Web Data Harvesting and Querying with Haskell: A Functional Programming Group Project

1. Introduction

This project demonstrates how to use Haskell to retrieve, store, and query structured data from the Transport for London (TfL) API. By leveraging functional programming concepts and a modular codebase, the application can download stop point information, validate the results, insert them into a local SQLite database, and provide users with commands to query or export stored records. In addition to meeting the base requirements—fetching data, saving it, and allowing queries—this project lays a foundation that can be extended with more complex features, such as enhanced validation or custom filtering logic.

2. Setup, Dependencies, Data Source, and Structure

The project relies on the stack tool to provide a consistent Haskell environment and an LTS snapshot for stable builds. Essential dependencies include http-conduit for robust HTTP fetching, aeson for decoding and encoding JSON, and sqlite-simple for lightweight database operations with SQLite. The chosen web API—the Transport for London (TfL) endpoint—supplies JSON-based stop point data, which we map onto strongly-typed Haskell records defined in Types.hs. Using Fetch.hs, we request data from the TfL API, while Parse.hs converts the raw JSON into a list of StopPoint values, ensuring the response is not empty before proceeding. The Database.hs module handles table creation, data insertion, and queries on a local SQLite database, keeping track of each stop point and its associated modes. Finally, Main.hs provides a command-line interface that integrates these components, offering commands to set up the database, load data, export records to JSON, and run custom queries. This layered design ensures clear separation of responsibilities and makes the application both maintainable and extensible.

3. Module Responsibilities

The application is organized into distinct modules that each handle a specific aspect of its functionality. Fetch.hs manages all HTTP requests to the web API and handles network-level errors. Parse.hs decodes the received JSON into typed Haskell structures, ensures the data meets minimal validity conditions (e.g., that there is at least one stop point), and can also produce JSON output for exporting database records. Database.hs focuses on creating SQLite tables, saving the parsed data, and executing queries against the stored records, leveraging sqlite-simple for simplicity. Finally, Main.hs integrates these components, providing a command-line interface with commands for database creation, data loading, data dumping, and querying.

4. Running & Using the Application

To build the application, run stack build from the project’s root directory. Once built, you can use stack run -- <command> to interact with the app. For example, stack run -- create initializes the SQLite database, stack run -- loaddata <lat> <lon> <radius> <modes> <stopTypes> fetches and stores records from the web API, stack run -- dumpdata exports all stored records to a dump.json file, and stack run -- query "<condition>" retrieves records from the database based on custom search conditions.

5. Challenges

Developing this application presented several technical challenges:

**Working with JSON and Strong Typing**:

* Mapping the JSON responses from the TfL API to strongly-typed Haskell data structures was challenging, especially given the nested and optional fields in the data. This required a deep understanding of the aeson library and the creation of custom parsing logic to ensure correctness and type safety.

**Database Integration**:

* Designing a database schema that could efficiently store structured data while remaining flexible for queries was non-trivial. Handling data insertion and retrieval using sqlite-simple while adhering to functional programming principles demanded careful abstraction in the Database.hs module.

6. Conclusion

The project demonstrates the use of Haskell for an application that integrates web data fetching, validation, storage, and querying. The project also ensures error handling and maintainable code. The modular design facilitates current functionality and makes it practical for future enhancements.

While challenges such as JSON parsing, database integration, and error handling added complexity, they also provided opportunities to deepen the technical knowledge regard,ng Haskell and functional programming. The project highlights the advantages of using functional programming for real-world tasks, including clear code separation, type safety, and composability.

Overall, this application provides a powerful, extensible tool for working with structured data from the TfL API and serves as an example of Haskell's capabilities for building reliable and scalable systems.