Extra: Design Choices and Project Structure Guide

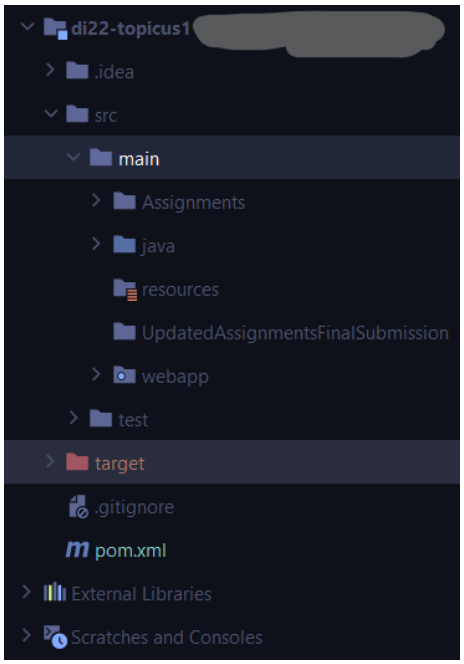
Topicus Team 1

Umair Aamir Mirza (s2928558)
Martin Demirev (s2965046)
Condu Alexandru-Stefan (s2769549)
Narendra Setty (s2944200)
Alexandru Lungu(s3006301)
Teodor Pintilie (s2920344)

Introduction:

This document contains a brief description of how the project is structured, and some key design choices that were made from a general point of view. Where required, specific design choices were analyzed, and some reflection was provided regarding further code improvement and structural awareness.

Project Structure: Zoomed Out



As shown in the screenshot on the right, the files for the project are organized in these particular general folders.

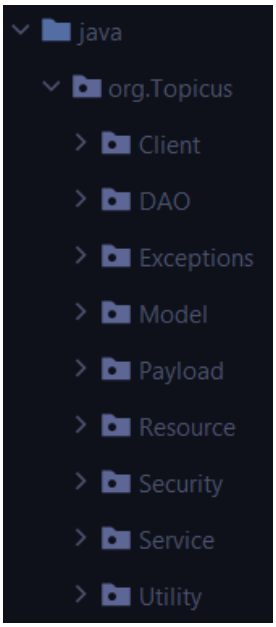
Main contains the following:

- ***java*:** contains source code for Java files.
- ***Assignments*:** old versions of the Project Assignments.
- ***UpdatedAssignmentsFinalSubmission*:** latest versions of assignments that are required, including additional reports that may be useful for understanding the project.
- ***webapp*:** folder contains the necessary source code for front-end feature implementation.

For the report, the greater focus will be on the back-end (***java***) and front-end (***webapp***).

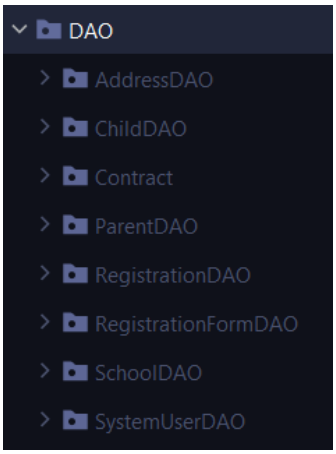
Project Structure: Back-end (Folder: src/main/java)

This package (***java***) contains the relevant source code for the majority of the back-end. Inside of the package, there are uniformly organized folders representing a particular group of the element.



The descriptions of the packages are found in the list below:

- **Client:** folder that consists of relevant client-focused services.
    - Contains the code for the *ContextUpdateListener*, which is essentially a class that resides on the server and sets the contextual data (in this case, the PostGreSQL Jar) so that the application can proceed without issues in loading the contextual information.
    - This was added to spare clients the issue of reloading the pages multiple times due to the PostGreSQL driver not being located.
    - Methods have JavaDocs for further clarification.
- **DAO:** this folder consists of all the DAOs (Data Access Objects) that are used on the back-end.
    - The Contract folder consists of the appropriate DAO methods that should be in each DAO regardless of the object(s) that it is responsible for managing.
    - Each DAO is an enum with a static method **getInstance()**, and the DAO classes are structured in the following manner:
        - SQL Queries used at the top.
        - Individual methods (for modularity purposes) for specific transactions.
        - Each transaction method has the relevant JavaDoc associated with it.

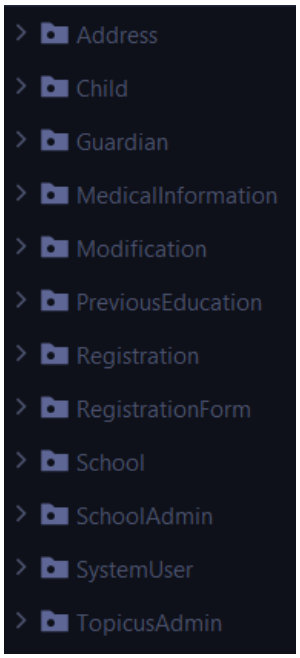


    - The only exception to the structuring of one DAO per DAO folder is the **RegistrationFormDAO**, which uses smaller DAOs to its advantage. Thus, these other DAOs which are not used in a notable amount elsewhere, were grouped in the same package for convenience.
- **Exceptions:** this folder consists of all the custom exceptions that were defined in the application structure.
    - Upon further reflection, some more custom exceptions being defined would have enabled a greater deal of control over the application to identify what was going

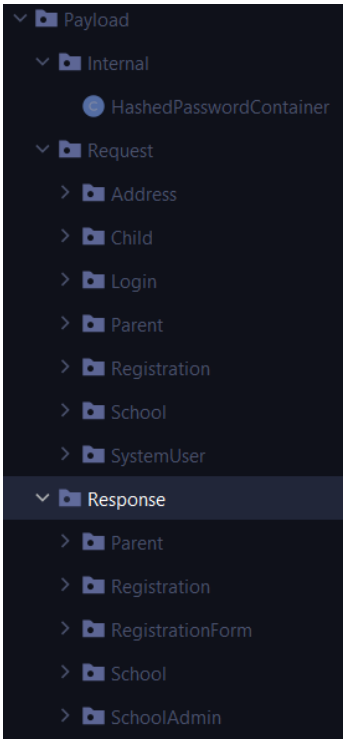
wrong in the particular areas, and to enable certain functionality for the responses and object construction.

- However, the essential custom exceptions have already been defined.

- **Model:** consists of all the object classes used for data representation.
  - The classes followed the following structure:
    - FIELD VARIABLES.
    - CONSTRUCTOR(S).
      - Default constructor for Jersey.
      - Full constructor for back-end to front-end parsing.
      - ID-Empty Constructor for front-end to back-end parsing.
    - GETTERS.
    - SETTERS.
    - UTILITY FUNCTIONS.
  - The classes in the model represent the real-world objects that were required for the modeling of this application.



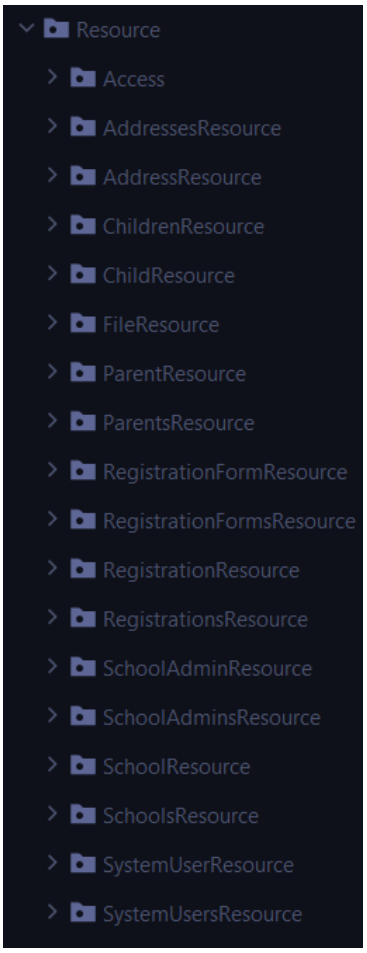
- **Payload:** this folder contains the structured request and responses used across the application for data transport purposes.
  - Focusing on the specific internal packages to this **Payload** package:
    - **Internal:**
      - Used for intra-server communication, any files moving between large, independent classes on the server that need special processing at different points.
    - **Request:**
      - This class contains requests (organized by folders for each Resource), and a request is essentially the body that the **fetch**(URI) should contain, in the appropriate syntax.
    - **Response:**
      - This class contains responses (organized by folders for each Resource), and a response is



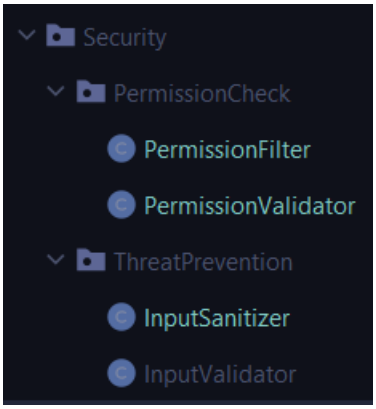
essentially what the front-end can expect from the server as a response to certain operations.

- This way, the front-end knows how to parse the response into JSON and use the data where applicable.

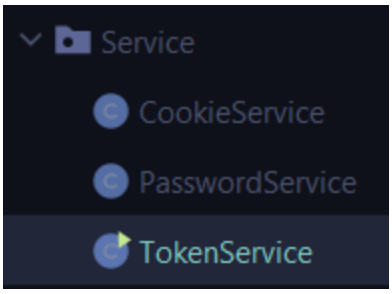
- This structure enabled much greater clarity in the independent development of the front-end and back-end, and due to this structure, the final connection of the application was possible without significant issues.
- The principle to recall, in order to understand the usages of the classes, is that:
  - **Request: Front-end → Back-end**.
  - **Response: Back-end → Front-end**.
  - **Internal: Back-end ↔ Back-end**.



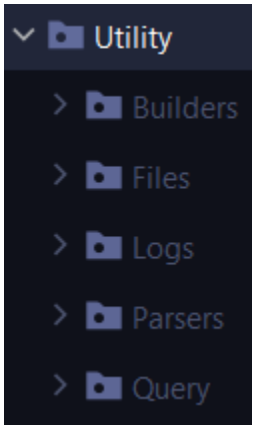
- **Resource:** this folder contains all of the resources that were required for the application to perform its functionalities in a successful manner.
  - Each resource contains the relevant JavaDoc that describes why it is required, and each method of the resource is also well-documented to explain the reasons why the method is required for usage.
  - Each of the Resource endpoints was tested thoroughly with the Postman tests, which are exported from Postman as the file *Topicus 1_Postman Collection.json*.
- **Security:** this package contains the necessary classes for authentication and input validation/sanitization.
  - **PermissionCheck:**
    - Focuses on the classes that manage the authentication to specific URIs and validate incoming requests for their permission levels.
  - **ThreadPrevention:**
    - Sanitization of input and assurance that certain input field types are respected to the data type format that is required.



- **Service:** this package consists of any necessary services for processing, protecting, and creating useful data.
    - The CookieService is used to create cookies and manage them.
    - The PasswordService is used to hash passwords with Argon2 for secure transportation, and also for validating if a user's credentials are correct.
    - The TokenService is used to implement JWT Tokens to the application, and contains the respective methods to validate them as well.



- **Utility:** this package consists of an abundance of utility classes that were used to make processes easier to conduct.
    - **Builders:** this package consists of relevant classes for the ConnectionPool (which was not implemented).
        - However, the classes were left to show the considerations for concurrency made.
    - **Files:** this package consists of the relevant classes for handling file transport in the server.
    - **Logs:** this package is used to manage the Logger that is used on the Server for convenience purposes.
    - **Parsers:** this package is used to contain parsers for data that is retrieved from the database to convert it into usable objects.
    - **Query:** this package contains files for the query optimization.



<u>Project Design Choices:</u>

From a technical standpoint, there were a variety of design choices made for the project. There are too many to discuss explicitly, but in general, the following design choices were the most relevant:

- Similar DAO Structure:
    - DAOs were made more modular to allow for easier testing, debugging, and to enforce separation of concerns (SOC).
    - In this manner, testing was able to be conducted in a methodical way to assure that the endpoints were working as functional.
    - Some DAOs were rushed to completion, meaning that the logical elements behind them were not as refined as some of the most popular and complete DAOs. This is

an area for improvement, as more time would have enabled refactoring of code for efficiency rather than focusing explicitly on functionality.

- Payload Folder:
    - The Payload folder was extremely critical in the independent development of the application front-end and back-end.
    - Knowing exactly what requests and responses would be sent allowed for the most clarity in performing functionalities.
- Project Package Structure:
    - The general package structure was very clean, enabling the organization of the project in a comprehensible manner. The only further additions would be the following:
        - Isolated JavaScript folders for easier navigation.
        - Isolated HTML folders for easier navigation.
- DAO Parsers:
    - Implementing parsers spared immense amounts of code redundancy, which was extremely useful for cleaner code.
- JavaScript Utility Classes:
    - Utility classes for JavaScript enabled more code efficiency than normal. Due to time constraints, not all the code could be refactored to include utility classes (especially the .js files made in the initial stages of the project, before the development team was comfortable with the language and its features).

However, we also wanted to highlight some more advanced features that we implemented that were beyond the scope of what was taught:

- Authentication Listener.
- Authentication Annotations.
- Logger Manager.

One of the points of improvement would be to refactor code in a more organized way. Independent development is good for projects of moderately sized development groups, however, if general utility classes are not defined at an early stage, it leads to code duplication and redundancy. This caused our JavaScript code to be less efficient than it should have been, but it is a point of improvement given that it was also critical for the development team to learn the language and implement the features before focusing on the refactoring of the code.

Conclusion:

This concludes the overview of the general project structure and design choices made. This brief report does not encompass all of the design choices made, which can be different for each given context that a method or file serves. However, the report offers an insight into the planning behind the development of features.