RDF-Based Form for Recording Fractures in Suspected Child Abuse Cases

Sagar Honnenahalli Somashekhar

Supervisor: Dr Fatima Maikore

A report submitted in fulfilment of the requirements for the degree of MSc in Advanced Computer Science

in the

Department of Computer Science

September 11, 2024

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Sagar Honnenahalli Somashekhar

Signature: Sagar Honnenahalli Somashekhar

Date: 11/09/2024

Abstract

This project aims to create a tool that helps gather and save information about fractures in kids, especially when there's a concern about child abuse. This tool is part of a larger system called ELECTRICA, which uses data to help protect children. It offers easy-to-use forms for recording family history and fracture details of kids who might be abused. This information is then turned into a special format called RDF and linked with the ELECTRICA system for better understanding. The tool mainly helps with collecting data and finding information using SPARQL. It also prepares for future connections with tools that can predict the chances of child abuse. This project lays the groundwork for possible future improvements, like using the data to make smarter guesses based on the ELECTRICA system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Child maltreatment constitutes a significant worldwide concern, exhibiting profound and enduring implications for physical and the psychosocial well-being of the affected children. In the United Kingdom, The National Society for the Prevention of Cruelty to Children, NSPCC, is a major organization committed to the protection of children. In 2021/22, the NSPCC helpline recorded concerns about physical abuse in 6,441 children [16]. These shocking statistics emphasise the need for efficient measures and procedures for identifying, recording, and responding to suspected child abuse more efficiently. This dissertation is intended to develop a semantically enriched tool that collects information about the fractures and any other relevant injuries of the children, and in particular suspected cases of child abuse. This tool will facilitate clinicians, researchers, and trainers by providing them with a systematic record of critical incidents, which complements the wider work of pervasive Computing Group and the associated ecosystem of tools under the ELEctronic tool for The Clinicians, Trainers, and Researchers In Child Abuse - ELECTRICA [3].

.

1.1 Motivation and Objectives

The motivation for this research program is given by the increasing emphasis on improving data collection and processing concerning incidents of child abuse. Having well-defined database is basically important for developing prediction models that determine the likelihood of child abuse in order to enable timely and appropriate interventions. In most cases, the current methods of data collection show a minimal unification and semantic consistency, resulting in heterogeneous and unreliable datasets. This work provides for this deficiency through the use of semantic web standards, incorporating them into the ELECTRICA ontology. The main objectives include:

1. Intuitive forms: Easy and simple webpages will be created where healthcare professionals can report a variety of fractures and their related injuries.

2. RDF Data Conversion: Data typed in the form needs to be converted into RDF

data, using great semantic care in conformance with standards established by the ELECTRICA ontology.

3. Ontology Integration: The process of integrating the form with the ELECTRICA ontology is essential for upholding semantic consistency and improving data interoperability.

4. Data Acquisition and Validation: SPARQL-based query framework is used for extracting and representing RDF data for verification.

The stretch goals include:

1. It is remarked that adding reasoning capabilities will enable the drawing of more advanced inferences about the ontology, but this should be an optional capability in the framework of the tool.

1.2 Overview of the Report

This tool is foreseen to achieve significantly higher levels of data acquisition in cases related to alleged child abuse. The implemented tool will allow for increased sharing and interoperability of the collected RDF data to the parties involved in child protection through a standardized and semantically enhanced means of entry and storage of information. Accordingly, this will eventually allow the development of more and more accurate predictive models and also effective interventions that, ultimately, will help to promote the child welfare system.

This project thus represents a fundamental step forward in the technology-based struggle against the abuse of children. The proposed semantically enhanced online form is supposed to constitute an important contribution to the researchers' and healthcare practitioners' toolkit for the protection of abused children.

Chapter 2

Literature Survey

2.1 Background Research

Extensive research has been carried out to detect and support child abuse victims, and much importance has been given to accurate data collection and analysis. There is also a greater need for the availability of standard tools in child protection services. For instance, NSPCC have revealed disturbing facts regarding incidents of physical abuse. While there are systems that document cases of child abuse, a significant portion of such systems do not adopt sophisticated web technologies to enhance the capacity for data sharing and analytics. Of these applications, semantic web technologies, including Resource Description Framework, have shown tremendous promise due to their structured data, allowing for high-level querying and inference. RDF, in addition to standards like OWL, enables capturing data that is well connected so that machines can easily understand them. Particularly, such functionality serves as a key enabler in the creation of predictive models within the context of child abuse detection, wherein it is important to comprehend and evaluate complex data connections. To make the most of this chapter, it is enriching to examine some of the current tools using these technologies. Recent research publications identify two important examples of this:

1. Semantic Technologies Applications in Healthcare: [17] This article discusses the use of semantic web technologies in healthcare, focusing on the advantages of RDF and OWL in structuring data for advanced analytics. It stresses the importance of these technologies in building interoperable data systems that allow predictive analytics, which improves patient outcomes. The findings indicate a possible relevance of these technologies in systems developed for the detection of child abuse, where the investigation of complex relationships among data forms the basis.

2. Tools for Data Integration in Healthcare: [20] This investigation examines a range of data integration tools and their implementations within the healthcare sector, specifically focusing on those utilizing semantic web technologies to enhance data interoperability and accessibility. This article also highlights case studies, showing how such tools improve data sharing between diverse systems and, in effect, enhance data analysis in

order to support decision-making. The results of this study can be used to progress the children protection framework by enhancing data accuracy and efficiency in collection and analysis.

2.2 Semantic Web Technologies

Semantic web technologies such as RDF, OWL, and SPARQL serve as fundamental components for developing interrelated and semantically abundant data. RDF enables the graphical representation of information pertaining to resources, thereby simplifying the process of linking relevant data across various domains [2]. OWL enhances RDF by introducing an expanded vocabulary for delineating properties and classes, which facilitates the construction of more intricate ontologies [11]. SPARQL-the query language for RDF-allows the ability to query and manipulate RDF data, making it a critical tool for retrieving data in semantic web applications
[19].

2.3 Ontology Integration

Ontologies are formalised frameworks that provide a structured representation of knowledge within a specific domain by defining a set of concepts and the relationships that exist between those concepts. These frameworks are crucial in various fields, including healthcare, as they enable the standardisation and organisation of data in a manner that enhances understanding, interoperability, and usability across different systems. By providing a common language and structure, Ontologies facilitate the integration of data from diverse sources, allowing for more comprehensive and accurate data analysis[14],[21]. In healthcare, the integration of Ontologies has been particularly impactful. Ontologies such as SNOMED CT, ICD, and HL7 have been widely adopted to standardise medical terminology, ensuring that data from different healthcare providers and systems can be understood and utilised effectively[5], [6]. This standardisation is essential for achieving data interoperability, which allows different healthcare systems to exchange and use information seamlessly. It also supports advanced analytics by enabling the aggregation and comparison of data across multiple sources, leading to more informed decision-making and improved patient outcomes [8]. In the context of child abuse detection, the use of Ontologies becomes even more critical. Child abuse is a complex and sensitive issue that requires careful documentation and analysis of various data points, including medical records, social services reports, and law enforcement data. The ELECTRICA ontology (ELEctronic tool for Clinicians, Trainers, and Researchers In Child Abuse) is specifically designed to address this need by providing a standardized framework for recording and analyzing data related to child abuse cases[21]. By integrating ELECTRICA into a system for collecting and storing data on fractures and related injuries in children, the tool ensures that all data collected is semantically consistent. This semantic consistency is crucial for several reasons. First, it allows for more accurate data analysis, as the relationships between different data points are clearly defined and understood. For example, the ontology can help identify patterns of injuries that are indicative of abuse, enabling more accurate risk assessments. Secondly, it facilitates data sharing between different stakeholders, such as healthcare providers, social services, and law enforcement, by ensuring that all parties are using the same terminology and data structures[14]. Furthermore, the use of an ontology like ELECTRICA supports advanced reasoning and inferencing capabilities. By defining the relationships between different concepts within the domain of child abuse, the ontology enables the system to make inferences based on the data it collects. For example, if a child presents with a pattern of injuries that matches a known pattern of abuse, the system can flag the case for further investigation. This ability to reason about the data in a structured way enhances the tool's effectiveness in detecting and preventing child abuse[21]. The integration of Ontologies like ELECTRICA in healthcare systems, particularly in the context of child abuse detection, provides significant benefits[21]. It ensures that data is semantically consistent, supports advanced analytics and reasoning, and facilitates better data sharing and interoperability across different systems. By leveraging these capabilities, the tool being developed can contribute to more accurate detection and prevention of child abuse, ultimately helping to protect vulnerable children.

2.4 Web Development Frameworks

2.4.1 React

React is a prevalent JavaScript library created by Facebook, with a primary emphasis on constructing user interfaces, especially within single-page applications. This library facilitates developers in assembling intricate UIs from diminutive and self-contained segments of code referred to as components. A significant benefit of React is its capacity to effectively update and render solely the components that have changed due to data modifications, as opposed to reloading the entire webpage. It's made possible by the virtual DOM, a lightweight version of the actual DOM. This allows React to make focused updates for better performance, ensuring that user experience is not compromised. React's component-oriented structure encourages reusability, thereby making the development process modular and more maintainable. It allows the developers to build custom components that include their proprietary logic and user interface, which can then be used in other parts of the application. This way, the development process is fast-tracked, and at the same time, it's easier to maintain and scale when the application grows. Besides, within the React ecosystem, various tools and libraries exist to further its power: from the very popular React Router that handles routing to libraries dealing with state management like Redux. Declarative features of React facilitate debugging and allow for deeper insights into the code, while a developer can describe the expected look of the UI at any given moment, and React will handle the updates accordingly[1].

2.4.2 Node.js

Node.js[24] is a powerful and flexible runtime environment based on Chrome's V8 JavaScript engine. It provides the ability to run JavaScript code on the server side, which originally ran only in the browser. By using a single language, JavaScript for both frontend and backend development, it simplifies the development process and code sharing between client and server becomes easier. Node.js has one of the biggest strong points: its event-driven, non-blocking I/O model, which makes it very fit for scalable network applications. Unlike other server-side environments, Node.js doesn't create multiple threads to handle requests concurrently; instead, Node.js works on a single thread with an event loop mechanism. All this enables Node.js to support thousands of concurrently active connections with relatively little overhead; thus, it's fit for real-time applications, such as chat services, online gaming, and collaboration tools where multiple users can interact in real time. Besides that, Node.js is backed by an enormous ecosystem of free open-source libraries and frameworks via npm-the Node Package Manager-that greatly speeds up development by providing prebuilt modules for almost every imaginable function, from database integration down to user authentication and API development. Node.js is very commonly used in conjunction with Express.js-a minimalist web framework that makes the creation of robust APIs and web applications easier. Express provides a thin layer of fundamental web application functionality and does not hide any Node.js features. It is designed to be unopinionated, allowing the developer to structure their applications however they see fit. Put together, React and Node.js provide an unbeatable combination to drive end-to-end web application development. React forms a dynamic and interactive UI, whereas Node.js efficiently takes up the responsibility of the back-end and responds to the client's request. Such a combination ensures that the web application is performant and scalable besides being easy to develop and maintain.

2.4.3 rdflib.js

A JavaScript library[4] specifically designed for working with RDF data in web applications. It provides a comprehensive interface for creating, parsing, querying, and serialising RDF graphs directly in JavaScript. This makes rdflib.js an essential tool for applications that need to handle RDF data efficiently within the browser or on the server side. The library supports SPARQL querying, allowing developers to execute complex queries on RDF data. It also allows the integration of RDF data with other Web technologies, allowing interoperability and sharing. Using rdflib.js, the tool easily converts form data to RDF format and allows interaction with RDF triple stores to keep the data nicely structured and stored according to standards of the semantic Web.

2.5 Querying Mechanisms

SPARQL is the standard query language for RDF data, enabling users to extract meaningful information from large datasets. The ability to perform complex queries on RDF data is critical for applications that require detailed data analysis and reporting. SPARQL's capability to query across diverse datasets makes it an indispensable tool for semantic web applications[19].

2.6 Predictive Models and Reasoning(Optional)

While the creation of predictive models goes beyond the scope of this project, it is necessary to understand how predictive models work to appropriately structure RDF data. In the context of child abuse detection, predictive models make use of machine learning algorithms that check patterns and predict the probability of abuse using historical data. The inclusion of reasoning capabilities within the developed tool can substantially enhance its functionality, allowing for the facilitation of complex inferencing based on the ontology, hence providing deeper insight into the data gathered.

2.7 Related Work

Several studies and projects have explored the use of semantic web technologies in healthcare and child protection. For example, the Child Abuse and Neglect Digital Repository (CAN-DR) project uses RDF to represent data on child abuse cases, demonstrating the potential of semantic technologies in this domain[13]. Similarly, the Ontology-Based Data Access (OBDA) approach has been applied in various healthcare applications to provide unified access to heterogeneous data sources[18]. These examples highlight the benefits of using semantic web technologies to improve data interoperability and support advanced analytics in child abuse detection and prevention.

2.8 Tools

In the development of this project, several tools and platforms were utilised to streamline the development process, facilitate collaboration, and ensure the integrity and functionality of the application.

2.8.1 GitHub

GitHub[9] is a web-based platform that offers version control and collaboration features. It is built on top of Git, an open-source version control system, and allows multiple developers to work on the same project simultaneously. GitHub's repository hosting service provides a centralised location for code storage, version tracking, and collaboration through pull requests and issue tracking. This makes it an essential tool for managing the project's source code, especially when working with distributed teams. In this project, GitHub was used to maintain the source code and track changes ensuring a seamless and organised development process.

2.8.2 VS Code

Visual Studio Code[25], developed by Microsoft, is a free, open-source code editor that supports a wide range of programming languages and frameworks. It offers a powerful development environment with features such as syntax highlighting, code completion, debugging, and integrated Git control. VS Code is highly extensible, allowing developers to install various extensions to enhance its functionality, such as linters, debuggers, and theming options. For this project, VS Code was the primary development environment, providing an efficient and customised interface for writing, testing, and debugging the code.

2.8.3 SHACL Playground

The SHACL Playground[22] is an online tool that allows developers to validate RDF data against SHACL shapes. SHACL (Shapes Constraint Language) is a W3C standard for describing and validating RDF graphs. The SHACL Playground provides an easy way to input RDF data and SHACL shapes, allows validation to be carried out, and the results of such validation to be studied. This tool was particularly useful to check that the RDF data generated by the application was conforming to the specified shapes and met the semantic and structural requirements set by the ELECTRICA ontology. The usability and direct feedback provided by the SHACL Playground made it an indispensable tool during the whole development and validation process.

Chapter 3

Requirements and Analysis

This chapter covers the system analysis of the RDF-based form applied to recording fractures in suspected cases of child abuse. This consists of user stories and requirements analysis.

3.1 User Stories

User stories outline the desired outcomes from the perspective of the target user, allowing developers to focus on solving real user problems. The following presents the user stories for the primary stakeholder of this web application:

1. As a healthcare professional, I want to record data on fractures and related injuries in children so that I can accurately document suspected cases of child abuse.
2. As a healthcare professional, I want to generate RDF data from form inputs so that the data can be used in predictive models to assess child abuse risk.
3. As a healthcare professional, I want to retrieve stored RDF data on child injuries so that I can analyse patterns and contribute to child welfare research.

3.2 Requirements Analysis

Requirements are divided into functional and non-functional categories, each playing a crucial role in the project's success.

3.2.1 Functional Requirements

The functional requirements are divided into categories based on the different aspects of the web application:

SL No.	Requirements	Importance
1	The system must provide a web-based form for recording data on family history, fractures and related injuries in children.	Mandatory
2	The system must convert form inputs into RDF format, adhering to the ELECTRICA ontology.	Mandatory
3	The system must store RDF data in a triplestore for easy retrieval and querying.	Mandatory
4	The system must provide a SPARQL-based querying mechanism for retrieving specific RDF data.	Mandatory
5	The system must integrate with the ELECTRICA ontology to ensure semantic accuracy.	Mandatory
6	The system may incorporate reasoning capabilities to perform inference based on the ontology.	Optional

Table 3.1: Functional Requirements.

3.2.2 Non - Functional Requirements

Non-functional requirements focus on the system's behaviour and quality, ensuring it meets user expectations and standards: Performance Requirements: The web application should respond to user action within an acceptable amount of time that does not noticeably pause when submitting forms or SPARQL queries. Usability requirements: The user interface should be intuitive, responsive, and accessible, with clearly labeled fields and controls. Extensibility Requirements: The system architecture should be modular, permitting future enhancements without any major restructuring of the codebase. Maintainability Requirements: The software should be well-documented and follow industry-wide best practices to enable easy maintenance and upgrades. Reliability Requirements: This application should ensure data integrity and availability, with adequate error-handling mechanisms incorporated. Compliance: The application should conform to the relevant data protection regulations and accessibility standards.

3.2.3 Platform Requirements

Platform requirements ensure that the web application is accessible and performs consistently across various environments and devices. The key platform requirements for this project are as follows: **Browser Compatibility**: The web application must be compatible with the latest versions of all major web browsers, including Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari. The application should degrade gracefully in older versions of these browsers, ensuring that core functionality remains accessible even if some advanced features are not supported. **Device Compatibility**: The web application must be fully responsive, adapting to different screen sizes and orientations. It should be usable on a variety of devices, including desktop computers, laptops, tablets, and smartphones. Touchscreen functionality must be supported, ensuring that users on mobile and tablet devices can interact with the application effectively. **Operating System Independence**: The application must function consistently across different operating systems, including Windows, macOS, Linux, iOS, and Android. No platform-specific features or dependencies should be used that could prevent the application from running on any of these systems.

3.3 Ethics

The data used in the project has been robustly anonymised before being accessed by the research team, ensuring that no personal or medical data can be re-identified by any party. This approach allows the project to develop a semantically enriched tool for collecting and storing data on fractures and related injuries without compromising the privacy of individuals who originally provided the data. Since the project primarily involves the development of a tool, there will be no primary data collection involving human participants. The use of anonymised secondary data mitigates any potential ethical concerns, ensuring that the project complies with all relevant ethical standards. Additionally, any external libraries or third-party tools utilised in the project are open-source and free for public use, with their application limited to analytical and design purposes only. This careful consideration of ethical requirements ensures that the project is conducted in a responsible and compliant manner.

3.4 Evaluation

The effectiveness and reliability of the developed system are assessed through a comprehensive evaluation process employing both usability and data validation methodologies. This ensures that the system not only meets the functional requirements but also provides a seamless and intuitive user experience while maintaining high data integrity and compliance with established Ontologies. The evaluation is conducted using the following criteria: **Jakob Nielsen's Usability Heuristics** [15]: This set of principles provides a framework for evaluating the user interface and interaction design of the system. The heuristics assess various aspects such as system visibility, match between system and real-world conventions, user control and freedom, consistency and standards, and recognition rather than recall. Applying these heuristics helps identify usability issues and guides improvements to enhance user satisfaction and efficiency. **Shapes Constraint Language (SHACL) for RDF Data Validation**[26]: SHACL is used to validate the RDF (Resource Description Framework) data generated and processed by the system. By defining specific shapes and constraints aligned with the ELECTRICA ontology[3], SHACL ensures that the data adheres to the required semantic structure and integrity. The validation process is facilitated using tools like SHACL Playground [22], enabling immediate feedback and verification of data compliance. Through these evaluation methods, the system is rigorously tested to ensure it delivers accurate, reliable, and user-friendly functionality for recording and analysing fracture data in suspected child abuse cases. The combined approach addresses both the human-centred aspects of system interaction and the technical correctness of data handling, thereby supporting the overarching goal of enhancing child protection efforts through effective data management.

Chapter 4

Planning

4.1 Risk Analysis

Risk analysis is a fundamental aspect of project management, crucial for anticipating potential challenges and preparing strategies to mitigate their impact on the project's success. In project management, risk management involves identifying, analysing, planning, and monitoring risks throughout the project lifeThe project plan outlines the timeline and key tasks necessary for the successful completion of the development and implementation of the semantically enriched tool for collecting and storing data on fractures and related injuries in children. The plan is designed to ensure that each phase of the project is completed in a structured and timely manner, with each task building upon the previous one to contribute to the overall goal. The Gantt chart(Figure 4.1) provided illustrates the timeline over a 13-week period, with each task scheduled to ensure the seamless progression of the project. The project begins with the setup of the foundational components such as the installation of the Apache Jena Fuseki Database and setting up the Express JS server. This is followed by the critical task of converting JSON input into RDF data using the ELECTRICA ontology, which is key to ensuring semantic consistency within the data. Subsequent weeks focus on implementing SPARQL queries for both data insertion and retrieval, which are essential for managing and accessing the RDF data. The development then moves towards creating the front-end components, including input screens for Family History and Fractures, as well as a dashboard for visualising the data. Towards the end of the timeline, the project shifts focus to refining the application by adding exception handling, improving API communication, and conducting evaluation and UI testing to ensure that the tool meets the expected standards of functionality and usability. Each task is aligned to ensure that the project progresses smoothly and is completed within the set timeline.cycle[23]. This proactive approach helps in enhancing the project's resilience and adaptability, ensuring that unforeseen events are handled effectively. For the development of a semantically enriched tool for recording and analyzing data related to suspected child abuse cases, particularly focusing on fractures, various risks have been identified. These risks include technical challenges such as data integration issues and external factors like changes in legal or ethical standards. Each identified risk is evaluated based on its Likelihood and Severity, both assessed on a scale of 1 to 5. This evaluation prioritises risk management efforts, ensuring that the most critical risks are effectively mitigated to safeguard the project's objectives. (Likelihood (1 - 5): 1 = very unlikely to 5 = very likely. Severity(1 - 5): 1 = Negligible effects, to 4 = Catastrophic.)

4.2 Project Plan

The project plan outlines the timeline and key tasks necessary for the successful completion of the development and implementation of the semantically enriched tool for collecting and storing data on fractures and related injuries in children. The plan is designed to ensure that each phase of the project is completed in a structured and timely manner, with each task building upon the previous one to contribute to the overall goal. The Gantt chart(Figure 4.1) provided illustrates the timeline over a 13-week period, with each task scheduled to ensure the seamless progression of the project. The project begins with the setup of the foundational components such as the installation of the Apache Jena Fuseki Database and setting up the Express JS server. This is followed by the critical task of converting JSON input into RDF data using the ELECTRICA ontology, which is key to ensuring semantic consistency within the data. Subsequent weeks focus on implementing SPARQL queries for both data insertion and retrieval, which are essential for managing and accessing the RDF data. The development then moves towards creating the front-end components, including input screens for Family History and Fractures, as well as a dashboard for visualising the data. Towards the end of the timeline, the project shifts focus to refining the application by adding exception handling, improving API communication, and conducting evaluation and UI testing to ensure that the tool meets the expected standards of functionality and usability. Each task is aligned to ensure that the project progresses smoothly and is completed within the set timeline.

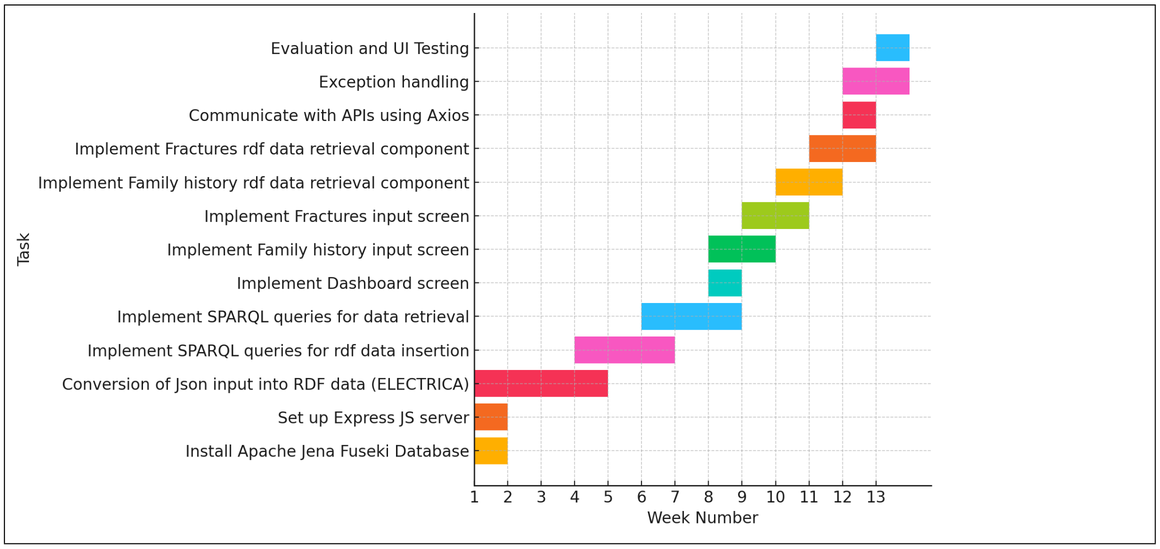


Figure 4.1: Gantt chart of Planning.

Risk	Impact	Likelihood	Severity	Action
Underestimation of Functional Requirements	Failure to complete the development plan within the allotted time.	3	5	Maintain regular communication with the mentor to revisit the requirements analysis and prioritise essential features.
Loss of Source code	Not able to deliver the project on time	1	5	Use version controlling tools like GitHub.
Semantic Inconsistency in RDF Data	Incorrect data formatting can lead to faulty predictions in child abuse models.	3	4	Validate RDF data using SHACL and regularly update ontology mappings.
Server Downtime or Data Loss	Inaccessibility of the web application could hinder data collection efforts.	2	4	Implement regular backups, redundant servers, and monitor system health continuously.
User Interface Design Flaws	Poor usability could lead to incorrect data entry by healthcare professionals.	4	3	Conduct usability testing using Jakob Nielsen's heuristics and refine UI based on feedback.
Inadequate Knowledge of Ontology	Lack of understanding could lead to improper use of the ELECTRICA ontology.	3	3	Read documentations on ontology usage for developers.
Dependency on External Libraries	If third-party libraries are deprecated, it could disrupt project progress.	3	4	Monitor the status of external libraries and have alternative solutions in place.

Table 4.1: Risk Assessment.

Chapter 5

System Design

This chapter provides an in-depth look at the design of the web-based tool aimed at collecting and storing data on fractures and related injuries in suspected child abuse cases. The tool leverages semantic web technologies to ensure data interoperability and advanced reasoning capabilities. The system's architecture is carefully crafted to support scalability, modularity, and seamless integration with the ELECTRICA ontology, ensuring both current functionality and future extensibility. **Mono repo** : The Rdf-Fractures project utilises a mono repo structure, where both the front-end and back-end code bases are housed in a single repository. This approach simplifies the management and development of the project by ensuring consistent versioning, streamlined dependency management, and a unified development workflow. The mono repo contains key directories such as frontend / for React-based front-end code, backend / for Node.js and Express.js back-end operations, and public/ for static assets. Centralised configuration files like package.json manage dependencies across the entire project. The mono repo structure offers several benefits, including easier refactoring, code sharing, and integration between different parts of the project. It also facilitates more efficient Continuous Integration and Continuous Deployment (CI/CD) processes, where both front-end and back-end changes can be tested and deployed simultaneously. To manage the complexity that comes with a mono repo, the project employs modular design principles, automation through CI/CD pipelines, and comprehensive documentation.

5.1 System Architecture

The system is composed of a multi-tier architecture that separates concerns across the front-end, back-end, and data management layers. This design as shown in Figure 5.1 facilitates clear delineation between user interaction, business logic, and data persistence, allowing each component to be developed, maintained, and scaled independently. **Front-End**: Built using React, the front-end handles the user interface and user experience aspects, providing an intuitive and responsive platform for data entry and visualisation. **Back-End**: Implemented with Node.js and Express, the back-end serves as the intermediary between the front-end and the database, handling API requests, data validation, and RDF conversion.

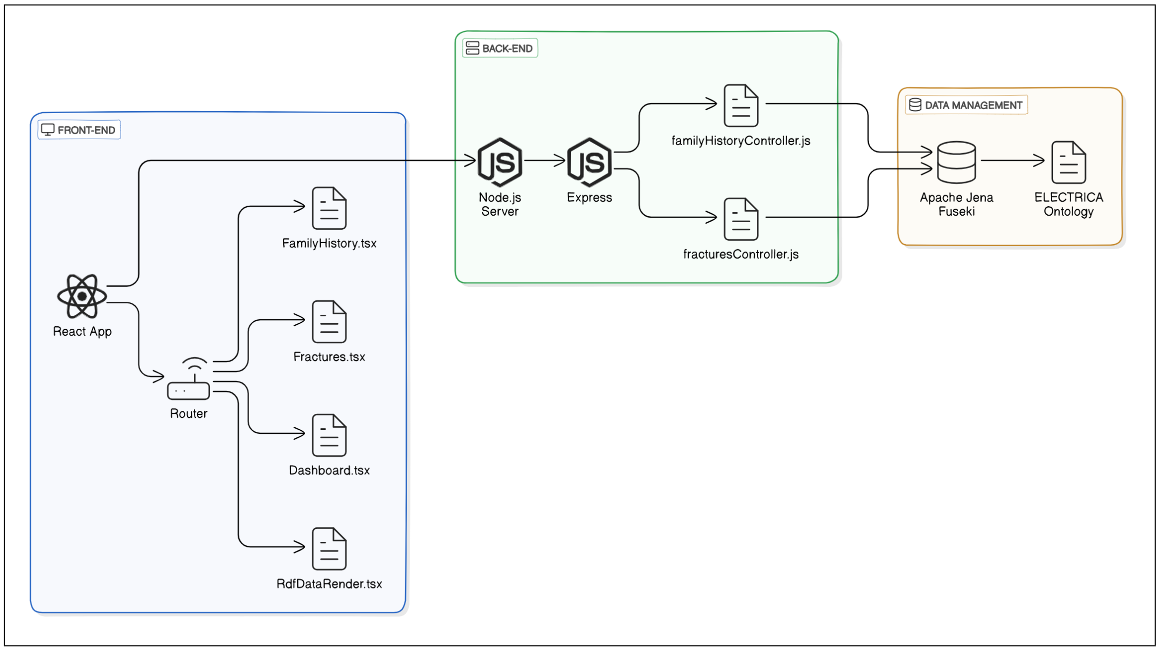


Figure 5.1: System Architecture.

Data Management: The Apache Jena Fuseki server is employed for RDF data storage, enabling efficient querying and reasoning over the data using SPARQL and SHACL.

5.2 Front-End Design

The front-end of the Rdf-Fractures application is developed using React, a powerful and flexible JavaScript library for building user interfaces. The design follows a component-based architecture, which promotes modularity, reusability, and ease of maintenance. Each part of the user interface is broken down into reusable components, allowing for consistent and efficient development. The design also integrates React Router for managing navigation within the application, ensuring a smooth user experience. The React-based front-end is organised into a series of components, each responsible for rendering a specific part of the user interface. These components include both functional components and class-based components, depending on the requirements of each part of the application. Here's a breakdown of the key components and their functionalities:

1. **App.tsx:** The root component of the application, which encapsulates all other components and sets up the application's global context, including routing. It Initialises the main layout, includes the router configuration, and handles global state management using React's built-in state management hooks.

2. **Dashboard.tsx:** Acts as the main landing page for users, providing an overview of the application's functionalities. It Displays key metrics, navigation options, and links to

various sections of the application. The dashboard serves as the entry point for users to access different tools and data views.

3. **FamilyHistory.tsx:** This component provides a form interface for users to input family history data. It handles the state of the form, validation, and submission to the backend.
4. **FamilyHistoryTable.tsx:** Displays the family history records in a tabular format. It includes features like sorting, filtering, and pagination to manage and view large datasets efficiently.
5. **Fractures.tsx:** Similar to the Family History component, this handles the input of fractures data, ensuring that the data conforms to the required format before being submitted to the backend.
6. **FracturesTable.tsx:** Renders a table of fractures data, allowing users to view, sort, and filter through the records.
7. **ChildDetailsCapture.tsx:** Provides an interface for capturing detailed information about a child, which may include demographic data, medical history, and other relevant details. This data is critical for context in evaluating the risk of abuse.
8. **RdfDataRender.tsx:** This component is responsible for displaying RDF data retrieved from the backend. It Implements SPARQL queries to fetch RDF data and renders it in a human-readable format.

5.2.1 Routing with React router

React Router is employed in the Rdf-Fractures application to manage navigation between different views. It allows the application to remain a single-page application (SPA) while providing the capability to navigate between different pages or components without reloading the browser. The router is configured in the App.tsx file, where routes are defined for each major component as shown in the Figure 5.2 React Router.

Dynamic Routing: The router setup allows for dynamic routing, where certain routes can include parameters to pass specific data to components, enhancing the flexibility of the navigation structure. **Protected Routes**: If the application includes authentication, React Router can also manage protected routes, ensuring that only authenticated users can access certain components.

5.2.2 State Management

State management in the Rdf-Fractures application is handled using React's useState hook, which provides a simple and effective way to manage local state within functional components. This approach aligns with the modern React paradigm of functional components and hooks, ensuring that the application is both performant and easy to maintain. Local State Management: Each component that requires stateful logic leverages the useState hook to manage its state.

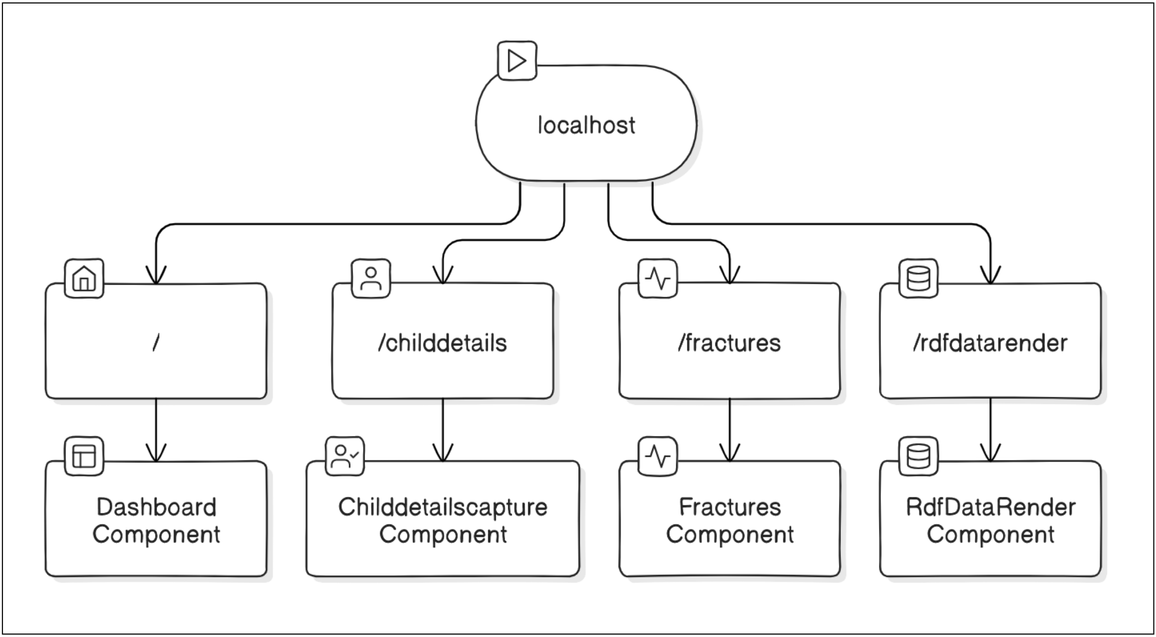


Figure 5.2: React Router

For example, form components like FamilyHistory.tsx and Fractures.tsx use useState to manage form inputs, validation states, and submission handling.

const [formData, setFormData] = useState(initialFormState);

The above line initialises the state with an initialFormState object and provides a function setFormData to update the state as the user interacts with the form.

Global State Considerations: Although the application primarily relies on local state management, it can integrate Context API for managing global state if needed. This would be useful for sharing state across multiple components, such as user authentication status or shared configuration settings.

State Updates and Re-renders: React's useState hook ensures that the component re-renders whenever the state is updated, keeping the UI in sync with the current application state.

5.2.3 API Integration with Axios

Axios is used to handle HTTP requests between the front-end and back-end. Each form component (e.g., FamilyHistory.tsx, Fractures.tsx) includes logic to submit data to the server and handle the server's response.

Error Handling: Axios interceptors can be implemented to globally handle errors, such as network issues or server errors, ensuring that users are informed of any issues with their requests.

Asynchronous Data Fetching: Components like RdfDataRender.tsx use Axios to fetch RDF data asynchronously, ensuring that the UI remains responsive while waiting for data.

5.2.4 Responsive Design using Tailwind CSS and DaisyUI

The application's responsive design and consistent styling are achieved using Tailwind CSS and DaisyUI.

Tailwind CSS: Tailwind CSS is a utility-first CSS framework that provides a vast array of classes to build responsive designs without leaving the HTML file. This approach enables rapid prototyping and a high degree of customization while maintaining a consistent design language across the application.

Responsive Utilities: Tailwind CSS includes built-in responsive utilities that automatically adjust the layout, padding, margins, and other design elements based on the screen size. This ensures that the application looks good and functions well on any device, whether it's a desktop, tablet, or mobile phone. Customization: Tailwind CSS allows for deep customization, enabling developers to define custom themes, colour schemes, and spacing that align with the application's branding.

DaisyUI: DaisyUI is a plugin for Tailwind CSS that provides a collection of pre-designed components, such as buttons, forms, cards, and modals. It simplifies the development process by offering ready-to-use, beautifully styled components that are fully responsive and consistent with the overall design language.

Component Library: DaisyUI's component library covers a wide range of UI elements, reducing the need to design and style these elements from scratch. This consistency in design contributes to a cohesive user experience throughout the application.

Theme Support: DaisyUI supports multiple themes, allowing the application to adapt to different colour schemes and design preferences effortlessly. Integration with Tailwind: Since DaisyUI is built on top of Tailwind CSS, it seamlessly integrates with the existing utility classes, making it easy to extend or customise components as needed.

5.3 Back-End Design

The backend of the Rdf-Fractures project is located in the backend/ directory of the monorepo, it is designed to provide a robust, scalable, and maintainable infrastructure for handling server-side operations. It leverages the power of Node.js and Express.js to create an efficient API that serves the front-end application and interacts with the RDF data store.

5.3.1 Back-End Directories and their Roles

controllers/: This directory houses the business logic for processing incoming requests. Each controller is focused on a specific domain within the application, such as family history (familyHistoryController.js) and fractures (fracturesController.js). The controllers receive requests from the client, interact with the necessary services or models, and return a response. For example, when a user submits data through a form, the corresponding controller validates the input, processes the data (e.g., converts it into RDF format), and saves it to the RDF data store. **routes/:** This directory defines the API endpoints that the front-end interacts with. Each route corresponds to a specific URL path and HTTP method (GET, POST, PUT, DELETE). The routes act as intermediaries between the client's requests and the controllers. For instance, the /api/familyhistory endpoint defined in familyHistoryRoutes.js directs requests to the appropriate controller function that handles the logic for family history-related operations.

5.3.2 Back-End Core Components and Technologies

Express.js: Express.js serves as the web framework that powers the backend API. It simplifies the creation of routes, handles middleware, and provides utilities for managing HTTP requests and responses. The backend uses Express.js to define its RESTful API endpoints, handling CRUD operations (Create, Read, Update, Delete) necessary for managing the application's data. Express allows for defining these routes in a clean and modular way, making the backend scalable and easy to maintain. **Node.js:** Node.js provides the runtime environment for executing JavaScript on the server side. It is known for its event-driven, non-blocking I/O model, which makes it ideal for applications that require high concurrency and real-time data processing. Node.js is used to handle the backend logic, manage asynchronous operations, and serve the API endpoints. Its ability to handle multiple requests simultaneously with minimal overhead is crucial for the application's performance, especially when dealing with multiple users and large datasets.

5.3.3 Data Management and Interaction with RDF

RDF Data Handling: The backend is designed to interact with an RDF data store, which holds semantically structured data crucial for the application's operation. The backend executes SPARQL queries to retrieve, insert, or update data in the RDF store. For instance, when a user submits a new fracture record, the backend converts this information into RDF triples and stores it in the database. Similarly, when querying for records, SPARQL is used to fetch and return the data in a structured format. **SPARQL Query Execution:** SPARQL serves as the query language for interacting with RDF data. It is essential for retrieving and manipulating the data stored in the RDF triple store. The backend uses SPARQL to execute queries that retrieve relevant data based on user requests. For example, a SPARQL query might be used to find all records of fractures that match certain criteria, such as those occurring in children under a certain age.

5.3.4 Security and Error Handling

Security: Security is a critical aspect of the backend, ensuring that the application is protected against common vulnerabilities such as SQL injection, cross-site scripting (XSS), and unauthorised access. The backend employs various security measures, such as input validation, to sanitise incoming data, and CORS (Cross-Origin Resource Sharing) policies to control access from external domains. If authentication is required, the backend would handle user sessions or tokens to manage access control. **Error Handling:** Proper error handling is essential for maintaining application stability and providing meaningful feedback to users. The backend includes middleware to catch and handle errors throughout the application. This involves logging the error details for further investigation and returning a standardised error message to the client to inform them of what went wrong.

5.3.5 Modularity, Scalability, and Maintainability

The backend is designed with modularity in mind, allowing different parts of the application to be developed, tested, and maintained independently. Each part of the backend (e.g., routes, controllers, services) is encapsulated in its own module. This makes it easier to add new features or modify existing ones without affecting the rest of the application.

5.4 Data Management and Semantic Integration

The Rdf-Fractures project utilises a semantically enriched database design centred around RDF (Resource Description Framework) technology, managed through Apache Jena Fuseki. This setup is crucial for handling the intricate data requirements of the application, particularly in the context of recording and analyzing fractures related to suspected child abuse cases. Below is a detailed exploration of the database structure, interaction mechanisms, and the role of master datasets in ensuring data consistency and integrity. At the core of the database is the RDF triple store, which stores all data in the form of triples: subject-predicate-object. This structure is integral to the Semantic Web, allowing for the flexible and interconnected storage of data. In the Rdf-Fractures project, data such as patient details, medical histories, and specific fracture records are stored as RDF triples. The graph-based structure facilitates the linking of related data points, making it possible to represent complex relationships and hierarchies inherent in medical data. The database integrates with the ELECTRICA ontology, a specialised framework that ensures all stored data adheres to a consistent semantic structure. By utilising the ELECTRICA ontology, the database maintains semantic consistency, enabling accurate data analysis and facilitating interoperability with other systems in the healthcare domain. The ontology defines the relationships between different concepts, such as the linkage between a patient and their medical conditions, ensuring that data is both meaningful and actionable.

5.4.1 Dataset Configuration and Management

objectMaster: The objectMaster dataset serves as a reference for all objects within the system, ensuring that data related to entities such as medical conditions, body parts, and healthcare providers are standardised and consistent across the application. This dataset acts as a controlled vocabulary or lookup table that the main dataset references, ensuring that the objects used in the triples are consistent and semantically correct.

predicateMaster: The predicateMaster dataset defines the relationships (predicates) between different entities in the system. This includes relationships like hasFracture which are crucial for maintaining the integrity and meaning of the data. The predicates in this dataset are used to establish connections between subjects and objects within the main dataset. By centralising these predicates, the system ensures that all relationships are defined consistently, preventing data corruption and redundancy.

myRdfDataset: The myRdfDataset is the primary dataset where all the actual data related to patients, their medical histories, and associated fractures are stored. This dataset is where the majority of the application's data is stored and queried. It references the objectMaster and predicateMaster datasets to ensure that all data is consistent and adheres to the predefined ontological structure. The myRdfDataset is crucial for the day-to-day operations of the application, as it holds all the dynamic data that the application needs to function.

5.4.2 Data Interaction Mechanisms

SPARQL Queries: SPARQL, the query language for RDF, is used to interact with the datasets. It allows for complex querying, retrieval, and manipulation of data stored in the RDF triple store. SPARQL queries are executed against the myRdfDataset to retrieve specific data points, such as all fractures associated with a particular patient or specific injury patterns. The query language's flexibility supports dynamic data interactions, making it an essential tool for both developers and end-users who need to query the data.

Data Insertion: The system allows for the dynamic insertion of RDF data, ensuring that the dataset remains current and accurate. New data entries, such as newly recorded fractures, are converted into RDF triples and inserted into the myRdfDataset. Updates to existing data are handled similarly, ensuring that the information remains accurate and up-to-date. These operations are managed through SPARQL Update queries, providing precise control over the dataset.

Fuseki Server: Apache Jena Fuseki acts as the server interface for the RDF store, providing endpoints for querying, updating, and managing the datasets. The Fuseki server hosts the objectMaster, predicateMaster, and myRdfDataset, each accessible via dedicated HTTP endpoints. These endpoints allow for seamless interaction with the datasets over the network, supporting both application functionality and administrative tasks such as dataset management and query execution.

Web-Based Management Console: The Fuseki server provides a web-based management console accessible via localhost:3030, which allows developers to run queries, inspect data, and manage the datasets. Through this console, developers can directly interact with the datasets, perform maintenance tasks, and monitor the data's integrity. This interface is crucial for the ongoing management and scaling of the application.

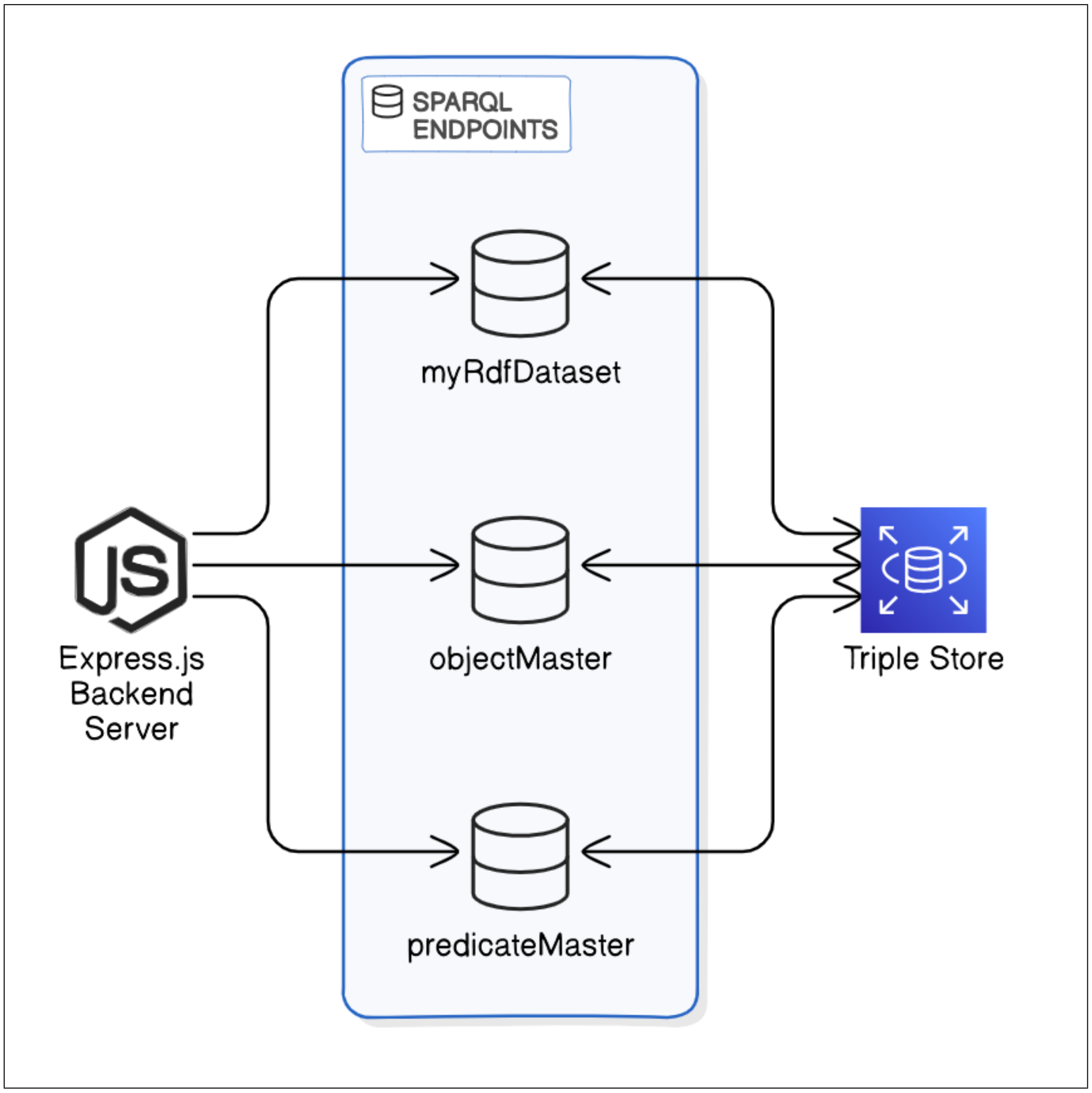


Figure 5.3: Triple Store.

5.5 User Flow

The user flow diagram Figure 5.4 illustrates the process that a healthcare professional follows when using the web-based tool designed for recording fractures and related injuries in children, particularly in suspected child abuse cases. The process begins when the healthcare professional accesses the web-based form. Initially, they input the child's family details and medical history into the form. This step is crucial as it provides context and ensures that all relevant background information is included. Following the input of family details, the professional enters specific information about the fractures, such as the type and location of the fracture, along with any other pertinent details. The system then validates the entered data for correctness and completeness, ensuring that all required fields are properly filled out and that the data adheres to the expected format. This validation step reduces errors and enhances the reliability of the collected information. Once the data is validated, it is transformed into RDF (Resource Description Framework) format using URIs (Uniform Resource Identifiers) specific to the ELECTRICA ontology. This transformation is essential for standardising the data according to the ontology's structure, making it interoperable with other systems. After transformation, the RDF data is submitted to a triple store, a specialised database for storing and querying RDF data. This storage ensures that the data is preserved in a structured format that supports advanced querying and analysis. Subsequently, the healthcare professional can query the stored RDF data using SPARQL, a query language designed for retrieving and manipulating RDF data. This allows for efficient retrieval of specific information from the database. The retrieved RDF data is then converted back into a human-readable format, ensuring that the structured data can be easily understood and used by healthcare professionals. Finally, the processed and formatted data is displayed to the healthcare professional, providing them with the necessary information to assess the situation and make informed decisions, thereby supporting efforts to detect and prevent child abuse effectively.

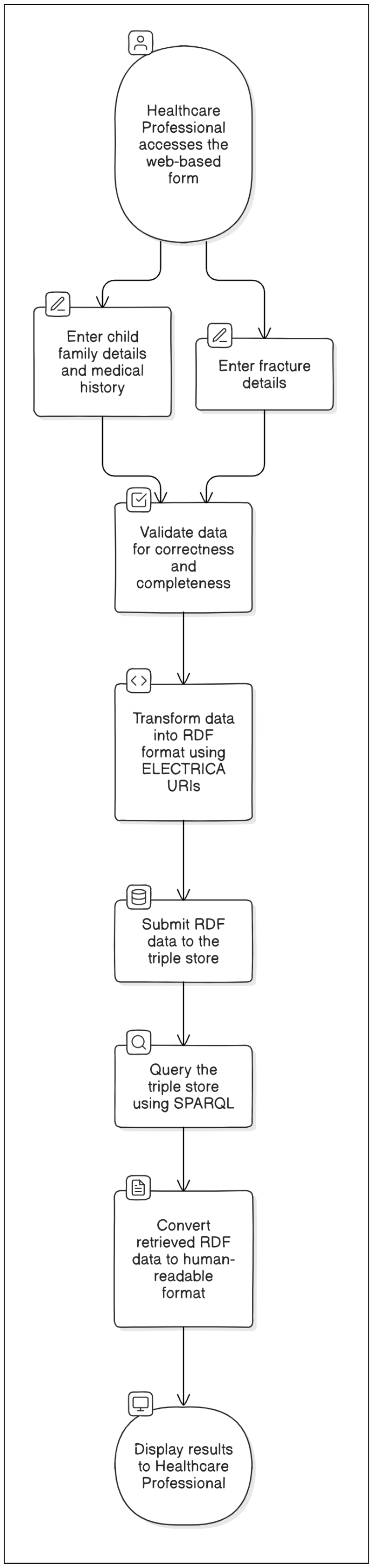

Healthcare Professional accesses the web-based form
Enter child family details and medical history
Enter fracture details
Validate data for correctness and completeness
Transform data into RDF format using ELECTRICA URIs
Submit RDF data to the triple store
Query the triple store using SPARQL
Convert retrieved RDF data to human-readable format
Display results to Healthcare Professional

Figure 5.4: User flow.

5.6 Key Design Considerations

Scalability: The system is designed to scale horizontally by adding more instances of the front-end, back-end, and data management layers as needed.

Modularity: Each component of the system is modular, making it easier to maintain and extend the system with new features or enhancements. Ontology Integration: The integration with the ELECTRICA ontology ensures that all data collected is semantically consistent, facilitating better data analysis and sharing.

Vite and Reconciliation: To optimise the performance and development workflow of the RDF Fractures web application, the project leverages two key technologies: Vite and React's reconciliation process. Vite, a modern build tool, significantly enhances the speed of development by serving code as native ES modules and providing fast Hot Module Replacement (HMR). This allows developers to see changes in real-time without reloading the entire application, which is particularly beneficial for handling complex data structures and frequent updates in the application [10]. React's reconciliation process further contributes to performance optimization by efficiently updating the DOM. Instead of re-rendering the entire UI whenever changes occur, React uses a Virtual DOM to track differences and only updates the parts of the actual DOM that have changed. This minimises the overhead associated with direct DOM manipulation and ensures that the application remains responsive, even during high-frequency data updates [1]. Together, Vite and React's reconciliation process provide a robust foundation for the RDF Fractures application. They enable fast development cycles, efficient real-time updates, and a smooth user experience, which are crucial for managing and displaying complex RDF data related to fractures in suspected child abuse cases. This combination ensures that the application can handle the demands of real-time data processing while maintaining high performance and usability.

Chapter 6

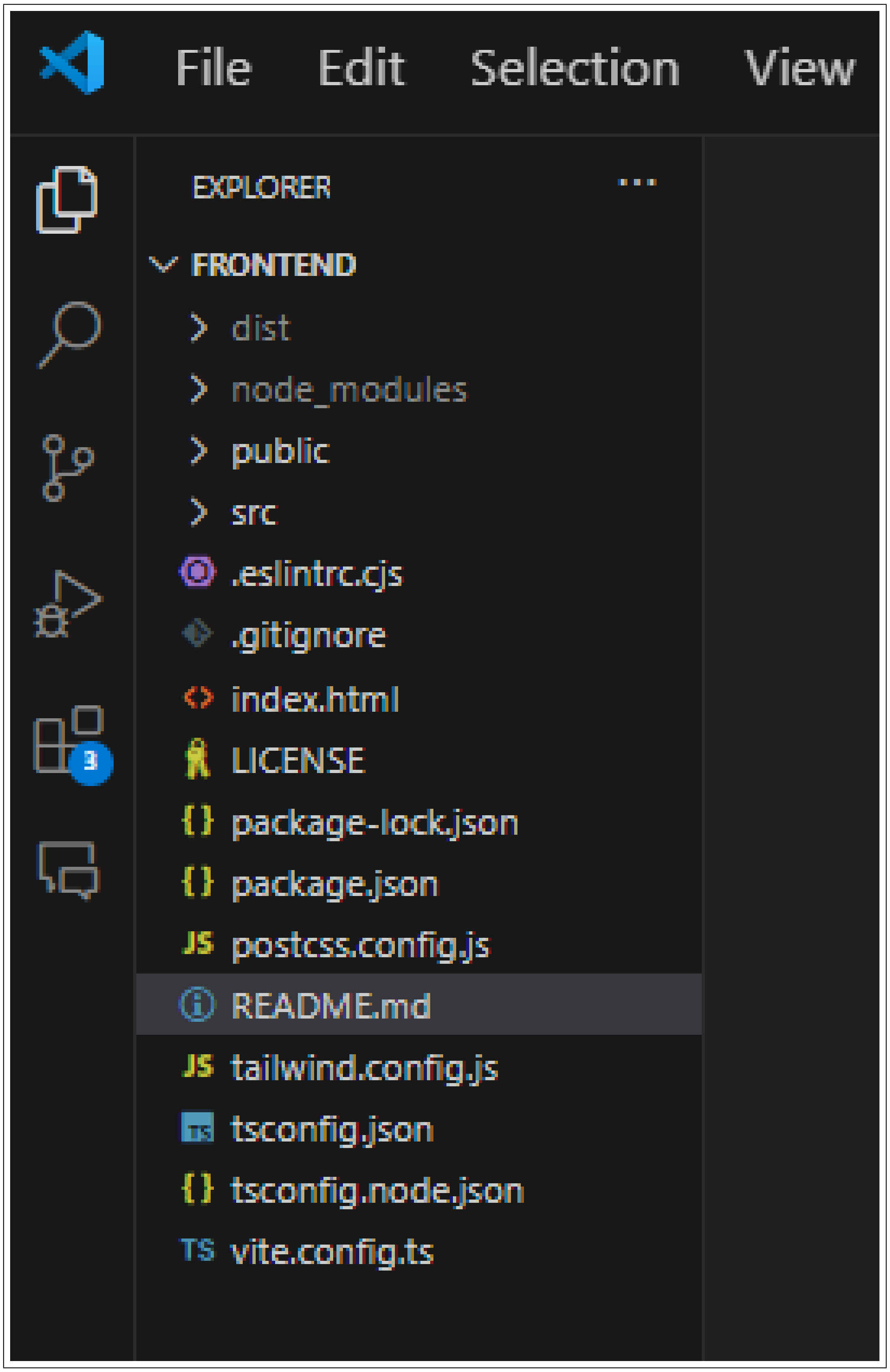
Implementation

This section will detail the implementation process by first explaining how the front-end and back-end servers are utilised to implement key features, it will then present the project outcomes in the form of a user manual.

6.1 File Directory

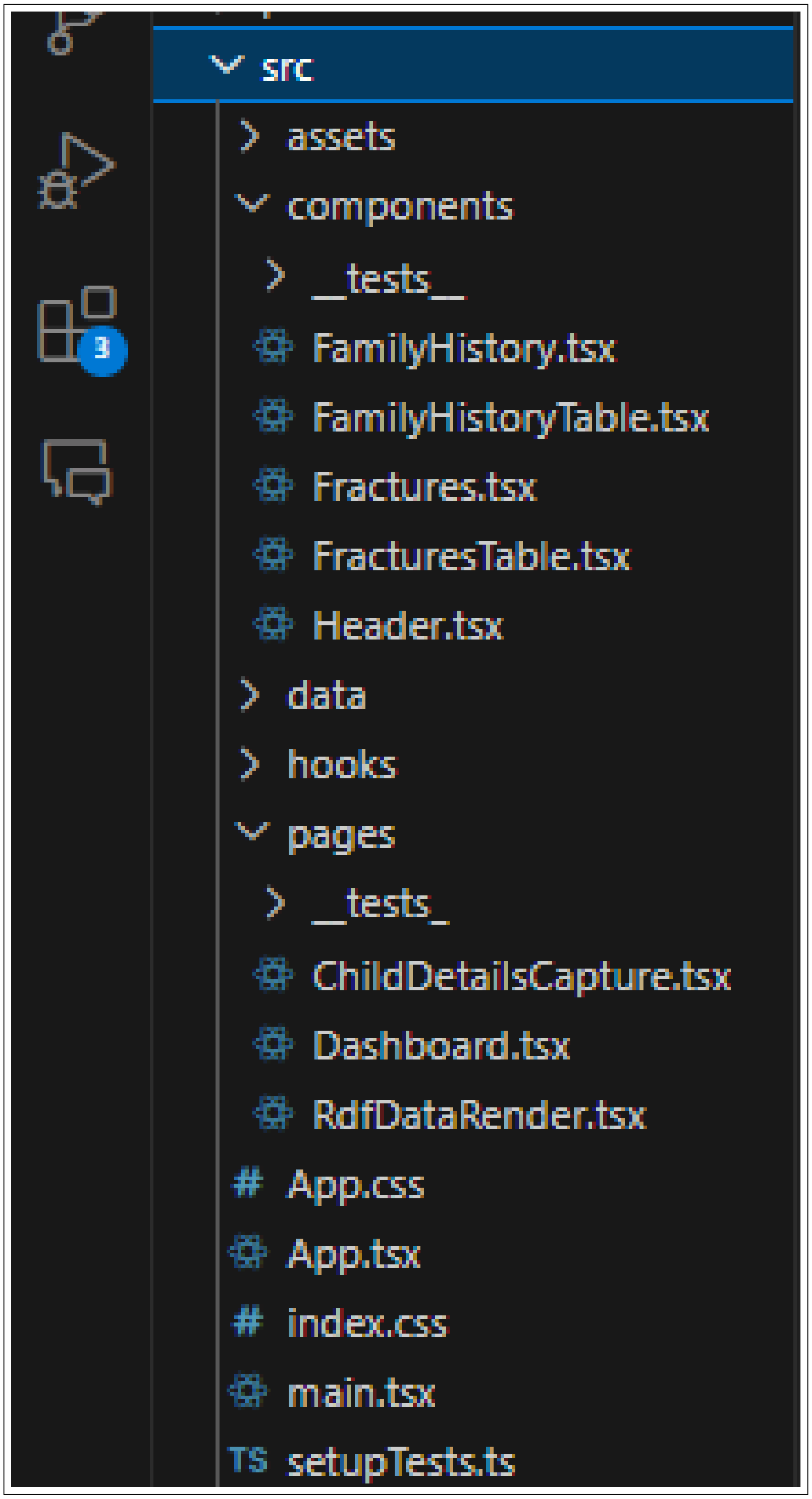
The project is structured into two main parts: the backend and frontend directories, each fulfilling distinct roles in the application architecture. The backend directory is responsible for handling server-side functionality. This includes managing API routes, controllers, models, and utilities. With Node.js and Express.js as the core technologies, the backend is designed to handle HTTP requests, perform data validation, interact with the database, and process business logic. The controllers (e.g., familyHistoryController.js, fracturesController.js) define how incoming requests are handled, while routes (e.g., familyHistoryRoutes.js, fracturesRoutes.js) map specific API endpoints to controller functions. The backend is also equipped with a .env file for environment variables, ensuring secure configuration, and it integrates middleware for error handling and logging. The frontend directory contains the client-side interface built using React with Vite.js for faster development and build processes. Components like FamilyHistory.tsx, Fractures.tsx, and Dashboard.tsx are modular, reusable pieces of the UI, responsible for rendering data and interacting with backend services via API calls. Hooks manage state and side effects, while utility files handle common data and configuration across the application. The frontend uses TailwindCSS for styling, offering a clean and responsive design, while the RdfDataRender.tsx component processes and displays data from the backend in a readable format for the end user.

6.1.1 React Application



```
FRONTEND
  dist
  node_modules
  public
  src
  .eslintrc.cjs
  .gitignore
  index.html
  LICENSE
  package-lock.json
  package.json
  postcss.config.js
  README.md
  tailwind.config.js
  tsconfig.json
  tsconfig.node.json
  vite.config.ts
```

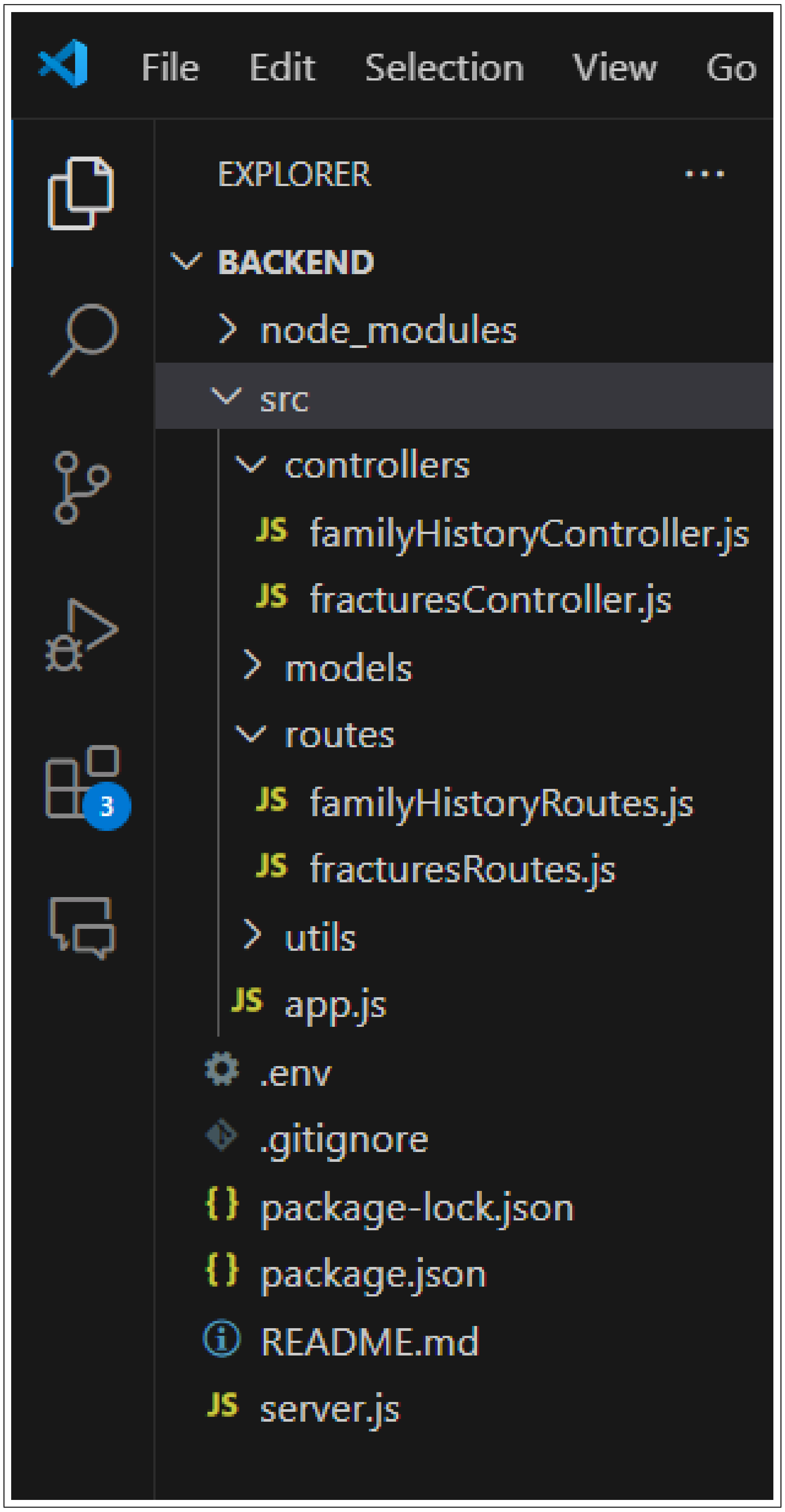
Figure 6.1: React Project Directory.



```
src
  assets
  components
    __tests__
    FamilyHistory.tsx
    FamilyHistoryTable.tsx
    Fractures.tsx
    FracturesTable.tsx
    Header.tsx
  data
  hooks
  pages
    __tests__
    ChildDetailsCapture.tsx
    Dashboard.tsx
    RdfDataRender.tsx
  App.css
  App.tsx
  index.css
  main.tsx
  setupTests.ts
```

Figure 6.2: React Project Source Directory.

6.1.2 Node Server



```
BACKEND
  > node_modules
  v src
    v controllers
        familyHistoryController.js
        fracturesController.js
    > models
    v routes
        familyHistoryRoutes.js
        fracturesRoutes.js
    > utils
      app.js
    .env
    .gitignore
    package-lock.json
    package.json
    README.md
    server.js
```

Figure 6.3: Node Project Directory.

The frontend is responsible for rendering the user interface and managing client-side interactions. Built with React and Vite.js, the frontend leverages reusable components to build a dynamic, modular UI. The components folder contains key elements such as FamilyHistory.tsx, Fractures.tsx, and Dashboard.tsx, which correspond to different sections of the application. Each component is responsible for rendering specific UI elements and interacting with backend APIs to display relevant data.

Additionally, the hooks folder contains custom hooks for managing the state and logic across different parts of the application. For instance, it centralizes state management for easier communication between components. The pages directory is used to organize main views or pages, such as Dashboard.tsx or ChildDetailsCapture.tsx, that provide different functionalities. The data folder stores configuration files and options, such as dropdown menu options, that are shared across multiple components, ensuring consistency and reusability.

For styling, TailwindCSS is used to apply a utility-first CSS approach, making the UI responsive and maintaining a cohesive design across the application. The frontend interacts with the backend using axios for sending API requests, and Toastify is integrated to provide user feedback on actions such as successful data submission.

The frontend directory structure (Figure 6.1), showcasing the organization of assets, components, and configuration files. The figure highlights how the frontend is modularized for clear separation between concerns, ensuring that each component or hook serves a specific function in the application.

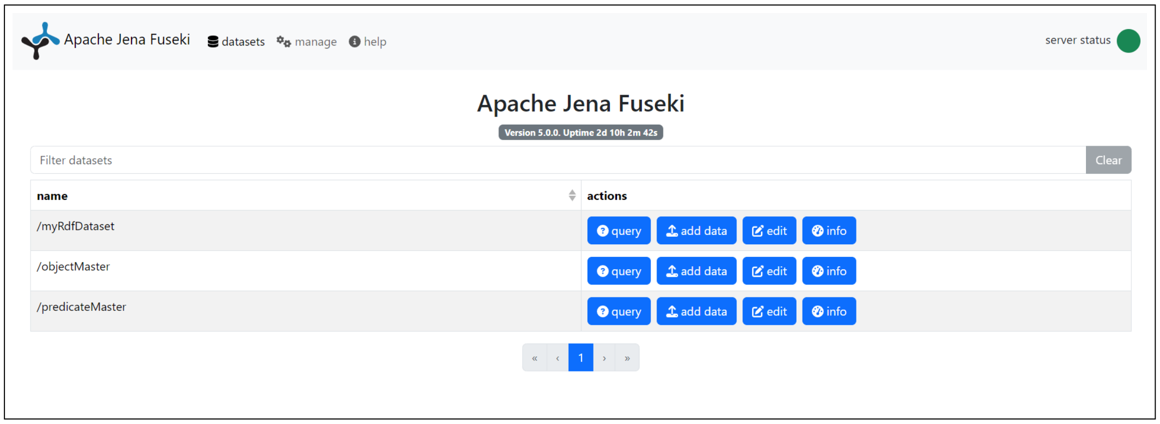
The backend directory is where all the server-side operations take place. Developed using Node.js and Express.js, the backend handles API requests, database operations, and core business logic. The controllers folder contains logic for handling specific routes, such as familyHistoryController.js and fracturesController.js. Each controller defines how the server responds to different HTTP requests (e.g., GET, POST) by interacting with models and performing the necessary operations like fetching or saving data.

The routes folder contains route definitions that map HTTP requests to controller functions. For example, familyHistoryRoutes.js and fracturesRoutes.js define routes such as /api/familyhistory and /api/fractures, respectively, and link them to the relevant controller methods.

Other utility files in the backend manage middleware, such as error handling and logging, to ensure smooth operations and better debugging. The server.js file initializes the Express application and listens for incoming connections.

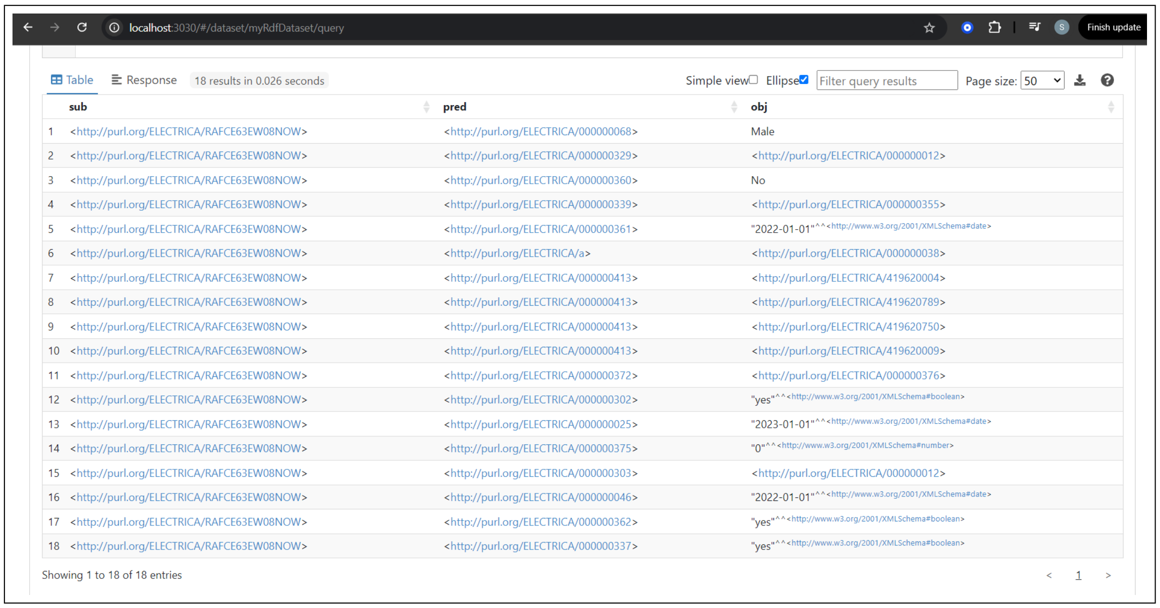
The backend directory structure(Figure 6.2), showing how controllers, routes, and models are organized to handle different aspects of the server-side logic. The figure emphasizes the separation between API route handling and business logic, making the backend efficient and easier to maintain.

6.2 Database



name	actions
/myRdfDataset	query · add data · edit · info
/objectMaster	query · add data · edit · info
/predicateMaster	query · add data · edit · info

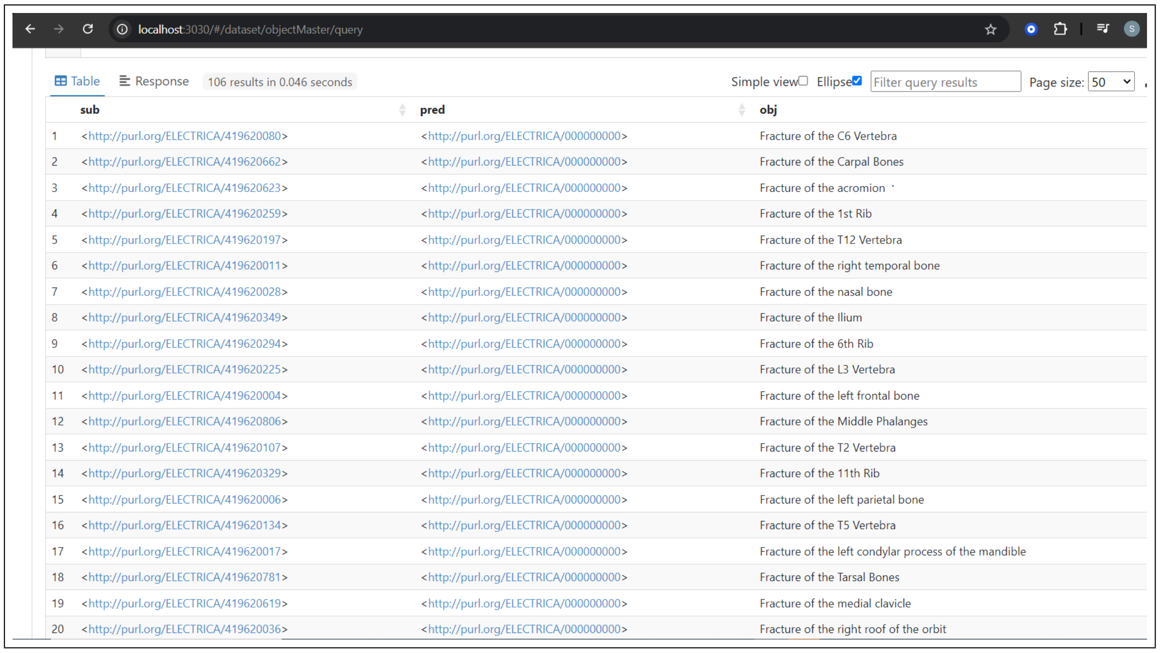
Figure 6.4: Datasets.



	sub	pred	obj
1	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000068>	Male
2	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000329>	<http://purl.org/ELECTRICA/000000012>
3	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000360>	No
4	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000339>	<http://purl.org/ELECTRICA/000000355>
5	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000361>	"2022-01-01"^^<http://www.w3.org/2001/XMLSchema#date>
6	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/a>	<http://purl.org/ELECTRICA/000000038>
7	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000413>	<http://purl.org/ELECTRICA/419620004>
8	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000413>	<http://purl.org/ELECTRICA/419620789>
9	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000413>	<http://purl.org/ELECTRICA/419620750>
10	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000413>	<http://purl.org/ELECTRICA/419620009>
11	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000372>	<http://purl.org/ELECTRICA/000000376>
12	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000302>	"yes"^^<http://www.w3.org/2001/XMLSchema#boolean>
13	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000025>	"2023-01-01"^^<http://www.w3.org/2001/XMLSchema#date>
14	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000375>	"0"^^<http://www.w3.org/2001/XMLSchema#number>
15	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000303>	<http://purl.org/ELECTRICA/000000012>
16	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000046>	"2022-01-01"^^<http://www.w3.org/2001/XMLSchema#date>
17	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000362>	"yes"^^<http://www.w3.org/2001/XMLSchema#boolean>
18	<http://purl.org/ELECTRICA/RAFCE63EW08NOW>	<http://purl.org/ELECTRICA/000000337>	"yes"^^<http://www.w3.org/2001/XMLSchema#boolean>

Showing 1 to 18 of 18 entries

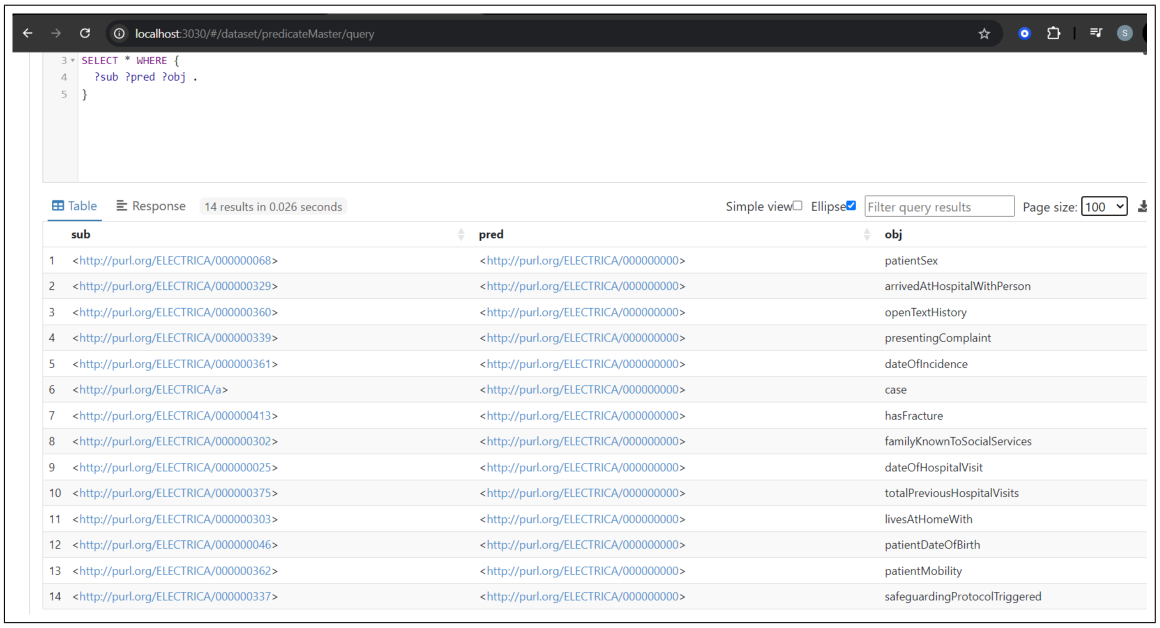
Figure 6.5: Patient Family History and Fractures input data.



106 results in 0.046 seconds

	sub	pred	obj
1	<http://purl.org/ELECTRICA/419620080>	<http://purl.org/ELECTRICA/000000000>	Fracture of the C6 Vertebra
2	<http://purl.org/ELECTRICA/419620662>	<http://purl.org/ELECTRICA/000000000>	Fracture of the Carpal Bones
3	<http://purl.org/ELECTRICA/419620623>	<http://purl.org/ELECTRICA/000000000>	Fracture of the acromion
4	<http://purl.org/ELECTRICA/419620259>	<http://purl.org/ELECTRICA/000000000>	Fracture of the 1st Rib
5	<http://purl.org/ELECTRICA/419620197>	<http://purl.org/ELECTRICA/000000000>	Fracture of the T12 Vertebra
6	<http://purl.org/ELECTRICA/419620011>	<http://purl.org/ELECTRICA/000000000>	Fracture of the right temporal bone
7	<http://purl.org/ELECTRICA/419620028>	<http://purl.org/ELECTRICA/000000000>	Fracture of the nasal bone
8	<http://purl.org/ELECTRICA/419620349>	<http://purl.org/ELECTRICA/000000000>	Fracture of the Ilium
9	<http://purl.org/ELECTRICA/419620294>	<http://purl.org/ELECTRICA/000000000>	Fracture of the 6th Rib
10	<http://purl.org/ELECTRICA/419620225>	<http://purl.org/ELECTRICA/000000000>	Fracture of the L3 Vertebra
11	<http://purl.org/ELECTRICA/419620004>	<http://purl.org/ELECTRICA/000000000>	Fracture of the left frontal bone
12	<http://purl.org/ELECTRICA/419620806>	<http://purl.org/ELECTRICA/000000000>	Fracture of the Middle Phalanges
13	<http://purl.org/ELECTRICA/419620107>	<http://purl.org/ELECTRICA/000000000>	Fracture of the T2 Vertebra
14	<http://purl.org/ELECTRICA/419620329>	<http://purl.org/ELECTRICA/000000000>	Fracture of the 11th Rib
15	<http://purl.org/ELECTRICA/419620006>	<http://purl.org/ELECTRICA/000000000>	Fracture of the left parietal bone
16	<http://purl.org/ELECTRICA/419620134>	<http://purl.org/ELECTRICA/000000000>	Fracture of the T5 Vertebra
17	<http://purl.org/ELECTRICA/419620017>	<http://purl.org/ELECTRICA/000000000>	Fracture of the left condylar process of the mandible
18	<http://purl.org/ELECTRICA/419620781>	<http://purl.org/ELECTRICA/000000000>	Fracture of the Tarsal Bones
19	<http://purl.org/ELECTRICA/419620619>	<http://purl.org/ELECTRICA/000000000>	Fracture of the medial clavicle
20	<http://purl.org/ELECTRICA/419620036>	<http://purl.org/ELECTRICA/000000000>	Fracture of the right roof of the orbit

Figure 6.6: Object Master.



```
SELECT * WHERE {
  ?sub ?pred ?obj .
}
```

14 results in 0.026 seconds

	sub	pred	obj
1	<http://purl.org/ELECTRICA/000000068>	<http://purl.org/ELECTRICA/000000000>	patientSex
2	<http://purl.org/ELECTRICA/000000329>	<http://purl.org/ELECTRICA/000000000>	arrivedAtHospitalWithPerson
3	<http://purl.org/ELECTRICA/000000360>	<http://purl.org/ELECTRICA/000000000>	openTextHistory
4	<http://purl.org/ELECTRICA/000000339>	<http://purl.org/ELECTRICA/000000000>	presentingComplaint
5	<http://purl.org/ELECTRICA/000000361>	<http://purl.org/ELECTRICA/000000000>	dateOfIncidence
6	<http://purl.org/ELECTRICA/a>	<http://purl.org/ELECTRICA/000000000>	case
7	<http://purl.org/ELECTRICA/000000413>	<http://purl.org/ELECTRICA/000000000>	hasFracture
8	<http://purl.org/ELECTRICA/000000302>	<http://purl.org/ELECTRICA/000000000>	familyKnownToSocialServices
9	<http://purl.org/ELECTRICA/000000025>	<http://purl.org/ELECTRICA/000000000>	dateOfHospitalVisit
10	<http://purl.org/ELECTRICA/000000375>	<http://purl.org/ELECTRICA/000000000>	totalPreviousHospitalVisits
11	<http://purl.org/ELECTRICA/000000303>	<http://purl.org/ELECTRICA/000000000>	livesAtHomeWith
12	<http://purl.org/ELECTRICA/000000046>	<http://purl.org/ELECTRICA/000000000>	patientDateOfBirth
13	<http://purl.org/ELECTRICA/000000362>	<http://purl.org/ELECTRICA/000000000>	patientMobility
14	<http://purl.org/ELECTRICA/000000337>	<http://purl.org/ELECTRICA/000000000>	safeguardingProtocolTriggered

Figure 6.7: Predicate Master.

The database system for the application is built on Apache Jena Fuseki, a powerful triple store designed to manage RDF (Resource Description Framework) data. This system captures complex relationships using the subject-predicate-object format, which is ideal for representing structured data in a flexible way. The architecture revolves around multiple datasets, including

/myRdfDataset, /objectMaster, and /predicateMaster, as shown in Figure 6.3. The Fuseki interface offers tools to query, add, and edit data in these datasets. Each dataset serves a unique purpose: /myRdfDataset holds individual patient records, /objectMaster contains fracture types and medical conditions, and /predicateMaster stores predicates that define the relationships between data points. Importantly, the URIs (Uniform Resource Identifiers) used in these datasets are semantically consistent with the ELECTRICA ontology, ensuring that the data model adheres to standardized conventions for RDF data representation.

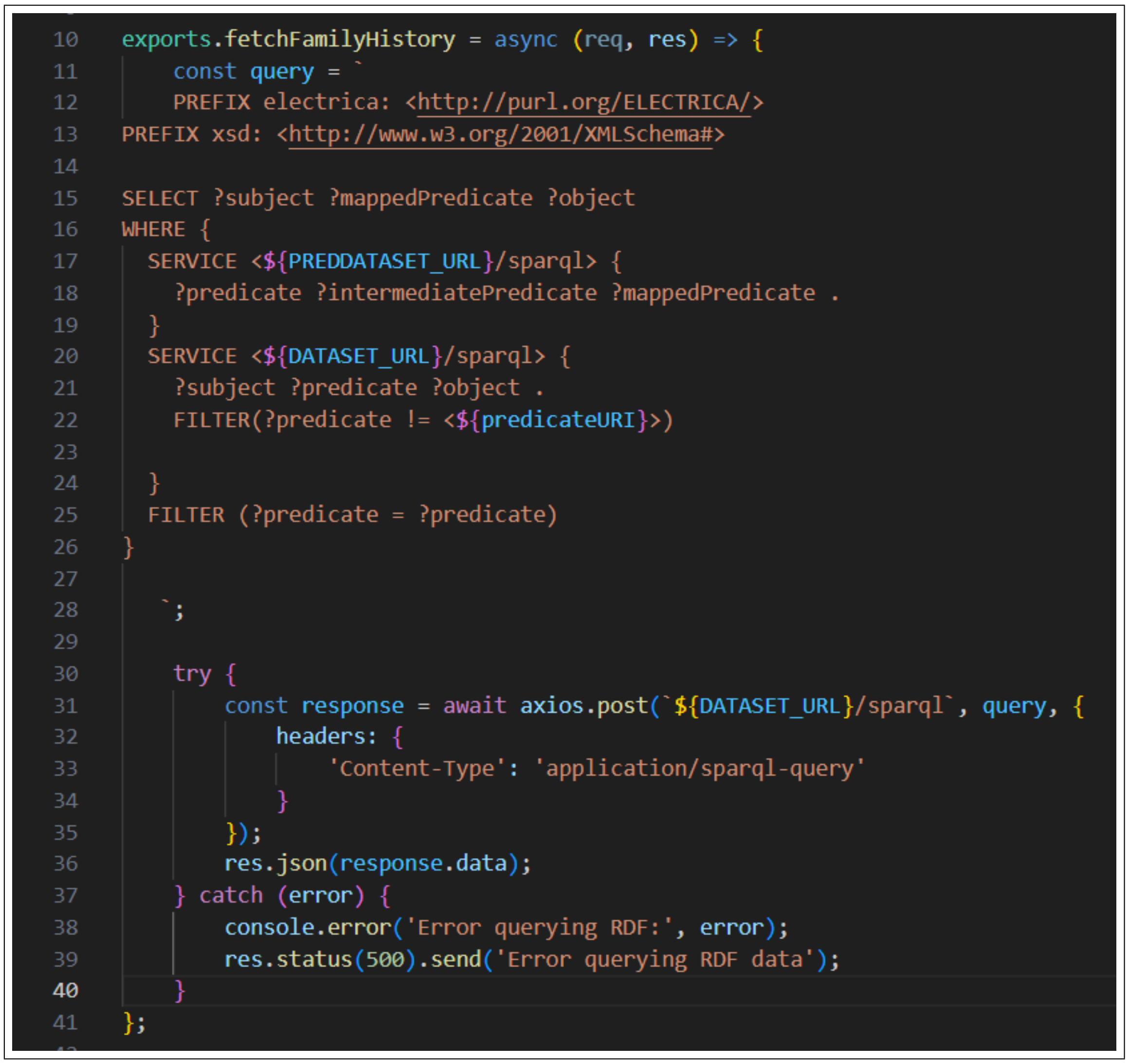
In Figure 6.4, a query on the /myRdfDataset dataset displays detailed patient-specific data. This dataset captures both personal and medical history, as well as information about fractures. Each entry is stored as an RDF triple, where the subject is a unique patient identifier, the predicate describes the data point (such as patientSex or presentingComplaint), and the object represents the value of that data point (such as ”Male” or ”Abdominal Pain”). Fractures are also recorded here, with triples linking a patient to specific fracture details like affected bones or areas of the body. This makes the /myRdfDataset critical for capturing the entire medical profile of a patient, including their fracture history. The use of ELECTRICA ontology URIs ensures that the subject and predicate labels are universally understandable and can be linked to external data systems if needed.

Figure 6.5 presents the /objectMaster dataset, which serves as a master repository for fractures. Unlike /myRdfDataset, this dataset does not store patient-specific data. Instead, it contains general definitions of fractures and their associated bones. For example, an RDF triple here might define a ”Fracture of the C6 Vertebra,” allowing the system to standardize the types of fractures that can be linked to patient records. This dataset plays a crucial role in maintaining an organized ontology of all possible fractures, ensuring consistency when recording patient data. The use of ELECTRICA ontology in this dataset further strengthens the semantic interoperability of the data, making it easier to integrate with other RDF-based healthcare systems.

Lastly, Figure 6.6 displays the /predicateMaster dataset, which stores the predicates used to link subjects and objects in the RDF triples. These predicates define the relationships between data points, such as patientSex, hasFracture, or presentingComplaint. The predicates ensure that relationships within the data are semantically meaningful, enabling efficient queries and analyses. By using ELECTRICA-compliant URIs for these predicates, the system maintains a high level of semantic consistency, which is crucial for making the data machine-readable and interoperable with other systems.

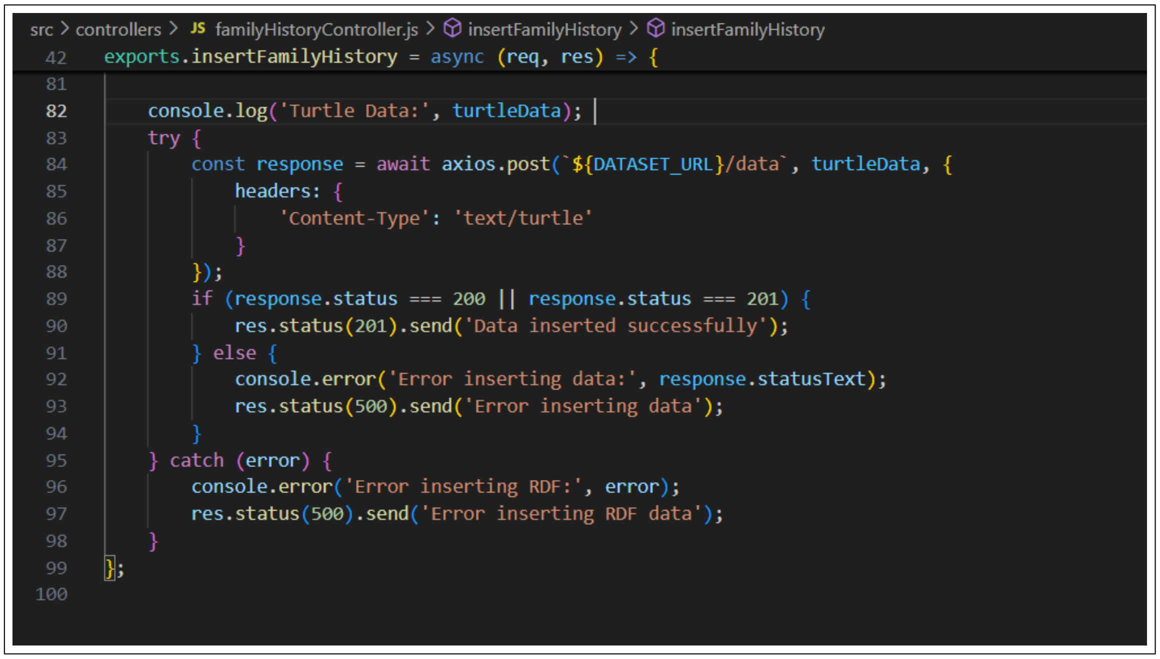
In summary, while Figure 6.4 focuses on patient data, including medical history and fractures, Figure 6.5 illustrates generalized fracture data that standardizes medical conditions across the system. This separation between specific patient data and general medical information ensures the system maintains clear, structured, and flexible data management. The RDF database model, enriched with ELECTRICA ontology URIs, allows for efficient tracking of patient conditions and facilitates easy querying of medical records. This design not only offers an organized and scalable solution for managing data but also ensures semantic consistency, enhancing the interoperability and reusability of the data across different ELECTRICA platforms.

6.3 Data Insertion and Querying in RDF Database



```
exports.fetchFamilyHistory = async (req, res) => {
    const query = `
    PREFIX electrica: <http://purl.org/ELECTRICA/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?subject ?mappedPredicate ?object
WHERE {
  SERVICE <${PREDDATASET_URL}/sparql> {
    ?predicate ?intermediatePredicate ?mappedPredicate .
  }
  SERVICE <${DATASET_URL}/sparql> {
    ?subject ?predicate ?object .
    FILTER(?predicate != <${predicateURI}>)

  }
  FILTER (?predicate = ?predicate)
}

   `;

    try {
        const response = await axios.post(`${DATASET_URL}/sparql`, query, {
            headers: {
                'Content-Type': 'application/sparql-query'
            }
        });
        res.json(response.data);
    } catch (error) {
        console.error('Error querying RDF:', error);
        res.status(500).send('Error querying RDF data');
    }
};
```

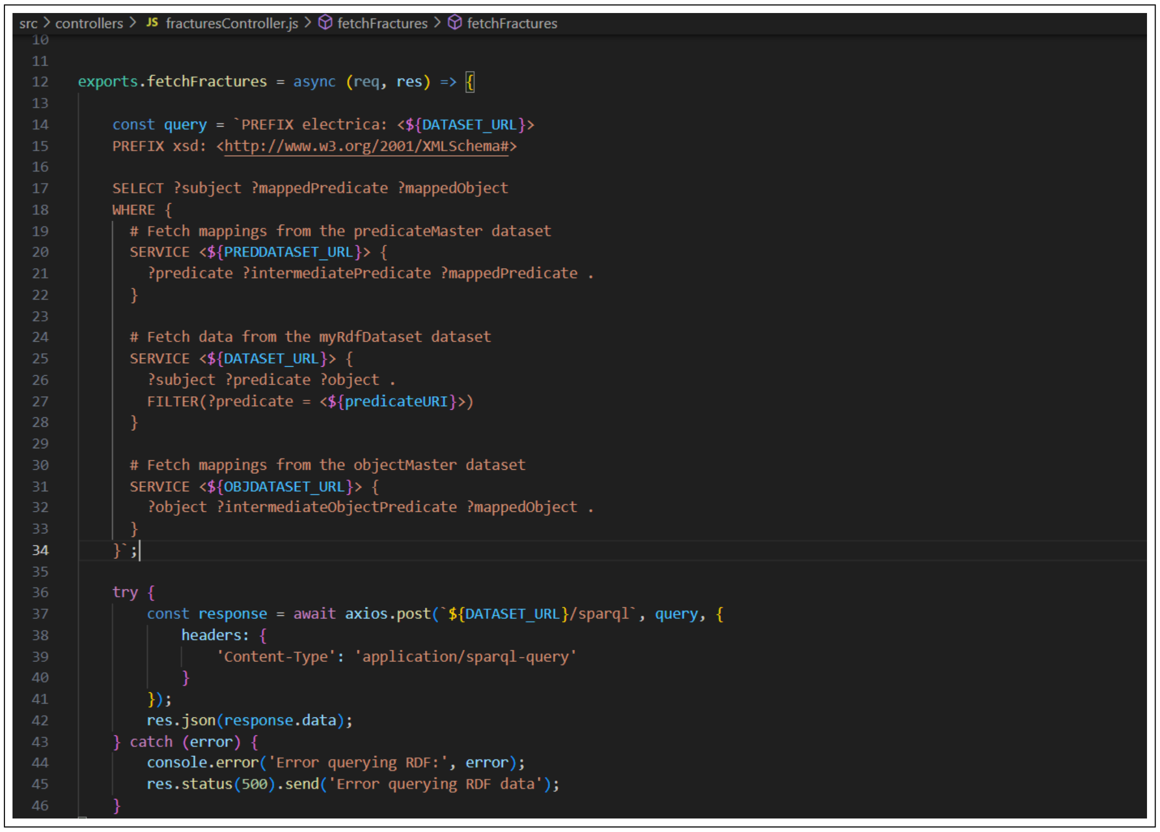
Figure 6.8: Fetch Family History.



src > controllers > JS familyHistoryController.js > insertFamilyHistory > insertFamilyHistory

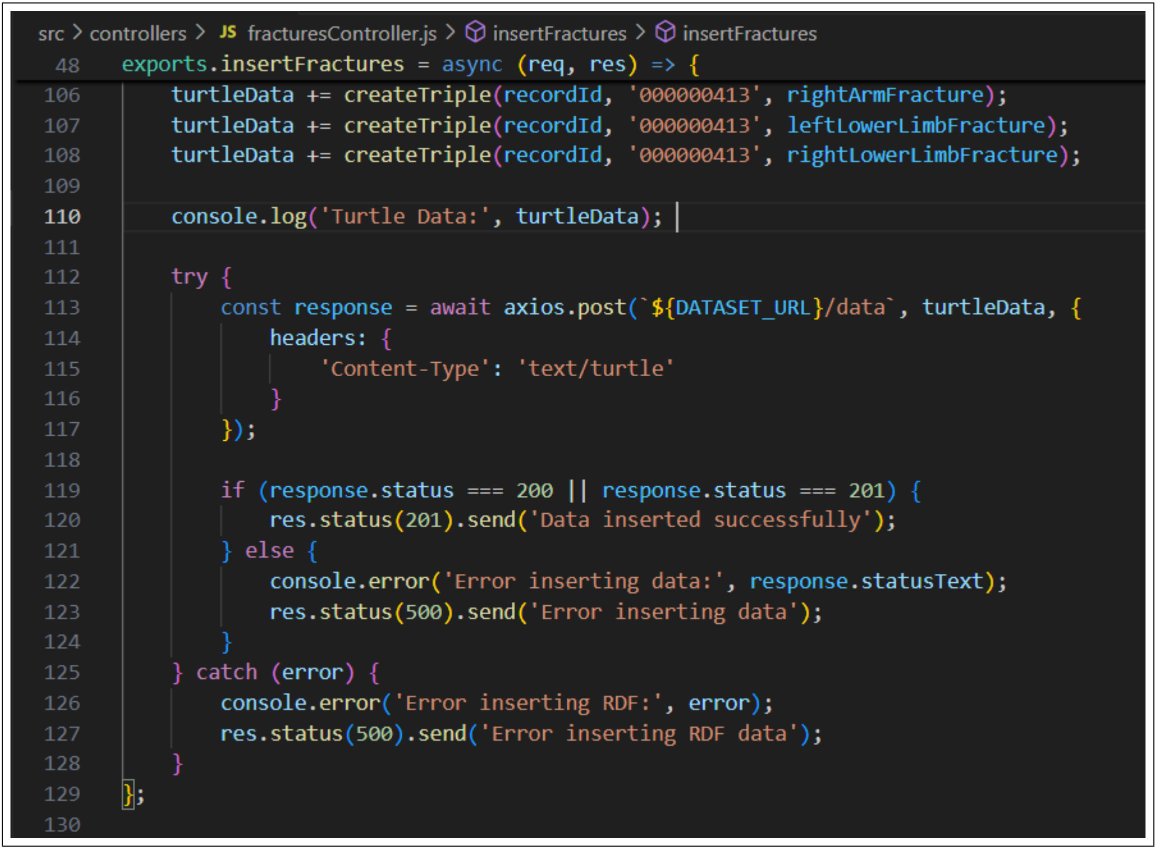
```
exports.insertFamilyHistory = async (req, res) => {

    console.log('Turtle Data:', turtleData);
    try {
        const response = await axios.post(`${DATASET_URL}/data`, turtleData, {
            headers: {
                'Content-Type': 'text/turtle'
            }
        });
        if (response.status === 200 || response.status === 201) {
            res.status(201).send('Data inserted successfully');
        } else {
            console.error('Error inserting data:', response.statusText);
            res.status(500).send('Error inserting data');
        }
    } catch (error) {
        console.error('Error inserting RDF:', error);
        res.status(500).send('Error inserting RDF data');
    }
};
```

Figure 6.9: Insert Family History.


src > controllers > JS fracturesController.js > fetchFractures > fetchFractures

```javascript

exports.fetchFractures = async (req, res) => {

    const query = `PREFIX electrica: <${DATASET_URL}>
    PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

    SELECT ?subject ?mappedPredicate ?mappedObject
    WHERE {
      # Fetch mappings from the predicateMaster dataset
      SERVICE <${PREDDATASET_URL}> {
        ?predicate ?intermediatePredicate ?mappedPredicate .
      }

      # Fetch data from the myRdfDataset dataset
      SERVICE <${DATASET_URL}> {
        ?subject ?predicate ?object .
        FILTER(?predicate = <${predicateURI}>)
      }

      # Fetch mappings from the objectMaster dataset
      SERVICE <${OBJDATASET_URL}> {
        ?object ?intermediateObjectPredicate ?mappedObject .
      }
    }`;

    try {
        const response = await axios.post(`${DATASET_URL}/sparql`, query, {
            headers: {
                'Content-Type': 'application/sparql-query'
            }
        });
        res.json(response.data);
    } catch (error) {
        console.error('Error querying RDF:', error);
        res.status(500).send('Error querying RDF data');
    }
```

Figure 6.10: Fetch Fractures.



src > controllers > JS fracturesController.js > insertFractures > insertFractures

```
exports.insertFractures = async (req, res) => {
    turtleData += createTriple(recordId, '000000413', rightArmFracture);
    turtleData += createTriple(recordId, '000000413', leftLowerLimbFracture);
    turtleData += createTriple(recordId, '000000413', rightLowerLimbFracture);

    console.log('Turtle Data:', turtleData);

    try {
        const response = await axios.post(`${DATASET_URL}/data`, turtleData, {
            headers: {
                'Content-Type': 'text/turtle'
            }
        });

        if (response.status === 200 || response.status === 201) {
            res.status(201).send('Data inserted successfully');
        } else {
            console.error('Error inserting data:', response.statusText);
            res.status(500).send('Error inserting data');
        }
    } catch (error) {
        console.error('Error inserting RDF:', error);
        res.status(500).send('Error inserting RDF data');
    }
};
```

Figure 6.11: Insert Fractures.

This section describes the backend implementation for querying and inserting data into the RDF (Resource Description Framework) database using Apache Jena Fuseki, with Express.js acting as the core server framework. The backend handles both Family History and Fracture Data, ensuring structured data storage and retrieval using the subject-predicate-object format common to RDF triple stores. This approach allows for capturing complex relationships while maintaining semantic consistency across the data.

The Family History data is managed through two main functions: fetchFamilyHistory and insertFamilyHistory. The fetchFamilyHistory function sends a SPARQL query to the myRdfDataset and predicateMaster datasets. Figure 6.7 shows how the function queries these two datasets simultaneously using the SERVICE clause. The myRdfDataset contains patient-specific data, while the predicateMaster dataset provides mappings of predicates to more human-readable terms. This process ensures that data retrieved from the RDF database is both semantically rich and easy to interpret. Express.js efficiently routes the request from the client-side, ensuring that the query results are returned smoothly.

On the other hand, the insertFamilyHistory function dynamically constructs RDF triples in Turtle format for new patient records. These triples represent patient data, such as date of birth, safeguarding status, and presenting complaints, in a structured RDF format. Each data point is formatted with the appropriate datatype (e.g., xsd:date for dates and xsd:boolean for booleans). Figure 6.8 shows how the data insertion process works. Express.js handles this operation by routing the POST request to Apache Jena Fuseki, ensuring that the patient data is inserted efficiently into the RDF database.

Similarly, Fracture Data is handled through two key functions: fetchFractures and insertFractures. The fetchFractures function, as depicted in Figure 6.9, queries across three datasets: myRdfDataset, objectMaster, and predicateMaster. This query retrieves patient-specific fracture information and maps it to standard medical conditions stored in the objectMaster dataset. Express.js processes the query and sends the results back to the client-side, allowing the Health care professional to view detailed fracture data alongside other patient information.

The insertFractures function constructs RDF triples representing different types of fractures, such as skull fractures, rib fractures, and spinal fractures. Figure 6.10 illustrates how each fracture is linked to the patient's recordId, ensuring that the data remains structured and easily retrievable. By sending these triples in Turtle format to Apache Jena Fuseki, the backend ensures that fracture data is properly stored in the RDF database. Express.js once again facilitates the routing of these POST requests, ensuring efficient data insertion and feedback to the user interface.

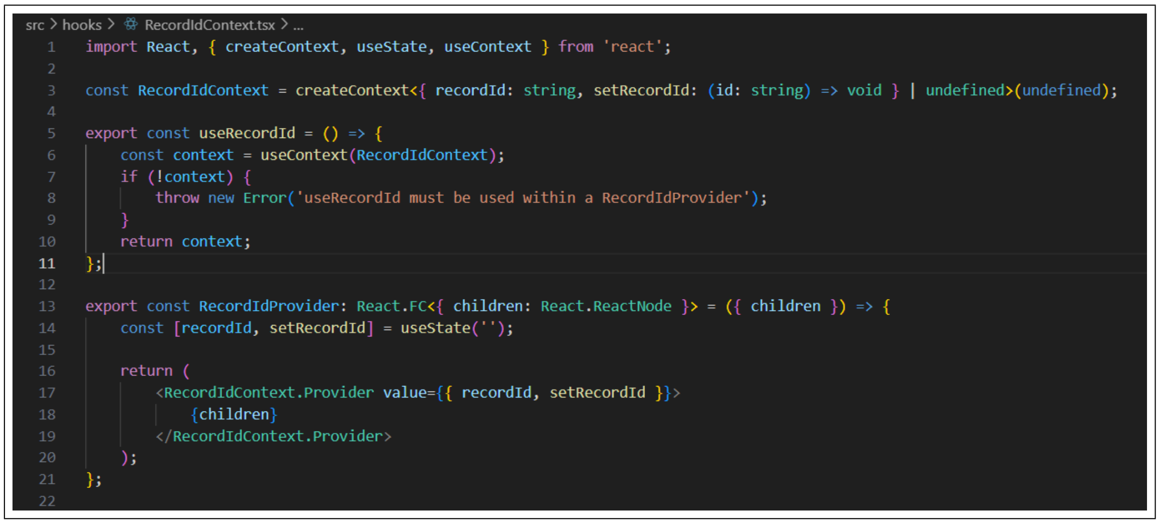
Express.js is crucial for managing the interaction between the client and the RDF database. It routes all operations, ensuring they adhere to the ELECTRICA ontology. This ensures semantic consistency in how data is stored and retrieved. The use of SPARQL queries, RDF triples, and consistent URIs guarantees that the system is interoperable with other RDF-based systems and can scale as more data is added.

Express.js plays a pivotal role in ensuring that all data operations—whether querying or inserting—are handled efficiently and semantically in line with RDF standards. The integration with Apache Jena Fuseki allows the system to manage complex healthcare records, such as Family History and Fractures data, in a structured and scalable way. Figures 6.7, 6.8, 6.9 and 6.10 demonstrate how data flows between the client-side, backend, and RDF database, ensuring the entire system is both robust and adaptable to future changes.

6.4 Data Handling in the React Frontend

This tool is responsible for handling family history and fracture data submission, and it utilises React Context for managing shared states.

6.4.1 React Context for Record ID Management



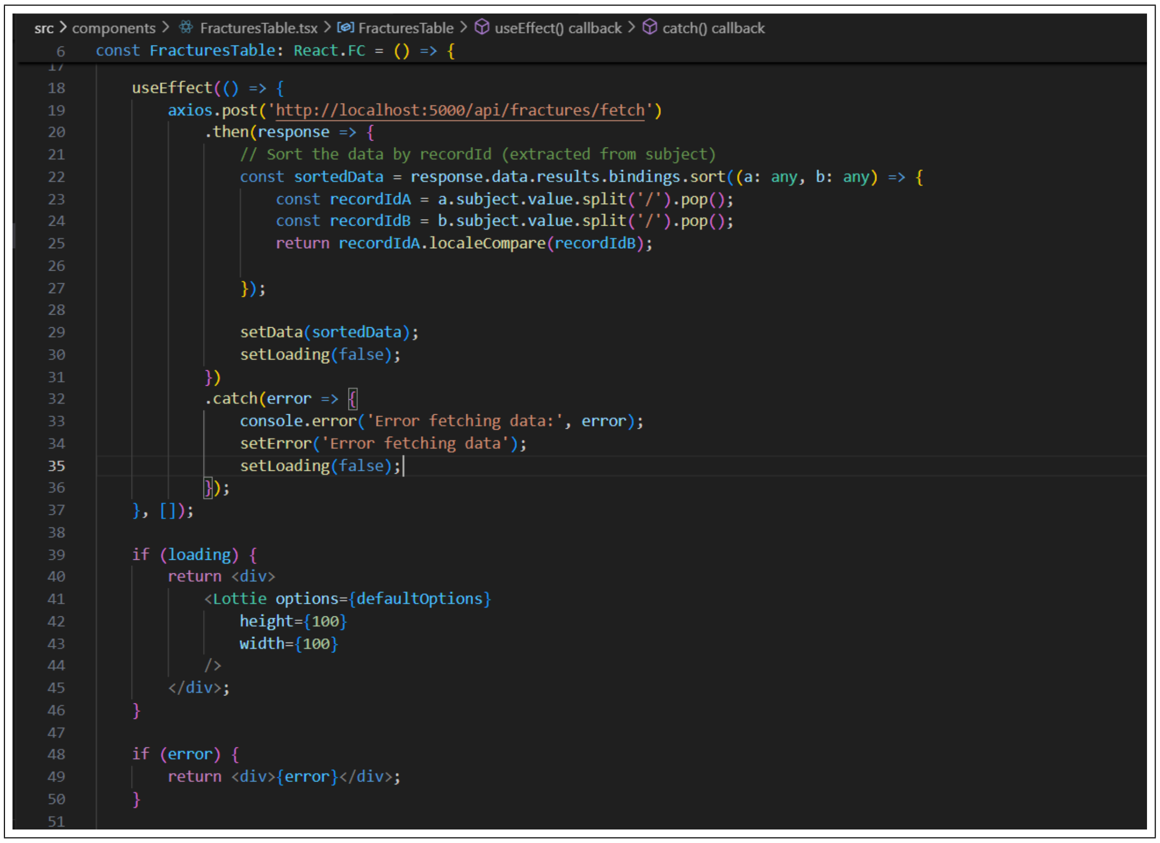
src > hooks > RecordIdContext.tsx > ...

```tsx
import React, { createContext, useState, useContext } from 'react';

const RecordIdContext = createContext<{ recordId: string, setRecordId: (id: string) => void } | undefined>(undefined);

export const useRecordId = () => {
    const context = useContext(RecordIdContext);
    if (!context) {
        throw new Error('useRecordId must be used within a RecordIdProvider');
    }
    return context;
};

export const RecordIdProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
    const [recordId, setRecordId] = useState('');

    return (
        <RecordIdContext.Provider value={{ recordId, setRecordId }}>
            {children}
        </RecordIdContext.Provider>
    );
};
```

Figure 6.12: Context API for RecordId management.

Figure 6.11 displays the code for creating a RecordIdContext using React's createContext and useContext hooks. The RecordIdContext is essential for passing the recordId (a unique identifier for each patient) throughout the component tree without manually prop-drilling through multiple layers of components. The useRecordId hook provides easy access to the recordId value in any component, ensuring consistency. This structure is encapsulated in the RecordIdProvider component, which manages the state of the recordId using React's useState hook. By wrapping the main application in RecordIdProvider, as shown in Figure 6.15, it becomes accessible to any child components such as the Family History and Fractures forms.

6.4.2 Fractures Data Table Fetching and Sorting

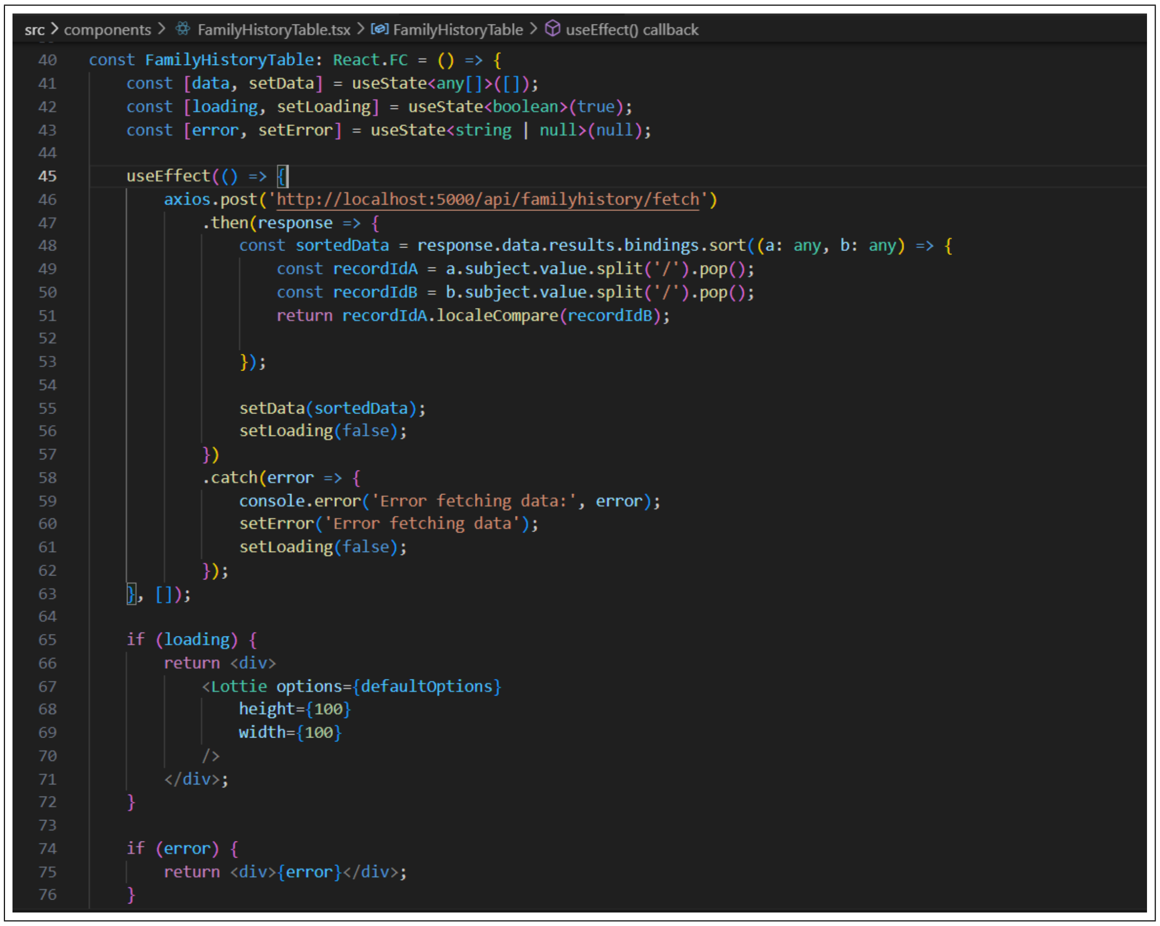

src > components > FracturesTable.tsx > FracturesTable > useEffect() callback > catch() callback

```
const FracturesTable: React.FC = () => {

    useEffect(() => {
        axios.post('http://localhost:5000/api/fractures/fetch')
            .then(response => {
                // Sort the data by recordId (extracted from subject)
                const sortedData = response.data.results.bindings.sort((a: any, b: any) => {
                    const recordIdA = a.subject.value.split('/').pop();
                    const recordIdB = b.subject.value.split('/').pop();
                    return recordIdA.localeCompare(recordIdB);

                });

                setData(sortedData);
                setLoading(false);
            })
            .catch(error => {
                console.error('Error fetching data:', error);
                setError('Error fetching data');
                setLoading(false);
            });
    }, []);

    if (loading) {
        return <div>
            <Lottie options={defaultOptions}
                height={100}
                width={100}
            />
        </div>;
    }

    if (error) {
        return <div>{error}</div>;
    }

```

Figure 6.13: Fractures data fetch.

Figure 6.12 illustrates how FracturesTable fetches and displays fracture data for the patients. The useEffect hook sends an HTTP POST request to the backend API using Axios to fetch data from the RDF database. Upon receiving the response, the data is sorted based on the recordId extracted from the RDF triples, ensuring that patient records are organized in a meaningful sequence. If the data is successfully fetched, it is displayed in a table format; otherwise, an error message or loading animation is shown, managed by conditional rendering based on loading and error states.

6.4.3 Family History Data Table



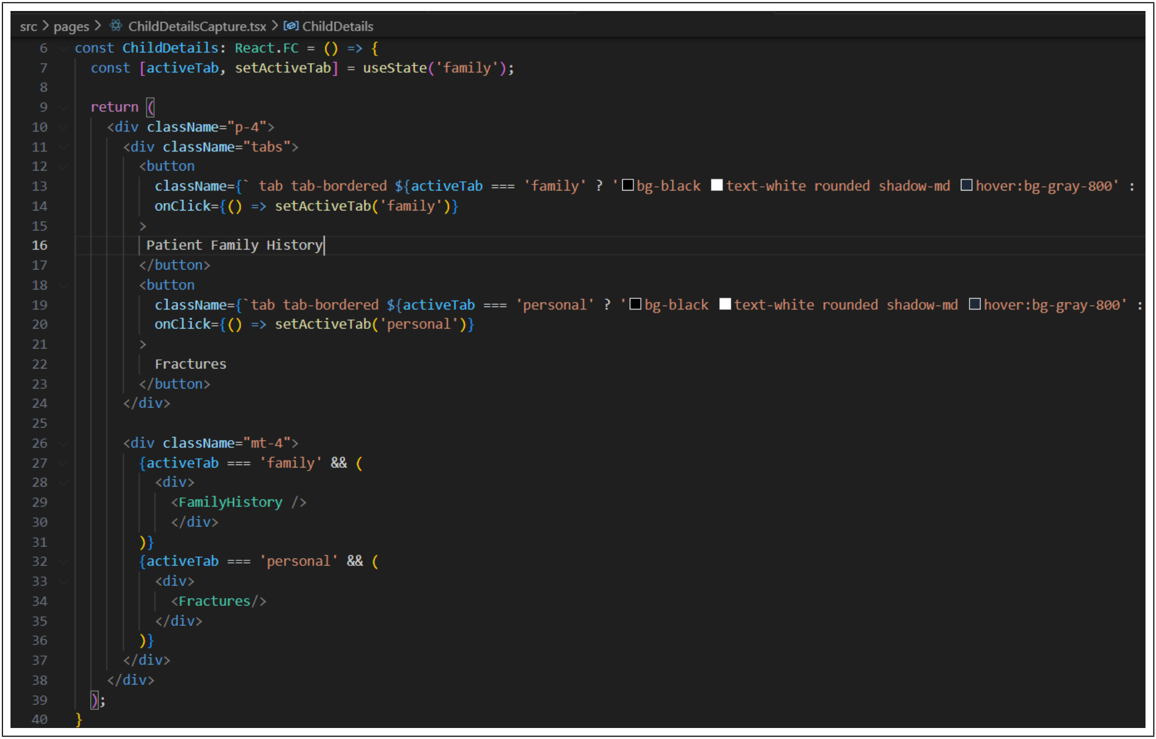
src > components > FamilyHistoryTable.tsx > FamilyHistoryTable > useEffect() callback

```
const FamilyHistoryTable: React.FC = () => {
    const [data, setData] = useState<any[]>([]);
    const [loading, setLoading] = useState<boolean>(true);
    const [error, setError] = useState<string | null>(null);

    useEffect(() => {
        axios.post('http://localhost:5000/api/familyhistory/fetch')
            .then(response => {
                const sortedData = response.data.results.bindings.sort((a: any, b: any) => {
                    const recordIdA = a.subject.value.split('/').pop();
                    const recordIdB = b.subject.value.split('/').pop();
                    return recordIdA.localeCompare(recordIdB);

                });

                setData(sortedData);
                setLoading(false);
            })
            .catch(error => {
                console.error('Error fetching data:', error);
                setError('Error fetching data');
                setLoading(false);
            });
    }, []);

    if (loading) {
        return <div>
            <Lottie options={defaultOptions}
                height={100}
                width={100}
            />
        </div>;
    }

    if (error) {
        return <div>{error}</div>;
    }
```

Figure 6.14: Family History data fetch.

Similarly, Figure 6.13 depicts the FamilyHistoryTable component. It follows the same logic as FracturesTable, fetching family history data through a backend API call. After receiving the data, it is sorted by the recordId, ensuring the organization of patient history records for easy navigation and display. The component displays the data in a table format once the loading state is resolved.

6.4.4 Managing Patient Details with Tabs



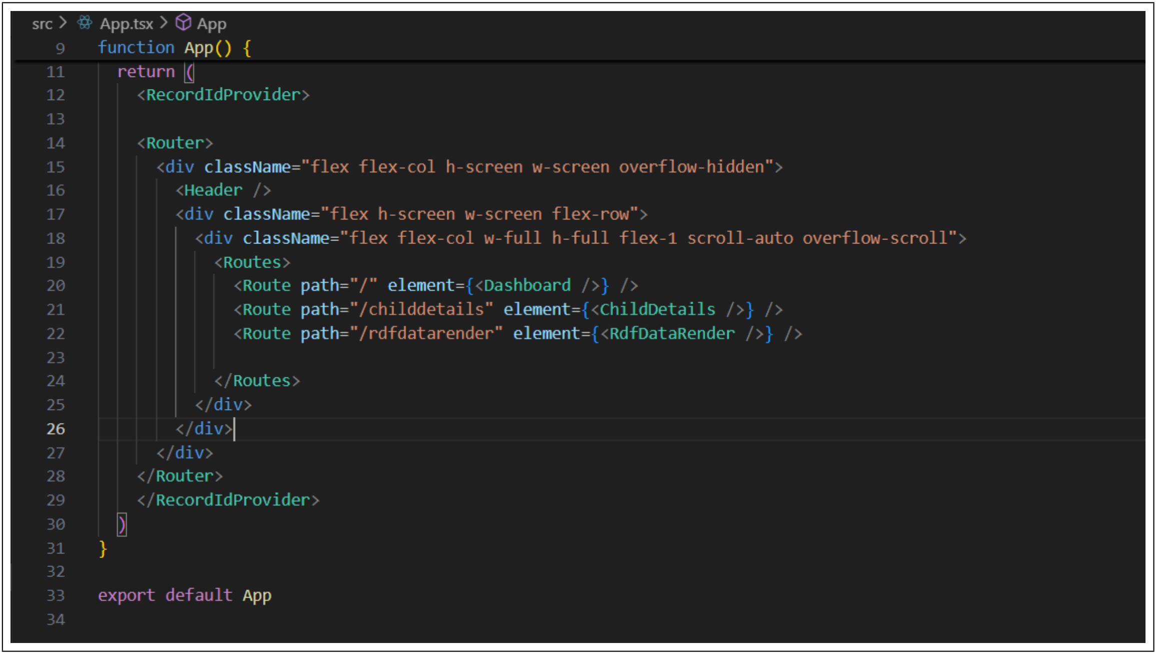
src > pages > ChildDetailsCapture.tsx > ChildDetails

```
const ChildDetails: React.FC = () => {
  const [activeTab, setActiveTab] = useState('family');

  return (
    <div className="p-4">
      <div className="tabs">
        <button
          className={` tab tab-bordered ${activeTab === 'family' ? 'bg-black text-white rounded shadow-md hover:bg-gray-800' :
          onClick={() => setActiveTab('family')}
        >
         Patient Family History
        </button>
        <button
          className={`tab tab-bordered ${activeTab === 'personal' ? 'bg-black text-white rounded shadow-md hover:bg-gray-800' :
          onClick={() => setActiveTab('personal')}
        >
          Fractures
        </button>
      </div>

      <div className="mt-4">
        {activeTab === 'family' && (
          <div>
            <FamilyHistory />
            </div>
        )}
        {activeTab === 'personal' && (
          <div>
            <Fractures/>
          </div>
        )}
      </div>
    </div>
  );
}
```

Figure 6.15: Child Details Capture Component.

In Figure 6.14, we see the ChildDetailsCapture component, which organizes the UI for capturing patient data using tabs. It uses the useState hook to manage the active tab, allowing the Health care professional to toggle between viewing Family History and Fractures sections. Each section renders the appropriate form component: FamilyHistory or Fractures, based on the currently active tab. This design ensures a clean separation of concerns, offering the Health care professional a straightforward method to interact with different sections of the patient record.

6.4.5 Routing and Component Rendering

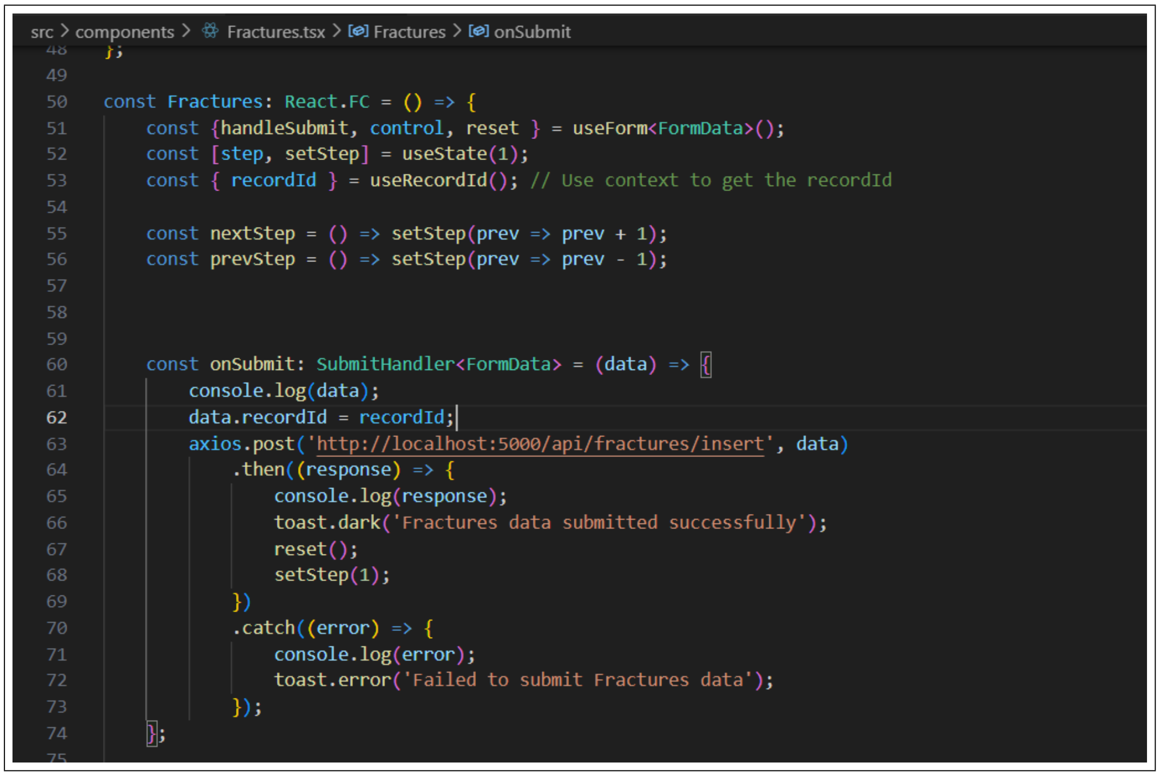

src > App.tsx > App

```
function App() {
  return (
    <RecordIdProvider>

    <Router>
      <div className="flex flex-col h-screen w-screen overflow-hidden">
        <Header />
        <div className="flex h-screen w-screen flex-row">
          <div className="flex flex-col w-full h-full flex-1 scroll-auto overflow-scroll">
            <Routes>
              <Route path="/" element={<Dashboard />} />
              <Route path="/childdetails" element={<ChildDetails />} />
              <Route path="/rdfdatarender" element={<RdfDataRender />} />

            </Routes>
          </div>
        </div>
      </div>
    </Router>
    </RecordIdProvider>
  )
}

export default App
```

Figure 6.16: App component.

Figure 6.15 demonstrates how React Router is utilised to manage the different views of the application. The App component defines routes for the Dashboard, ChildDetails, and RdfDataRender pages. The RecordIdProvider wraps the entire routing structure, ensuring that the recordId is available across all components rendered within these routes. This structure allows for scalable navigation, making it easier to add or modify routes in the future.

6.4.6 Submitting Fracture Data

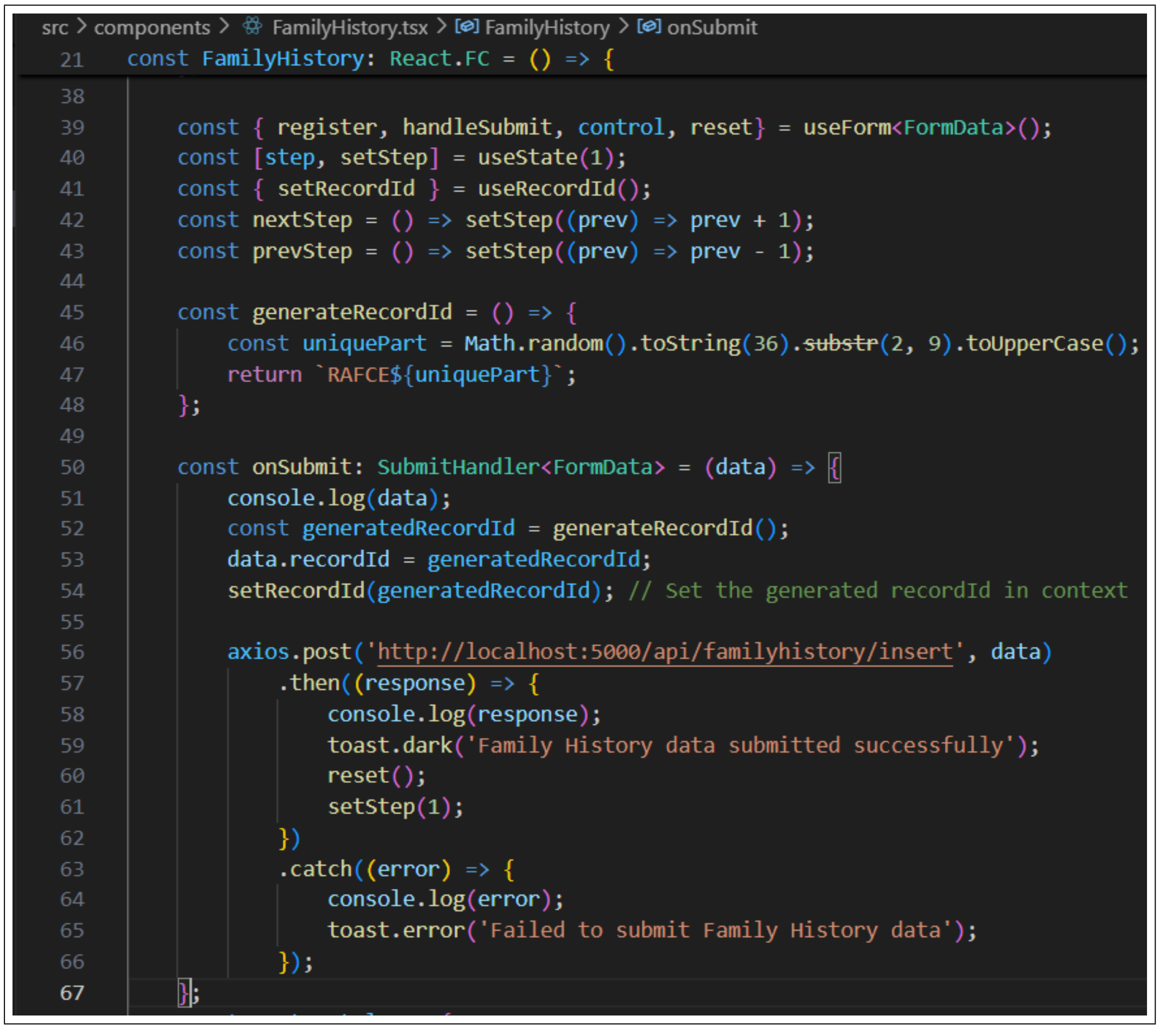

src > components > Fractures.tsx > Fractures > onSubmit

```
};

const Fractures: React.FC = () => {
    const {handleSubmit, control, reset } = useForm<FormData>();
    const [step, setStep] = useState(1);
    const { recordId } = useRecordId(); // Use context to get the recordId

    const nextStep = () => setStep(prev => prev + 1);
    const prevStep = () => setStep(prev => prev - 1);



    const onSubmit: SubmitHandler<FormData> = (data) => {
        console.log(data);
        data.recordId = recordId;
        axios.post('http://localhost:5000/api/fractures/insert', data)
            .then((response) => {
                console.log(response);
                toast.dark('Fractures data submitted successfully');
                reset();
                setStep(1);
            })
            .catch((error) => {
                console.log(error);
                toast.error('Failed to submit Fractures data');
            });
    };
```

Figure 6.17: Fractures Submit.

Figure 6.16 focuses on the Fractures form component responsible for capturing fracture details. It uses the useRecordId hook to retrieve the current recordId and includes it in the submission payload. The handleSubmit function manages the form submission by making an API call to insert the fracture data into the RDF database. Upon successful submission, the form is reset, and a success message is shown using Toastify.

6.4.7 Submitting Family Data



src > components > FamilyHistory.tsx > FamilyHistory > onSubmit

```
const FamilyHistory: React.FC = () => {

    const { register, handleSubmit, control, reset} = useForm<FormData>();
    const [step, setStep] = useState(1);
    const { setRecordId } = useRecordId();
    const nextStep = () => setStep((prev) => prev + 1);
    const prevStep = () => setStep((prev) => prev - 1);

    const generateRecordId = () => {
        const uniquePart = Math.random().toString(36).substr(2, 9).toUpperCase();
        return `RAFCE${uniquePart}`;
    };

    const onSubmit: SubmitHandler<FormData> = (data) => {
        console.log(data);
        const generatedRecordId = generateRecordId();
        data.recordId = generatedRecordId;
        setRecordId(generatedRecordId); // Set the generated recordId in context

        axios.post('http://localhost:5000/api/familyhistory/insert', data)
            .then((response) => {
                console.log(response);
                toast.dark('Family History data submitted successfully');
                reset();
                setStep(1);
            })
            .catch((error) => {
                console.log(error);
                toast.error('Failed to submit Family History data');
            });
    };
```

Figure 6.18: Family History Submit.

Similarly, in Figure 6.17, the FamilyHistory form component allows users to enter and submit patient family history data. A unique recordId is generated using the generateRecordId function and stored in the context for use in subsequent forms. The form data, along with the recordId, is sent to the backend via an Axios POST request, and upon successful submission, the form is reset.

6.5 Project Outcomes

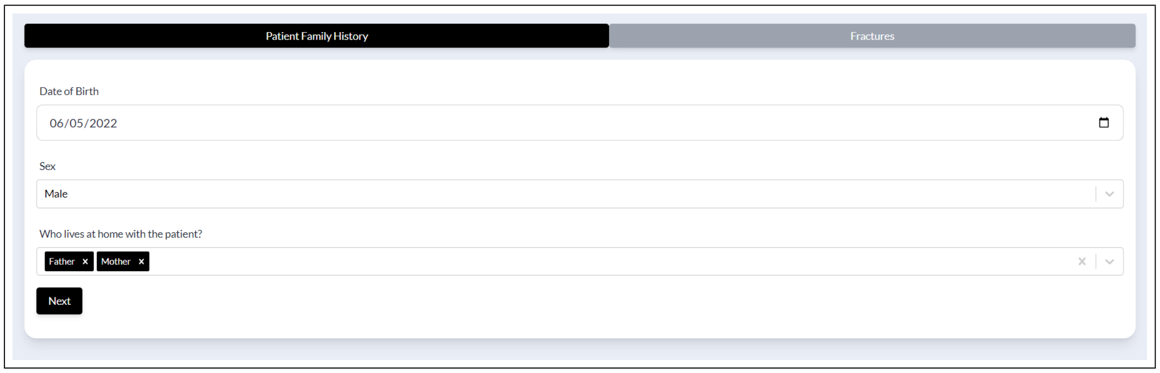
The Fracture Log Dashboard serves as the central hub for healthcare professionals using the application to record and analyse data related to child abuse cases. At the top, a black header bar with a navigation menu allows Health care professionals to access the Record and View sections easily. The dashboard is divided into two key sections:

Child Abuse Overview: This section provides an overview of the importance of addressing child abuse, citing specific statistics from the NSPCC that highlight the need for better tools to identify and address suspected abuse. This contextual information helps Health care professionals understand the significance of the data they are recording.

Fracture Log: This section describes the core purpose of the tool - recording and analyzing data on fractures and related injuries in children, particularly in cases of suspected child abuse. The tool is part of the broader ELECTRICA initiative, which aims to standardise the recording of critical incidents across multiple domains.

The page design is user-friendly, with rounded corners, a clear layout, and consistent styling. The interface is designed to be intuitive, ensuring that Health care professionals can easily navigate the system and focus on the task of recording and viewing critical data efficiently.

The Patient Family History section of the Fracture Log application is designed to capture essential details about a child's family and medical background. This form is part of a multi-step process that allows healthcare professionals to systematically enter data related to suspected child abuse cases.



Date of Birth: 06/05/2022

Sex: Male

Who lives at home with the patient? Father × Mother ×

Figure 6.19: Patient Family History form.

Step 1: Basic Patient Information The first screen in this section prompts the Health care professional to enter the child's Date of Birth, and Sex. These fields ensure that the patient's basic demographic data is accurately recorded. Once the required fields are filled, the Health care professional can proceed by clicking the Next button.

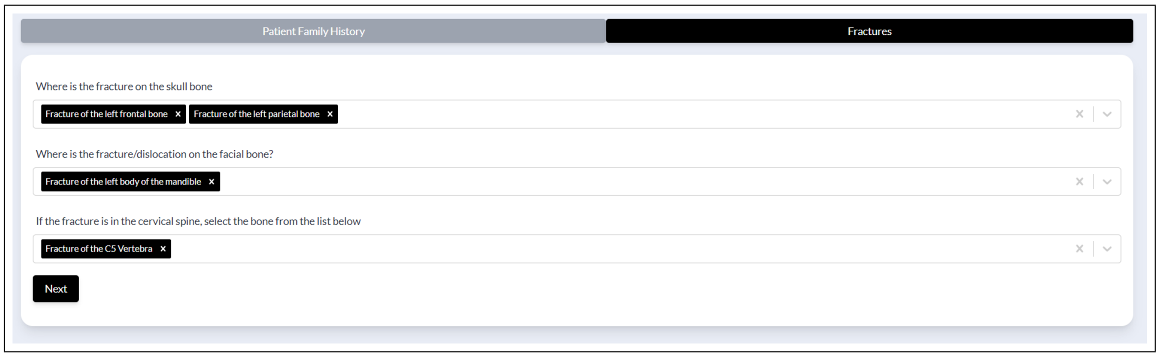
Step 2: Household and Visit Information This screen collects information on who lives at home with the patient, the Date of Visit, and who accompanied the child when they were brought in for the visit. This information helps contextualise the patient's living situation and the circumstances of their current visit. Again, after filling out this information, the Health care professional clicks Next to proceed.

Step 3: Social and Medical History On the third screen, the form asks if the current hospital visit triggered any safeguarding protocols, whether the family is known to social services, and the total number of hospital visits since birth. This data is essential for identifying any patterns of concern and understanding the patient's history within social and medical systems. Again, after filling out this information, the Health care professional clicks Next to proceed.

Step 4: Presenting Complaint and History This screen collects detailed information about the incident leading to the hospital visit. Healthcare professionals select the Presenting Complaint from a dropdown list, provide detailed notes in the History field, and record the date of the incident or when the issue was first noticed. This information helps establish a timeline and context for assessing the child's condition. Again, after filling out this information, the Health care professional clicks Next to proceed.

Step 5: Incident Details and Pre-existing Conditions The final screen in the Patient Family History section allows the healthcare professional to document the Presenting Complaint for the current hospital visit, including a history of the incident and the date it occurred. Additionally, the form inquires whether the child is normally able to walk or cruise, and if there are any pre-existing conditions that might be relevant to the patient's current state. After reviewing and ensuring all information is accurate, the Health care professional clicks Submit to finalise the input for the Patient Family History.

The Patient Fractures section of the Fracture Log application is designed to collect detailed information regarding fractures and dislocations in a child's body. This form is part of a multi-step process that allows healthcare professionals to systematically enter data.



Where is the fracture on the skull bone

Fracture of the left frontal bone × Fracture of the left parietal bone ×

Where is the fracture/dislocation on the facial bone?

Fracture of the left body of the mandible ×

If the fracture is in the cervical spine, select the bone from the list below

Fracture of the C5 Vertebra ×

Figure 6.20: Fractures form.

Step 1: Basic Fracture Information The first screen in this section prompts to enter the fields where the healthcare professional can specify the location of fractures on the skull and facial bones. These dropdown list allow the Health care professional to select the precise areas affected by the injury, ensuring accurate and detailed documentation. Once the required fields are filled, the Health care professional can proceed by clicking the Next button.

Step 2: Spinal Fractures This screen focuses on documenting any fractures in the patient's spine. The form provides dropdown list for the Cervical Spine, Thoracic Spine, and Lumbar Spine sections, where the Health care professional can select the specific vertebrae affected. This step ensures that all spinal injuries are recorded with precision, aiding in comprehensive injury assessment. After completing this information, the Health care professional clicks Next to proceed.

Step 3: Rib and Sacral Spine Fractures On the third screen, the form asks the healthcare professional to document fractures in the Sacral Spine/Coccyx and Ribs. There are options to select the specific ribs affected on both the left and right sides. This data is crucial for understanding the extent of rib and lower spine injuries, which are often critical indicators in cases of suspected abuse. After filling out this information, the Health care professional clicks Next to continue.

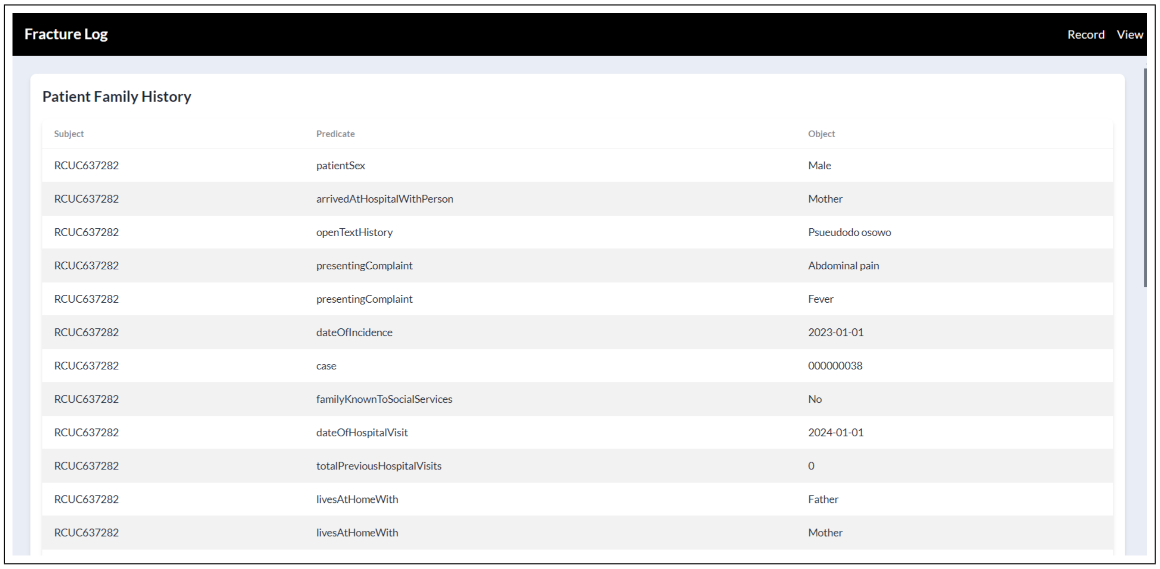
Step 4: Pelvis, Sternum, and Shoulder Girdle Fractures This screen collects information on fractures in the Pelvis, Sternum, and Shoulder Girdle. The healthcare professional selects the specific bones affected using dropdown menus. This detailed documentation helps in accurately assessing injuries in these central areas, which are essential for evaluating the overall impact of trauma. The Health care professional clicks Next to proceed after completing this step.

Step 5: Upper Limb Fractures The fifth screen focuses on fractures in the Upper Limbs. The form allows the Health care professional to document injuries in the arms, wrists, and hands for both the left and right sides. Each dropdown menu helps in selecting the specific bones affected, ensuring comprehensive coverage of potential injuries. The Health care professional clicks Next to move on to the final step.

Step 6: Lower Limb Fractures and Submission The final screen in the Patient Fractures section asks the healthcare professional to document fractures in the Lower Limbs, including the legs and feet, by selecting the specific bones affected on both sides. After reviewing all the entered information for accuracy, the Health care professional clicks Submit to finalise and save the data in the system. This final step ensures that all relevant fracture data has been thoroughly documented and is ready for further analysis or reporting.

The Data Render screen in the Fracture Log application is designed to display the data that has been entered into the system in a structured and easily understandable format. This screen provides a detailed view of both the Patient Family History and the Fractures data that have been recorded, ensuring that healthcare professionals can review and verify the information effectively. Patient Family History Section The first section of the Data View screen focuses on the Patient Family History(Figure 6.20). The data is organised into three main columns: Subject, Predicate, and Object. Subject: This column typically contains a unique identifier for the patient (e.g., RAFCE0001), which is consistent across all the patient's records. Predicate: This column describes the specific attribute or relationship being recorded (e.g., patientSex, arrivedAtHospitalWithPerson, presentingComplaint). Object: This column provides the value or description associated with the predicate (e.g., Male, Father, Blood in stool). This structure allows the Health care professional to quickly understand the details of the patient's family history, including the context of the hospital visit, social service involvement, and any medical concerns.

Fractures Section Below the Patient Family History, the Fractures section (Figure 6.21) is displayed in a similar format. This section lists each fracture or dislocation identified during the patient's examination: Subject: As in the Patient Family History, this typically

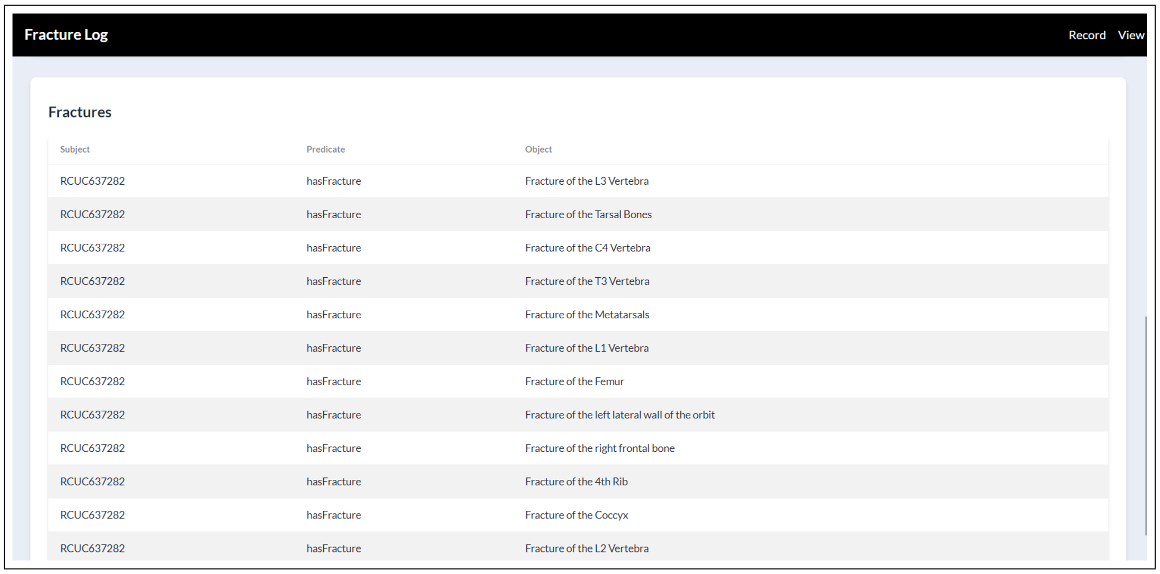


Patient Family History

Subject	Predicate	Object
RCUC637282	patientSex	Male
RCUC637282	arrivedAtHospitalWithPerson	Mother
RCUC637282	openTextHistory	Psueudodo osowo
RCUC637282	presentingComplaint	Abdominal pain
RCUC637282	presentingComplaint	Fever
RCUC637282	dateOfIncidence	2023-01-01
RCUC637282	case	000000038
RCUC637282	familyKnownToSocialServices	No
RCUC637282	dateOfHospitalVisit	2024-01-01
RCUC637282	totalPreviousHospitalVisits	0
RCUC637282	livesAtHomeWith	Father
RCUC637282	livesAtHomeWith	Mother

Figure 6.21: Patient Family History View.

references the unique patient identifier. Predicate: This indicates the specific type of injury or condition being documented (e.g., hasFracture). Object: This provides detailed information about the location and nature of the fracture (e.g., Fracture of the left frontal bone, Fracture of the medial clavicle). This section ensures that all the injury data is accurately captured and can be reviewed in relation to the patient's overall medical history. The Data Render screen is crucial for enabling healthcare professionals to review entered data, ensuring that all necessary information has been accurately recorded before it is used for further analysis or decision-making in suspected child abuse cases.



Fractures

Subject	Predicate	Object
RCUC637282	hasFracture	Fracture of the L3 Vertebra
RCUC637282	hasFracture	Fracture of the Tarsal Bones
RCUC637282	hasFracture	Fracture of the C4 Vertebra
RCUC637282	hasFracture	Fracture of the T3 Vertebra
RCUC637282	hasFracture	Fracture of the Metatarsals
RCUC637282	hasFracture	Fracture of the L1 Vertebra
RCUC637282	hasFracture	Fracture of the Femur
RCUC637282	hasFracture	Fracture of the left lateral wall of the orbit
RCUC637282	hasFracture	Fracture of the right frontal bone
RCUC637282	hasFracture	Fracture of the 4th Rib
RCUC637282	hasFracture	Fracture of the Coccyx
RCUC637282	hasFracture	Fracture of the L2 Vertebra

Figure 6.22: Fractures View.

Chapter 7

Evaluation and Testing

To evaluate the system, the following criteria will be used:

7.1 Jakob Nielsen's Heuristics

1. **Visibility of System Status [15]** Observation: When a user submits a form, there is a clear alert indicating that the system has processed the request. Test: Observed the form submission process FamilyHistory and Fractures component Result: Users are informed of the status of their submission instantly through loading spinners and success/error messages. Explanation: A loading state is set when the form is submitted, and based on the response, a success or error toast notification is displayed to inform the user of the submission status.

2. **Match Between System and the Real World[15]** Observation: The language used in the forms and buttons is clear, understandable, and avoids technical jargon. Test: Reviewed labels and instructions in FamilyHistory and Fractures component. Result: Users can understand the labels and instructions without confusion. Explanation: Using clear and familiar terminology helps users understand the purpose of each field without needing additional explanations or instructions.

3. **User Control and Freedom[15]** Observation: There are clear options to undo actions or navigate back to previous steps in multi-step forms. Test: Observed navigation in multi-step forms in FamilyHistory and Fractures component. Result: Users can easily navigate back or cancel actions. Explanation: Providing navigation buttons in multi-step forms ensures that users can easily go back to the previous steps if they need to correct any information or review their inputs.

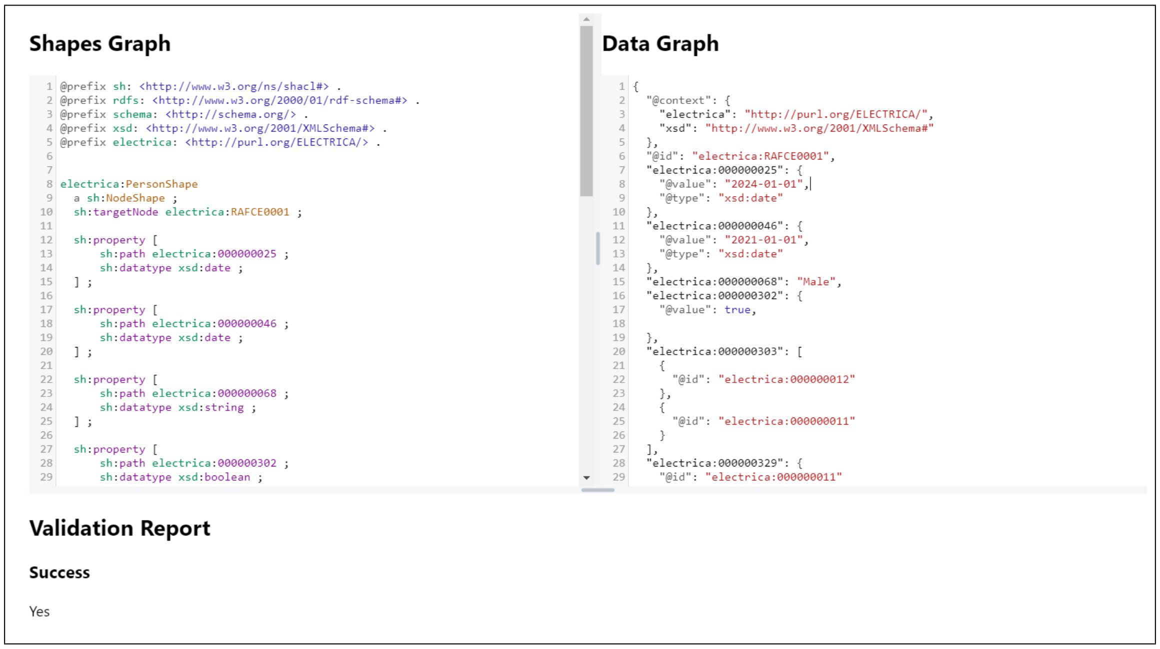
4. **Consistency and Standards[15]** Observation: Buttons, icons, and terminology are consistent throughout the application. Test: Reviewed different components such as ChildDetailsCapture ,FamilyHistory, and Fractures. Result: Users can recognize patterns and understand functionalities without re-learning. Explanation: Consistent

design elements, such as button styles, ensure that users can easily identify interactive elements and understand their functions throughout the application.

5. **Recognition Rather Than Recall**[15] Observation: Dropdowns and auto-suggestions are used effectively, reducing the need for users to remember information. Test: Checked the usage of dropdowns in forms. Result: Users can select options easily without needing to recall information. Explanation: By providing dropdowns with predefined options, users can quickly select the appropriate value without having to remember or type it, which reduces cognitive load and enhances user experience.

7.2 SHACL for RDF data

The Shapes Constraint Language(SHACL) is a tool used to validate RDF data, making sure it follows a specific format. This ensures semantic consistency of RDF data with an existing ontology specific to ELECTRICA. The RDF data graphs represent entities described by a variety of properties. Each property must conform to specific data types or structures, and some properties may have multiple values that need to be validated. To perform the SHACL validation, SHACL Playground [22] was utilised. The tool facilitated the straightforward input of RDF data and SHACL shapes, providing immediate validation feedback. The data was deemed valid if all properties within the specified nodes adhered to the constraints outlined in the SHACL shapes. This encompassed correct data types for literals, such as dates and booleans, and accurate use of IRIs in designated instances.



Shapes Graph

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix electrica: <http://purl.org/ELECTRICA/> .


electrica:PersonShape
  a sh:NodeShape ;
  sh:targetNode electrica:RAFCE0001 ;

  sh:property [
      sh:path electrica:000000025 ;
      sh:datatype xsd:date ;
  ] ;

  sh:property [
      sh:path electrica:000000046 ;
      sh:datatype xsd:date ;
  ] ;

  sh:property [
      sh:path electrica:000000068 ;
      sh:datatype xsd:string ;
  ] ;

  sh:property [
      sh:path electrica:000000302 ;
      sh:datatype xsd:boolean ;
```

Data Graph

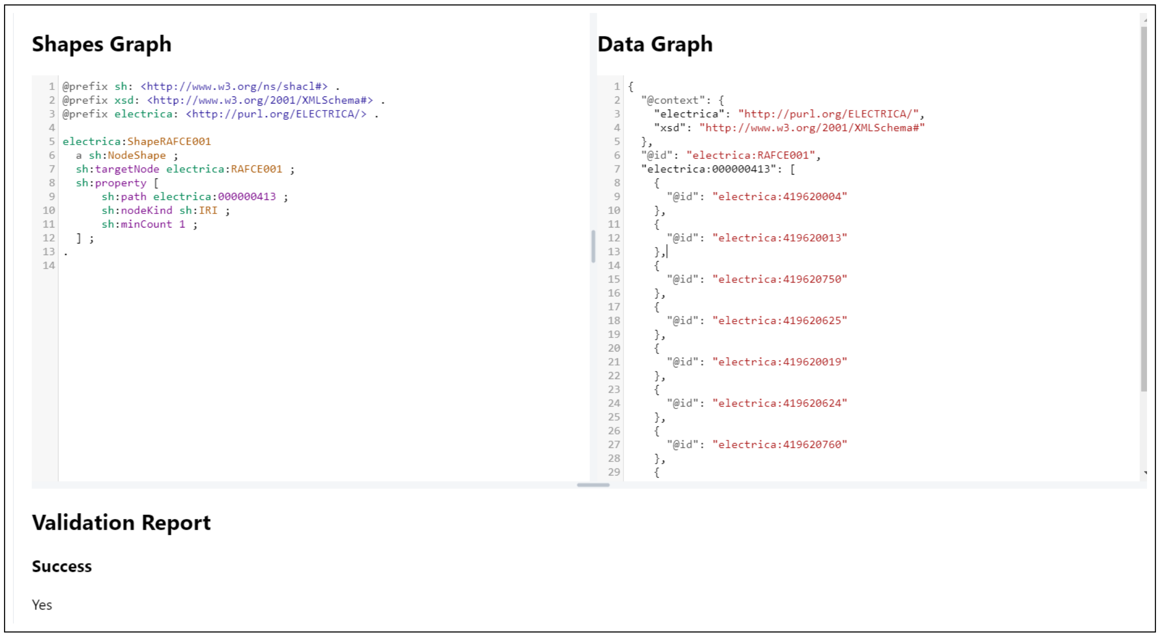
```
{
  "@context": {
    "electrica": "http://purl.org/ELECTRICA/",
    "xsd": "http://www.w3.org/2001/XMLSchema#"
  },
  "@id": "electrica:RAFCE0001",
  "electrica:000000025": {
    "@value": "2024-01-01",
    "@type": "xsd:date"
  },
  "electrica:000000046": {
    "@value": "2021-01-01",
    "@type": "xsd:date"
  },
  "electrica:000000068": "Male",
  "electrica:000000302": {
    "@value": true,

  },
  "electrica:000000303": [
    {
      "@id": "electrica:000000012"
    },
    {
      "@id": "electrica:000000011"
    }
  ],
  "electrica:000000329": {
    "@id": "electrica:000000011"
```

Validation Report

Success

Yes

Figure 7.1: Family History RDF data.



Shapes Graph

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix electrica: <http://purl.org/ELECTRICA/> .

electrica:ShapeRAFCE001
  a sh:NodeShape ;
  sh:targetNode electrica:RAFCE001 ;
  sh:property [
      sh:path electrica:000000413 ;
      sh:nodeKind sh:IRI ;
      sh:minCount 1 ;
  ] ;
.

```

Data Graph

```
{
  "@context": {
    "electrica": "http://purl.org/ELECTRICA/",
    "xsd": "http://www.w3.org/2001/XMLSchema#"
  },
  "@id": "electrica:RAFCE001",
  "electrica:000000413": [
    {
      "@id": "electrica:419620004"
    },
    {
      "@id": "electrica:419620013"
    },
    {
      "@id": "electrica:419620750"
    },
    {
      "@id": "electrica:419620625"
    },
    {
      "@id": "electrica:419620019"
    },
    {
      "@id": "electrica:419620624"
    },
    {
      "@id": "electrica:419620760"
    },
    {
```

Validation Report

Success

Yes

Figure 7.2: Fractures RDF data.

The data was considered valid if all properties in the nodes conformed to the constraints defined by the SHACL shapes [26]. This included correct data types for literals (e.g., dates, booleans) and correct usage of IRIs where expected. The use of SHACL for validating RDF data proved to be highly effective in ensuring data quality, semantic consistency, and structural integrity. By defining precise shapes for each node, it was possible to enforce strict data typing and alignment with the intended semantic structure. The successful validation of the RDF data graphs demonstrates the robustness of SHACL in managing and validating complex datasets. Furthermore, the defined SHACL shapes ensure that the data conforms not only to syntactic requirements but also to the semantic rules established by the ELECTRICA ontology. This alignment ensures that the data integrates seamlessly with the existing ontology, facilitating interoperability and consistent interpretation across various applications. By maintaining this semantic consistency, the data can be effectively utilised within the broader ELECTRICA framework, supporting more accurate and meaningful analysis. By following this structured approach, the project aims to develop a robust and efficient tool for recording data on fractures in suspected child abuse cases, thereby enhancing child protection efforts through improved data collection.

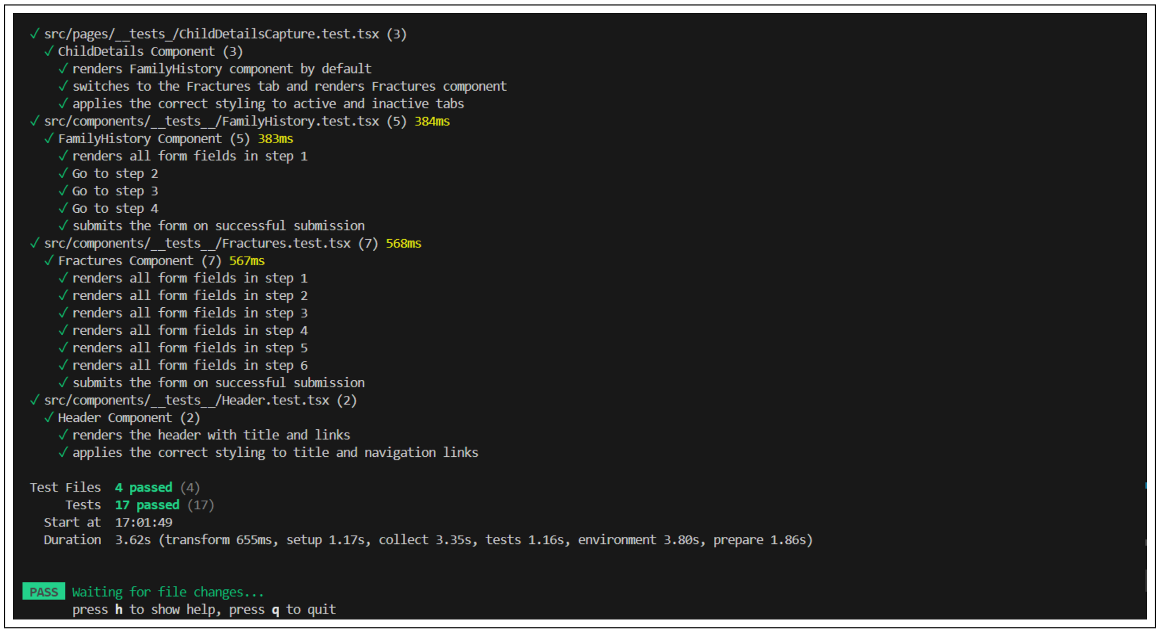
7.3 Requirements Completion

Table 7.1 shows the status of each requirement that was gathered initially before the development of system.

SL No.	Requirements	Importance	Status
1	The system must provide a web-based form for recording data on family history, fractures and related injuries in children.	Mandatory	Completed: Implemented through React components (FamilyHistory.tsx, Fractures.tsx), allowing users to input detailed data via dynamic forms
2	The system must convert form inputs into RDF format, adhering to the ELECTRICA ontology.	Mandatory	Completed: The backend handles RDF conversion using Turtle format through insertFamilyHistory and insertFractures functions
3	The system must store RDF data in a triplestore for easy retrieval and querying.	Mandatory	Completed: Data is stored in Apache Jena Fuseki using RDF triples, with datasets like /myRdfDataset, /predicateMaster, and /objectMaster, supporting structured storage and retrieval
4	The system must provide a SPARQL-based querying mechanism for retrieving specific RDF data.	Mandatory	Completed: SPARQL queries are executed via the backend API for fetching family history and fracture data
5	The system must integrate with the ELECTRICA ontology to ensure semantic accuracy.	Mandatory	Completed: All RDF triples follow the ELECTRICA ontology, ensuring consistent semantic representation
6	The system may incorporate reasoning capabilities to perform inference based on the ontology.	Optional	Not Implemented: This feature was considered optional and has not been incorporated in the current implementation.

Table 7.1: Functional Requirements Completion Status.

7.4 Unit Testing

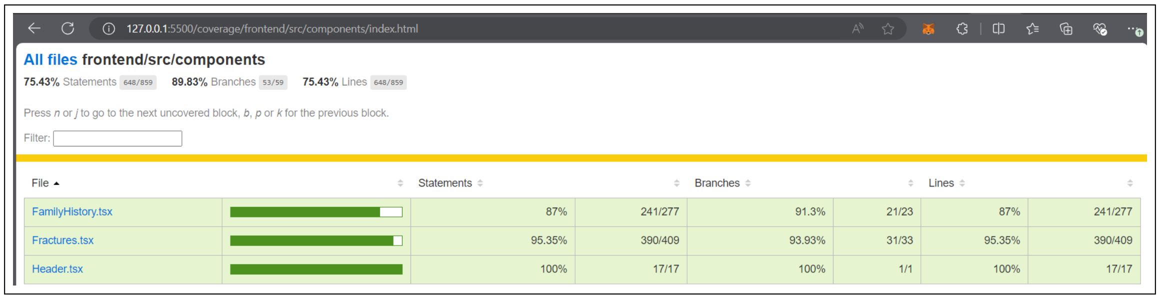


```
 ✓ src/pages/__tests_/ChildDetailsCapture.test.tsx (3)
   ✓ ChildDetails Component (3)
     ✓ renders FamilyHistory component by default
     ✓ switches to the Fractures tab and renders Fractures component
     ✓ applies the correct styling to active and inactive tabs
 ✓ src/components/__tests__/FamilyHistory.test.tsx (5) 384ms
   ✓ FamilyHistory Component (5) 383ms
     ✓ renders all form fields in step 1
     ✓ Go to step 2
     ✓ Go to step 3
     ✓ Go to step 4
     ✓ submits the form on successful submission
 ✓ src/components/__tests__/Fractures.test.tsx (7) 568ms
   ✓ Fractures Component (7) 567ms
     ✓ renders all form fields in step 1
     ✓ renders all form fields in step 2
     ✓ renders all form fields in step 3
     ✓ renders all form fields in step 4
     ✓ renders all form fields in step 5
     ✓ renders all form fields in step 6
     ✓ submits the form on successful submission
 ✓ src/components/__tests__/Header.test.tsx (2)
   ✓ Header Component (2)
     ✓ renders the header with title and links
     ✓ applies the correct styling to title and navigation links

 Test Files  4 passed (4)
      Tests  17 passed (17)
   Start at  17:01:49
   Duration  3.62s (transform 655ms, setup 1.17s, collect 3.35s, tests 1.16s, environment 3.80s, prepare 1.86s)


 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Figure 7.3: Unit Testing.

The figure 7.3 displays the results of a successful unit testing session using the Vitest framework and React Testing Library. In this instance, four test files have been executed, and all tests within these files have passed, totaling 17 successful tests. The tests are categorized into different components such as the ChildDetails component, FamilyHistory component, Fractures component, and Header component. Each component has specific tests associated with it, verifying features like rendering of components (e.g., rendering the FamilyHistory component by default in the ChildDetails component), form submissions, and tab switching behavior. The tests also check the correct rendering of form fields in multiple steps for components like FamilyHistory and Fractures. The summary at the bottom of the results shows that the tests took around 3.62 seconds to complete, with additional time spent on transforming, setting up, and preparing the environment for testing. Overall, this result indicates a robust and passing test coverage for the various components in the system [12], [7].

7.5 Test Coverage



127.0.0.1:5500/coverage/frontend/src/components/index.html

All files frontend/src/components

75.43% Statements 648/859 **89.83%** Branches 53/59 **75.43%** Lines 648/859

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Lines	
FamilyHistory.tsx		87%	241/277	91.3%	21/23	87%	241/277
Fractures.tsx		95.35%	390/409	93.93%	31/33	95.35%	390/409
Header.tsx		100%	17/17	100%	1/1	100%	17/17

Figure 7.4: Test Coverage.

The Test coverage results indicate a comprehensive level of testing across the components in the project. Overall, the codebase has 75.43% coverage for statements, 89.83% for branches, and 75.43% for lines, reflecting the extent to which the code is covered by unit tests. Specifically, the FamilyHistory.tsx component has 87% statement coverage, 91.3% branch coverage, and 87% line coverage. The Fractures.tsx component shows higher coverage with 95.35% for both statements and lines, and 93.93% for branches. These results highlight that critical components like Fractures.tsx and FamilyHistory.tsx are thoroughly tested, ensuring their reliability and stability. Maintaining high coverage helps minimize bugs and ensures the quality of the codebase.

Chapter 8

Conclusion

The aim of this project was to develop a system for recording and querying medical data, specifically focusing on family history and fractures in cases of suspected child abuse. By utilizing RDF (Resource Description Framework) and adhering to the ELECTRICA ontology, the project aimed to create a system that could capture patient data in a semantically meaningful format. Through the development of forms for both Family History and Fractures data entry, combined with the backend's ability to query and store this information in RDF format, the project successfully met its key objectives.

The web-based application can be found on GitHub. The source code can be accessed here: https://github.com/sgr-0007/Rdf-fractures. It was built with React on the frontend and Express.js on the backend, facilitating smooth interaction between the user interface and the RDF database managed by Apache Jena Fuseki. This architecture ensured that healthcare professionals could enter data in a structured manner and retrieve it efficiently for further analysis. The ability to query this RDF data using SPARQL also highlighted the flexibility of the system in handling complex healthcare data and ensuring interoperability with other systems that adhere to semantic web principles.

By comparing this system to traditional databases, it becomes evident that the use of RDF triples and an ontology like ELECTRICA provides significant advantages in terms of data standardization and semantic accuracy. The system ensures that data is not only stored efficiently but can also be easily queried to identify patterns or extract meaningful insights.

The evaluation of the system demonstrated its effectiveness in storing and querying structured data related to child abuse cases. Although a few additional features, such as reasoning capabilities and expanded ontology integration, were marked as future work, the system as developed achieved its primary goal. With further refinement and expansion, this system could serve as a valuable tool for healthcare professionals managing complex medical records in child protection cases.

8.1 Future Work

The system successfully meets the core requirements for recording and querying medical data related to family history and fractures in suspected child abuse cases but several areas present opportunities for future work. One key enhancement involves expanding the ontology integration. Although the system currently uses the ELECTRICA ontology to ensure semantic consistency, integrating additional medical ontologies could further enhance the system's ability to capture a wider range of medical conditions and relationships. This would improve the system's applicability across different medical domains, broadening its utility beyond the specific use case of child abuse cases.

Additionally, implementing reasoning capabilities within the system represents another significant opportunity for future work. By incorporating semantic reasoning, the system could infer new information from the existing data, thereby enabling more intelligent querying. For instance, based on the RDF data and ontological definitions, the system could infer relationships between various symptoms and potential medical conditions, allowing healthcare professionals to gain deeper insights into a patient's condition without manually querying each specific data point.

Another area for future work involves enhancing the user interface to provide more personalised experiences for healthcare professionals. This could include customised dashboards, advanced filtering options for SPARQL queries, and more sophisticated data visualization tools to aid in decision-making. Providing a more intuitive and responsive user interface could improve usability and efficiency, particularly in high-pressure environments where quick access to data is critical.

Moreover, the system could be extended to support mobile platforms, enabling healthcare professionals to access and input data on the go. This would increase flexibility and accessibility, particularly in remote or under-resourced areas where desktop access might be limited. Finally, exploring data privacy and security measures, such as encryption and access control mechanisms, will be critical as the system expands to handle sensitive medical data.

These advancements, while beyond the scope of the current project, would further solidify the system as a comprehensive and flexible tool for managing medical records in diverse healthcare settings.

Bibliography

[1] Akinade, G. React reconciliation, 2023. [Online]. Available: https://dev.to/gbengacode/react-reconciliation-59am [Accessed: Sep. 9, 2024].

[2] Antoniou, G., and van Harmelen, F. *A Semantic Web Primer.* MIT Press, Cambridge, MA, USA, 2004.

[3] BioPortal. Electrica ontology. [Online]. Available: https://bioportal.bioontology.org/ontologies/ELECTRICA.

[4] Capadisli, S., Guy, A., Kellogg, G., and Berners-Lee, T. rdflib.js: A javascript library for working with rdf, 2022. GitHub Repository, [Online]. Available: https://github.com/linkeddata/rdflib.js.

[5] Cornet, R., and Keizer, N. D. Forty years of snomed: A literature review. *BMC Medical Informatics and Decision Making 8*, 1 (2008), 1–6.

[6] Dietze, H. E. S., and Hayashi, R. L. Hl7 ontology and ontological tools for semantic interoperability in healthcare. In *Proceedings of the 2006 IEEE International Conference on Systems, Man and Cybernetics* (2006), pp. 4347–4351.

[7] Dodds, K. React testing library: Simple and complete react dom testing utilities, 2024. React Testing Library Documentation.

[8] García-Crespo, A., et al. Snomed-ct: Building blocks for semantically enriched medical records. *IEEE Transactions on Information Technology in Biomedicine 14*, 1 (2010), 123–129.

[9] GitHub. Github: Where the world builds software, 2024. [Online]. Available: https://github.com/.

[10] Haxel, E. Vite: A modern build tool, 2021. Vite Documentation.

[11] Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., and Rudolph, S. *OWL 2 Web Ontology Language Primer*, 2012. W3C Recommendation.

[12] Kent, S. Vitest: A blazing fast unit testing framework, 2024. Vitest Documentation.

[13] Marsa, M., Matic, A., Calderita, L. V., and Rodriguez, F. J. Can-dr: Child abuse and neglect digital repository. *Computers in Human Behavior 94* (2019), 49–59.

[14] Marshall, J. A., Chen, J. R., and Istepanian, R. S. H. Medical ontology: Towards intelligent and integrated medical informatics systems. In *Proceedings of the 3rd IEEE International Symposium on Bioinformatics and Bioengineering* (2003), pp. 45–52.

[15] Nielsen, J. 10 usability heuristics for user interface design, 1995. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/.

[16] NSPCC. Child abuse and neglect statistics for england, 2021/22. [Online]. Available: https://www.nspcc.org.uk.

[17] O'Sullivan, J., et al. A knowledge engineering approach to deploying clinical diagnostic prediction models in healthcare workflows. *Scientific Reports 13*, 1 (2023), 1–17. [Online]. Available: https://www.nature.com/articles/s41598-023-34011-3.

[18] Poggi, A., Lembo, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Rosati, R. Linking data to ontologies. *Journal on Data Semantics 10* (2008), 133–173.

[19] Prud'hommeaux, E., and Seaborne, A. *SPARQL Query Language for RDF*, 2008. W3C Recommendation.

[20] Qin, S., Yu, H., Zeng, C., Zhao, Q., and Li, L. A semantic integration model for healthcare data: Standards, tools, and validation. *Procedia Computer Science 217* (2023), 666–675. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187628592300342X.

[21] Rubin, D. L., et al. National center for biomedical ontology: Advancing biomedicine through structured organization of scientific knowledge. *OMICS: A Journal of Integrative Biology 11*, 4 (2007), 348–352.

[22] SHACL Playground, Zazuko. Shacl playground. [Online]. Available: https://shacl-playground.zazuko.com/.

[23] Sommerville, I. *Software Engineering*, 9th ed. Addison-Wesley, Boston, MA, USA, 2011.

[24] Tilkov, S., and Vinoski, S. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing 14*, 6 (2010), 80–83.

[25] Visual Studio Code. Code editing. redefined., 2024. [Online]. Available: https://code.visualstudio.com/.

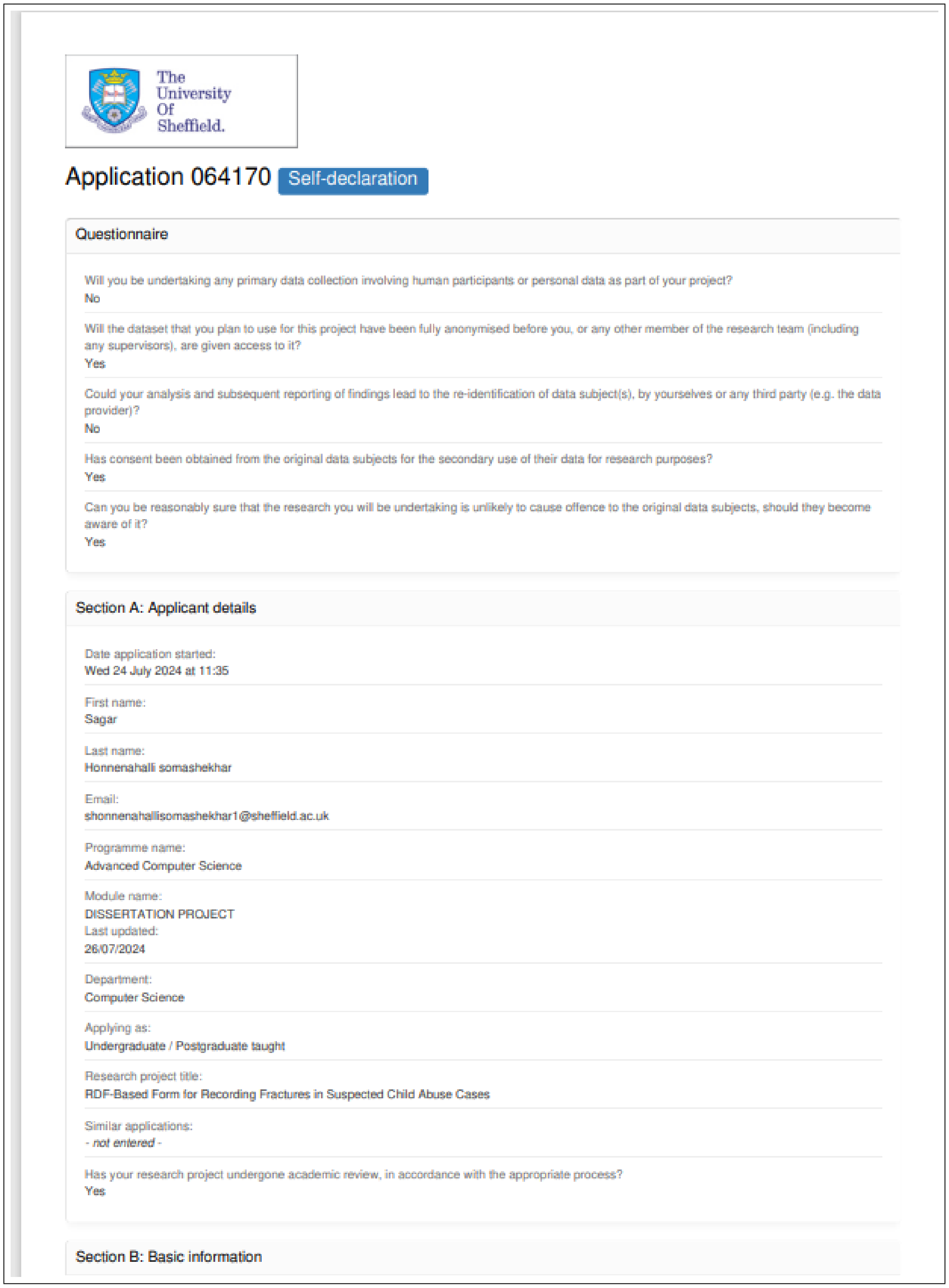
[26] World Wide Web Consortium. *SHACL - Shapes Constraint Language*, Jul. 20 2017. W3C Recommendation.

Appendices

Appendix A

Ethics Application

This Appendix contains the Ethics Application that was sent to the University of Sheffield Ethics department seeking approval for using anonymised secondary data.



The University Of Sheffield.

Application 064170 Self-declaration

Questionnaire

Will you be undertaking any primary data collection involving human participants or personal data as part of your project?
No

Will the dataset that you plan to use for this project have been fully anonymised before you, or any other member of the research team (including any supervisors), are given access to it?
Yes

Could your analysis and subsequent reporting of findings lead to the re-identification of data subject(s), by yourselves or any third party (e.g. the data provider)?
No

Has consent been obtained from the original data subjects for the secondary use of their data for research purposes?
Yes

Can you be reasonably sure that the research you will be undertaking is unlikely to cause offence to the original data subjects, should they become aware of it?
Yes

Section A: Applicant details

Date application started:
Wed 24 July 2024 at 11:35

First name:
Sagar

Last name:
Honnenahalli somashekhar

Email:
shonnenahallisomashekhar1@sheffield.ac.uk

Programme name:
Advanced Computer Science

Module name:
DISSERTATION PROJECT
Last updated:
26/07/2024

Department:
Computer Science

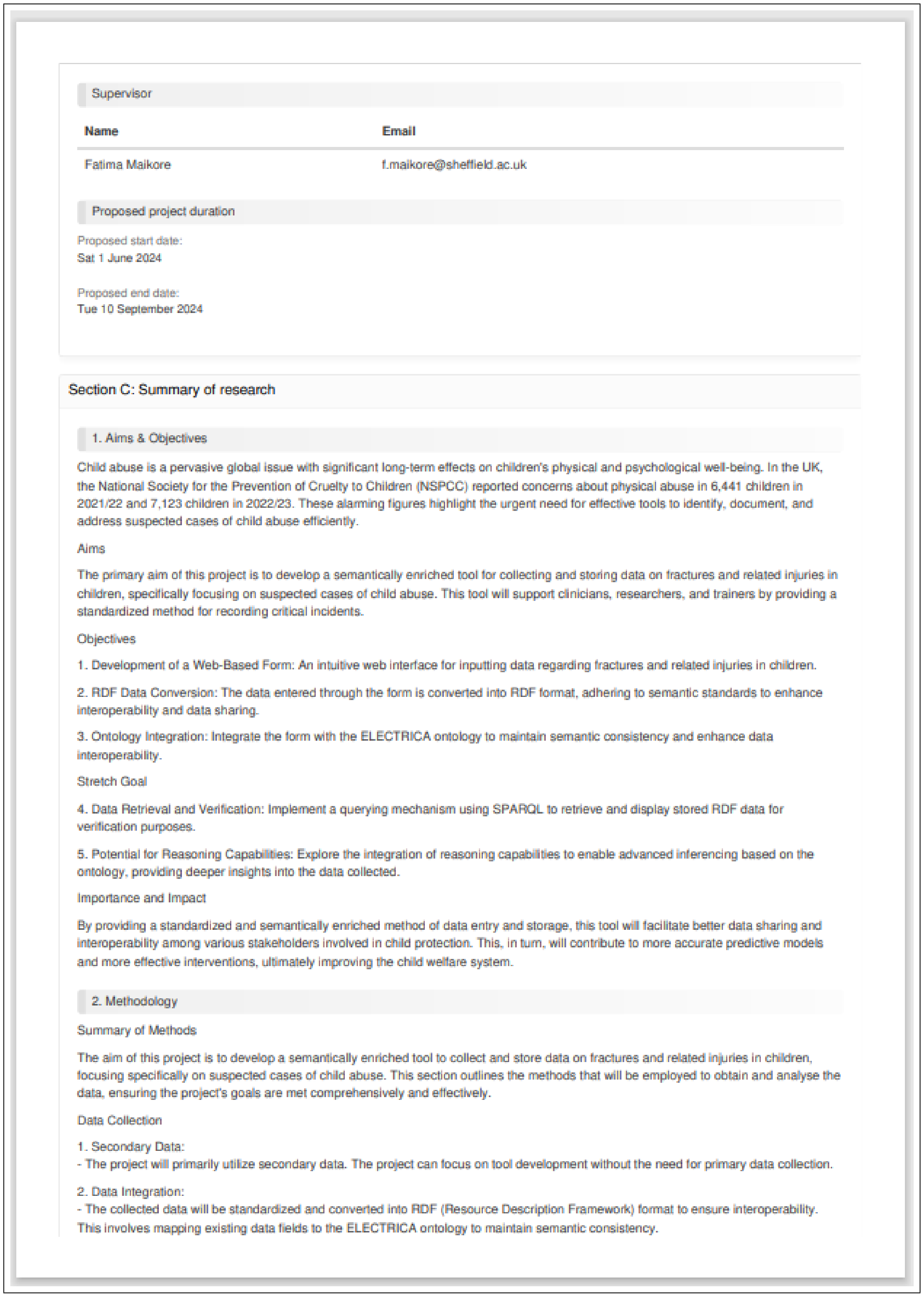
Applying as:
Undergraduate / Postgraduate taught

Research project title:
RDF-Based Form for Recording Fractures in Suspected Child Abuse Cases

Similar applications:
- not entered -

Has your research project undergone academic review, in accordance with the appropriate process?
Yes

Section B: Basic information



Supervisor

Name	Email
Fatima Maikore	f.maikore@sheffield.ac.uk

Proposed project duration

Proposed start date:
Sat 1 June 2024

Proposed end date:
Tue 10 September 2024

Section C: Summary of research

1. Aims & Objectives

Child abuse is a pervasive global issue with significant long-term effects on children's physical and psychological well-being. In the UK, the National Society for the Prevention of Cruelty to Children (NSPCC) reported concerns about physical abuse in 6,441 children in 2021/22 and 7,123 children in 2022/23. These alarming figures highlight the urgent need for effective tools to identify, document, and address suspected cases of child abuse efficiently.

Aims

The primary aim of this project is to develop a semantically enriched tool for collecting and storing data on fractures and related injuries in children, specifically focusing on suspected cases of child abuse. This tool will support clinicians, researchers, and trainers by providing a standardized method for recording critical incidents.

Objectives

1. Development of a Web-Based Form: An intuitive web interface for inputting data regarding fractures and related injuries in children.

2. RDF Data Conversion: The data entered through the form is converted into RDF format, adhering to semantic standards to enhance interoperability and data sharing.

3. Ontology Integration: Integrate the form with the ELECTRICA ontology to maintain semantic consistency and enhance data interoperability.

Stretch Goal

4. Data Retrieval and Verification: Implement a querying mechanism using SPARQL to retrieve and display stored RDF data for verification purposes.

5. Potential for Reasoning Capabilities: Explore the integration of reasoning capabilities to enable advanced inferencing based on the ontology, providing deeper insights into the data collected.

Importance and Impact

By providing a standardized and semantically enriched method of data entry and storage, this tool will facilitate better data sharing and interoperability among various stakeholders involved in child protection. This, in turn, will contribute to more accurate predictive models and more effective interventions, ultimately improving the child welfare system.

2. Methodology

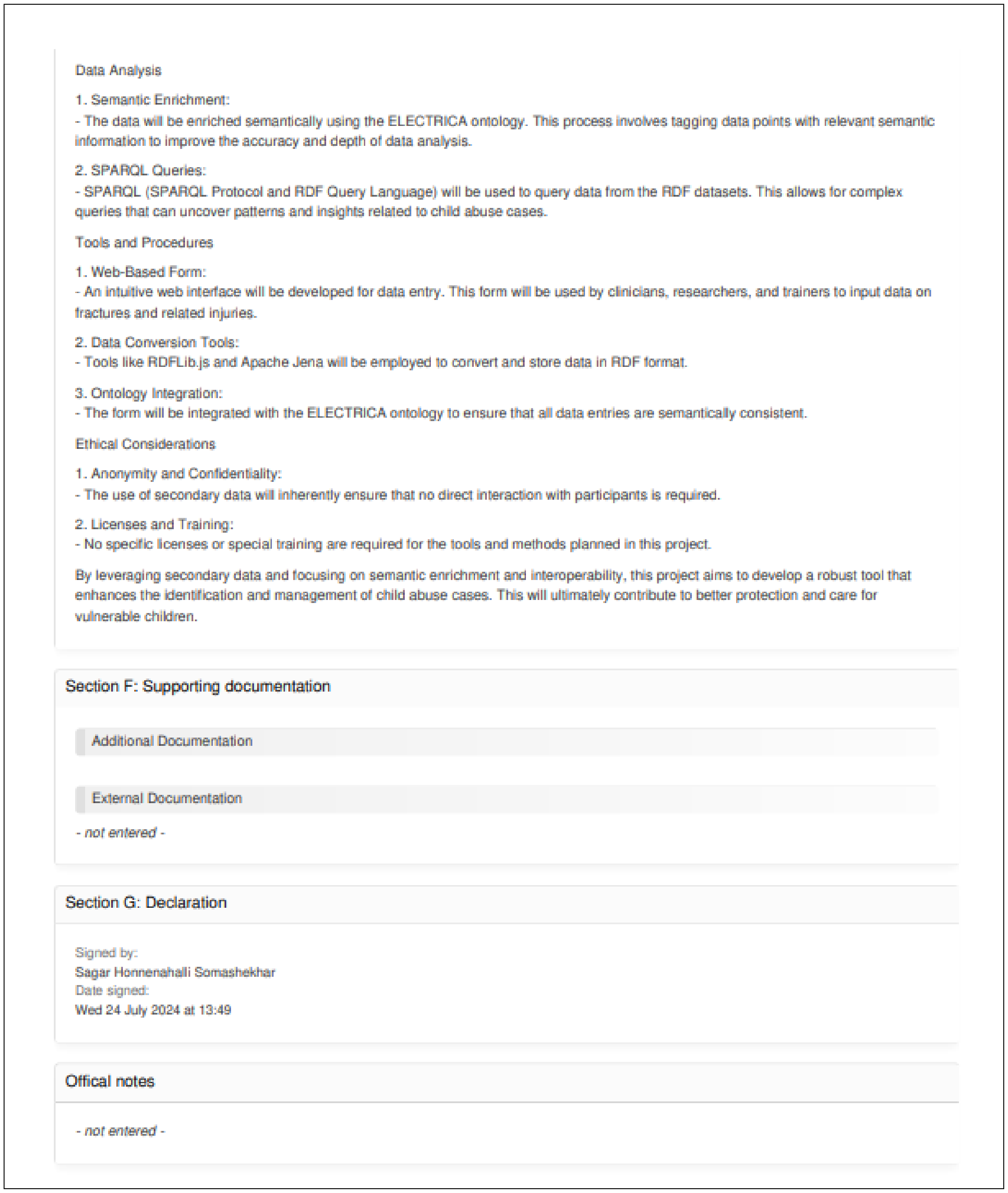
Summary of Methods

The aim of this project is to develop a semantically enriched tool to collect and store data on fractures and related injuries in children, focusing specifically on suspected cases of child abuse. This section outlines the methods that will be employed to obtain and analyse the data, ensuring the project's goals are met comprehensively and effectively.

Data Collection

1. Secondary Data:
- The project will primarily utilize secondary data. The project can focus on tool development without the need for primary data collection.

2. Data Integration:
- The collected data will be standardized and converted into RDF (Resource Description Framework) format to ensure interoperability. This involves mapping existing data fields to the ELECTRICA ontology to maintain semantic consistency.



Data Analysis

1. Semantic Enrichment:
- The data will be enriched semantically using the ELECTRICA ontology. This process involves tagging data points with relevant semantic information to improve the accuracy and depth of data analysis.

2. SPARQL Queries:
- SPARQL (SPARQL Protocol and RDF Query Language) will be used to query data from the RDF datasets. This allows for complex queries that can uncover patterns and insights related to child abuse cases.

Tools and Procedures

1. Web-Based Form:
- An intuitive web interface will be developed for data entry. This form will be used by clinicians, researchers, and trainers to input data on fractures and related injuries.

2. Data Conversion Tools:
- Tools like RDFLib.js and Apache Jena will be employed to convert and store data in RDF format.

3. Ontology Integration:
- The form will be integrated with the ELECTRICA ontology to ensure that all data entries are semantically consistent.

Ethical Considerations

1. Anonymity and Confidentiality:
- The use of secondary data will inherently ensure that no direct interaction with participants is required.

2. Licenses and Training:
- No specific licenses or special training are required for the tools and methods planned in this project.

By leveraging secondary data and focusing on semantic enrichment and interoperability, this project aims to develop a robust tool that enhances the identification and management of child abuse cases. This will ultimately contribute to better protection and care for vulnerable children.

Section F: Supporting documentation

Additional Documentation

External Documentation

- not entered -

Section G: Declaration

Signed by:
Sagar Honnenahalli Somashekhar
Date signed:
Wed 24 July 2024 at 13:49

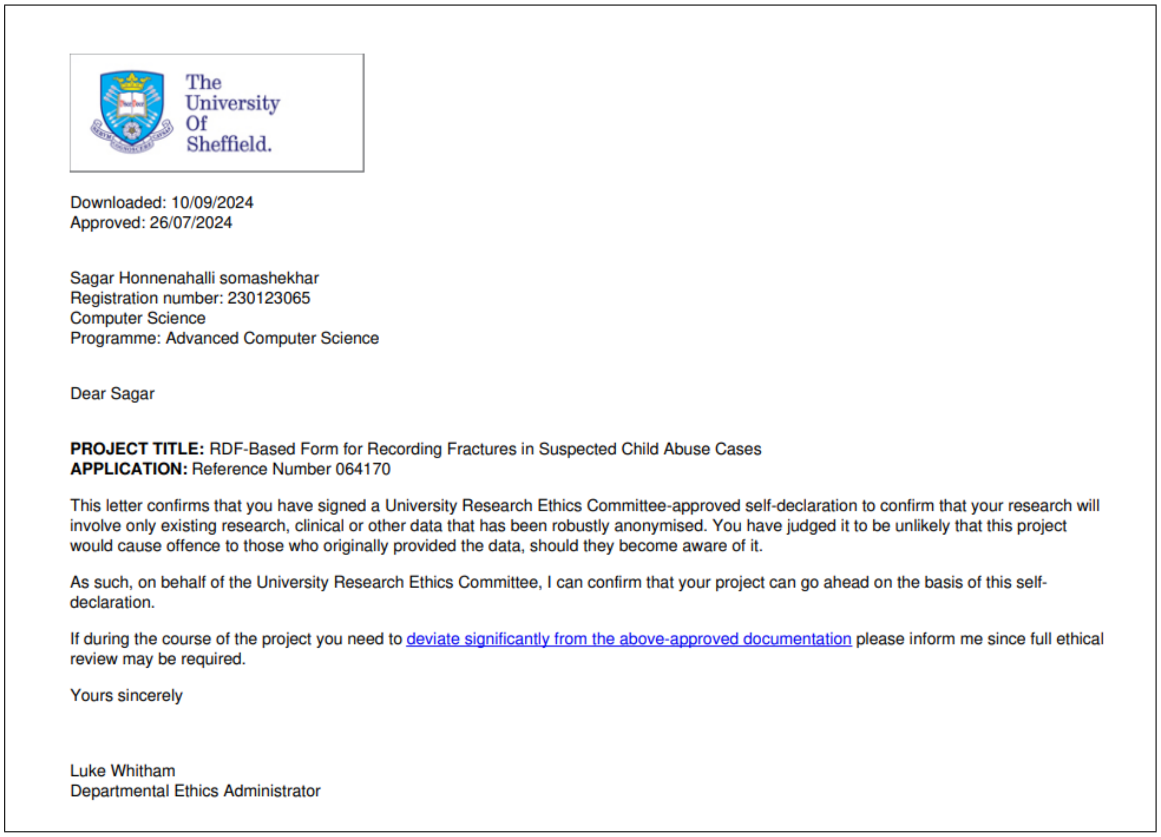
Offical notes

- not entered -

Appendix B

Ethics Approval

This Appendix contains the approval of my Ethics application by the Ethics department.



The University Of Sheffield.

Downloaded: 10/09/2024
Approved: 26/07/2024

Sagar Honnenahalli somashekhar
Registration number: 230123065
Computer Science
Programme: Advanced Computer Science

Dear Sagar

PROJECT TITLE: RDF-Based Form for Recording Fractures in Suspected Child Abuse Cases
APPLICATION: Reference Number 064170

This letter confirms that you have signed a University Research Ethics Committee-approved self-declaration to confirm that your research will involve only existing research, clinical or other data that has been robustly anonymised. You have judged it to be unlikely that this project would cause offence to those who originally provided the data, should they become aware of it.

As such, on behalf of the University Research Ethics Committee, I can confirm that your project can go ahead on the basis of this self-declaration.

If during the course of the project you need to deviate significantly from the above-approved documentation please inform me since full ethical review may be required.

Yours sincerely

Luke Whitham
Departmental Ethics Administrator