NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES
“COMPUTER SCIENCE”

MASTER THESIS

Managing the Data Modeling Lifecycle with the DOLAR Type-Manager Web-app

Ioanna P. Mourtzaki

Supervisor: **Alexis Delis,** Professor
Kostas Saidis, Visiting Lecturer

Athens

March 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
«ΠΛΗΡΟΦΟΡΙΚΗ»

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Διαχείριση του κύκλου ζωής των μοντέλων με χρηση του DOLAR Type Manager Web-app

Ιωάννα Π. Μουρτζάκη

Επιβλέπων: **Αλέξης Δελής**, Καθηγητής
Κώστας Σαΐδης, Επιτετραμμένος Λέκτορας

Αθήνα

Μάρτιος 2024

MASTER THESIS

Managing the Data Modeling Lifecycle with the DOLAR Type-Manager Web-app

Ioanna P. Mourtzaki
R.N.: cs22200021

SUPERVISOR: **Alexis Delis,** Professor
Kostas Saidis, Visiting Lecturer

THESIS COMMITEE: **Alexis Delis,** Professor
Panagiotis Rondogiannis, Professor

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Διαχείρηση του κύκλου ζωής των μοντέλων με χρηση του DOLAR Type Manager Web-app

Ιωάννα Π. Μουρτζάκη
Α.Μ.: cs22200021

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής**, Καθηγητής
Κώστας Σαΐδης, Επιτετραμμένος Λέκτορας

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Αλέξης Δελής**, Καθηγητής
Παναγιώτης Ροντογιάννης, Καθηγητής

ABSTRACT

In today's rapidly evolving digital landscape, the volume, diversity and variety of information continue to expand at an unprecedented pace. Organizations face the challenge of effectively managing their ever-growing datasets, which are often held in different representations, types, structures, and formats. The DOLAR (Data Object Language And Runtime) framework offers a comprehensive solution for managing the complexities of information expansion by virtualizing the information space. DOLAR introduces a virtual object environment, which serves as an abstraction layer between the data model, the data store and the application/business logic. This abstraction allows for seamless integration of new datasets into existing applications, as well as straight-forward reuse of existing datasets to new applications.

At the core of the DOLARL lies a storage-agnostic representation of (a) the data models (called DOLAR prototypes) and (b) the objects instantiated by these models (called virtual objects). The prototypes encapsulate essential data characteristics such as field types, their characteristics and their relationships and the framework supports multiple, distinct yet equivalent, syntactic representations for defining and storing the proptotypes (namely XML, JSON and the FLY domain-specific language). DOLAR also offers a Type Manager extension, a seamless DOLAR prototype introspection and modification API, that uniformly hides the underpinnings of the underlying DOLAR prototype syntax.

This thesis builds upon DOLAR's Type Manager extension to introduce the Type Manager Web App, a user-friendly and intuitive interface for managing DOLAR prototypes. This web app is developed using Java Spring Boot for the backend endpoints and React framework for the development of the user interface. Additionally, it is packaged as an Electron application, providing a self-contained desktop experience. Users can manage various aspects of DOLAR prototypes through the Type Manager Web App, including creating, modifying, and deleting prototypes, managing inheritance relationships, as well as executing batch actions that resemble refactorings, traditionally performed by application developers through IDEs. The application enables users to interact with DOLAR prototypes effortlessly through a web-based interface, without having to learn a custom syntax or new language. The integration of Spring Boot ensures robust and efficient backend functionality, while React facilitates a responsive and interactive frontend experience. The Electron framework allows for the distribution of the Type Manager Web App as a standalone desktop application, enhancing convenience and flexibility.

SUBJECT AREA: Data modeling

KEYWORDS: data representation, DOLAR prototypes virtual objects, web application

ΠΕΡΙΛΗΨΗ

Στην σημερινή εποχή της πληροφορίας που διανύουμε είναι γνωστό ότι ο όγκος και η ποικιλία των διαθέσιμων δεδομένων επεκτείνονται συνεχώς με ραγδαίους ρυθμούς και αυτό καθιστά ζωτικής σημασίας την αποτελεσματική διαχείριση των πληροφοριών. Ωστόσο, οι παραδοσιακές προσεγγίσεις στην διαχείριση πληροφοριών αδυνατούν να συμβαδίσουν με τη δυναμική φύση των δεδομένων. Μία από τις κύριες προκλήσεις είναι η στενή σύζευξη που υπάρχει μεταξύ της επιχειρηματικής λογικής, και της αποθήκευσης και αναπαράστασής της ίδιας της πληροφορίας. Αυτή η σύζευξη καθιστά δύσκολη την εισαγωγή νέων τύπων δεδομένων ή την τροποποίηση υφιστάμενων δομών. Για την αντιμετώπιση αυτού του προβλήματος και την αποτελεσματικότερη διαχείρηση πληροφοριών, ο διαχωρισμός της επιχειρηματικής λογικής από την αποθήκευση και την αναπαράσταση πληροφοριών έχει αναγνωριστεί ως λύση για την επίτευξη ευελιξίας, επεκτασιμότητας και προσαρμοστικότητας. Η παρούσα εργασία εστιάζει στον διαχωρισμό της λογικής της πληροφορίας από την αποθήκευση και την αναπαράστασή της, στο πλαίσιο του συστήματος DOLAR (Dynamic Object-oriented Virtual Information Space). Το DOLAR παρέχει μια ολοκληρωμένη λύση για τη διαχείριση πολύπλοκων πληροφοριών, αξιοποιώντας ένα περιβάλλον εικονικού χώρου πληροφοριών, όπου η πληροφορία υπάρχει, ανακτάται και τροποποείται, χωρίς να χρειάζεται άμεση πρόσβαση στην ίδια την αναπαράσταση της πληροφορίας, καθώς και στον χώρο στον οποίο έχει αποθηκευτεί. Για τη βελτίωση της διαχείρισης των πρωτοτύπων DOLAR, η εργασία παρουσιάζει την εφαρμογή Type Manager Web App, η οποία αναπτύχθηκε με χρήση Java Spring Boot και React. Η εφαρμογή αυτή συνεισφέρει στην αφαίρεση της πολυπλοκότητας της διαχείρησης πληροφοριών, επιτρέποντας στους χρήστες να αλληλεπιδρούν με τα DOLAR μοντέλα και να τα επεξεργάζονται εύκολα, μέσω μιας web διεπαφής. Συγκεκριμένα, η εργασία ασχολείται με την αρχιτεκτονική της εφαρμογής, το σχεδιασμό της διεπαφής (UI Design), καθώς και την υλοποίηση της. Επιπλέον, παραθέτει ένα end-to-end παράδειγμα για να περιγράψει τη διαδικασία τροποποίησης ενός DOLAR μοντέλου μέσω της εφαρμογής. Διαχωρίζοντας την επιχειρηματική λογική από την αποθήκευση και την αναπαράσταση πληροφοριών, η εφαρμογή Type Manager Web App προσφέρει μια ολοκληρωμένη λύση για τη διαχείριση των DOLAR μοντέλων, διασφαλίζοντας έτσι επεκτασιμότητα και προσαρμοστικότητα ενόψει των διευρυνόμενων απαιτήσεων της πληροφορίας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μοντελοποίηση δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: αναπαράσταση δεδομένων, ψηφιακή πληροφορία, web application

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In today's rapidly evolving digital landscape, the volume and variety of information continue to expand at an unprecedented pace. Organizations face the challenge of effectively managing this ever-growing pool of data, which often comes in several types, structures, and formats [1]. Traditional approaches to information management struggle to keep up with these dynamics, leading to complex development processes, increased costs, and difficulties in adapting to changing business requirements. [2] [3] [4]

One key issue is the tightly coupled nature of business logic with the storage and representation of information. In many systems, the data model and the underlying storage technology are intricately intertwined, making it challenging to introduce new data types or modify existing structures without significant effort and potential disruptions. This tight coupling creates dependencies and limits the flexibility needed to accommodate the expanding information landscape. [5]

To address these challenges, researchers and practitioners have recognized the need for decoupling the business logic of information from its storage and representation [6]. By separating the logical model from the physical implementation, organizations can achieve greater flexibility, scalability, and adaptability when dealing with diverse data types and evolving business requirements.

The DOLAR (Data Object Language And Runtime) framework, introduced by Saidis, Smaragdakis, and Delis offers a comprehensive solution for managing the complexities of information expansion [7]. DOLAR leverages a virtual information space environment, which serves as an abstraction layer between the data model and the actual business logic. This abstraction allows for seamless integration of new data types and structures into existing applications, without the need for knowing and understanding the data model representation.

At the core of the DOLAR lies a storage-agnostic representation of a) the data models (called DOLAR prototypes) and b) the data objects instantiated by these models (called virtual objects). These prototypes encapsulate essential characteristics, relationships, and field information in different syntactic representations, abstracting away the complexities of data representation. This empowers users to define and modify models in terms of different, yet equivalent, syntactic representations. The framework currently supports three variations, namely XML, JSON, and FLY.

To augment the capabilities of the DOLAR framework, a Type Manager Extension has been developed, that offers a seamless DOLAR prototype introspection and modification API, regardless of the underlying DOLAR prototype syntax. For example, the action “add field X to type Y” can be triggered through the Type Manager and it will seamlessly perform this action in any of the three syntactic variations that is currently selected.

This API offers methods that facilitate prototype refactoring actions swiftly and accurately. Users can manage various aspects of DOLAR prototypes through this API, including creating, modifying, and deleting prototypes, managing inheritance relationships, fields, and field groups, as well as executing batch actions for efficient bulk operations.

Building upon the DOLAR framework and the Type Manager Extension, this thesis introduces the Type Manager Web App. This web app is developed using Java Spring Boot for the backend endpoints and React framework for the development of the user interface. Additionally, it is packaged as an Electron application, providing a self-contained desktop experience.

The Type Manager Web App provides a user-friendly interface for managing DOLAR prototypes. By abstracting away the complexities of backend operations, the application enables users to interact with DOLAR prototypes effortlessly through a web-based interface. The integration of Spring Boot ensures robust and efficient backend functionality, while React facilitates a responsive and interactive frontend experience. The Electron framework allows for the distribution of the Type Manager Web App as a standalone desktop application, enhancing convenience and flexibility.

This thesis explores the architecture, user interface design and implementation of the Type Manager Web App. It delves into the backend RESTful APIs developed with Spring Boot, the frontend UI design implemented in React, and their asynchronous communication logic. Additionally, an end-to-end example is included to highlight the process of adding a field to a DOLAR prototype. The thesis concludes with a discussion regarding the web app, possibilities of future work, and some references.

This thesis aims to provide a comprehensive solution for decoupling business logic from information storage and representation. This enables developers and organizations to effectively manage the complexities of prototype refactoring and data modeling, ensuring scalability and adaptability in the face of expanding information requirements.

2. DOLAR DATA MODELING LIFECYCLE

The DOLAR (Data Object Language And Runtime) framework offers a comprehensive solution to the complexities of information expansion. By introducing a virtual information space environment, DOLAR harnesses the power of automation and abstraction to facilitate the integration of novel data types within existing applications. This approach significantly mitigates the expenses associated with expansion while ensuring the scalability and adaptability of the system. [6]

2.1 DOLAR Prototypes

At its core, DOLAR revolves around the concept of DOLAR prototypes and Virtual Objects, which serve as the fundamental building blocks for representing and manipulating data within diverse digital environments. DOLAR is based on a revolutionary approach to data modeling that lies a storage-agnostic representation of a) the data models (called DOLAR prototypes) and b) the objects instantiated by these models (called virtual objects). One of the most compelling aspects of DOLAR is its ability to abstract away the complexities of data representation, empowering users to define and adjust models through diverse but interchangeable syntactic representations. Presently, the framework offers support for three variations: XML, JSON, and FLY.

In essence, DOLAR enables users to focus solely on defining or refining data models according to their requirements, without needing to delve into the underlying intricacies of how these models are represented internally. The beauty of DOLAR lies in its utilization of storage-agnostic representation as a standardized format for defining and representing data models. Each representation contains attributes crucial for the model, including relationships, fields, schemes, and other essential details.

In other words, DOLAR prototypes provide a logical abstraction of data stored in various artifacts, fostering a unified and standardized perspective regardless of underlying storage intricacies. This cohesive approach could be illustrated through the detailed depiction of a DOLAR prototype, named ‘Photo’. Below, you will find representations of the 'Photo' prototype in both XML, JSON, and FLY formats. These representations elucidate the structured definitions and attributes encapsulated within each format,

exemplifying the versatility and adaptability of DOLAR prototypes in managing and representing complex data structures.

XML representation

```xml
<?xml version="1.0" encoding="utf-8"?>

<prototype>
    <inherits id="Model"/>
    <inherits id="CommonMetadata"/>
    <inherits id="ItemBase"/>
    <inherits id="PhotoContainer"/>

    <label lang="el">Φωτογραφία</label>
    <label lang="en">Photo</label>

    <meta id="Factory" />
    <meta id="Indexable" />

    <field id="caption">
        <label lang="el">Λεζάντα</label>
        <label lang="en">Caption</label>
        <type>TEXT</type>
        <map>true</map>
        <meta id="sortable"/>
    </field>

    <field id="depictedSubject">
        <label lang="el">Απεικονιζόμενο θέμα</label>
        <label lang="en">Depicted subject</label>
        <type>TEXT</type>
    </field>


    <fieldGroup id="migration_files">
        <field id="filepath">
            <label lang="en">Filepath</label>
            <type>TEXT</type>
        </field>
        <field id="filename">
            <label lang="en">Filename</label>
            <type>TEXT</type>
        </field>
    </fieldGroup>

</prototype>
```

JSON representation

```json
{
  "id" : "Photo",
  "inherits" : [ "CommonMetadata", "Core_Node", "Core_Timestamps", "ItemBase",
"Model", "PhotoContainer" ],
  "labels" : [ {
    "lang" : "el",
    "value" : "Φωτογραφία"
  }, {
    "lang" : "en",
    "value" : "Photo"
  } ],
  "meta" : [ {
    "id" : "Factory",
    "value" : ""
  }, {
    "id" : "Indexable",
    "value" : ""
  } ],
  "fields" : [ {
    "id" : "parents",
```

```
    "meta" : [ ],
    "type" : "REF",
    "transient" : false,
    "array" : true,
    "map" : false,
    "constraint" : "/butterfly/core/Container",
    "defaultValue" : ""
  }, {
    "id" : "address",
    "labels" : [ {
      "lang" : "el",
      "value" : "Διεύθυνση"
    }, {
      "lang" : "en",
      "value" : "Address"
    } ],
    "meta" : [ ],
    "type" : "BIGTEXT",
    "transient" : false,
    "array" : false,
    "map" : false,
    "constraint" : "",
    "defaultValue" : ""
  }, {
    "id" : "caption",
    "labels" : [ {
      "lang" : "el",
      "value" : "Λεζάντα"
    }, {
      "lang" : "en",
      "value" : "Caption"
    } ],
    "meta" : [ ],
    "type" : "TEXT",
    "transient" : false,
    "array" : false,
    "map" : true,
    "constraint" : "",
    "defaultValue" : ""
  }, {
    "id" : "depictedSubject",
    "labels" : [ {
      "lang" : "el",
      "value" : "Απεικονιζόμενο θέμα"
    }, {
      "lang" : "en",
      "value" : "Depicted subject"
    } ],
    "meta" : [ ],
    "type" : "TEXT",
    "transient" : false,
    "array" : false,
    "map" : false,
    "constraint" : "",
    "defaultValue" : ""
  } ],
  "fieldGroups" : [ {
    "id" : "timestamps",
    "meta" : [ ],
    "fields" : [ {
      "id" : "createdAt",
      "labels" : [ {
        "lang" : "en",
        "value" : "Creation timestamp (milliseconds since the Epoch)"
      } ],
      "meta" : [ ],
      "type" : "NUMBER",
      "transient" : false,
      "array" : false,
      "map" : false,
      "constraint" : "",
```

```
      "defaultValue" : ""
    }, {
      "id" : "modifiedAt",
      "labels" : [ {
        "lang" : "en",
        "value" : "Modification timestamp (milliseconds since the Epoch)"
      } ],
      "meta" : [ ],
      "type" : "NUMBER",
      "transient" : false,
      "array" : false,
      "map" : false,
      "constraint" : "",
      "defaultValue" : ""
    } ]
  }, {
    "id" : "files",
    "labels" : [ {
      "lang" : "el",
      "value" : "Αρχεία"
    }, {
      "lang" : "en",
      "value" : "Files"
    } ],
    "meta" : [ ],
    "fields" : [ {
      "id" : "thumb",
      "labels" : [ {
        "lang" : "el",
        "value" : "Μικρογραφία"
      }, {
        "lang" : "en",
        "value" : "Thumbnail"
      } ],
      "meta" : [ ],
      "type" : "REF",
      "transient" : false,
      "array" : false,
      "map" : false,
      "constraint" : "/butterfly/core/JpegFile|/butterfly/core/PngFile",
      "defaultValue" : ""
    }, {
      "id" : "hqFile",
      "labels" : [ {
        "lang" : "el",
        "value" : "Εικόνα υψηλής ποιότητας"
      }, {
        "lang" : "en",
        "value" : "High quality image file"
      } ],
      "meta" : [ ],
      "type" : "REF",s
      "transient" : false,
      "array" : false,
      "map" : false,
      "constraint" :
"/butterfly/core/TiffFile|/butterfly/core/JpegFile|/butterfly/core/PngFile",
      "defaultValue" : ""
    }, {
      "id" : "webFile",
      "labels" : [ {
        "lang" : "el",
        "value" : "Εικόνα web ποιότητας"
      }, {
        "lang" : "en",
        "value" : "Web image file"
      } ],
      "meta" : [ ],
      "type" : "REF",
      "transient" : false,
      "array" : false,
```

```
      "map" : false,
      "constraint" : "/butterfly/core/JpegFile|/butterfly/core/PngFile",
      "defaultValue" : ""
    } ]
  }, {
    "id" : "migration_files",
    "meta" : [ ],
    "fields" : [ {
      "id" : "filepath",
      "labels" : [ {
        "lang" : "en",
        "value" : "Filepath"
      } ],
      "meta" : [ ],
      "type" : "TEXT",
      "transient" : false,
      "array" : false,
      "map" : false,
      "constraint" : "",
      "defaultValue" : ""
    }, {
      "id" : "filename",
      "labels" : [ {
        "lang" : "en",
        "value" : "Filename"
      } ],
      "meta" : [ ],
      "type" : "TEXT",
      "transient" : false,
      "array" : false,
      "map" : false,
      "constraint" : "",
      "defaultValue" : ""
    } ]
  } ]
}
```

FLY representation

```
@annotate(
     label: map("en","Photo", "el","Φωτογραφία"),
     Factory: "true",
     Indexable: "true"
)
type Photo inherits data.Model, data.CommonMetadata, data.ItemBase,
niovity.butterfly.core.PhotoContainer, niovity.butterfly.core.DCItem {
    @annotate(
       label: map("en","Caption", "el","Λεζάντα"),
       sortable: "true"
     )
     caption Map(Text)

    @annotate(
       label: map("en","Depicted subject", "el","Απεικονιζόμενο θέμα")
    )
    depictedSubject Text

    @annotate(migration_files) {
        @annotate(
          label: map("en","Filepath"),
          sortable: "true"
```

```
        )
        filepath Map(Text)

    @annotate(
       label: map("en","Filename"),
       sortable: "true"
    )
    fileName Map(Text)
  }
}
```

The XML representation of the 'Photo' prototype provides a structured and hierarchically organized definition of its attributes and properties. Each element within the XML document delineates specific characteristics of the prototype, such as labels in different languages, metadata, fields with their respective identifiers, types, and constraints.

At the same time, the JSON representation offers a concise and compact depiction of the 'Photo' prototype's attributes in a key-value pair format. Each attribute is encapsulated within its corresponding identifier, accompanied by relevant labels, metadata, and constraints. This representation displays the versatility of JSON in representing complex data structures with minimal verbosity.

Regarding the FLY representation, the declaration introduces the prototype named 'Photo', which inherits attributes and functionalities from foundational prototypes. Annotations are included throughout the representation to provide metadata and information for handling each attribute within the DOLAR framework.

All representations of the ‘Photo’ prototype contain the following information:

Inheritance and Property Definitions

Within all representations, the 'Photo' prototype is articulated as inheriting foundational properties from prototypes such as "Model," "CommonMetadata," and "ItemBase." This inheritance establishes the core attributes and functionalities of the 'Photo' prototype, ensuring consistency and coherence across various implementations.

Multilingual Labeling

All representations uphold the principle of multilingual labeling, catering to diverse linguistic contexts. Labels provided in both Greek and English languages enhance accessibility and usability for a wider audience, accommodating users with different language preferences or requirements.

Metadata Specification

Metadata elements such as "Factory" and "Indexable" are consistently defined and incorporated into the representations. These metadata specifications offer contextual information crucial for the efficient management and utilization of 'Photo' prototype instances, ensuring that relevant metadata is readily available for downstream processes.

Field Definitions and Grouping

In the representations, fields encapsulating specific data attributes, such as "caption" and "depictedSubject," are meticulously defined. Additionally, field grouping is employed to enhance organizational clarity and facilitate modular design principles. These

groupings aid in structuring and organizing data attributes, promoting maintainability and extensibility of the prototype definitions.

In essence, all the syntactic representations maintain equivalence in terms of fundamental attributes, properties, and structural coherence. All representations collectively offer a comprehensive and unified depiction of the "Photo" prototype, underscoring its versatility and adaptability within the proposed system architecture.

2.2 Type Manager Extension

To augment the capabilities of the DOLAR framework, a Type Manager Extension has been developed, offering a seamless DOLAR prototype introspection and modification API. This extension enhances the framework's flexibility by providing users with the ability to interact with DOLAR prototypes regardless of the underlying syntax variation. For instance, actions such as "add field X to type Y" can be effortlessly executed through the Type Manager, seamlessly accommodating any of the three syntactic variations currently selected.

When executing the "add field X to type Y" operation in the Photo prototype through the Type Manager Extension, the XML representation, for example, will be updated to include the newly added field. For instance, upon adding the "captureDate" field to the Photo prototype (“add field captureDate to type Photo”), the XML representation would incorporate the following definition:

```
<field id="captureDate">
    <type>DATE</type>
    <transient>false</transient>
    <array>false</array>
    <map>false</map>
</field>
```

This XML snippet illustrates how the Type Manager Extension seamlessly integrates modifications into the XML representation of DOLAR prototypes, ensuring consistency and coherence across all representations.

2.3 DOLAR Prototype Lifecycle

In addition to simplifying prototype management, the API offered by the Type Manager Extension streamlines prototype refactoring actions swiftly and accurately. Users can leverage a variety of methods provided by the API to manage various aspects of DOLAR prototypes, enhancing their flexibility and efficiency in handling data modeling tasks. These methods include creating, modifying, and deleting DOLAR prototypes, as well as managing inheritance relationships, fields, and field groups.

With the API, users gain granular control over prototype management tasks. They can retrieve DOLAR prototypes by namespace, allowing for organized and efficient access to prototype information. Detailed information about specific prototypes can be fetched, providing users with comprehensive insights into prototype attributes and configurations. Moreover, users can seamlessly create new prototypes or delete existing ones, facilitating the adaptation of the data model to evolving requirements.

The API empowers users to establish or remove inheritance relationships between prototypes, enabling flexible and dynamic modeling of data structures. This capability is essential for maintaining coherence and consistency within the data model while accommodating changes and updates over time. Additionally, users can manage fields and field groups with ease, defining and organizing data attributes according to specific requirements.

Furthermore, the API supports batch actions for efficient bulk operations, allowing users to execute repetitive tasks quickly and effortlessly. This feature enhances productivity by automating routine operations and reducing manual effort. Overall, the comprehensive set of methods provided by the API facilitates agile and efficient data modeling workflows, empowering users to design and manage DOLAR prototypes effectively.

The Type Manager Extension encapsulates the core functionality necessary for DOLAR prototype refactoring, leveraging the robustness and efficiency of command-line interactions. By housing the actual actions methods, the Type Manager Extension ensures that prototype refactoring processes are executed swiftly and accurately, without compromising on performance or reliability.

It is worth noting that the approach taken in developing the Type Manager Extension bears resemblance to the principles underlying the Language Server Protocol (LSP) utilized in Integrated Development Environments (IDEs). Just as LSP abstracts the language-specific features of IDEs, allowing for the development of language-independent tools and services, the Type Manager Extension abstracts the syntactic complexities of DOLAR prototypes, providing a unified interface for managing and refactoring prototypes irrespective of their underlying syntax. This abstraction layer fosters interoperability and extensibility, enabling the integration of the DOLAR framework with diverse tools and systems while maintaining consistency and coherence in data modeling workflows. [8]

By adopting a similar philosophy to LSP, the Type Manager Extension enhances the accessibility and usability of DOLAR prototype management, enabling users to interact with prototypes consistently and intuitively.

In summary, the discussion surrounding the DOLAR prototypes approach and the innovative Type Manager Extension underscores the importance of flexible frameworks and efficient management tools in modern software development. By decoupling application logic from contextual elements and providing users with streamlined methods for prototype management and refactoring, significant advancements can be made in terms of system flexibility, scalability, and usability.

3. TYPE MANAGER WEB APP

3.1 Overview

Built upon the foundation laid by the DOLAR framework and the Type Manager Extension, this thesis introduces the Type Manager Web App – an intuitive interface designed to support prototype refactoring processes. Developed using modern technologies, the Type Manager Web App leverages the Type Manager Extension actions to create a seamless user experience for managing DOLAR prototypes.

The development of the Type Manager Web App was motivated by the need to enhance user experience and accessibility in managing DOLAR prototypes within the existing Type Manager Extension. While the Type Manager Extension provided powerful functionality for prototype refactoring, its command-line interface presented a steep learning curve and required users to have a comprehensive understanding of the underlying syntactic representation of prototypes. The Type Manager Web App aims to address these limitations by providing a user-friendly web-based interface that abstracts away the complexities of the backend operations. By leveraging the power of Spring Boot and React, the app offers a seamless and interactive experience for users to create, modify, and manage DOLAR prototypes. Furthermore, by packaging the app as

an Electron application, it can be distributed as a standalone desktop app, providing convenience and flexibility to users.

The Type Manager Web App abstracts away the complexities of backend operations, allowing users to interact with DOLAR prototypes effortlessly through a user-friendly web interface. By consuming the endpoints provided by the Type Manager Extension, the Type Manager Web App extends the accessibility and usability of prototype refactoring functionalities to a broader audience.

Through intuitive UI components and streamlined workflows, users can view, create, modify, and delete prototypes with ease. The Type Manager Web App ensures that users can execute prototype refactoring actions without needing to delve into the intricacies of backend operations or command-line interactions. Instead, users can focus on defining and refining data models according to their requirements, fostering agility and efficiency in data modeling tasks. In essence, the Type Manager Web App serves as a bridge between backend actions methods and frontend user interfaces, offering a holistic solution for prototype refactoring and data modeling within the DOLAR ecosystem.

One of the standout features of the Type Manager Web App is its isolation of the prototype representation, which enhances user experience and simplifies data management. Users can engage in prototype refactoring without needing to know the intricate details of the underlying model representation. Instead, they interact with the user interface, which abstracts away the complexities of backend operations, thus streamlining the refactoring process.

Another notable aspect of the Type Manager Web App is its flexibility in executing prototype refactoring actions. Users have the option to execute each action separately - such as adding a field, editing a field, or refactoring inheritance—or to bundle multiple actions into a batch and execute them together with a single call to the backend. This capability streamlines the refactoring process, allowing users to execute complex actions efficiently and effectively.

One other important feature of the Type Manager Web App is its standalone nature, made possible by Electron. Users can access the application as a self-contained desktop application, eliminating the need for constant internet connectivity or reliance on web browsers. This standalone functionality enhances user convenience and flexibility, allowing users to manage DOLAR prototypes seamlessly across various computing environments.

Through its utilization of Electron, the Type Manager Web App offers a robust and versatile solution for managing DOLAR prototypes. By combining the flexibility of web technologies with the power of native desktop integration, the application delivers a seamless user experience, empowering users to streamline prototype refactoring tasks with ease.

3.2 Architecture

The architecture of the Type Manager Web App is carefully designed to provide a robust and efficient framework for prototype management. At its core, the app is divided into two main components: the Front-End and the Back-End.

The Front-End is developed using React, a popular JavaScript library for building user interfaces. It leverages the Electron framework, enabling the creation of cross-platform desktop applications. The use of React allows for the construction of modular components, including the Side Menu, the Modals, and the Notifications. This

component-based approach ensures a clean and organized codebase, making it easier to manage and scale the application.

The Back-End is powered by Java Spring Boot, a versatile framework that simplifies the development of production-ready applications. Within the Back-End, API endpoints are defined through controllers, which act as the interface between the Front-End and the core application logic. The service layer encapsulates the intricate details of prototype management, handling actions such as creating, updating, and deleting DOLAR prototypes. The Spring Boot Controller orchestrates these actions, ensuring a seamless and efficient process.

The constructive collaboration between the Front-End and Back-End is key to the app's success. The Front-End communicates with the Back-End through well-defined API endpoints, enabling the exchange of data and actions. This integration ensures a responsive and interactive user experience, with changes in the Front-End triggering corresponding actions in the Back-End.

The modular nature of the architecture enhances scalability and maintenance. Each component, from the smallest UI element to the entire Navigation Menu of the application, is encapsulated within its own module. This modular approach simplifies updates, bug fixes, and the addition of new features, contributing to the overall agility of the development process.

The following diagram provides a high-level overview of the system architecture:

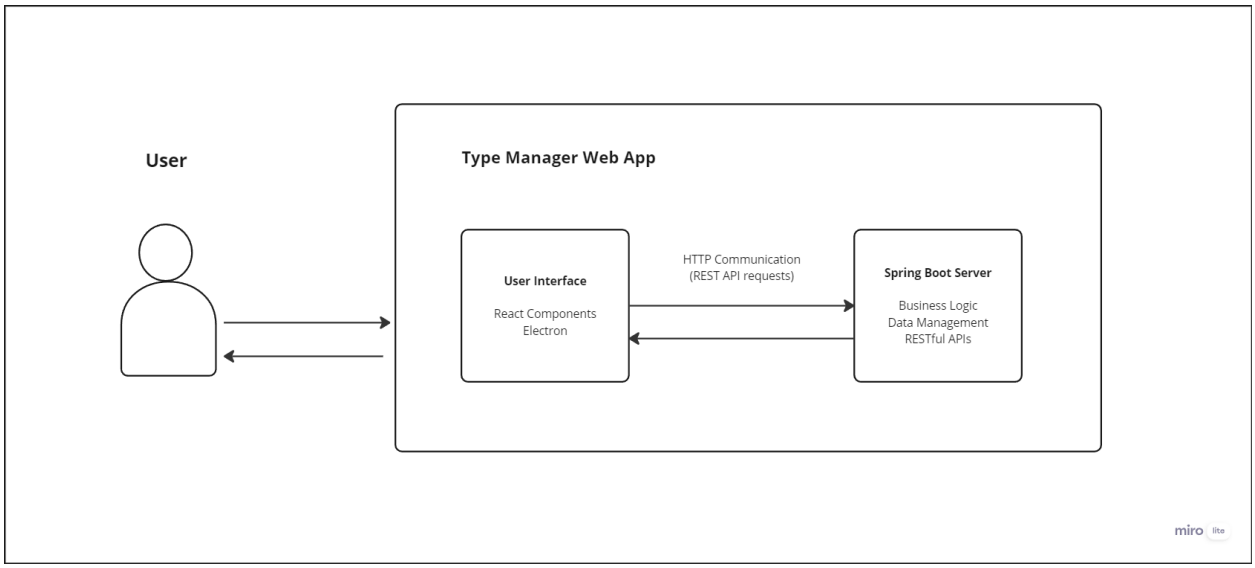

Figure 1: Type Manager Web App Architecture

In summary, the architecture of the Type Manager Web App is a carefully orchestrated blend of cutting-edge technologies, aiming to provide users with a seamless experience in managing DOLAR prototypes. The division of responsibilities between the Front-End and Back-End, coupled with a modular design, lays the foundation for a scalable, maintainable, and responsive application.

3.3 Back-End RESTful APIs

The back-end RESTful APIs serve as the backbone of the Type Manager Web App, orchestrating the communication between the front end and the data storage. The API,

implemented using Java Spring Boot, follows RESTful principles, offering a set of well-defined endpoints for various operations on DOLAR prototypes. The high-level design of the back-end focuses on providing a robust and efficient mechanism for prototype management, including features such as creating DOLAR prototypes, updating fields, managing inheritance relationships, and more. The Type Manager Web App utilizes the following endpoints:

1. **Retrieve DOLAR prototypes by Namespace**: Retrieves prototype names belonging to a specific namespace, facilitating namespace-based exploration.
2. **Retrieve Prototype Details**: Fetches detailed information about a specific prototype, including inheritance details and field information.
3. **Create New Prototype**: Creates a new prototype with the given name, ensuring uniqueness.
4. **Delete Prototype**: Deletes the identified prototype.
5. **Add Inheritance to Prototype**: Establishes an inheritance relationship for the prototype, reflecting the added inheritance in the response.
6. **Remove Inheritance from Prototype**: Removes an inheritance relationship from the identified prototype.
7. **Create Field**: Adds a new field to the prototype, with parameters like field ID, type, and constraint**.**
8. **Update Field for Prototype**: Modifies an existing field for the prototype, with parameters like field ID, type, and constraint.
9. **Delete Field**: Removes a field from the prototype.
10. **Add Field Group to Prototype**: Adds a new field group to the prototype.
11. **Remove Field Group from Prototype**: Removes a field group from the prototype.
12. **Add Batch Actions**: Executes bulk operations on multiple prototypes simultaneously.

These RESTful endpoints provide a robust API for managing DOLAR prototypes and related entities. The design follows industry-standard conventions, ensuring clarity and ease of use for developers. For a consolidated overview of all endpoints, a summary table is available in the appendix.

3.4 Front end UI Design

The Type Manager Web App is meticulously designed to provide users with a robust and user-friendly interface for creating, modifying, and deleting DOLAR prototypes. The design principles revolve around seamless navigation, a well-organized information hierarchy, and effective user interaction. The following paragraphs detail key aspects of the design.

3.4.1 Homepage and Navigation

The homepage serves as the user's gateway to the Type Manager Web App, featuring a side menu meticulously organized into prototype groups. This design choice

enhances user efficiency, allowing them to quickly locate and select DOLAR prototypes of interest. The side menu's collapsible structure ensures a clean and organized interface, minimizing visual clutter and providing users with a focused view of available prototype categories.

The strategically placed "Create new prototype" button on the homepage is a pivotal element of the navigation and information hierarchy. This button serves as an entry point for users who wish to contribute to the digital library by adding new DOLAR prototypes. By clicking this button, users are seamlessly directed to the "Create New Prototype" page, where they can provide a name for the prototype and initiate the creation process. This separation of the creation process from the prototype exploration ensures a clear distinction in user workflows.

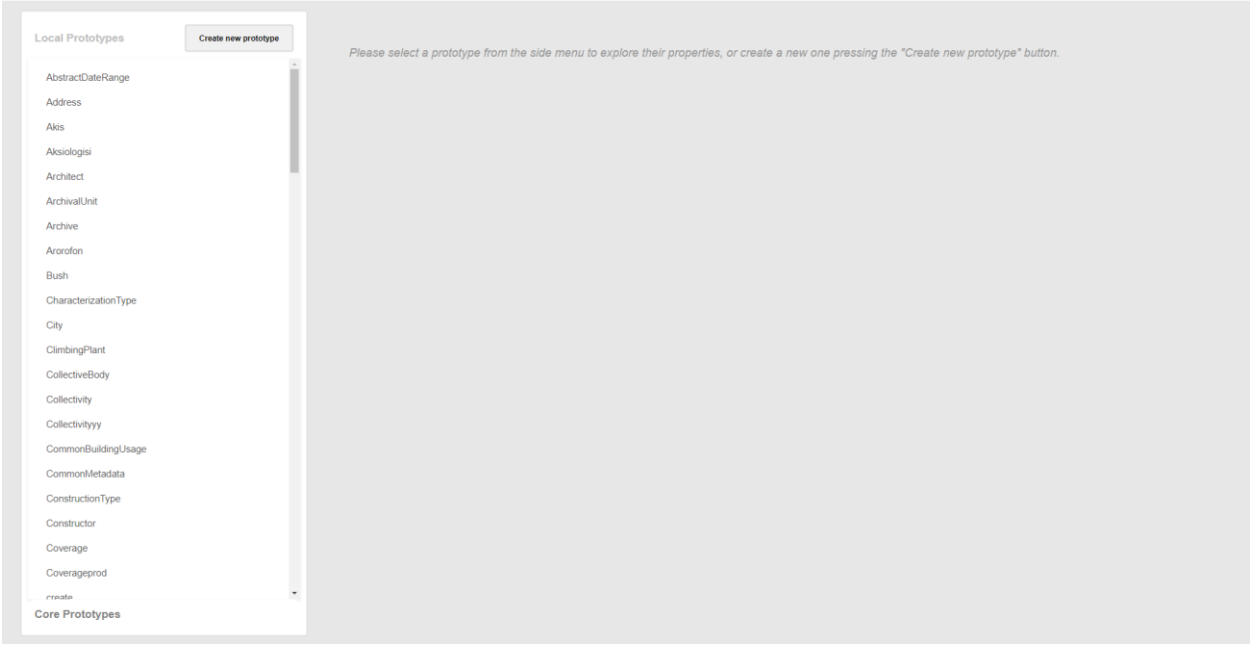

Figure 2: UI Design – Homepage

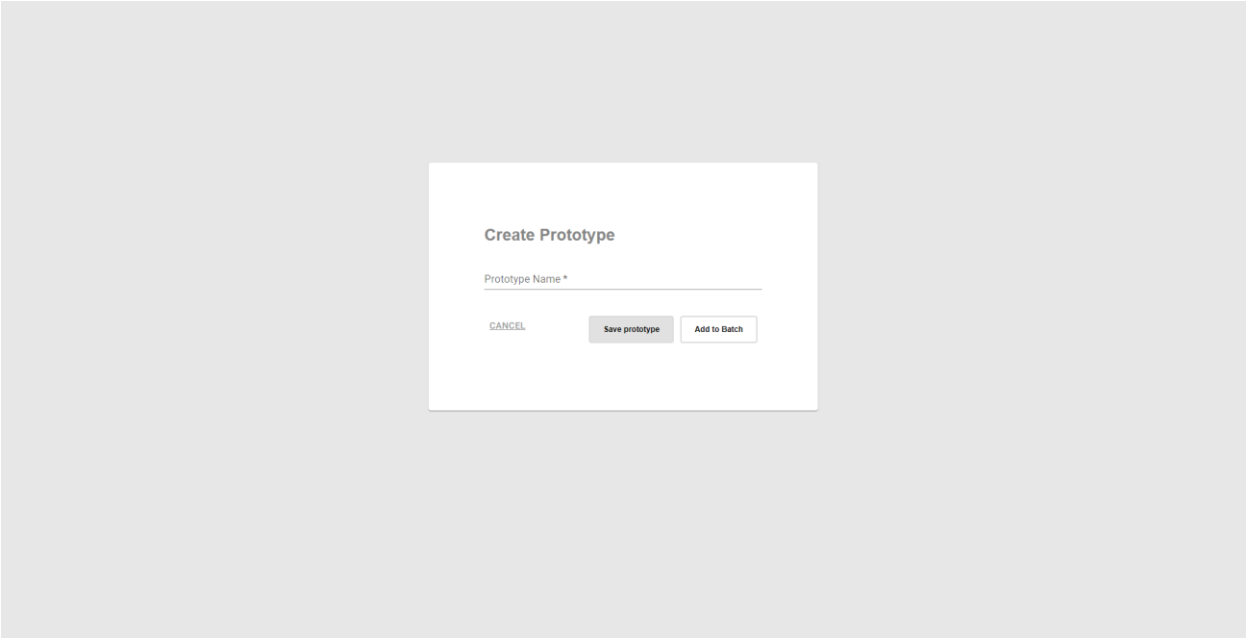

Figure 3: UI Design - Create New Prototype page

3.4.2 Side Navigation Menu

The Side Menu serves as a fundamental element in navigating through the Type Manager Web App. Positioned prominently on the homepage, the Side Menu categorizes DOLAR prototypes into the "Local" and "Core" groups, presenting users with a clear distinction between editable and read-only DOLAR prototypes.

The "Local" DOLAR prototypes represent the user's creations and editable DOLAR prototypes. Users have complete control over these DOLAR prototypes, enabling actions such as editing, adding, and performing various management tasks. The "Local" group forms an integral part of the user's workspace, facilitating a dynamic and interactive environment for prototype development.

In contrast, the "Core" DOLAR prototypes encompass third-party libraries or predefined DOLAR prototypes imported into the system. These DOLAR prototypes are read-only, emphasizing their role as reference or foundational components. Users are restricted from editing or modifying "Core" DOLAR prototypes, ensuring the integrity of these essential components.

The collapsible design of the Side Menu optimizes screen real estate, allowing users to toggle between different prototype groups effortlessly. Clicking on a prototype group triggers an expansion, revealing a detailed list of DOLAR prototypes within that namespace. This dynamic responsiveness enhances user engagement, providing an efficient mechanism for users to explore and select DOLAR prototypes based on their categorization.

Notably, the Side Menu's functionality extends beyond navigation. It persistently accompanies users across pages, ensuring continual access to prototype groups. This consistent presence reinforces a sense of orientation and ease of use, enabling users to seamlessly transition between different sections of the system. The selected prototype is visually emphasized, aiding users in identifying their current location within the prototype hierarchy.

In essence, the Front-End UI Design prioritizes user-friendly interactions and clear categorization, promoting a cohesive and efficient experience in managing both "Local" and "Core" DOLAR prototypes.

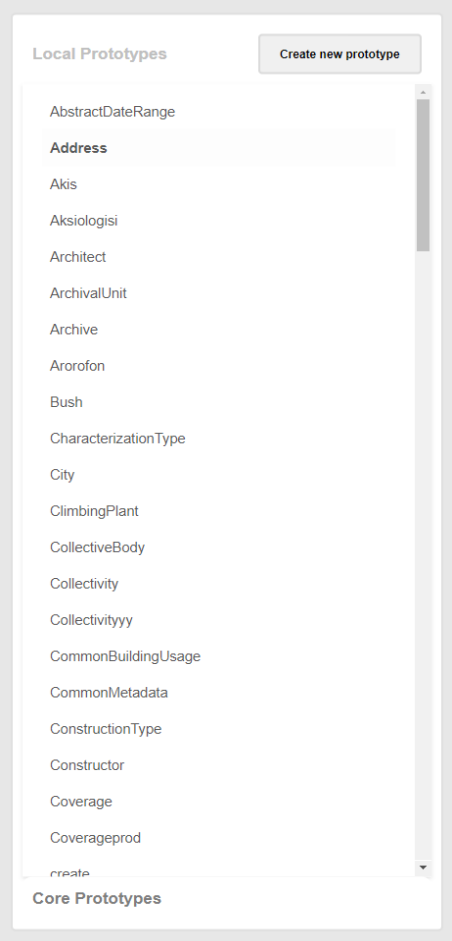

Figure 4: UI Design - Side Menu - Local DOLAR prototypes

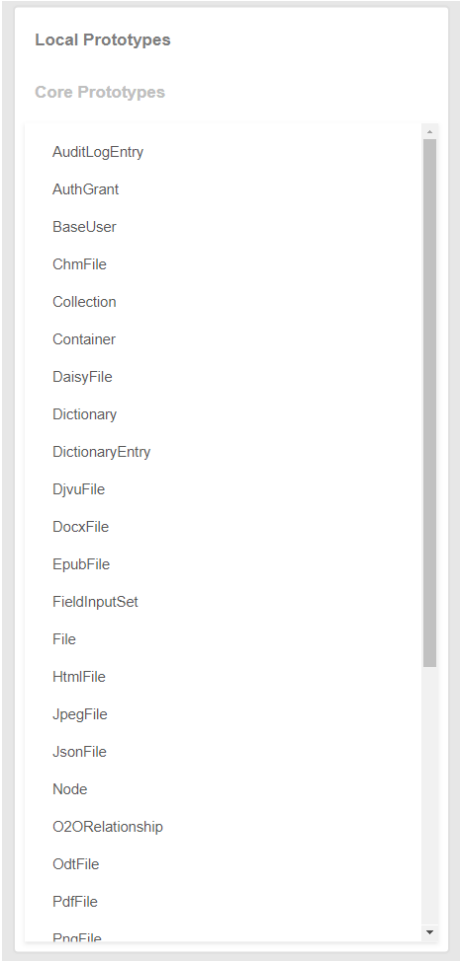

Figure 5: UI Design - Side Menu - Core DOLAR prototypes

3.4.3 Prototype Information Page

Upon selecting a prototype, users are directed to the "Prototype Information" page. The following section delves into the intricacies of the "Prototype Information Page," a central hub within the system where users interact with individual DOLAR prototypes. Organized into distinct sections, this page aims to provide users with an efficient experience. The Inheritance Section (Inherited Prototypes) allows users to navigate and manipulate the prototype's relationships, managing both direct and indirect inherited DOLAR prototypes. Meanwhile, the Fields Section (Structure) meticulously organizes attributes, offering dynamic tools for adding, updating, and removing fields and field groups.

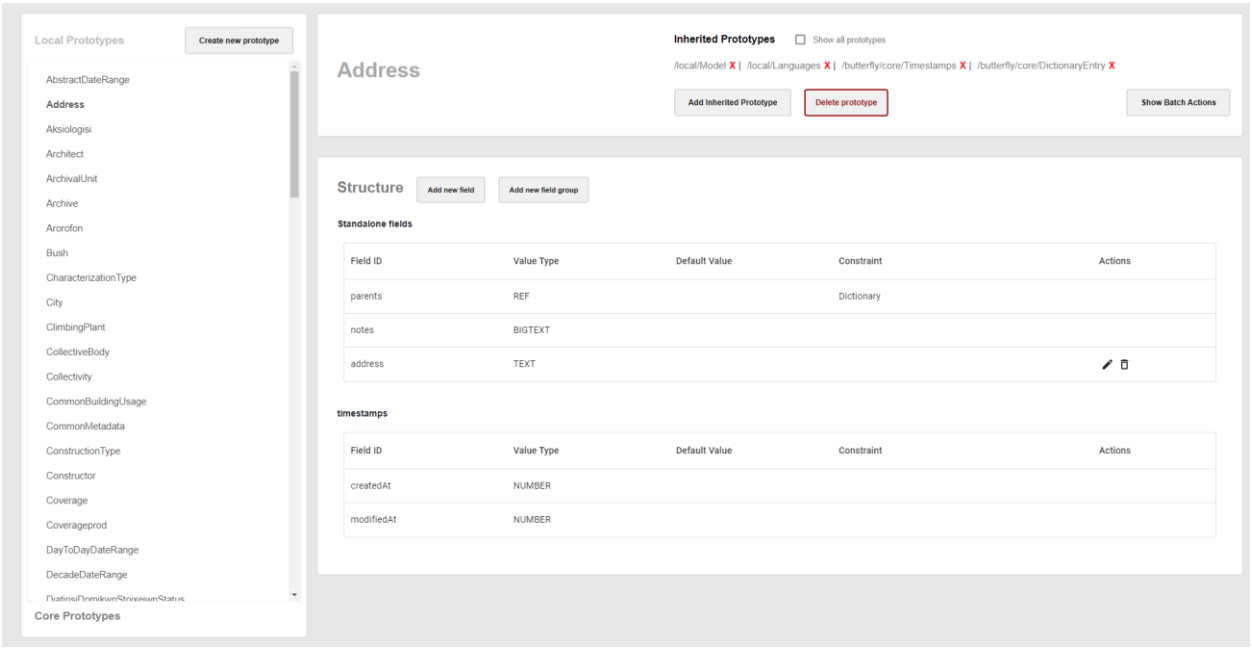

Figure 6: UI Design - Prototype Information page

Inheritance Section

The Inheritance Section provides a comprehensive overview of the prototype's lineage, encompassing both direct and potentially indirect inherited DOLAR prototypes. Users can interact with this section to manage inherited DOLAR prototypes. Removal of directly inherited DOLAR prototypes is facilitated by the "X" icon next to the prototype name. Additionally, a toggle button extends the view to include indirect inherited DOLAR prototypes, enhancing exploration. To incorporate new inherited DOLAR prototypes, users can utilize a dedicated button that initiates the process through a modal, allowing them to select the desired prototype for inheritance.

Fields Section

The Fields Section organizes information about the prototype's attributes, grouping them based on field groups for clarity. Users have the flexibility to efficiently manage fields associated with the prototype. Adding a new field is simplified through a button, triggering a modal for input. The “Edit” option facilitates updates to existing fields, while the “Delete” option removes fields linked to the prototype. Fields are presented in sub-tables categorized by field groups, fostering a structured representation of the prototype's attributes. The modal for updating a field is shown in the below image, as well as the confirmation dialog for the deletion of a field. Users can dynamically add new field groups and remove existing ones, promoting adaptability. A specific sub-table labeled "Standalone fields" ensures the orderly presentation of fields that do not belong to any predefined group, maintaining a visually tidy interface.

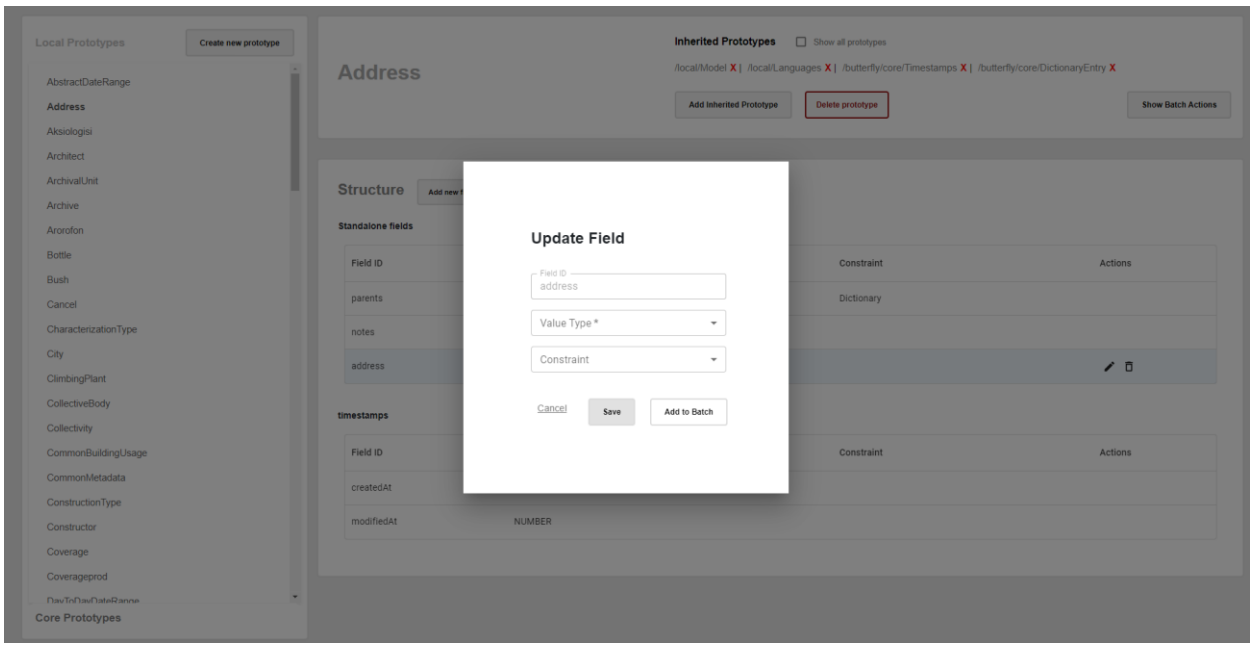

Figure 7: UI Design - Update Field modal

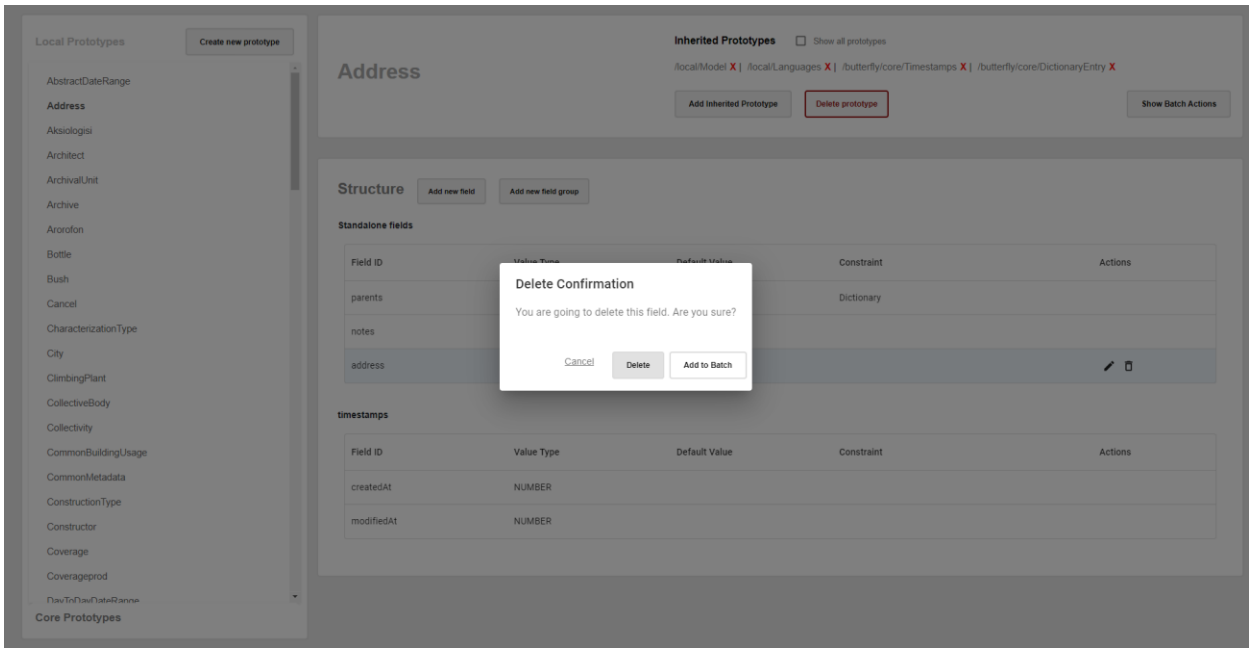

Figure 8: UI Design - Delete Field confirmation dialog

3.4.4 Modal Components

In the Type Manager Web App, Modal components play a pivotal role in facilitating user interactions, particularly during user actions. The modal, leveraging the Material-UI (MUI) library, is strategically employed to provide users with a streamlined experience. The following outlines the modal workflow within the system:

When a user initiates specific actions, such as adding a new field, the Modal comes into play, ensuring a focused and distraction-free interaction. Clicking on the corresponding button triggers the display of the Modal, prompting users to input the necessary information for the intended action.

Within the modal, users are presented with fields to complete, capturing essential data for the action. This design choice ensures a seamless and guided experience, minimizing the potential for errors or omissions.

Within each action available, such as adding or updating fields, two buttons coexist in the modal– the "Save" button and the "Add to Batch" button. This thoughtful design allows users to choose between immediate execution and deferred action storage. When users press the "Save" button, the action is sent to the server and executed promptly. On the other hand, if users opt for the "Add to Batch" button, the action is saved locally in the browser's storage.

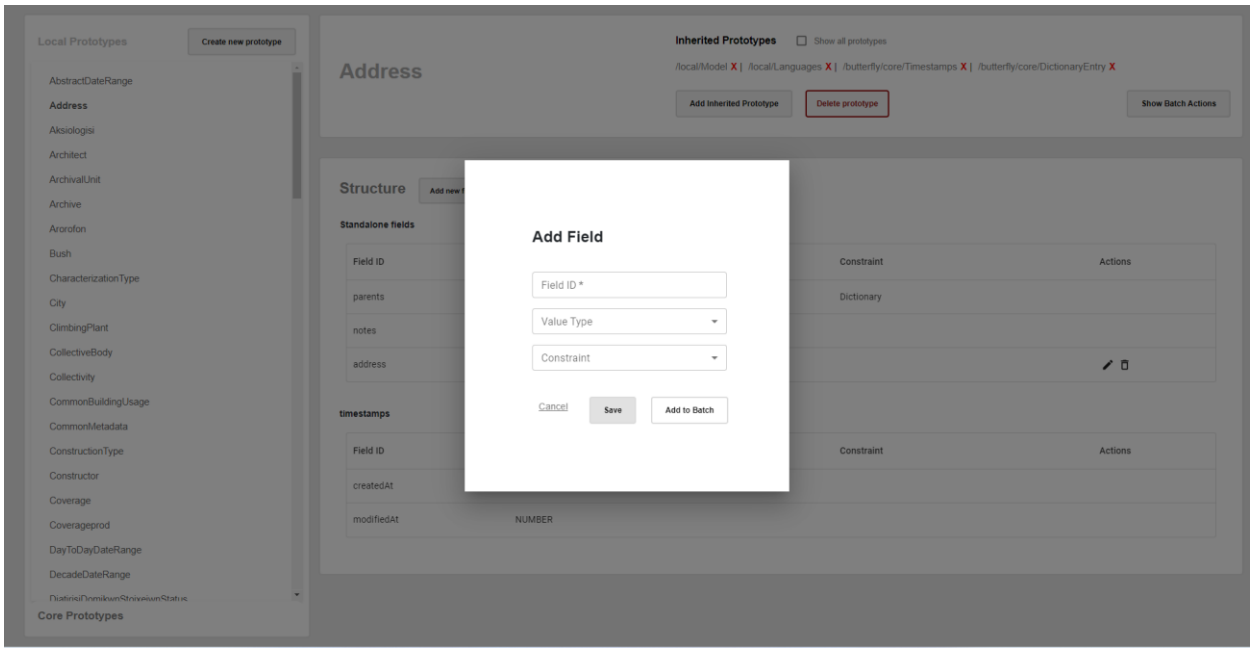

Figure 9: UI Design - Add New Field modal

Following action execution, whether immediate or deferred, the Modal automatically closes, contributing to a clean and uncluttered interface. This closure is complemented by immediate feedback through a visually clear indication of the action's success or failure.

Users have the option to cancel the operation by pressing the cancel button within the modal. Cancelling results in the closure of the modal without executing the intended action, offering users flexibility in their interactions.

As part of the user feedback system, an alert notification is displayed upon action completion, providing real-time information about the outcome. This notification mechanism ensures users are promptly informed about the success or failure of their actions.

This thoughtful design of modal interactions aligns with the broader user experience principles of the Type Manager Web App, emphasizing clarity, efficiency, and user-friendly interactions.

To access and manage the actions saved at the batch, users can utilize the "Show Batch Actions" button on the "Prototype" page. This action triggers the display of a modal, known as the "Batch Actions Modal," providing an organized overview of all saved actions.

3.4.4.1 Batch Actions Modal

The "Batch Actions Modal" is designed to offer users a detailed and structured view of the deferred actions. Upon pressing the "Show Batch Actions" button, users are presented with a modal that categorizes actions by prototype. For each DOLAR prototype, there is a list of saved actions, including the type of action (e.g., "Update Field") and specifics of the change (e.g., field ID, type, constraint).

This modal empowers users to selectively manage deferred actions. They can remove individual actions by interacting with "X" buttons next to each action entry. Additionally, users have the option to clear the entire batch with a single click, providing a streamlined way to reset or revise their deferred actions. Once users are satisfied with the batched actions, they can press the "Save Actions" button within the modal. This action initiates the process of sending a JSON string containing all the batched actions to the server for execution. Simultaneously, the modal closes, ensuring a clean and uncluttered interface.

The "Batch Actions Modal" introduces a user-friendly means of managing deferred actions, aligning seamlessly with the broader user experience principles of the Type Manager Web App.

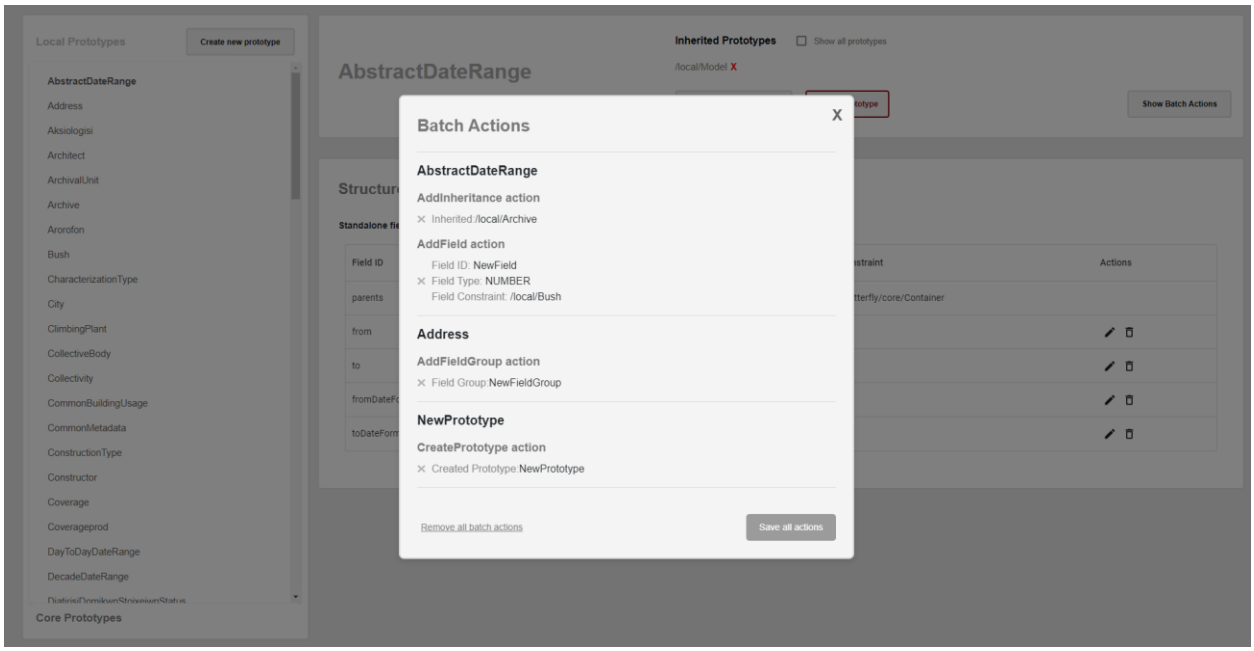

Figure 10: Batch Actions Modal

3.4.5 Notifications

The Notifications feature is a crucial component embedded in the Type Manager Web App, designed to provide real-time and informative feedback to users. This section plays a pivotal role in enhancing user awareness and transparency, ensuring that users are promptly notified about system actions and changes, ultimately contributing to a positive user experience.

When users perform actions, such as adding or removing DOLAR prototypes, updating fields, or making other modifications within the system, the Notifications system comes into play. These notifications serve as immediate feedback, informing users about the outcome of their actions. This initiative-taking communication is vital in keeping users

well-informed and engaged with the ongoing processes within the Type Manager Web App.

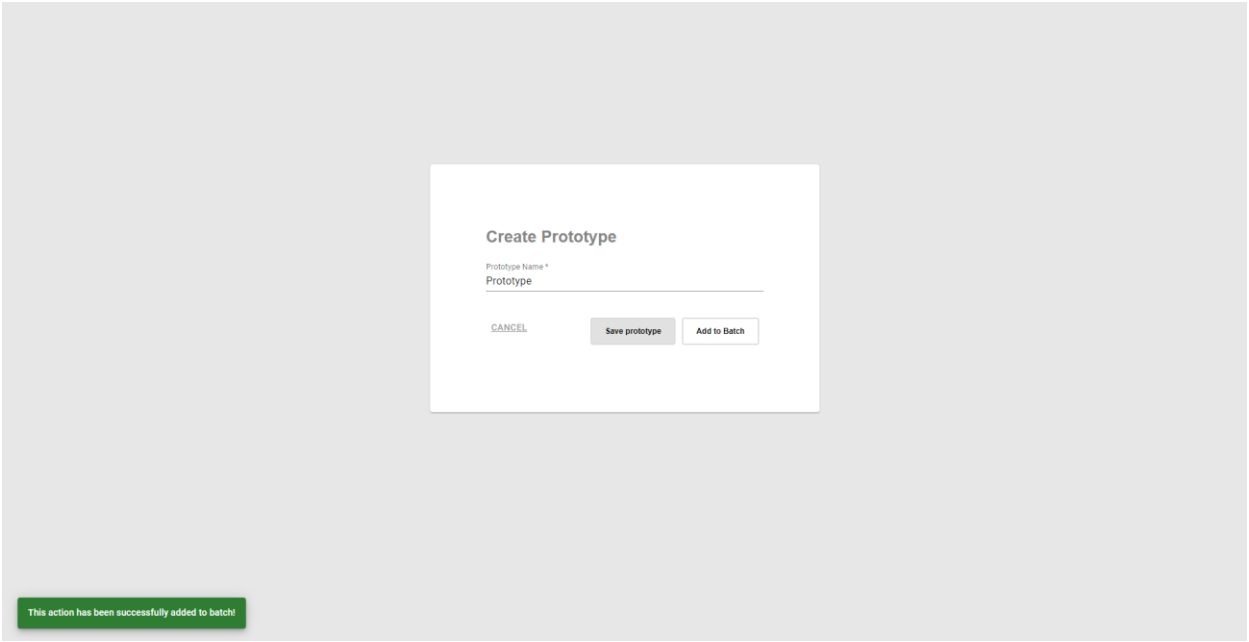

Figure 11: UI Design – Notification for successfully adding a prototype

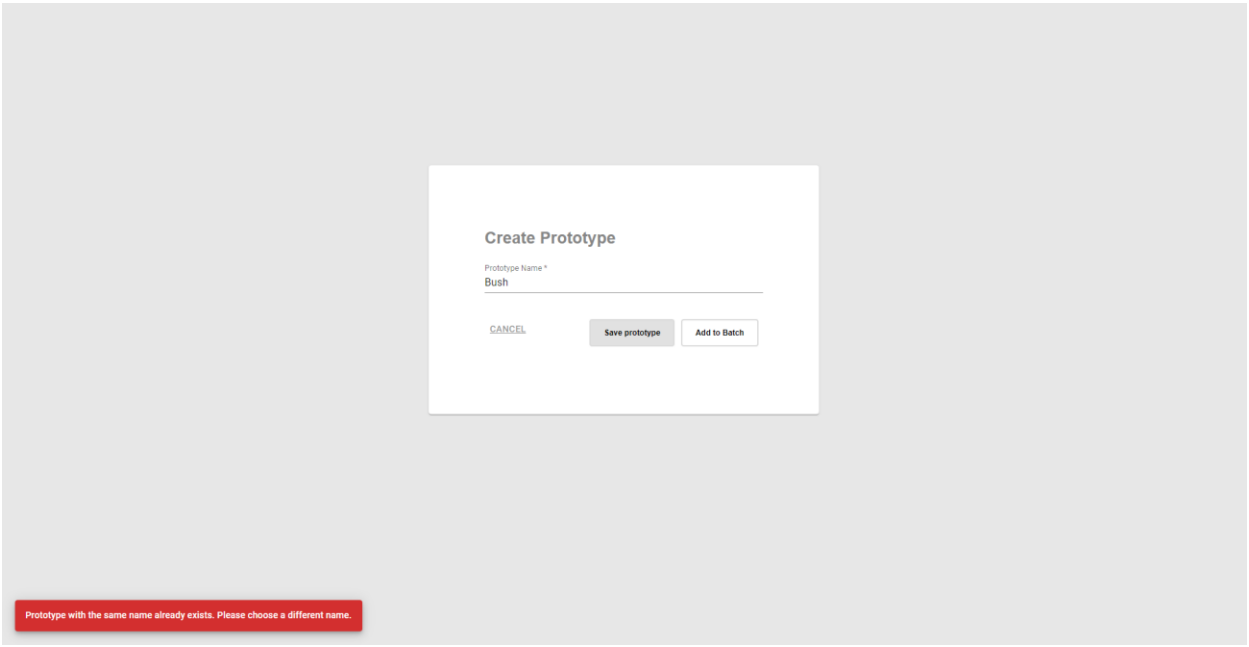

Figure 12: UI Design - Notification for unsuccessfully adding a prototype

The design of the notifications is carefully crafted to balance visibility and non-intrusiveness. Each notification is presented in a clear and unobtrusive manner, avoiding disruption to the user's workflow while ensuring that essential information is readily accessible. The language used in these notifications is concise and informative, conveying the essential details of the action performed.

Moreover, the Notifications system supports accessibility by incorporating features such as color-coding to signify the nature of the notification. This approach caters to users

with diverse needs, ensuring that the system communicates effectively regardless of individual user preferences or potential limitations.

In summary, the design prioritizes a user-centric approach, ensuring an intuitive and efficient user experience. The detailed consideration of navigation, information presentation, interaction, and feedback mechanisms establish a foundation for a robust Type Manager Web App. These design choices set the stage for a smooth transition to the "Implementation" section, where the theoretical concepts will be translated into practical, functional elements of the system.

3.5 Implementation

The Type Manager Web App represents a culmination of cutting-edge technologies strategically integrated to deliver a robust and user-centric solution. At its core, the system relies on the powerful Java Spring Boot framework for the back-end, ensuring a solid foundation for scalable and maintainable operations. Augmenting this, the front-end is meticulously crafted using React, a declarative JavaScript library celebrated for its efficiency in constructing dynamic and interactive user interfaces. The integration of Electron extends React's capabilities, allowing the system to effortlessly function as a standalone desktop application. To ensure a contemporary and cohesive user interface, the Material-UI (MUI) styling library is deployed, aligning with Google's Material Design principles for an aesthetically pleasing and user-friendly experience.

3.5.1 Front-End Implementation

The front-end implementation of the Type Manager Web App relies on the powerful combination of React with Electron, providing users with an immersive and responsive user interface. The utilization of React, a JavaScript library for building user interfaces, offers a component-based approach that fosters modular and reusable code. This not only enhances code organization but also streamlines development and maintenance.

3.5.1.1 React Front-End Implementation

The frontend of the Type Manager Web App is meticulously crafted using React, a declarative JavaScript library renowned for its efficiency in constructing dynamic and interactive user interfaces. React's component-based architecture forms the backbone of the system, promoting code reusability and maintainability. Each distinct element, from the Side Menu to modals and alerts, is encapsulated within separate React components, facilitating a modular and organized codebase. This choice not only enhances development speed but also paves the way for a more scalable and extensible system. React's virtual DOM ensures optimal rendering performance, contributing to a seamless and responsive user experience. [9]

React's popularity and widespread adoption in the industry demonstrate its effectiveness as a front-end development tool [10]. Its robust ecosystem provides a vast array of community-driven libraries and packages that extend its capabilities and simplify common tasks [11]. Additionally, React's strong community support offers extensive documentation, online forums, and tutorials, making it easier for developers to learn and troubleshoot issues [12].

3.5.1.2 Electron Integration

The integration of Electron amplifies the capabilities of the React frontend, enabling the Type Manager Web App to function effortlessly as a standalone desktop application. Electron leverages web technologies to package React applications into cross-platform desktop apps, extending the reach and accessibility of the system. This integration ensures a consistent user experience across various operating systems, maintaining the responsiveness and interactivity characteristic of web applications. By leveraging Electron, the Type Manager Web App transcends traditional web application boundaries, providing users with the flexibility of a desktop application. [13]

Electron has gained significant popularity as a framework for building desktop applications using web technologies [14]. Its rich feature set and cross-platform compatibility make it a preferred choice for developers seeking to create desktop applications with web technologies. The Electron community actively contributes to the framework's growth, offering a wide range of plugins and extensions that enhance its functionality and enable integration with native operating system capabilities [15].

3.5.1.3 Material-UI Styling Library

The user interface of the Type Manager Web App is enhanced by the integration of the Material-UI (MUI) styling library. Material-UI is a popular React UI framework that implements Google's Material Design principles. By utilizing Material-UI components, the system achieves a modern and cohesive look and feel. The library offers pre-designed, customizable components, ensuring consistency in design elements and enhancing the overall user experience. This strategic use of Material-UI aligns the system with contemporary design standards and contributes to an aesthetically pleasing and user-friendly interface. The system extensively utilizes Material-UI components, including buttons, modals, forms, and navigation elements, to maintain a cohesive design language throughout the application. [16]

Material-UI has gained traction in the React community due to its comprehensive set of components and ease of integration. Its flexible theming capabilities allow developers to customize the visual aspects of the application to align with branding guidelines or specific design requirements [17]. Material-UI's active development and frequent updates ensure ongoing support and the incorporation of new features. [18]

In conclusion, the front-end implementation revolves around React and Electron, combining the modular and component-driven nature of React with the desktop application capabilities of Electron. This amalgamation results in a responsive and efficient user interface, offering a seamless experience across different platforms. The emphasis on components ensures a well-organized and maintainable codebase, facilitating a smooth development process and enhancing the overall user interaction with the Type Manager Web App.

3.5.2 Back-End Implementation

The back end of the Type Manager Web App is powered by Java Spring Boot, a widely adopted and highly regarded framework for building robust and scalable Java-based applications.

3.5.2.1 Spring Back-End Implementation

Spring Boot simplifies the development process by providing conventions and defaults, allowing developers to focus on business logic rather than boilerplate code. Leveraging the Spring Boot framework enhances the system's maintainability and reliability, with features such as dependency injection, aspect-oriented programming, and a vast ecosystem of extensions. This choice aligns with industry best practices, ensuring a solid foundation for the Type Manager Web App's backend operations. [19]

The system architecture comprises Configuration, Controller, Service, PrototypeEncoder, and Application files.

The Configuration file (ApiConfiguration) plays a crucial role in configuring the Type Manager Web App, specifically setting up the virtual space for DOLAR prototypes. The primary purpose of the file is to provide configuration details for the system's virtual space setup. In this context, a virtual space represents the location where DOLAR prototypes are stored. This externalization of configuration details enhances flexibility, allowing for easy modifications and adaptability to different environments or use cases.

The Controller class (ApiController) acts as the Controller, exposing various endpoints for communication with the front end. These endpoints include functionalities for retrieving DOLAR prototypes, creating, updating, and deleting DOLAR prototypes, managing fields, handling inheritance relationships, and working with field groups. For instance, the getPrototypes, createType, deleteType, createField, addInheritance, and other methods define the API endpoints responsible for these operations.

The Service class (ApiService) encapsulates the logic for prototype management. It coordinates the actions requested by the Controller, interacting with the data repository, which in this case is a folder in the code directory where DOLAR prototypes are stored. The Service class performs actions such as loading DOLAR prototypes, creating DOLAR prototypes, updating fields, and managing inheritance relationships.

The PrototypeEncoder class (ApiPrototypeEncoder) is instrumental in ensuring the correct encoding of DOLAR prototypes. When data needs to be exchanged between the back end and the front end, the encoding process is vital to represent the DOLAR prototypes in a format that can be efficiently transmitted and understood by both sides. More specifically, the PrototypeEncoder class is responsible for encoding the details of DOLAR prototypes, translating them into a format suitable for communication. This class is tightly coupled with the Service component, where it collaborates to correctly encode DOLAR prototypes based on specific requirements.

The main class (ApiApplication) is a fundamental component in the Spring Boot application, serving as the entry point for the Type Manager Web App. The primary purpose of the ApiApplication class is to bootstrap the Spring Boot application. It initializes the Spring context, configures the application, and starts the embedded web server. It launches the Type Manager Web App. By encapsulating the startup logic within this class, developers benefit from the auto-configuration capabilities of Spring Boot and a well-organized entry point for the application.

In summary, the integration of React with Electron for the front end and Java Spring Boot for the back-end results in a comprehensive and efficient Type Manager Web App. The implementation adheres to a modular structure, where components are encapsulated for maintainability. This design, coupled with clear communication between the front end and back end, ensures a well-organized and user-friendly system that successfully realizes the outlined design concepts.

3.5.2.2 Endpoints

The Type Manager Web App exposes a set of RESTful endpoints through the ApiController class in the Spring Boot application, providing a versatile API for the management of DOLAR prototypes, fields, inheritance relationships, and field groups. The rest of this subsection delves into the essential endpoints and their respective functionalities.

Please note that edit operations (POST, PUT, DELETE) mentioned in the following endpoints apply exclusively to "Local" DOLAR prototypes, representing user-created DOLAR prototypes. On the other hand, GET and read-only operations are designed to encompass both "Local" and "Core" DOLAR prototypes.

1.Retrieve DOLAR prototypes by Namespace

This endpoint retrieves the IDs of the DOLAR prototypes belonging to a specific namespace identified by {id}. The response includes a list with the IDs of the DOLAR prototypes, facilitating namespace-based exploration. The data of this endpoint is used to populate the namespaces of the side menu, the category of the “Local DOLAR prototypes” and the “Core DOLAR prototypes”.

Endpoint	Method	Path Variable	Response
/api/type/category/{id}	GET	namespace ID	list of prototype IDs

2.Retrieve Prototype Details

This endpoint fetches detailed information about a specific prototype identified by {id}. The response includes the prototype's ID, inheritance details, and field information, providing a comprehensive snapshot. The data of this endpoint is used to populate the page with the prototype information.

Regarding the inheritance details, the endpoint returns two different lists of DOLAR prototypes: the directed inherited DOLAR prototypes which are the DOLAR prototypes that are directly inherited from the original prototype, and the transitively inherited DOLAR prototypes, which are all the DOLAR prototypes that are inherited from the prototype. The transitively inherited DOLAR prototypes are displayed by toggling the corresponding button in the UI, while the user can remove only the directed inherited DOLAR prototypes from the prototype page.

Regarding the field details, the endpoint returns a list with all the fields that belong to the prototype. Additionally, the response contains a “definedInThis” field which indicates which fields belong to the specific prototype. The user can edit and delete the fields that are “defined in this”, because these fields are actually defined in the prototype, and they are not inherited.

The endpoint also returns a list of field groups, which contains all the field groups that are defined in this prototype. For every field contained in each field group, the basic field information (field ID, type, default value, constraint, defined in this) is also being returned.

Endpoint	Method	Path Variable	Response

/api/type/{id}	GET	prototype ID	- prototype ID
			- inheritance details
			  o directed inherited prototypes
			  o transitively inherited prototypes
			- field details
			  o field ID
			  o type
			  o default value
			  o constraint
			  o defined in this
			- field group details
			  o field details

3.Create New Prototype

This endpoint is used to create a new prototype with the given name {id}. The response contains the details of the newly created prototype. The user has to provide a unique name for the prototype, because the system checks if an existing prototype has the provided name.

Endpoint	Method	Path Variable	Response
/api/type/{id}	POST	prototype ID	prototype details

4.Delete Prototype

This endpoint is used to delete the prototype identified by {id}. This action requires confirmation in the front-end to avoid accidental deletions.

Endpoint	Method	Path Variable	Response
/api/type/{id}	DELETE	prototype ID	-

5.Add Inheritance to Prototype

This endpoint establishes an inheritance relationship for the prototype identified by {id}. The request parameter includes the inherited prototype. The response provides updated details of the prototype, reflecting the added inheritance.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/inheritance	POST	prototype ID	inherited prototype ID	prototype details

6.Remove Inheritance from Prototype

This endpoint removes one DOLAR prototype from the prototype identified by {id}. The request parameter includes the DOLAR prototype to be removed. The response provides updated details of the prototype, reflecting the removed inheritance relationships.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/inheritance	DELETE	prototype ID	inherited prototype ID	prototype details

7.Add Field to Prototype

This endpoint adds a new field to the prototype identified by {id}. The request includes parameters like field ID, type, and constraint. The response provides updated details of the prototype, reflecting the newly added field. All the created fields are included in the “undefined” group category by default.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/field	POST	prototype ID	field ID type constraint	prototype details

8.Update Field in Prototype

This endpoint modifies an existing field for the prototype identified by {id}. The request includes parameters like field ID, type, and constraint. The response provides updated details of the prototype, reflecting the modified field. It is worth mentioning that the user can update only the fields that belong to the prototype and are directly “defined in this”. Only these specific fields have the “Edit” option next to them, helping the user to understand which are editable and which are read-only.

Note: In the UI the user can modify the type and the constraint of the field, as in the “Edit Field” modal the field ID is read-only and is only sent in the request to define which is the updating field.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/field	PUT	prototype ID	field ID type constraint	prototype details

9.Delete Field from Prototype

This endpoint removes a field from the DOLAR prototype identified by {id}. The request includes the field ID. The response provides updated details of the prototype, reflecting the removed field. It is worth mentioning that the user can delete only the fields that

belong to the prototype and are directly “defined in this”. Only these specific fields have the “Delete” option next to them, helping the user to understand which are deletable.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/field	DELETE	prototype ID	field ID	prototype details

10. Add Field Group to Prototype

This endpoint adds a new field group to the prototype identified by {id}. The request includes the group ID. The response provides updated details of the prototype, reflecting the added field group.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/group	POST	prototype ID	group ID	prototype details

11. Delete Field Group from Prototype

This endpoint removes a field group from the prototype identified by {id}. The request includes the group ID. The response provides updated details of the prototype, reflecting the removed field group.

Endpoint	Method	Path Variable	Request Parameter	Response
/api/type/{id}/group	DELETE	prototype ID	group ID	prototype details

12. Add Batch Actions

This endpoint executes a batch of actions specified as a string parameter and converts it into a JSON file for processing. The request includes the batch actions as a string parameter. The response provides feedback on the execution status.

Endpoint	Method	Request Parameter	Response
/api/type/batch	POST	jsonString	feedback on the execution status

These RESTful endpoints provide a comprehensive API for managing DOLAR prototypes and related entities in the Type Manager Web App. They cover essential actions such as creating, updating, and deleting DOLAR prototypes, fields, inheritance relationships, and field groups. The design follows REST conventions, making it intuitive for developers to interact with the system programmatically.

Furthermore, to provide a comprehensive overview of the entire API structure, a summary table consolidating all endpoints can be found in the appendix.

3.5.2.3 Payload

3.5.2.3.1 Prototype Payload

The response payload returned for almost every Type Manager Web App request provides a comprehensive overview of the prototype's structure, inheritance details, and associated fields. The payload includes the following key attributes:

Prototype Information:

- **ID:** A unique identifier assigned to the prototype, allowing for unambiguous reference.
- **inherits:** Directly inherited DOLAR prototypes, indicating the immediate parent DOLAR prototypes.
- **inheritsTransitively:** All DOLAR prototypes inherited, both directly and indirectly, creating a full inheritance hierarchy.

Field Group Categorization:

- Fields are organized and categorized based on their associated field group IDs, providing a structured view of the prototype's fields.
- Within each field group, individual fields are listed, each with the aforementioned attributes.

Field Information:

- **ID**: Unique identifier for each field within the prototype, facilitating precise identification.
- **type:** Specifies the data type of the field, offering insights into the nature of the stored information.
- **defaultValue:** Indicates the default value assigned to the field, aiding in understanding the expected initial state.
- **constraint:** Identifies other DOLAR prototypes that serve as constraints for the current field, establishing relationships.
- **isDefinedInThis:** A binary indicator, conveying whether the field is defined directly within the current prototype.

This payload structure ensures clarity and completeness in representing the prototype's composition, inheritance relationships, and the attributes associated with each field. It allows users to quickly comprehend the prototype's structure, aiding in efficient navigation and informed decision-making during system interactions. The categorization of fields into groups adds an additional layer of organization, enhancing the payload's readability and user comprehension.

Consider the example of a DOLAR prototype named "LocationDetails":

```
{
    "id": "LocationDetails",
    "inherits": [
        "/local/Model"
    ],
```

```
    "inheritsTransitively": [
        "/local/Model",
        "/butterfly/core/Node"
    ],
    "fields": [
        {
            "id": "parents",
            "type": "REF",
            "array": true,
            "constraint": "/butterfly/core/Container",
            "defaultValue": "",
            "isDefinedInThis": false
        },
        {
            "id": "centerGeoPoint",
            "type": "EMBED",
            "array": false,
            "constraint": "GeoPoint",
            "defaultValue": "",
            "isDefinedInThis": true
        }
    ],
    "fieldGroups": {
        "files": [
            {
                "id": "geoJSON",
                "type": "REF",
                "array": false,
                "constraint": "/butterfly/core/JsonFile",
                "defaultValue": "",
                "isDefinedInThis": true
            }
        ]
    }
}
```

In this example, "LocationDetails" inherits from "/local/Model" and transitively inherits from "/local/Model" and "/butterfly/core/Node." The prototype includes fields such as "parents" and "centerGeoPoint," each with specific attributes. Additionally, the field "geoJSON" is appropriately placed within the "files" field group, exemplifying the organization of fields based on their associated field group IDs.

The isDefinedInThis attribute serves as a binary indicator, providing valuable information about the origin of each field. When isDefinedInThis is set to true, as in the case of "centerGeoPoint" in the example, it indicates that the field is explicitly defined within the current prototype ("LocationDetails"). On the other hand, when isDefinedInThis is false, as in the case of "parents," it signifies that the field is inherited from one of the parent DOLAR prototypes, in this instance, from "/local/Model."

This attribute is instrumental in understanding the source of each field, whether it is locally defined in the current prototype or inherited from parent DOLAR prototypes.

3.5.2.3.2 Batch Actions Payload

In addition to the standard payload, when users press the "Save Actions" button within the "Batch Actions Modal," a JSON file is generated and sent to the server. This JSON, encapsulated under the "actions" key, contains an array of objects, each representing a deferred action:

```
{
    "actions": [
      {
        "method": "CreatePrototype",
        "prototypeName": "Bush",
        "prototypePath": "/local/Bush"
      },
      {
        "method": "DeletePrototype",
        "prototypeName": "Address",
        "prototypePath": "/local/Address"
      },
      {
        "method": "AddField",
        "prototypeName": "Monument",
        "prototypePath": "/local/Monument",
        "id": "region",
        "type": "REF",
        "constraint": "/local/User"
      },
      {
        "method": "UpdateField",
        "prototypeName": "Address",
        "prototypePath": "/local/Address",
        "id": "region",
        "type": "REF",
        "constraint": ""
      }
    ]
}
      {
        "method": "RemoveField",
        "prototypeName": "Bush",
        "prototypePath": "/local/Bush",
        "id": "location"
       },
      {
        "method": "AddFieldGroup",
        "prototypeName": "Architect",
        "prototypePath": "/local/Architect",
        "id": "Locations"
      },
      {
        "method": "RemoveFieldGroup",
```

```
      "prototypeName": "Bush",
      "prototypePath": "/local/Bush",
      "id": "Timestamps"
    },
    {
      "method": "AddInheritance",
      "prototypeName": "Address",
      "prototypePath": "/local/Address",
      "inheritedPrototypes": ["/local/Region"]
    },
    {
      "method": "RemoveInheritance",
      "prototypeName": "City",
      "prototypePath": "/local/City",
      "removePrototypes": ["/local/Languages", "/local/Model"]
    }
  ]
}
```

This JSON includes detailed information about each deferred action, such as the action type ("CreatePrototype," "DeletePrototype," "AddField," etc.), the relevant prototype name, the path of the DOLAR prototype and additional parameters specific to each action type. More specifically, let us break down the structure of the Batch Actions JSON:

- Each object within the "actions" array represents a deferred action that the user has chosen to include in the batch.
- The "action" attribute specifies the type of action to be performed, such as "CreatePrototype," "DeletePrototype," "AddField," and others.
- For actions like "AddField," "UpdateField," "RemoveField," and "AddFieldGroup," specific details about the field or field group are included, such as "id," "type," and "constraint."
- The "prototypeName" attribute identifies the ID of the DOLAR prototype to which the action is applied.
- The "prototypePath" attribute identifies the path of the DOLAR prototype to which the action is applied.
- For actions involving inheritance ("AddInheritance" and "RemoveInheritance"), the associated DOLAR prototypes are listed under "inheritedPrototypes" or "removePrototypes" respectively.

This structured JSON representation ensures that the server can accurately interpret and execute the batched actions. The server processes each object within the "actions" array sequentially, applying the specified actions to the corresponding DOLAR prototypes. This approach ensures the preservation of data integrity and consistency within the Type Manager Web App, even when users choose to defer and batch their actions for execution at a later time. The detailed information in the JSON file enables the server to precisely execute each action, reflecting the user's intentions within the system.

3.5.3 Asynchronous Communication Logic

The Type Manager Web App employs a robust asynchronous communication logic to establish seamless data exchange between the front-end React application and the back-end Spring Boot server. The underlying functions, including getData, postData, deleteData, and putData, play a pivotal role in enabling efficient asynchronous HTTP requests through the utilization of the fetch API. This section elucidates the intricacies of this communication logic, shedding light on its structure and key components.

Common Structure

The core asynchronous functions share a common structure, enhancing consistency and simplifying maintenance. Each function accepts a URL parameter denoting the API endpoint and an optional formData parameter containing data destined for the server. Leveraging the fetch API, these functions ensure an agile and responsive communication flow between the front end and back end.

Request Configuration

A fundamental aspect of these functions is the configuration of the HTTP request. Parameters such as method, mode, cache, credentials, headers, and body are meticulously set to tailor the request to the specific needs of the operation. The use of the await keyword ensures that the asynchronous fetch operation completes before proceeding, maintaining a synchronous appearance in the code.

Response Handling

Upon receiving a response from the server, the system employs the response.json() method to parse the response as JSON. This asynchronous operation yields the actual data transmitted by the server, a critical step for effective handling of server responses and extraction of pertinent information.

Modularity and Reusability

Encapsulating this asynchronous logic into utility functions enhances modularity and promotes reusability. The approach ensures that the code remains organized, making it easier to manage and extend as the application evolves. As a result, the Type Manager Web App achieves a high degree of maintainability and adaptability in handling diverse data interactions with the back end.

```
async function postData(url = "", formData = new FormData()) {
  const response = await fetch(url, {
    method: "POST",
    mode: "cors",
    cache: "no-cache",
    credentials: "same-origin",
    headers: {},
    redirect: "follow",
    referrerPolicy: "no-referrer",
    body: formData,
  });
```

```
  return response.json();
}
```

In essence, the well-structured asynchronous communication logic is a cornerstone of the React front-end implementation, contributing to a responsive and user-friendly experience during data exchanges with the Spring Boot back end.

3.5.4 End-to-End Example: Adding a Field

To illustrate the seamless flow of the "Add Field" functionality in the Type Manager Web App, an end-to-end example is presented. This process involves the interaction between the front-end React application, the back-end Spring Boot server, and the underlying service responsible for managing DOLAR prototypes.

3.5.4.1 Front-End Implementation

On the front end, the React application manages user interactions and initiates the process of adding a field. The user triggers the addition of a new field by clicking the "Add new field" button.

```
<button className="add-field-button" onClick={handleAddField}>
    Add new field
</button>
```

The **handleAddField** function resets the selected field information and opens the modal for adding a new field, as shown in the code below.

```
const handleAddField = () => {
    setSelectedFieldId(null);
    setSelectedFieldType(null);
    setSelectedFieldConstraint(null);
    setShowFieldModal(true);
};
```

The modal, displayed conditionally based on the **showFieldModal** state, enables users to input the details of the new field. The modal includes fields for the "Field ID," "Value Type," and "Constraint," along with buttons to save, cancel, or add the field to the batch.

```
<Modal open={showFieldModal} onClose={handleCancelField} aria-
labelledby="modal-modal-title"
       aria-describedby="modal-modal-description">
    <Box sx={style}>
        <div className="add-field-form">
            <h3 className="add-new-field">Add Field</h3>

            <FormControl sx={{width: '300px'}}>

                <Stack spacing={2} sx={{margin: '10px'}}>

               // Input fields for Field ID, Value Type and Constraint
```

```
                </Stack>
            </FormControl>

            <div className="field-button-container">

                <button className="cancel-field-button"
onClick={handleCancelField}>
                    Cancel
                </button>

                <button className="save-action-button" onClick={saveField}>
                    Save
                </button>

                <button className="add-action-batch-button"
onClick={addToBatch}>
                    Add to Batch
                </button>

            </div>
        </div>
    </Box>
</Modal>
```

3.5.4.1.1 Adding a Field

The **saveField** function is triggered when the user confirms the field addition and works as a wrapper around the **handleSaveField** function, providing additional error handling and cleanup after attempting to save a new field.

```
const saveField = async () => {
    try {
        await handleSaveField(selectedFieldId, selectedFieldType,
selectedFieldConstraint);
        handleCancelField();
    } catch (error) {
        console.error('Error saving field:', error);
    }
};
```

Then, the **handleSaveField** function receives the input that the user provided in the modal, and orchestrates the communication with the back end:

```
const handleSaveField = async (selectedFieldId, selectedFieldType,
selectedFieldConstraint) => {
    setShowFieldModal(false);
    const formData = new FormData();

    if (selectedFieldType !== null)
        formData.append('type', selectedFieldType);

    if (selectedFieldConstraint !== null) {
        let group = null;
        if (selectedFieldConstraint.includes("Local")) {
            group = "/local/";
        } else {
            group = "/butterfly/core/";
        }
        const prototype =
selectedFieldConstraint.substring(selectedFieldConstraint.lastIndexOf('/') +
1);
```

```
        const constraint = group + prototype;
        formData.append('constraint', constraint);
    }

    formData.append('fieldId', selectedFieldId);

    postData(`${baseURL}/${prototypeName}/field`, formData).then((data) => {

        setOpenFieldAlert(true);
        setTimeout(() => {
            setOpenFieldAlert(false);
        }, 3000);
    });
};
```

The **handleSaveField** method orchestrates the saving of a new field on the front end. After closing the field modal, it constructs a FormData object containing essential field details. This function then sends a POST request to the server endpoint for creating a new field, with the URL dynamically generated based on the prototype name. Upon successful execution, it logs the server response and displays an alert, providing immediate feedback to the user about the success of the field addition. Error handling is incorporated to log any issues with the server request, ensuring a robust and responsive field creation process in the Type Manager Web App.

The **postData** function is a crucial part of this asynchronous communication logic. It uses the modern fetch API to send a POST request to the specified URL, in this case, targeting the endpoint responsible for creating a new field. The function's structure ensures proper configuration of the request, including containing the form data.

```
async function postData(url = "", formData = new FormData()) {
    const response = await fetch(url, {
        method: "POST",
        mode: "cors",
        cache: "no-cache",
        credentials: "same-origin",
        headers: {},
        redirect: "follow",
        referrerPolicy: "no-referrer",
        body: formData,
    });

    return response.json();
}
```

This structured approach to front-end logic ensures that the user's action triggers a well-defined process of creating a new field in the Type Manager Web App.

3.5.4.1.2 Adding a field to Batch

On the other hand, the **handleAddFieldToBatch** function manages the addition of the field to a batch for deferred actions. It constructs an action object and updates the local storage with the batched actions.

```
const handleAddFieldToBatch = async (selectedFieldId, selectedFieldType,
selectedFieldConstraint) => {
    setShowFieldModal(false);

    let constraint = "";

    if (selectedFieldConstraint !== null) {
```

```
        let group = null;
        if (selectedFieldConstraint.includes("Local")) {
            group = "/local/";
        } else {
            group = "/butterfly/core/";
        }
        const prototype =
selectedFieldConstraint.substring(selectedFieldConstraint.lastIndexOf('/') +
1);
        constraint = group + prototype;
    }

    const actionObject = {
        action: 'AddField',
        prototypeName: prototypeInfo.id,
        id: selectedFieldId,
        type: selectedFieldType,
        constraint: constraint,
    };

    const existingActions = JSON.parse(localStorage.getItem('batchActions'))
|| [];

    const isActionExists = existingActions.some(
        (existingAction) =>
            existingAction.action === actionObject.action &&
            existingAction.prototypeName === actionObject.prototypeName &&
            existingAction.id === actionObject.id &&
            existingAction.type === actionObject.type &&
            existingAction.constraint === actionObject.constraint
    );

    if (isActionExists) {
        setActionExistsAlert(true);
        setTimeout(() => {
            setActionExistsAlert(false);
        }, 3000);
    } else {
        existingActions.push(actionObject);
        localStorage.setItem('batchActions',
JSON.stringify(existingActions));

        setAddedToBatchAlert(true);
        setTimeout(() => {
            setAddedToBatchAlert(false);
        }, 3000);
    }
};
```

The **handleAddFieldToBatch** function manages the addition of a field action to a batch for deferred processing. Upon clicking the "Add to Batch" button, it constructs an action object with essential details such as the action type, prototype name, field ID, type, and constraint. Before adding the action to the batch, it checks for the existence of a similar action in the batch to prevent duplicates. If the action is unique, it updates the local storage with the new action and displays an alert to inform the user of a successful addition to the batch. The method ensures efficient batch management, allowing users to defer actions in the Type Manager Web App.

The **addToBatch** function, triggered by the "Add to Batch" button, calls the **handleAddFieldToBatch** method**.**

```
const addToBatch = () => {
    handleAddFieldToBatch(selectedFieldId, selectedFieldType,
selectedFieldConstraint);
};
```

3.5.4.1.3 Executing Batch actions

The Batch Actions Modal component serves as a user interface for executing batch actions within the Type Manager Web App. This modal is triggered when the user clicks the "Show Batch Actions" button on the Prototype page. The modal allows users to view, manage, and execute a collection of batch actions that they have assembled.

The Batch Actions Modal component encapsulates several key functionalities:

1. **Displaying Batch Actions**: Upon opening the modal, it presents a list of batch actions grouped by prototype names. Each action type is displayed along with its corresponding details.

```
Object.entries(groupedActions).map(([prototypeName, actions]) => (
    <div key={prototypeName}>
        <h3>{prototypeName}</h3>
        {Object.entries(actions).map(([actionType, actionList]) => (
            <div key={actionType}>
                <h4>{actionType} action</h4>
                <ul>
                    {actionList.map((action, index) => (
                        <li key={index} className="action-item">
                            {/* Display action details */}
                        </li>
                    ))}
                </ul>
            </div>
        ))}
        <hr />
    </div>
))
```

2. **Removing Batch Actions**: Users can remove individual batch actions by clicking on the delete icon associated with each action item.

```
<span
    className="delete-action-icon"
    onClick={() => handleRemoveAction(index)}>
      X
</span>
```

3. **Removing All Batch Actions**: The modal provides an option to remove all batch actions at once using the "Remove all batch actions" button.

```
<button className="remove-all-button" onClick={handleRemoveAllActions}>
    Remove all batch actions
</button>
```

4. **Executing Batch Actions**: Users can execute all batch actions by clicking on the "Save all actions" button.

```
<button className="batch-save-button" onClick={handleSaveBatch}>
    Save all actions
</button>
```

The **handleSaveBatch** method is responsible for executing the batch actions stored in the local storage and communicating with the backend server to process these actions.

When the user clicks the "Save all actions" button in the modal, this method is triggered.

```
const handleSaveBatch = async () => {
    const jsonString = convertArrayToString(batchActions);

    const formData = new FormData();
    formData.append('jsonString', jsonString);

    postData(`${baseURL}/batch`, formData)
        .then(response => response.text())
        .then(data => {

            if (data === "OK") {
                setOpenAlert(true);
                localStorage.removeItem('batchActions');
                const hasCreatePrototype = batchActions.some(action =>
action.action === 'CreatePrototype');
                handleCloseModal(hasCreatePrototype);
                setTimeout(() => {
                    handleCloseModal();
                }, 2000);
            } else {
                setOpenAlertError(true);
            }
        })
        .catch(error => {
            console.error('Error saving batch:', error);
            setResponse("An error occurred while saving the batch.");
            setShowResponse(true);
        });
};
```

It performs the following steps:

Convert Actions to JSON String: The method converts the batch actions stored in the local storage into a JSON string format. This string represents the payload to be sent to the server for processing.

```
const convertArrayToString = (batchActions) => {
    const actionsArray = batchActions.map(action => {
        const properties = Object.entries(action)
            .map(([key, value]) => {
                if ((key === 'inheritedDOLAR prototypes' || key ===
'removeDOLAR prototypes') && typeof value === 'string') {
                    return `"${key}": ["${value}"]`;
                } else {
                    return `"${key}": "${value}"`;
                }
            });

        return `{ ${properties.join(', ')} }`;
    });

    return `{ "actions" : [ ${actionsArray.join(', ')} ] }`;
};
```

Prepare FormData: The JSON string containing batch actions is appended to a FormData object. This FormData object is used to send the payload to the server via a POST request.

Send POST Request: The method sends a POST request to the backend server's /api/type/batch endpoint, along with the FormData containing the batch actions payload.

Handle Response: Upon receiving a response from the server, the method processes the response data. If the response indicates success (data === "OK"), it displays a success message using a Snackbar component and removes the batch actions from the local storage. If an error occurs, it displays an error message.

Error Handling: If an error occurs during the HTTP request, such as network issues or server errors, it catches the error and displays an error message indicating that an error occurred while saving the batch.

The **handleSaveBatch** method encapsulates the logic for processing and executing batch actions, including converting them into a suitable format, sending them to the server, and handling the server's response.

It ensures a smooth user experience by providing feedback on the execution status of batch actions, both for successful executions and errors.

5. **Feedback on Execution**: After executing the batch actions, the modal provides feedback on the execution status. If the execution is successful, a success message is displayed using a Snackbar component with a green color scheme. In case of an error during execution, an error message is shown using the same Snackbar component with a red color scheme.

```
<Snackbar open={openAlert} onClose={() => setOpenAlert(false)}
autoHideDuration={2000}>
    <Alert icon={false} severity="success" sx={{ width: '100%' }}>
        Batch actions executed successfully!
    </Alert>
</Snackbar>
<Snackbar open={openAlertError} onClose={() => setOpenAlertError(false)}
autoHideDuration={2000}>
    <Alert icon={false} severity="danger" sx={{ width: '100%' }}>
        An error occured while executing the batch..
    </Alert>
</Snackbar>
```

6. **Closing the Modal**: Users can close the modal by clicking the close button (X) or by clicking outside the modal. Additionally, the modal automatically closes after a certain duration when the batch actions are executed successfully.

```
<button className="batch-close-button" onClick={handleCloseModal}>
    X
</button>
```

3.5.4.2 Back-End Implementation

3.5.4.2.1 Adding a Field

The back-end functionality is encapsulated in the ApiController class, defined as a @RestController in the Spring Boot application. The relevant endpoint for adding a field is mapped using the @PostMapping annotation:

```
@RestController
@RequestMapping("/api")
@CrossOrigin(maxAge = 3600)
public class ApiController {

    private final ApiService apiService;

    public ApiController(ApiService apiService) {
        this.apiService = apiService;
    }

    @PostMapping("/type/{id}/field")
    public void createField(@PathVariable String id, @RequestParam String
fieldId, @RequestParam(required = false) String type,
                            @RequestParam(required = false) String
constraint, HttpServletResponse response) throws IOException {

        Prototype prototype = apiService.addField(id, fieldId, type,
constraint);
        JsonPrototypeEncoder encoder = new
JsonPrototypeEncoder(response.getOutputStream(), true);
        prototype.encode(encoder, true);
    }
}
```

In this example, when a POST request is made to /api/type/{id}/field, the createField method is invoked. It delegates the field creation operation to the ApiService, passing the necessary parameters.

The corresponding service class, ApiService, is responsible for interacting with the prototype and executing the necessary actions:

```
@Service
public class ApiService {

    private final VirtualSpaceSetup setup;

    public ApiService(VirtualSpaceSetup setup) {
        this.setup = setup;
    }

    private static final String workingDir = System.getProperty("user.home");

    public Prototype addField(String prototypeName, String id, String type,
String constraint) {
        constraint = (constraint != null) ? "/local/" + constraint : null;
        return new PrototypeActionsBatch(setup.getVirtualSpace(), setup)
                .addField("/local/" + prototypeName, id, type, constraint)
                .execute((PrototypeLifecycle.Result result) ->
result.modified().get(0));
    }

}
```

The ApiService class orchestrates the addition of a field, constructing the necessary parameters and utilizing the PrototypeActionsBatch to execute the field addition operation.

The PrototypeActionsBatch class plays a pivotal role in orchestrating various actions related to DOLAR prototypes. This class, which implements the PrototypeLifecycle interface, encapsulates a series of actions to be performed on DOLAR prototypes within the virtual space.

```
public class PrototypeActionsBatch implements PrototypeLifecycle {

    private final VirtualSpace space;
    private final DOLAR prototypestorage storage;
    private final List<PrototypeAction> actions = new ArrayList<>();
    private final Map<String, Prototype> typesBeingProcessed = new
LinkedHashMap<>();
    private final Set<String> markedForDeletion = new LinkedHashSet<>();

    public PrototypeActionsBatch(VirtualSpace space, DOLAR prototypestorage
storage) {
        this.space = space;
        this.storage = storage;
    }

    @Override
    public PrototypeLifecycle createPrototype(String prototypeName) {
        ensureNotExists(prototypeName);
        actions.add(new CreatePrototype(space, prototypeName,
typesBeingProcessed));
        return this;
    }
```

In the snippet above, the PrototypeActionsBatch class provides a convenient way to batch multiple actions, ensuring atomicity and consistency in the management of DOLAR prototypes. The createPrototype method adds a "create" action to the batch, indicating the intention to create a new prototype with the specified name.

Furthermore, the CreatePrototype class, located in the dislib.typeman.actions package, represents the specific action of creating a prototype. It extends the PrototypeActionBase class, which serves as the foundation for various prototype-related actions.

```
package dislib.typeman.actions;

import com.niovity.dolar.space.Prototype;
import com.niovity.dolar.space.VirtualSpace;

import java.util.Map;

public class CreatePrototype extends PrototypeActionBase {

    protected final VirtualSpace space;

    public CreatePrototype(VirtualSpace space, String prototypeName,
Map<String, Prototype> typesBeingProcessed) {
        super(Type.CREATE, space, prototypeName, typesBeingProcessed);
        this.space = space;
    }

    @Override
    public void execute() throws PrototypeActionException {
        Prototype proto =
space.getVirtualSpace().newPrototype(prototypeName);
        typesBeingProcessed.put(prototypeName, proto);
    }
}
```

The CreatePrototype class defines the execution logic for the creation action. When executed, it utilizes the virtual space to create a new prototype with the specified name and adds it to the typesBeingProcessed map, ensuring that the system keeps track of DOLAR prototypes currently in the process of creation or modification.

In essence, the combination of the PrototypeActionsBatch and CreatePrototype classes provides a structured and extensible framework for handling various prototype-related actions in the back end of the Type Manager Web App.

3.5.4.2.1 Executing Batch Actions

The controller defines an endpoint /api/type/batch that handles batch actions sent from the front-end. It receives a JSON string representing the batch actions via a POST request. The controller method responsible for handling this endpoint is named executeBatch.

```
@PostMapping("/type/batch")
public String executeBatch(@RequestParam String jsonString) throws
IOException {
    String fileName = "batchToExecute.json";
    String userHome = System.getProperty("user.home");
    String filePath = userHome + fileName;

    try (FileWriter writer = new FileWriter(filePath, false)) {
        writer.write(jsonString);
    }

    return apiService.executeBatch(filePath);
}
```

This method receives a JSON string containing batch actions as a request parameter named jsonString. Then, it writes the received JSON string to a file named batchToExecute.json in a specified directory. The method then delegates the execution of batch actions to a service named apiService.executeBatch(filePath) and returns the result obtained from the service.

The service layer contains the business logic for executing batch actions. The executeBatch method in the service class processes the batch actions stored in the JSON file and performs the necessary operations on DOLAR prototypes accordingly.

```
public String executeBatch(String jsonFile) {

    try {
        PrototypeLifecycle batch = Config.newLifeCycle();
        JsonActions jsonActions = new JsonActions();
        List<TimestampedActionData.ActionData> actions =
jsonActions.parseJson(new File(jsonFile));
        jsonActions.process(batch, actions);
        batch.execute((PrototypeLifecycle.Result result) -> {
           return "Batch executed successfully.";
        });
        return "OK";
    }  catch(Exception e) {
        return ("There was an error. Error: " + e.getMessage());
    }
}
```

The executeBatch method takes the file path of the JSON file containing batch actions as input. It initializes a new PrototypeLifecycle object and parses the JSON file to extract the batch actions. The batch actions are then processed using the process method, which applies the actions to the DOLAR prototypes. Finally, the execute method is called to execute the batch actions, and the result is returned as a string indicating success or failure.

In summary, this end-to-end example showcases a streamlined process of adding a field, involving coordinated interactions between the front-end React application, the back-end Spring Boot server, and the underlying service responsible for managing DOLAR prototypes.

4. CONCLUSION AND FUTURE WORK

In this thesis, we developed the Type Manager Web App, a user-friendly web-based interface that enhances the management of DOLAR prototypes within the existing Type Manager Extension. By addressing the limitations, the Type Manager Web App provides a seamless and interactive experience for users to create, modify, and manage DOLAR prototypes.

Through the development process, we implemented various endpoints and functionalities within the app, including retrieving prototype details, creating, and deleting prototypes, managing inheritance relationships, adding, and removing fields, and executing batch actions. These features empower users to efficiently work with multiple prototypes and perform bulk operations, significantly improving productivity and streamlining the prototype management process.

The integration of technologies such as Spring Boot, React, and Electron played a crucial role in the success of the project. Leveraging the power of Spring Boot, we achieved a robust and scalable backend, while React enabled the development of a dynamic and responsive user interface. By packaging the app as an Electron application, we ensured its compatibility across multiple platforms and provided users with the flexibility to use it as a standalone desktop application.

While the Type Manager Web App has demonstrated its effectiveness in addressing the identified limitations, there is still room for further improvement and expansion. Future work can focus on incorporating additional features, such as advanced search and filtering capabilities, as well as Role-Based Access Control.

In our forthcoming efforts, we aim to enhance the Type Manager Web App to better cater to the needs of our users. Firstly, we aim to enhance the functionality of the interface by incorporating search capabilities for specific fields and introducing filtering options. These additions will empower users to efficiently locate and manage relevant information, further enhancing the usability and effectiveness of the system. Additionally, we will introduce a feature that enables users to categorize new DOLAR prototypes into specific groups during the creation process, improving organization within the system. Furthermore, users will have more control over defining new fields, with the ability to add additional attributes beyond the basics, such as customizable metadata and descriptions.

The introduction of Role-Based Access Control (RBAC) will add an extra layer of security, allowing administrators to define specific permissions for different user roles. Another valuable addition could be implementing a feature that allows users to view the history of changes made to a prototype. This feature would provide transparency and accountability, allowing users to track the evolution of a prototype over time.

Overall, the development of the Type Manager Web App has successfully achieved its objectives of enhancing user experience and accessibility in managing DOLAR prototypes. The app serves as a valuable tool for researchers and developers working with DOLAR-based systems, facilitating efficient and streamlined prototype management processes.

APPENDIX

Table 1: Back-End Implementation Endpoints

Description	Endpoint	Method	Path Variable	Request Parameter	Response
Retrieve DOLAR prototypes by Namespace	/api/type/category/{id}	GET	namespace ID	-	list of prototype IDs

Retrieve Prototype Details	/api/type/{id}	GET	prototype ID	-	- prototype ID
					- inheritance details
					  ○ directed inherited prototypes
					  ○ transitively inherited prototypes
					- field details
					  ○ field ID
					  ○ type
					  ○ default value
					  ○ constraint
					  ○ defined in this
					- field group details
					  ○ field details

Create New Prototype	/api/type/{id}	POST	prototype ID	-	prototype details
Delete Prototype	/api/type/{id}	DELETE	prototype ID	-	-
Add Inheritance to Prototype	/api/type/{id}/inheritance	POST	prototype ID	inherited prototype ID	prototype details
Remove Inheritance from	/api/type/{id}/inheritance	DELETE	prototype ID	inherited prototype ID	prototype details

Prototype					
Add Field in Prototype	/api/type/{id}/field	POST	prototype ID	field ID type constraint	prototype details
Update Field in Prototype	/api/type/{id}/field	PUT	prototype ID	field ID type constraint	prototype details
Delete Field from Prototype	/api/type/{id}/field	DELETE	prototype ID	field ID	prototype details
Add Field Group to Prototype	/api/type/{id}/group	POST	prototype ID	group ID	prototype details
Delete Field Group from Prototype	/api/type/{id}/group	DELETE	prototype ID	group ID	prototype details
Add Batch Actions	/api/type/batch	POST	-	jsonString	feedback on the execution status

REFERENCES

[1] Batini, C., Cappiello, C., Francalanci, C., & Maurino, A. (2009). Methodologies for data quality assessment and improvement. *ACM computing surveys (CSUR)*, *41*(3), 1-52.

[2] Laranjeiro, Nuno & Soydemir, Seyma & Bernardino, Jorge. (2015). A Survey on Data Quality: Classifying Poor Data. 10.1109/PRDC.2015.41.

[3] Loshin, David. (2011). Business Impacts of Poor Data Quality. 10.1016/B978-0-12-373717-5.00001-4.

[4] Haug, Anders & Zachariassen, Frederik & Liempd, Dennis. (2011). The costs of poor data quality. Journal of Industrial Engineering and Management. 4. 10.3926/jiem..v4n2.p168-193.

[5] Colburn, T., Shute, G. Decoupling as a Fundamental Value of Computer Science. *Minds & Machines* **21**, 241–259 (2011). https://doi.org/10.1007/s11023-011-9233-3

[6] Hellerstein, J. M. (2003). Toward network data independence. *ACM SIGMOD Record*, *32*(3), 34-40.

[7] Saidis, K., Smaragdakis, Y. and Delis, A. (2011), DOLAR: virtualizing heterogeneous information spaces to support their expansion. Softw: Pract. Exper., 41: 1349-1383. https://onlinelibrary.wiley.com/doi/10.1002/spe.1050

[8] Gunasinghe, N., Marcus, N. (2022). Understanding the Language Server Protocol. In: Language Server Protocol and Implementation. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-7792-8_2

[9] React, "Documentation," React website, [Online]. Available: https://legacy.reactjs.org/docs/getting-started.html [Accessed: 12/03/2024].

[10] Facebook. (2021). React - A JavaScript library for building user interfaces. Retrieved from [React website]: https://reactjs.org/

[11] Awesome React. (n.d.). Retrieved from [GitHub repository]: https://github.com/enaqx/awesome-react

[12] React Community. (n.d.). Retrieved from [React website]: https://reactjs.org/community/support.html

[13] Electron, "Documentation," Electron website, [Online]. Available: https://www.electronjs.org/docs [Accessed: 12/03/2024].

[14] Electron. (n.d.). Retrieved from [Electron website]: https://www.electronjs.org/

[15] Electron Forge. (n.d.). Retrieved from [Electron Forge website]: https://www.electronforge.io/

[16] Material-UI, "MUI: A popular React UI framework," Material-UI website, [Online]. Available: https://mui.com/ [Accessed: 12/03/2024].

[17] Material-UI. (n.d.). Theming. Retrieved from [Material-UI website]: https://mui.com/customization/theming/

[18] Material-UI. (n.d.). Changelog. Retrieved from [Material-UI website]: https://mui.com/guides/changelog/

[19] Spring Boot, "Documentation," Spring Boot website, [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/index.html [Accessed: 12/03/2024].