Ian Bryan Banda

Banda.ian45@gmail.com

Abstract

A front-end web based client that will be used will revolutionize Taxpayer management and will be used by MRA Tax officers

Taxpayersys

V 1.0

Contents

[Introduction 2](#_Toc88844397)

[Languages 2](#_Toc88844398)

[Why Web-based? 2](#_Toc88844399)

[Paradigm 2](#_Toc88844400)

[Feature List 2](#_Toc88844401)

[Functionality Description 2](#_Toc88844402)

[Reception Page 2](#_Toc88844403)

[Tax Officer/User Login 3](#_Toc88844404)

[Welcome after login 4](#_Toc88844405)

[Pressing the New Taxpayer button 5](#_Toc88844406)

[Taxpayer editing 7](#_Toc88844407)

[Taxpayer Deletion 9](#_Toc88844408)

[Logging out 9](#_Toc88844409)

[MVC: Model-View-Controller 10](#_Toc88844410)

[Model 10](#_Toc88844411)

[View 10](#_Toc88844412)

[Controller 10](#_Toc88844413)

[The Front Controller pattern 10](#_Toc88844414)

[Registry 10](#_Toc88844415)

[Registry Factory 11](#_Toc88844416)

[Registry + MVC 11](#_Toc88844417)

[Folder/File structure 11](#_Toc88844418)

[CLASS/OBJECT DEFINITION 12](#_Toc88844419)

[The registry object 12](#_Toc88844420)

[Registry objects 12](#_Toc88844421)

[.htaccess 18](#_Toc88844422)

[Models 19](#_Toc88844423)

[Taxpayer Model – Taxpayer Class (Models/Taxpayer.php) 19](#_Toc88844424)

[User Model – User Class (Models/User.php) 19](#_Toc88844425)

[Views 19](#_Toc88844426)

[Templates 19](#_Toc88844427)

[Css 19](#_Toc88844428)

[Controllers 19](#_Toc88844429)

[Authcontroller Class (Auth/controller.php) 19](#_Toc88844430)

[Taxpayerscontroller Class (Taxpayers/controller.php) 20](#_Toc88844431)

# Introduction

TaxpayerSys 1.0 is a web based frontend client that will be used by Malawi Revenue Authority (MRA) Tax Officer to register new Taxpayers into the MRA database. The system works by consuming the web services provided by the RESTful API provided by the official MRA web system.

# Languages

The system being web-based, has been developed using a collection of programming, markup, and scripting languages including. The presentation layer has been written using HTML styled with CSS. The behavioral layer has been developed using pure javascript and JQuery, whilst the content has been written in Php.

# Why Web-based?

The system has been designed to be web-based due to the portability and platform independence that was needed since not all Tax Officers will have their PCs throughout. When hosted, Tax Officers can register new Taxpayers even using their phones since it is both PC and Mobile friendly.

# Paradigm

The system has been created using the object-oriented paradigm which allows for easy code management, re-use, and debugging.

# Feature List

This system has been created a frontend-type of a system, which directly implies it won’t be able to run independent of the web service it consumes on.

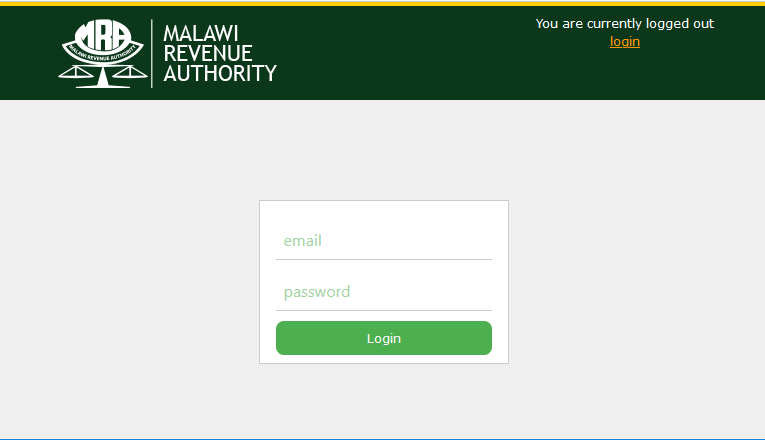
Despite this system not having in own database due to it running on the web service provided via the MRA RESTful API, it will still provide its intended user the following features:

1. Logging into the system
2. Logging out if the system
3. Creation/Registration of New Taxpayers.
4. Editing of existing Taxpayer’s details.
5. Deletion of existing Taxpayers

# Functionality Description

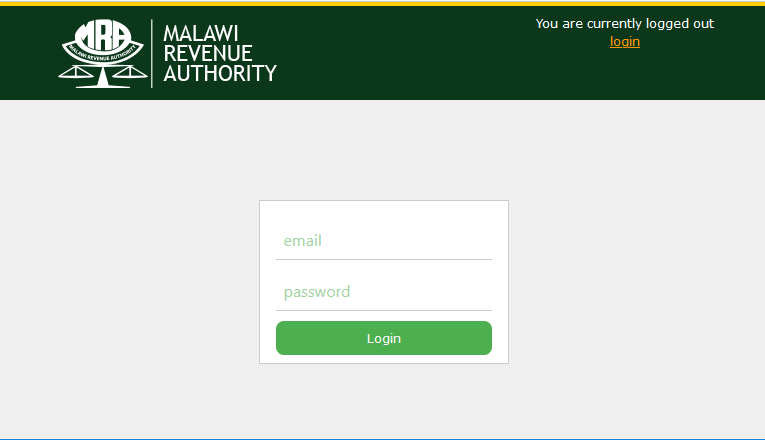
## Reception Page

The index.php will act as the main reception page of this web based system. Upon first visitation, one should expect to encounter the graphical interface below

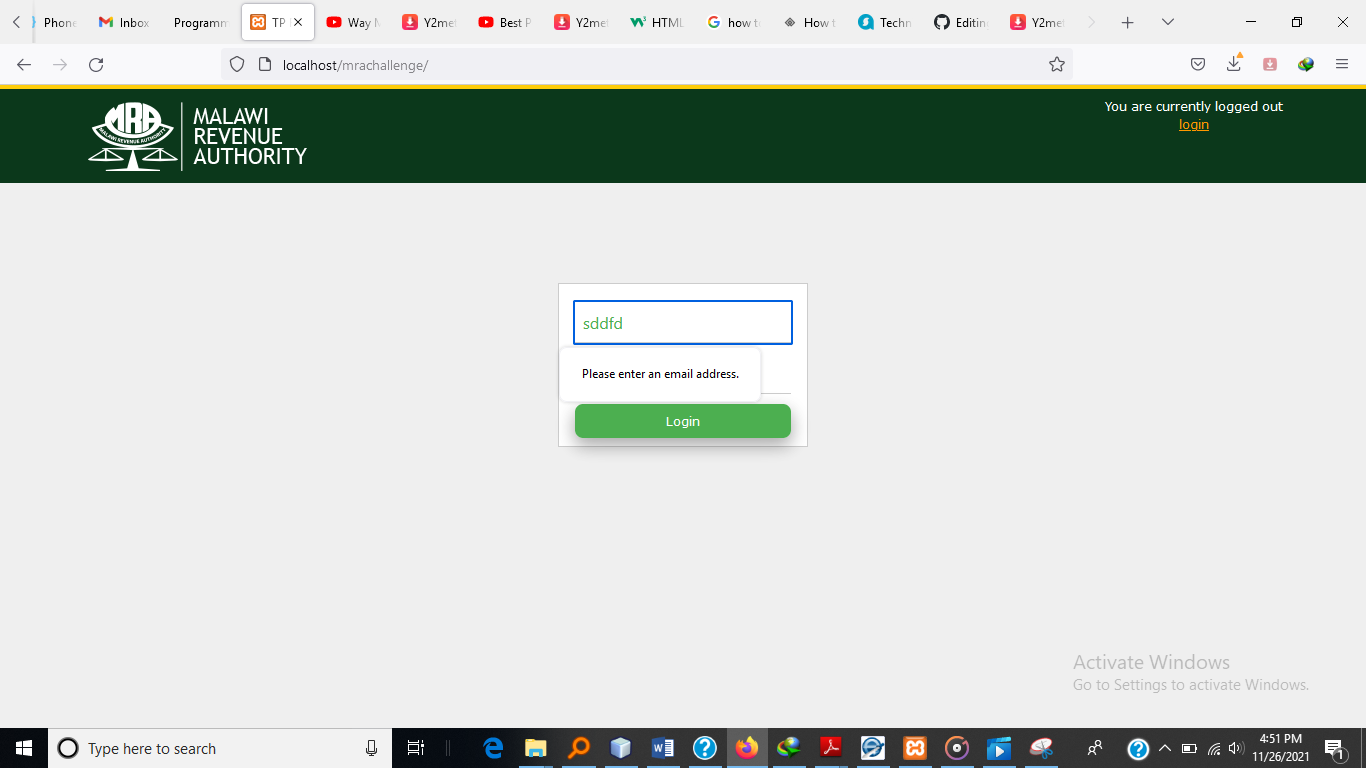


## Tax Officer/User Login

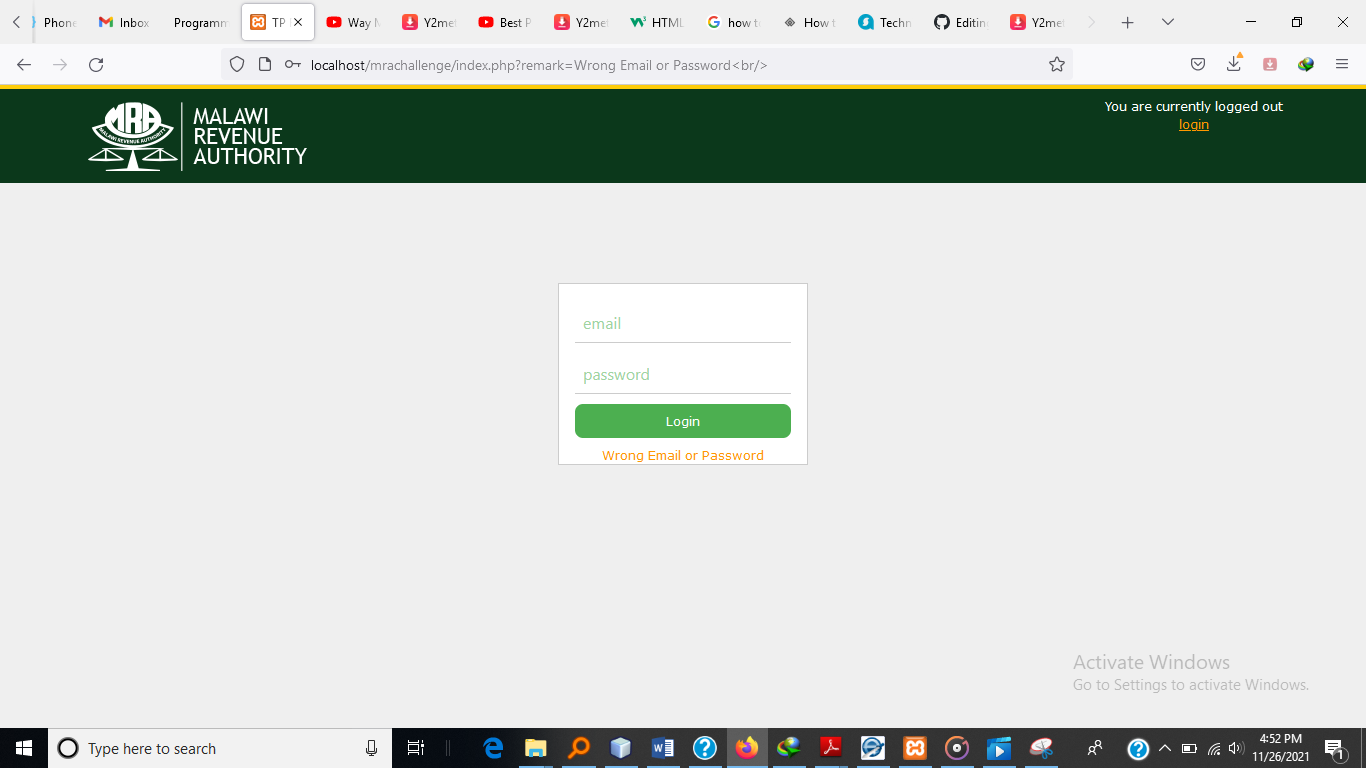
Users: Tax officers, will be able to log into the system by inputting an email and a password. The email and password are supposed to be activated by an admin on the web service. The screen below shows



When the inputs are submitted, the system will attempt to validate and then authenticate user details if the validation conditions are met. Below are sample screens of a rejected submission



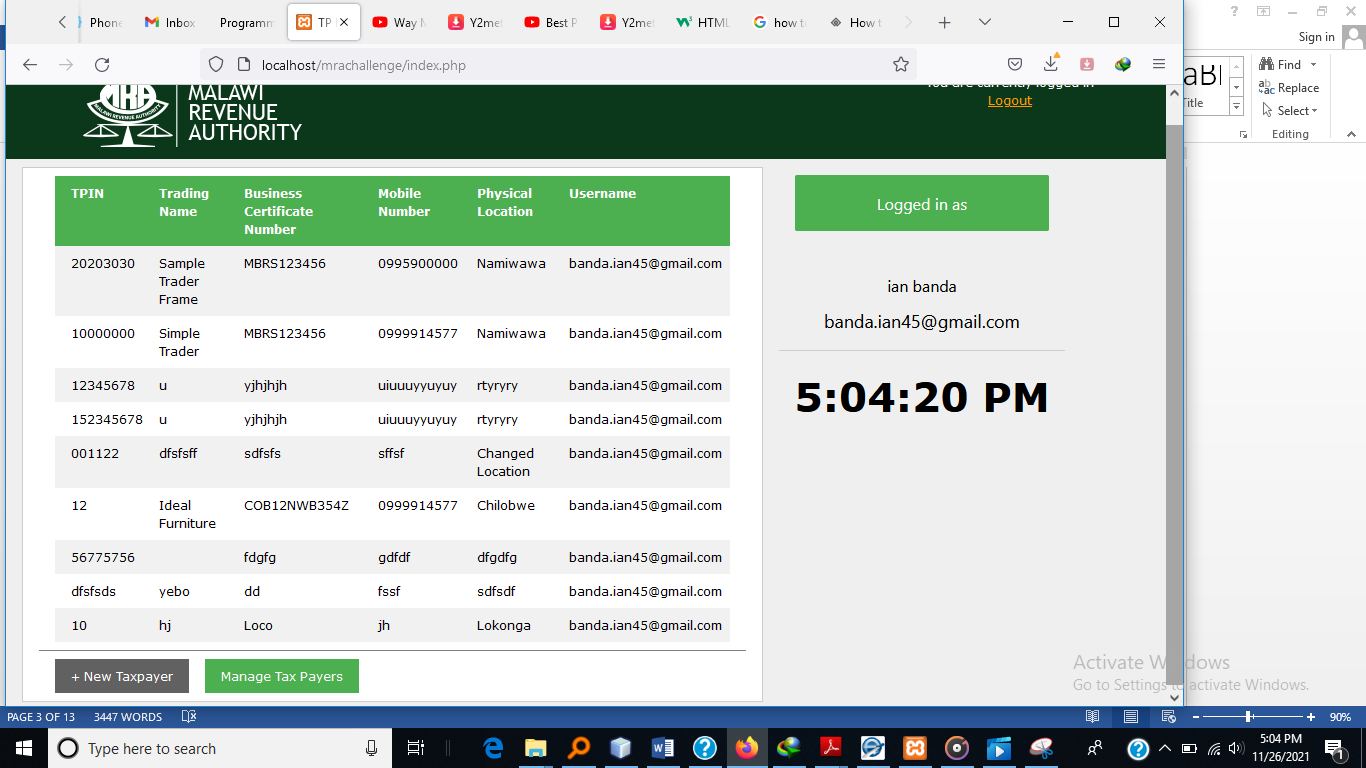
Error generated by html 5 validation



Error generated using Php validation upon submission

## Welcome after login

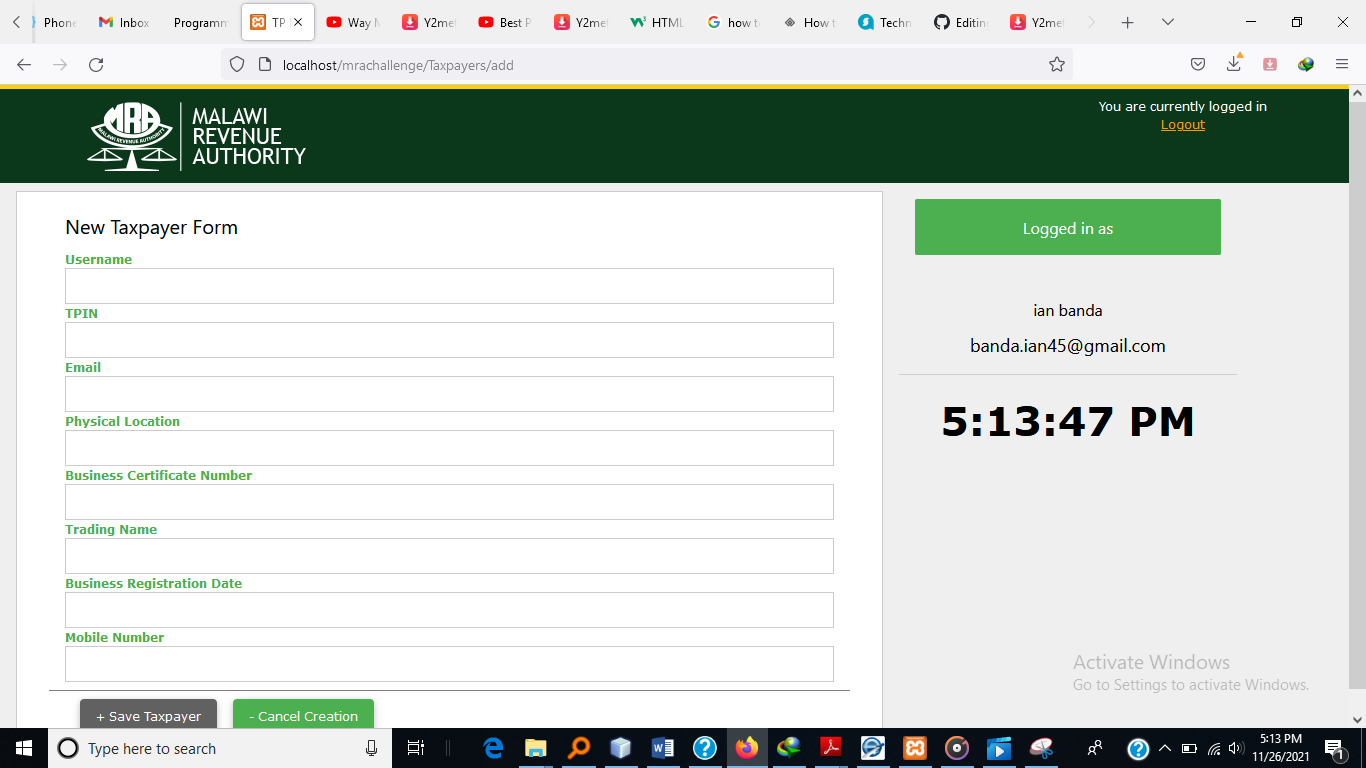
The following screen will be displayed after the officer has successfully logged in



The initial view after logging in will display the previous Taxpayers that the logged in Tax officer has previously created. This view will also display details about the user, details like the officers full name and email username will be displayed. A digital clock will also be displayed below the user details. Two buttons: 1.New taxpayer and 2. Manage Taxpayers are shown right below the taxpayer list.

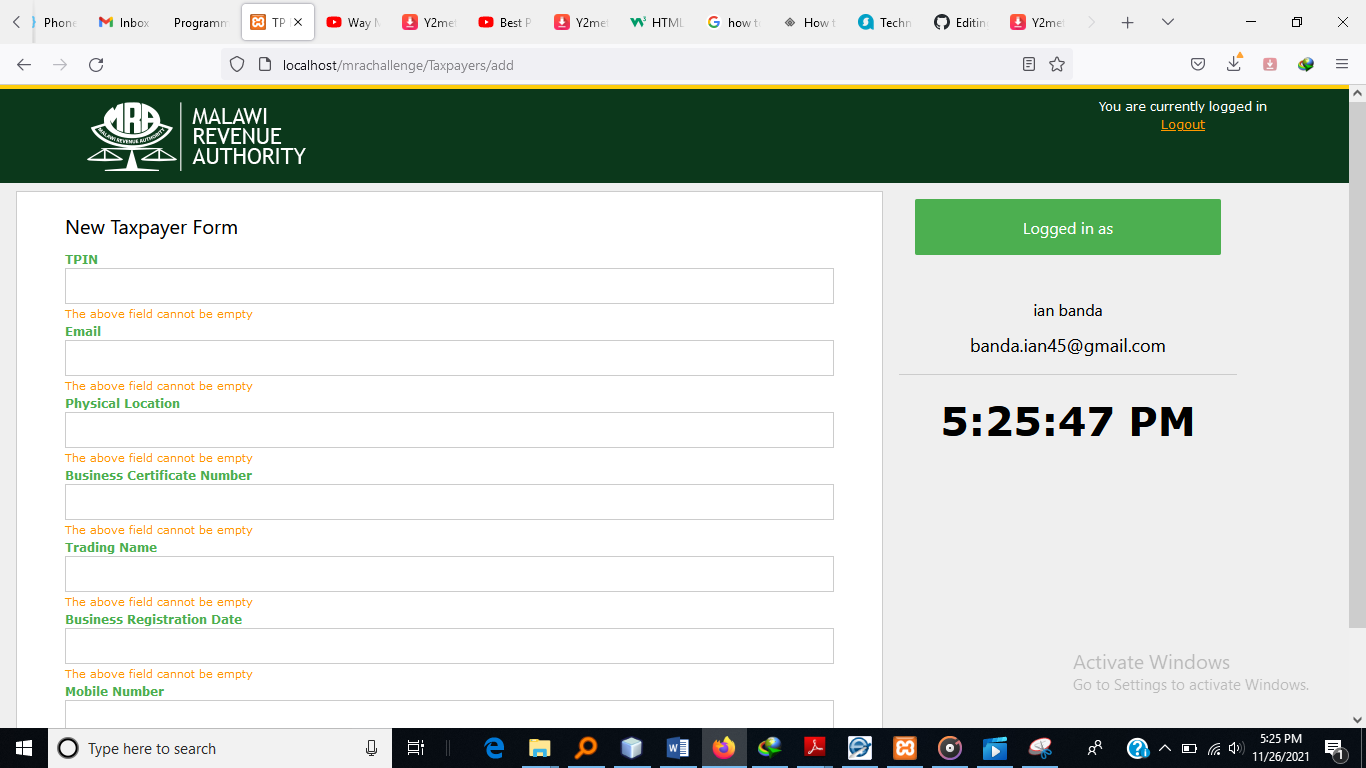
## Pressing the New Taxpayer button

The view below will show upon pressing the **New Taxpayer** button.

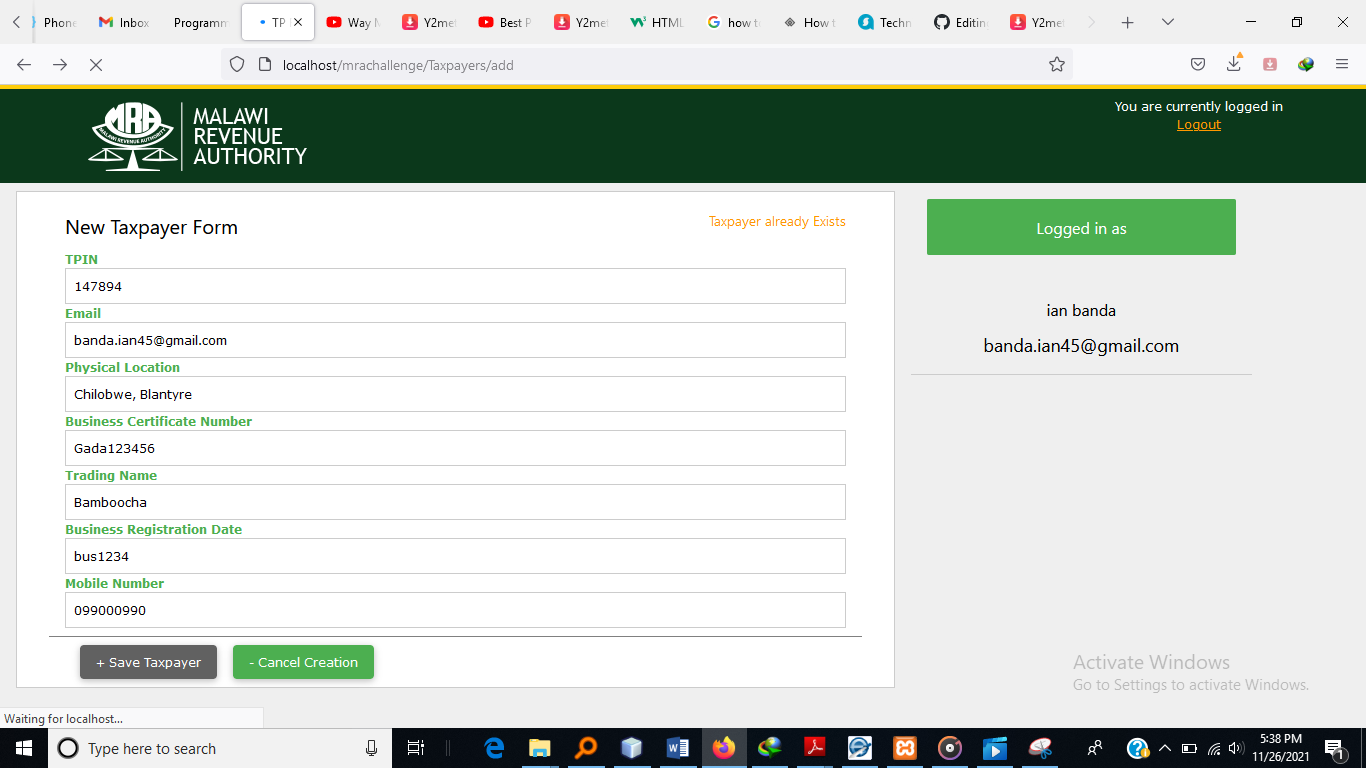


The officer will be required to input all the fields and press **+ Save Taxpayer** to submit the details of a new Taxpayer being registered.

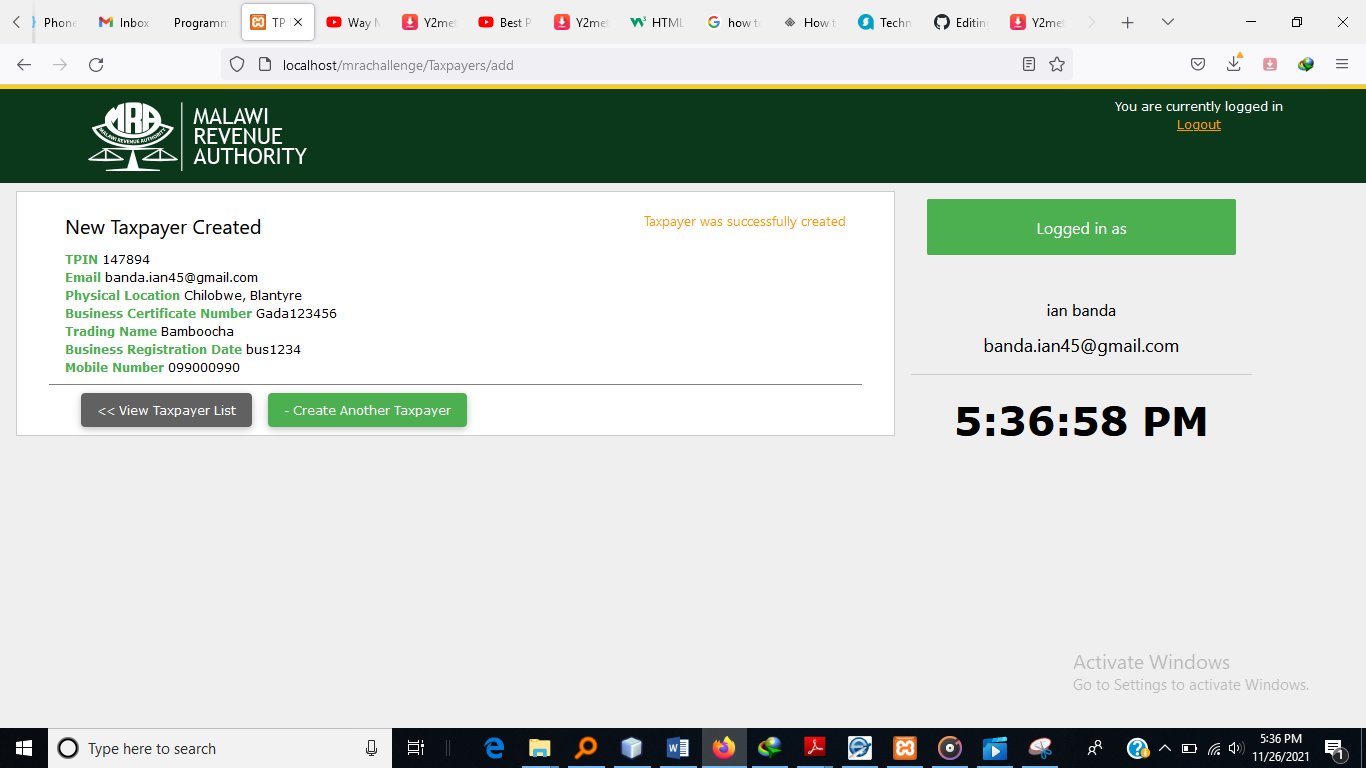
Upon submission, the form will be validated and the screen below is a sample of a screen of a submission rejection because the user omitted a certain field



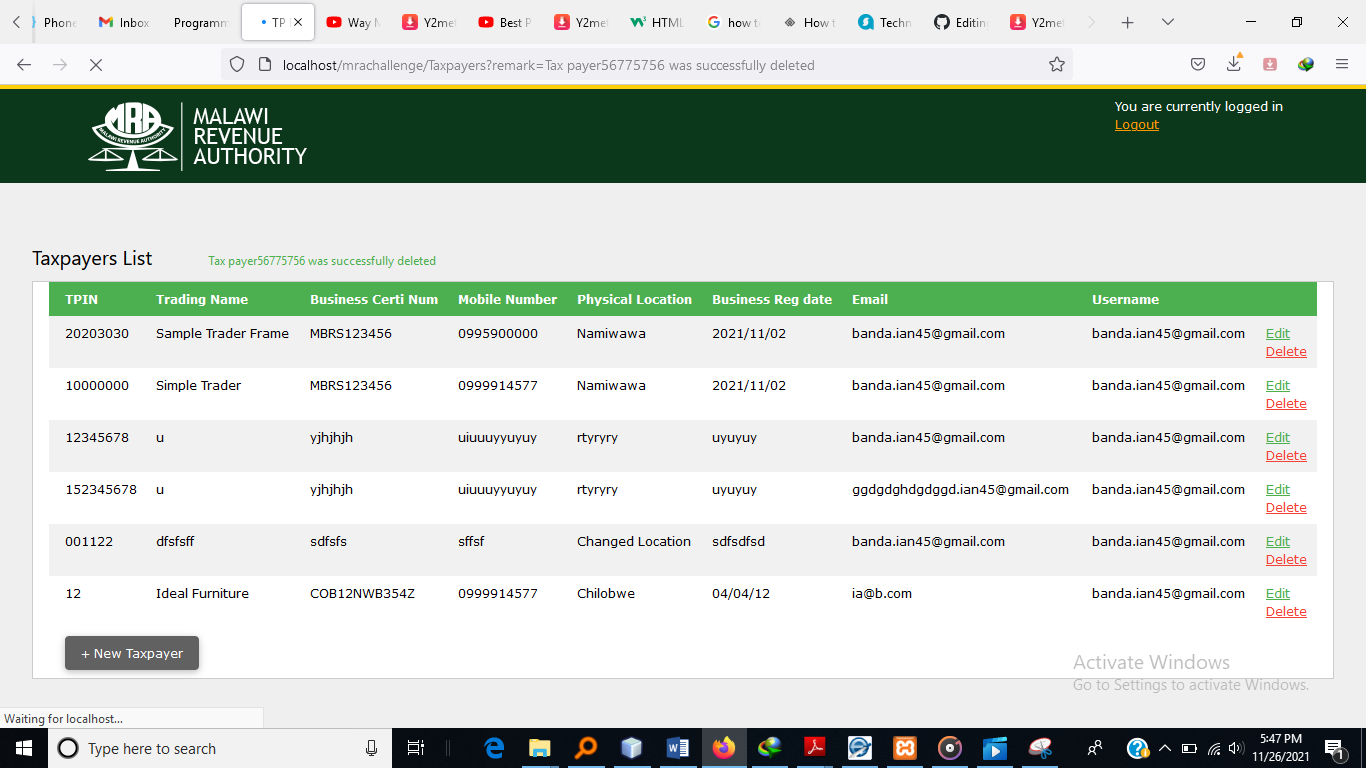
The screen below will be displayed when a Taxpayer that is already registered is attempted to be created



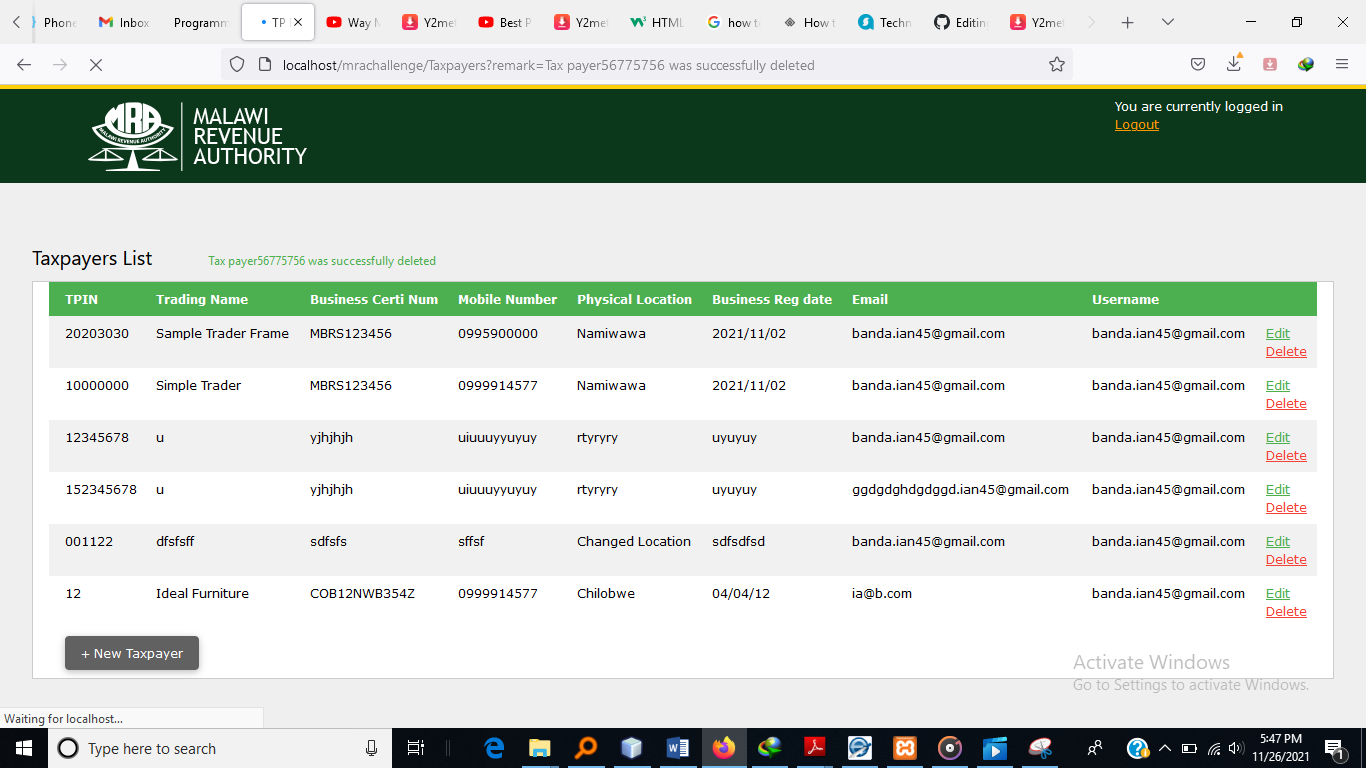
Upon a successful submission, the Taxpayer will be registered and the following screen will be displayed



The taxpayers list can be accessed by pressing the **<< View taxpayer list** button or from the initial login screen by pressing the **Manage Taxpayer** button. The screen for the main tax payers list will be shown as below:

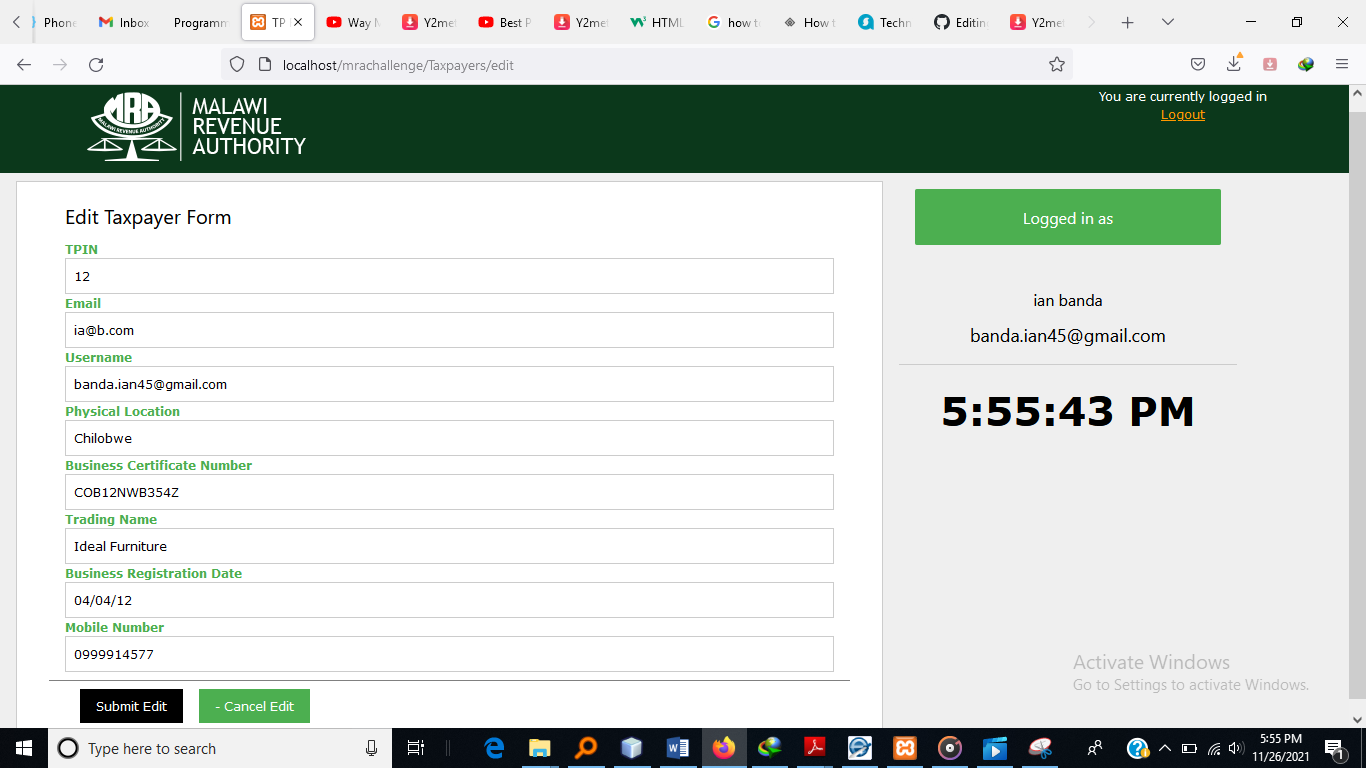


The above screen will give the officer with two hyperlink through which the Taxpayer can be edited or deleted through



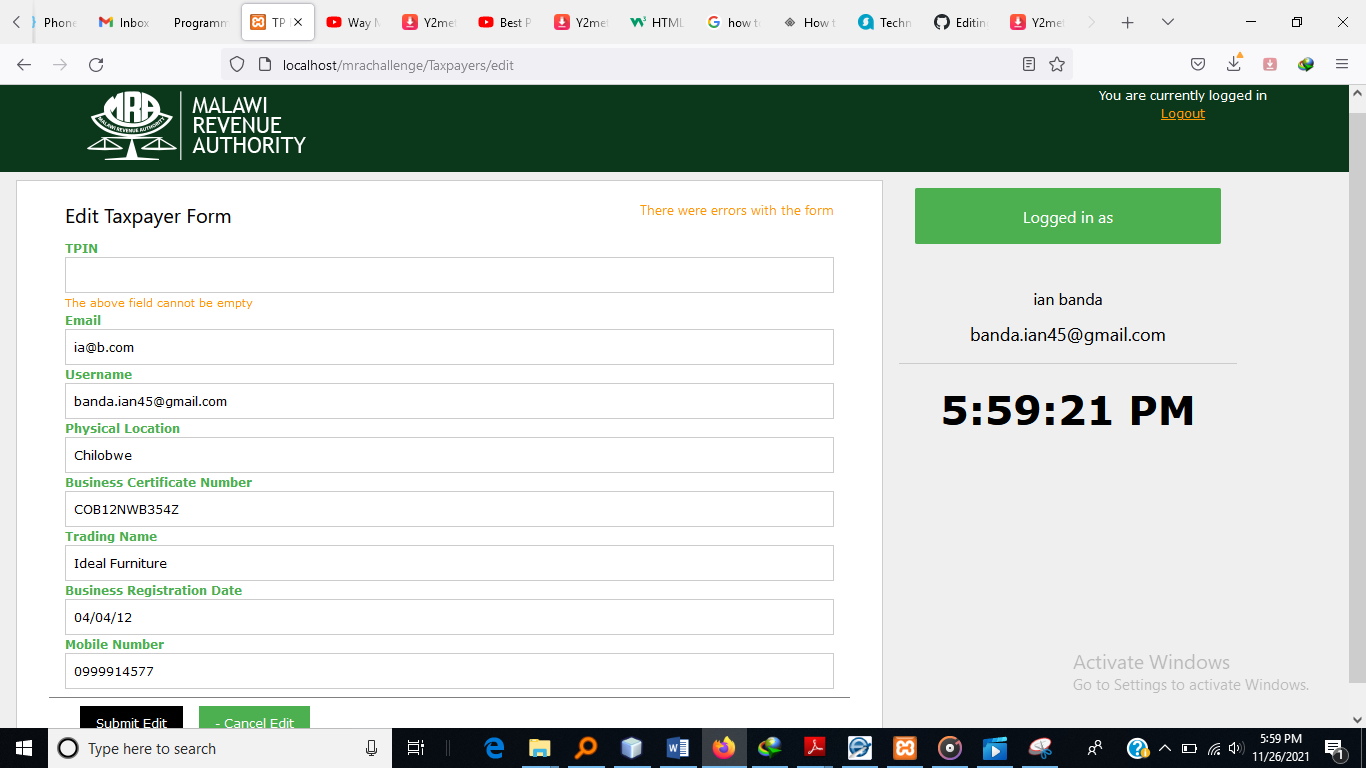
## Taxpayer editing

Upon pressing the **edit** hyperlink, a form which can be edited is displayed with the values of the Taxpayer filled into the input fields as below

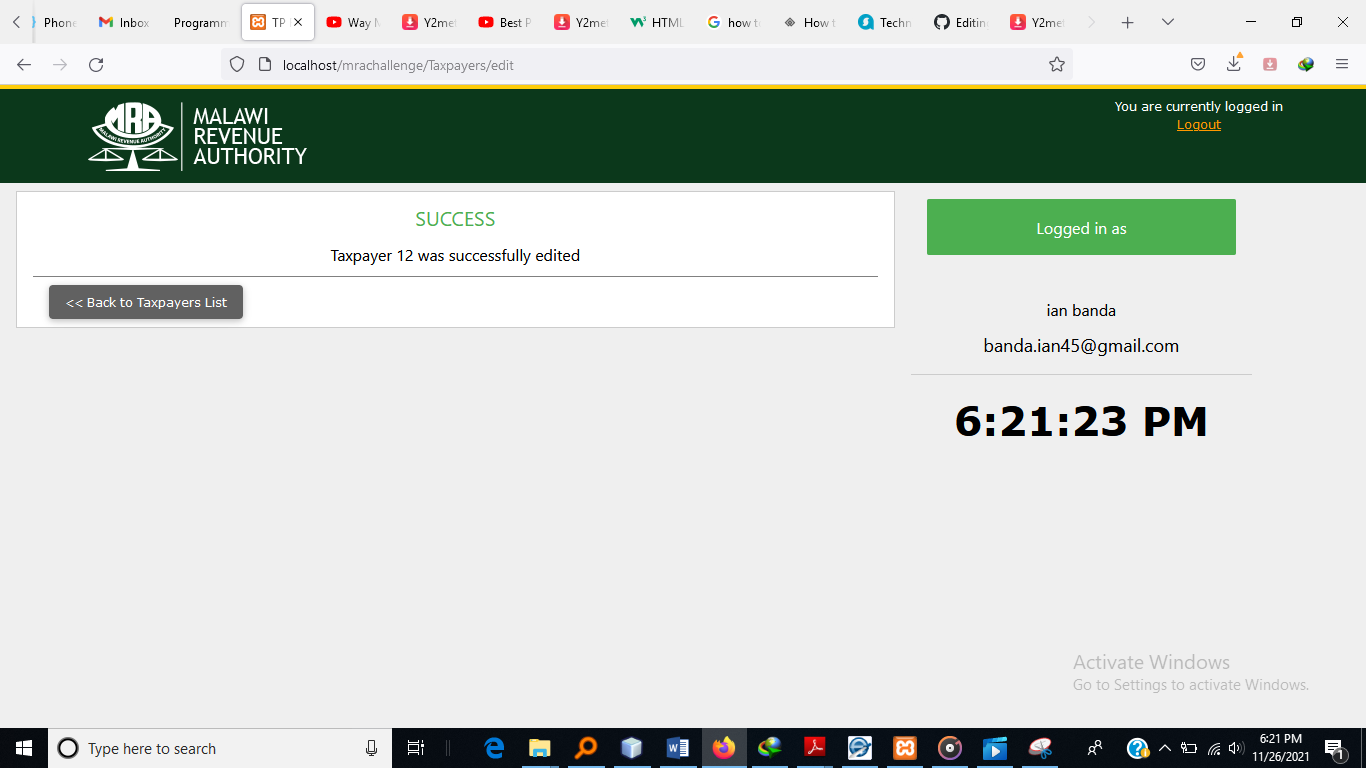


When the taxpayer edit form has been submitted, it will be validated and there will be two possible outcomes: acceptance or rejection.

The following is a sample of a taxpayer form that has been rejected

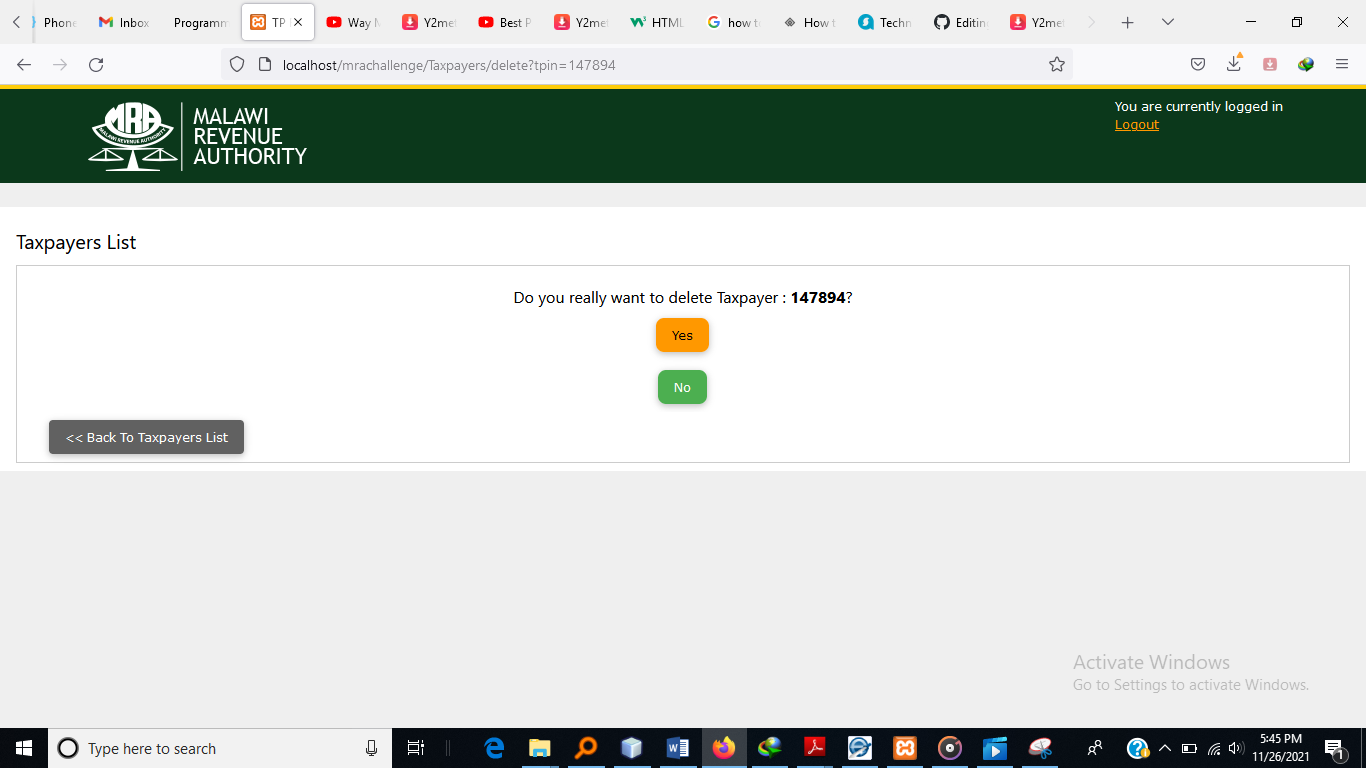


The screen below will be displayed once the form has been accepted



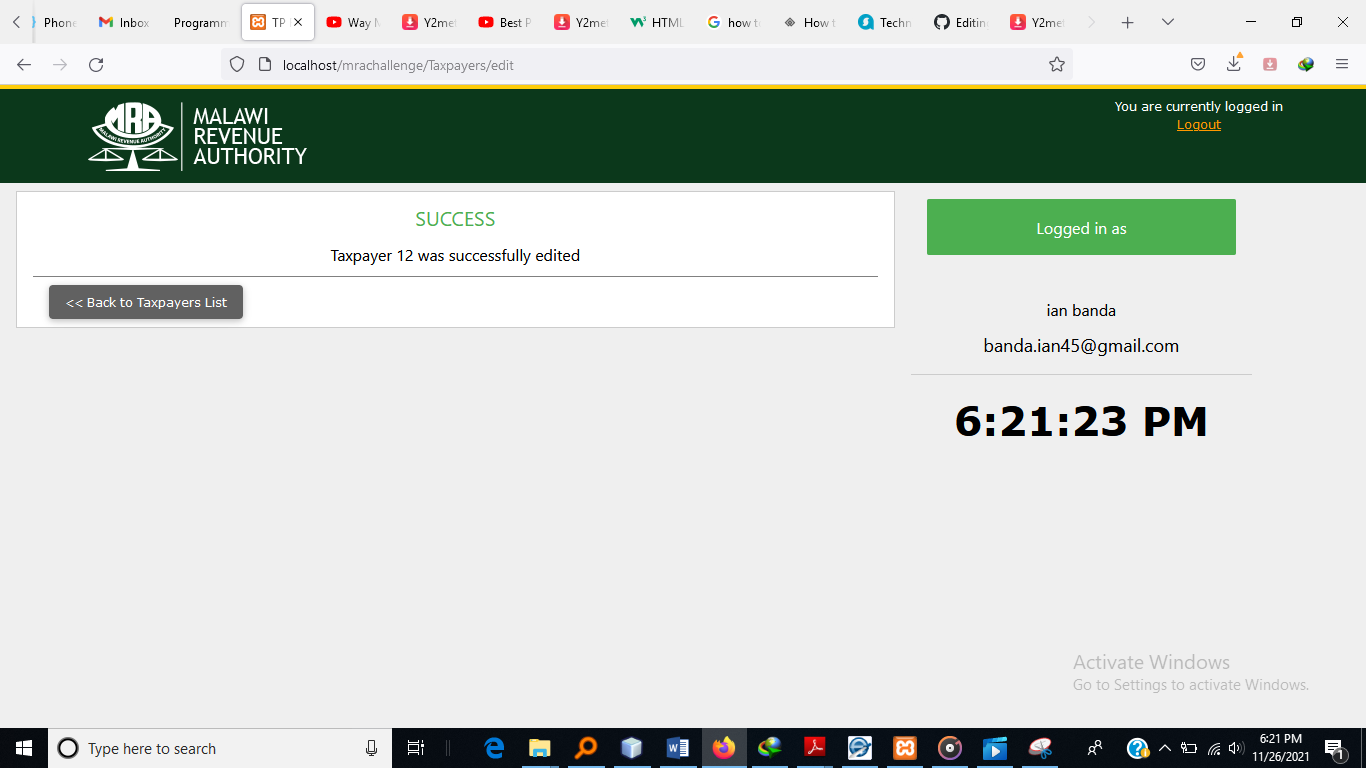
## Taxpayer Deletion

The screen below will be displayed when the user presses the **Delete Taxpayer** button. The user will be prompted to either continue with taxpayer deletion or not. When the **No** button is selected the manage taxpayers list will then be displayed without deleting the user, whilst choosing will both delete the Taxpayer and display the Manage Taxpayers list



## Logging out

As long as the user is logged in, the section shown below will be displayed at the top right corner of every page. Pressing the **logout** hyperlink will initiate a user signout process which when successful will end up displaying the initial login page.



# MVC: Model-View-Controller

The system has been developed using the Model-View-Controller pattern which is an architectural design pattern designed to separate the user interface from the business logic of an application. The user interface (view) uses the controller to interact with the logic and data of the application (model).

## Model

Within our system, the models will be PHP classes that temporarily store, manage, and process data and business logic. Access to the data provided by the web service will be handled by an API/Web service access layer (Auth.class.php in the registry folder), which the models will make use of. The models will link closely with the data provided by the Web service, representing the data in a more suitable way, which is easier to access and manipulate than accessing the database directly.

## View

In our system, the view will be made up of a combination of template files (which will contain HTML and placeholders for dynamic data), images, CSS files, and JavaScript. The templates will be merged and outputted to the user's browser on the fly by the controller.

## Controller

The controllers will be a series of PHP classes, which process the user's request, and interact with the model, as well as generate views. Technically, some of our JavaScript (particularly where JQuery is used) also makes up a part of the controller, as it interacts between the view, and the model; these instances are extensions of the controller.

# The Front Controller pattern

The Front Controller pattern is a single file, through which all requests are routed (in our case, using Apache's mod\_rewrite). In our case, this is the index.php file. This file will process the user's request, and pass the request to the appropriate controller.

By using a single front controller, our core includes files, core settings, and other common requirements that can all be performed, so that we know regardless of the user request these will have taken place.

# Registry

Within most web application frameworks, there are numerous core objects, or objects containing core functionality that every aspect of the application will need to have access to. The registry pattern provides us with a means to store all of these core objects within one central object, and access them from within.

These functions will be abstracted into their own object that will be stored centrally within our registry. The rest of our social networks code can access all of these objects and features directly from our registry. The architecture of the registry is illustrated in the following screenshot:

## Registry Factory

Another design pattern that we will make use of is the Factory pattern. To save the need of creating all of the objects that our registry is going to manage, and passing them to the registry, we will simply tell the registry the name of the object we wish to create. The registry will then include the necessary class for us, and create (instantiate) the object for us. The registry then stores the newly created object in its array of objects. It is called a factory

# Registry + MVC

By combining the MVC architecture with the registry and front controller pattern, we now have a framework where all the requests come through a central file, which creates the registry, and creates the necessary controllers. The controllers create various models where appropriate, and in some cases, pass control to other controllers, before generating and manipulating the templates to generate the views as appropriate. The following diagram shows all of these components working together:

Web Service

Page Objects

Template Files

Authentication

API Class

Template Management

URL Processing

Registry Object

The Rest of the System

# Folder/File structure

Another important part of the system planning process is the directory structure that we are going to use. This will help me ensure that all files are properly organized, so that when we want to find or edit a particular file, we know exactly where to look. My use of the MVC and Registry patterns gives me a way to separate certain files, by classifying them as models, views, controllers or related to the registry:

* Controllers
  + Auth
    - Controller.php
  + Taxpayers
    - controller.php
* Models
* Registry
* Views
* Js
* Files

# CLASS/OBJECT DEFINITION

Each of the classes and php files that have been thoroughly commented which has been done to help simplify the step-by-step understanding.

## The registry object

The registry object itself is very straightforward; it needs to contain two arrays, one to store any settings and data we wish to centrally store within the registry, and one to store the core objects that we wish to access via the registry.

Most of the objects require access to the registry object, and this includes objects stored within the registry. To provide it access, we pass the registry object as a parameter to the objects constructor. Remember: this allows that object to *reference* this instance of the.

For each of these two arrays, we need two methods: one to store data or an object within the relevant array, and another to retrieve the data or object. Because we are going to use a Factory Method for storing objects, this code will be different from the code for storing settings.

### Registry objects

The registry object itself is the easy bit; its purpose is to hold data and other objects. It is the objects that will be held in here that will be more complicated. The objects that the registry will use will include:

* URL processing : urlprocessor.class.php
* Template management : template.class.php
* User authentication : auth.class.php
* Input validation : validator.class.php
* Page Structure management : page.class.php
* API/Web Service management : api.class.php

#### URL processing: urlprocessor.class.php

(Note: All the code and structure of the methods and details of this class can be found within the php file (registry/urlprocessor.class.php) , commented for the further descriptions

Since we are using a single frontend controller, we need to process the incoming

URL, in particular the page $\_GET variable, to work out how to handle the users

request. This is generally done by breaking the variable down in parts, separated

by a forward slash.

Manually setting the URL path is done using setURLPath(..), a simple setter method.

The getURLData method processes the incoming URL, and breaks it down into parts, building up an array of "URL bits".

The rest of our code needs to access the URL bits to determine what they need to do, so we use the method getURLBits()

The getURLBit() method is used to return one url bit from the bits array.

Another getter we need is to get the URL path, getURLPath()

#### Template management : template.class.php

(Note: All the code and structure of the methods and details of this class can be found within the php file (registry/template.class.php) , commented for the further descriptions

Every page request needs to display something to the user, and for each user the page will normally be different, and contain dynamic data from the webservice.

For example, when any user views the list of all registered taxpayers, they will all see the same page layout; however, the list of taxpayers will be different.

The template manager will be taking a series of template files, which contain the HTML to be sent to the browser, and manage data, which should be inserted into it, as well as process this dynamic replacement of data.

Additional templates will be able to be included within a template, should they be required.

The data and template contents will be stored in a Page object; the management of this object and its processing will be handled by the template object. Below is a walkthrough summary of summary of template class (registry/template.class.php).

Firstly, we need to create the object, which involves assigning our registry to a variable, including the page class, and instantiating a page object.

Since the views are made up of a number of template files, we need to be able to

include these files and send them to our page object. Certain pages might be made

up of two templates, others may be made up of three or more. To make this flexible,

instead of defining parameters for this method, we instead take however many

templates are passed as parameters and include them, in order, to our page object.

Within our template files, we may need to insert other templates. For instance, as we mentioned earlier, if one user has comments enabled on their profile, and another doesn't, then they will use the same main template, however, different templates will be inserted dynamically into them.

We can do this by taking a $tag (which is something contained within the template already included), and a template $bit, which is included and placed within the main template where the $tag was found.

These templates bits that we insert into our page object need to actually be replaced into the current page, which is where the replaceBits method comes in. This iterates through the list of template bits, and performs the replacement. The replacement is done in order, so if we wanted to insert a template into a page, and then insert another template into that one, we can do, so long as they were added in order.

The replacement is a simple str\_replace to find the tag, and replace it with the contents from the template.

Data that we wish to have placed into our templates works in a similar way to template bits, except that we can simply replace the tag with the data passed, as opposed to the contents of another file.

Upon receiving “GET” data from the web service and placing it in an array of custom data, we will iterate through the data as rows and place them on the page. To facilitate this, the replaceTags method also accepts data as an array; if it is an array, the first item of the array indicates the type of data (Query or Array) and the second array points to a cache reference, which indicates where it is stored in the database object. Control is then passed to a suitable method to perform more advanced replacements.

In summary, the template management mechanism operates in the following pattern:

* Our templates use template tags (such as {heading}) to indicate where dynamically generated data should be inserted.
* Sometimes, these template tags are placeholders for other files.
* Templates are parsed by the replaceTags method.
* Templates are inserted into template tags via the replaceBits method.
* If the replacement for a template tag contains another tag, there may be some template tag replacements that we need to do

When replacing a part of a template with a loop of data, the replacement is shown in the template slightly differently, normally starting with <!-- START tagname --> and ending with <!-- END tagname -->, containing a number of {tags} within.

The (cached) results of some data processing stored in an array. More often, this would be if we access data from the web service, and then modify the data afterwards. We would cache it, and send it to the template engine.

If we had a single row of data from an array of data fields from one of our models, we would be able to quickly convert all of this data into template tag variables. The following method does this for us, and to prevent overlap with existing tags, we can also pass a prefix that is added to the tag.

Finally, just before sending the output to the browser, we need to perform all of our replacements. This is achieved through the public function parseOutput()

This templating system replaces template tags formatted as {templatetag}, as opposed to $templatetag, {$templatetag}, or {$template->tag}.

#### User authentication : auth.class.php

(Note: All the code and structure of the methods and details of this class can be found within the php file (registry/auth.class.php) , commented for the further descriptions

The auth.class.php is being used for user authentication. The login and logout functionalities are located within this class. In total, there are 6 functions (constructor included) and 3 variables. The variables are $loggedin, $user, and $registry. The descriptions of every one of these variables can be found within the code itself.

The sessionAuthenticate() method authenticates a user if the neccessary session variables (email and password) are set. The login() method prompts the web service via the api class to checked if the user under authentication exists on web service’s database. In a similar manner, the logout() method is used to tell both the client and the web service that a logged in user is logging out, this method does the actual logging out.

#### Input validation : validator.class.php

(Note: All the code and structure of the methods and details of this class can be found within the php file (registry/validator.class.php) , commented for the further descriptions

To ensure that data passed between this client and the webservice it consumes on, there is a need to validate all data that will be passed between the two systems, and this is all achieved by the validator class. This class is made up of two variables, and four methods. The variable are $fields (an associative array to hold a list of all fields undergoing validation) and $registry to hold the $registry variable passed via the constructor upon instantiation of the validator object.

The four methods in include:

1. sanitizeData() method

This method uses php’s stripslashes() method to sanitize the data value from a field

1. validateEmail() method

called upon by the validate() method if the field under validation is of “*email*” type

1. validate() method

This method does the entire validation

1. diplayError() method

Used to display errors found any, from the fields under validation

#### Page Structure management : page.class.php

(Note: All the code and structure of the methods and details of this class can be found within the php file (registry/page.class.php) , commented for the further descriptions

The actual content from the templates and replacement data will be stored in our page object.

I have used some variables to store the replacement data, such as tags, post-parse tags and, the content of the page as defined by the templates it is built from.

We will need to set our page title variable and get it, so we need a getter and setter for this.

We will also need to be able to update the content variable, for instance, after adding a new template bit, or performing some replacement on the content. This will be achieved through the setContent method.

We need to be able to add tags to our replacement array using the following: public function addTag( … )

We also need to get the tags we wish to replace, so that our template object can perform the replacements using the method: public function getTags()

Before working on the bits, we will get a list of all the template bits we need to process into the page using the getBits() method.

We often need to just access a specific loop block within our page; this method makes this easy, by searching for us using regular expressions, and returning it. All this will be done via the getBlock(..) method.

Just like we need the setContent(..) method, we need to get the content from the page, so we use the getContent method.

Finally, when we are ready to output the content to the browser, we will do some final replacements. These are of template tags that we want to have in a template, but may not always replace.

Once this is done, the content is returned to be output to the browser. Via the getContentToPrint(..) method

#### API/Web Service management : api.class.php

(Note: All the code and structure of the methods and details of this class can be found within the php file (registry/api.class.php) , commented for the further descriptions

This class has been designed to isolate and manage all communications between this client system and the web services that it consumes. It is made up of just two methods; getDBResults() and processRequest().

The public method getDBResults() is used to access the JSon data received from the web service after the processRequest() method has been called.

The processRequest() function has 3 parameters; $data, $method (REST method), and $url. When the 3 arguments are passed to the function they will be used to query the web service.

## .htaccess

We are routing all of our requests through our index.php file, and by passing further information as the page $\_GET parameter, this results in URLs which look like: http://mrachallenge.com/?page=Taxpayers/delete/1012. This isn't a particularly nice looking URL, so we use the Apache mod\_rewrite module (which most hosts have installed by default—if you use WAMPServer for development, you may need to enable it in the Apache Modules menu), to take a nicer URL such as http://mrachallenge.com/Taxpayers/delete/1, which eliminates the need for ?page=, and translates it into the other format. This is rewritten by defining a rewrite rule in a .htaccess file within our code.

## Models

### Taxpayer Model – Taxpayer Class (Models/Taxpayer.php)

This class acts as a blueprint for all Taxpayer objects that will be used during one’s session on this system. As shown in the source in the Taxpayer.php file, the class contains all the variables that are attributes of a Taxpayer. It also contains all the setters and getters of the respective variables. One method worthy the mention is the save() method which is used in saving a new Taxpayer or an edit of an existing Taxpayer. The save method posts the data to the web service via the API Class

### User Model – User Class (Models/User.php)

This class employs encapsulation principles is made of 7 variables namely: firstname, lastname, email, password, loginfailurereason, and registry. All these variable except registry can be accessed through the getter ad setter methods that have been declared within the class. Besides the constructor and the getFullname() method (which is used to return a string concatenation of the firstname, space, and lastname), all of the methods in this class are either getters or setters of each of the above-mentioned variables as shown within the source of the class

## Views

This folder comprises of two main subfolders: the templates, and css folders.

### Templates

This is a folder made up of template biuts and sub-folders holding other template bits. These bits are used for defining the user interface of this system

### Css

The css folder is a collection of the stylesheets that have been used to style the HTML defined in the template bits

## Controllers

### Authcontroller Class (Auth/controller.php)

This class acts as a controller other sections of the system and the Auth registry object (Registry/auth.class.php). It contains an sub-abstraction of the login and logout methods. This class has the definitions that determine the necessary actions to be taken depending on the results from registry Auth class (in Registry/auth.class.php).

### Taxpayerscontroller Class (Taxpayers/controller.php)

This class handles the taxpayer objects and operations.