuss how the Notification Database System is used in the back-end. As an example, we will use Thank You Notification used to notify post owners when someone sends Thank You to them. This type of notification has one receiver, which is the owner, and at least one actor, which is the person who sends the Thank You.

```
public function actionRequestThanks() {
     $data = array();
     $model = new ThanksForm();
     //put  data  into model
    if($model->requestThanks()) {
        $this->createNotificationPostThanks($_POST['stuff_id']);
        $data['status'] = 1;
        return json_encode($data);
     }
}
```

Figure 50- The createNotificationPostThanks method is called to create the notification

The figure above shows the controller to process Thank You request. Once the server successfully stores the Thank You information, *createNotificationPostThanks* is called to create a model that sends the notification to the post owner.

```
private function createNotificationPostThanks($post_id) {
        $notification_model = new NotificationPostThanksForm();
        $notification_model->post_id = $post_id;
        $notification_model->new_actor_id = Yii::$app->user->getId();
        $notification_model->create(NotificationPostThanksForm::POST_THANKS);
}
```

Figure 51- NotificationPostThanksForm is created and used

NotificationPostThanksForm requires two fields, which are the actor id and the post id before using it. After that, the *create* method is called to send the Thank You notification to the post owner.

NotificationPostThanksForm affects four tables at once, which are *Notification*, *NotificationActor*, *NotificationReceiver* and *NotificationExtraValue* table. Before affecting these tables, the model has to know the notification type name and the notification verb name of the Thank You Notification, which are “post” and “post_thanks”.

```
public function create() {
        $this->post = Post::find()
                ->where(['stuff_id' => $this->post_id])->one();
        $notification = $this->getNotification();
        if(!$notification) {
            $notification = new Notification();
            $notification->notification_type_name = self::NOTIFICATION_TYPE_NAME;
            $notification->notification_verb_name = self::POST_THANKS;
            $notification->url_key_value = $this->post_id;
            if(!$notification->save()) {
                return false;
            }
        }
```

Figure 52- Example of Notification Implementation

Each row in Notification table has key value, notification type name and notification verb name. In Thank You Notification, the key value is the post id. Firstly, this model checks whether the associated notification in Notification table has been created. This is done by calling *getNotification* method that returns the notification if the row exists otherwise it will return false. When the associated notification does not exist, the notification must be created.

```
$notification_extra_value = $this->getNotificationExtraValue();
if(!$notification_extra_value) {
    $notification_extra_value = new NotificationExtraValue;
    $notification_extra_value->notification_type_name = self::NOTIFICATION_TYPE_NAME;
    $notification_extra_value->url_key_value = $this->post_id;
    $notification_extra_value->extra_value = CommonLibrary::cutText($this->post['title']);
    if(!$notification_extra_value->save()) {
        return false;
    }
}
```

Figure 53- Example of Notification Extra Value Implementation

Some notifications have to use NotificationExtraValue table depending on the notification_type_name column. Furthermore, Thank You Notification requires using NotificationExtraValue table. Each NotificationExtraValue row contains a key value, extra value, and notification type. For Thank You Notification, the extra value column is the same with the post title.

```
$notification_receiver = $this->getNotificationReceiver($notification->notification_id);
if(!$notification_receiver) {
    $notification_receiver = new NotificationReceiver();
    $notification_receiver->notification_id = $notification->notification_id;
    $notification_receiver->receiver_id = $this->post['poster_id'];
    if(!$notification_receiver->save()) {
        return false;
    }
} else {
    $notification_receiver->is_read = 0;
    $notification_receiver->update();
```

Figure 54- Example of Notification Receiver implementation

Thirdly, the receiver of the Thank You notification is the owner of the post. The model has to check whether the notification receiver has existed. If it does not exist, it should be created in *NotificationReceiver* table. Otherwise, the *is_read* column in the corresponding notification receiver has to be set to zero.

```
$notification_actor = $this->getNotificationActor($notification->notification_id);
if(!$notification_actor) {
    $notification_actor = new NotificationActor();
    $notification_actor->notification_id = $notification->notification_id;
    $notification_actor->actor_id = $this->new_actor_id;
    if(!$notification_actor->save()) {
        return false;
    }
} else {
    $notification_actor->updated_at = time();
    $notification_actor->update();
}
```

Figure 55- Example of Notification Actor implementation

Lastly, the actor of the Thank You notification is the current user. The model has to check whether the notification actor has existed. If it does not exist, it should be created in NotificationActor table. Otherwise, the *updated_at* column in the corresponding NotificationActor row should be updated with the current time.

5.3. Front-end

In Notification Front-end, two main components are *NotificationList* and *NotificationItem* components. The relationship between *NotificationList* and *NotificationItem* is a parent-child relationship because *NotificationList* can contain many *NotificationItem* components.

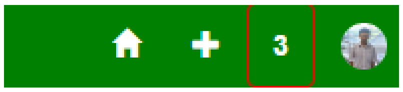
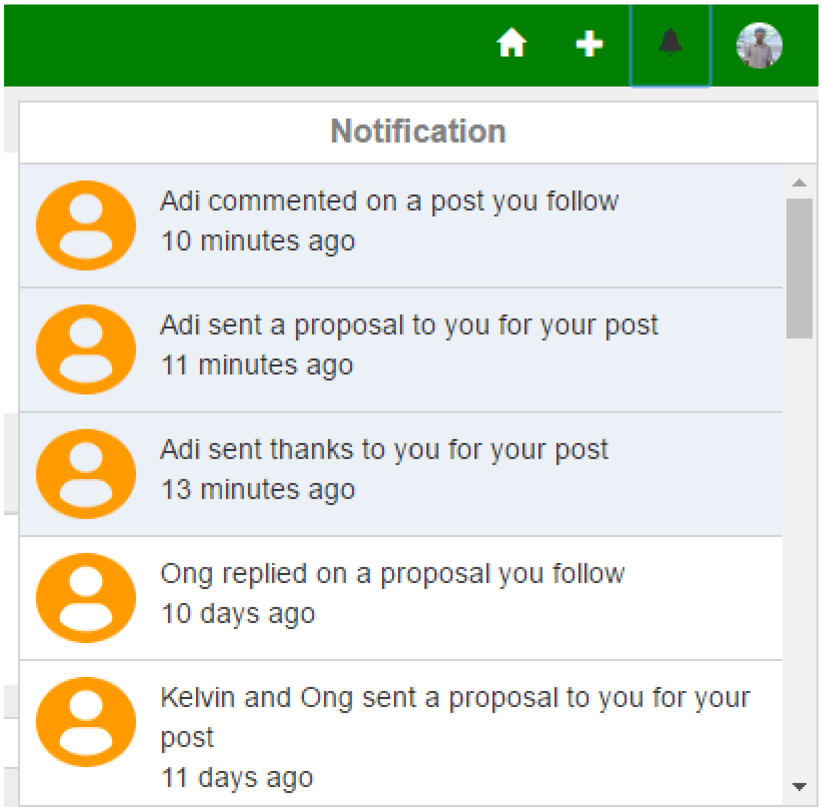


Figure 56- shows there are 3 new notifications

After *NotificationList* component is created, the AJAX request is sent immediately to an

API describes in section 4.2.1 to get the total new notifications. Furthermore, the result receives from the API is shown in the menu bar (See Figure 41).



Notification

Adi commented on a post you follow
10 minutes ago

Adi sent a proposal to you for your post
11 minutes ago

Adi sent thanks to you for your post
13 minutes ago

Ong replied on a proposal you follow
10 days ago

Kelvin and Ong sent a proposal to you for your post
11 days ago

Figure 57- UI for Notification List component that contains five NotificationItem components.

When the user clicks the notification icon in the menu bar, the client will send another AJAX request to an API in section 4.2.2 to retrieve the Latest Notification Items. Furthermore, the *notif_last_seen* column in User table also has to be updated.

There are two types of notification item which read notification item and unread notification item. The read notification items are shown with the white background color and the unread notification items are shown with the light blue background color. If the unread notification is clicked, the AJAX request is sent immediately to an API to change the unread notification to read the notification.

6. Result

In this chapter, the author will show the development progress that has been made from January to October 2016. The progress includes the user interface, the performance, layout responsiveness, and data analytics.

6.1. Well-Performing Application

Google page speed test is a tool provided by google to analyze a site whether it has followed web best practices. Therefore, this tool can be used to find a way to tweak the performance of the application. Furthermore, this tool gives scores that show how the web is performing. According to the documentation, a score of 85 or higher means the web is performing well.

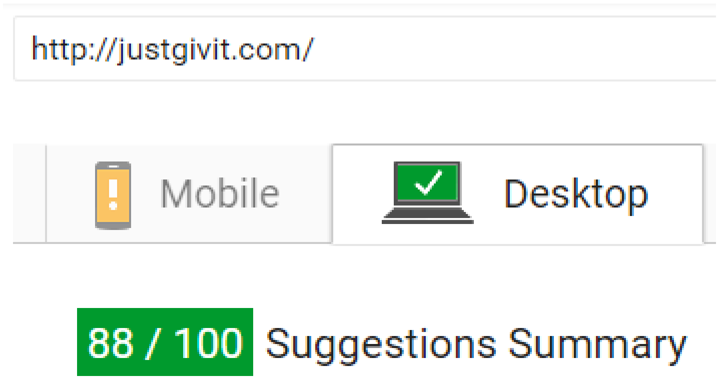


Figure 58- Google Page Speed Score for Justgivit Desktop View

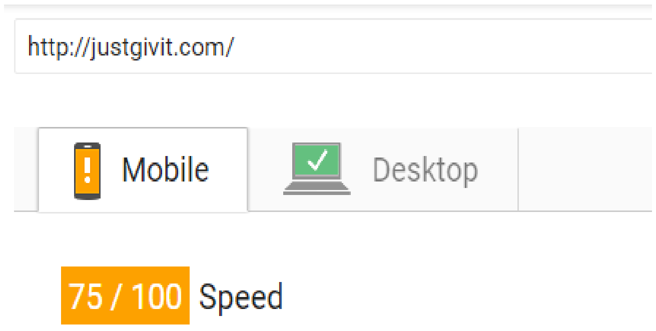


Figure 59- Google Page Speed Score for Justgivit Mobile View

The score given by Google Page speed test to Justgivit.com is 88/100 for the desktop and 75/100 for the mobile. This shows that Justgivit.com is performing quite well.

Another benchmark that is useful to show how fast the performance of a website is Dom Content Loaded time and Load time. The Dom content loaded time shows the duration of the request sent by the front-end until the Dom Content Loaded event is fired. The Dom content loaded event is fired when the DOM tree has been fully constructed. On the other hand, Load time is the length of duration from the request sent by the browser until the load event is fired. The load event is fired when all the images and sub-frames have been retrieved.

Performance benchmark	Dom Content Load time	Load time
1st trial	2.49s	5.74s
2nd trial	1.63s	5.17s
3rd trial	2.43s	5.89s
4th trial	2.38s	5.70s
Average	2.2325s	5.625s

Table 5- The Dom Content Loaded time and the Load time of Home Page

The testing of Dom Content Loaded time and Load time is conducted in Hall 11, Nanyang

Technological University using Chrome browser. The result shows that Dom Content Loaded time is approximately 2.23 second, which is quite good remembering the limited resource that this project is using. However, the load time is slower than the Dom content loaded time which around 5.625 seconds. This happens because the browser has to download many images at one time when User opens the home page. The solution to this is to implement caching for the image in the browser. Unfortunately, due to limited time and resources, this will not be implemented until this report is finished.

6.2. Data Analytics and User Feedback

Justgivit.com is currently in Public Beta version, which means everyone can try out the application. Since the application has not been marketed yet, there were only approximately 100 users currently registered in the system by early October 2016. Most of them came either from Singapore or from Indonesia. Most of these registered users helped to test the site and gave constructive feedbacks to the author.

By early October 2016, there were 66 items in sites to be given and requested which books happened to be the most popular item, which was 23 items, and gadget came next with six items. Currently, there were only three items successfully transacted since there were only a few bidders in the site.

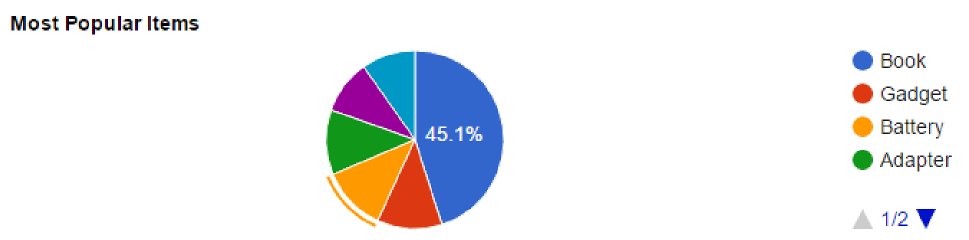


Figure 60- Most Popular Items

Most Users say that the site should have a better communicate like SMS or Email. Although email communication is supported, some of these emails from the registered users are not. Furthermore, some users use a Facebook email account that has been obsolete; therefore, the progress of the post cannot be notified to the email.

In the next version, Author will enforce the user to validate a valid email and support SMS communication. The author will also find an admin to help in managing the site while the author focuses on the business and development site.

7. Conclusions & Future Works

The aim of this project is to develop a production ready web and android application, which support high usability and well-performing application. This aspiration has made this project exciting and challenging at the same time. Throughout this FYP project, the project has included many usable features such as Bidding System, Comment System, Notification Systems, and Authentication System. Furthermore, the project also supports responsive layout and receive a decent performance score from Google Page Speed Test.

Although there are many implementations that have been done in these 10 months, there are still many things we can do to improve the applications such as:

- Provide a way to share content in our systems to social networks to ease sharing content with the social networks' users.
- Implement Native Application to enhance user experience compared to using Web view application that we are using currently.
- Improve UI and Layout design should also be considered in the future to enhance user experience.

8. Bibliography

[1] Yii2 Framework. [Online]. Available: http://www.yiiframework.com/

[2] Web view. [Online]. Available: https://developer.android.com/guide/webapps/webview.html

[3] Amazon Web Services. Available: https://aws.amazon.com/

[4] WAMP Server. Available: http://www.wampserver.com/en/

[5]. PHP Documentation. Available: http://php.net/docs.php

[6]. Totally Free Stuff. Available: http://www.totallyfreestuff.com/

[7]. MySQL Documentation. Available: https://dev.mysql.com/doc/

[8]. Gulp JS. Available: http://gulpjs.com/

[9]. Amazon Linux AMI. Available: https://aws.amazon.com/amazon-linux-ami/

9. References

[1] UNEP Year Book 2009. Retrieved September 16, 2016, from http://www.unep.org/yearbook/2009

[2] Hirth, M., Hoßfeld, T., & Tran-Gia, P. (2011). Anatomy of a Crowdsourcing Platform - Using the Example of Microworkers.com. *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. doi:10.1109/imis.2011.89

[3] Horvát, E., Uparna, J., & Uzzi, B. (2015). Network vs Market Relations. *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015 - ASONAM '15*. doi:10.1145/2808797.2808904

[4] Hui, J. S., Gerber, E. M., & Gergle, D. (2014). Understanding and leveraging social networks for crowdfunding. *Proceedings of the 2014 Conference on Designing Interactive Systems - DIS '14*. doi:10.1145/2598510.2598539

[5] Kim, Y. (2015). Libero. *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems - CHI EA '15*. doi:10.1145/2702613.2726964

[6] Zhao, W., Du, Z., Yang, Y., Zhang, C., & Liao, W. (2014). LifeDelivery. *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing Adjunct Publication - UbiComp '14 Adjunct*. doi:10.1145/2638728.2638761

[7] Cvijikj, I. P., Kadar, C., Ivan, B., & Te, Y. (2015). Towards a crowdsourcing approach for crime prevention. *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers - UbiComp '15*. doi:10.1145/2800835.2800971

[8] Kickstarter. Retrieved October 16, 2016, from https://www.kickstarter.com/help/stats

[9] Indiegogo. Retrieved October 16, 2016, from https://www.indiegogo.com/about/our-story

10. Appendix

10.1. SQL Statement for counting New Notification

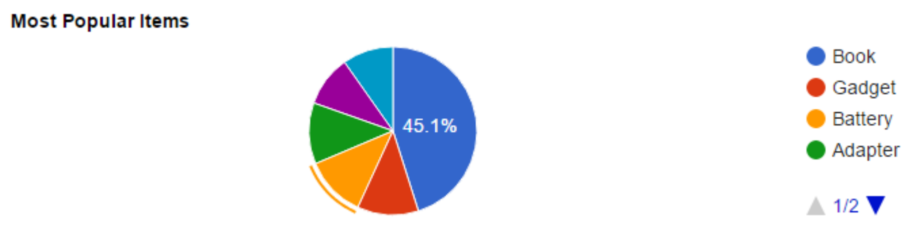
```
SELECT count(*)
from (SELECT notification.notification_id
        from notification, notification_receiver
        where notification_receiver.notification_id = notification.notification_id
                and notification_receiver.receiver_id = :user_id) notification_entity
 left join(
         SELECT na.updated_at,na.actor_id, n.notification_id
         from notification n, notification_actor na
         where n.notification_id = na.notification_id and
                 na.updated_at = (SELECT max(na1.updated_at)
                                         from notification_actor na1
                                         where n.notification_id = na1.notification_id
                                                 and na1.actor_id <> :user_id)
                                         ) last_actor
on notification_entity.notification_id = last_actor.notification_id
where last_actor.actor_id is not null and updated_at >
          (SELECT notif_last_seen from user where id = :user_id)
```

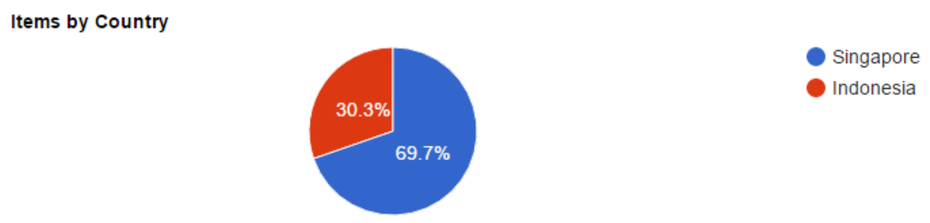
10.2. SQL Statement for listing Notification

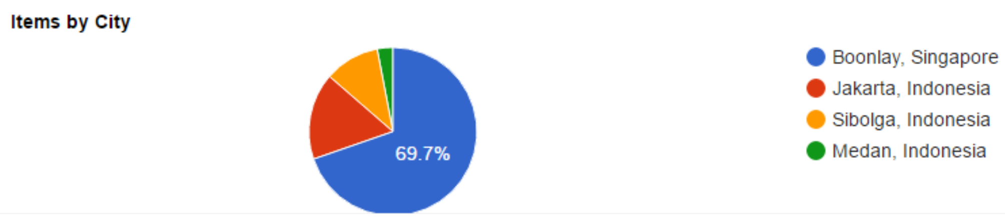
```
SELECT notification_entity.*,notification_actors.actors,
            last_actor.profile_pic, last_actor.updated_at, last_actor.actor_id,
            notification_extra_value.extra_value
from (SELECT notification_type.url_template,
                     notification_verb.text_template,
                     notification_verb.text_template_two_people,
                     notification_verb.text_template_more_than_two_people,
                     notification_receiver.is_read,
                     notification.notification_id, notification.notification_type_name,
                     notification.notification_verb_name, notification.url_key_value
          from notification_type, notification_verb, notification, notification_receiver
          where notification_type.notification_type_name =   notification_verb.notification_type_name
                   and notification_verb.notification_verb_name = notification.notification_verb_name
                   and notification_verb.notification_type_name = notification.notification_type_name
                   and notification_receiver.notification_id = notification.notification_id
                   and notification_receiver.receiver_id = :user_id) notification_entity
         left join
                  (SELECT n.notification_id,
                              na.updated_at,
                              group_concat(actor.first_name SEPARATOR '%,%') as actors
                   FROM notification n, notification_actor na, user actor
                   WHERE n.notification_id = na.notification_id
                              and actor.id = na.actor_id and actor.id <> :user_id
                   group by na.notification_id
                   order by na.updated_at desc
                   ) notification_actors
on notification_actors.notification_id = notification_entity.notification_id
left join(
             SELECT u.profile_pic, n.notification_id, na.updated_at,na.actor_id
             from notification n, notification_actor na, user u
             where n.notification_id = na.notification_id and
                      na.actor_id = u.id and
                      na.updated_at = (SELECT max(na1.updated_at)
                                               from notification_actor na1
                                               where n.notification_id = na1.notification_id
                                                        and na1.actor_id <> :user_id )
            ) last_actor
```

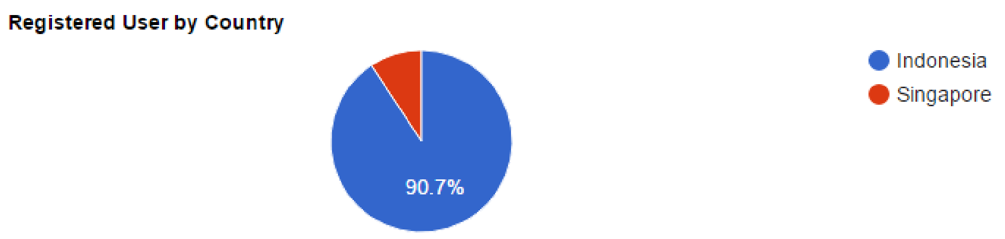
```
on notification_entity.notification_id = last_actor.notification_id
left join notification_extra_value
on notification_extra_value.notification_type_name = notification_entity.notification_type_name
    and notification_extra_value.url_key_value = notification_entity.url_key_value
where last_actor.actor_id is not null
order by last_actor.updated_at desc
```

10.3. Data Analytics


Most Popular Items
Book
Gadget
Battery
Adapter
1/2


Items by Country
Singapore
Indonesia


Items by City
Boonlay, Singapore
Jakarta, Indonesia
Sibolga, Indonesia
Medan, Indonesia


Registered User by Country
Indonesia
Singapore