1. Objectives

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 will be tested against the United States' Form 1040 Individual Income Tax Return form to showcase its features.

2. Architecture Overview

The proposed library consists of two main components, a CI4 library for the creation and use of forms within web applications, and a MySQL relational database to store the structures and responses received for these forms. A CI4 library is used to provide a collection of reusable classes and functions for granular form customisations. MySQL is chosen as the library's database for its performance, scalability, maturity, reliability, and robust features. Programmers are required to set up both components for their web application to utilise the library's functionalities. A high-level overview of the library's design is provided in Figure 1 below.

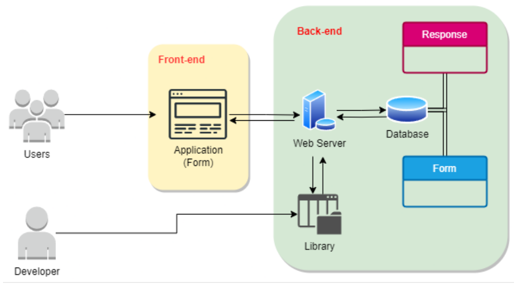

Figure 1: High-Level System Architecture

Relational 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.

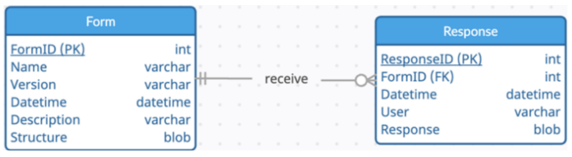

Figure 2: Relational Database Schema

The Form table will be used to store the serialised structure of forms, as well as its date of creation and version number for accounting. The Response table will be used to store serialised responses provided by the end users of these forms. Programmers are expected to maintain user details independently as such information will not be stored by the library. The Response table simply 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, whilst giving programmers flexibility in defining their own database design to complement the library.

3. Integration and Interoperability

To facilitate form operations, the library provides a suite of functions, that can be categorised as Form and Utility functions. Form functions are used to handle the main structuring, creation, and use of forms within the programmers' web applications. Utility functions are used to enhance Form functions, providing capabilities such as the import and export of forms, as well as copying the structure of pre-existing forms through cloning. Some examples of these functions are provided in Table 1 below.

Form Functions		Utility Functions	
createForm (id, array[])	Creates a new form template with a specified ID, using the form elements specified in the given array. This form template will be serialised and stored into the database.	exportForm (id, file_type)	Exports the specified form template into a specified file format (e.g. csv, pdf)
fetchForm(id)	Fetches the serialised form template with the specified ID from the database and returns it to the caller.	convertForm()	Converts the form to a different file format
updateForm (id, array[])	Updates the form template with the specified ID in the database with a new set of form elements provided in the given array.	importForm(id)	Imports a form template from a file
deleteForm (id, array[])	Deletes the form template with a specified ID from the database.	previewForm(id)	Creates a preview of the specified form template
validateForm (id, array[])	Checks whether the form response data provided in the array passes the rules and constraints of its corresponding form template.	cloneForm(id)	Generates a copy of the specified file template
storeData(id, array[])	Stores the form response data in the provided array into the database for a form template specified by the ID.		
listForms()	Returns all the form templates in the database in an array.		

Table 1: Form Functions

To illustrate the usage of the proposed library, a sequence diagram detailing a typical workflow is provided in Figure 3.

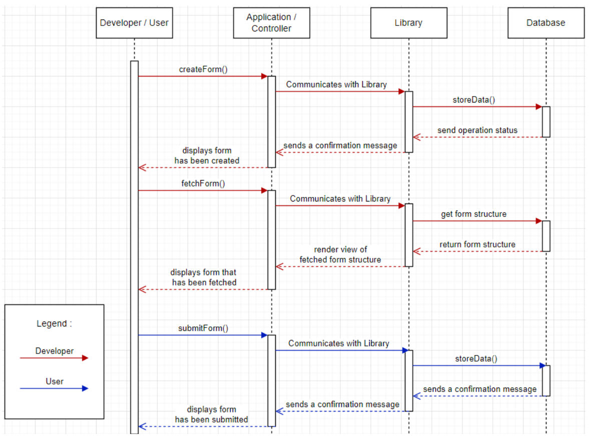
Developer / User
Application / Controller
Library
Database
createForm()
Communicates with Library
storeData()
send operation status
sends a confirmation message
displays form has been created
fetchForm()
Communicates with Library
get form structure
return form structure
render view of fetched form structure
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 3: Sequence Diagram of typical library usage

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 will include two models, Form and Response, which are each modelled after its respective database table. These models will 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 will be responsible for querying form templates, based on the unique FormID provided by the library's controller. It will retrieve the necessary serialised form structures and return it to the library for further processing. The Response model will handle CRUD operations of form response data from the controller to the database.

Views

Since the library stores forms as serialised data, modular views will be provided to enable programmers to render their forms dynamically, upon fetching from the database. These modular views will obtain the serialised 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. Predefined views that provide a general layout for programmers' forms and other webpage components will also be provided to enhance flexibility and efficiency in the rendering and presentation of forms.

Controllers

As mentioned previously, a suite of functions are provided by the library for programmers to use in their CI4 web applications' controllers. Programmers can utilize these functions to coordinate the flow of data between the models and views, for the creation, retrieval and processing of forms, as well as the storage of form responses, and updating of views for display.

4. Maintainability and Extensibility

Our approach on maintainability and extensibility includes proper documentation and the exclusion of user management from the library's database design. This separation ensures the library's independence to work solely on generating forms to prevent potential conflicts with the user management system, allowing for a smoother integration with the programmers' web applications. This separation also promotes extensibility and flexibility as the integration can be adjusted to the specific requirements of each project, while leveraging on the user management system. Lastly, this separation will result in lesser complexity, allowing for easier modification and maintenance of form generation, which ensures the library's database design to be adaptative.

5. Scalability

To handle increasing data volumes and evolving requirements, 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.

Customization Options

When creating new form templates, input fields are specified using JSON data types. This allows for dynamic customization of each input, tailoring it to suit specific form requirements and ensuring data integrity and scalability.

Example format:

```
$fields = json_encode([ ['label' => 'Name', 'type' =>
'text', 'css_class' => 'form-control', 'placeholder'
=> 'Enter your name', 'required' => true, 'disabled'
=> true] ]);
```

The library provides features to specify optional attributes such as "required" and "disabled", which allow the creation of sub forms and enforce data inputs in the form elements. The "css_class" attribute enables developers to apply custom styles and alignment to the form inputs when deployed in web applications. Additionally, the "type" option can be used to specify different input types, such as radio buttons or checkboxes, enabling the creation of diverse forms.

Performance Optimisation

We have implemented key performance optimizations to ensure efficient and scalable form generation, improving responsiveness and resource utilization, particularly for large datasets. The following techniques have been employed:

Caching Mechanism

Generated form components are cached, reducing the overhead of repeated form generation, and eliminating unnecessary database queries.

Example format:

```
// Form generation and caching
$formComponents= $CustomformLibrary ->
generateFormComponents($serializedData);
$cache->set('form_components', $formComponents, 3600);

//Retrieval from cache
$cachedFormComponents = $cache->
get('form_components');
```

Lazy Loading Mechanism

Resources are loaded only when needed, optimizing the resource usage, and improving the load time. This is to cater to larger forms or forms that require sub-components.

Example format:

```
window.addEventListener('scroll', function() {
  var formElements =
document.querySelectorAll('.lazy-load');
  formElements.forEach(function(element) {
    if (element.offsetTop < window.innerHeight +
window.pageYOffset) {
      element.style.display = 'block';
      element.classList.remove('lazy-load');
    }
  });
});
```

6. Reliability & Availability

Redundancy, fault tolerance, backup mechanisms and recovery plans are highly essential in an enterprise web architecture, CI4 provides several ways for developers to improve the reliability and availability of their webpage.

Database backup

CI4 database utility class offers developers a way to back up their database into a variable as well as the ability to write that variable into a file onto the server. Additional functions can be called to allow the file to be downloaded onto the local PC. Developers can execute this by calling the dbutil() class and invoking the backup() function to save the current database based on the current env settings. Afterwards, the developer can also call helper('file') and helper('download') to write the file into the server and download it to a local machine, respectively.

Error, exception handling and logging

CI4 provides the use of error functions globally without the developer needing to worry about the inclusion of the error class. The developer can call upon these error functions to help keep track of any errors that might occur throughout the script as well as aid in debugging the web framework.

Developers can call upon functions such as show_error(), show_404() and log_message() to help them customize their error pages to display and and give them insight on what errors have occurred through customized messages to be included in the log files. Additionally, CI4 automatically logs their show_404() calls, developers can make use of this feature to identify errors in their script.

Unit testing

CI4 includes a unit test class that can be included to help conduct test for functions within the application. This can help developers ensure that their functions are working properly as intended and increase the reliability of their web framework.

Developers can include the unit test class by loading the library('unit test') class, afterwards they can test their functions with the use of the run() function. Developers will be able pass their variable into the run function and specify their expected results. Afterwards, they can echo the result of their test or generate a report on all their test cases with the use of the report() function.

7. Security

To provide protections against common web security vulnerabilities, various safeguards are adopted by the library. For instance, a zero-trust approach is adopted when receiving user inputs. User inputs will be sanitised through the use of the htmlentities(), filter_var() and filter_input() functions to prevent the execution of malicious code, as well as regular expressions that only allow accepted characters to protect against cross site scripting and injection attacks. The library also adopts the use of CI4's built-in Object Relational Mapping approach for database interactions instead of using raw SQL queries to prevent direct code execution and mitigate the risk of SQL injection attacks.

Additionally, the library will provide developers the option of encrypting form data before storing it in the database using the openssl_encrypt() function and employment of strong encryption algorithms such as AES. This will ensure that confidential data such as Personal Identifiable Information (PII) are protected and secured, which is particularly pertinent to the target use-case for US Tax Returns. To prevent Man-In-The-Middle (MITM) attacks of fraudulent form data submission, the library will also provide developers with the option of hashing form data using the hash() function, whilst employing secure hashing algorithms such as SHA512. Developers can hash these form data on the client side and compare it with the hash received on the server side to ensure that the data has not been tampered with. This provides the option for integrity checks to prevent data tampering.

Protection against unauthorised access will be enforced using the principle of least privilege for database access. This will be implemented using GRANT query to ensure privileges are granted to respective authorised users only. Cross-Site Request Forgery (CSRF) is a common malicious session riding technique used to execute unauthorised commands on an authenticated user. CI4 has a built-in CSRF protection tool to insert a hidden CSRF field in forms. This can be enabled by altering config.php file using $config['csrf_protection'] = TRUE; and CSRF tokens can be integrated in forms through <? csrf_field() ?>. To strengthen security measures, token regeneration on every form submission can be enforced to invalidate all previous token. This mitigates session fixation CSRF attacks prone during form submissions and can be enabled through CI4's tool by using $config['csrf_regenerate'] = TRUE;.