# Assignment 3

## Topicus Team 1

Umair Aamir Mirza (s2928558)

Martin Demirev (s2965046)

Condu Alexandru-Stefan (s2769549)

Narendra Setty (s2944200)

Alexandru Lungu(s3006301)

Teodor Pintilie (s2920344)

## Introduction

This report serves to give a detailed insight into how the development team was able to set up the security modules for the project, and an insight into how the functionality of the software was tested effectively to prevent unforeseen usage circumstances. It also goes into detail regarding the design choices that were made to ensure that the user base for the project was capable of performing the functions that they required.

## Security Analysis

### Independent Security Modules:

One of the decisions taken during the planning of the security implementation was to have two independent modules for security; one for front-end, and one for back-end. Another decision taken collectively was to use any available, credible security packages that would assist in our efforts to minimize the security risks of the application. Some packages that we looked into were:

- **DOMpurify:** sanitizing HTML input to prevent any XSS attacks.
- **de.mkammerer Argon2:** more secure and robust algorithm to protect the user credentials on the database against brute force attacks.
- **org.bouncycastle bcprov-jdk15on:** for the use of cryptographic algorithms for data protection and handling.
- **org.springframework.security spring-security-crypto:** for the use of encryption and password management.
- **com.auth0 java-jwt AND com.okta.jwt okta-jwt-verifier:** to implement JWT tokens in the application, which will be discussed below.

As stated, we have made some initial research into such packages, and will await further instructions whether all of them can be used. The JWT tokens were already allowed for usage, and have been implemented in the application, but the other packages were researched so that their methods can be included once clearance has been provided.

## General Structure of Application Security:

As stated, the independent security modules on the front-end and on the back-end allow for focus on more specific elements. The contents of both modules are described in the list below:

- **Front-End Module:**
    - Input Sanitization with Libraries, Methods:
        - SQL Injection.
        - XSS Protection.
    - Authentication:
        - JWT Token in each Request.

- **Back-End Module:**
    - Input Sanitization with Libraries, Methods, Prepared Statements:
        - SQL Injection.
        - Stored XSS Protection.
    - Password Protection:
        - Argon2 for Password Hashing.
    - Authentication Management:
        - JWT Token Provision and Validation.

## JWT Token-based Authentication and Access Management:

Beginning with the uppermost layer of the security implementation, authentication and access management is conducted through the use of JWT tokens. There were several reasons for using JWT tokens, predominantly being that they are the modern mechanism of authenticating users and roles in an application. Additionally, JWT tokens are created with a special seed that only the server knows. Thus, in order to deduce if a token originates from the server, the appropriate seed must be used for decoding the token. If the token cannot be decoded by the server, then the user is using a falsely supplied JWT token, and should not be authenticated. Furthermore, JWT tokens enable secure transportation of parameters and other information in the token body itself. In our case, we will transport the ID of the user, along with their role level in the token which will be later decoded to deduce which user is requesting which data from the server, and whether they have the sufficient permissions to carry out this request or not. JWT tokens will be sent with every user request, and the server will process them accordingly. Currently, with regards to the management of the tokens, we are investigating the possibility of putting a filter class using the @Provider annotation in Jersey on the path “/api”, which will validate the token in one place, instead of having the validation at all the resources separately. We also integrated the JWT Tokens to include an expiry date, to ensure that only active users are authenticated.

The functionalities of the JWT token are conducted through the **CookieService** and **TokenService** classes in the **Service** package.

## Front-end SQL Injection and XSS Protection:

To prevent SQL injection and XSS attacks, the **fe-security** package includes the **inputSanitizer.js** file. The code, for now, includes two arrays, namely “sqlRejectWords” and “xssRejectWords” have been created which contain commonly used SQL terms (such as SELECT, -- ) and XSS terms (such as <, src) respectively. These arrays serve as a blacklist for potentially dangerous terms. However, it is important to note that blacklisting is quite ineffective due to the fact that different encoding schemes can surpass the blacklisting itself. Hence, the function **inputSanitizer()** also includes conversion to UTF-8, along with other means that are being investigated. Though DOMPurify was permitted for use, the issue with including it in the security module was that it required time to understand the package and its utilization, however, such time was not available given the backlog of work for the project itself. However, the basic structure for input validation was created, and further extensions to the project will be developed from adding proven libraries like DOMPurify into the validation.

## Front-end Input Validation and Sanitization:

Within the relevant functions for both signing up and logging in, the inputSanitizer() function is called to validate the username, email and password. If any of these inputs fail the sanitization check by returning false, an alert is displayed on the screen notifying the user of the invalid input. In the signing-up process, the code provides an additional check to ensure that the email is in the correct format. Finally, a basic password confirmation check is also performed to check whether the input of the entered password matches the confirmed password. Additionally, the functionality is used in places where user input is collected, for all user groups to ensure that there are no open security breaches. Though the inputSanitizer needs to be strengthened, it is a satisfactory implementation given the context of the application, but can always be strengthened.

## Back-End Security Against SQLi:

SQL injection is a serious threat to a lot of applications, especially ones such as ours which rely heavily on a database in order to function properly. After a meeting in which we discussed how we should protect our database, we have decided on using the following methods:

- Prepared Statements: this ensures that user input is not directly embedded into SQL statements, but rather it is supplied during runtime and replaces the placeholders that were previously set in the queries. This ensures a separation between data obtained via user input(which can be maliciously altered), from SQL logic.
- Input Validation and Sanitization: We plan on enforcing strict validation rules to restrict the data type and format of input data, preventing the execution of potentially malicious SQL code. All of the data fields received from user input are checked, first against data type validators, which ensure both type safety and user error management, and second against several regular expression matchers, which ensure that the format of the data corresponds to the standards requested by our back-end definition.
    - Format checking and request validation are done on the request object themselves on a basic level. However, we aim to create two classes, **InputValidator** and **InputSanitizer** which will be able to handle any obscure, potentially harmful data from entering the database and creating issues with the application.
- Input Filtering: In addition to validation and sanitization, we also check the input against a list of predefined blacklisted values, such as common SQL keywords. Black-listing is a relatively low-level mechanism for preventing SQLi, but the inclusion of it only serves to make our security implementation stronger.
    - A similar sort of issue arose with implementing packages for Security for the back-end, the time required to learn the features for the packages was beyond the time available at the point, and with more pressing features to implement, a basic implementation of security was conducted.
- Special Character Escaping: To prevent SQL injection attacks, we implemented techniques to escape special characters. We used escaping functions provided by our programming language to neutralize the special ability of these characters in SQL queries.

## Password Hashing:

For storing passwords inside the database we are using Argon2 which is considered a state-of-the-art, memory-hard password-hashing function that provides robust protection against various security threats. In order to implement this we created a few components such as a HashedPasswordContainer that has two fields for storing a hashed password and its salt which we use to add them to the database. Therefore, instead of storing plain text, we store a hash and a salt which later can be verified with the user input to check if the password matches the hash that we stored. All of these functionalities are provided in the PasswordService class where we implemented the methods for hashing, verifying and creating the container for the hash and salt. Using Argon2 will prevent brute force attacks given the memory requirements for computing the password, which ensures user credential security.

## Software Testing

### API/Web Service Testing:

We performed thorough Web Service Testing on our API resources using Postman. This tool allowed us to send HTTP requests to our web servers and receive adequate responses, enabling us to verify the functionality and reliability of our API.

In order to have a centralized API testing environment, we have created a shared Postman collection, which contains all the requests necessary for testing the web service resources. This has multiple benefits. The obvious one is the ability to add requests which can be seen by every member of the team. Another very important benefit is the ability to give examples of API usage. Since we've split our team into front-end and back-end development, Postman has enabled us to communicate more freely in regard to web resource access/usage.

Since we have multiple resources, we created folders for each of them and began testing. Each and every individual resource has been tested on the following use cases:

- retrieval: using GET requests with adequate JSON body responses
- creation: using POST requests with JSON body and adequate headers
- update: using PUT requests with JSON body and adequate headers
- deletion: using DELETE requests

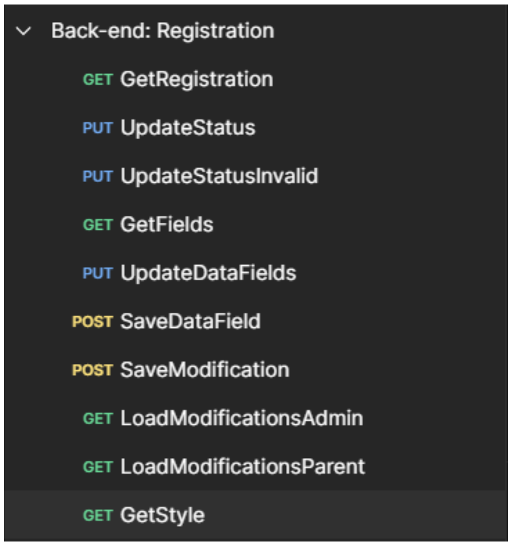
During the testing process, we encountered several bugs which wouldn't be very easy to detect without actually verifying the respective HTTP requests. Postman has definitely made our job a lot easier, and by using it, we have managed to create, thoroughly test and complete all the API resources necessary for our project, such that they are fully functional and ready to be deployed.

This level of API testing enabled us to also change certain structural details of our application, some of which are listed below:

- User-Account Interaction: changed so that it includes a more efficient way of accessing the account, a separate path for performing the log-in, and also the introduction of JWT Tokens.
- RegistrationViewContainers: the use of containers to send large amounts of data at the same time enabled the presentation of registration lists for the school administrator, and also cards for the parents on the front-end.
- Creation of Request Objects: many of these can be seen in the **Payload** package of the project, where request objects are sent and response objects are provided to reveal only the necessary information to clients.
- Invalid Request/Access Parameter Testing: enabled us to manage permissions on the application and moderate access to certain resources with appropriate Jersey Annotations (@RolesAllowed).

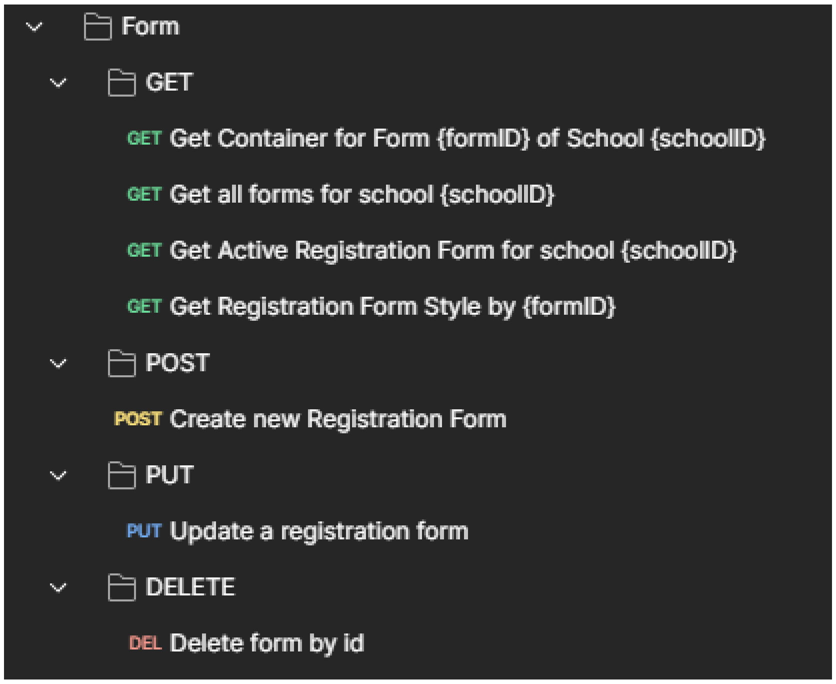
Below is a screenshot of the Postman requests that were used in the endpoint testing:

- **Registration:**


Back-end: Registration
GET GetRegistration
PUT UpdateStatus
PUT UpdateStatusInvalid
GET GetFields
PUT UpdateDataFields
POST SaveDataField
POST SaveModification
GET LoadModificationsAdmin
GET LoadModificationsParent
GET GetStyle

-

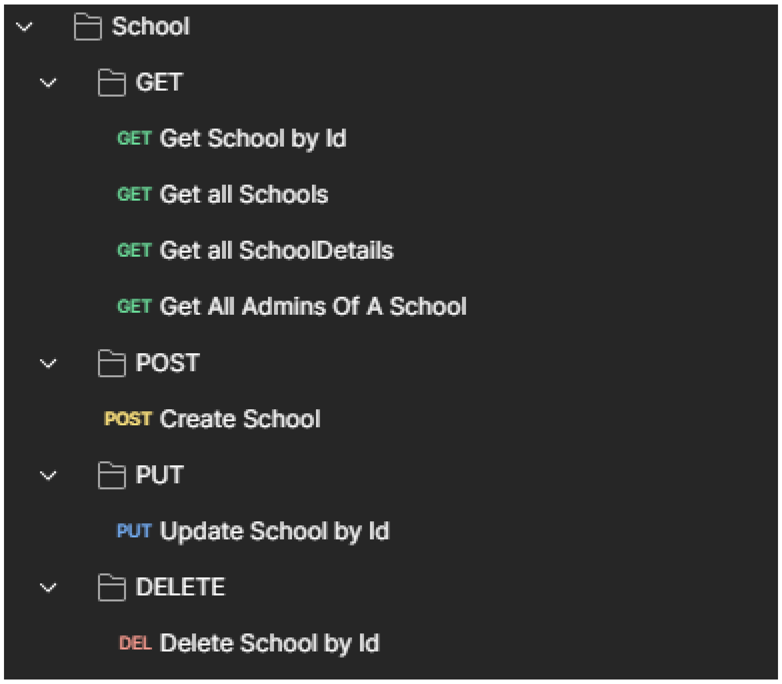
- **Registration Form:**



Form
- GET
  - GET Get Container for Form {formID} of School {schoolID}
  - GET Get all forms for school {schoolID}
  - GET Get Active Registration Form for school {schoolID}
  - GET Get Registration Form Style by {formID}
- POST
  - POST Create new Registration Form
- PUT
  - PUT Update a registration form
- DELETE
  - DEL Delete form by id

-

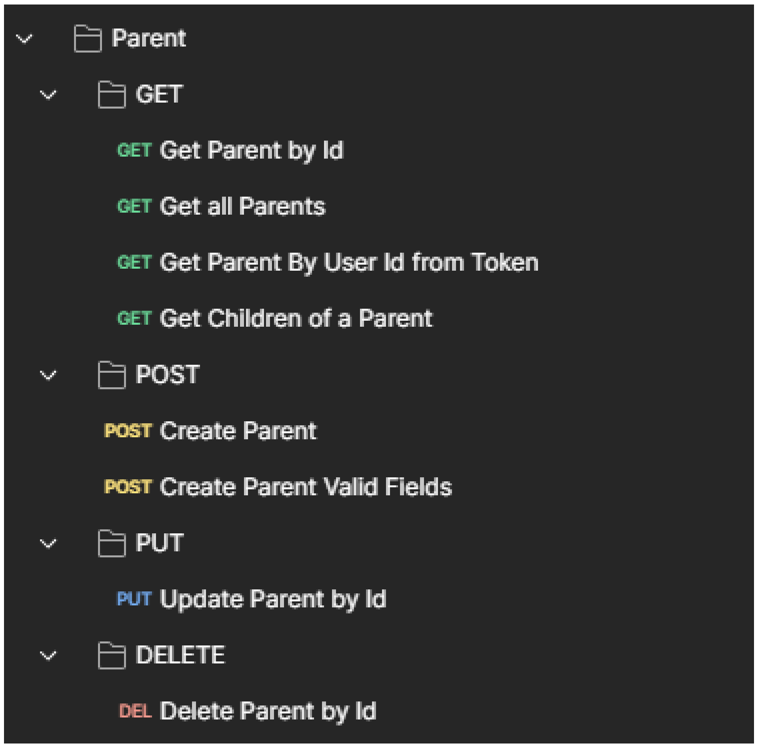
- **School:**



School
- GET
  - GET Get School by Id
  - GET Get all Schools
  - GET Get all SchoolDetails
  - GET Get All Admins Of A School
- POST
  - POST Create School
- PUT
  - PUT Update School by Id
- DELETE
  - DEL Delete School by Id

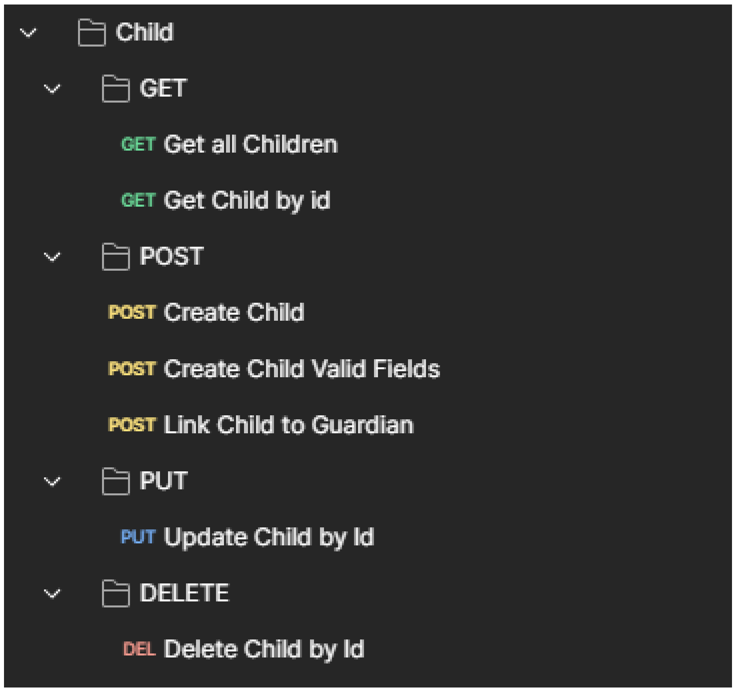
-

- **Parent:**



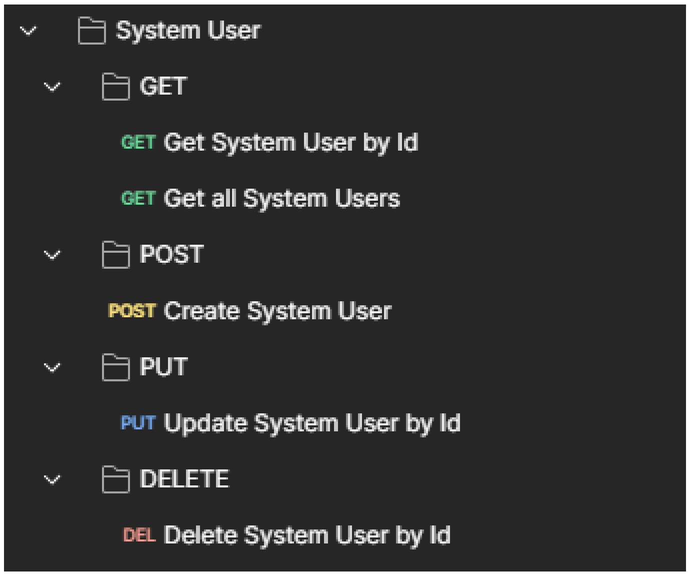
Parent
GET
GET Get Parent by Id
GET Get all Parents
GET Get Parent By User Id from Token
GET Get Children of a Parent
POST
POST Create Parent
POST Create Parent Valid Fields
PUT
PUT Update Parent by Id
DELETE
DEL Delete Parent by Id

- **Child:**

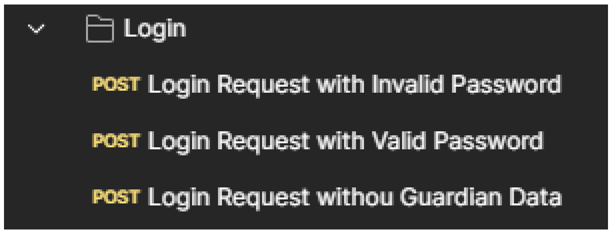


Child
GET
GET Get all Children
GET Get Child by id
POST
POST Create Child
POST Create Child Valid Fields
POST Link Child to Guardian
PUT
PUT Update Child by Id
DELETE
DEL Delete Child by Id

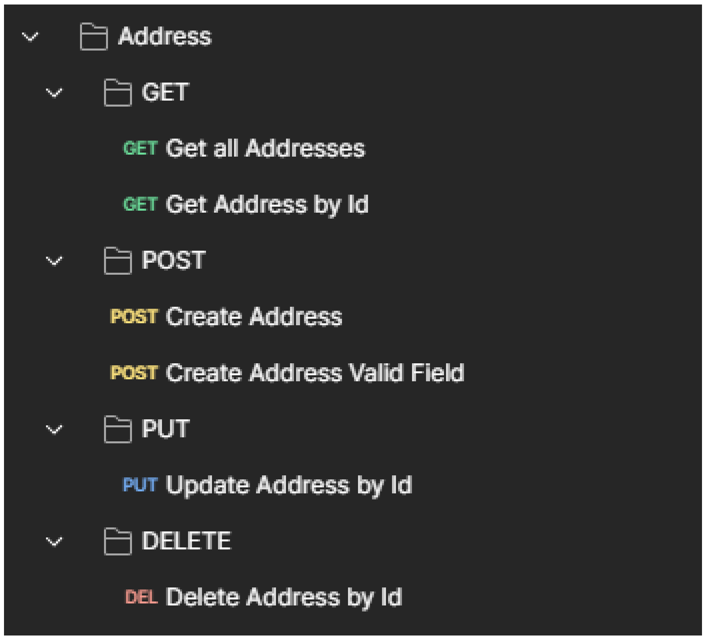
- **System User:**


System User
GET
GET Get System User by Id
GET Get all System Users
POST
POST Create System User
PUT
PUT Update System User by Id
DELETE
DEL Delete System User by Id

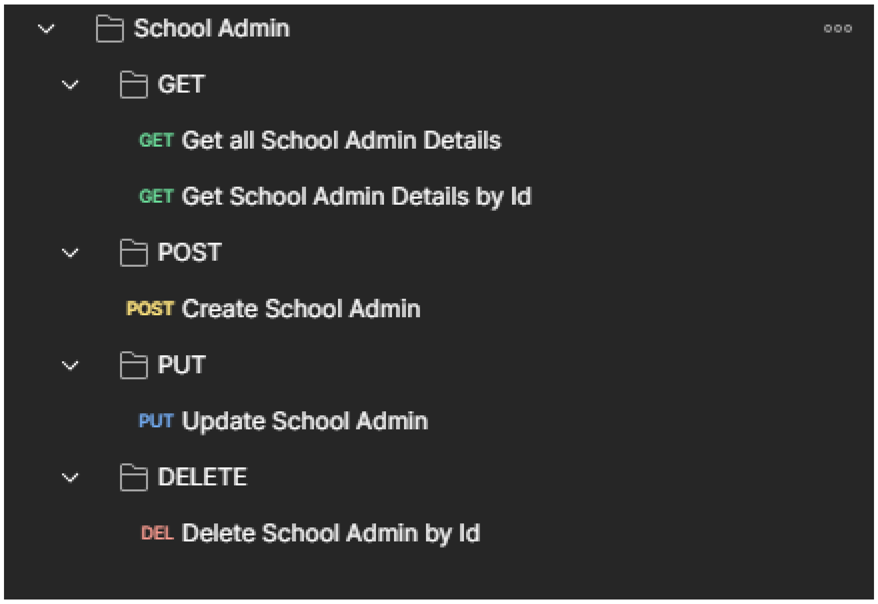
- **Login:**


Login
POST Login Request with Invalid Password
POST Login Request with Valid Password
POST Login Request withou Guardian Data

- **Address:**


Address
GET
GET Get all Addresses
GET Get Address by Id
POST
POST Create Address
POST Create Address Valid Field
PUT
PUT Update Address by Id
DELETE
DEL Delete Address by Id

- **School Admin:**


School Admin
GET
GET Get all School Admin Details
GET Get School Admin Details by Id
POST
POST Create School Admin
PUT
PUT Update School Admin
DELETE
DEL Delete School Admin by Id

## <u>Unit Testing (Back-end):</u>

For the course of the project thus far, unit testing was not investigated extensively, though an implementation plan was devised. Research has been conducted into how code can be written to test the functionalities of the back-end and interaction with the database. The planning can be summarized in the list below:

- Creation of independent JUnit Testing classes per Resource:
    - CRUD DAO Testing:
        - With the same record that is created, using the capabilities of the resource to manipulate the data.
            - We decided to focus the Unit Testing more on the DAOs as opposed to the endpoints, because the endpoints and their permissions are sufficiently tested with Postman as described above.
            - That being said, the DAOs have inner functionalities that are not entirely exposed to the Resources.
    - Application Service Testing:
        - Testing any relevant services that are used on the back-end for data manipulation or creation.

JUnit Tests available for the application testing are:

- AddressDAOTest.
- ChildDAOTest.
- ParentDAOTest.
- SchoolDAOTest.
- RegistrationDAOTest.
- SystemUserDAOTest.
- FormComponentDAOTest.
- RegistrationFormDAOTest.
- RegistrationFormMetaDataTest.
- SectionDAOTest.
- StyleDAOTest.

It is important to note that the test classes may not run properly due to the fact that the .properties file is required in a particular location. However, the tests can be seen in the project files for their functional completion. The method to add the .properties file to the appropriate location has been described and documented in the project README file as well.

## User Black-Box Testing (Front-End):

For the front-end, testing was done through using actual users rather than the development team themselves, to prevent bias in the functionality assessment. The development team requested friends and family members to use the application features in a given contextual situation, and collected the necessary feedback to improve the application. Some of the improvements that were suggested are included below:

- Inclusion of Instructional Pop-ups.
- Color-Coding of Buttons and Registration Statuses.
- First Response to Server Causes No Information on the Page (inconvenience).
- Instructions or Descriptions for Each Page.
- Styling and Animations Minimization (on some PCs, the animations lead to web pages being displayed slowly).

The feedback above is not particularly difficult to implement, barring the aspect with the first request to the server, which we will attempt to fix by reinstalling the drivers for PostGreSQL. However, this enabled the development team to understand that the state of the front-end is good, as the functionalities are being performed as required. Most of the enhancements mentioned were included in the fourth and final

SPRINT, some were left out as they would cause larger refactoring changes than intended, which would require re-evaluation of application data transfer.

### Project System Testing:

In the appendices of this report, the System Testing table shows the various tests that were conducted to judge the performance of the system (on a basis of expected results versus actual results). The System Testing for the project was conducted in a logical manner, where the outcome of the testing was used to not only improve the dysfunctional code, but also to improve the quality and organization of the code.

## Conclusion

The security and software testing report enabled the development team to re-assess, and redistribute the workload with a greater focus on making sure that the system is as robust as possible.

## Appendix A: System Testing Table

Color Code: Parent School Admin Topicus Admin All Users

| Action: | Expected Outcome: | Actual Outcome: | Discussion: |
|---|---|---|---|
| Login to the application from the Login page with the correct credentials.<br><br>(Parent with Account) | The expected outcome is that the parent is granted access to their dashboard, where all of their existing registrations are shown. | The actual outcome is that the parent is granted access to their dashboard, where all of their existing registrations are shown. | The key derivations from this aspect were to make sure that if there were not any Registrations received, the dashboard should not display an error, rather it should just show nothing. |
| Login to the application from the Login page with the wrong credentials.<br><br>(Parent with Account) | The expected outcome is that an alert will pop up warning the user that the credentials are wrong. | The actual outcome is that an alert will pop up warning the user that the credentials are wrong. | This action is used to ensure that our password hashing feature is able to verify the user password stored in the database with the input. |
| Parent opens dashboard (Parent with Account) | The expected outcome is that all the registrations of the parent are loaded when the user enters the page | The actual outcome is that all the registrations of the parent are loaded when the user enters the page | This action should check if the registration is correctly fetched. |
| Parent opens 'Apply for Registration' | The expected outcome is that a search page will be displayed where based on a search bar the user can select the school of choice. | The actual outcome is that a search page is displayed and based on a search bar the user can select the school of choice. | This action is verifying if the search bar is a functional component. |
| Parent chooses a school | The expected outcome is that the school forms are loaded and the user can press apply to which registration is suitable for him | The actual outcome is that the school forms are displayed and the parent can choose whatever he likes | This action is just useful for seeing if the forms associated with that school are properly fetched. |
| Parent apply for a registration | The expected result is that the registration is displayed on the frontend for parents to fill it in. | The actual result is that the registration is displayed on the frontend for parents to fill it in. | This action shows if we display the data properly and we fetched it correctly from the database. |
| Parent puts in wrong data in the form | The expected outcome is that an alert will pop up when the parent tries to submit the form saying that the input is invalid. | The actual output is that an alert is popping up when the parent tries to submit the form saying that the input is invalid. | The purpose of this action is to show if the validator that we use on the frontend can correctly detect errors. |

| | | | |
|---|---|---|---|
| Parents puts in the form proper data | The expected outcome is that the application will be finished and the user will be directed back to the dashboard where the registration is now visible. | The actual outcome is that the application will be finished and the user will be directed back to the dashboard where the registration is now visible. | This action is meant to check if the registration is successfully created and stored in the database. |
| Selects one of the filters in the Dashboard page | The expected outcome is that the page will load only the registrations that match the filters. | The actual outcome is that the page will load only the registrations that match the filters. | This action is important to observe if the filters the application set are properly matching the registrations. |
| Click 'Forms' from the navigation bar | The expected outcome is that the page will load all the forms stored in the database. | The actual outcome is that the page is loading all the forms stored in the database. | This action shows if the page is properly fetching the database data and it is displaying in the right format. |
| Click 'Edit' button on a random form | The expected outcome is that the page will switch to an editing one similar to the 'Create Form' where all the information is loaded with the possibility of editing. | The actual outcome is that the page shifts to 'Create Form' in the navigation bar and loads the current form details with the possibility of editing. | This action shows if the details of a form are correctly stored and if modifying them works. |
| Click 'Create Form' from the navigation bar | The expected outcome is that an empty page will load where you have the features built for creating forms. | The actual outcome is that the page loads the previous form selected details and the necessary tools for editing it. | This action is supposed to show if the functionality of adding a new form works properly. |
| The user logs in | The expected result is that the user is redirected to the dashboard where is supposed to see all schools and all admins from the database displayed in the right format. | The actual result is that the user is redirected to the dashboard where is supposed to see all schools and all admins from the database displayed in the right format. | This action shows if the schools and the admins are fetched and displayed correctly on the frontend. |
| The user fills in a school/admin form and submits on the 'Add School/Admin' button | The expected result is that the page will alert the user if the request worked or not. In case it works it will return to the dashboard where the added resource will be visible. | The actual result is that the page will alert the user if the request worked or not. In case it works it will return to the dashboard where the added resource will be visible. | This action shows if the post requests go well through the application and if data is manipulated properly. |
| After the user clicks the remove admin/school buttons.<br>(Topicus Administrator) | The expected result is that the page will automatically reload and the deleted school/admin will not be visible anymore. | The actual result is that the page is automatically reloaded and the deleted school/admin is not visible anymore. | The reason for this action is to check if the school/admin is truly deleted in the database since the dashboard is fetching the database each time it is loaded. |
| Clicks 'My Profile' in the | The expected result is that My Profile | The actual result is that My Profile is loading | This action shows if the application is |

| | | | |
|---|---|---|---|
| navigation bar | should load all the details of the account of the user. | all the details of the account of the user. | correctly fetching the user details. |
| The ‘View’ buttons in all the application | The expected result is that the page will load the details of the specific resource from the database and display them in a pleasant format. | The actual result is that the page will load the details of the specific resource from the database and display them in a pleasant format. | This action is testing if the get requests are correctly sent and data is correctly manipulated inside the application. |