CSC1106: Report

Group 7

2200692 Pang Zi Jian Adrian

2200795 Ashley Tay Yong Jun

2200959 Peter Febrianto Afandy

2201018 Jeffrey Yap Wan Lin

2201052 Nur Hakeem Bin Azman

2201159 Ryan Lai Wei Shao

—

24 July 2023

Table of Contents

Table of Figures

Introduction

This project is aimed at developing a CodeIgniter 4 (CI4) library that enables programmers to create customisable forms for use within their web applications and store form responses. The library has been tested against the United States' Form 1040 Individual Income Tax Return form to showcase its features. Two project deliverables have been completed for this project, namely a bare-bones web application project for developers to build on to utilise the form builder library, as well as a user-friendly administrator dashboard web application that is built on the library to showcase its features. The administrative portal also serves as a reference for developers on how to utilise the library's various features.

System Overview

Custom Library Overview

The custom form builder library consists of two main components, each playing a crucial role in enabling the creation and utilization of customizable forms within generic web applications. These components include a custom CI4 library and a MySQL relational database. Figure 1 shows a high-level overview of the library's design, illustrating the interactions between the CI4 library, the MySQL database, and their integration into the web application environment. This system design empowers developers to create dynamic and user-friendly forms, harnessing the full potential of the custom library. To leverage the functionalities of the custom library, programmers are required to set up both components within their web application environment.

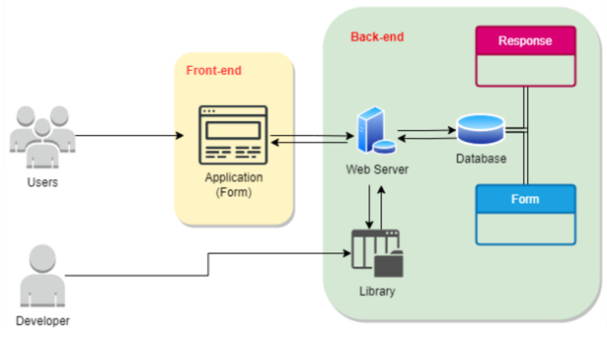


Figure 1: High-Level System Architecture

The custom CI4 library serves as the backbone of our project, offering a comprehensive collection of reusable classes and functions specifically designed for the creation, customization, and handling of forms. With this library, programmers gain the ability to finely tailor the forms to match their web application's unique requirements, thus enhancing user experience and engagement.

To store the structures and responses received for the generated forms, we employed a MySQL relational database. The decision to utilize MySQL is driven by its exceptional performance, scalability, maturity, reliability, and robust features, ensuring efficient data management for the forms.

Database Design

The relational database consists of two tables, Form and Response. The Entity Relational Diagram for the database is provided in Figure 2 below.

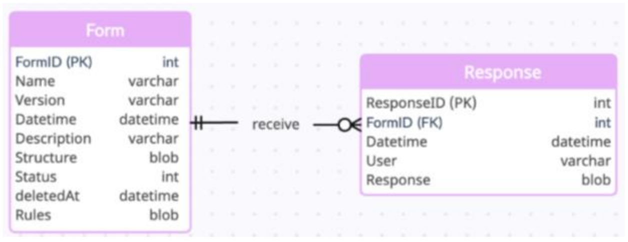


Figure 2: Relational Database Schema

The Form table is used to store the serialized structure of forms, along with their date of creation, version number for auditing, and custom serialised rules for the form template. The Response table is used to store serialized responses provided by end users of these forms.

Programmers are expected to maintain user details independently since such information will not be stored by the library. The Response table includes a 'User' field to allow programmers to identify users who have submitted a form response. This intentional separation of information ensures that the library remains lightweight while providing programmers with the flexibility to define their own database design to complement the library. Additionally, migration files are included for each table to facilitate fast deployment of the custom library.

Library Design

The library consists of the following components:

Name	Full File path	Description
Form Templates	app/Config/FormTemplates	Directory for developers to specify their custom forms, according to the library's custom format ($fields and $Rules
CustomFormLibrary	app/Libraries/CustomFormLibrary.php	File that contains all library functions and specifications.
Services	app/Config/Services.php	File used to obtain and load a shared instance of the custom library class. This enables developers to easily use and invoke the library's functions throughout their project.
FormModel	app/Models/FormModel.php	File that contains all functions and specifications for the Form database table used by the library to store developer-defined forms.
FormResponseModel	app/Models/FormResponseModel.php	File that contains all functions and specifications for the FormResponse database table used by the library to store user responses received for developer-defined forms.
FormBuilder	app/Database/Migrations/FormBuilder.php	File that contains specifications to automatically create the database tables needed by the library upon CI4 migration.
validation_helper	app/Helpers/validation_helper.php	File that contains custom functions used by the library for form validations and security (ie encryption).
Autoload	app/Config/Autoload.php	File used by the library to automatically load the validation_helper functions upon invocation.
Filters	app/Config/Filters.php	File used by the library for setting and enforcing default security features, including the use of CI4's Shield package for authentication and protection of all routes, and requiring CSRF protection for all POST requests.
Auth	app/Config/Auth.php	File used by CI4's Shield authentication package that is supported by the library.
Email	app/Config/Email.php	File used by the library to send emails to end users. Developers can utilize the preconfigured options for ease of development.
Routes	app/Config/Routes.php	File used by library for the setting of custom routes (mainly for CI4 Shield package).

Key Features and Functionalities

Integration with CI4's Form Helper

The custom library created references the CodeIgniter Form Helper function to replicate some of their key form creation functions, including the generation of various HTML tags. This similarity with the CodeIgniter Form Helper allows developers to leverage on familiar syntax and functionalities while utilizing the custom library for form creation.

Custom HTML Form Tag Creation

The library provides a suite of functions for developers to create customizable HTML tags dynamically, such as attributes_creator() and new_radio(). The primary objective of these form tag creation functions is to provide developers with flexibility and simplicity in utilizing the library to customize and create unique forms. By utilizing these functions, developers can easily generate form elements such as input fields, dropdowns, checkboxes, and more, with the ability to define attributes, labels, validation rules, and styling options.

Integration with Bootstrap 5

Furthermore, the library provides seamless integration with Bootstrap 5, a popular CSS framework, enabling developers to leverage its powerful grid system and responsive design capabilities. By incorporating Bootstrap classes into the form creation functions, the library allows developers to easily define rows and columns within their forms and utilize its design capabilities. Bootstrap utilizes a grid system based on a 12-column layout. With the custom library, developers can specify the desired column layout for their form elements using Bootstrap classes such as "col-4" or "col-md-6". This ensures that the form elements are aligned and displayed responsively across different screen sizes.

In addition to defining columns, the library also supports Bootstrap classes for various form components, such as inputs, labels, buttons, and more. Developers can utilize classes like "form-control" for input fields, "form-check" for checkboxes and radio buttons, ensuring consistent styling and compatibility with Bootstrap's predefined styles. By leveraging Bootstrap classes, developers can easily create visually appealing and mobile-friendly forms without the need for extensive CSS customization. The integration of Bootstrap's grid system and CSS classes within the custom library simplifies the process of designing and structuring forms, ensuring a seamless and consistent user experience across different devices and screen sizes.

CRUD Functions

The library supports CRUD (Create, Read, Update, Delete) functionalities primarily for managing developers' form templates and form responses received from end users. It provides the ability to create, retrieve, update, and delete forms templates and responses based on developers' needs. Additionally, the library includes a central repository that enables users to manage and archive form templates, organizing them into versions as required for auditing purposes. This versioning capability allows users to track and preserve different iterations of form templates over time.

To facilitate data storage and management, the library integrates with a MySQL database. This integration ensures that form data and templates are stored persistently, allowing developers to access and retrieve them whenever needed. The library also supports the soft deletion of form templates, using the Form table's Status and deletedAt columns for data persistence. The SQL database serves as a reliable and secure repository for preserving form-related information.

Utility Functions

In addition to the form creation functions, the library also supports utility functions to further enhance its extensibility and usefulness for developers. These utility functions expand the capabilities of the library, providing developers with additional options for managing and sharing their forms. The two key utility functions provided are shown in the table in the next page.

Function	Parameters	Description
export_to_pdf	($formData)	Function to export a given form into PDF, allowing for the printing of any HTML elements. This function is supported by the wkhtmltoPdf binary which supports Bootstrap CSS styling. This function returns a PDF string to allow users to output the exported form based on web app needs, e.g with <iframe>.
getFormHTML	($filename)	Function to generate a HTML dump of a form specified according to the library's form template configuration. It allows users to generate the form from their predefined PHP form template file, by reading the file contents and returning the dynamically created HTML form.

Process of Form Template Creation

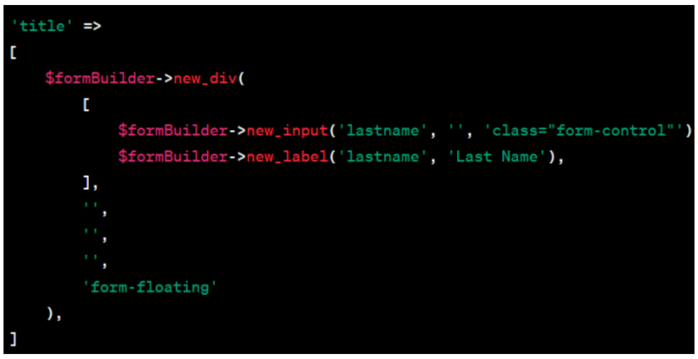
Developers can follow the following steps to create a custom form template in their CodeIgniter web application:

1. **Creating the Form Template File:**
   Developers need to create a PHP file with a .php extension containing the form template structure. We recommend placing this file in the 'App/Config/' directory under a folder named 'FormsTemplate' for easy reference. Developers can take reference from the sample forms provided by the library. Developers can specify the structure of their forms by defining the $fields associative array, as well as validation rules to validate form submissions by defining the $Rules associative array.

2. **Initializing the Custom Library:**
   To start building the form, developers should initialize an instance object of our custom library. For instance, in the CodeIgniter project provided, you can use the following code: *$formBuilder = service('CustomFormLibrary');* This instance will be used to create and manage the form elements.

3. **Creating the Form Tag:**
   Begin by creating the form tag using the form_open() function and specifying the URL where the form will be submitted to.

4. **Form Elements and Sections:**
   Form elements should be grouped under section names. To achieve this, you can use the new_div() function provided by our library. For example:



```
'title' =>
[
    $formBuilder->new_div(
        [
            $formBuilder->new_input('lastname', '', 'class="form-control"')
            $formBuilder->new_label('lastname', 'Last Name'),
        ],
        '',
        '',
        '',
        'form-floating'
    ),
]
```

Figure 3: Form elements creation

   In the above example, we create a 'title' section with an input field for the user's last name. Developers can then utilize the various functions provided by the library to create the required form fields, such as using the new_checkbox() function to create a checkbox field. For more details on the various functions for form creation, please refer to the source code or Notion documentation.

5. **Custom Rules (Optional):**
   If necessary, developers can create custom rules for form validation. These rules can be defined as follows:
   $Rules = ['name' => 'required'];
   This example sets a rule requiring the 'name' field to be filled in. The variable name for these rules must be precisely 'Rules' for proper functionality.

6. **Storing the Form Structure:**

   Once the form structure is created, the two variables, $fields (containing the form structure) and $Rules (if applicable), are passed to our library for further processing and storage into the MySQL database. An example of this process is demonstrated in the CodeIgniter project folder provided in our source code. Please refer to the attached Notion documentation for more elaborate explanations and detailed usage of form element creation functions in our library.

Form Rendering and Display

To illustrate the usage of the library, a sequence diagram detailing a typical workflow is provided in Figure 4.

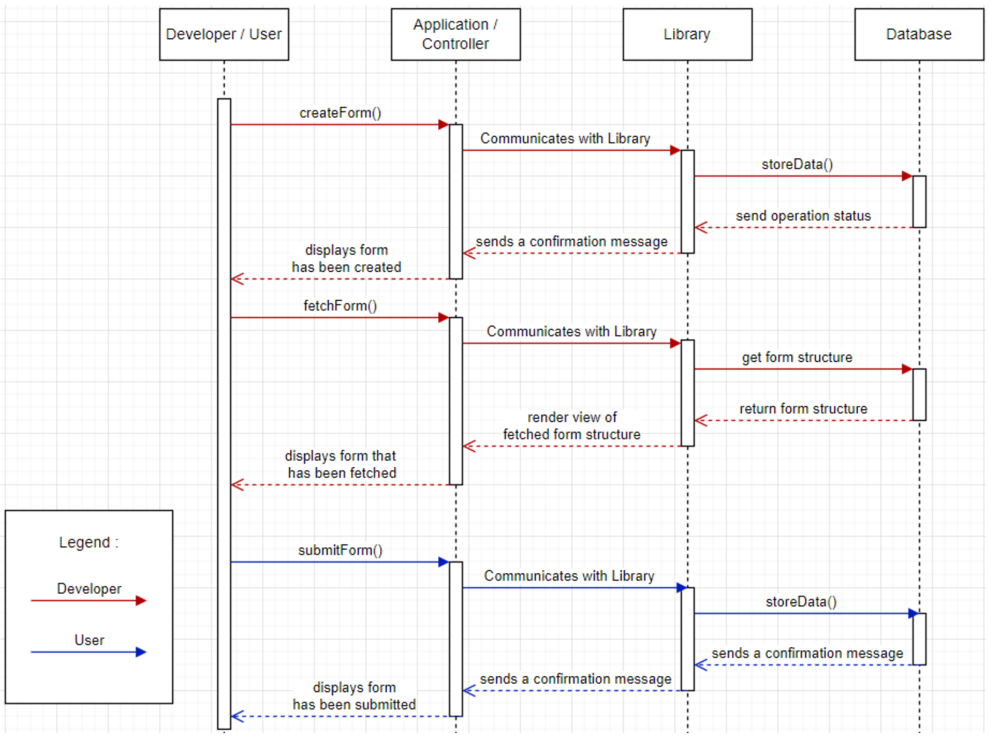

Developer / User
Application / Controller
Library
Database
createForm()
Communicates with Library
storeData()
send operation status
sends a confirmation message
displays form has been created
fetchForm()
Communicates with Library
get form structure
return form structure
render view of fetched form structure
displays form that has been fetched
Legend :
Developer
User
submitForm()
Communicates with Library
storeData()
sends a confirmation message
sends a confirmation message
displays form has been submitted

Figure 4: Sequence Diagram of typical library usage

The custom CI4 library streamlines the process of rendering and displaying forms within web applications. By following the steps outlined in the "Process of Form Building" section, developers can easily generate dynamic and customizable forms. The library utilizes the form template structure created by developers and converts it into HTML elements, allowing seamless integration into web pages.

The custom library offers a variety of HTML form elements that can be customized and styled to suit the web application's design requirements. Developers can apply CSS classes and styles to the form elements to ensure a consistent and visually appealing user interface.

The generated forms are designed with responsiveness in mind, ensuring that they adapt and display optimally on various devices, including desktops, tablets, and smartphones. The library automatically generates responsive markup with Bootstrap classes and styling, which developers can also specify during form creation.

Form Submission and Data Management

To integrate the library's functions together with programmers' web applications, the library is structured according to CI4's MVC model, which separates web applications' logic from its presentation.

Models

The library includes two models, Form and Response, which are each modelled after their respective database table. These models handle direct interactions with the database, performing Create, Read, Update and Delete (CRUD) operations through an Object-Relational Mapping (ORM) approach, whereby the use of direct SQL queries is avoided and replaced with object-oriented function calls. The Form model is responsible for querying form templates, based on the unique FormID provided by the library's controller. It will retrieve the necessary serialized form structures and return it to the library for further processing. The Response model handles CRUD operations of form response data from the controller to the database.

Views

Since the library stores forms as serialized data, programmers are required to utilize modular views to render their forms dynamically, upon fetching from the database. Sample modular views that developers can take reference from are provided in the admin web application provided. These modular views will obtain the serialized form structure and components from the database to generate the necessary HTML code to render the form. This dynamic approach allows for flexible and customizable form rendering based on the specific requirements of each form. Developers can utilize predefined views to provide a general layout for their forms and other webpage components to enhance the flexibility and efficiency in rendering and the presentation of forms.

Controllers

As mentioned previously, a suite of sample functions is provided by the library under the administrative web application for programmers to reference and use in their CI4 web applications' controllers. Programmers can use these functions to coordinate the flow of data between the models and views, for the creation, retrieval and processing of forms, storage of form responses, and updating views for display. Developers' controllers should utilize the various library functions to control their web applications.

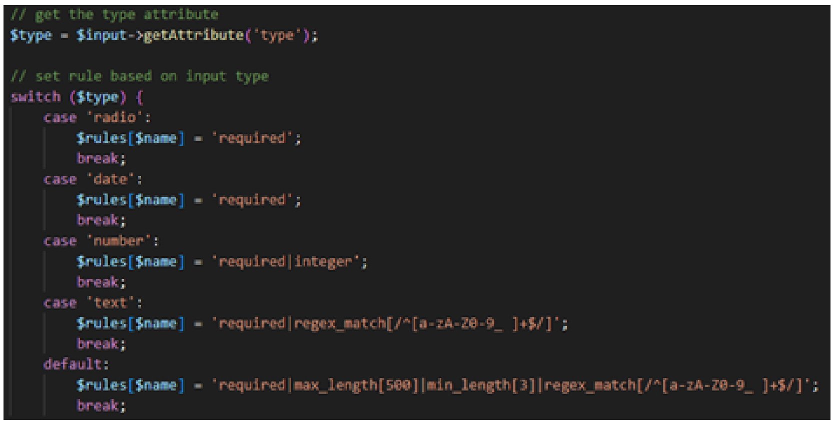
Security Management

Validation

Proper form input validations are recommended as it helps to mitigate the risk of malicious users exploiting vulnerabilities in the application and preventing them from executing malicious actions or injecting malicious content into the application. The validation of form responses is supported by the library through the specification of the $Rules associative array when developers specify their custom form templates, and through functions such as generateRulesFromHTML() which analyses the developer's form template and automatically generates predefined form validation rules. An example of validation would be in ensuring that specific fields in a form are filled up by end users properly. In the context of the F1040 form, most of the fields are compulsory to be filled, and the use of validation helps to ensure the validity of form responses by displaying error fields if users fail to fill up specific fields in the form.

Automatic Generation of Form Validation Rules

The library provides the generateRulesFromHTML() function to dynamically generate rules for individual fields based of HTML structure from the usage of the DOM. While there are multiples way to generate rules dynamically, the DOMdocument object allows us to read the html structure and identify the input elements within the html structure. As such, we can afford to differentiate between different html inputs and give a more varied rule generation through html input types. Based on the type of attributes in the input elements, we set validation rules accordingly to individual fields using regex. For example:



```php
// get the type attribute
$type = $input->getAttribute('type');

// set rule based on input type
switch ($type) {
    case 'radio':
        $rules[$name] = 'required';
        break;
    case 'date':
        $rules[$name] = 'required';
        break;
    case 'number':
        $rules[$name] = 'required|integer';
        break;
    case 'text':
        $rules[$name] = 'required|regex_match[/^[a-zA-Z0-9_ ]+$/]';
        break;
    default:
        $rules[$name] = 'required|max_length[500]|min_length[3]|regex_match[/^[a-zA-Z0-9_ ]+$/]';
        break;
```

Figure 5: Pre-defined Rules for dynamically generating form validation rules

- For all the fields excluding checkboxes, the function sets the ‘required’ rule for them all.
- For the number field, the function specifies integer rules to ensure no alphabets are entered.
- As most of the fields are text, we are only able to set minimal rules to cater for all the fields such as minimum length of 3 to 500 and accepting only alphanumeric characters.

The library also provides the generateRulesFromPOST() function which is a much simpler and provides a generic way to generate rules dynamically, which is recommended for much simpler forms as these rules are only generated after receiving POST data and cannot be used to determine that all fields that are required are filled up. The function takes in the POST data and assigns a rule to each key of the POST data before verifying it. Thus, only word limits and regex characters will be focused on validating such data. Overall, these functions are designed to simply help to ease the burden of the developer’s job in creating rules individually to all the fields and enhance the validation process of dynamic form generation smoothly.

Validation of Uploaded Files

Apart from the form validation rules on POST data received, the library supports the validation of uploaded files using the validateUploadedFile() function. This function validates the file size and file extension of the uploaded file based on pre-defined rules provided to it. An error message is displayed if the file has failed the validation while successful uploads are stored on our server and split into folders with unique numeric names with their file path saved into the database together with other fields.

Input Sanitization

In addition to validation, the library includes input sanitization functions such as sanitizeInput() in the validation_helper.php file to enhance security, by stripping out HTML tags or unnecessary spaces and escaping special characters from user’s input, helping to prevent SQL injection or cross site scripting attempts.

Data Encryption

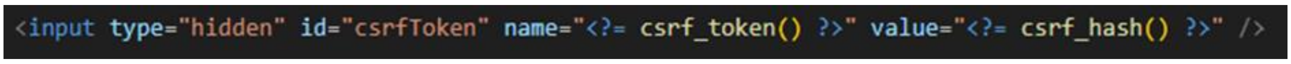
The library supports for the encryption of data using the validate() function in the validation_helper.php file. Developers can choose whether to encrypt data or not by passing in a true or false parameter. The encryption driver used is the default openSSL driver and the key used is a static key generated by CodeIgniter itself and declared in the environment file. A symmetric static key is chosen such that encryption and decryption of the value will use the same key to avoid any errors.

To begin the encryption process, the code instantiates the 'Encrypter' class from CodeIgniter's config services and verifies if encryption is needed from $encrypt value. When encryption is required, the function iterates through the $data array which holds user inputs. And each input is trimmed to remove leading or trailing white spaces. Next, the 'Encrypter' object is utilized to encrypt the data using the encryption key specified in the configuration.

Ultimately, encryption is used to transform readable data into unreadable form using an encryption key to ensure that even if there is a data breach, unauthorized parties will not be able to decipher the original information.

Cross-Site Request forgery (CSRF)

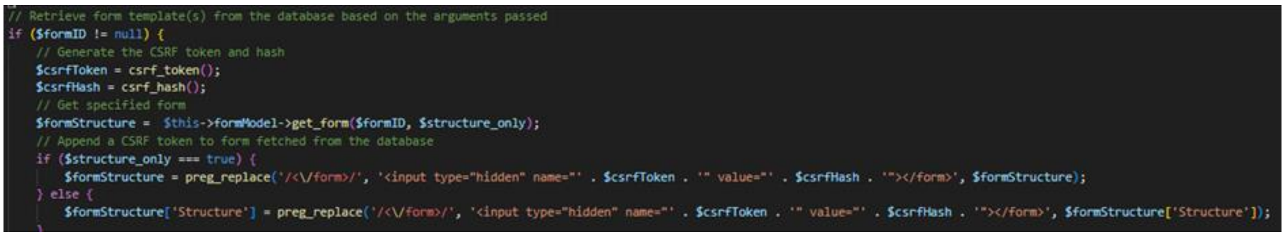
It is crucial for developers to implement CSRF tokens into their web applications to protect against CSRF attacks from malicious attackers who will attempt unauthorized actions on behalf of authenticated users. This can be performed by adding a hidden field in their forms such as:



```
<input type="hidden" id="csrfToken" name="<?= csrf_token() ?>" value="<?= csrf_hash() ?>" />
```

Figure 6: CSRF Token and Hash in HTML

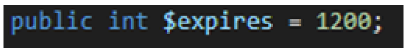
By default, the library implements support for CSRF by automatically appending dynamically generated CSRF tokens to developers' predefined forms, upon fetching from the database using the getForm() function. The following snippet shows the function's implementation by using CI4's built-in functions to generate a CSRF token and CSRF hash value:



```
// Retrieve form template(s) from the database based on the arguments passed
if ($formID != null) {
    // Generate the CSRF token and hash
    $csrfToken = csrf_token();
    $csrfHash = csrf_hash();
    // Get specified form
    $formStructure =  $this->formModel->get_form($formID, $structure_only);
    // Append a CSRF token to form fetched from the database
    if ($structure_only === true) {
        $formStructure = preg_replace('/<\/form>/', '<input type="hidden" name="' . $csrfToken . '" value="' . $csrfHash . '"></form>', $formStructure);
    } else {
        $formStructure['Structure'] = preg_replace('/<\/form>/', '<input type="hidden" name="' . $csrfToken . '" value="' . $csrfHash . '"></form>', $formStructure['Structure']);
```

Figure 7: Generation of CSRF Token and Hash

The bare-bones web application whereby the library functions are included is also configured with CSRF protections by default. For example, it reduces the default generated CSRF token's lifespan to twenty minutes from the default two hours, as shown below. In the unfortunate event that an attacker manages to obtain the CSRF token, the reduced time window will limit the period in which they can utilize the token for a CSRF attack.



```
public int $expires = 1200;
```

Figure 8: Configuration for CSRF lifespan

<u>Scalability</u>

The custom CI4 library has been designed with scalability in mind to ensure its seamless integration and optimal performance in various web application environments. As such we have implemented the following strategies:

Data-Driven Approach

Our library utilizes serialized form structures and form data stored in the database to generate form components and retrieve stored information. This approach preserves the form structure, including various input components, and speeds up the deserialization process. By decoupling the form generation logic from specific data structures, our library can easily adapt to diverse datasets and scale with the application's growth.

Optimized Query Handling

The custom library is designed to use optimized database queries to fetch and store form structures and responses. With well-crafted queries and indexing strategies, the library ensures that database operations are fast and efficient, minimizing response times and resource consumption.

Customization Options

When creating new form templates, input fields are specified using the library's predefined form template format, whereby form structures can be stored in separate files. Moreover, each element's creation can be paired with developer's choice of classes or id identification, allowing for further styling within their web applications. This allows for dynamic customization of each input, tailoring it to suit specific form requirements and ensuring data integrity and scalability with flexibility in integration.

Performance Optimization

Code Optimization

The custom CI4 library follows best practices for clean and optimized code, reducing redundant computations and improving execution times. Such practices ensure that the library remains performant even when dealing with large-scale form structures and response data.

Maintainability

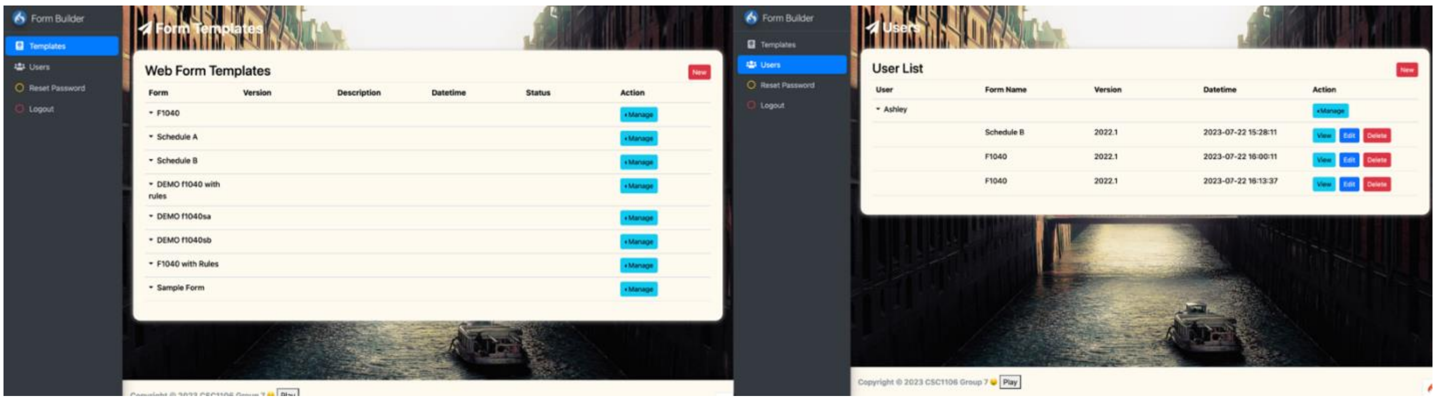
To maintain a concise and manageable codebase, the library adopts a selective approach when creating functions. Only essential functions that play a pivotal role in form building and management are incorporated. This deliberate decision reduces unnecessary overhead and keeps the library's core functionalities lightweight and focused.

Testing

Administrator Dashboard Web Application

To demonstrate the seamless integration and practical usage of the custom library, a simulated development environment was created, featuring an admin dashboard as the focal point. The admin dashboard serves as a platform for administrators to manage their created forms and form templates, showcasing the comprehensive capabilities of the library.

Within the admin dashboard, admin can effectively track and manage user-submitted form responses through the implementation of CRUD operations, utilizing function calls from the custom library. The library seamlessly facilitates the creation, retrieval, updating, and deletion of user-specific forms, providing a user-friendly interface for efficient form management.



Web Form Templates

Form	Version	Description	Datetime	Status	Action
F1040					Manage
Schedule A					Manage
Schedule B					Manage
DEMO f1040 with rules					Manage
DEMO f1040sa					Manage
DEMO f1040sb					Manage
F1040 with Rules					Manage
Sample Form					Manage

User List

User	Form Name	Version	Datetime	Action
Ashley				Manage
	Schedule B	2022.1	2023-07-22 15:28:11	View Edit Delete
	F1040	2022.1	2023-07-22 16:00:11	View Edit Delete
	F1040	2022.1	2023-07-22 16:13:37	View Edit Delete

Figure 9: Form Template & User Response Management

Additionally, the admin dashboard enables the creation and management of various form templates for users. Leveraging the custom library's function calls, administrators can utilize CRUD operations to design and customize form templates tailored to specific user needs. To ensure a streamlined workflow, the library incorporates soft deletion functionality, allowing for archiving and preserving past form templates as versions, ensuring historical continuity and reference.

Furthermore, the admin dashboard leverages the custom library's powerful functions to generate PDF files. This capability is demonstrated when users view their created forms, as well as when administrators access form templates. The library seamlessly generates the content of the PDF files, providing a professional and accessible format for viewing and sharing form-related information.

To enhance usability, the admin dashboard leverages the form HTML dump function provided by the custom library. When administrators create a new form, the library dynamically generates the form based on a config file defined by the admin. The resulting form structure is then returned as an HTML dump, enabling administrators to have a better understanding of the form's structure and layout.

Moreover, the admin dashboard includes a preview feature, allowing administrators to visualize the form structure before adding it to the database. This preview functionality utilizes the flexible export to PDF function offered by the custom library, ensuring a comprehensive and reliable form creation experience.

Overall, the admin dashboard, in conjunction with the custom library, provides developers and administrators with a robust and elegant environment for efficient form management, seamless integration, and comprehensive testing capabilities. It also serves as a reference on how to utilize the library, allowing programmers to better develop their customized form applications.

Conclusion

In conclusion, the development of the custom CodeIgniter 4 (CI4) library aimed at creating customizable forms has been a resounding success. Throughout the project, our team dedicated significant effort to design a robust and efficient solution that empowers web application developers to generate dynamic forms tailored to their specific needs. The library's seamless integration with CI4, coupled with its optimized code and performance-oriented utility functions, ensures a smooth and responsive user experience. By adhering to best practices in code optimization, we have achieved an excellent level of scalability, enabling the library to handle large-scale form structures and user responses without compromising performances.

With the Form 1040 Individual Income Tax Return form as a showcase example, the library demonstrated its versatility in handling complex form structures and efficiently storing user responses in the MySQL relational database. The intentional separation of user details allowed the library to remain lightweight while providing developers the flexibility to define their database design.

The custom CI4 library offers a comprehensive set of functions that simplify the form-building process and enable developers to create user-friendly and responsive forms for their web applications. The meticulously designed form rendering, and display process ensures compatibility with various devices, allowing for a consistent user experience across different platforms.

In summary, our custom CI4 library offers a powerful toolkit for form creation, management, and storage, while maintaining scalability, code optimization, and flexibility. As a result of this project, web application developers now have a reliable and efficient solution to design and implement customizable forms with ease. The success of this project not only showcases our team's dedication to excellence but also opens the door to future enhancements and expansions of the library's capabilities.

With the completion of this project, we confidently present our custom CI4 library as a valuable contribution to the developer community, and we hope that it will serve as a useful resource for simplifying and enhancing form-building endeavors in various web applications.