Chapter 1. Introduction to PushButtonCMS CMS Applications

Welcome to the exploration of CMS applications using PushButtonCMS!

A CMS application is a platform designed to simplify the creation, management, and delivery of digital content on the web. Unlike traditional static websites, a CMS application offers a dynamic environment that allows for easy content manipulation and organization.

In the context of PushButtonCMS, an application usually composed of two distinct modules, each serving a specific purpose:

1. **Visitor-Facing Module**: This module forms the public-facing aspect of the application. It caters to the audience visiting the website, offering a seamless and engaging experience. Content displayed here is accessible to all visitors.

2. **Registered/Privileged User Module**: The second module operates as the secured section of the application, available to registered or privileged users. It encompasses functionalities and content reserved for specific user groups, providing a more tailored and comprehensive experience.

Within these modules, various additional files contribute to the application's functionality and appearance:

- **Templates**: Structured layouts defining the visual presentation of content.
- **CSS and JavaScript Files**: Style and interactivity enhancements for a richer user experience.
- **Helper Libraries**: Additional tools aiding in the development and customization of modules.

Understanding these fundamental components sets the stage for comprehending how PushButtonCMS empowers developers and users to create versatile, engaging, and secure CMS applications. In the upcoming chapters, we'll delve deeper into the technical aspects of building and managing these modules within PushButtonCMS.

Sample Poll Application

Throughout this tutorial, we'll create a sample poll application as an illustrative example. This application will showcase the functionalities of PushButtonCMS in building a user-friendly poll creation and management system. We'll use this example to explore various aspects of CMS application development.

In the upcoming chapters, we'll delve deeper into the technical aspects of building and managing these modules within PushButtonCMS.

Chapter 2. Create Your First Module

This setup will create a basic module within PushButtonCMS. It is just a regular Hello World application - we will add the magic later. It is a visitor facing module, so we have no any restrictions here.

File Structure:

Create the following files in your PushButtonCMS project:

- `modules/polls.php` - file for controllers of the module
- `themes/default/polls.tpl` - file for views of the module

Controllers file

File: `modules/polls.php`

```
<?php

pb_on_action('sample', function () {
    pb_title('Polls title');
    pb_template('polls');
    pb_set_tpl_var('name', 'world');
});
```

Explanation:

A controller is created to activate upon triggering an associated action. Controller is typically a callback function, executing instantly upon the activation of the current action. In our example we created a controller for `view` action.

- `pb_on_action('view', function () { ... });`: Associates the 'view' action with a callback controller function.

Usually a callback function should set a page title, define which template should be used to display the data and provide a data to the template.

- `pb_title('Polls title');`: Sets the title of the page.
- `pb_template('polls');`: Specifies the template file `polls.tpl`. Don't use `.tpl` extension as it will be added by default.
- `pb_set_tpl_var('name', 'world');`: Sets a variable named *name*. You can have several `pb_set_tpl_var` executions to assign as much variables as you need.

You **cannot** use these reserved template variable names in `pb_set_tpl_var`:

- `action`
- `mode`
- `panel`

Template/Views file

File: `themes/default/polls.tpl`

```
{if $m.action eq 'view'}
    {include file="block_begin.tpl"}

    Hello {$m.name}! This is the polls index!

    {include file="block_end.tpl"}
{/if}
```

Explanation:

The template will receive all assigned data as the values of a `$m` array. The current action is accessible via `$m.action` variable.

- `{if $m.action eq 'view'}`: Checks the action in the URL.
- `{include file="block_begin.tpl"}`: Includes the beginning block template.
- `Hello {$m.name}! This is the polls index!`: Displays a greeting with a name passed from the controller.
- `{include file="block_end.tpl"}`: Includes the ending block template.

URL Structure:

To access the module, use the URL: `index.php?m=polls&d=view`

- `index.php`: Application entry point.
- `m=polls`: Indicates the 'polls' module.
- `d=view`: Specifies the 'view' action within the 'polls' module.

Testing:

Visit `index.php?m=polls&d=view` to trigger the 'view' action, rendering the 'polls.tpl' template and displaying the greeting message.

Chapter 3. Creating the Base Admin Module for Polls

The following file serves as a framework for an admin-facing module. We'll build upon this framework in upcoming chapters to add the actual code.

File: `modules/polls-admin.php`

```
<?php

/*
 * Module Name: Polls Management
 */

use PB\Access\PBAccess;
use PB\Common\Redirect;
use PB\Core\PBURL;
use PB\UI\UI;

pb_on_action('admin', function ()
    {
        pb_title('Polls Management');
        $ui=new UI();
        $ui->NotificationInfo('Polls Management administration');
        $ui->Output(true);
    });

pb_on_action('install', function ()
    {
        PBAccess::AdminRequired();
        pb_register_module(pb_current_module(), 'Polls Management');
        pb_register_autoload('poll-models');
        execsql("CREATE TABLE `polls` (
            `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
            `title` varchar(255) NOT NULL,
            `user_id` int(11) unsigned NOT NULL DEFAULT '0',
            PRIMARY KEY (`id`) )
            ENGINE=MyISAM DEFAULT CHARSET=utf8;");
        execsql("CREATE TABLE `poll_answers` (
                `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
                `poll_id` int(11) unsigned NOT NULL DEFAULT '0',
                `title` varchar(255) NOT NULL,
                `votes` int(11) unsigned NOT NULL DEFAULT '0',
                PRIMARY KEY (`id`), KEY `poll_id` (`poll_id`,`title`) )
                ENGINE=MyISAM DEFAULT CHARSET=utf8;");
        Redirect::Now(PBURL::CurrentModule('admin'));
    });

pb_on_action('uninstall', function ()
    {
        PBAccess::AdminRequired();
        pb_unregister_module(pb_current_module());
        pb_unregister_autoload('poll-models');
        execsql("DROP TABLE `poll_answers`;");
        execsql("DROP TABLE `polls`;");
        Redirect::Now(PBURL::AdminModulesManagement());
    });
```

Explanation:

This file defines three mandatory controllers, namely 'admin,' 'install,' and 'uninstall,' which form the core components of any administrative module.

- `admin`: This action serves as the module's dashboard, providing an interface for administrative tasks.
- `install`: Handles the installation process by creating necessary tables and registering the module within the CMS.
- `uninstall`: Handles the cleanup process when the module is uninstalled, ensuring the proper removal of the module and associated tables when uninstalling.

The top comment is mandatory for CMS detection and displaying the module in the Modules Management section for later installation.

```
/*
 * Module Name: Polls Management
 */
```

Instructions for Installation:

1. Log in with administrator credentials.
2. Navigate to the Modules Management section.
3. Click on the **Add Module** button.
4. Find 'Polls Management' iwithin the available modules and click **Install**.
5. The module should now be installed, redirecting the user to the admin dashboard of the Polls module.

What next?

This framework establishes the groundwork for an administrative module for polls, containing essential controllers required for administrative functionalities.

In the upcoming chapter, we will explore the functionalities and code within this file in greater detail. This deeper examination will provide a more thorough understanding of its operations and inner workings.

Chapter 4. Base Admin Module for Polls Explained

Let's explore the functionalities embedded within each section of the code snippet. This detailed examination unveils the purpose and significance of individual lines, unraveling the essential actions responsible for configuring, installing, and uninstalling the 'Polls Management' module within PushButtonCMS. Each section plays a crucial role in defining the administrative interface, managing database tasks, and ensuring a seamless integration of the module, shedding light on its inner workings.

Use Statements

Imports necessary classes for accessing PushButtonCMS functionalities.

```
use PB\Access\PBAccess;
use PB\Common\Redirect;
use PB\Core\PBURL;
use PB\UI\UI;
```

Controller for 'admin' action

This action serves as the module's dashboard, providing an interface for administrative tasks. Let's display a sample message for now. We will add more code here later.

```
pb_on_action('admin', function () { ... });
```

Define the title of the page as 'Polls Management'.

```
pb_title('Polls Management');
```

Notification Display & UI Output:

Here, we will utilize the basic UI component to simplify view creation. The UI() class helps create a user interface without building a template file. You just need to use the standard components from `\PB\UI` library.

Initializing the UI component:

```
$ui = new UI();
```

Displaying a simple information notification:

```
$ui->NotificationInfo('Polls Management administration');
```

To complete the interface initialization, execute Output() with true. This invokes a default UI template and tells the CMS to display it.

```
$ui->Output(true);
```

Controller for 'install' action

This controller manages installation tasks for the module.

Here we will register the module so CMS will know that it exists. Also, we will create needed tables for later usage.

```
pb_on_action('install', function () { ... });
```

Ensure only administrators can launch the installation process. If the user is not an administrator, a default access denied page will be displayed, and the controller code following this line will not be executed.

```
PBAccess::AdminRequired();
```

Register the module in the CMS installed modules:

```
pb_register_module(pb_current_module(), 'Polls Management');
```

Inform the CMS about an additional autoloading file for additional functions (in our case for declaring the models). We will create the file later.

```
pb_register_autoload('poll-models');
```

Create the 'polls' and 'poll_answers' tables in the database:

```
execsql("CREATE TABLE `polls`...");
execsql("CREATE TABLE ...");
```

Be cautious when using the `execsql(...)` function as it enables the execution of virtually any SQL statement, which could pose security risks.

And the `Redirect::Now(<url>)` statement will immediately redirect the visitor to the `<url>`. No code after this statement will be executed. `PBURL::CurrentModule(<action>)` generates a URL for launching a specified action within the current module. Therefore, the next block will redirect the user to the **admin** controller described above.

```
Redirect::Now(PBURL::CurrentModule('admin'));
```

Controller for 'uninstall' action

Handles cleanup when the module is uninstalled.

```
pb_on_action('uninstall', function () { ... });
```

Restrict anyone from uninstalling the module via the browser, for example, by opening `index.php?m=polls-admin&d=uninstall` in browser.

```
PBAccess::AdminRequired();
```

Unregister 'Polls Management' from the CMS and remove the autoloading file.

```
pb_unregister_module(pb_current_module());
pb_unregister_autoload('poll-models');
```

Delete the 'poll_answers' and 'polls' tables from the database as they are no longer needed.

```
execsql("DROP TABLE `poll_answers`;");
execsql("DROP TABLE `polls`;");
```

Redirect the user to the modules administration page using `PBURL::AdminModulesManagement()`.

```
Redirect::Now(PBURL::AdminModulesManagement());
```

What's Next

In the next chapters, we will build a simple poll management tool with basic `UI` components.

Chapter 5. Adding a Simple Form for Poll Creation

This section introduces a basic form setup for creating polls within the PushButtonCMS environment. The form enables users to input poll questions and answers. The data handling functionality will be integrated in subsequent chapters.

Use Statements

These 'use' statements are essential for accessing and utilizing various functionalities within PushButtonCMS.

```
use PB\Access\PBAccess;
use PB\Common\Redirect;
use PB\Core\PBURL;
use PB\PB;
use PB\UI\Buttons;
use PB\UI\Form;
use PB\UI\UI;
```

Controller for 'admin' Action

We modify the admin dashboard by adding a button to create a new poll. The `\PB\UI\Buttons` component allows displaying one or more buttons. Thus, we remove the sample notification and add a button instead.

```
pb_on_action('admin', function ()
    {
        pb_title('Polls Management');
        $ui = new UI();

        // Initialize a component for buttons
        $b = new Buttons();
        // Adding a button to redirect to the 'add-poll' action for creating a new poll
        $b->Button('Add Poll', PBURL::CurrentModule('add-poll'));
        // Adding the button to the UI
        $ui->Add($b);

        $ui->Output(true);
    });
```

Controller for 'create-poll' Action

The 'create-poll' action will handle the actual saving of poll data, to be implemented in subsequent updates.

```
pb_on_action('create-poll', function ()
    {
        // We will add actual saving here later
        Redirect::Now(PBURL::CurrentModule('admin'));
    });
```

It's important to have a 'create-poll' controller before the 'add-poll' controller in the code. This organization will aid in error handling later.

Controller for 'add-poll' Action

This controller manages the addition of a new poll by presenting a form for users to input poll details. We'll use a `Form` component. Usage for a POST request is simple: `new Form(<url to handle form>)`.

We will add four text fields using `AddText(<variable name>, <field title>, <true if this field is required>)`. The first one will have the cursor in it (that's why we use `SetFocus()`) to enhance user input convenience. Question and two answers are mandatory, while the third answer is optional.

```
pb_on_action('add-poll', function () {
    pb_title('Add Poll');
    $ui = new UI();

    // Create a new form
    $f = new Form(PBURL::CurrentModule('create-poll'));
    // Adding form elements for poll creation
    $f->AddText('question', 'Poll Question', true)->SetFocus();
    $f->AddText('answer1', 'Answer 1', true);
    $f->AddText('answer2', 'Answer 2', true);
    $f->AddText('answer3', 'Answer 3');
    // Load values submitted via POST request into the form - for error handling later
    $f->LoadValuesArray(PB::Requests()->POSTAsArray());
    // Add the form to the UI
    $ui->Add($f);

    $ui->Output(true);
});
```

By default, the `Form` element represents an HTML-form with a POST method.

What's Next

Subsequent chapters will focus on implementing functionality to handle the submitted poll data.

Chapter 6. Form Error Handling

In this section, we'll delve into error handling mechanisms integrated into the 'add-poll' and 'create-poll' actions. These enhancements enable effective validation and display of error messages when creating polls.

Controller for 'create-poll' Action

The 'create-poll' action is responsible for validating the submitted data before proceeding with poll creation. It checks if essential fields such as question and answers are provided. If any field is empty, it adds an error message using `PB::Errors()->AddError()`. If no errors are found, the poll is considered created, a success popup notification is displayed using `pb_notify('Poll created!')`, and the user is redirected to the admin dashboard.

```
pb_on_action('create-poll', function ()
    {
        if (PB::POST('question')->isEmpty())
            PB::Errors()->AddError('Question required');
        if (PB::POST('answer1')->isEmpty())
            PB::Errors()->AddError('First answer required');
        if (PB::POST('answer2')->isEmpty())
            PB::Errors()->AddError('Second answer required');

        if (!PB::Errors()->HasErrors())
            {
                // We will add actual saving here later
                pb_notify('Poll created!');
                Redirect::Now(PBURL::CurrentModule('admin'));
            }
        else
            {
                pb_set_action('add-poll');
            }
    });
```

To access POST data, `PB::POST(<name>)` could be used. To access GET data, we can use `PB::GET(<name>)`. Both of them will return a `RequestParam` object for the respective `<name>` variable within GET or POST. You can use the following methods to work with the values:

- `Exists()` will detect an absent value (for example if there was no `<name>` key in the POST request).
- `isEmpty()` will detect absent or empty values.
- `AsString()`, `AsFloat()`, `AsInt()`, `AsBool()`, `AsArray()` will return the value converted to the described type.
- `AsAbsInt()` will also make negative values positive.
- `AsStringOrDefault(<default-value>)` will return `<default-value>` if the value is empty or absent.
- `isStringEqual(<string-to-compare>)` helps in quick comparisons.

Please mention `pb_set_action('add-poll');` at the end of the controller function. It changes the active action to `add-poll`, but it will not immediately trigger the controller for the `add-poll` action. The controller for the `add-poll` action will actually be executed after the completion of the `create-poll` controller if the `add-poll` controller is present in the code below. If the code for the `add-poll` controller is above the `create-poll` or it is not in this file, the `add-poll` controller will not be executed.

Controller for 'add-poll' Action

The 'add-poll' action handles the presentation of the form for adding a new poll. It incorporates error handling by displaying UI errors using `PB::Errors()->DisplayUIErrors($ui)`. `PB::Errors()` can be filled in `create-poll` controller. If errors are present, the form displays the error messages.

```
pb_on_action('add-poll', function ()
     {
        pb_title('Add Poll');
        $ui = new UI();

        // Display UI errors, if any
        PB::Errors()->DisplayUIErrors($ui);

        // Create a new form
        $f = new Form(PBURL::CurrentModule('create-poll'));

        // Adding form elements for poll creation
        $f->AddText('question', 'Poll Question', true)->SetFocus();
        $f->AddText('answer1', 'Answer 1', true);
        $f->AddText('answer2', 'Answer 2', true);
        $f->AddText('answer3', 'Answer 3');

        // Load values submitted via POST request into the form for error handling
        $f->LoadValuesArray(PB::Requests()->POSTAsArray());

        // Add the form to the UI
        $ui->Add($f);
        $ui->Output(true);
    });
```

Let's take a closer look at the following code line:

```
$f->LoadValuesArray(PB::Requests()->POSTAsArray());
```

You can try omitting this block and opening the Add Poll page. You won't see any difference because there's actually nothing to do in this situation. However, if you attempt to submit a form with invalid data, you will encounter an empty form after the error. The code `$f->LoadValuesArray(...)` is intended to preload provided data into the form inputs. It retrieves the values from an array if there are keys with the names of the input elements. The `PB::Requests()` helper can provide you with GET and POST data as an array using `POSTAsArray()` or `GETAsArray()`.

What's Next

These implementations ensure proper validation and error handling during the poll creation process, offering users clear guidance and notifications when essential information is missing or incorrectly entered.

In next chapters we will create models to be able to save the data.

Chapter 7. Creating Basic Models for Polls and Poll Answers

To interact with the database, we need models for polls and poll answers. Let's create a file named `modules/preload/poll-models.php` (remember our autoload registration - this file will be included before any controller, making the models available to all controllers).

Poll model

In this file, we'll start by declaring a class that extends the base `PBDBObject` class, representing an active record with a rich model pattern.

```
class Poll extends \PB\DB\ORM\PBDBObject
    {
        // Methods to be defined...
    }
```

This class requires the declaration of three methods:

- `TableName()`: Should return the database table name for this object.
- `FieldNameForID()`: Represents the field name for the primary key. The primary key is accessible via the `ID()` method and is treated as an integer.
- `FieldNameForTitle()`: Represents the field name for the title. The title will be accessible using the `Title()` method and will be treated as a string. If there is no title field, you should return an empty string.

For example:

```
public static function TableName(): string
    {
        return 'polls';
    }

public static function FieldNameForID(): string
    {
        return 'id';
    }

public static function FieldNameForTitle(): string
    {
        return 'title';
    }
```

Additionally, declare a function to access the `user_id` field:

```
public function UserID(): int
    {
        return $this->FieldIntValue('user_id');
    }
```

This uses the `FieldIntValue(<field-name>)` method, providing various helpers for accessing fields as different types. The convention suggests using specific methods to access specific field types, such as `FieldFloatValue()`, `FieldStringValue()`, etc.

Field Value Helper Methods Explained

Within PushButtonCMS models, various `Field...Value()` methods exist to facilitate the retrieval and interpretation of data from database fields. These methods are tailored to handle different data types effectively:

- `FieldFloatValue()` - Retrieves the value from a field and converts it to a floating-point number.
- `FieldMoneyValue()` - Similar to `FieldFloatValue()`, this method specifically caters to monetary values by rounding the float to two decimal places, commonly used for financial data.
- `FieldIntValue()` - Fetches the value from a field and interprets it as an integer.
- `FieldBoolValue()` - Used to interpret integer fields as boolean values. It returns `true` if the field's value is greater than zero, otherwise `false`.
- `FieldStringValue()` - Retrieves the value from a field and treats it as a string.

These helper methods are designed to streamline data retrieval from database fields while ensuring appropriate interpretation according to specific data types within PushButtonCMS models.

How to Create a Record

We're only setting up reading methods for now, with no write methods declared. However, we need to create the poll somehow. Let's add a static method for this:

```
public static function Create(string $poll_question, int $created_by_user_id): self
{
    return self::CreateWithParams([
        'title' => $poll_question,
        'user_id' => $created_by_user_id,
    ]);
}
```

This `Create` method will generate a row in the database table and return an object representing the created row. The convention suggests using a `Create` method for object creation, allowing usage with `Poll::Create(...)`.

Poll Answer Model

Let's also create a similar class for poll answers, keeping the structure similar. The combined file will look like this:

```
<?php

/**
 * @method static self initSingleton($id)
 * @method static self UsingCache($id)
 * @method static self initNotExistent()
 */
class Poll extends \PB\DB\ORM\PBDBObject
    {

        public static function TableName(): string
            {
                return 'polls';
            }

        public static function FieldNameForID(): string
            {
                return 'id';
            }

        public static function FieldNameForTitle(): string
            {
                return 'title';
            }

        public function UserID(): int
            {
                return $this->FieldIntValue('user_id');
            }

        public static function Create(string $poll_question, int $created_by_user_id): self
            {
                return self::CreateWithParams([
                    'title'=>$poll_question,
                    'user_id'=>$created_by_user_id,
                ]);
            }

    }

/**
 * @method static self initSingleton($id)
 * @method static self UsingCache($id)
 * @method static self initNotExistent()
 */
class PollAnswer extends \PB\DB\ORM\PBDBObject
    {

        public static function TableName(): string
            {
                return 'poll_answers';
            }

        public static function FieldNameForID(): string
            {
                return 'id';
            }

        public static function FieldNameForTitle(): string
            {
                return 'title';
            }

        public function PollID(): int
            {
                return $this->FieldIntValue('poll_id');
            }

        public function VotesCount(): int
            {
                return $this->FieldIntValue('votes');
            }

        public static function Create(Poll $poll, string $answer_title): self
            {
                return self::CreateWithParams([
                    'title'=>$answer_title,
                    'poll_id'=>$poll->ID(),
                    'votes'=>0,
                ]);
            }

    }
```

This file contains a DocBlock before every class, aiding development with widely-used IDEs. It introduces useful helper functions for handling object initialization and cache usage, making object manipulation more efficient and consistent.

DocBlock Functions Explained

- `initSingleton(<primary-key-value>)` - Creates a singleton object associated with the provided primary key value. Any subsequent objects with the same primary key initialized with `initSingleton` will reference the same object instance, enhancing memory efficiency.
- `UsingCache(<primary-key-value>)` - Optimizes performance by caching object data. Newly created objects with the specified primary key will be initialized using cached data. Be cautious: modifications to objects won't update the cache, potentially leading to outdated data if not managed carefully.
- `initNotExistent()` - Facilitates the creation of an object instance pointing to a non-existent (not found) row in the database table. This instance cannot be updated but is helpful for initializing objects when actual data isn't present.

What's next

In the next chapter, we'll utilize these models to create polls.

Chapter 8. Writing Poll Data to the Database

Let's Update the Controller

The 'create-poll' controller within 'modules/poll-admin.php' has undergone significant updates to handle the storage of poll-related data into the database.

This revised controller has been designed to handle the input of poll questions and up to three answers, ensuring a robust storage mechanism in the database using the `Poll` and `PollAnswer` models.

Let's delve into the specifics of these modifications and why they were structured as such:

```
pb_on_action('create-poll', function ()
    {
        if (PB::POST('question')->isEmpty())
            PB::Errors()->AddError('Question required');
        if (PB::POST('answer1')->isEmpty())
            PB::Errors()->AddError('First answer required');
        if (PB::POST('answer2')->isEmpty())
            PB::Errors()->AddError('Second answer required');
        if (!PB::Errors()->HasErrors())
            {
                // Creating a new poll entry
                $poll = Poll::Create(PB::POST('question')->EscapedString(), PB::User()->ID());

                // Creating entries for answers 1 and 2
                PollAnswer::Create($poll, PB::POST('answer1')->EscapedString());
                PollAnswer::Create($poll, PB::POST('answer2')->EscapedString());

                // Creating an entry for answer 3 if provided
                if (!PB::POST('answer3')->isEmpty())
                    PollAnswer::Create($poll, PB::POST('answer3')->EscapedString());

                pb_notify('Poll created!');
                Redirect::Now(PBURL::CurrentModule('admin'));
            }
        else
            pb_set_action('add-poll');
    });
```

Changes Explained

Handling Question and Answers

- **`$poll = Poll::Create(...)`**: This line initializes a new 'Poll' object, capturing the submitted question and associating it with the currently logged-in user.

- **`PollAnswer::Create(...)`**: For each answer (answer 1 and answer 2), a corresponding 'PollAnswer' entry is created, linked to the newly formed poll. If a third answer is provided in the form, another 'PollAnswer' entry is generated, continuing the association with the same poll.

The decision to create entries separately for each answer within the 'PollAnswer' table allows for scalability and ease of retrieval when handling multiple answers associated with a single poll. By structuring the code this way, the system accommodates flexibility in the number of potential answers while maintaining a clear linkage to the corresponding poll.

Safeguarding Against HTML/JS Injection

When handling form inputs submitted by users, especially those destined for display on web pages, sanitizing the input data is crucial

The usage of `PB::POST(<var-name>)->EscapedString()` primarily serves as a safety measure against HTML and JavaScript injections. It escapes characters in the string, ensuring that special characters (like angle brackets, quotes, etc.) are properly handled.

In the context of form submissions for poll creation (`Poll` and `PollAnswer` models), using `EscapedString()` on the POST data ensures that user-supplied values for the poll question and answers are sanitized against HTML/JS injections. This practice helps to maintain the integrity and security of the application by neutralizing characters that could potentially be exploited to inject harmful scripts or content into the webpage.

Future Expansion

It's essential to note that in upcoming developments, we'll enhance the functionality to handle an arbitrary number of answers more conveniently. This will streamline the process of managing polls with numerous answers, offering a more adaptable solution for varied polling scenarios.

Chapter 9. Declaring Lists for Data Retrieval

Leveraging PBDBList for Data Retrieval

To efficiently manage and retrieve data from the database, the `PBDBList` class provides a flexible and robust solution. It offers methods for setting filters, specifying limits and offsets, and fetching data from the database.

In the context of our polls application, we have created two lists: `PollList` and `PollAnswersList`. These lists extend the `PBDBList` class and declare the specific object class they handle using the `ObjectClassName` method.

Here is an overview of the methods provided by `PBDBList`.

Methods for Data Retrieval

Working with Preloaded Data

When working with preloaded data, set filters, limits, offsets, and use the `Load()` method to preload the filtered dataset. Then, employ the following methods:

- `Item($index)`: Retrieves a specific item at the given index.
- `EachItem()`: Iterates over each item in the list.
- `FirstItem()`: Retrieves the first item in the list.
- `LastItem()`: Retrieves the last item in the list.
- `GetItemWithID($id)`: Retrieves an item based on its ID.

Working with Large Datasets without Preloading

When dealing with large datasets without preloading, set filters, limits, offsets, and use the `Open()` method to access the dataset. Utilize:

- `Fetch()`: Retrieves the next item in the filtered dataset.

Additional Methods for List Management

- `SetFilterField...`: These methods set filters based on various field types, such as `SetFilterFieldIntValue`.
- `OrderByField($field, $asc)`: This method orders the list based on a specified field and in either ascending or descending order.

These methods offer a powerful means of customizing and optimizing data retrieval. They are exclusively available for use within PBDBList descendant classes, ensuring that data logic remains encapsulated within the list classes.

Example Implementation

Below are examples of two lists, `PollList` and `PollAnswersList`, each inheriting from `PBDBList`. These classes are used to manage datasets for polls and their associated answers.

Let's add two lists to `modules/preload/poll-models.php`.

```
use PB\DB\ORM\PBDBList;

//Previous code...

/**
 * @method Poll Item($index)
 * @method Poll[] EachItem()
 * @method Poll|NULL Fetch()
 * @method Poll|NULL FirstItem()
 * @method Poll|NULL LastItem()
 * @method Poll|NULL GetItemWithID($id)
 */
class PollList extends PBDBList
    {

        protected function ObjectClassName(): string
            {
                return Poll::class;
            }

        public function FilterUser(int $user_id): void
            {
                $this->SetFilterFieldIntValue('user_id', $user_id);
            }

    }

/**
 * @method PollAnswer Item($index)
 * @method PollAnswer[] EachItem()
 * @method PollAnswer|NULL Fetch()
 * @method PollAnswer|NULL FirstItem()
 * @method PollAnswer|NULL LastItem()
 * @method PollAnswer|NULL GetItemWithID($id)
 */
class PollAnswersList extends PBDBList
    {

        protected function ObjectClassName(): string
            {
                return PollAnswer::class;
            }

        public function FilterPoll(int $poll_id): void
            {
                $this->SetFilterFieldIntValue('poll_id', $poll_id);
            }

        public function OrderByVotes(bool $asc=true): void
            {
                $this->OrderByField('votes', $asc);
            }

    }
```

These lists allow for setting filters, sorting, and efficient data retrieval, enhancing the management of dataset interactions within the PushButtonCMS environment.

PollList

`PollList` extends `PBDBList`, a class that facilitates managing and retrieving data from a database. This particular list deals specifically with polls. Let's explore its components:

The mandatory `ObjectClassName()` method defines the class name of the objects this list holds, which is `Poll` in this case. It ensures that the list manages and contains instances of the specified class.

```
 protected function ObjectClassName(): string
    {
        return PollAnswer::class;
    }
```

The next method filters the list based on the `user_id` field. This function helps retrieve a subset of polls based on the user context. It sets the filter to restrict data retrieval to only those polls associated with the provided user ID.

```
public function FilterUser(int $user_id): void
    {
        $this->SetFilterFieldIntValue('user_id', $user_id);
    }
```

PollAnswersList

Similarly, `PollAnswersList` extends `PBDBList` and manages a list of `PollAnswer` objects.

Like in `PollList`, the `ObjectClassName()` method specifies the class name of the objects this list holds, which is `PollAnswer`. It ensures that the list contains instances of the specified class.

This method filters the list based on the poll ID. It restricts data retrieval to poll answers associated with the provided poll ID. It helps to fetch answers specific to a particular poll.

```
public function FilterPoll(int $poll_id): void
    {
        $this->SetFilterFieldIntValue('poll_id', $poll_id);
    }
```

Let's add a method to order the list based on the number of votes. It arranges the poll answers in either ascending or descending order based on the number of votes each answer has received. This function assists in presenting poll answers based on their popularity or other voting criteria.

```
public function OrderByVotes(bool $asc=true): void
    {
        $this->OrderByField('votes', $asc);
    }
```

- `OrderByID(bool $asc=true)`: Orders the list based on the object IDs. It arranges the objects in either ascending (`$asc=true`) or descending (`$asc=false`) order of their IDs.
- `OrderByTitle(bool $asc=true)`: Orders the list based on the titles of the objects. It arranges the objects in either ascending or descending order of their titles.

What's Next

In the next chapters we'll use declared lists to dislpay admin interface and a voting page.

Chapter 10. Displaying a Poll List in Admin Interface

To visualize the poll list in the administrative section, let's integrate the `Grid` UI component. Begin by including the `Grid` class in the 'use' section of your code in `modules/polls-admin.php`.

```
use PB\UI\Grid;
```

In the 'admin' controller, enhance the interface by adding a button to view the polls:

```
pb_on_action('admin', function () {
    // ...
    $b->Button('View Polls', PBURL::CurrentModule('view-polls'));
    // ...
});
```

Now, implement the 'view-polls' controller to render the list of polls. This controller manages the display of the polls within a grid layout.

```
pb_on_action('view-polls', function ()
    {
        pb_title('Polls');
        $ui = new UI();
        $grid = new Grid();

        // Column Additions and Setup
        $grid->AddCol('title', 'Title');
        $grid->AddCol('view', 'View on website');
        $grid->AddEdit();
        $grid->AddDelete();

        $polls = new PollList();
        $polls->FilterUser(PB::User()->ID());
        $polls->Limit(10);
        $polls->Offset(PB::GET('from')->AsInt());
        $polls->OrderByID(false);
        $polls->Load();

        foreach ($polls->EachItem() as $poll)
            {
                // Populate Grid with Poll Data
                $grid->Label('title', $poll->Title());
                $grid->URL('title', PBURL::CurrentModule('answers', ['id'=>$poll->ID()]));
                $grid->Label('view', 'View on website');
                $grid->URL('view', PBURL::Module('polls', 'details', ['id'=>$poll->ID()]), true);
                $grid->NewRow();
            }

        if ($grid->RowCount() === 0)
            $grid->SingleLineLabel('Nothing found');

        $ui->Add($grid);
        $ui->AddPaginatorForList($polls);
        $ui->Output(true);
    });
```

Retrieving Polls with PollList

Let's configure the `PollList` to display a limited set of polls, organized with the help of the paginator.

```
$polls = new PollList();
$polls->FilterUser(PB::User()->ID());
$polls->Limit(10);
$polls->Offset(PB::GET('from')->AsAbsInt());
$polls->OrderByID(false);
$polls->Load();
```

Let's break down the code:

Instantiating PollList

Here, a new instance of the `PollList` class is created, indicating that we are dealing with a list of polls previously declared in `modules/preload/poll-models.php`.

```
$polls = new PollList();
```

Applying User Filter

The `FilterUser` method is used to filter the polls based on the current user's ID. This ensures that only polls associated with the logged-in user are fetched.

```
$polls->FilterUser(PB::User()->ID());
```

Setting Data for Pagination

The `Limit` method restricts the number of polls to be retrieved to some reasonable value. In this case, it limits the result to 10 polls.

```
$polls->Limit(10);
```

The `Offset` method determines the starting point of the polls to be displayed. It uses the 'from' GET parameter, which is automatically created by the paginator. The `AsAbsInt` method ensures that the offset is a positive integer.

```
$polls->Offset(PB::GET('from')->AsAbsInt());
```

Ordering by ID

The `OrderByID` method specifies the sorting order for the polls. In this case, it sorts them by ID in descending order (`false` indicates descending order).

```
$polls->OrderByID(false);
```

Loading Polls

Finally, the `Load` method executes the query and loads the polls based on the specified configuration.

```
$polls->Load();
```

This configuration prepares the `PollList` to fetch a limited set of polls according to the specified criteria, facilitating an organized and user-friendly display in the code below.

Iterating Through Polls for Display

To iterate through the list of polls fetched using the `PollList` instance, the code utilizes a foreach loop with the `EachItem()`

```
foreach ($polls->EachItem() as $poll)
     {
         // Code block to handle each poll
     }
```

The `EachItem()` method is used within a foreach loop to iterate through each poll fetched by the `PollList` instance, assigning each poll object to the variable `$poll`. Inside this loop, you can access individual poll objects and perform operations or display information related to each poll. For instance, you can access specific attributes of the poll, such as its title, ID, or any other relevant data, to populate a user interface or perform specific actions for each poll in the list.

Grid Component

The Grid component in PushButtonCMS is a versatile tool used to construct and display tabular data in a structured layout. It allows for the creation of tables with columns, each designed to showcase specific information. This component facilitates various functionalities within the grid, such as adding columns, including edit and delete actions, and displaying data rows efficiently. With features like pagination and customizable column settings, the Grid component provides a user-friendly interface for managing and presenting data in a systematic manner.

Grid::AddCol

The `Grid::AddCol()` method is used to define and add columns to the grid layout. It specifies the columns to be displayed in the grid, assigning each column a label and an identifier. In the provided code, `AddCol('title', 'Title')` creates a column labeled 'Title' that displays the titles of the polls.

Grid::AddEdit

The `Grid::AddEdit()` method is employed to include an 'Edit' action within the grid, typically as a column and a specific icon for each row. It sets up an editing feature that allows users to modify or update the content displayed in a row, providing a link to trigger the editing functionality.

Grid::AddDelete

The `Grid::AddDelete()` method is used to incorporate a 'Delete' action within the grid, often as a separate column. It establishes a mechanism to remove or delete specific rows of data from the grid or the associated dataset. Typically, this functionality would be activated through a button or link provided in the grid.

In the provided code:

- `AddCol('title', 'Title')` creates a column labeled 'Title' to display the titles of the polls.
- `AddCol('view', 'View on website')` creates a column labeled to allow to open a poll page for regular visitors.
- `AddEdit()` configures an 'Edit' action within the grid, presumably allowing users to modify the poll details.
- `AddDelete()` sets up a 'Delete' action, likely enabling the admin to remove selected polls from the list.

Grid::Label

The `Grid::Label()` method facilitates the insertion of text-based content into specific grid cells. It populates the designated cell with the provided text, in this case, the title of the polls.

Grid::URL

The `Grid::URL()` method is used to generate clickable links within the grid layout. It adds a hyperlink to a particular cell, enabling users to navigate to other pages or access further details associated with the polls.

In the context of this code:

- `Grid::URL('title', PBURL::CurrentModule('answers', ['id'=>$poll->ID()]));` sets up a link that directs to the voting page for a specific poll within the administrative section.
- `Grid::URL('view', PBURL::Module('polls', 'details', ['id'=>$poll->ID()]), true);` configures a link that leads to a standard view page on the website for a given poll. The last parameter `true` means that the link should be open in a new tab or window of the browser.

Keep in mind that these pages haven't been created yet; we'll implement them in forthcoming chapters.

Finalizing Grid Row

Once all essential columns and data for a specific row in the grid have been added, the `NewRow()` method is employed to denote the completion of the current row.

This method serves as a delimiter, signaling the conclusion of data insertion for the current row in the grid layout, and subsequently readies the grid to progress to the next row.

Paginator

The paginator feature plays a crucial role in managing and navigating through the poll list efficiently. It's conveniently added using the following code:

```
$ui->AddPaginatorForList($polls);
```

This paginator operates by creating a 'from' GET parameter, which serves as an offset within the list. It's seamlessly integrated and initialized directly from the current PBDBList, streamlining the process of navigating through multiple polls without overwhelming the interface.

Chapter 11. Displaying Poll Details for Visitors

To showcase the details of a poll without the voting functionality at this stage, we'll create a view that presents the poll question and its associated answers.

Preparing Poll Answers List

To retrieve the list of answers for a poll in a convenient way, a method named `Answers()` has been added to the `Poll` model. This method fetches the associated answers using `PollAnswersList` and arranges them by ID.

```
class Poll extends PBDBObject
    {
        // ... existing code ...

        public function Answers(): PollAnswersList
            {
                $poll_answers = new PollAnswersList();
                $poll_answers->FilterPoll($this->ID());
                $poll_answers->OrderByID();
                $poll_answers->Load();
                return $poll_answers;
            }
    }
```

Styling the Poll Display

A CSS file, `themes/default/css/polls.css`, has been created to stylize the poll elements.

```
.poll-question {
    font-size: 2em;
    padding: 0.2em;
}

.poll-answer {
    font-size: 1.5em;
    padding: 0.1em 1em;
}
```

Details Controller

A new controller for the 'details' action in `modules/polls.php` has been set up to handle the display of poll details. It fetches the poll and its associated answers, sets the title, and includes the CSS file.

```
use PB\PB;

pb_on_action('details', function ()
    {
        $poll=new Poll(PB::GET('id')->AsInt());
        if (!$poll->Exists())
            return;

        pb_title('Vote!');
        pb_add_cssfile('css/polls.css');

        pb_template('polls');

        pb_set_tpl_var('question', $poll->Title());

        $answers=[];
        foreach ($poll->Answers()->EachItem() as $poll_answer)
            {
                $answers[]=[
                    'id'=>$poll_answer->ID(),
                    'title'=>$poll_answer->Title(),
                ];
            }
        pb_set_tpl_var('answers', $answers);

    });
```

The controller begins by retrieving the poll based on the provided ID using the Poll model. If the requested poll does not exist, the controller returns a 404 error result.

```
$poll=new Poll(PB::GET('id')->AsInt());
if (!$poll->Exists())
    return;
```

Adding CSS File

The `pb_add_cssfile(<path-within-theme>)` function is used to include a CSS file for styling the poll display. This function looks for the specified CSS file in the current theme's directory. If the file is not found in the current theme, it defaults to using the same file located in the 'themes/default/' directory.

In our case `themes/default/css/polls.css` will be included.

Providing Data For the Template

This line specifies the template file named 'polls.tpl' that will be used for rendering the content. It sets up the rendering environment for the subsequent data.

```
pb_template('polls');
```

Here, the title of the poll fetched from the database is assigned to the template variable 'question'. This will enable the 'question' variable to be accessed and used within the 'polls.tpl' template.

```
pb_set_tpl_var('question', $poll->Title());
```

This part initializes an empty array named 'answers' and populates it with details of each answer associated with the poll. It iterates through the list of answers using the `EachItem()` method provided by the `PollAnswersList` and constructs an array containing each answer's ID and title.

```
$answers=[];
foreach ($poll->Answers()->EachItem() as $poll_answer)
    {
        $answers[]=[
            'id'=>$poll_answer->ID(),
            'title'=>$poll_answer->Title(),
        ];
    }
```

Finally, this line assigns the array of answers to the template variable `answers`. It allows the `answers` variable to be accessed and utilized within the `polls.tpl` template for displaying the list of answers associated with the poll.

```
pb_set_tpl_var('answers', $answers);
```

This code prepares the necessary data to be displayed within the `polls.tpl` template, ensuring the availability of the poll's title and associated answers for rendering in the front-end.

View Template

The view template in `themes/default/polls.tpl` renders the poll question and its associated answers.

```
{if $m.action eq 'details'}
    {include file="block_begin.tpl"}

    <div class="poll-question">{$m.question}</div>

    {foreach from=$m.answers item=answer}
        <div class="poll-answer">{$answer.title}</div>
    {/foreach}

    {include file="block_end.tpl"}
{/if}
```

Here's a breakdown of the elements within this template:

- `{if $m.action eq 'details'}`: This conditional check verifies whether the current action is 'details' to ensure that the subsequent content is displayed only when the action matches 'details'.
- `{include file="block_begin.tpl"}` and `{include file="block_end.tpl"}`: These lines are responsible for including the 'block_begin.tpl' and 'block_end.tpl' files of block enclosing templates.
- `<div class="poll-question">{$m.question}</div>`: This snippet renders the poll question. The question's content is fetched from the template variable `question`, which was set in the controller logic (`pb_set_tpl_var('question', $poll->Title());`).
- `{foreach from=$m.answers item=answer}`: This 'foreach' loop iterates through the `answers` array, assigned in the controller logic (`pb_set_tpl_var('answers', $answers);`). For each answer in the `$m.answers` array, it assigns the current answer to the variable `answer`.
- `<div class="poll-answer">{$answer.title}</div>`: Inside the loop, this line displays the title of each answer in a separate 'poll-answer' styled division. It fetches the title of each answer from the `answer` variable within the loop.

Checking Created Code via Interface Navigation

- Open the Modules Management interface within your application.
- From the Modules Management interface, locate and access the 'Polls Management' section.
- Within the 'Polls Management' section, there should be an option or button labeled 'View Polls'. Click on it.
- Click on **View on website** next to the specific poll title. This link should direct you to the `details` page for that particular poll.

What's Next

This chapter sets up the groundwork for displaying poll details, showcasing the question and associated answers in a stylized format. The subsequent chapter will introduce the voting functionality.

Chapter 12. Implementing Voting System

To enable the voting mechanism, modifications are made to the `PollAnswer` model, the addition of new controllers, and enhancements to the existing controllers and views.

Details Controller Modification

The controller modification in the 'details' action of `modules/preload/poll-models.php` has been enhanced to provide necessary variables for the template:

```
pb_on_action('details', function ()
    {
        // Existing code...
        pb_set_tpl_var('vote_url', PBURL::CurrentModule('vote', ['id' => $poll->ID()]));
        pb_set_tpl_var('error_message', PB::Errors()->GetErrorsAsString());
    });
```

`PB::Errors()->GetErrorsAsString()` retrieves and concatenates any set errors as a string, or returns an empty string if no errors have been set. This ensures that if any errors occur, they will be accessible in the template for display.

View Adjustment

The 'details' view is enhanced to handle errors and display the voting form:

```
{if $m.action eq "details"}
    {include file="block_begin.tpl"}

    {if $m.error_message neq ""}
        <div class="alert alert-danger">{$m.error_message}</div>
    {/if}

    <form method="post" action="{$m.vote_url}">
        <div class="poll-question">{$m.question}</div>

        {foreach from=$m.answers item=answer}
            <div class="poll-answer">
                <label>
                    <input type="radio" name="answer_id" value="{$answer.id}" />
                    {$answer.title}
                </label>
            </div>
        {/foreach}

        <input type="submit" value="Vote" />
    </form>

    {include file="block_end.tpl"}
{/if}
```

This template displays the poll question and answer options in a form allowing users to vote. If any error occurs during voting, it will be displayed above the form.

PollAnswer Model Update

The `Vote()` method is added to the `PollAnswer` model in the `modules/preload/poll-models.php` file:

```
class PollAnswer extends PBDBObject
    {
        // Existing code...

        public function Vote(): void
            {
                $this->Increment('votes');
            }
    }
```

This method increments the 'votes' field for the selected answer.

An alternative approach for achieving this could be through the `UpdateValues()` method, explicitly setting the 'votes' field:

```
public function Vote(): void
    {
        $this->UpdateValues([
            'votes'=>$this->VotesCount()+1,
        ]);
    }
```

However, using `Increment()` is a more efficient and direct way to handle this particular task. It specifically handles incrementing a numerical field, in this case, the 'votes' count, without the need for explicitly retrieving and setting the updated value.

New Controllers

Two new controllers are created in `modules/polls.php` to manage the voting process and display a 'Thank You' message upon successful voting.

The use block will look like this:

```
use PB\Common\Redirect;
use PB\Core\PBURL;
use PB\PB;
use PB\UI\Buttons;
use PB\UI\UI;
```

vote

The 'vote' controller manages the voting process:

```
pb_on_action('vote', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        if (!$poll->Exists())
            return;

        $selected_answer_id = PB::POST('answer_id')->AsInt();
        $answer = $poll->Answers()->GetItemWithID($selected_answer_id);
        // If the selected answer exists, increment its vote count and redirect to 'thank-you' contro
        if ($answer !== NULL)
            {
                $answer->Vote(); // Increment the vote count for the selected answer
                Redirect::Now(PBURL::CurrentModule('thank-you')); // Redirect to the 'thank-you' cont
            }
        else
```

```
            {
                // If no answer is selected, set an error message and switch to the 'details' control
                PB::Errors()->AddError('Please select an answer');
                pb_set_action('details');
            }
    });
```

In the absence of a selected answer, this controller will set an error message and switch the active action to 'details' to execute that controller after the current one finishes.

The order of declaring controllers is important. Ensure that 'vote' is declared before 'details' to guarantee proper execution of the 'details' controller in case of an error.

If no errors occur, the selected answer's vote count will increment, and the user will be redirected to the 'thank-you' controller within the current module.

We attempted to retrieve the selected answer ID using `PB::POST('answer_id')->AsInt()`. The method `PBDBList::GetItemWithID(<object-id>)` will return the object with the provided ID from preloaded items or `NULL` if not found.

Remember, the template file contains a radio item:

```
<input type="radio" name="answer_id" value="{$answer.id}" />
```

This radio input allows users to select an answer by its ID for voting purposes.

thank-you

This controller is responsible for displaying a 'Thank You' message after a successful vote. To streamline and expedite the development process, a UI component is utilized without the need for a specific template.

```
pb_on_action('thank-you', function ()
    {
        pb_title('Thank You');
        $ui = new UI();
        $ui->NotificationSuccess('You voted successfully!');
        $b = new Buttons();
        $b->Button('View polls', PBURL::CurrentModule('view'));
        $ui->Add($b);
        $ui->Output(true);
    });
```

Additional information about UI::Notification components:

1. `UI::NotificationSuccess(<message>)`: This method is used to display a success message or notification to the user.
2. `UI::NotificationError(<message>)` is used to display error messages or notifications to users.
3. `UI::NotificationWarning(<message>)` indicates situations where there might not be an error but warns users about potential issues or important information.
4. `UI::NotificationInfo(<message>)` doesn't signify an error or warning but offers additional information or guidance.

These notification methods can be used to convey different types of messages to users, allowing for clear and concise communication based on the nature of the information being presented.

Your instructions are clear and concise. However, I've made a few minor adjustments for clarity and completeness:

Test it Out

Now, you can test the functionality. Navigate to the poll, select an answer, and click the 'Vote' button.

If everything is successful, you should be automatically redirected to the 'Thank You' page.

To confirm that the voting process has worked, check the database to ensure that the `votes` field has been updated.

It's important to note that in this example, you can vote as many times as you want. While this is suitable for testing purposes, a real-world voting application should implement proper counting mechanisms. However, for the sake of simplicity, we won't cover this aspect in our example.

Chapter 13. Managing Answers in Admin Interface

Managing answers associated with polls is crucial for maintaining accurate data within the administrative section. Let's implement functionalities to add, edit, and delete answers for polls.

Extending the PollAnswer Model

We expand the `PollAnswer` model by introducing a method named `SetAnswerWithVotes`. This method allows the setting of a new answer title along with the votes count for a specific answer within the system.

```
class PollAnswer extends PBDBObject {
    // Existing code...
    public function SetAnswerWithVotes(string $new_answer_title, int $new_votes_count): void
        {
            $this->UpdateValues([
                'title' => $new_answer_title,
                'votes' => $new_votes_count,
            ]);
        }
}
```

This method facilitates the modification of an answer's title and votes count simultaneously, ensuring accurate updates within the database.

Controllers for Managing Answers

These controllers streamline the management of answers associated with polls in the administrative section, ensuring smooth CRUD operations for answers within the system.

answers

Displays a structured list of answers for a specific poll in an admin-friendly grid format. The structure is generally similar to the `view-polls` controller described earlier, but with the incorporation of new functionalities.

- `PB::Show404IfEmpty(<object>)` is utilized to display a 404 error page if the provided object is empty or not found. This ensures a smoother user experience when handling missing or erroneous data. It's crucial to note that if the 404 error page is executed due to the absence of the poll or answer, the subsequent code following `PB::Show404IfEmpty()` won't be executed.

- `PB::BreadCrumbs()` generates breadcrumbs to aid in navigation. In this context, it is used to create a trail allowing easy return to the main polls section.

- `PBURL::Current()` returns the current URL. It is used in `PBURL::CurrentModule()` for the `edit` and `delete` columns to provide a URL to return to after these actions.

```
pb_on_action('answers', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);

        PB::BreadCrumbs()
          ->Add('Polls', PBURL::CurrentModule('view-polls'))
          ->AddCurrent();

        pb_title('Answers - '.$poll->Title());
        $ui=new UI();

        $grid=new Grid();
        $grid->AddCol('index', '#');
        $grid->AddCol('answer', 'Answer');
        $grid->AddCol('votes', 'Votes');
        $grid->AddEdit();
        $grid->AddDelete();
        foreach ($poll->Answers()->EachItem() as $index=>$answer)
            {
                $grid->Label('index', $index+1);
                $grid->Label('answer', $answer->Title());
                $grid->Label('votes', $answer->VotesCount());
                $grid->URL('edit', PBURL::CurrentModule('edit-answer',
                    ['id'=>$poll->ID(), 'answer'=>$answer->ID()], PBURL::Current()));
                $grid->URL('delete', PBURL::CurrentModule('delete-answer',
                    ['id'=>$poll->ID(), 'answer'=>$answer->ID()], PBURL::Current()));
                $grid->NewRow();
            }
        if ($grid->RowCount()===0)
            $grid->SingleLineLabel('Nothing found');
        $ui->Add($grid);

        $b=new Buttons();
        $b->Button('Add Answer', PBURL::CurrentModule('add-answer',
            ['id'=>$poll->ID()], PBURL::Current()));
        $ui->Add($b);

        $ui->Output(true);
    });
```

This controller provides administrators with an organized view of answers associated with a specific poll, offering seamless navigation and functionality for effective management.

delete-answer

This controller handles the deletion of a specific answer associated with a poll.

```
pb_on_action('delete-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $answer=$poll->Answers()->GetItemWithID(PB::GET('answer')->AsInt());
        PB::Show404IfEmpty($answer);
        $answer->Remove();
        Redirect::ReturnToNow();
    });
```

The `Remove()` method effectively removes the associated row for the object within the database. However, it's essential to note that even after calling `Remove()`, the object in memory persists but cannot execute any database-related operations.

`Redirect::ReturnToNow()` triggers an immediate redirection using the `returnto` parameter present in the current GET request. If the returnto parameter is set to a specific URL, the user will be redirected there instantly.

You can set up the `returnto` parameter in the URL manually or utilize helper functions. Both `PBURL::Module()` and `PBURL::CurrentModule()` have a last parameter to set up the return URL, offering flexibility in managing redirection after actions.

The content looks mostly accurate. Just a couple of adjustments for clarity:

Create Answer Controllers

In this section, two controllers handle the addition of new answers to a poll. I't important to declare `create-answer` before `add-answer` to maintain a proper error handling.

create-answer

Responsible for creating a new answer for a specific poll.

```
pb_on_action('create-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        PB::Errors()->SwitchActionOnError('add-answer');
        if (PB::POST('answer')->isEmpty())
            PB::Errors()->AddError('Answer required');
        else
            {
                PollAnswer::Create($poll, PB::POST('answer')->EscapedString());
                Redirect::ReturnToNow();
            }
    });
```

In this controller, `PB::Errors()->SwitchActionOnError(<action>)` sets an alternative action in case an error occurs during the execution of a specific action. If `PB::Errors()->AddError()` is called or has already been called, it triggers a switch to the specified action (it will be executed upon the finishing of currrent controller).

add-answer

Displays a form to add a new answer to a poll.

```
pb_on_action('add-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        pb_title('Add Answer to Poll - '.$poll->Title());
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new Form(PBURL::CurrentWithAction('create-answer'));
        $f->AddText('answer', 'Answer', true)
          ->SetFocus();
        $f->LoadValuesArray(PB::Requests()->POSTAsArray());
        $ui->Add($f);
        $ui->Output(true);
    });
```

In this code, `PBURL::CurrentWithAction()` generates a URL with an updated action parameter based on the current URL query. It's a convenient method to pass parameters between multiple controllers when they share the same parameters except for the action.

Update Answer Controllers

These controllers efficiently handle the editing and updating of existing answers associated with polls, providing a straightforward user interface for such modifications.

update-answer

Manages the update process for an existing answer, similar to the approach taken in `create-answer` and `create-poll` .

Two fields, namely `answer` and `votes`, are used for validation and to update the respective properties simultaneously. While this approach might differ in a real-life application where more intricate data handling could be implemented, this demonstration illustrates how to update a rich model.

```
pb_on_action('update-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $answer=$poll->Answers()->GetItemWithID(PB::GET('answer')->AsInt());
        PB::Show404IfEmpty($answer);
        PB::Errors()->SwitchActionOnError('edit-answer');
        if (PB::POST('answer')->isEmpty())
            PB::Errors()->AddError('Answer required');
        elseif (PB::POST('votes')->AsInt()<0)
            PB::Errors()->AddError('Wrong votes count');
        else
            {
                $answer->SetAnswerWithVotes(
                    PB::POST('answer')->EscapedString(),
                    PB::POST('votes')->AsInt()
                );
                Redirect::ReturnToNow();
            }
    });
```

edit-answer

Renders a form for editing an existing answer.

```
pb_on_action('edit-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $answer=$poll->Answers()->GetItemWithID(PB::GET('answer')->AsInt());
        PB::Show404IfEmpty($answer);
        pb_title('Edit Answer');
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new Form(PBURL::CurrentWithAction('update-answer'));
        $f->AddText('answer', 'Answer', true)
          ->WithValue($answer->Title())
          ->SetFocus();
        $f->AddText('votes', 'Votes Count', true)
          ->WithValue($answer->VotesCount());
        $f->LoadValuesArray(PB::Requests()->POSTAsArray());
        $ui->Add($f);
        $ui->Output(true);
    });
```

In the code, `Form::WithValue()` is used to pre-fill the form fields with the existing data when editing an answer. Additionally, the `LoadValuesArray()` method is used to potentially overwrite pre-filled values if an error occurred and the POST data contains information sent with the form. Ensure that the `LoadValuesArray()` method is called after all

`WithValue()` methods to avoid overwriting pre-filled values incorrectly.

What's Next

We are almost done with polls management. Let's make final adjustments and review.

Chapter 14. Final Steps in Polls Management Module

For now, we have almost completed everything. There's a small adjustment needed - the ability to update the poll question and delete the poll. So, we should add some new controllers to the `modules/polls-admin.php` file and make slight modifications to the poll list controller.

Update Poll Controllers

Modify Poll List

Let's update the poll list by incorporating URLs for the `edit` and `delete` columns. Also, here's an example demonstrating how to configure a custom deletion confirmation for individual rows using `CustomMessageBox(<prompt>)`. Additionally, we'll place a block for adding a new poll below the table and a paginator for user convenience.

```
pb_on_action('view-polls', function ()
    {
        // Existing code...
        foreach ($polls->EachItem() as $poll)
            {
                // Existing code...
                $grid->URL('edit', PBURL::CurrentModule('edit-poll',
                    ['id'=>$poll->ID()], PBURL::Current()));
                $grid->URL('delete', PBURL::CurrentModule('delete-poll',
                    ['id'=>$poll->ID()], PBURL::Current()));
                $grid->CustomMessageBox('delete', 'Are you sure to delete this poll?');
                $grid->NewRow();
            }
        // Existing code...
        $b=new Buttons();
        $b->Button('Add Poll', PBURL::CurrentModule('add-poll', [], PBURL::Current()));
        $ui->Add($b);

        $ui->Output(true);
    });
```

edit-poll controller

The `edit-poll` controller utilizes a `POSTForm`, which functions similarly to a regular `Form` component but is specifically designed for POST requests. Notably, it automatically loads current POST values into the form before invoking the UI, similar to the earlier manual loading with `$f->LoadValuesArray(PB::Requests()->POSTAsArray());`.

```
pb_on_action('edit-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        pb_title('Edit Poll');
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new POSTForm(PBURL::CurrentWithAction('update-poll'));
        $f->AddText('question', 'Question', true)
```

```
            ->WithValue($poll->Title())
            ->SetFocus();
          $ui->Add($f);
          $ui->Output(true);
      });
```

update-poll controller

The `update-poll` controller operates similarly to previous examples. However, as the 'question' field is designated as a `Title` for the `Poll` model, we utilize the default `SetTitle(<new-value>)` method to update it.

```
pb_on_action('update-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        PB::Errors()->SwitchActionOnError('edit-poll');
        if (PB::POST('question')->isEmpty())
            PB::Errors()->AddError('Question required');
        else
            {
                $poll->SetTitle(PB::POST('question')->EscapedString());
                Redirect::ReturnToNow();
            }
    });
```

Delete the Poll

At this point, we need to delete the poll, and the controller for it is quite similar to what we've done before.

```
pb_on_action('delete-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $poll->Remove();
        Redirect::ReturnToNow();
    });
```

What we need to do is modify the `Poll` model to handle the proper deletion process. The records in the `poll_answers` table are linked to this poll, so we need to delete them before removing the record from the `polls` table.

There are two ways to achieve this. One approach is to configure the database tables for cascade deletion, allowing the RDBMS to manage deletion automatically.

However, this may not always be the best solution, especially when additional actions are necessary, such as removing associated files or making external API calls.

Therefore, we'll handle it programmatically within our `Poll` model by declaring the `OnRemoveBeforeStart()` method. This method is automatically invoked before the actual deletion of a record for a `Poll` model.

```
class Poll extends PBDBObject
    {
        // Existing code...
        protected function OnRemoveBeforeStart(): void
            {
                foreach ($this->Answers()->EachItem() as $answer)
                    $answer->Remove();
            }
    }
```

This code ensures that all associated answers are deleted before removing the poll record.

Final Adjustments to the Code

To enhance the future edit flexibility, we'll replace all `Form` elements in our example with `POSTForm`.

Also, let's add some more breadcrumbs for better user experience.

Additionally, we'll update the redirection in the `create-poll` controller using `Redirect::NowReturnToOr(<default-url>)`. This method redirects to the provided `returnto` URL or the default URL if no redirection URL is provided.

```
Redirect::NowReturnToOr(PBURL::CurrentModule('admin'));
```

We also want to restrict unauthorized users from managing polls. To accomplish this, we'll add the following block right after the `use` section in the `modules/polls-admin.php` file:

```
PBAccess::EnforceLogin();
```

This function redirects visitors to the sign-in form if they are not logged in.

What Could Be Done?

We could further enhance security by verifying the user associated with the poll using `PBAccess::UserRequired`. This function redirects anyone other than the specified user (and admin) to the 'Access Denied' page.

```
pb_on_action('edit-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        PBAccess::UserRequired($poll->UserID());
        //...
    });
```

However, for the sake of simplicity, we won't implement this here.

What Should Be Done?

If you wish for regular users to create and manage their polls, consider placing a link `index.php?m=polls-admin&d=view-polls` in the navigation menu for logged-in users.

What's Next

Please review the finalized version of `modules/polls-admin.php`.

Chapter 15. Finalized Polls Administration Module

```
<?php

/*
 * Module Name: Polls Management
 */

use PB\Access\PBAccess;
use PB\Common\Redirect;
use PB\Core\PBURL;
use PB\PB;
use PB\UI\Buttons;
use PB\UI\Grid;
use PB\UI\POSTForm;
use PB\UI\UI;

PBAccess::EnforceLogin();

pb_on_action('delete-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $answer=$poll->Answers()->GetItemWithID(PB::GET('answer')->AsInt());
        PB::Show404IfEmpty($answer);
        $answer->Remove();
        Redirect::ReturnToNow();
    });

pb_on_action('create-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        PB::Errors()->SwitchActionOnError('add-answer');
        if (PB::POST('answer')->isEmpty())
            PB::Errors()->AddError('Answer required');
        else
            {
                PollAnswer::Create($poll, PB::POST('answer')->EscapedString());
                Redirect::ReturnToNow();
            }
    });

pb_on_action('add-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        PB::BreadCrumbs()
          ->Add('Polls', PBURL::CurrentModule('view-polls'))
          ->Add($poll->Title(), PBURL::CurrentModule('answers', ['id'=>$poll->ID()]))
          ->AddCurrent();
        pb_title('Add Answer to Poll - '.$poll->Title());
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new POSTForm(PBURL::CurrentWithAction('create-answer'));
        $f->AddText('answer', 'Answer', true)
          ->SetFocus();
        $ui->Add($f);
        $ui->Output(true);
    });

pb_on_action('update-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $answer=$poll->Answers()->GetItemWithID(PB::GET('answer')->AsInt());
        PB::Show404IfEmpty($answer);
        PB::Errors()->SwitchActionOnError('edit-answer');
        if (PB::POST('answer')->isEmpty())
            PB::Errors()->AddError('Answer required');
        elseif (PB::POST('votes')->AsInt()<0)
            PB::Errors()->AddError('Wrong votes count');
        else
            {
                $answer->SetAnswerWithVotes(
                    PB::POST('answer')->EscapedString(),
                    PB::POST('votes')->AsInt()
                );
                Redirect::ReturnToNow();
            }
    });

pb_on_action('edit-answer', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $answer=$poll->Answers()->GetItemWithID(PB::GET('answer')->AsInt());
        PB::Show404IfEmpty($answer);
        PB::BreadCrumbs()
          ->Add('Polls', PBURL::CurrentModule('view-polls'))
          ->Add($poll->Title(), PBURL::CurrentModule('answers', ['id'=>$poll->ID()]))
          ->AddCurrent();
        pb_title('Edit Answer');
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new POSTForm(PBURL::CurrentWithAction('update-answer'));
        $f->AddText('answer', 'Answer', true)
          ->WithValue($answer->Title())
          ->SetFocus();
        $f->AddText('votes', 'Votes Count', true)
          ->WithValue($answer->VotesCount());
        $f->LoadValuesArray(PB::Requests()->POSTAsArray());
        $ui->Add($f);
        $ui->Output(true);
    });

pb_on_action('answers', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);

        PB::BreadCrumbs()
          ->Add('Polls', PBURL::CurrentModule('view-polls'))
          ->AddCurrent();

        pb_title('Answers - '.$poll->Title());
        $ui=new UI();

        $grid=new Grid();
        $grid->AddCol('index', '#');
        $grid->AddCol('answer', 'Answer');
        $grid->AddCol('votes', 'Votes');
        $grid->AddEdit();
        $grid->AddDelete();
        foreach ($poll->Answers()->EachItem() as $index=>$answer)
            {
                $grid->Label('index', $index+1);
                $grid->Label('answer', $answer->Title());
                $grid->Label('votes', $answer->VotesCount());
                $grid->URL('edit', PBURL::CurrentModule('edit-answer',
                    ['id'=>$poll->ID(), 'answer'=>$answer->ID()], PBURL::Current()));
                $grid->URL('delete', PBURL::CurrentModule('delete-answer',
                    ['id'=>$poll->ID(), 'answer'=>$answer->ID()], PBURL::Current()));
                $grid->NewRow();
            }
        if ($grid->RowCount()===0)
            $grid->SingleLineLabel('Nothing found');
        $ui->Add($grid);

        $b=new Buttons();
        $b->Button('Add Answer', PBURL::CurrentModule('add-answer',
            ['id'=>$poll->ID()], PBURL::Current()));
        $ui->Add($b);

        $ui->Output(true);
    });

pb_on_action('delete-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        $poll->Remove();
        Redirect::ReturnToNow();
    });

pb_on_action('update-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        PB::Errors()->SwitchActionOnError('edit-poll');
        if (PB::POST('question')->isEmpty())
            PB::Errors()->AddError('Question required');
        else
            {
                $poll->SetTitle(PB::POST('question')->EscapedString());
                Redirect::ReturnToNow();
            }
    });

pb_on_action('edit-poll', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);
        pb_title('Edit Poll');
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new POSTForm(PBURL::CurrentWithAction('update-poll'));
        $f->AddText('question', 'Question', true)
          ->WithValue($poll->Title())
          ->SetFocus();
        $ui->Add($f);
        $ui->Output(true);
    });

pb_on_action('create-poll', function ()
    {
        if (PB::POST('question')->isEmpty())
            PB::Errors()->AddError('Question required');
        if (PB::POST('answer1')->isEmpty())
            PB::Errors()->AddError('First answer required');
        if (PB::POST('answer2')->isEmpty())
            PB::Errors()->AddError('Second answer required');
        if (!PB::Errors()->HasErrors())
            {
                $poll=Poll::Create(PB::POST('question')->EscapedString(), PB::User()->ID());
                PollAnswer::Create($poll, PB::POST('answer1')->EscapedString());
                PollAnswer::Create($poll, PB::POST('answer2')->EscapedString());
                if (!PB::POST('answer3')->isEmpty())
                    PollAnswer::Create($poll, PB::POST('answer3')->EscapedString());
                pb_notify('Poll created!');
                Redirect::NowReturnToOr(PBURL::CurrentModule('admin'));
            }
        else
            pb_set_action('add-poll');
    });

pb_on_action('add-poll', function ()
    {
        pb_title('Add Poll');
        $ui=new UI();
        PB::Errors()->DisplayUIErrors($ui);
        $f=new POSTForm(PBURL::CurrentModule('create-poll'));
        $f->AddText('question', 'Poll Question', true)
            ->SetFocus();
        $f->AddText('answer1', 'Answer 1', true);
        $f->AddText('answer2', 'Answer 2', true);
        $f->AddText('answer3', 'Answer 3');
        $ui->Add($f);
        $ui->Output(true);
    });

pb_on_action('view-polls', function ()
    {
        pb_title('Polls');
        $ui=new UI();
        $grid=new Grid();

        // Column Additions and Setup
        $grid->AddCol('title', 'Title');
        $grid->AddCol('view', 'View on website');
        $grid->AddEdit();
        $grid->AddDelete();

        $polls=new PollList();
        $polls->FilterUser(PB::User()->ID());
        $polls->Limit(10);
        $polls->Offset(PB::GET('from')->AsAbsInt());
        $polls->OrderByID(false);
        $polls->Load();

        foreach ($polls->EachItem() as $poll)
            {
                // Populate Grid with Poll Data
                $grid->Label('title', $poll->Title());
                $grid->URL('title', PBURL::CurrentModule('answers', ['id'=>$poll->ID()]));
                $grid->Label('view', 'View on website');
                $grid->URL('view', PBURL::Module('polls', 'details',
                    ['id'=>$poll->ID()]), true);
                $grid->URL('edit', PBURL::CurrentModule('edit-poll',
                    ['id'=>$poll->ID()], PBURL::Current()));
                $grid->URL('delete', PBURL::CurrentModule('delete-poll',
                    ['id'=>$poll->ID()], PBURL::Current()));
                $grid->CustomMessageBox('delete', 'Are you sure to delete this poll?');
                $grid->NewRow();
            }

        if ($grid->RowCount()===0)
            $grid->SingleLineLabel('Nothing found');

        $ui->Add($grid);
        $ui->AddPaginatorForList($polls);

        $b=new Buttons();
        $b->Button('Add Poll', PBURL::CurrentModule('add-poll', [], PBURL::Current()));
        $ui->Add($b);

        $ui->Output(true);
    });

pb_on_action('admin', function ()
    {
        pb_title('Polls Management');
        $ui=new UI();
        $b=new Buttons();
        $b->Button('Add Poll', PBURL::CurrentModule('add-poll'));
        $b->Button('View Polls', PBURL::CurrentModule('view-polls'));
        $ui->Add($b);
        $ui->Output(true);
    });

pb_on_action('install', function ()
    {
        PBAccess::AdminRequired();
        pb_register_module(pb_current_module(), 'Polls Management');
        pb_register_autoload('poll-models');
        execsql("CREATE TABLE `polls` (
            `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
            `title` varchar(255) NOT NULL,
            `user_id` int(11) unsigned NOT NULL DEFAULT '0',
             PRIMARY KEY (`id`) ) ENGINE=MyISAM DEFAULT CHARSET=utf8;");
        execsql("CREATE TABLE `poll_answers` (
            `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
            `poll_id` int(11) unsigned NOT NULL DEFAULT '0',
            `title` varchar(255) NOT NULL, `votes` int(11) unsigned NOT NULL DEFAULT '0',
            PRIMARY KEY (`id`), KEY `poll_id` (`poll_id`,`title`) )
            ENGINE=MyISAM DEFAULT CHARSET=utf8;");
        Redirect::Now(PBURL::CurrentModule('admin'));
    });

pb_on_action('uninstall', function ()
    {
        PBAccess::AdminRequired();
        pb_unregister_module(pb_current_module());
        pb_unregister_autoload('poll-models');
        execsql("DROP TABLE `poll_answers`;");
        execsql("DROP TABLE `polls`;");
        Redirect::Now(PBURL::AdminModulesManagement());
    });
```

Chapter 16. Polls Index for Visitors

Presently, we've established the functions for managing polls. It's time to create an index of polls for visitors, granting them access to the latest polls available for voting. Additionally, let's incorporate poll results on the 'thank you' page.

Polls Index

Controller Update

To enhance the visitor-facing controller for the 'view' action in the `modules/polls.php` file, we'll transition from a basic 'Hello World' example to real functionality.

```
pb_on_action('view', function ()
    {
        pb_title('Polls');
        pb_template('polls');

        $polls = new PollList();
        $polls->DisableNoFiltersBlocker();
        $polls->Limit(10);
        $polls->Offset(PB::GET('from')->AsAbsInt());
        $polls->OrderByID(false);
        $polls->Load();

        $polls_data = [];

        foreach ($polls->EachItem() as $poll)
            {
                $polls_data[] = [
                    'title' => $poll->Title(),
                    'vote_url' => PBURL::CurrentModule('details', ['id' => $poll->ID()]),
                ];
            }

        pb_set_tpl_var('polls', $polls_data);

        pb_pagination_init_for_list($polls);
    });
```

Loading the list data is the initial step.

```
$polls = new PollList();
$polls->DisableNoFiltersBlocker();
$polls->Limit(10);
$polls->Offset(PB::GET('from')->AsAbsInt());
$polls->OrderByID(false);
$polls->Load();
```

This process resembles what was done in the admin section. However, since there are no filters required here, we used the `DisableNoFiltersBlocker()` method. The `PBDBList` includes a built-in blocker to prevent overloading the database with queries lacking filters.

Then, we assign the poll data for the view and initialize the pagination with the `pb_pagination_init_for_list` helper.

```
pb_pagination_init_for_list($polls);
```

To initialize pagination manually, we could use `pb_pagination_init(<total-count>, <current-limit>, <current-offset>, <custom-url>)`.

View Update

The view in `themes/default/polls.tpl` also requires an update.

```
{if $m.action eq "view"}
    {include file="block_begin.tpl"}

    <ul class="polls-list-container">
        {foreach from=$m.polls item=poll}
            <li><a href="{$poll.vote_url}">{$poll.title}</a></li>
        {/foreach}
    </ul>

    {include file="paginator.tpl"}

    {include file="block_end.tpl"}
{/if}
```

This code shares similarities with what we've previously built. However, we need to integrate a paginator here. We'll include a pagination block using the following code.

```
{include file="paginator.tpl"}
```

Thank You with Vote Results

We're bypassing the use of `$poll->Answers()` to load the answers because we require a custom order. Loading answers mirrors what we did with polls, but we won't be implementing pagination here since there won't be a large volume of answers.

```
pb_on_action('thank-you', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);

        pb_title('Thank You');
        pb_template('polls');

        $poll_answers = new PollAnswersList();
        $poll_answers->FilterPoll($poll->ID());
        $poll_answers->OrderByVotes(false);
        $poll_answers->Load();

        $answers_data=[];
        foreach ($poll_answers->EachItem() as $answer)
            {
                $answers_data[]=[
                    'title'=>$answer->Title(),
                    'votes'=>$answer->VotesCount(),
                ];
            }
        pb_set_tpl_var('answers', $answers_data);

        pb_set_tpl_var('polls_url', PBURL::CurrentModule('view'));
    });
```

Let's incorporate a 'thank-you' view in the template to exhibit an ordered list of answers along with their respective vote counts:

```
{if $m.action eq "thank-you"}
    {include file="block_begin.tpl"}

    <div class="alert alert-success">
        You voted successfully!
    </div>

    <ul class="polls-list-container">
        {foreach from=$m.answers item=answer}
            <li>{$answer.title} - <strong>{$answer.votes} votes</strong></li>
        {/foreach}
    </ul>

    <div class="text-center">
        <a class="btn btn-primary" href="{$m.polls_url}">View Polls</a>
    </div>

    {include file="block_end.tpl"}
{/if}
```

What's Next

Please review the finalized version of visitor facing section.

Chapter 17. Finalizing Visitor-Facing Functionality

Now, we'll culminate the visitor-oriented functionalities by refining the controller and views for the polls section.

modules/polls.php

```
<?php

use PB\Common\Redirect;
use PB\Core\PBURL;
use PB\PB;

pb_on_action('thank-you', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);

        pb_title('Thank You');
        pb_template('polls');

        $poll_answers = new PollAnswersList();
        $poll_answers->FilterPoll($poll->ID());
        $poll_answers->OrderByVotes(false);
        $poll_answers->Load();

        $answers_data=[];
        foreach ($poll_answers->EachItem() as $answer)
            {
                $answers_data[]=[
                    'title'=>$answer->Title(),
                    'votes'=>$answer->VotesCount(),
                ];
            }
        pb_set_tpl_var('answers', $answers_data);

        pb_set_tpl_var('polls_url', PBURL::CurrentModule('view'));
    });


pb_on_action('vote', function ()
    {
        $poll = new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);

        $selected_answer_id = PB::POST('answer_id')->AsInt();
        $answer = $poll->Answers()->GetItemWithID($selected_answer_id);
        if ($answer !== NULL)
            {
                $answer->Vote();
                Redirect::Now(PBURL::CurrentModule('thank-you', ['id'=>$poll->ID()]));
            }
        else
            {
                PB::Errors()->AddError('Please select an answer');
                pb_set_action('details');
            }
    });


pb_on_action('details', function ()
    {
        $poll=new Poll(PB::GET('id')->AsInt());
        PB::Show404IfEmpty($poll);

        pb_title('Vote!');
        pb_add_cssfile('css/polls.css');

        pb_template('polls');

        pb_set_tpl_var('question', $poll->Title());

        $answers=[];
        foreach ($poll->Answers()->EachItem() as $poll_answer)
            {
                $answers[]=[
                    'id'=>$poll_answer->ID(),
                    'title'=>$poll_answer->Title(),
                ];
            }
        pb_set_tpl_var('answers', $answers);

        pb_set_tpl_var('vote_url', PBURL::CurrentModule('vote', ['id'=>$poll->ID()]));
        pb_set_tpl_var('error_message', PB::Errors()->GetErrorsAsString());

    });


pb_on_action('view', function ()
    {
        pb_title('Polls');
        pb_template('polls');

        $polls=new PollList();
        $polls->DisableNoFiltersBlocker();
        $polls->Limit(10);
        $polls->Offset(PB::GET('from')->AsAbsInt());
        $polls->OrderByID(false);
        $polls->Load();

        $polls_data=[];

        foreach ($polls->EachItem() as $poll)
            {
                $polls_data[]=[
                    'title'=>$poll->Title(),
                    'vote_url'=>PBURL::CurrentModule('details', ['id'=>$poll->ID()]),
                ];
            }

        pb_set_tpl_var('polls', $polls_data);

        pb_pagination_init_for_list($polls);

    });
```

themes/default/polls.tpl

```
{if $m.action eq "view"}
    {include file="block_begin.tpl"}

    <ul class="polls-list-container">
        {foreach from=$m.polls item=poll}
            <li><a href="{$poll.vote_url}">{$poll.title}</a></li>
        {/foreach}
    </ul>

    {include file="paginator.tpl"}

    {include file="block_end.tpl"}
{/if}


{if $m.action eq "details"}
    {include file="block_begin.tpl"}

    {if $m.error_message neq ""}
        <div class="alert alert-danger">{$m.error_message}</div>
    {/if}

    <form method="post" action="{$m.vote_url}">
        <div class="poll-question">{$m.question}</div>

        {foreach from=$m.answers item=answer}
            <div class="poll-answer">
                <label>
                    <input type="radio" name="answer_id" value="{$answer.id}" />
                    {$answer.title}
                </label>
            </div>
        {/foreach}

        <input type="submit" value="Vote" />
    </form>

    {include file="block_end.tpl"}
{/if}


{if $m.action eq "thank-you"}
    {include file="block_begin.tpl"}

    <div class="alert alert-success">
        You voted successfully!
    </div>

    <ul class="polls-list-container">
        {foreach from=$m.answers item=answer}
            <li>{$answer.title} - <strong>{$answer.votes} votes</strong></li>
        {/foreach}
    </ul>

    <div class="text-center">
        <a class="btn btn-primary" href="{$m.polls_url}">View Polls</a>
    </div>

    {include file="paginator.tpl"}
    {include file="block_end.tpl"}
{/if}
```

Chapter 18. Completing Poll Models Implementation

In this chapter, we'll review and integrate the finalized version of the `modules/preload/poll-models.php` file, which encapsulates the data models and their functionalities essential for managing polls and associated answers within the application.

```
<?php

    use PB\DB\ORM\PBDBList;
    use PB\DB\ORM\PBDBObject;

    /**
     * @method static self initSingleton($id)
     * @method static self UsingCache($id)
     * @method static self initNotExistent()
     */
    class Poll extends PBDBObject
        {

            public static function TableName(): string
                {
                    return 'polls';
                }

            public static function FieldNameForID(): string
                {
                    return 'id';
                }

            public static function FieldNameForTitle(): string
                {
                    return 'title';
                }

            public function UserID(): int
                {
                    return $this->FieldIntValue('user_id');
                }

            public static function Create(string $poll_question, int $created_by_user_id): self
                {
                    return self::CreateWithParams([
                        'title'=>$poll_question,
                        'user_id'=>$created_by_user_id,
                    ]);
                }

            public function Answers(): PollAnswersList
                {
                    $poll_answers = new PollAnswersList();
                    $poll_answers->FilterPoll($this->ID());
                    $poll_answers->OrderByID();
                    $poll_answers->Load();
                    return $poll_answers;
                }

            protected function OnRemoveBeforeStart(): void
                {
                    foreach ($this->Answers()->EachItem() as $answer)
                        $answer->Remove();
                }

        }

    /**
     * @method static self initSingleton($id)
```

```
 * @method static self UsingCache($id)
 * @method static self initNotExistent()
 */
class PollAnswer extends PBDBObject
    {

        public static function TableName(): string
            {
                return 'poll_answers';
            }

        public static function FieldNameForID(): string
            {
                return 'id';
            }

        public static function FieldNameForTitle(): string
            {
                return 'title';
            }

        public function PollID(): int
            {
                return $this->FieldIntValue('poll_id');
            }

        public function VotesCount(): int
            {
                return $this->FieldIntValue('votes');
            }

        public static function Create(Poll $poll, string $answer_title): self
            {
                return self::CreateWithParams([
                    'title'=>$answer_title,
                    'poll_id'=>$poll->ID(),
                    'votes'=>0,
                ]);
            }

        public function Vote(): void
            {
                $this->Increment('votes');
            }

        public function SetAnswerWithVotes(string $new_answer_title, int $new_votes_count): void
            {
                $this->UpdateValues([
                    'title'=>$new_answer_title,
                    'votes'=>$new_votes_count,
                ]);
            }

    }

/**
 * @method Poll Item($index)
 * @method Poll[] EachItem()
 * @method Poll|NULL Fetch()
 * @method Poll|NULL FirstItem()
 * @method Poll|NULL LastItem()
 * @method Poll|NULL GetItemWithID($id)
 */
class PollList extends PBDBList
    {

        protected function ObjectClassName(): string
            {
                return Poll::class;
            }

        public function FilterUser(int $user_id): void
            {
                $this->SetFilterFieldIntValue('user_id', $user_id);
            }

    }

/**
```

```
 * @method PollAnswer Item($index)
 * @method PollAnswer[] EachItem()
 * @method PollAnswer|NULL Fetch()
 * @method PollAnswer|NULL FirstItem()
 * @method PollAnswer|NULL LastItem()
 * @method PollAnswer|NULL GetItemWithID($id)
 */
class PollAnswersList extends PBDBList
    {

        protected function ObjectClassName(): string
            {
                return PollAnswer::class;
            }

        public function FilterPoll(int $poll_id): void
            {
                $this->SetFilterFieldIntValue('poll_id', $poll_id);
            }

        public function OrderByVotes(bool $asc=true): void
            {
                $this->OrderByField('votes', $asc);
            }

    }
```

Chapter 19. Embracing Easy Module Building in PushButtonCMS

Creating modules within PushButtonCMS has been showcased through this basic example, highlighting the simplicity and efficiency of its development framework. This system is specifically crafted for rapid application development, ensuring a minimal learning curve for project initiation.

PushButtonCMS offers a low entry threshold, allowing developers to swiftly kickstart their projects without diving deep into complex concepts. The example presented here, though basic, covers a wide array of functionalities crucial to understanding the system's fundamentals.

The straightforward structure and intuitive design of PushButtonCMS streamline the process of building modules. This simplicity doesn't compromise the system's capability to handle intricate functionalities. It's a testament to how a simple yet powerful framework can empower developers to create robust applications with ease.

By providing a clear path for module creation and leveraging its inherent flexibility, PushButtonCMS encourages developers to focus on crafting efficient solutions rather than grappling with convoluted technicalities. This ease of use fosters a conducive environment for rapid prototyping and deployment, making it an ideal choice for various projects, whether small-scale or larger in scope.

PushButtonCMS stands as a testament to the philosophy that simplicity doesn't equate to limitations; instead, it promotes agility, creativity, and efficient development, empowering developers to bring their ideas to life swiftly and effectively.