

# Master's Thesis

## Concepts of Data-Oriented Programming for Web Technologies in the Digital Humanities

Thesis submitted in partial fulfillment of the requirements for the Degree of

Master of Science in Computer Science

to the Department of Mathematics, Natural Sciences and Computer Science at the  
University of Applied Sciences Mittelhessen

by

Sebastian Enns

May 24, 2023

Referent: Prof. Dr. Peter Kneisel

Korreferent: Prof. Dr. Andreas Kuczera



# Declaration of Independence

I hereby declare that I have independently written the present work, using only the specified sources and aids, and have clearly indicated any quotations. The work has not been submitted to any other examination authority in the same or similar form, nor has it been published.

Gießen, May 24, 2023

A handwritten signature in black ink, appearing to read 'Sebastian Enns' in a cursive style.

Sebastian Enns



This Thesis explores the utilization of data-oriented programming concepts in web technologies to improve software management and enhance data accessibility in the Digital Humanities field. Combining computational methods with traditional Humanities practices, the Digital Humanities field seeks to study and analyze cultural and historical information. However, efficiently handling and examining the increasing volume of data, particularly in creating scholarly digital editions, poses significant challenges.

By employing a comprehensive methodology involving literature analysis, use cases, concept development, and project implementation, this research investigates the potential of data-oriented design and programming to streamline digital edition development. Data-oriented design organizes code and data, improving efficiency in managing and analyzing complex data sets while ensuring scalability and maintainability over time.

Through two case studies, namely The Socinian Correspondence and Hildegardis Bingenensis projects, the practical implementation of developed data-oriented programming concepts in web applications for the Digital Humanities is demonstrated. These projects highlight the flexibility, maintainability, and modularity of digital editions.

The findings validate the effectiveness and potential of data-oriented programming concepts in digital edition development, contributing to advancements in the Digital Humanities field. Addressing limitations and exploring architecture-level concepts, technical constraints, and user perspectives are essential for future research and development. Therefore, this Thesis offers insights into improving software management and data accessibility, advancing the development of digital editions in the Digital Humanities.



Diese Thesis untersucht die Anwendung von datenorientierten Programmierkonzepten in Webtechnologien zur Verbesserung der Softwareverwaltung und Verbesserung der Datenzugänglichkeit im Bereich der Digitalen Geisteswissenschaften. Durch die Kombination von computerbasierten Methoden mit traditionellen geisteswissenschaftlichen Praktiken streben die Digitalen Geisteswissenschaften die Untersuchung und Analyse von kulturellen und historischen Informationen an. Das effiziente Handhaben und Untersuchen des stetig wachsenden Datenvolumens, insbesondere bei der Erstellung von wissenschaftlichen digitalen Editionen, stellt jedoch erhebliche Herausforderungen dar.

Diese Forschung untersucht mittels einer umfassenden Methodik, die Literaturanalyse, Anwendungsfälle, Konzeptentwicklung und Projektdurchführung umfasst, das Potenzial von datenorientiertem Design und Programmierung zur Vereinfachung der Entwicklung digitaler Editionen. Datenorientiertes Design organisiert Code und Daten, verbessert die Effizienz im Umgang mit und in der Analyse von komplexen Datensätzen und gewährleistet dabei Skalierbarkeit und Wartbarkeit über die Zeit.

Anhand von zwei Fallstudien, nämlich den Projekten Die Sozinianischen Briefwechsel und Hildegardis Bingensis, wird die praktische Implementierung der entwickelten datenorientierten Programmierkonzepte in Webanwendungen für die Digitalen Geisteswissenschaften demonstriert. Diese Projekte unterstreichen die Flexibilität, Wartbarkeit und Modularität digitaler Editionen.

Die Ergebnisse bestätigen die Wirksamkeit und das Potenzial von datenorientierten Programmierkonzepten in der Entwicklung digitaler Editionen und tragen zur Weiterentwicklung im Bereich der Digitalen Geisteswissenschaften bei. Die Auseinandersetzung mit Einschränkungen und die Erkundung von architekturellen Konzepten, technischen Beschränkungen und Benutzersichten sind für zukünftige Forschung und Entwicklung unerlässlich. Insofern bietet diese Arbeit Einblicke in die Verbesserung der Softwareverwaltung und Datenzugänglichkeit und fördert die Entwicklung von digitalen Editionen in den Digitalen Geisteswissenschaften.





# Contents

1	Introduction	1
1.1	Statement of the problem	1
1.2	State of the art	2
1.3	Research	2
1.4	Methodology	4
2	Fundamentals	5
2.1	Project: The Socinian Correspondence	5
2.2	Project: Hildegardis Bingensis: Liber epistolarum	7
2.3	GraphQL - Query Language	8
2.3.1	Fundamentals	9
2.3.2	Operational Semantics	10
2.3.3	Examples	14
2.3.4	Performance	16
2.4	Neo4j - Graphdatabase	17
2.4.1	Fundamentals	17
2.4.2	Operational Semantics	19
3	State of the art: Data-oriented paradigm	21
3.1	Data-oriented Programming	21
3.1.1	Principles of data-oriented programming	24
3.1.2	Differences between object and data-oriented programming	31
3.2	Data-oriented Architecture	32
3.3	Conclusion	34
4	Concept Development	35
4.1	Use cases	35
4.1.1	Architecture layer	36
4.1.2	Application layer	41
4.2	Requirement Analysis	43
4.3	Architecture Concepts	45
4.3.1	Utilizing data-oriented design	46
4.3.2	Service Traversal	47

4.4	Application Concepts . . . . .	48
4.4.1	Utilizing data-oriented programming . . . . .	49
4.4.2	Data Access Objects . . . . .	50
4.4.3	Data Traversal . . . . .	52
5	Implementation . . . . .	53
5.1	Overview . . . . .	55
5.1.1	Technology Stack . . . . .	56
5.1.2	Data Refinement . . . . .	58
5.2	Concept Implementation . . . . .	60
5.2.1	The Socinian Correspondence . . . . .	61
5.2.2	Hildegardis Bingensis: Liber epistolarum . . . . .	64
6	Evaluation . . . . .	65
6.1	Use Cases . . . . .	65
6.2	Concepts Re-implementation . . . . .	66
6.3	Concepts Acceptance . . . . .	67
6.4	Outcomes and Impact . . . . .	68
6.5	Limitations and Future Work . . . . .	69
7	Conclusion . . . . .	71
7.1	Summary . . . . .	71
7.2	Contributions . . . . .	73
7.3	Limitations . . . . .	74
7.4	Future Work . . . . .	76
	Bibliography . . . . .	79
	Abbreviations . . . . .	83
	List of Figures . . . . .	83
	Listings . . . . .	85
	Appendices . . . . .	85
A	Use Cases . . . . .	87

# 1 Introduction

The Digital Humanities is a fascinating field that merges computational methods with traditional Humanities practices to study and analyze cultural and historical information. With the rapidly increasing volume of data in the Humanities, it is becoming increasingly crucial to find efficient ways to handle and examine this information, especially when it comes to creating scholarly digital editions [Pie16]. They are digital replicas of texts, which aim to offer a dependable, scholarly representation of the original work. These editions can include the complete text of the work and additional resources like annotations, translations, and supplementary materials [Pie16].

Developing scholarly digital editions calls for a great deal of attention to detail and accuracy, precision, and completeness. It also involves the utilization of various tools and techniques for digitization, data management, and text analysis. As a result, developing these digital editions demands a collaboration of many skills and expertise from different fields, including knowledge of the Humanities and Computer Science.

## 1.1 Statement of the problem

The use of data-oriented design and data-oriented programming in the creation of digital editions has the potential to improve the efficiency of the edition development process. By organizing code and data in a way that maximizes the use of available computing resources, data-oriented design can help to streamline the management and analysis of large and complex data sets. In addition, the use of data-oriented programming can help to ensure the scalability and maintainability of digital editions over time [Fed20].

Overall, the utilization of data-oriented programming in the development of scholarly digital editions has the potential to make a significant contribution to the field of Digital Humanities, as well as to the wider field of Software Engineering and Digital Scholarship. However, there is a need for research on the application of data-oriented programming in this context in order to identify best practices and to better understand its potential benefits and limitations.

### 1.2 State of the art

Scholarly digital editions are digital versions of physical texts that often include additional layers of interpretation and analysis. These editions are usually created using specialized software tools like XML editors and TEI encoders, which allow for the creation and management of structured, annotated text [Cum13]. This structure enables the implementation of sophisticated digital editions that go beyond simple digitization and provide new ways of studying and researching texts.

XML, a standardized format for representing and encoding text, is a key technology for the creation of digital editions [Nel01]. It allows for the development of structured, annotated text that can be easily manipulated and queried and is often used with TEI encoding schemes, which provide standardized tags and attributes for representing the structure and content of texts [Nel01]. XML editors like Oxygen are software tools for creating and managing XML-encoded text, often with user-friendly interfaces and features like validation and transformation capabilities [Bar04]. These tools may also support TEI encoding schemes for creating compliant digital editions. XML documents can be imported into eXist, a database management system for XML data, where they can be stored and queried using XQuery and transformed into relational database formats like SQL for use in web-based applications [Wie, Mei03].

Web-based technologies, such as PHP, HTML, and CSS, and content management systems like Typo3, are commonly used in the development of scholarly digital editions [Göh04]. These technologies enable the implementation of user-friendly, web-based interfaces for editing and encoding text and have been shown to facilitate the construction of more sophisticated editions that go beyond simple digitization and provide new ways of studying and researching texts [Dah06]. The integration of these technologies with content management systems has also been shown to increase the efficiency and reliability of digital edition creation and publishing workflows. They can be used to develop web-based platforms for publishing and sharing digital editions, making them more widely available to the research community [Dah06].

### 1.3 Research

The use of data-oriented programming concepts in web technologies has the potential to enhance data management and analysis in the Digital Humanities, but there is still a lack of understanding of how to effectively utilize these concepts in web contexts. This gap in knowledge prompts the question: how can data-oriented programming concepts be utilized in web technologies to improve software management and data accessibility in the Digital Humanities? By addressing this question, it is possible to gain a deeper

understanding of the benefits and limitations of data-oriented programming in the Digital Humanities and how it can be used to improve the usability and functionality of existing tools and systems. This can contribute to the advancement of the field and allow researchers to more effectively manage and analyze data in the Digital Humanities, but also enabling software engineers to create digital editions faster and efficient.

One area of research is the building of data-oriented architectures and applying concepts of data-oriented programming onto the application layer of data-oriented architectures. This research focuses on developing architectures that can handle significant big volumes of data and support unusual processing requirements. To achieve this, it is necessary to explore different design patterns and principles to create scalable and efficient systems. This includes the development or usage of effective data storage and retrieval mechanisms, communication components that can facilitate the exchange of data between different components in the system, and data processing mechanisms that can handle significant big volumes of data while maintaining high levels of performance. By leveraging the benefits of data-oriented programming concepts, this Thesis aims to develop concepts of data-oriented architectures that can meet the growing demands of modern applications in the Digital Humanities [Fab18, Jos07].

Another area of research in the use of data-oriented programming concepts in web technologies is the development of data models and schemes for representing and encoding digital objects across different platforms or architectures. These models and schemes provide a standardized set of conventions for representing and structuring digital objects, and enable the creation of interoperable and reusable digital objects that can be easily shared and reused across different systems [Fla18, Fla15]. Technologies like GraphQL can be used in conjunction with these data models and schemes to enable the efficient and flexible management and data traversal. By developing and using these data models and schemes, it will be possible to understand and use digital objects in the Digital Humanities more effectively [Har18b].

The aim of this research is to explore the potential of using data-oriented programming concepts in web technologies to improve the understanding of data sets and their interrelationships in the Digital Humanities. This includes examining the benefits and limitations of using data-oriented programming concepts in software engineering and how they can be used to enhance the usability and functionality of existing tools and systems. Additionally, the research aims to identify and address any challenges or issues that may arise in the process of utilizing data-oriented programming concepts in web technologies for data management and analysis. Through this research, the goal is to contribute to the advancement of the field and to enable researchers and practitioners to more effectively understand data and use them to build fully maintainable digital editions in the Digital Humanities with ease.

## 1.4 Methodology

The study and development of data-oriented programming concepts in the Digital Humanities is an interdisciplinary field that combines elements of Computer Science, Information Technology, and the Humanities. Various scientific research methods such as literature analysis, use cases, concept development, and project implementation can be used to study this topic, ensuring the accuracy and reliability of the findings.

To conduct a comprehensive analysis of the topic, an extensive review of the literature on data-oriented programming concepts and data-oriented architecture in the Digital Humanities and related domains will be carried out. The review will encompass various published materials such as academic papers and books that explore the use, applicability, and significance of data-oriented programming concepts and architecture in the field. By performing a detailed examination of the available literature on data-oriented architecture, it is possible to develop a better comprehension of the current state of research and pinpoint areas that require further investigation to bridge existing gaps in knowledge. The use of data-oriented architecture is increasingly relevant in various domains, including the Digital Humanities, as it enables the effective management and utilization of large volumes of data for complex tasks [Fab18].

In addition to the literature review, the concept development of data-oriented programming will include use cases and a requirement collection. Use cases provide an effective research methodology to obtain an overview of necessary requirements. Analyzing examples of how concepts can be applied in various projects and situations can offer valuable insights into their practical usage and enable the identification of best practices for their implementation. The requirement collection process will involve gathering and analyzing stakeholder needs and expectations to identify the key functionalities and features required for successful implementation of data-oriented concepts and architecture in the Digital Humanities. By integrating these research methods, it will be possible to develop a comprehensive understanding of the practical application and requirements for an effective implementation.

After defining the data-oriented programming concepts, the next step is to advance towards the realization and implementation of projects that can leverage these concepts. This process encompasses designing and developing digital platforms that integrate data-oriented concepts, as well as performing user testing and evaluation to guarantee the effectiveness and usability of the implemented concepts. To ensure the success of these projects, it is critical to engage with experts and stakeholders in the field and solicit their valuable feedback.

## 2 Fundamentals

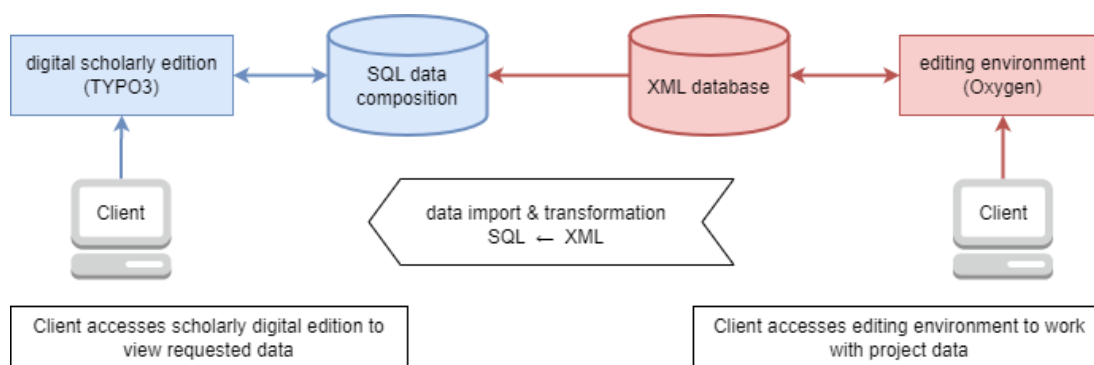
This chapter provides a foundation for understanding the core concepts and principles that are being used in this Thesis. It is designed to give readers a basic understanding of the terminology, processes, and techniques that will be covered in greater detail in later chapters. This chapter is essential for building a strong foundation and will serve as a reference point as readers progress through the rest of this Thesis.

### 2.1 Project: The Socinian Correspondence

The Socinians were a Unitarian religious community that emerged in the sixteenth century and emphasized the subjective rationality of the individual in matters of religious faith. They developed a close-knit correspondence network with leading figures in theology, astronomy, and politics, and surviving letters of Socinian adherents are valuable resources for understanding the transconfessional struggle in Europe during the Early Modern period. Socinian theology introduced radical tolerance into contemporaneous debates by allowing practitioners to engage in evidence-based scientific discussions without traditional metaphysical considerations. The Socinian Correspondence Project aims to create a historical-critical edition of the Socinian correspondence, featuring textual-critical commentaries and descriptions of the subject matter and content of the letters, as well as images of astronomical phenomena and previously unpublished political reports [Tos, Dau16].

The already existing scholarly digital edition encompasses the period from roughly 1580 to 1740 and will include 2047 letters cataloged to date. The letters reflect the interdependencies between historicizing, rationalist approaches to religious subject matter, early modern scientific research and methodology, and political correspondence. They also provide valuable insight into the wider scholarly networks that spanned across Europe during the period under investigation. The digital edition enables analysis of the material using criteria such as geography, persons involved, and topics discussed by the correspondents through a systematic indexing of the text. Moreover, it reflects the pervasive interconnectivity of the Socinian correspondence network, which traverses boundaries of religion, geography, subject matter, and media [Tos, Dau16].

## Technical background



**Figure 2.1:** The Socinian Correspondence: Project Workflow.

The Socinian Correspondence has already been integrated into a scholarly digital edition using software standards that were up-to-date at the time of creation. The component of the letter is a subset of TEI5 P5 XML [Tos]. The editorial work on that component is done using the Oxygen XML editor and the resulting content is stored in an XML database, as depicted in the Figure 2.1. Correspondence metadata and links to people, places, or entities were noted as XML lists. The scholarly digital edition itself is a PHP-based open source content management system called Typo3, which has been extended by various features such as data versions, Linked Open Data (LOD) standards, and URIs [Tos, Bau11]. Typo3 requires a working database on SQL, in this respect it was necessary to import the data from the XML database and transform it into an appropriate format to make it visible on the scholarly digital edition.

However, the character of a network is not easily modeled with XML, and as a result, there have been attempts to transfer the data into a graph database. The project continued to pursue the goal of combining XML and graph technologies to facilitate the mapping and linking of Socinian correspondences in the most effective and efficient way possible. The challenges faced include converting the existing data set of XML data into an appropriate format for the graph database and creating a user-friendly environment for data analysis and visualization [Tos]. While object-oriented programming principles were considered during the creation of the scholarly digital edition, the project's main focus is on processing and analyzing data after it has been imported and transformed into the required database. Therefore, the use of data-oriented programming may prove to be more efficient here. Although various technologies have been already mentioned and implemented, the priority is to create an environment that allows for easy analysis and visualization of generic data sets. Therefore, in order to preserve, continue, and display existing and future data sets, the project needs to take into account both the project requirements and the characteristics of the data sets. One possible approach to meet these needs and expand the project's capabilities is to shift from object-oriented to data-oriented programming paradigm.



## 2.2 Project: Hildegardis Bingensis: Liber epistolarum

Hildegard of Bingen was a 12th century German Benedictine abbess, writer, composer, philosopher, and polymath. She is best known for her extensive works on theology, medicine, and natural history, as well as for her contributions to the development of medieval music and drama [Fur18]. Hildegard was born in the town of Bockelheim, in the present-day German state of Rhineland-Palatinate, in 1098. She entered the monastery at the age of eight and was later appointed as the abbess of the monastery of Saint Rupert in Bingen. She is also known for her role as a spiritual leader and adviser to popes and other important figures of the time [Fur18].

Apart from her visionary writings, sermons, linguistic works, and spiritual chants, Hildegard wrote several letters. These letters were written between 1146/47 and her passing and are preserved in a complex tradition. They exist as both scattered fragments and as a collection, with six relevant manuscripts from Hildegard's temporal and spatial environment. When comparing the six relevant collections for the project, significant variations can be observed concerning the texts' extent, addressees, and vocabulary [lib12]. This project presents the complete work of Hildegard's letters, which focuses on the book of letters 'Liber epistolarum' from the Giant Codex and has been composed into a theological work. Therefore, the individual letters in this collection are viewed as parts of a theological-literary composition rather than as evidence of a historically-occurring exchange of letters [lib12].

### Technical background

Unlike the previous project, the Hildegardis Bingensis project currently lacks its own scholarly digital edition. However, the project's editorial work is progressing rapidly, and there is already a need for such a platform. Both the project and the compiled data set have specific requirements that need to be met.

Currently, the text and metadata are stored in a Neo4j graph database that incorporates the TEI's semantics. The project plans to model multidimensional structures using graph-based data, which would allow for the creation of various annotation hierarchies. To carry out the editorial work, the project uses the Speedy standoff property editor [Nei12], which has been customized to meet the project's specific needs. While the complete text of the Liber epistolarum is already available in the editor, further markup of all relevant information can be done using it.

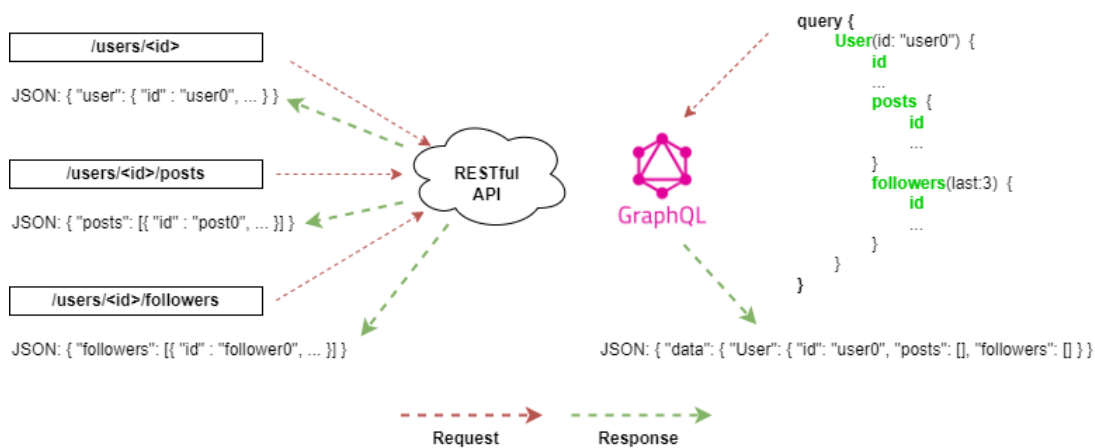
Therefore, it is important for the future scholarly digital edition to be able to work with the existing and expandable data set, analyze it, and visualize it in a suitable context. The main focus is to work with and around the data sets.

## 2.3 GraphQL - Query Language

According to its formal specification, GraphQL is both a query language and an execution engine that can describe the data model requirements and capabilities for client-server applications. It is not mandatory to use a particular programming language or persistence mechanism to create GraphQL application servers (GraphQL APIs). Instead, GraphQL uses a unified language to encode the capabilities of a type system based on five design principles: hierarchical, product-centric, strong-typing, client-specified queries, and introspective [Fou18, QM23].

In general, GraphQL is a powerful tool for building APIs that can offer more flexibility and efficiency compared to traditional REST APIs. Its ability to allow clients to request specific data from a server and define a flexible and hierarchical data model can help improve the efficiency of data retrieval and reduce the strain on server resources. Additionally, GraphQL's support for the creation of APIs that can be used across a variety of platforms and devices makes it a valuable tool for developers [QM23].

Moreover, the ecosystem of tools and libraries available for GraphQL integration into existing systems can further enhance its usability. These tools include building GraphQL servers, executing GraphQL queries, and integrating GraphQL into various front-end frameworks. By leveraging these tools, developers can quickly and easily implement GraphQL into their existing systems and take advantage of its benefits [Bri20].

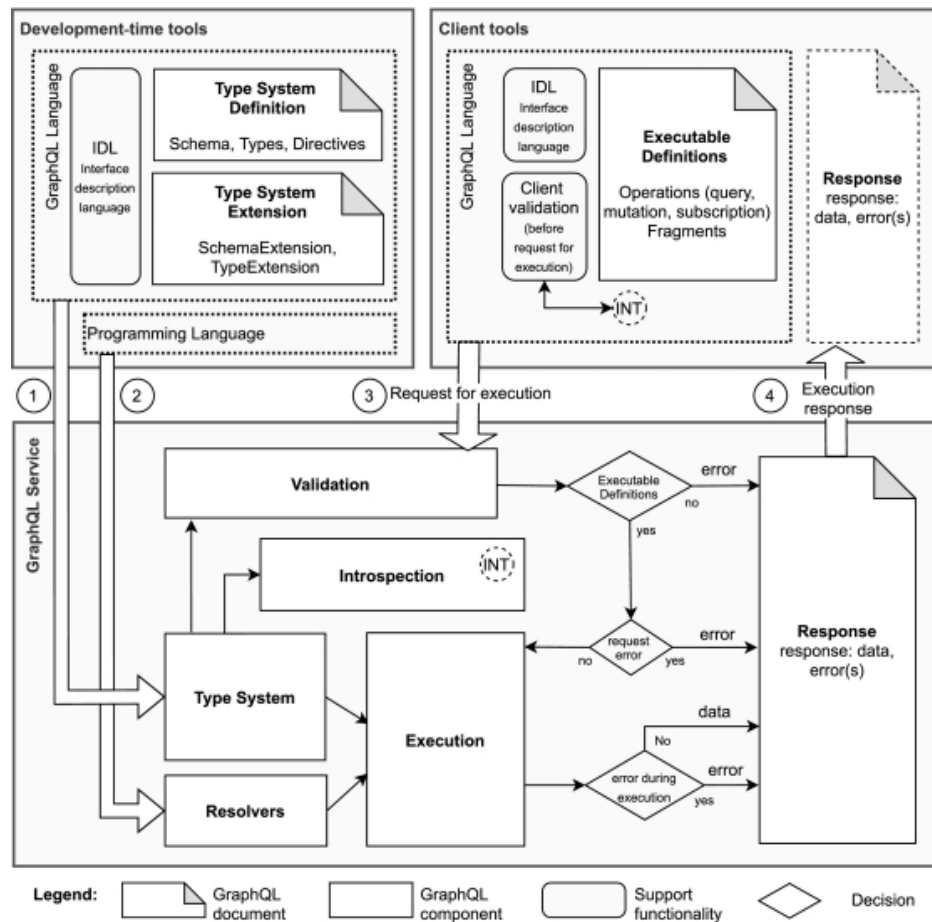


**Figure 2.2:** Example of over- and under-fetching in REST & GraphQL [QM23].

GraphQL offers a more adaptable and efficient method of creating APIs than REST. REST follows a predetermined set of endpoints, as illustrated in Figure 2.2, that return predefined data structures. In contrast, GraphQL allows querying of all necessary data within a single request, enhancing the flexibility and efficiency of data retrieval [QM23].

Interacting with REST APIs may require clients to make multiple requests to obtain all the necessary and related data, which can lead to over-fetching of data. Over-fetching occurs when the client receives more data than it needs for its current requirements. This can result in inefficient utilization of server resources, sluggish performance, and unnecessary network traffic. GraphQL allows clients to acquire related data in a single request, preventing over-fetching and improving data retrieval efficiency [QM23].

### 2.3.1 Fundamentals



**Figure 2.3:** Principal components interaction of the GraphQL paradigm [QM23].

Figure 2.3 shows the GraphQL paradigm and its functional structure according to the GraphQL formal specification [Fou18]. The paradigm includes the interaction between high-level components such as the GraphQL language and its grammar, type system, introspection, as well as execution and validation engines [QM23].

The process starts with (1) the development-time tools that use the GraphQL language and Interface Description Language (IDL) to define and generate the type system or type system extensions. This is followed by the generation of the service (GraphQL API),

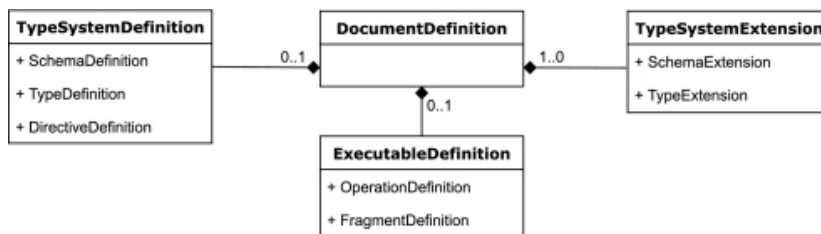
which involves generating type system Resolvers using a programming language. The service generated from the development-time tools is then consumed (2) [QM23, Fou18].

To execute an operation, a document containing executable operation definitions (queries, mutations, or subscriptions) or fragments is generated and sent to the GraphQL service for execution (3). Prior to sending the request, the client tools utilize the GraphQL service’s introspection to perform syntax validation. The GraphQL service validates that the document solely contains Executable Definitions and then proceeds to execute and retrieve the requested data from the Type System and Resolvers [QM23, Fou18].

Finally, the GraphQL service sends the result as a response document, usually in JSON format, to the client tools (4). The response document may include error messages if there are errors during execution [QM23, Fou18].

### 2.3.2 Operational Semantics

A GraphQL document can include various definition instances such as Type System Definitions of GraphQL services or the actual queries that clients execute on them. The GraphQL language offers a complete syntax to generate these documents. This means that a single GraphQL document can comprise of different instances of definitions, including Type System, Type System Extension, or Executable Definitions [QM23, Fou18].



**Figure 2.4:** GraphQL document definition [QM23].

The Type System Definition outlines the data capabilities of a GraphQL server’s Type System, along with the input types of the query variables. Type System Extension represents a GraphQL type system that extends from an original type system. This construct can be used to represent data that a GraphQL client only accesses locally or by a GraphQL service that is itself an extension of another service. Executable Definition documents contain operations or fragments that can be executed [Fou18, QM23].

In addition, the GraphQL language features an IDL to define the type system of a GraphQL service. This IDL is utilized by various tools to provide useful utilities such as client code generation or service bootstrapping. The following describes the main components of a GraphQL service [Fou18, QM23].

## Schema

The capabilities of a GraphQL service's collective type system are defined in terms of types and directives, as well as the root operation types for each kind of operation, namely query, mutation, and subscription [Fou18, QM23].

## Types

The fundamental unit of the GraphQL schema is a type, which can be one of the following [Fou18, QM23]:

- Scalars: primitive value; string, int, float, bool, or an **unique identifier (ID)**.
- Enums: set of possible values.
- Objects: In GraphQL, an object is a group of fields where each field has a type within the system. Fields are like functions that return values and can sometimes take arguments that alter their behavior, which are typically mapped directly to a function within a GraphQL server implementation. As a result, the response trees are made up of Scalars and Enums at the leaf level, with intermediate levels being composed of Objects. This allows for flexible type hierarchies to be defined.
- Input objects: set of input fields; it can be Scalars, Enums, or other Input Objects.
- Interfaces: list of fields.
- Unions: list of possible types.

There are also two wrapping types that effectively wrap or contain a 'named type':

- List: a field that represents a list of other types; for this reason, the list type wraps another type.
- Non-null: wraps another type and denotes that the value will never be null.

## Directives

In GraphQL, directives offer a means to specify custom runtime execution and type validation behavior in a GraphQL document. These directives can be used to convey supplementary information related to operations, fragments, fields, and types [QM23].

### Operations

Executable Definitions consist of three types of operations that are the primary elements of a GraphQL API. These operations include queries, mutations, and subscriptions. Queries are used for retrieving data in a read-only manner, while mutations are used for updating data and then retrieving it. Finally, subscriptions are used for long-running requests that fetch data in response to events from the source [Fou18, QM23].

### Variables

Introducing variables in GraphQL queries maximizes their reusability, while avoiding expensive string building in clients during runtime.

### Input Values

Input values in GraphQL can be represented as Scalar values, Enumeration values, Lists, or Input Objects.

### Arguments

In GraphQL, arguments can modify the behavior of fields that allow them to be used. These arguments are usually mapped to functions within the implementation of a GraphQL server.

### Selection Sets

Selection sets are used to limit GraphQL requests to only the necessary subset of information by primarily including fields. The aim is to avoid the problems of over-fetching and under-fetching data.

## Fields

Fields are used to represent individual pieces of data and can describe complex data or relationships with other data. They are often thought of as conceptual functions that return specific values.

Each field of each type is associated with a resolver function that must be provided by the GraphQL server developer. The resolver function is invoked when a field is executed to produce the response value [Fou18, Pri17].

## Fragments

Fragments serve as the fundamental building block of composition in GraphQL, enabling the reuse of frequently repeated field selections.

## Responses

The response generated by a GraphQL operation is usually in JSON format and represented as a map consisting of three entries: data, errors, and/or extensions. The data entry contains the result of the requested operation's execution, while errors are present if the execution encounters any errors. The extensions entry is used by implementers to extend the protocol without any content restrictions [Fou18, QM23].

## Introspection

Introspection of the schema in a GraphQL server is possible by querying the type system using the GraphQL language itself.

## 2.3.3 Examples

GraphQL performs syntactical validation and ensures that requests are unambiguous and error-free within a given schema before execution. However, a request that has been previously validated may be executed without explicit validation by a GraphQL service. Only Executable Definitions of operations and fragments are considered during execution, while Type System Definitions and extensions are non-executable and are not taken into account [Fou18].

```

(a) schema {
  query: Query
  mutation: Mutation
}

(b) type Query {
  humans (id: Int!): [Human]
  droid (id: ID!): Droid
}

(c) type Mutation {
  createHuman (human: HumanInput): Human
}

(d) enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

(e) type Character {
  name: String!
  appearsIn: [Episode]!
}

(f) interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}

(g) type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  totalCredits: Int
}

(h) type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}

(i) input HumanInput {
  name: String!
  friends: [String]
  appearsIn: [String]!
  totalCredits: Int
}

(j) union SearchResult = Human | Droid

```

**Figure 2.5:** GraphQL type system example [QM23].

Figure 2.5 displays examples of key components in a GraphQL Type System Definition. It includes the following elements [QM23, Fou18]:

- (a) Service schema that outlines available operations (query and mutation).
- (b) Query types (humans, droid) that take Scalar type arguments and return Lists.
- (c) Mutation (createHuman) that takes an input type argument and returns an object.
- (d) Enum type (Episode) that specifies a set of values.
- (e) Type object (Character) including fields, "!" indicating it does not accept null values.
- (f) Interface type (Character) with additional fields compared to (e).
- (g) Implementation (Human) of the Character interface that includes the interface fields and an additional field (totalCredits).
- (h) Implementation (Droid) of the Character interface that includes the interface fields and an additional field (primaryFunction).
- (i) Input Object (HumanInput) with fields used in the createHuman mutation.
- (j) Union type (SearchResult) used to search for data in Human or Droid types.



GraphQL processes a request by using its execution facility, which operates within the boundaries of a schema provided by the relevant GraphQL service. Requests that pass the validation rules are executed, while those with validation errors fail without execution, accompanied by a list of reported errors within the response. It is important to note that requests that pass validation can still present data errors during execution. In such cases, the execution returns both the data result and an error description [Fou18, Pri17]. Below is an example of a possible mutation with some query variations:

```
(a) Mutation example
mutation NewHumans{
  createHuman(name:"Leia Organa", friends:["Han Solo", "R2-D2"],
    appearsIn:["NEWHOPE", "EMPIRE", "JEDI"], totalCredits: 1000000)
  {
    id
    name
    friends{ name }
  }
}

(b) Query example 1
{
  humans {
    id
    name
    friends{ name }
  }
}

(c) Query example 2
query queryOneHuman{
  humans (id:1) {
    id
    name
    friends{ name }
  }
}

(d) Query example 3
query queryWithFragment{
  humans (id:1) {
    ...fragmentTwoFields
    friends{ name }
  }
}
fragment fragmentTwoFields on Human {
  id
  name
}
```

**Figure 2.6:** GraphQL type system example [QM23].

In Figure 2.6, examples of common operations in the executable definition are presented, using the type system defined in Figure 2.5 [QM23, Fou18]. These definitions are sent to the GraphQL service by client tools for request execution:

- (a) Demonstrates the execution of the mutation operation (`createHuman`) through the `NewHumans` request, and shows three variations of the query operation (`humans`).
- (b) Shows the execution of the query without defining the request name.
- (c) Defines the request (`queryOneHuman`) to execute the query ‘`humans`’ with the ‘`id`’ parameter set to one, returning the record with that identifier.
- (d) Defines the request (`queryWithFragment`) to execute the query using the `FragmentTwoFields` fragment, which selects specific fields from the `Human` implementation.

```
{
  "data": {
    "humans": [
      {
        "id": 1,
        "name": "Leia Organa",
        "friends": [ "Han Solo", "R2-D2" ]
      }
    ]
  }
}
```

**Figure 2.7:** Response to mutation and query operations executed in Figure 2.6 [QM23].

### 2.3.4 Performance

When it comes to building information systems of high quality that can cater to varying sizes, standardization and mechanisms play a pivotal role. Web services provide the capability to incorporate Quality of Service (QoS) measures in their offerings, and there are numerous frameworks available for this purpose. QoS encompasses various aspects, including functional and non-functional characteristics, that are crucial for optimizing the performance of web services. It is essential for each category of web service to establish benchmarks to assess their performance. Among service providers, the QoS attribute of web services holds a high priority due to the unpredictable and dynamic nature of web services, including their performance [Har18b, Red13, Jam12, Ced16].

Performance plays a crucial role in the context of web services, with its significance being widely recognized. Web services have established their superiority and ease of implementation compared to other existing technologies in this domain. The efficiency of a web service is reflected in how quickly it can process and respond to requests. This performance aspect can be measured using indicators such as throughput and response time. Throughput refers to the number of requests that can be serviced within a specific time frame, while response time measures the time taken by the server to reply after processing. It's important to note that both response time and throughput are influenced by the workload experienced by the web server at any given moment [Har18b, Red13].

A recent research paper which was evaluating the performance of GraphQL in contrast to REST has found that GraphQL outperforms REST in terms of performance related to resource consumption and utilization. The study showed that GraphQL is more efficient and stable in utilizing processing resources for CPU load performance, with a minimal variation. Additionally, GraphQL demonstrates superior memory consumption efficiency and stability [Law21].

The researchers attribute the stability of GraphQL's resource consumption and CPU utilization to its ability to selectively request data fields or properties for specific requests, resulting in more efficient utilization of computing resources for fixed data. Moreover, GraphQL's single endpoint request and flexible response formats allow clients to avoid large data transfers or sparse data responses, which can occur with REST API services. This results in more efficient data transfers and optimized resource utilization [Law21].

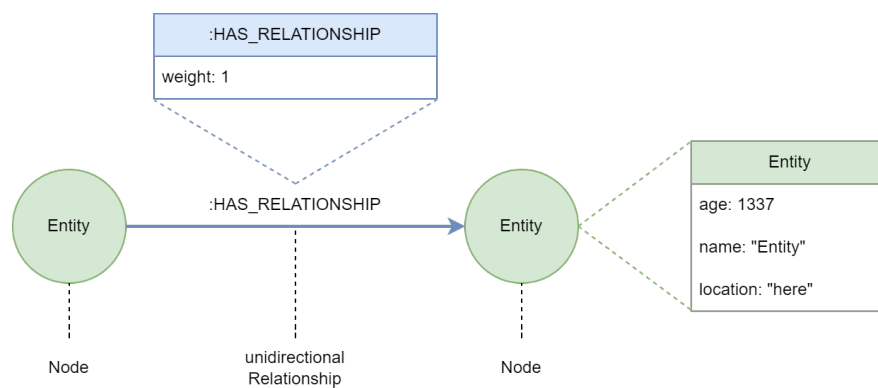
In summary, according to the research findings, GraphQL performs better than REST in terms of performance, specifically in resource consumption and utilization, due to its more efficient CPU load performance, memory consumption, and optimized data transfer approach. Please note that exact percentages and numerical values can be found in the original research paper [Law21].

## 2.4 Neo4j - Graphdatabase

Neo4j is a widely-used graph database management system that is designed for storing, retrieving, and managing graph data. Graph data represents relationships and connections between entities, with nodes representing entities and edges representing relationships. Neo4j uses a labeled property graph model with rich data modeling capabilities, and supports a query language called Cypher for querying graph data in a concise and expressive manner. It is known for its high performance, scalability, and features such as data integrity and ACID compliance. Neo4j is commonly used in domains like social networks, recommendation systems, fraud detection, and more. It provides a flexible and efficient solution for managing and querying graph data, making it popular among developers and data professionals for building graph-powered applications [VB14, Gui17].

### 2.4.1 Fundamentals

Neo4j, a graph database management system, provides a powerful way to represent data using the graph data model. In a graph, data is organized into nodes, which can be thought of as entities or objects, and relationships, which represent the connections or associations between nodes. Nodes are the fundamental units of data in a graph database and can have properties associated with them, such as name, age, or location, to store additional information about the entity [VB14, Neo23].



**Figure 2.8:** An example of a labeled property graph in Neo4j.

Relationships as shown in Figure 2.8, on the other hand, connect nodes and can also have a type, describing the nature of the connection, and properties associated with them, capturing additional information and characteristics about the relationship. They have a direction, which can be outgoing (->) or incoming (<-), indicating the direction of traversal. In general, relationships are used for traversing the graph, allowing for efficient navigation and querying of connected data [VB14, Neo23].

One of the key features of Neo4j is its query language called Cypher. It is a pattern-based query language that allows for flexible and efficient querying of the graph. Besides that, it allows to specify patterns of nodes and relationships to retrieve the desired data from the graph. Moreover, Cypher provides powerful querying capabilities, including filtering, sorting, aggregation, and traversal, making it a robust tool for working with graph.

The following Cypher query retrieves data, involving two nodes labeled as ‘Entity’ that are connected by a relationship labeled as ‘HAS\_RELATIONSHIP’. The query uses the MATCH clause to specify the pattern of nodes and relationships to be matched in the graph. It then uses the RETURN clause to specify data to be returned.

Listing 2.1: Example code block in Cypher.

```
1 MATCH (e1:Entity)-[r:HAS_RELATIONSHIP]-(e2:Entity)
2 WHERE e1.age > 1 AND e2.location = 'here'
3 RETURN e1.name AS e1_name, e1.age AS e1_age, e1.location AS e1_location,
4         r.weight AS relationship_weight,
5         e2.name AS e2_name, e2.age AS e2_age, e2.location AS e2_location
```

The `(e1:Entity)` and `(e2:Entity)` are node patterns labeled as ‘Entity’, representing the two nodes in the graph. The `(e1)` and `(e2)` are variables used to reference these nodes. The `[r:HAS_RELATIONSHIP]` is a relationship pattern labeled as ‘HAS\_RELATIONSHIP’ between the two ‘Entity’ nodes, with `r` as a variable to reference the relationship. The WHERE clause in the Cypher query is used to filter the results of the graph pattern matching in the MATCH clause based on specified conditions. Only nodes that satisfy these conditions will be included in the result. The RETURN clause specifies the data to be returned. It includes attributes of both ‘Entity’ nodes such as name, age, and location, retrieved using `e1.name`, `e1.age`, `e1.location`, `e2.name`, `e2.age`, and `e2.location` respectively. It also includes the ‘weight’ attribute of the ‘HAS\_RELATIONSHIP’ relationship, retrieved using `r.weight`.

Performance is a crucial aspect of querying graph databases with Cypher. Cypher queries are designed to efficiently traverse the graph and retrieve the desired data based on patterns of nodes and relationships. To optimize performance, it’s important to consider various factors such as the size and structure of the graph, the complexity of the query, and the indexing and caching mechanisms in place. Properly leveraging indexes, avoiding unnecessary traversals, and using appropriate filtering and aggregation techniques can significantly improve query performance. Additionally, understanding how Cypher queries are translated into graph database operations and the underlying data storage model can help in writing efficient queries. Regular performance monitoring, profiling, and tuning can further enhance the performance of Cypher queries, enabling faster and more effective querying of graph databases [VB14].

## 2.4.2 Operational Semantics

Neo4j's operational semantics specify how queries are evaluated on a graph data model, where data is represented as nodes and relationships. Queries in Cypher are executed by traversing the graph and applying operations to nodes and relationships. In this chapter, several operational semantics are presented, which will at least be named in the course of this Thesis [Neo23].

### Pattern Matching

Cypher queries express patterns that are matched against the graph data model. Patterns consist of nodes, a relationship, and properties, and are used to specify the structure of the desired results. For example, `(n:Person) -[:FRIENDS_WITH]->(m:Person)` matches two Person nodes that are connected by a FRIENDS\_WITH relationship.

### Traversal

Traversal is the process of following relationships in the graph to navigate from one node to another. Cypher queries use various traversal techniques, such as depth-first search or breadth-first search, to explore the graph and retrieve data. In the Cypher query `MATCH (n) -[:FRIENDS_WITH]->(m) RETURN *`, traversal is employed to retrieve pairs of connected nodes that are linked by the FRIENDS\_WITH relationship.

### Operations

Cypher queries allow for operations on nodes and relationships to manipulate data, such as filtering, aggregation, and sorting. For example, in the query `MATCH (n:Person) WHERE n.age > 18 RETURN n.name`, the condition `n.age > 18` filters Person nodes based on their age property, and the query returns their name property.

### Indexing

Neo4j allows for indexing of nodes and relationships to optimize query performance. Indexes can be created on properties, labels, or relationship types to speed up data retrieval and filtering.

### Transaction Management

Neo4j supports ACID (Atomicity, Consistency, Isolation, Durability) transactions to maintain data integrity and consistency. Transactions can be used to group multiple operations into a single atomic unit of work and ensure that the graph database remains in a consistent state.

### Graph Visualization

Neo4j provides graph visualization tools that allow for visual exploration and analysis of the graph data. These tools provide visual representations of the graph structure and can be used for data exploration, analysis, and debugging query results.

### Data Modeling and Schema Enforcement

Neo4j allows for flexible data modeling, where nodes and relationships can have arbitrary properties and labels. However, it also supports schema enforcement, allowing users to define and enforce constraints on the structure and properties of the graph data, ensuring data consistency and integrity.

### APOC

APOC (Awesome Procedures On Cypher) is a powerful plugin for Neo4j that extends the operational semantics of the Cypher query language. It provides a wide range of additional procedures and functions that enable advanced operations on graph data. These include data transformation, data import/export, graph algorithms, spatial operations, and more. APOC adds rich operational semantics to Neo4j, allowing for sophisticated data manipulation, transformation, and analysis on graph data beyond the built-in capabilities of Cypher.

## 3 State of the art: Data-oriented paradigm

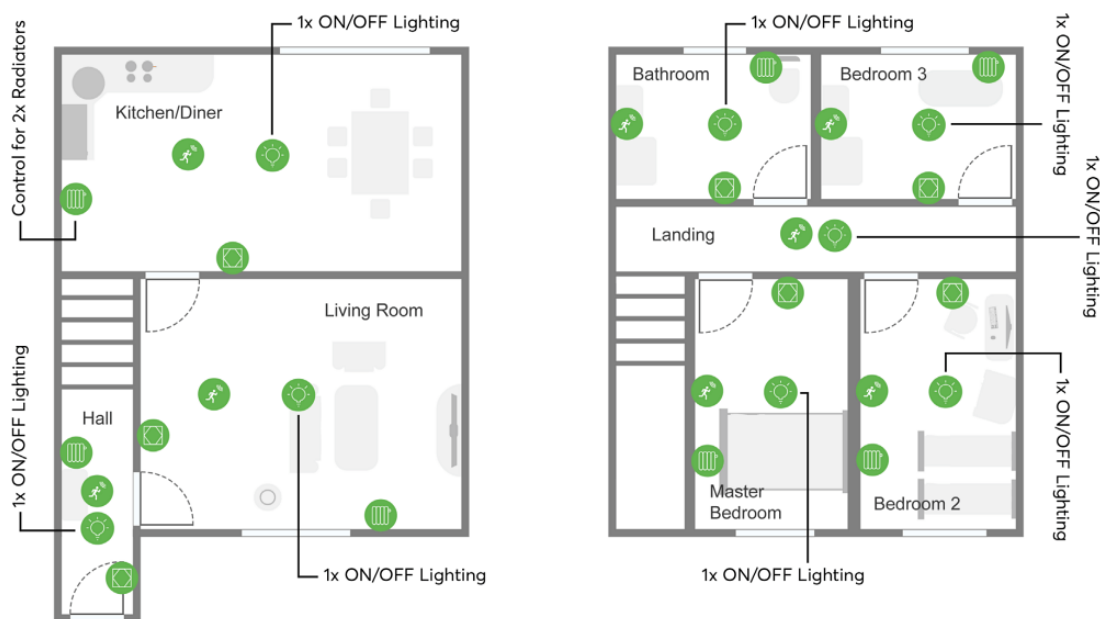
This chapter serves as a critical examination of the existing body of knowledge related to data-oriented programming. It delves into the fundamental concepts of it, providing a comprehensive overview of its principles, various usage contexts, and differences compared to object-oriented programming. The core principles of data-oriented programming, including efficient data representation and manipulation, performance optimization, and scalability, will be explored. Additionally, this chapter conducts a review of the application of data-oriented paradigms in the field of web and game development, highlighting the significance of data driven design approaches in analyzing and interpreting complex data structures. Through an in-depth exploration of literature, this chapter aims to provide readers with a foundation in understanding the key concepts of data-oriented programming and data-oriented architectures.

### 3.1 Data-oriented Programming

Data-oriented design has been around for decades, but was officially named by Noel Llopis in his 2009 article ‘High-performance programming with data-oriented design’ [Llo11, Fab18]. The concept focuses on understanding the data and how to transform it given knowledge of the target hardware. This approach to software development is becoming increasingly important in the age of multi-core machines and parallel processing. Unlike object-oriented design, which combines code and data, data-oriented design encourages the separation of code from data. It takes cues from the data that is seen or expected, avoids assuming anything about the problem domain, and allows for simpler coupling and decoupling of pieces of data and transformations [Fab18].

Data-oriented programming is a related programming paradigm that provides guidelines for representing and manipulating data at the center of software systems, such as frontend or backend web applications and web services. It provides guidelines for how to work with data. In this programming paradigm, questions about both the problem and data used for the problem need to be asked before attempting a solution. This is illustrated in the example of the light switch problem, which addresses the representation of data for the action of turning on a light. By asking questions about the problem and necessary data, data-oriented programming aims to avoid assuming

anything about the problem domain and instead takes cues from the data that is seen or expected [Sha22, Bay22]. In Figure 3.1, a smarthome with several devices is depicted that enables users to manage multiple lights through voice commands. However, for individuals who wish to turn on or off a single light in this smarthome, the control method may not be immediately apparent without prior knowledge of the light’s name and control process. Moreover, the device becomes non-functional during periods of Internet outages. This serves as an example of an overly complex solution to a fundamental problem that necessitates additional data for a simple operation. This scenario is comparable to the development of software that attempts to abstract away from the main problem it aims to solve. While such software may address a complex set of issues, the original purpose of the software becomes obscured or difficult to access [Bay22].



**Figure 3.1:** Floorplan of a well-lit smart home [lox].

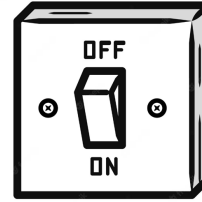
It is crucial not to introduce unnecessary complexity into software solutions, as it often leads to more complicated and error-prone code that requires extensive testing and debugging. However, even a seemingly simple problem such as turning on a light requires the representation of a single bit of data, which should be the focus of the solution. By removing all extraneous data, a data-oriented design solution can be achieved, as demonstrated in Figure 3.2, which represents the data and conversion of that data effectively. Additional considerations, such as safety measures, should also be carefully evaluated before being added to the solution. Furthermore, it is important to consider the usability of the solution, as illustrated by the example of the light switch, where the labeling of the on/off positions may vary by region. For example, in some countries, the top position of a light switch indicates ‘off’, while in others, it indicates ‘on’. Therefore,



it is important to carefully consider all relevant factors and provide the user with all the necessary information to perform the task, as demonstrated in Figure 3.3 [Bay22].



**Figure 3.2:** Simple light switch.



**Figure 3.3:** Labeled light switch.

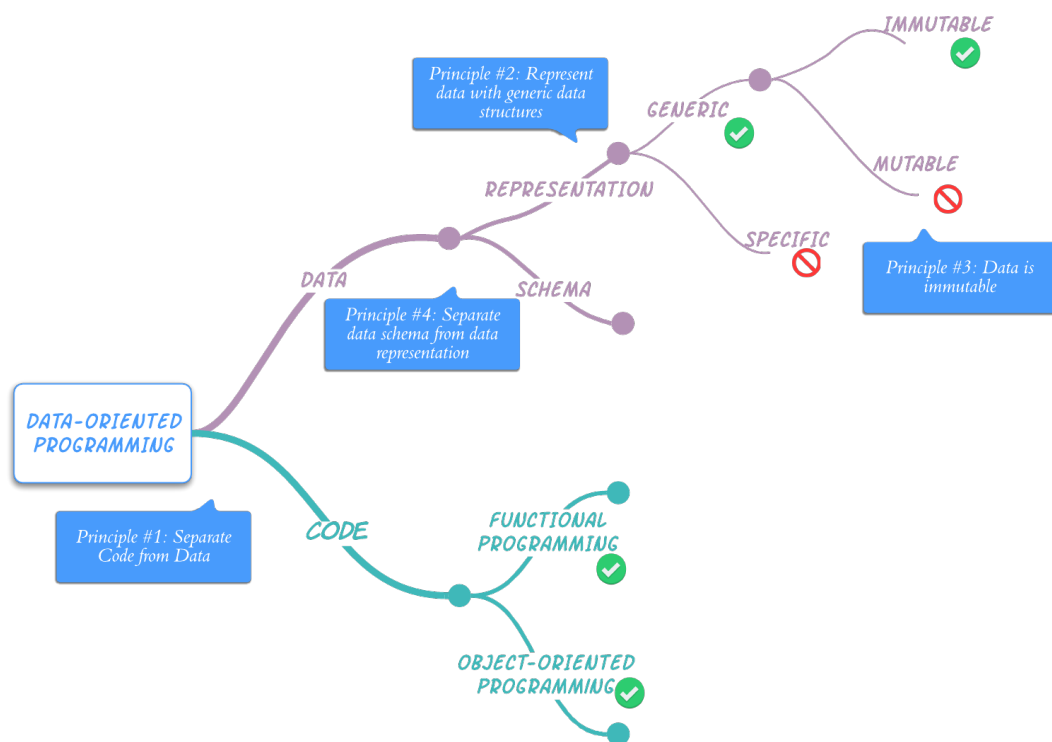
In the process of designing a software system in a data-oriented manner, it is essential to prioritize data over code. This means that the data is designed and reviewed regularly throughout the development process to ensure that it is structured in a way that is conducive to the desired outcome. In data-oriented design, data is seen as the primary component of the system, around which the rest of the software is built. This approach contrasts with traditional software development, which often focuses primarily on writing code and then fitting the data to fit that code [Bay22, Fab18].

Developing software using a data-oriented programming approach offers a more effective way to handle changes compared to object-oriented design. In object-oriented design, the problem domain and implementation are coupled, which can lead to inertia when designs change. However, the data-oriented approach recognizes changes in design by understanding changes in the data, allowing for easier code changes when the data source changes. Additionally, data-oriented design simplifies the coupling and decoupling of pieces of data and transforms, making it easier to refactor code than in object-oriented design. However, it is important to keep track of the data requirements for each operation and be aware of the potential danger of desynchronization. In some cases, using a mix of data-oriented and object-oriented approaches may be beneficial [Fab18].

Although data-oriented design has several advantages, there is a common misconception that it can be achieved through a static library or set of templates for all problems. This approach is not practical because real-world data is not generic and often contains unique values and patterns that cannot be captured by generic solutions. While using existing templates may enhance readability and reduce bugs, it can make the code harder to read if the solution is not well-matched to the existing generic approach. It is often preferable to hard code a new algorithm that has sufficient testing, focusing only on the facts about the data and simple data rather than stateful objects [Fab18, Bay22].

## 3.1.1 Principles of data-oriented programming

Data-oriented programming is a programming paradigm focused on data manipulation at the center of software systems, such as web applications and web services. Unlike object-oriented programming, data-oriented programming separates code from data and concentrates on data understanding and manipulation. It requires questioning the problem and data to achieve an optimal solution. While the term ‘data-oriented programming’ is commonly used, it is worth noting that the paradigm has been referred to by different names by various authors over the past decade. Despite this variation in naming, the core principles of data-oriented programming remain the same.



**Figure 3.4:** Principles of data-oriented programming: An Overview [Sha22].

This programming paradigm is guided by fundamental principles that emphasize the separation of code and data, the use of generic data structures, and the immutability of data, as shown in the Figure 3.4. These principles are not specific to any programming language and can be implemented in various languages including object-oriented programming languages such as Java, C#, and C++, as well as functional programming languages like Clojure, OCaml, and Haskell, and hybrid languages such as JavaScript, Python, Ruby, and Scala. This versatility provides developers with the opportunity to apply data-oriented programming principles across multiple languages, making them a valuable tool for software development [Jos07].

**Principle #1: Separate code from data**

The first principle of data-oriented programming is a fundamental design principle that advocates for the separation of code and data. This principle is not limited to a particular programming paradigm, and it is possible to adhere to or violate it in both functional programming and object-oriented programming contexts.

In object-oriented programming, the principle can be implemented by consolidating code as methods within a static class. Conversely, in functional programming, the principle can be violated by obscuring state within the lexical scope of a function [Sha22].

Listing 3.1: An example of object-oriented programming following this principle.

```
1 class Player {
2   constructor(playerName, discriminator, games) {
3     this.playerName = playerName;
4     this.discriminator = discriminator;
5     this.games = games;
6   }
7 }
8
9 class UserData {
10  static username(data) {
11    return data.playerName + "#" + data.discriminator;
12  }
13 }
14
15 const data = new Player("Sebb", "1337", 420);
16 UserData.username(data); // "Sebb#1337"
```

Compliance with this principle can be accomplished in classes by ensuring that the code consists of static methods and that the data is encapsulated within data classes, which function as containers of data.

Listing 3.2: An example of functional programming following this principle.

```
1 function createPlayerData(playerName, discriminator, games) {
2   return {
3     playerName: playerName,
4     discriminator: discriminator,
5     games: games
6   };
7 }
8
9 function username(data) {
10  return data.playerName + "#" + data.discriminator;
11 }
```

One of the benefits of adhering to the first principle is the ability to reuse code in different contexts. Consider a scenario where there are two entities: players and spectators. Although the player and spectator entities are distinct, they share two common data fields, namely `playerName` and `discriminator`. Additionally, the logic for computing the username of players and spectators is the same, as it involves retrieving the values of these two fields. However, in traditional object-oriented programming the code for computing username is locked inside the `Player` class, making it challenging to reuse the code for spectators. By separating code and data, this issue can be resolved, enabling the code to be reused effortlessly in different contexts [Sha22, Fab18].

Furthermore, the separation of code and data allows for an isolated testing of code. Without separation of code and data, objects must be instantiated to test their methods. For instance, testing the `username` code inside the `createPlayer` function requires a player object instantiation, as demonstrated in the Listing 3.3. By separating code and data, testing can be carried out without the need for objects, which enhances the reliability of the testing process and simplifies it.

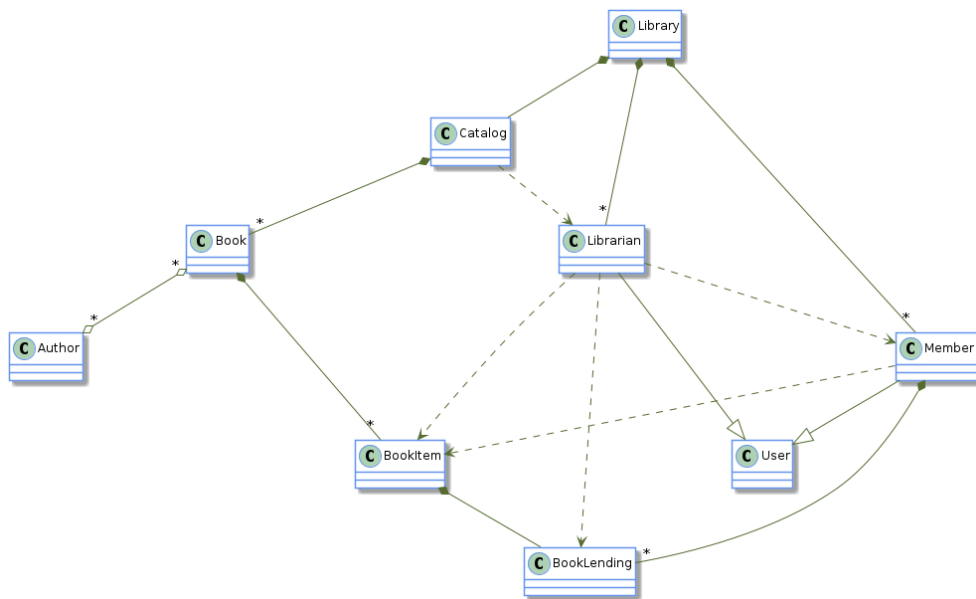
Listing 3.3: An example of decoupling for isolated testing.

```
1 // Instantiating a player object
2 const player = createPlayer("Lux", "0042", 420);
3 player.username() === "Lux#0042"
4
5 // Decoupling data code from object
6 const data = {
7   playerName: "Lux",
8   discriminator: "0042"
9   games: 420
10 };
11 UserData.username(data) === "Lux#0042";
```

---

Finally, applying the first principle results in less complex systems. This benefit refers to the type of complexity that makes systems challenging to understand, as defined in the ‘Out of the Tar Pit’ paper [Mos06]. It is unrelated to the complexity of resources used by a program [Sha22]. Separating code and data simplifies the system structure and reduces dependencies, making it more comprehensible and less intricate. Though this benefit may be challenging to comprehend, it is critical in developing software systems that are scalable and maintainable. As they grow in size, the number of lines of code increases exponentially, leading to an increase in system complexity. Object-oriented programming suffers from two types of complexity: state-derived complexity, which occurs due to the use of mutable state and side effects, and control-derived complexity, which arises from the use of control structures such as loops and conditionals [Mos06]. Functional programming, on the other hand, has the potential to avoid state-derived complexity by relying on immutable data structures and avoiding side effects [Mos06].

The complexity of a software system can be influenced by how its code and data are structured [Sha22]. In contrast to a system where code and data were intertwined as in object-oriented and functional programming, data-oriented programming separates the two, making it easier to comprehend. This approach enables developers to understand the code and data components independently, rather than having to consider them together as a single entity. By breaking the system down into distinct parts, the overall complexity is reduced. Disconnected parts are less complex than a single, monolithic entity, allowing for better comprehension and ease of maintenance.



**Figure 3.5:** Class diagram of a fictitious library management tool [Sha22].

This insight is exemplified in the class diagram of a hypothetical library management system [Sha22], where code and data are intertwined as shown in Figure 3.5. The complexity of the system is evident, regardless of one’s familiarity with the specific classes involved, due to the numerous dependencies between system entities. Of these entities, the Librarian entity is the most complex, as it is connected to six other entities through a combination of code and data relations. However, in this design, the Librarian entity combines both code and data, and is thus involved in both types of relations. To reduce the complexity of the system, it is possible to divide each entity into separate code and data entities without any other modifications to the system. This would allow the code and data components to be understood independently, and potentially simplify the overall system design [Sha22, Mos06].

Adopting the first principle provides several benefits, including the ability to reuse code in diverse contexts, test code in isolation, and reduce system complexity. These benefits enhance the reliability, maintainability, and scalability of software systems, making them easier to comprehend and modify.

#### Principle #2: Use generic data structures

In data-oriented programming, generic data structures such as maps and arrays are frequently employed to represent data in the domain. They are capable of effectively representing the majority of data entities that are encountered in typical applications. Nevertheless, certain scenarios may call for the use of other generic data structures such as sets, lists, and queues [Sha22].

To comply with this principle, utilizing a specific class to represent entities should be avoided. A more suitable alternative is to utilize a generic data structure, such as a map, a dictionary or an associative array, to represent different attributes of an entity. For instance, these can consist of several attributes, including `playerName`, `discriminator`, and `games`, among others, which collectively offer an overview of a given entity such as in Listing 3.4.

Listing 3.4: Using maps to create generic structures for players.

```
1 function createPlayer(playerName, discriminator, games) {
2   const data = new Map();
3
4   data.playerName = playerName;
5   data.discriminator = discriminator;
6   data.games = games;
7
8   return data;
9 }
```

In general, using generic data structures to represent data offers several benefits. These include the ability to use generic functions that are not limited to a specific use case and a flexible data model that can represent a wide variety of data entities. However, there are disadvantages associated with implementing this principle. One such is the slight performance hit incurred due to the use of generic data structures instead of class related structures [Sha22]. When a program instantiates data using specific classes, accessing the value of a class member becomes faster due to the compiler's ability to perform optimization based on its knowledge of the data structure. This leads to improved program efficiency and execution speed [Jos07].

Another disadvantage is that no data schema is required, which can lead to potential errors if the data is not properly structured. Besides that, no compile-time check is performed to ensure that the data is valid, and in some statically typed languages, explicit type casting may be necessary. Despite these costs, the advantages of using generic data structures typically outweigh the drawbacks, making it a valuable tool for effective data representation in data-oriented programming [Jos07].

### Principle #3: Maintaining immutability of data

Data-oriented programming is particularly stringent when it comes to modifications to data stored independently from the code and organized with generic data structures. Therefore, mutations to data are not allowed, and changes are made by creating new versions of the data. Although the reference to a variable may be updated to point to a new version of the data, the actual value of the data must not change [Sha22]. By avoiding mutable data structures and instead using immutable data structures, data can be safely shared across different parts of a program or architecture, which can lead to better concurrency and fewer bugs [Jos07].

Creating new versions of data instead of mutating existing ones can be done by cloning the original data before modifying it, but this approach can be inefficient, especially with large data sets. To embrace immutability in an efficient way, many programming languages provide libraries that offer efficient implementations of persistent data structures, such as Clojure's persistent data structures [Sha22].

Listing 3.5: Improving data immutability with spread operations in JavaScript.

```
1 // Original player object
2 const player = {
3   playerName: "Bagni",
4   discriminator: "2023",
5   games: 10
6 };
7
8 // Update the player's game stats
9 const updatedPlayer = { ...player, games: player.games + 1 };
10
11 UserData.toString(player) // "Bagni#2023, Games: 10"
12 UserData.toString(updatedPlayer) // "Bagni#2023, Games: 11"
```

When data is mutable, passing it as a function argument can lead to unintended data modification. For instance, JavaScript passes objects by reference, meaning that the function modifies the original object. Therefore, especially in the case of asynchronous JavaScript code, the behavior of the code becomes unpredictable. In such cases, the value of a particular data object can change before its usage, leading to incorrect outputs or even crashes. These problems are also applicable in other programming languages.

Moreover, in web frameworks like React, if an app doesn't follow data immutability, it requires checking every nested part of the UI data. However, if it adheres to data immutability, data comparison can be optimized for scenarios where data isn't modified. This is because if the object address remains the same, it can be concluded that the data hasn't changed, and no further checking is required [Sha22].

#### Principle #4: Distinguish schema from representation

To organize data into immutable and specific structures, we need a way to describe how the data is arranged. This is achieved through the use of ‘data schemes’, which outline the structure of the data and is kept separate from the actual data. One of the key benefits of this approach is that it allows developers to choose which portions of data require a schema and which ones do not [Sha22, Jos07]. As an example to illustrate this approach, a request to add a Player to a system using GraphQL will be demonstrated. Due to the versatility and clarity provided by GraphQL schema, it is a useful tool for defining the expected structure of data as already mentioned in chapter 2.3.

In the event of a request to append a Player to a system, essential information such as their `username`, `discriminator`, and the number of `games` played may be included. In data-oriented programming, request data is represented using generic data structures, such as a dictionary, with the already mentioned fields being anticipated. To describe the data structure, a distinct data schema is established, which defines the projected configuration of the data. The following Listing 3.6 shows an example of a GraphQL schema that could be used to represent the structure.

Listing 3.6: Example implementation of a GraphQL schema.

```
1 type Player {
2   username: String!
3   discriminator: String!
4   games: Int
5 }
6
7 input PlayerInput {
8   username: String!
9   discriminator: String!
10  games: Int
11 }
```

This schema outlines the expected shape of the request data, comprising the mandatory `username` and `discriminator` fields and the optional `games` field. By utilizing a data schema to define the format of data, data-oriented programming empowers developers to determine which components of data necessitate a schema and which not. Furthermore, separating the data schema from data representation in programming offers several advantages, including the flexibility to choose which data requires validation, optional fields, sophisticated data validation conditions, and the possibility of automatic generation of data model visualization [Sha22]. However, the separation of data schema and representation in programming places the responsibility of implementing data validation entirely on developers, who must decide where it is necessary, which may result in a weaker link between data and its schema [Jos07].



### 3.1.2 Differences between object and data-oriented programming

In contrast to object-oriented programming, the data-oriented programming approach focuses on understanding and manipulating the data to achieve the desired output. By treating data as mere facts that can be interpreted as needed, the likelihood of tangling the facts with their contexts and mixing unrelated data is reduced. This approach eliminates the potential limitations of data reuse and the introduction of unrelated data into the same class, which is often seen in object-oriented design. The focus remains on the data transformation and its final destination context [Fab18].

<b>Data-oriented programming</b>	<b>Object-oriented programming</b>
Exposes data	Hides data by encapsulating
Hides the code	Exposes methods - code
Separates data and code	Intermix data and code
Strict separation of parser, validator, transformer and logic	Combined processing, no restrictions
Generic data structures, such as dictionaries and maps, to represent entities	Classes represent entities
Loosely-coupled	Tightly-coupled

**Table 3.1:** A comparison between principles of programming paradigms [Jos07]

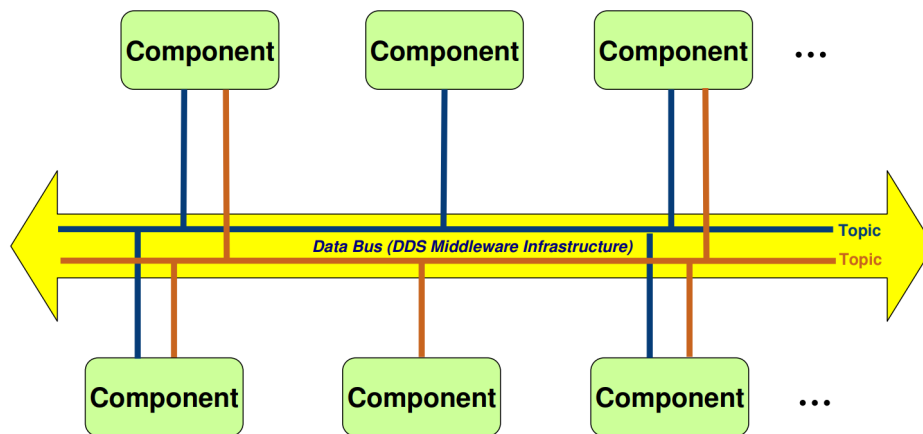
Both programming paradigms are distinct methodologies that address disparate issues. Object-oriented programming fosters strong interdependence between modules, whereas data-oriented programming encourages independence. The choice of programming paradigm depends on the nature of the problem at hand. Object-oriented programming is well-suited for applications that involve complex and dynamic systems, such as video games or graphical user interfaces. In these systems, objects can represent real-world entities and interact with each other to produce the desired behavior. On the other hand, data-oriented programming is better suited for systems that require high performance and efficiency, such as scientific simulations or data analysis. In these systems, the focus is on the manipulation of data sets and the optimization of algorithms that operate on them [Jos07, Fab18]. However, game development is increasingly adopting data-oriented programming techniques to improve performance and scalability. This shift is driven by the growing complexity of modern game engines, which require the processing of large amounts of data in real-time. Game engines often involve data sets that require frequent updates, such as positions and velocities of objects. By optimizing data for efficient access and manipulation, data-oriented programming can improve performance and reduce overhead compared to traditional techniques [Bay22].

## 3.2 Data-oriented Architecture

Data-oriented programming addresses two important issues in system design - incremental and independent development, and impedance mismatch. Incremental and independent development involves developing an application in small modules that can be independently tested, integrated, and deployed. This approach allows developers to quickly identify and fix issues while reducing the likelihood of introducing new bugs. Impedance mismatch refers to the mismatch between the design of the database and the design of the application code, which can lead to performance issues and errors. Therefore, it is a suitable solution for integration challenges in a scalable and agile manner. However, it is important to understand that the use of a design paradigm does not guarantee well-designed systems, as with object-oriented paradigm, data-oriented programming can also be misused. It provides guidance for building loosely-coupled systems, but the design process is ultimately a human activity. Paradigms and tools can only assist in the process [Jos07].

In software engineering, the architecture of an application or an ecosystem consisting of various components is a crucial aspect that can significantly affect its success. It should be noted that designing the architecture of an application differs significantly from that of an entire ecosystem. Various paradigms and methodologies can be employed to achieve optimal results. Data-oriented programming is a paradigm that is commonly used at the application architecture layer. Data-oriented design, on the other hand, can be utilized to plan out an entire architecture consisting of loosely-coupled components that can take advantage of the data-oriented programming paradigm [Fab18, Jos07].

A data-oriented architecture is a design approach that emphasizes the development of loosely coupled applications. It allows for the independent addition and removal of components without requiring knowledge of other components. The architecture demonstration shown in Figure 3.6 facilitates the creation, usage, and deletion of data readers and data writers of topics independently by a component, without the need for centralized configuration or changes. A middleware infrastructure could automatically establish direct data paths between the data readers and data writers of a topic, which are also managed by a middleware infrastructure. Data writers are components that publish data to a topic, while data readers are components that subscribe to data from a topic. A topic is a named entity that serves as a communication channel between data writers and readers, enabling the exchange of data in a loosely coupled manner. This generic architecture principle can be specialized into various styles such as data flow architecture, event-driven architecture, and service-oriented architecture, which are commonly used in Edge and Enterprise systems [Jos07].



**Figure 3.6:** Data-oriented architecture for loosely coupled applications [Jos07].

Edge systems bring computing resources and application services closer to the end-users or devices at the network edge, reducing the distance data must travel between the device generating it and the data processing or storage facility. In contrast, Enterprise systems are integrated software applications that support complex business processes and enterprise-wide information management, comprising various functional modules such as finance, human resources, supply chain management, and customer management.

In web development, data writers are typically the client-side components of a web application that send data to the server through HTTP requests, while data readers are server-side components that receive the data and process it before sending a response back to the client. The topic can be thought of as a communication channel established between the client and server, enabled by technologies such as web socket or server-sent events, which allow for the real-time exchange of data in a loosely coupled manner. An example of data processing with loosely coupled components is the use of GraphQL as server-side query language for APIs, in conjunction with a server and a Neo4j database. In this architecture, the server is responsible for receiving and processing GraphQL queries from the client, which are then used to retrieve data from the Neo4j database. Each of these components is independent and can be replaced or updated without affecting the others. The server acts as an intermediary between the client and the database, providing a layer of abstraction and enabling more efficient data retrieval. The Neo4j database stores the data and responds to the queries sent by the server. Meanwhile, GraphQL provides a flexible and intuitive way for clients to specify the data they need from the database, allowing for a more efficient and tailored data retrieval process. This architecture provides a scalable and modular approach to data processing, where each component can be independently developed, tested, and maintained, resulting in a more robust and maintainable system.

### 3.3 Conclusion

In conclusion, data-oriented programming is a design paradigm that emphasizes the manipulation and transformation of data to achieve desired outcomes. By treating data as immutable, independent facts, data-oriented programming reduces the likelihood of tangling facts with their contexts and mixing unrelated data. In contrast to object-oriented programming, data-oriented programming focuses on the transformation and the final destination of data, rather than on the entities that generate it [Fab18, Sha22].

To organize data into immutable and specific structures, developers use data schemes, which outline the structure of the data and is kept separate from the actual data. This approach offers several advantages, including the flexibility to choose which data requires validation, optional fields, sophisticated data validation conditions, and the possibility of automatic generation of data model visualization [Sha22].

The architecture of an application or an ecosystem consisting of various components is a crucial aspect that can significantly affect its success. Data-oriented programming addresses two important issues in system design - incremental and independent development, and impedance mismatch. Incremental and independent development involves developing an application in small modules that can be independently tested, integrated, and deployed. Data-oriented architecture is a design approach that emphasizes the development of loosely coupled applications. It allows for the independent addition and removal of components without requiring knowledge of other components [Fab18, Jos07].

While data-oriented programming is commonly used in Edge and Enterprise systems, game development is increasingly adopting data-oriented programming techniques to improve performance and scalability. However, the use of a design paradigm does not guarantee well-designed systems, as with object-oriented paradigm, data-oriented programming can also be misused. It provides guidance for building loosely-coupled systems, but the design process is ultimately a human activity. Paradigms and tools can only assist in the process [Bay22, Jos07].

In essence, data-oriented programming is a powerful paradigm that empowers developers to efficiently manage data and create scalable and modular systems. Its focus on data transformation, loosely coupled components, and independent development make it an excellent choice for a wide range of applications. By using data schemes, developers can create efficient and flexible data structures that enhance the performance and maintainability of their applications. The growing popularity of data-oriented programming in game development and other areas underscores its value and effectiveness as a programming paradigm.

## 4 Concept Development

The concept development phase is a crucial part of the software engineering process. It is characterized by the exploration and refinement of ideas for a software project, which lays the foundation for the entire development cycle. This chapter delves into various aspects of concept development, including the use of use cases to identify user requirements for the scholarly digital edition, the gathering of data-oriented and project-specific requirements, and the development of architecture and application concepts. The result is a software concept that is optimized to meet the needs of the end-users while adhering to scientific principles and best practices in software engineering.

### 4.1 Use cases

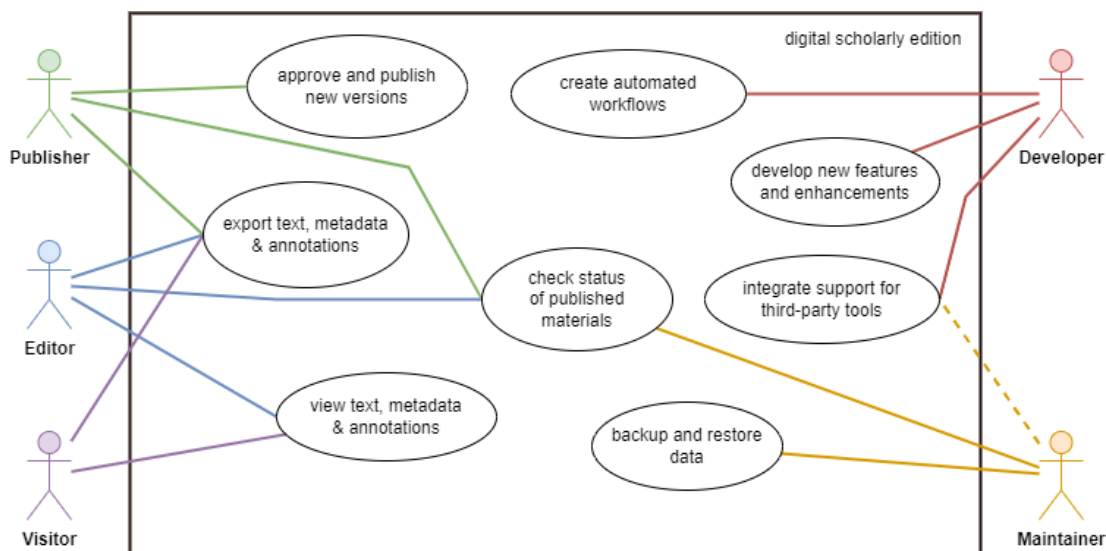
In software development, use cases are fundamental for the engineering of requirements. They provide a standardized approach to describing how users interact with a system or software application to achieve their objectives [Kul12]. Specifically, when developing software applications for scholarly digital editions in the Digital Humanities, they play a role in ensuring that the final product meets the needs of its users.

Use cases provide a structured approach to defining a goal-oriented set of interactions between external actors and the system under consideration. Actors in a use case can be users, roles, or other systems. The use case itself is initiated by a user with a specific goal. Successful completion of it is achieved when the goal is met. In general, they enable developers to capture and document the interactions between actors and the system without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, bounding the scope of the system. Additionally, they are written in an easy-to-understand narrative using domain-specific vocabulary, providing users with an engaging and validated process [Mal01].

The subsequent chapters will examine the role of use cases in software development for scholarly digital editions in the Digital Humanities on different layers. By analyzing them, it is possible to gain insights into user needs and goals, leading to the development of software that cater to the unique requirements of the Digital Humanities.

The following use cases have been developed using Derek Coleman’s proposed standard use case template, which is completed and available in the appendix [A]. They serve as functional requirements that are necessary to develop the software architecture. Although UML does not provide a specific template for writing them, the Coleman template can be adapted with minor adjustments to produce clear and concise scenarios. This approach has been shown to be effective in creating well-structured and easily understandable use cases [Mal01]. Moreover, it is important to note here that the following scenarios were created in collaboration with experts in the Digital Humanities, through weekly meetings. These cases primarily relate to ongoing or upcoming projects in this field. While there may be other use cases that could be relevant, their importance lies in their ability to integrate seamlessly with the existing software. Therefore, any additional requirements that can be easily linked to the current software would strengthen the development concept, which is going to be mapped out in this chapter.

#### 4.1.1 Architecture layer



**Figure 4.1:** Project-oriented needs for an architecture layer: use case diagram.

At the architectural level, a data-oriented approach can be used to fulfill coarse functional requirements that stem from either the work at hand or the requirements of scholarly digital editions in the Digital Humanities. This design approach emphasizes the development of loosely coupled applications and allows for the independent addition and removal of components without requiring knowledge of other components. The architecture is essentially composed of layers of easily replaceable and loosely coupled components, as described in the preceding chapters. In the following, use cases of the architecture will be defined and explained. These use cases provide functional requirements for the architecture and its components, which will be collected in the subsequent chapters together with requirements from other areas. This process ensures

that the architecture is not only designed to meet the specific needs of its intended application domain but also takes into account broader requirements and best practices from related fields. Such an approach enables the development of a more robust and adaptable architecture that can evolve and meet changing demands over time.

A scholarly digital edition involves a variety of actors who play different roles in the creation, management, and distribution of digital texts. The Editor is responsible for creating and editing texts, while the Publisher manages the publication and distribution of the edition. Visitors use the digital edition for research and study purposes, while Maintainers maintain the technical infrastructure of the system. Finally, Developers are responsible for developing and maintaining the software that powers the digital edition. Each actor plays a critical role in ensuring the success of the scholarly digital edition, working together to provide a valuable resource for scholars and researchers in the field.

#### Approve and publish new versions

This use case involves approving and publishing new versions of a scholarly digital edition. The publisher is responsible for reviewing the new version and approving it if it meets quality standards, after which it can be published. If the new version falls short of the standards, the publisher sends it back to the editor for revisions. In addition to meeting functional requirements such as reliability, security, and usability, the use case addresses the challenges of a complex and time-consuming publishing process.

To ensure a smooth workflow for approving and publishing new versions, it is important to consider a set of requirements. These may include providing the publisher with access to the latest version of the edition, enabling them to review and approve new versions that meet quality standards, ensuring secure publication to prevent unauthorized access, providing a user-friendly interface or process for approving and publishing new versions, and allowing the publisher to send new versions back to the editor for revisions if they fail to meet quality standards [A.1].

#### Check status of published materials

In the process of checking the status of published materials in a scholarly digital edition, actors such as the publisher, editor, and maintainer need to be able to view the status of published materials and take necessary actions based on their respective roles. The non-functional requirements of this use case include displaying the status of published materials accurately and reliably, and providing a simple and user-friendly process for all actors involved. However, the use case acknowledges that different projects in the Digital Humanities may use different status codes, which could potentially pose issues.

For a seamless and streamlined process for verifying the status of published materials, several requirements should be considered. These include providing the publisher, editor, and maintainer with access to the latest version of the digital edition, enabling actors to easily locate the section displaying the status of published materials, allowing actors to review and take necessary actions based on their role and the status of published materials. The use case should also accommodate different types of status codes for different projects, allow the publisher and editor to mark published materials with different statuses, and accurately and reliably display the status of published materials. Additionally, it should offer a simple and user-friendly process for all actors involved to check the status of published materials, and enable them to view a collection of published materials along with their respective statuses [A.2].

### Export text, metadata and annotations

This use case requires the exporting of text, metadata, and annotations from a scholarly digital edition. Actors, including the publisher, editor, and visitor, should have the ability to select and export the desired materials in a specific format for scientific work, research, personal reference, or other purposes. To ensure a seamless experience, non-functional requirements for this use case include completing the export process in a reasonable amount of time and providing a simple and user-friendly process for all actors involved. However, the use case also acknowledges that a lack of consistency or existence in versioning may pose potential issues.

Several requirements must be taken into account to establish an efficient process for exporting materials from a scholarly digital edition. These may include enabling actors to easily locate the section of the digital edition that contains the desired materials, allowing them to select the text, metadata, and annotations they wish to export, and facilitating the export of the selected materials in the desired format. The use case should also cater to exporting materials for scientific work, research, personal reference, or other purposes, enable visitors to export metadata to aid with citation and referencing, and provide a straightforward and user-friendly process for all actors involved in exporting materials. It should complete the export process in a reasonable amount of time, maintain consistency in versioning to ensure that the exported materials match at least the state of the digital edition, and support exporting materials in various formats to meet visitors' requirements [A.3].



### Create automated workflows

Automated workflows can be created within the digital edition architecture by developers with appropriate access, permissions, tools, and basic knowledge. Developers navigate to the designated section of the architecture, create or modify workflows based on requirements, and test them for proper functioning. Non-functional requirements include scalability and maintainability of the workflow creation process, although varying workflows for different projects may pose challenges.

To enable the creation of automated workflows, several requirements should be considered, such as providing developers with necessary resources and access to the architecture. The use case should also support easy navigation to the relevant section of the architecture, enable creation and modification of workflows, and allow for testing of functionality. Scalability and maintainability of the workflow creation process should be prioritized, along with providing documentation and resources to aid developers in creating and maintaining workflows. Lastly, the architecture should accommodate different workflows for different projects [A.5].

### Develop new features and enhancements

This use case describes the process of developing new features and enhancements in the architecture and application layers of a scholarly digital edition. The process involves identifying the required feature or enhancement, creating a design document, coding the feature or enhancement, testing it for correct functioning, and deploying it to the digital edition architecture as a component. Scalability, maintainability, and compatibility of the developed components are important non-functional requirements. However, usually specific knowledge are necessary for different parts of a project.

In order to enable the development of new features and enhancements, several requirements should be considered. These include providing the developer with access to the necessary layers, permissions, and tools, supporting a design process that enables the developer to create a design document describing requirements, architecture, and implementation plan. The use case should allow the developer to code the feature or enhancement in adherence to the design document, test it for correct functioning, and support scalability to accommodate an increase in the number or complexity of features. The system should be easy to maintain and update for long-term maintainability, ensure compatibility with existing software and architecture, and provide documentation and resources to support the developer in the development process [A.6].

### Integrate support for third-party tools

The process of integrating third-party tools into the architecture and application layers of a scholarly digital edition involves the developer evaluating the tool, integrating it into the digital edition, testing the integration, and deploying the tool. The integration process should be scalable, easy to maintain and update, and compatible with existing software and architecture. However, tightly-coupled components in different projects may pose challenges, due to their monolithic behavior.

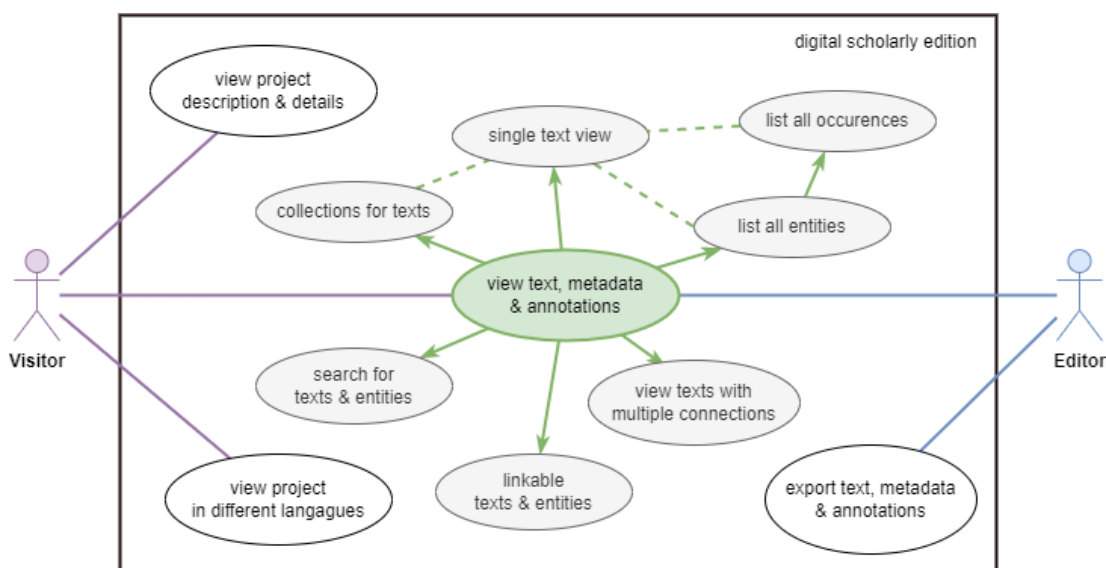
To ensure successful integration of third-party tools and web services into the digital edition, several requirements should be considered. These include ease of integration for developers, modularity of the integration process, a standardized interface, maintenance and update capabilities, and effective error handling. Additionally, the integration process should offer flexibility and be well-documented, allowing for the addition and removal of third-party tools without disrupting the system. By meeting these requirements, the integration process can be made more effective, beneficial for developers, and ultimately, for the digital edition architecture as a whole [A.7].

### Backup and restore data

A reliable backup and restore system is essential for the digital edition to prevent data loss due to system failure or corruption. The maintainer should initiate the process at regular intervals or as needed, with the backup system creating a secure copy of the data stored in a safe location. The maintainer should test the backup to ensure it can be successfully restored if necessary. The restore system should retrieve the backup data and restore it to the digital edition in case of data loss.

Several non-functional requirements should be considered to make the backup and restore system effective, including reliability and performance. It is crucial to ensure that the backup process takes place at regular intervals or as needed and that the backup data is encrypted and securely stored. Additionally, the backup should be tested to ensure that it can be successfully restored if required. The restore system must be capable of retrieving backup data and restoring it to the digital edition in case of data loss. To future-proof the backup and restore system, it should be designed to be flexible and scalable to accommodate potential changes or developments [A.8].

## 4.1.2 Application layer



**Figure 4.2:** Project-oriented needs for an application layer: use case diagram.

A software architecture consists of multiple components, including client and server applications in a web-based context. In a data-oriented approach at the application layer, developers focus on building flexible, loosely coupled components that can adapt to changing user needs and technological advancements. Data is the primary building block of the application, and each layer of the system is built on a set of well-defined and easily replaceable data components. Data-oriented programming concepts will be developed and implemented to ensure that the application offers several benefits, such as the ability to independently add or remove components without requiring knowledge of other components, greater flexibility, and agility in developing and maintaining complex systems. Use cases for this layer are determined by specific application domain needs, broader requirements, and best practices from related fields. By following this process, the architecture is designed to meet the specific needs of the intended user base while also taking into account considerations such as data integrity, availability, and scalability.

Visitors and Editors are two essential actors in the web-application of the scholarly digital edition. Visitors utilize the digital edition for research and study purposes, while Editors are responsible for reviewing created and edited texts. The client-server application plays a crucial role in enabling these actors to use the system efficiently. In the following sections, exemplary use cases will be provided to give an overview of how the application layer can support the needs of Visitors and Editors. These use case templates can be found in the appendix [A], and will provide detailed examples of how Visitors and Editors can use the system to achieve their goals.

### View text, metadata and annotations

This use case involves the process of accessing text, metadata, and annotations in the scholarly digital edition by visitors and editors. Both visitors and editors should have permission to view the latest version of the digital edition, along with its associated content. The viewing process may involve searching for a single text or different annotations, viewing all annotations in a text, viewing a collection containing all texts, and searching by text metadata. The viewing process should be simple, user-friendly, and easily accessible for visitors and editors. To ensure a seamless experience for visitors and editors, the digital edition should provide a user-friendly interface for accessing its content. The system should support various methods for viewing text and annotations, such as searching for a specific text or annotation, viewing all annotations in a text, viewing a collection containing all texts, and searching by text metadata. It should also be capable of handling recursive data relationships that may arise in the process.

By addressing these requirements, the viewing process can be made more efficient, user-friendly, and beneficial for the visitors and editors of the scholarly digital edition. The digital edition's interface should be designed to be accessible and easy to use, while also allowing for complex searches and the handling of recursive data relationships. This will enable visitors and editors to easily access and interact with the digital edition's content, regardless of their technical expertise level [A.4].

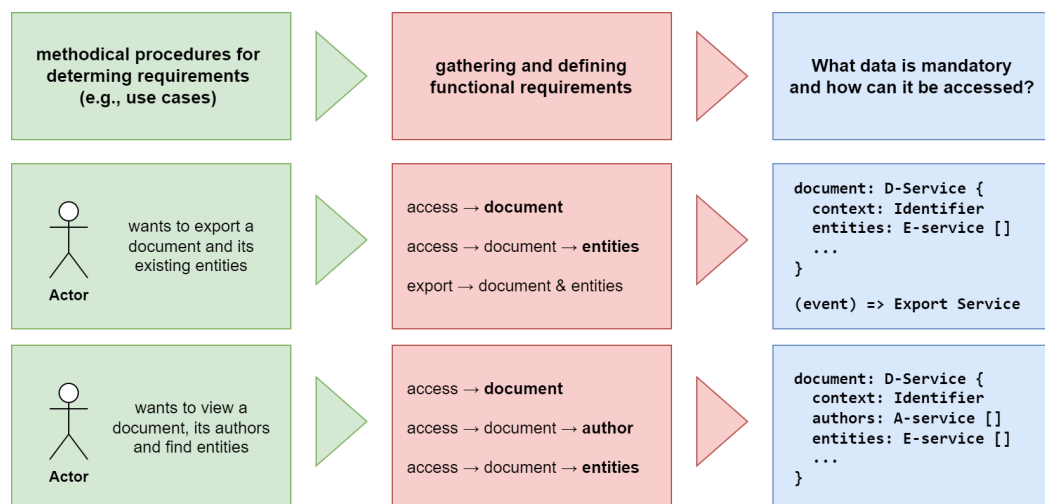
### View project description and details

This use case involves how visitors can access the project description and details of the scholarly digital edition. The visitors can learn about the project's scope, purpose, objectives, team members, schedule, and financial support sources through this process. To ensure that visitors can easily access project description and details, the digital edition should have a well-organized and easily accessible section that provides information about the project's scope, purpose, and objectives, as well as any other relevant information. This section should be user-friendly and easy to navigate, allowing visitors to quickly find the information they need.

Overall, this use case is straightforward and does not have any significant issues or variations. By providing easy access to project description and details, the digital edition can help visitors better understand the scope and purpose of the project. This information can help visitors make informed decisions about their use of the digital edition's content [A.9].

## 4.2 Requirement Analysis

Requirements play a critical role in the development of both the architecture of various applications and the architecture of an individual application. They can be defined as the set of functional and non-functional criteria that a system or application must meet to be considered acceptable for its intended purpose. By providing a clear understanding of what needs to be achieved, requirements help stakeholders and end-users determine whether the software meets their needs. For a data-driven architecture, various requirements can be identified based on methodical procedures for determining requirements such as use cases, which have been discussed in earlier sections. These requirements focus mainly on the architecture layer, where the system components are designed to meet functional and non-functional criteria. However, some requirements are also relevant to the applications themselves. Therefore, it is important to consider both the application and architecture-level requirements when designing data-oriented applications, such as digital editions.



**Figure 4.3:** Concept: Processing of data-oriented requirements.

Requirements analysis is a complex task that involves identifying the functionalities of a system while also considering data-oriented requirements. Generally, requirements with a data-oriented nature undergo a new concept of evaluation. This process entails collecting requirements, which can be accomplished through various methods such as use cases, and transforming them into Data Access Patterns, as depicted in Figure 4.3. In addition to access patterns, functional requirements must also be considered in the set of gathered information. For instance, if a user intends to export a document in a different format as a functional requirement, an access procedure must be performed before the document can be exported. Similarly, if the user wants to just access the document and examine the authors, which translates into simple Data Access Patterns.

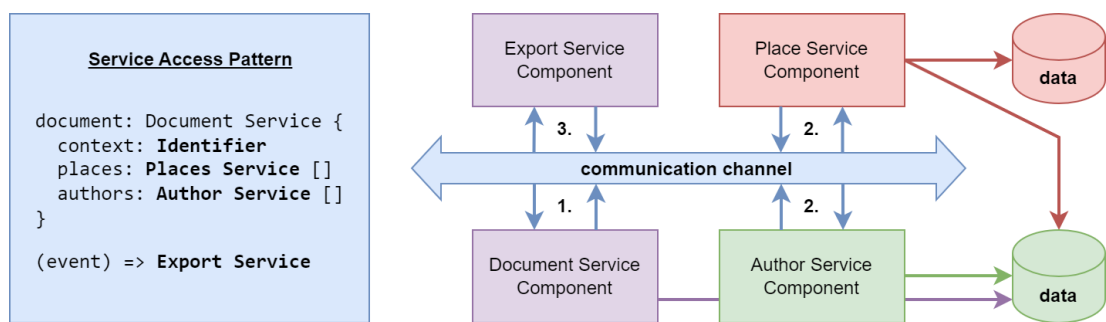
When focusing on the data itself, it becomes crucial to carefully consider which data to query and how to divide it, especially in a distributed architecture where various services and applications need to communicate through middlewares. Taking this approach allows for thoughtful planning of data set partitioning, access, and systematic processing prior to implementation. Moreover, this approach demonstrates the expansion of object data structures and the traversal of object attributes using Service Access Patterns and Data Access Objects. As explained in the following chapters, due to interconnected objects, it is indeed possible for recursive traversal to occur across multiple levels from one service to another. Overall, this new approach to requirements development not only facilitates the implementation of data-oriented applications but also promotes a data-driven perspective in distributed architectures.

In addition to the primary concept of developing requirements using data-oriented access patterns, it is crucial to recognize that software development encompasses other requirements arising from architecture patterns and the chosen programming paradigm, as discussed in Chapter 3.3. One important requirement in this regard is the establishment of a communication framework between loosely coupled components, especially in distributed architectures. Incorporating a middleware requirement can effectively fulfill this need. By incorporating middleware, a communication channel is created that facilitates the triggering of specific events using a topic and enables seamless data exchange between components. Furthermore, as described in subsequent chapters and illustrated in Figure 4.3, enabling the traversal of requests allows each attribute to trigger a service call while simultaneously traversing through the object. This distinction can be observed between Service Access Patterns, which require middleware communication and enable data extraction from other applications or direct processing, and Data Access Objects, which are used for traversing objects or types and directly communicating with a database, often within an application. Therefore, the principles of data-oriented programming discussed in Chapter 3.1.1 are also relevant in this context.

In summary, software engineering requirements analysis goes beyond simply identifying the necessary functionalities of a system. The data-oriented approach involves collecting requirements, translating them into Data Access Patterns and functional requirements, and addressing systematic issues that may arise before implementation. Adhering to data-oriented design rules and principles of data-oriented programming is essential. This approach empowers the implementation of complex architectures and applications from a data-oriented perspective, ensuring certain benefits for software development.

## 4.3 Architecture Concepts

To effectively implement a data-oriented architecture, it is important to carefully consider the requirements of the system and adhere to the principles of data-oriented design. One key aspect is ensuring that the communication channel or service is capable of handling the required Service Access Pattern as shown in Figure 4.4 and discussed earlier. This pattern should include both context and attributes, with the context usually identified by a unique identifier such as a UUID string. Additionally, functional requirements may be necessary to facilitate further data transformation, and these calls should be centered on the data. In the field of Digital Humanities, persistent identifiers are commonly used to uniquely identify all objects within the system. By following these best practices, a data-oriented architecture can be designed in an efficient manner.



**Figure 4.4:** Concept: Service Access Patterns.

The Service Access Pattern, as shown in Figure 4.4, not only identifies the context of services but also describes them based on attributes. Additionally, it can incorporate functional requirements as event-based methods to specify the output data format, transformation process, or other specific operations. For instance, in Figure 4.4, the service pattern interacts with the Document Service first (1.), followed by the concurrent retrieval of data for Places and Authors (2.). After all the data is available and processed, any dependencies that exist for the same types are traversed in the form of a graph, similar to the approach used in GraphQL. The final step is to process the event, which in this case involves exporting the data in a particular format (3.). The response format can be tailored to meet the application’s needs, such as JSON or HTML.

The architecture is composed of a set of loosely coupled components, each having its local storage or connection to a shared database. These components are referred to as services and communicate with each other via a middleware communication channel that can act as the primary interface or solely for communication purposes. In this construction, communication occurs on topics using writer and reader, as explained in chapter 3.2. Therefore, a Service Access Pattern is necessary, which describes the context identifier and services based on attributes, including functional requirements.

However, it should be noted that excessively nested query structures with numerous service calls may lead to performance issues, which can be addressed in various ways. One approach is to restrict the depth of objects or aggregate the number of service calls and process them in batches. It is also worth mentioning that similar issues have been encountered and resolved in the context of Microservices, Edge systems, and Enterprise systems, as discussed in previous chapters. Therefore, the advantages and disadvantages of these systems should also be taken into consideration. Since this is only a methodical approach to processing data-oriented requests.

Overall, the loosely coupled architecture facilitates easy modifications and extensions, as each component can be retrieved and edited independently. Even if a service or component becomes unavailable, the system can still deliver other retrieved contents, although specific attribute results may be missing. The Service Access Pattern segregates the processing or event actions and data, simplifying the separation of components that only work with code or data. This approach helps in managing and maintaining the system. The basic principles of distributed systems still apply, with the only difference being the approach to communication and calling different components in a loosely coupled component architecture.

### 4.3.1 Utilizing data-oriented design

To effectively utilize the data-oriented design approach, certain rules should be followed. In data-oriented architectures, prioritizing data over code is crucial. This involves designing the application or ecosystem around data rather than the code, resulting in flexibility and scalability. This is already achieved through the separation of data access and functional requirements using the aforementioned Service Access Patterns.

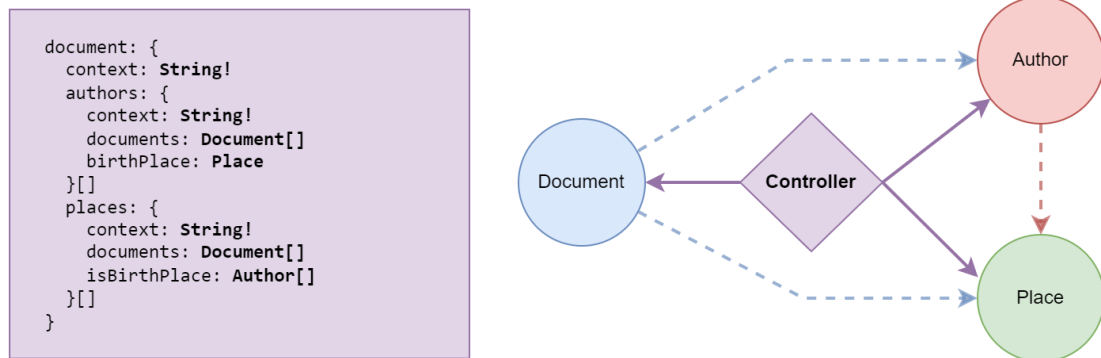
In addition, a middleware infrastructure can be leveraged to automatically establish direct data paths between data readers and writers of a topic [Jos07]. This simplifies system management and communication between loosely coupled components. Furthermore, when combined with a Service Access Pattern that defines the context identifier and services using attribute-based descriptions, developers can optimize data retrieval and processing. This pattern follows a systematic event-based chain of calls, ensuring consistent data handling while minimizing the risk of errors and discrepancies.

Therefore, modular development is another important rule in data-oriented design. Applications or ecosystems should be developed in small, independently testable, integratable, and deployable modules. This approach enables incremental and independent development, facilitating faster issue identification and resolution while reducing the likelihood of introducing new bugs. This is often already achieved by the nature of distributed systems, where the goal is to have loosely coupled components that typically



focus on specific objectives. By specializing the generic architecture principle into various styles such as data flow architecture, event-driven architecture, or service-oriented architecture based on system needs, developers can create applications tailored to specific organizational requirements [Jos07].

### 4.3.2 Service Traversal



**Figure 4.5:** Concept: Service traversal with controller.

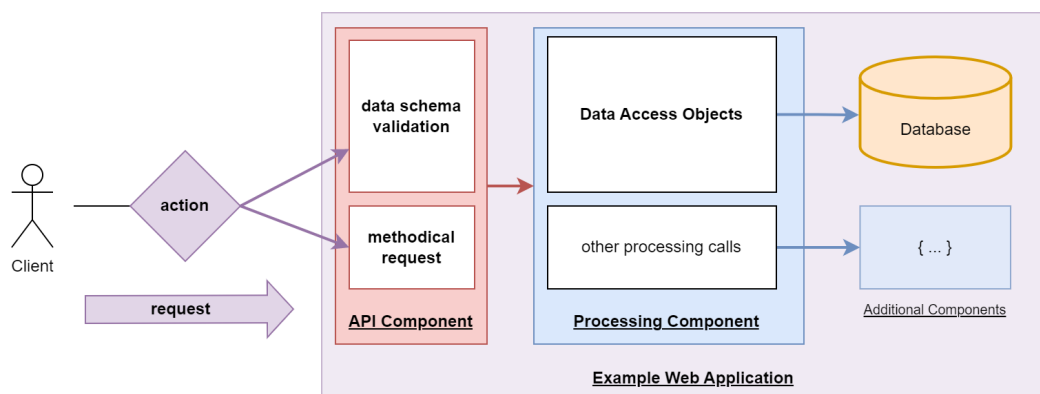
Assuming that a document contains authors and places related to the document, the authors may reference other documents they have written, as well as the document from which they were inferred as the author. The places may also reference all documents in which they have been mentioned. Additionally, places may be referenced by authors, so authors are also listed in places. This creates a network of interconnected elements that can be traversed in different directions.

To gain access to this network, it is necessary to establish a Service Access Pattern. It is recommended to carefully select an initial starting point for the request and perform schema validation prior to proceeding. Adherence to performance regulations is crucial, and it is advisable to restrict the depth of the structure. If required, aggregating requests into batches can prove advantageous. The responsibility of managing the starting point is bestowed upon a controller, as depicted in Figure 4.5. This controller possesses the schema and is capable of validating and executing diverse traversals. Serving as a symbolic entity, the controller can be implemented as a loosely coupled component, such as the document service, or integrated into middleware. The key lies in employing a controller that effectively validates the model and directs service requests to the components through a dedicated communication channel. The process of traversal can be likened to the mechanisms employed by the GraphQL validation and execution engine in handling requests, as extensively discussed in chapter 2.3.

## 4.4 Application Concepts

To ensure a data-oriented web application functions properly, it is necessary to adhere to the architectural constraints of the data-oriented design approach and principles of data-oriented programming during development. This creates an application that is independent from other applications and internally loosely coupled, as its components are already designed to be independent according to these principles.

Assuming that a web application comprises a client that generates a user interface and a server that handles methodological processes and queries, and given the previous discussion on distributed systems, the focus will be on a loosely coupled component of this system - the server. A server-side application will be used as an example of a possible web service component in a larger system, emphasizing data-oriented programming and the straightforward conceptual structure of the server application. To keep it simple, the web server will be able to fetch and process all data sources within the designated resolved components. Therefore, no distributed service will be intentionally built.



**Figure 4.6:** Concept: Example web application.

Figure 4.6 demonstrates that when a client makes a request, the API component accepts it first, and then separates it into data validation and methodical procedures. The API component can be a basic REST interface or, as previously mentioned, a GraphQL interface. The latter is preferred due to the advantages provided by the validation and execution engine, as discussed in chapter 2.3. Moreover, it supports object traversal and eliminates the need to write separate processes for resolving data structures, as these structures are already present, because each field can have its own resolver.

Using the GraphQL API component, the request object can be traversed and aggregated with Data Access Objects in accordance with the principles of data-oriented programming, facilitating efficient data validation and aggregation. They provide static methods, creating interfaces for data storage requests that are carried out independently of the data object. Once the aggregation is complete, the response can be sent.

Traversing the request in the API component enables the determination and execution of tasks at the attribute level. This approach shares similarities with the principles employed in designing a distributed data-oriented architecture. However, it offers unique advantages when applied within the scope of an application.

In this approach, each attribute of an object corresponds to a task that is processed by its own field resolver. The method tree traversal ensures that each method call is associated with a context linked to the root element of the current object. Consequently, the context for the location of a nested attribute is always accessible. For instance, consider an author object with the attribute `documents`. If each document contains the attribute `places` that requires resolution, the method call for `places` will always have access to the identifier of the corresponding document.

The data access methods utilized during attribute resolution are static and can be invoked from any location as long as the context is available. Following the principles of data-oriented programming, these methods prioritize data-centricity, making them straightforward to maintain and develop. Their primary purpose is to access the relevant storage or processing methods to retrieve the semantic content required at a given point. To facilitate this approach, Data Access Objects are introduced. They comprise a collection of static methods that are associated with a specific data category. These methods streamline data access tasks by encapsulating them within a cohesive unit.

This approach not only supports data reading and writing operations but also accommodates the reflection of various events, such as Data Access Patterns and functional requirements. For instance, when fulfilling a functional requirement to export records in a specific format, as identified during the requirement analysis and discussed in Chapter 4.2, the process can involve accessing the records followed by invoking a dedicated procedure to export them in the desired format.

#### 4.4.1 Utilizing data-oriented programming

Data-oriented programming is guided by four core principles. First, it emphasizes separating code from data, leading to improved code reuse, isolated testing, reduced system complexity, and enhanced maintainability. Second, it favors generic data structures, enabling flexible data models, generic functions, and simplified representation of diverse data types. Third, it prioritizes data immutability, enhancing concurrency, reducing bugs, and facilitating data comparison. Finally, it advocates for separating data schema and representation, allowing for flexible data validation, optional fields, validation conditions, and automated data model visualization [Sha22].

A practical implementation of these principles involves the use of Data Access Objects, which act as an abstraction layer between code and data. They bring numerous advantages, such as enhanced code reuse, simplified testing, and reduced system complexity. These objects are typically implemented as static methods, enabling them to operate on data independently without the need for additional code. This approach facilitates the separation of data access logic from other components, resulting in improved maintainability and easier updates. By leveraging generic data structures instead of specific classes, they can also handle various data types and operations, fostering flexibility and simplicity in software design.

As mentioned earlier, maintaining the immutability of the aggregated object is crucial during request processing and attribute iteration. To achieve this, data records obtained during a request, which need modification, are not directly modified. Instead, a copy is created to ensure that the request object and the aggregated response object remain intact and unaffected in other areas of the system.

The separation of data schema and representation offers developers the ability to achieve flexible data validation, incorporating optional fields and data validation conditions. Additionally, this approach automates the visualization of the data model, simplifying comprehension and maintenance. GraphQL is a powerful tool for invoking schema validations, allowing for the definition and traversal of a schema. However, there are alternative libraries that can also be utilized for this purpose.

### 4.4.2 Data Access Objects

Using Data Access Objects can simplify access to specific data structures. In common object-oriented approaches, Data Access Patterns are used to separate semantic database code from the actual code. In such cases, entities like users are provided with a User Access Object that explicitly retrieves data from the database for that type of object. This approach supports concurrency and transactions, while also separating code structures that may have syntactic similarities but differ in their semantic execution [Noc04]. However, it is still limited to explicit objects such as users and cannot be easily extended to other object instances without the need for inheritance or instantiation.

In the context of data-oriented programming, the utilization of Data Access Objects can be further customized and extended to align with the aforementioned principles. Rather than associating a single object with an access object, a static class architecture can be employed to provide an access object for each data category. As mentioned earlier, the separation of methods from code that differs in semantics has already been accomplished. Introducing an abstraction layer also resolves hierarchical relationships.

In the scenario where an object possesses nested and repeatable attributes, with each attribute capable of independent querying from the database, this approach brings notable advantages, particularly for attributes derived from relationships with other data records. The nesting of objects that preserve the same attributes simplifies the creation of recursive operations. Consequently, there is no requirement for implementing numerous additional functions or definitions beforehand, as the same static methods can be repeatedly invoked.

Listing 4.1: Static method implementation in a Text Data Access Object.

```

1 export default class TextDAO {
2   public static async getText(textId: string): Promise<IText> {
3     const query = `Cypher Query for Text attribute`;
4     const result = await Neo4jDriver.runQuery(query, { textId });
5
6     const text = result.records[0]?.get("text") ?? null;
7     return text;
8   }
9
10  public static async getEntities(textId: string) { ... }
11
12  public static async getDocument(textId: string) { ... }
13 }

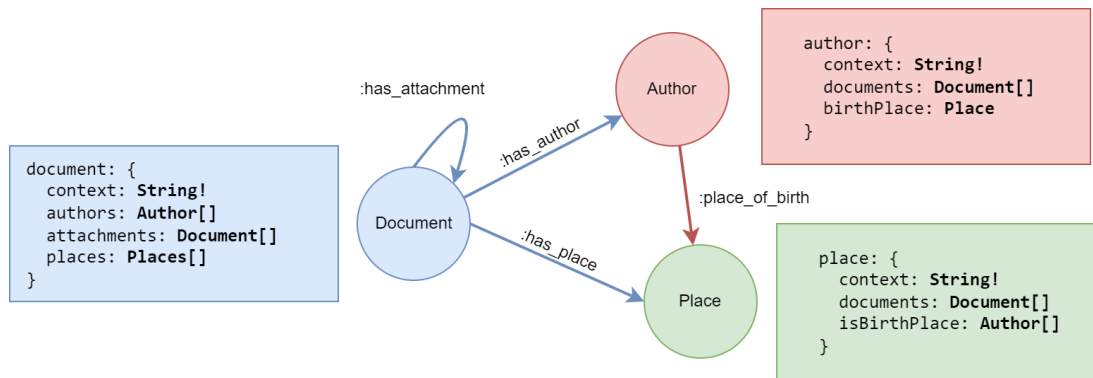
```

Listing 4.1 demonstrates the use of static methods in a Text Data Access Object, exemplifying the application of data-oriented programming principles. The static method `getText` retrieves the text node from a Neo4j graph database. Employing a Cypher query, it locates the text node based on the provided context, returning the result without any further modifications. If no node is found, it appropriately returns `null`. This method is invoked whenever there is a need to locate text data within its associated context.

Therefore, the `TextDAO` class serves as an abstraction layer, housing static methods that specifically handle text contexts. These methods can be accessed anytime for querying text entities or establishing document contexts, necessitating only the relevant context for their operation. Consequently, instantiation is unnecessary, as only the required data for querying is needed initially.

In summary, Data Access Objects, in this context, encapsulate a collection of static methods that can be accessed from any scope and provide data-oriented information efficiently within a specific context. These methods can be repeatedly used throughout the traversal of request objects.

## 4.4.3 Data Traversal



**Figure 4.7:** Concept: Cyclic data dependencies.

Considering a document that includes authors and associated places, authors can reference other documents they have written, as well as the document from which they derive as authors. Similarly, places can refer to all documents in which they are mentioned. This example can be examined from both an architectural perspective and within the context of the application itself. It represents a network of interconnected elements that can be traversed in multiple directions. The primary distinction between traversing at the architectural and application levels lies in the execution process. Traversal explicitly transpires within the application through a method for handling the request and the various Data Access Objects discussed in the preceding chapter.

The process of accessing the network begins by identifying the necessary data to fulfill a specific request. For instance, if a user requests information about a document, its authors, associated places, and other documents connected to the authors and their authors, the problem can be expanded as required. The resolution entails identifying the relevant attributes and gradually populating them using Data Access Objects.

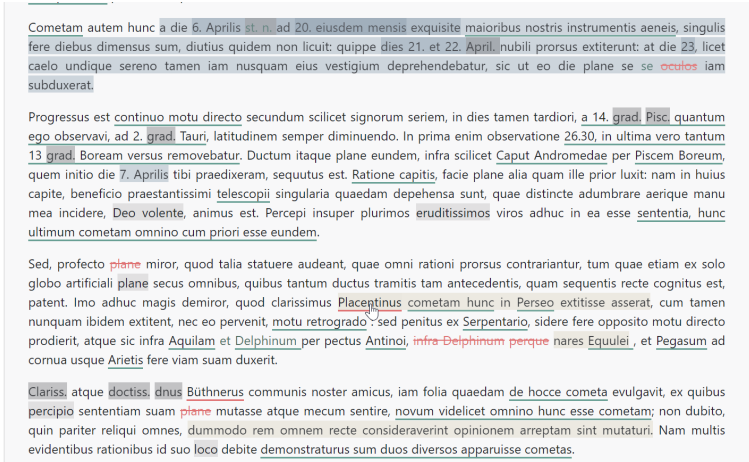
Technologically, this can be achieved through the use of GraphQL and its validation and execution engine. As individual fields are traversed, their corresponding resolvers are invoked, triggering the retrieval of related data through Data Access Objects within the current context. The engine automatically generates a tree structure, processing all nested objects and their attributes from root to leaf. By executing the Data Access Objects within their respective contexts, the required data is queried, ultimately resulting in the delivery of a response.

To further enhance the capabilities of Data Access Objects, a graph database can be employed. This enables the exploitation of cyclic dependencies, facilitating smoother recursion and more efficient data retrieval.

## 5 Implementation

This chapter focuses on implementing the previously discussed concepts for digital editions in the field of Digital Humanities. The implementation specifically addresses web applications, involving the development of a digital edition with a client-server architecture and integration of third-party tools into the workflow. It serves as an appendix to this Thesis and will be gradually released afterwards as an open-source application to the broader Digital Humanities community as the projects advance.

The implementation is relevant to two projects mentioned in the fundamentals section: The Socinian Correspondence and Hildegardis Bingensis. Both projects were considered during the requirements gathering phase, utilizing use cases and contributing to the development of common concepts in data-oriented applications and architectures. Each project underwent iterative implementation using the aforementioned concepts, with regular usability evaluations conducted through monthly meetings. A structured agenda defined the parts of the applications to be tested, enabling gradual implementation, improvement, and testing of the concepts. Simple use cases within both projects further confirmed the functionality of the defined concepts.



[ 2r ]

Cometam autem hunc a die 6. Aprilis st. n. ad 20. eiusdem mensis exquisite maioribus nostris instrumentis aeneis, singulis fere diebus dimensus sum, diutius quidem non licuit: quippe dies 21. et 22. April. nubili prorsus extiterunt: at die 23. licet caelo undique sereno tamen iam nusquam eius vestigium deprehendebatur, sic ut eo die plane se se oculos iam subduxerat.

Progressus est continuo motu directo secundum scilicet signorum seriem, in dies tamen tardiori, a 14. grad. Pisc. quantum ego observavi, ad 2. grad. Tauri, latitudinem semper diminuendo. In prima enim observatione 26.30, in ultima vero tantum 13 grad. Boream versus removebatur. Ductum itaque plane eundem, infra scilicet Caput Andromedae per Piscem Boreum, quem initio die 7. Aprilis tibi praedixeram, sequutus est. Ratione capitis, facie plane alia quam ille prior luxit: nam in huius capite, beneficio praestantissimi telescopii singularia quaedam deprehensa sunt, quae distincte adumbrare aeri que manu mea incidere, Deo volente, animus est. Percepi insuper plurimos eruditissimos viros adhuc in ea esse sententia, hunc ultimum cometam omnino cum priori esse eundem.

Sed, profecto plane miror, quod talia statuere audeant, quae omni rationi prorsus contrariantur, tum quae etiam ex solo globo artificiali plane secus omnibus, quibus tantum ductus tramitis tam antecedentis, quam sequentis recte cognitus est, patent. Imo adhuc magis demiror, quod clarissimus Placentinus cometam hunc in Perseo extitisse asserat, cum tamen nunquam ibidem extitent, nec eo pervenit, motu retrogrado: sed penitus ex Serpentario, sidere fere opposito motu directo prodierit, atque sic infra Aquilam et Delphinum per pectus Antinoi, infra-Delphinum perque nares Equulei, et Pegasus ad cornua usque Arietis fere viam suam duxerit.

[ 2v ]

Clariss atque doctiss, dnus Büthnerus communis noster amicus, iam folia quaedam de hocce cometa evulgavit, ex quibus percipio sententiam suam plane mutasse atque mecum sentire, novum videlicet omnino hunc esse cometam; non dubito, quin pariter reliqui omnes, dummodo rem omnem recte consideraverint opinionem arrectam sint mutaturi. Nam multis evidentibus rationibus id suo loco debite demonstratur sum duos diversos apparuisse cometas.

Liste mit Annotationen schließen

» Placentinus « x  
Placentinus, Johannes

» Placentinus cometam hunc in Perseo extitisse asserat « x

Hevelius bezieht sich auf eine Information, die ihm Lubieniecki am 24. April 1665 hatte zukommen lassen. Da Placentinus ein geübter Astronom war, handelt es sich zweifellos um ein bloßes Versehen, eine Verwechslung der Namen "Perseus" und "Pegasus". Im Postskript zum Brief vom 29. Mai 1665 an Hevelius macht Lubieniecki auf die Berichtigung durch Placentinus aufmerksam.

**Figure 5.1:** Placentinus: An interconnected annotation entity in the text view of the new Socinian Correspondence scholarly digital edition.

The Socinian Correspondence project faced significant challenges right from its inception. As explained in the project description in Chapter 2.1, the initial attempt to create an

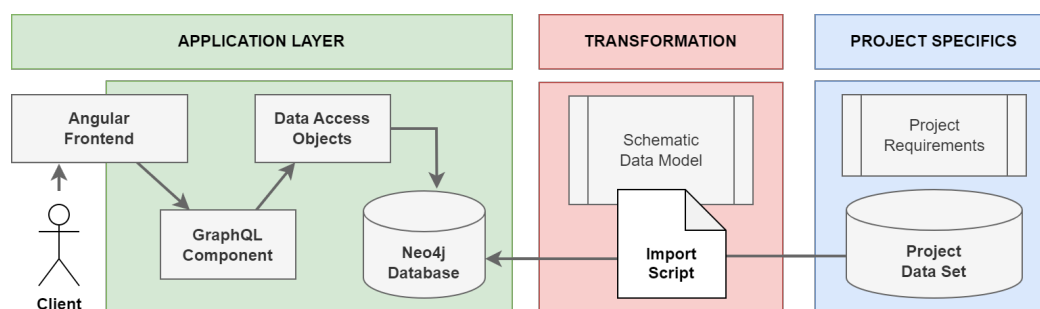
interconnected network using object-oriented programming proved to be too complex and was ultimately abandoned. This decision was made due to the negative impact on performance and data handling, as the networks of data became less transparent and integration became more difficult. The inclusion of outdated technologies like Typo3 and the incorporation of additional languages further complicated matters, with a focus on functionalities overshadowing a thorough examination of the actual data and limited querying to essential elements only. To implement the new scholarly digital edition, the SQL database had to be discarded as well. The goal was to migrate to a graph database to enable the creation of interconnected relationships between elements as illustrated in Figure 5.1. The hierarchical structure of XML posed challenges, making it either impossible or more difficult to establish such connections, often resulting in redundant entries and lists. Consequently, an import or transformation script was developed to facilitate the conversion from the XML database to the Neo4j database. This script handles the necessary transformations to ensure a successful transfer of data between the two databases. Additionally, it was crucial to maintain consistency with the existing workflows, allowing stakeholders to continue their work without the need for retraining on new technologies or potential error consequences. This ensured uninterrupted progress and minimized disruptions. Further details regarding transformation and its implications for implementing data-oriented concepts will be explored in subsequent chapters.

Hildegardis Bingensis did not have an existing digital edition, but the data sets were available in a suitable format in a graph database. Therefore, there was no need for specific data format conversion as mentioned in the previous paragraph. However, the data needed to be adapted to a data model that aligned with a data-centric system rather than an object-oriented development approach. Since the new scholarly digital edition had already been developed as part of the Socinian Correspondence project, another developer took on the task of transferring it to the new project, embracing the data-oriented concepts and initiating further platform development. Despite some differences in the data model, the developer was able to leverage the existing server and client functionalities without encountering significant challenges. This project presents an opportunity to evaluate and validate the concepts applied in the initial development of the digital edition.

Throughout the implementation of the concepts, the data model underwent several revisions. These revisions were informed by weekly iterative meetings, with additional workshops conducted to establish cohesion among the interdisciplinary fields of Digital Humanities and Computer Science.



## 5.1 Overview



**Figure 5.2:** Scholarly Digital Edition: Components and specifics.

The scholarly digital edition, created as a part of this Thesis, was primarily implemented for The Socinian Correspondence project, incorporating the concepts devised for its development. The implementation took into account the specific requirements of the project and the existing data sets, which were stored in multiple databases. These data sets had to be transformed to conform to a standardized data model suitable for the application. The application itself consisted of various components, such as the Angular frontend, GraphQL interface, Data Access Objects, and the Neo4j database. A simplified representation of the application setup can be seen in Figure 5.2. Tools and components not relevant to this chapter were not included.

In the domain of Digital Humanities, projects often possess data sets stored in formats like XML or MySQL databases, as discussed in the fundamentals chapter. These data sets are typically processed and will continue to be processed in their current formats in the future. However, these formats often lack flexibility and impose limitations on the potential for expansion within the application context. To address these limitations, the transformation process aims to convert the existing data sets into a more versatile and manageable format, guided by a predefined data model. This transformation ensures smooth functionality for previously designed interfaces in new projects and simplifies the integration of new interfaces. Moreover, it guarantees adherence to data-centric principles, facilitating the effective utilization of data-oriented approaches rather than being restricted by object-oriented paradigms.

Therefore, the primary objective of the transformation is to establish a transparent and coherent data model that can be utilized not only by the current project but also by future undertakings in the field of digital edition development. This data model prioritizes a data-centric approach, allowing for extensibility and enabling developers to implement it in various formats, including leveraging Neo4j. To achieve this, the data model underwent refinement through rigorous discussions and workshops involving experts from the Digital Humanities field. These deliberations carefully considered the specific requirements of individual projects as well as broader initiatives.

After the transformation was completed, the web application underwent necessary adjustments to ensure efficient utilization. The platform is accessible to end users through Angular, providing them with access to the implemented functionalities in the frontend. Additionally, users can engage with the GraphQL interface, which is provided as an open API. This enables users to leverage the frontend features while utilizing the Graph Query Language and making new queries to the database through the dedicated Data Access Objects, as mentioned earlier. The data-oriented approach, as discussed, offers notable advantages, as each attribute or attribute group is associated with its respective Data Access Object. It is important to note that the developed digital edition operates as an independent application and does not function within a distributed network. Consequently, only the concepts pertinent to the application level were implemented, which have been elucidated in this Thesis.

### 5.1.1 Technology Stack

The technology stack utilized in these projects is essential for accomplishing the intended goals. This chapter presents a comprehensive overview of the stack, emphasizing the meticulous selection and integration of various components. The chosen stack comprises Angular for the frontend, Node.js Express for the server, GraphQL as the query language, and Neo4j as the graph database. Only the relevant components pertaining to the developed concepts will be discussed in this chapter.

The seamless functioning of the digital edition heavily relies on the integration and interaction among the technology stack components. The Angular frontend communicates with the Node.js Express server via GraphQL. Queries and mutations in GraphQL are transmitted from the frontend to the server, which interacts with the Neo4j database to retrieve or manipulate the requested data using Data Access Objects. The GraphQL resolvers in the server translate incoming queries into data access queries, retrieve the pertinent data from Neo4j, and return it to the frontend in a structured format. Therefore, this stack offers a scalable frontend, a lightweight server, an optimized data querying language, and a graph database capable of handling highly interconnected data structures. This combination of technologies establishes a robust foundation for the successful implementation of data-oriented concepts at the application layer.

#### Frontend: Angular

Angular is a widely adopted frontend framework known for developing dynamic web applications. It utilizes TypeScript and HTML to create a robust and modular structure. Angular offers a component-based architecture, two-way data binding, and dependency injection, making it scalable, maintainable, and robust. Its component-based architecture

promotes code reusability and modularity, enabling the development of responsive user interfaces. Additionally, Angular's strong support for TypeScript enhances development productivity and code quality.

#### Server: NodeJs Express

Node.js Express is a lightweight and popular web application framework designed for Node.js. It simplifies server-side development by providing a minimalistic yet powerful set of tools. Express handles routing, middleware management, and HTTP request handling. It was chosen for its simplicity and widespread usage. Its lightweight nature allows for efficient handling of multiple concurrent requests. Express provides an intuitive interface for handling HTTP requests, making it an ideal choice for building server-side components and integrating a GraphQL endpoint.

#### Query Language: GraphQL

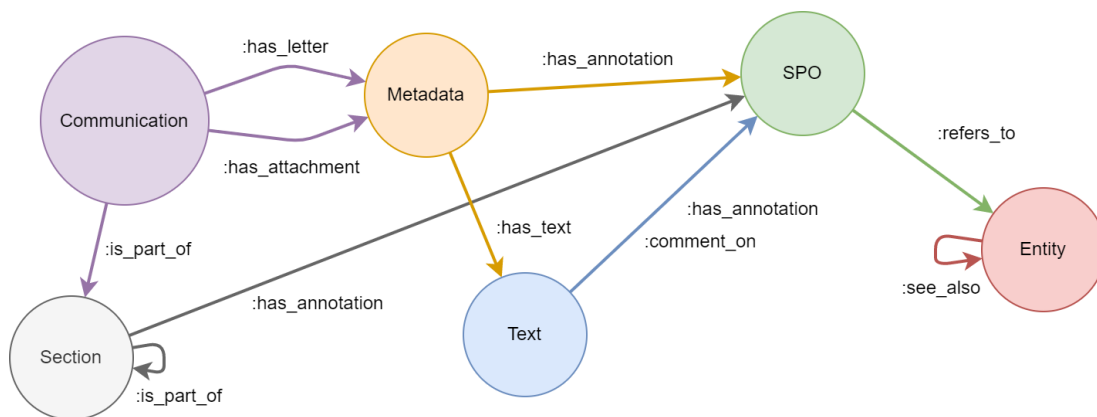
GraphQL is a query language for APIs that enables efficient data retrieval and manipulation. It offers a flexible and declarative approach to data fetching, allowing clients to request specific data structures. GraphQL reduces over-fetching and under-fetching commonly encountered in traditional RESTful APIs. By providing a precise and customizable data retrieval mechanism, GraphQL minimizes network overhead and improves performance. It allows clients to request only the required data, resulting in more efficient communication between the frontend and backend components. For further details on GraphQL, please refer to the fundamentals covered in Chapter 2.3.

#### Graph Database: Neo4j

Neo4j is a graph database that utilizes graph structures to store and process data. It offers high performance for graph-related queries and provides a flexible schema that adapts to evolving data models. Neo4j allows for efficient representation and traversal of relationships, making it ideal for applications with highly interconnected data. The selection of Neo4j as the database aligns with the research project's requirements. Its graph database model is well-suited for representing and querying highly interconnected data structures. The use of Neo4j facilitates efficient querying and analysis of complex data sets, supporting the research objectives effectively. For further details on Neo4j, please refer to the fundamentals covered in Chapter 2.4.

### 5.1.2 Data Refinement

As previously discussed, it is necessary for projects to be built on a unified data model. Data-oriented programming is not a one-size-fits-all solution for every problem. Additionally, not all projects can be abstracted and generalized in a way that allows them to be solved with a single paradigm. Furthermore, the successful implementation of these concepts ultimately depends on human expertise, as they need to be implemented correctly and without unnecessary complexity. Nevertheless, it is possible to bring projects closer to a unified data schema, enabling the developed web application to seamlessly integrate with new projects that adhere to the same data model. Of course, some adjustments may be required to create additional interfaces, address specific issues, or meet particular requirements. Furthermore, projects can still maintain their own data models as long as there is a component or layer where a transformation can be performed. For example, an import script can be utilized to convert a project's data model into the unified data model created for the web application.



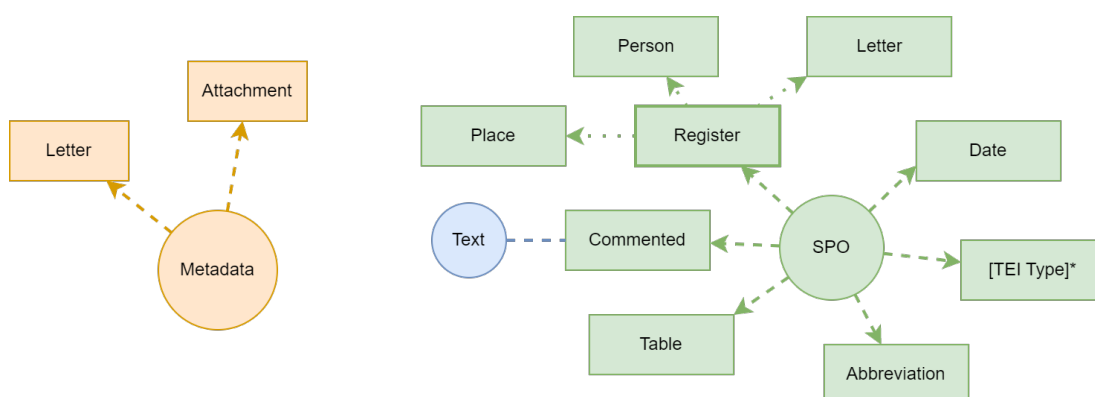
**Figure 5.3:** Unified Neo4j data model for the Socinian Correspondence.

In Figure 5.3, a portion of The Socinian Correspondence data model is illustrated, which has been refined and generalized for the web application. This refinement process involved weekly iterative developer meetings, monthly project meetings, and occasional workshops that specifically focused on improving the data model, with the participation of developers from different projects. The data model places emphasis on the data itself, stored in a Neo4j graph database, and can be accessed efficiently from the server using Data Access Objects and Cypher queries.

One notable aspect is the communication node, which can be seen as a parent node summarizing the metadata of attachments and letters. Furthermore, these communication nodes can also be grouped into sections. These sections can be recursively divided into further subsections as needed. Thanks to the cyclic relationships and the advantages of graph databases, the nodes can be traversed quickly and to the greatest extent possible.

By utilizing data-oriented programming, these benefits can be exposed from the server to the client. Additionally, communication and section nodes can be customized for other projects, offering a range of extension possibilities.

Another example can be observed with metadata, text, and SPO (Standoff Properties) nodes [Nei19]. SPO nodes can be understood as annotations within a text, such as editorial markings that reference entities, provide additional textual annotations, or offer other editorial hints. SPO nodes exhibit significant diversity and should be kept as flexible as possible during the digital edition process, allowing for the addition of new types of annotations in the form of SPO nodes with different relations.



**Figure 5.4:** The Socinian Correspondence: Extended view of node structures.

As illustrated in Figure 5.3, a distinction can also be made between metadata for letters and attachments, which can be letters themselves but with a different relation to a communication process. This creates ambiguity, which is abstracted by the communication node and its relations. The same applies to annotations, as there are numerous types that cannot be solely identified through a single relation in the data model. If this model were transformed into an object-oriented data model, complex structures with various inheritances would emerge around metadata, communication, SPO, and text nodes, as explained in the introduction to data-oriented principles in Chapter 3.1.1 and illustrated in Figure 5.4.

By focusing on the data itself and abstracting through relations, it becomes possible to represent different data models that adhere to the basic structures of the generalized data model presented. Moreover, this data model can be seamlessly integrated into GraphQL schemas, allowing for the definition of various data categories for Data Access Objects. In GraphQL, each interconnected attribute corresponds to a relation in the Neo4j database and facilitates bidirectional transitions between nodes. The aim is to enable extensive querying while considering performance limitations. As concluded in Chapter 3.3, there are multiple potential implementations to enhance the performance of deeply nested queries.

## 5.2 Concept Implementation

The implementation aimed to apply data-oriented programming concepts to web applications in the field of Digital Humanities, as previously discussed. Initially, the project requirements and use cases were analyzed, leading to the classification of functional and non-functional requirements. The functional requirements were further examined, with a specific emphasis on their data-centric nature. This analysis resulted in the identification of Data Access Patterns, which provided valuable insights into the system architecture and how to access the previously defined data model. Adjustments to the unified data model were then made based on the findings derived from the requirements analysis and Data Access Patterns.

The unified data model not only serves as a foundation for the web application but is also utilized by Data Access Objects and the GraphQL interface, both of which operate based on a schema. This adherence to a schema aligns with the data representation principle of data-oriented programming, satisfying two key principles of the paradigm: the separation of code and data, and a schema-oriented approach. The remaining principles are addressed through the implementation of Data Access Objects, the validation and execution engine of GraphQL, and the enforcement of immutability in the resolvers responsible for handling attribute fields.

In the frontend, appropriate methods are used to query the GraphQL interface. It validates the structure of the query and executes the necessary functions for each attribute. These functions, known as resolvers, access the Data Access Objects, which are organized based on the data model and have predefined methods. These methods invoke Cypher queries to access the Neo4j graph database and retrieve the necessary attribute information. Recursive operations occur when traversing nodes in the Neo4j data model. By extending schemas in GraphQL, traversals can be performed bidirectionally through relationships. Alternatively, if GraphQL was not utilized, a custom validation and execution engine would need to be developed, possibly delivered through a REST interface. However, this can be easily achieved using commonly available libraries.

Once the data is queried and made accessible through GraphQL, user interfaces have been developed in the Angular frontend to meet the requested functional and non-functional requirements. Overall, development commenced with a data-oriented approach, implementing the principles and rules of data-oriented programming and leveraging the previously established concepts for application-level use. Evaluation, progress monitoring, and refinement occurred through weekly iterative meetings and monthly structured project-oriented usability tests, involving discussions with editors and project managers.

### 5.2.1 The Socinian Correspondence

This chapter provides scientific insights by showcasing various visual representations that elucidate the concepts previously addressed and thoroughly expounded upon in the earlier sections. The Socinian Correspondence serves as the primary undertaking, acting as the initial project implemented before subsequently expanding upon the same foundation to realize the implementation of Hildegardis Bingensis, incorporating project-specific extensions.

The screenshot displays a web interface for the Socinian Correspondence. On the left, a dark sidebar contains a nested list of corpora under the heading 'Teilkorpus Lubieniecki'. The list includes: 'Partikularkorpus Lubieniecki – Guericke d. Ä.', 'Partikularkorpus Lubieniecki – Guericke d. J.', 'Partikularkorpus Lubieniecki – Hevelius', 'Partikularkorpus Lubieniecki – Müller', 'Partikularkorpus Lubieniecki – Placentinus', 'Partikularkorpus Lubieniecki – Rautenstein', 'Partikularkorpus Lubieniecki – Reyher', 'Partikularkorpus Lubieniecki – Rötlin', 'Partikularkorpus Lubieniecki – Schletzer', 'Partikularkorpus Lubieniecki – Sivers', and 'Partikularkorpus Lubieniecki – Stegmann'. Below this is 'Teilkorpus Ruarus'. The main content area on the right features a 'Read more...' link at the top, a search bar with the text 'Sort: by date sent' and 'Search through corpus...', and a table of sources. The table indicates 'You are viewing sources 1 to 10 from a total of 10.' and has columns for 'Title', 'Variants & Attachments', and 'Status'.

Title	Variants & Attachments	Status
<a href="#">Stanislaw Lubieniecki (Hamburg) an Otto von Guericke d. J. (Hamburg), 23. Dezember 1664</a>		Depth development
<a href="#">Otto von Guericke d. J. (Hamburg) an Stanislaw Lubieniecki (Hamburg), 24. Dezember 1664</a>	1	Depth development
<a href="#">Stanislaw Lubieniecki (Hamburg) an Otto von Guericke d. J. (Hamburg), 24. Dezember 1664</a>	2	Depth development
<a href="#">Otto von Guericke d. J. (Hamburg) an Stanislaw Lubieniecki (Hamburg), 26. Dezember 1664</a>	1 1	Depth development

**Figure 5.5:** The Socinian Correspondence: Example of nested sections.

Figure 5.5 depicts the representation of several correspondences within different sections. By utilizing infinitely nested sections and communications that encapsulate the summaries of letters and their variations and attachments, it becomes possible to incorporate communications into new areas as frequently as possible. The illustration showcases some of these areas, which are organized on two levels: a particular corpus that aggregates a series of communication processes, and a subset corpus that combines several particular corpora. From an interface perspective, this implies sending the same request to the server, while the frontend implementation involves various tree representations tailored to the preferences of the end-user, ensuring optimal visualization.

Similar interfaces for querying different contexts are also required in other scenarios. For instance, when retrieving a list of all possible entities and their corresponding references derived from the relationships to SPO. However, the connection between annotations and the entity is not the only information that can be derived. By traversing the path from the SPO node towards the relations pointing to the SPO node, various elements such as letters, variations, attachments, communication processes, texts, or sections associated with the desired entity can be identified. These elements are referenced through annotations, and the frontend sends a more specific request to the server, which not only requires the SPO node for the entity but also requests all texts, metadata, and sections associated with it.

The screenshot shows a web interface for the Socinian Correspondence. On the left is a navigation menu with options: 'Register of persons', 'Location register', 'Subject index', 'Bible passages register', and 'All register'. The 'Register of persons' is selected. The main content area displays the profile for 'Aristoteles', including birth and death dates (384 v. Chr. and 322 v. Chr.), normdata, and a list of mentions: 'Ariston von Chios (1)', 'Aristoteles (14)', and 'Artemidorus von Parion (1)'. An arrow points from the 'Aristoteles (14)' mention to a 'Mentions:' list on the right. This list contains several entries, each with a link to a specific correspondence, such as 'Stanislaw Lubieniecki (Hamburg) an Johannes Hevelius (Danzig), 2. April 1666' and 'Stanislaw Lubieniecki (Hamburg) an Otto von Guericke d. Ä. (Magdeburg), 15. August 1665'.

Figure 5.6: The Socinian Correspondence: Entity mentions listing.

Using the concepts explained in previous chapters, GraphQL is effectively employed as a query language communicating with the database. It adheres to data-oriented programming paradigms, ensuring scalability, minimal maintenance overhead, and providing an optimal, non-opinionated interface to the Neo4j graph database. An example of such usage can be observed in Figure 5.6, which demonstrates how a user navigates through the registry to access the representation of an entity and subsequently desires to retrieve all occurrences of that entity within the entire interconnected structure.

The screenshot shows a detailed view of a communication node. At the top, it identifies the sender and receiver: 'Samuel Reyher (Kiel) an Stanislaw Lubieniecki (Hamburg), 29. Oktober 1666'. Below this is a 'Metadata' section with fields for 'Sender', 'Sending location', 'Date of dispatch', 'Receiver', 'Receiving location', and 'Date of receipt'. An 'Abstract' section is also present. The main content area shows the letter text, starting with 'Kilonio Hamburgum XIV. Kal. Novemb. Juliani M DC LXVI.' and 'Generose domine, fautor honoratissime!'. A right-hand sidebar contains a list of annotations, each with a title and a link to the corresponding text in the letter, such as '» stellis istis, quae quartae magnitudinis censentur et 25° circiter' and '» novam stellam'. A 'Clear list with annotations' link is at the top of the sidebar.

Figure 5.7: The Socinian Correspondence: Communication view.

Another example can be observed in Figure 5.8. This view represents a communication view that encompasses all relevant information associated with the communication node. It displays all SPO nodes within the texts, identifies and presents various types of texts such as variations and attachments, and includes the necessary metadata linked to the texts. Moreover, this view can also be utilized for the attachments themselves, which, in turn, can form a network of texts, potential variations, metadata, and SPO nodes,



although this is not mandatory. Furthermore, this view can be used to open simple variations if only the variations with their corresponding nodes need to be displayed.

In the background, a single query is sent to the GraphQL interface, which then consolidates the nodes to form an appropriate response. The primary node in this context is always the communication node. If the communication node is completely absent, the request naturally yields no results. However, this situation allows the frontend to use the same identifier for the opened document, enabling the search for a metadata node or, if unsuccessful, a text node. In any scenario, it is possible to traverse in any direction through the relationships that the nodes possess using GraphQL and the Data Access Objects to gather the necessary information and present it to the end-user in the desired format. Additionally, as depicted in Figure 5.8, it is also possible to open the annotations themselves, which can contain further information, texts, comments, or links. These details can also be realized through GraphQL requests and the SPO data category. With the information obtained from the GraphQL request, relationships can be identified and presented to the user on the website in a suitable manner.

Finally, the user can also switch to the previously displayed entity view if the annotation represents a referenced entity. At no point does the user encounter a dead-end, as they can traverse the graph at any time without being overwhelmed by the complex interconnections. All of this is achieved through the server and the GraphQL interface, which provide the user with data-centric queries in an easily maintainable form and with minimal response time. These queries can be expanded upon at any time and are also valuable for exploring other contexts. Furthermore, they offer opportunities for further research, such as in the realm of search and filtering. They enable the discovery not only of specific textual passages but also of nodes and their interconnected relationships. This capability allows for uncovering connections that may transcend linguistic barriers or identify instances where a word is used as a synonym and referenced to an entity.

Search results for "mainz"

A total of 5 results were found with this search term.

2 letters from total corpus   3 register entries

**Johann Ernst von Rautenstein (Regensburg) an Stanislaw Lubieniecki (Hamburg), 23. Februar 1665**

» ... Gravel sei nach Regensburg zurückgekehrt, um den **Mainzer** Erzbischof zu drängen, damit dieser den Ka... «

**Stanislaw Lubieniecki (Hamburg) an Johann Friedrich Schletzer (Frankfurt am Main), 12. September 1665**

» ...ände dem Kaiser wegen des Konfliktes zwischen dem **Mainzer** Erzbischof und dem Kurfürsten der Pfalz (d... «

**Figure 5.8:** The Socinian Correspondence: Search for relations.

### 5.2.2 Hildegardis Bingensis: Liber epistolarum

The implementation of Hildegardis Bingensis was undertaken by a different developer, utilizing the same technology stack and concepts developed in this Thesis. The project primarily revolves around a codex, which consists of brief epistolary texts arranged to form a cohesive theological whole [Kuc]. Within these subdivisions, communication nodes are found, representing letters but containing additional information and details that were not present in The Socinian Correspondence. Therefore, modifications were made solely in the frontend to optimize the display of the required information. Extensions were implemented through Data Access Objects, which were designed to accommodate additional attributes as new relations to other nodes emerged in certain areas. While the foundation was the generalized data model, extensions and relations had to be incorporated, particularly within the SPO nodes.

Throughout the development of this project, several investigations were conducted to evaluate the acceptance and implementation of the concepts proposed in this Thesis. These included examining the time investment, assessing the developer's perception of complexity, familiarizing oneself with the data-oriented backend application, and gauging the developer's acceptance of the employed concepts. However, the evaluation did not follow an acceptance model as it would exceed the scope of the work, but it could serve as a potential avenue for future research interfaces. These aspects were evaluated through iterative weekly meetings using systematic questions aimed at capturing these considerations.

The time investment for the project implementation was relatively low. In a short period, the project was completed and able to display all the letters that had been transformed and imported into the Neo4j graph database. Some adjustments were made in the Data Access Objects, but the most significant modifications occurred in the Angular frontend to meet the project-specific requirements. As for complexity, the understanding and handling of the data-oriented paradigm were examined. It was observed that, aside from the usual questions regarding certain thoughts within the frontend, working with the data-oriented backend was relatively self-explanatory as the developer quickly acclimated to the structures. This was attributed to the well-defined and encapsulated definitions and the aggregation of objects facilitated by the Data Access Objects. Thus, any adjustments were typically made in the GraphQL schema or the Data Access Objects, and GraphQL resolvers were only added when necessary. Consequently, the acceptance of the developer toward the implementation and concepts of data-oriented programming was considered high. Currently, the project continues to be developed using the same technology stack and is being expanded with third-party tools that align with the data-oriented concepts presented in this Thesis.

## 6 Evaluation

This chapter focuses on the evaluation of the concepts developed and discussed in this Thesis. It is important to clarify that only the concepts at the application level will be assessed, as the evaluation of architecture-level concepts and discussions on performance and limitations related to Service Access Patterns will be addressed in future work.

The evaluation of the application-level concepts involved analyzing use cases in regular developer meetings and project meetings with stakeholders. Additionally, these concepts were applied, extended, and utilized by a different developer in another project, with structured observation. Systematic interviews were conducted with this developer during the regular developer meetings to identify any issues, questions, or challenges encountered. Furthermore, the acceptance of the concepts was evaluated through workshops held during a lecture, where students were tasked with developing multiple projects incorporating these concepts. The evaluation process encompassed systematic observation during project work and an assessment of the outcomes delivered.

### 6.1 Use Cases

The use cases were defined and structured both project-specific and project-generic, based on the input from developers and stakeholders from multiple projects. These use cases were then tested during the weekly developer meetings and monthly project meetings with key stakeholders. The weekly meetings provided valuable insights, allowing for the identification of improvements in the concepts and user interfaces of the frontend applications, as well as the introduction of new requirements. The monthly meetings systematically tested the use cases according to a predefined agenda.

The application use cases, described in Chapter 4.1.2 and provided in detail in the appendix A of this Thesis, were compiled by developers and stakeholders from multiple projects. These projects involved interdisciplinary experience and specific goals, as both The Socinian Correspondence and Hildegardis Bingensis projects are publicly funded projects that must meet the mentioned use cases and other project-specific requirements to receive positive evaluations and secure future funding.

Feedback collection played a crucial role in the evaluation process. Surveys and structured observations were conducted to gather feedback from stakeholders and developers. Their valuable input helped identify limitations and challenges encountered during the evaluation. These obstacles included technical constraints, data availability, and time constraints, which had implications for the implementation and evaluation of the concepts. The evaluation results indicated the successful implementation of the application-based use cases, including tasks such as viewing project descriptions and details, accessing text, metadata, and annotations, as well as other project-specific requirements. It is important to note that the variation in use cases across different projects prevented generalization. However, the flexibility of the developed concepts to adapt to new requirements and project-specific scenarios was evident. The successful implementation of new requirements arising from the secondary implementation of Hildegardis Bingensis further validated the effectiveness of the concepts for data-oriented programming in digital editions.

## 6.2 Concepts Re-implementation

The concepts at the application level were first applied and implemented within The Socinian Correspondence Project. Through the use of use cases and detailed evaluation, it was determined that the implementation of these concepts was successful. All requirements from stakeholders were met, and developers expressed positive feedback during weekly meetings regarding the use of data-oriented development. To further enhance this positive feedback, the application of data-oriented concepts at the application level was tested in the second project, Hildegardis Bingensis.

Another developer took over the implementation of the application, incorporating components from The Socinian Correspondence, as explained in Chapter 5.2.2. This included both the client and server applications. During the initial development phase, the time investment, perception of complexity, familiarity with the data-oriented backend application, and the developer's acceptance of the employed concepts and their continuation within the project were evaluated.

The results indicated that the developer not only saved time by adopting both the client and server components, but also by utilizing generalized data model and optimally usable Data Access Objects. As a result, the project became operational within a few hours. Only a few adjustments in the user interface and expansions in the Data Access Objects were required to address specific project-related requirements that couldn't be generalized. Weekly meetings revealed that the developer quickly adapted to and grasped the concepts. The developer had only minor inquiries regarding the execution in the frontend, unrelated to the actual data-oriented concepts in the server. Working

with GraphQL, Data Access Objects, and the Neo4j database posed no significant challenges. The developer rated the complexity as low, as the application could easily be extended with third-party tools and specific project requirements. Hence, it can be concluded that there is a high level of acceptance among developers for utilizing these concepts in digital editions. This was also confirmed through workshops conducted with other developers in various locations. Additionally, the acceptance was assessed through collaboration with master's students, as described in the following chapter.

## 6.3 Concepts Acceptance

The concepts of data-oriented application development and its principles were introduced to students in the Master's program in Computer Science during a lecture. Students first familiarized themselves with these concepts and then worked with a technology stack consisting of GraphQL and Neo4j as the database. In a workshop, students were required to build a data-oriented application based on a generalized data model reflecting the social media platform Twitter. This application utilized GraphQL as the interface, Data Access Objects for communication with the graph database, and Neo4j as the underlying graph database.

The learning process took place over several steps and lecture sessions. Initially, the foundational components were established, and the principles of data-oriented development were taught. Subsequently, the usage of Data Access Objects in conjunction with the Neo4j graph database was explained, with students implementing initial examples. Next, a data model for the entire course was established, and GraphQL was introduced as a lecture topic, accompanied by exercises and examples. As a final step, students were required to implement a server-side application that integrated and utilized the concepts they had learned. The implementation proceeded smoothly, and students quickly adapted to the new concepts. Acquiring a different programming paradigm alongside object-oriented programming appeared to pose no difficulties. On the contrary, students embraced the challenge of implementing data-oriented programming concepts as efficiently as possible.

During the lecture, students were also tasked with developing a digital edition for the Regesta Imperii project. Although this student project is not the focus of this Thesis, it serves as another successful example developed using data-oriented concepts. The server-side application for this project utilized the same technology stack as the exercise sessions, while students had the freedom to choose the frontend technologies. Therefore, the use of the data-oriented concepts discussed in this Thesis led to the acceptance of new possibilities for students who were new to the realm of data-oriented web development.

## 6.4 Outcomes and Impact

The evaluation of the concepts developed in this Thesis has provided valuable insights into their effectiveness and acceptance among developers and stakeholders. The use cases and their testing in regular developer meetings and project meetings allowed for the identification of improvements and the introduction of new requirements. The successful implementation of these scenarios in different projects, such as The Socinian Correspondence and Hildegardis Bingensis, demonstrated the flexibility and adaptability of the concepts to meet project-specific requirements. The positive feedback from developers and stakeholders further validated the effectiveness of the concepts for data-oriented programming in digital editions.

The re-implementation of the concepts in the Hildegardis Bingensis project by another developer showcased the time-saving benefits and ease of adoption of the data-oriented approach. The developer quickly grasped the concepts and was able to make the project operational within a short period of time. The low perceived complexity and the ability to extend the application with third-party tools and specific project requirements further highlighted the acceptance and practicality of the concepts.

The acceptance of the concepts was also assessed through workshops conducted with master's students. The students successfully implemented a data-oriented application based on a generalized data model, utilizing GraphQL and Neo4j. The learning process was smooth, and students embraced the challenge of implementing data-oriented programming concepts. The successful development of the Regesta Imperii project using data-oriented concepts further reinforced the acceptance and possibilities offered by these concepts.

Overall, the evaluation results indicate that the concepts developed in this Thesis have been well-received and have had a positive impact on the development of digital editions. The concepts have demonstrated their effectiveness in meeting project-specific requirements, saving time, and providing flexibility for developers. The acceptance and successful implementation of the concepts by different developers and students validate their practicality and potential for broader adoption in data-oriented application development.

## 6.5 Limitations and Future Work

While the evaluation of the application-level concepts has provided valuable insights, there are certain limitations that should be acknowledged. Firstly, the evaluation focused primarily on the application level, and the assessment of architecture-level concepts and discussions on performance and limitations related to Service Access Patterns were left for future work. Future evaluations should address these aspects to provide a more comprehensive understanding of the concepts.

Secondly, the evaluation process relied on a limited number of projects and developers. While the projects involved interdisciplinary experience and specific goals, the generalization of the evaluation results may be limited. Future work should involve a larger sample size and a wider range of projects to validate the findings and assess the scalability of the concepts.

Additionally, the evaluation process identified technical constraints, data availability, and time constraints as obstacles during the implementation and evaluation of the concepts. Addressing these limitations and exploring ways to overcome them would be valuable in further enhancing the effectiveness and applicability of the concepts.

Furthermore, the evaluation primarily focused on the perspectives of developers and stakeholders. Future work should also consider the perspectives of end-users and assess the impact of the concepts on user experience and usability.

In conclusion, while the evaluation of the concepts developed in this Thesis has provided promising results and demonstrated their effectiveness in digital edition projects, further research and evaluation are needed to address the limitations and explore their broader applicability. The findings from this evaluation serve as a foundation for future work and highlight the potential for the adoption of data-oriented programming concepts in the development of digital editions.





## 7 Conclusion

This concluding chapter presents an overview of the concepts of data-oriented programming for web technologies in the field of Digital Humanities. The chapter highlights the findings obtained through the evaluation of these concepts, focusing on their effectiveness in meeting project-specific requirements and their acceptance among developers, stakeholders, and master's students. Furthermore, it discusses the limitations encountered during the evaluation process and identifies potential avenues for future research and improvement. By summarizing the key outcomes, this chapter offers a comprehensive understanding of the implications and potential applications of data-oriented programming in the development of digital edition projects in the context of the Digital Humanities.

### 7.1 Summary

This Thesis investigates the application of data-oriented programming concepts in web technologies to enhance software management and improve data accessibility in the Digital Humanities domain. Digital Humanities merges computational methods with traditional practices in the Humanities to study and analyze cultural and historical information. However, effectively managing and analyzing the vast amount of data, along with its intricate interconnections, poses significant challenges in this field, especially when developing scholarly digital editions intended to present information to end users in a suitable manner.

The research investigates the application of data-oriented design and programming to optimize the efficiency of developing digital editions. Through the organization of code and data, data-oriented design simplifies the management and analysis of intricate data sets, ensuring the long-term scalability and maintainability of digital editions. The utilization of data-oriented programming holds promise for various domains, including the Digital Humanities, Software Engineering, and Digital Scholarship. The proposed concepts offer valuable guidance to developers in effectively managing data and constructing maintainable and modular systems. This, in turn, enhances software management and data accessibility within the Digital Humanities, whether on a per-application basis or within distributed architectures.

The findings of this Thesis highlight the importance of interconnectivity and leveraging existing data sets in the Digital Humanities. The developed data-oriented concepts shed light on key areas of advancement, including the construction of data driven architectures capable of effectively managing large volumes of data. Additionally, they emphasize the necessity of developing robust models and schemas that enable the representation and encoding of digital objects. Moreover, Data Access Patterns have been introduced as a means to extract data-centric requirements from functional requirements across different projects. These endeavors aim to improve the usability, maintainability and functionality of current tools and systems, while simultaneously addressing challenges and concerns in data-oriented programming for efficient data management and analysis within the field of Digital Humanities.

The methodology employed in the research involves literature analysis, use cases, concept development, and project implementation to study data-oriented programming concepts in the Digital Humanities. Literature analysis provides insights into existing research on data-oriented programming and architecture, emphasizing the significance of data driven architecture in managing large data sets. Use cases, and requirement collection, including the utilization of Data Access Patterns, offer practical insights and identify essential functionalities and features required for effective utilization of data-oriented concepts and architecture. The realization and implementation of projects involve designing and developing digital platforms, user testing, and evaluation with active engagement from experts and stakeholders.

The Thesis also presents two case studies: The Socinian Correspondence and Hildegardis Bingensis projects, which focus on implementing digital editions in web applications for the Digital Humanities. These projects showcase the practical implementation of data-oriented programming concepts. They utilized technologies such as Angular, Node.js Express, GraphQL, and Neo4j to create scalable and modular systems. The implementations followed a data-centric approach, transforming existing data sets into a unified and manageable format suitable for the applications. Evaluation of the implementations demonstrated positive results in terms of time investment, complexity, and acceptance of the data-oriented concepts.

In conclusion, the research and case studies validate the effectiveness and potential of data-oriented programming concepts in improving software management and data accessibility in the Digital Humanities. The findings contribute to the advancement of the field and provide insights into the benefits and challenges associated with data-oriented programming. Further research and evaluation are necessary to explore the broader applicability of these concepts and address limitations, enhancing maintainability and data accessibility of scholarly digital editions in the Digital Humanities.

## 7.2 Contributions

This chapter discusses the research contributions in various key areas, emphasizing their impact on the field of Digital Humanities. These contributions encompass advancements in methodology, practical implementations, interdisciplinary collaborations, and their overall significance in the development of digital editions in the Digital Humanities.

The research significantly contributes to methodology by examining the integration of data-oriented programming concepts into web technologies for the Digital Humanities. It introduces principles of data-oriented design and programming, underscoring the crucial role of data organization and manipulation in effectively managing and analyzing intricate data sets. Moreover, the research highlights the importance of employing a data-oriented architecture to handle substantial amounts of data efficiently. By offering insights into requirements through the identification of Data Access Patterns, as well as the development of data models and schemas, the research enriches the methodological underpinnings of data management and analysis in the Digital Humanities.

The practical contributions of the research are demonstrated through the implementation of digital editions for the Socinian Correspondence and Hildegardis Bingensis projects. These case studies showcase the practical feasibility and effectiveness of employing data-oriented programming concepts in web applications. Through iterative meetings, workshops, and collaborations, the research engages experts from the Digital Humanities and Computer Science fields, resulting in the development of digital platforms that enhance the accessibility and usability of scholarly digital editions. The implementations integrate technologies such as Angular for the frontend, Node.js Express for the server, GraphQL as the query language, and Neo4j as the graph database. The research also introduces Data Access Patterns and Data Access Objects, providing a structured approach to simplify and optimize access to specific data structures.

The research fosters interdisciplinary collaborations by bridging the gap between the Digital Humanities and Computer Science domains. Involving experts from both fields in the design and development process facilitates knowledge exchange and collaboration. This interdisciplinary approach enables a comprehensive approach to addressing software management and data accessibility challenges in the Digital Humanities. Through such collaborations, the research promotes a deeper understanding of the complexities and requirements of digital edition development, while leveraging expertise from diverse disciplines to drive innovation and advancements.

The overall significance of the research lies in its contribution to improving software management and data accessibility in the Digital Humanities. By leveraging data-oriented programming concepts and methodologies, the research enhances the efficiency and maintainability of digital edition development. The practical implementations and

case studies provide tangible evidence of the effectiveness of the proposed methodologies. Additionally, the interdisciplinary collaborations fostered by the research facilitate knowledge exchange and collaboration between the Digital Humanities and Computer Science fields. The research also addresses existing challenges and issues in data-oriented programming, providing valuable insights for further advancements in the field. Overall, the research contributes to the advancement of the Digital Humanities by offering practical solutions, methodological frameworks, and interdisciplinary collaborations that enhance software management and data accessibility.

### 7.3 Limitations

The utilization of data-oriented programming concepts in web technologies for software management and data accessibility in the Digital Humanities has several limitations that need to be considered. These limitations arise from various aspects of the research and implementation process.

One limitation of this Thesis lies in the focus on specific use cases and project-specific requirements. The findings and conclusions are based on the analysis of The Socinian Correspondence and Hildegardis Bingensis projects, which possess their own distinct characteristics and needs. It is crucial to acknowledge that the applicability of the data-oriented programming concepts may vary when implemented in other digital edition development scenarios. The limitations of the research context should be considered when extrapolating the findings to different projects. Although the implementation of the developed data-oriented concepts in Regesta Imperii during student workshops demonstrates their potential for broader project applicability, it is essential to further investigate and evaluate their effectiveness in diverse contexts. Future research should aim to explore the generalizability of certain parts and their limitations of data-oriented programming concepts across a wider range of digital edition projects.

Another limitation is the potential lack of diversity in the test cases used during the evaluation of the data-oriented programming concepts. The evaluation focused on application-level concepts and requirements, which may not cover the full range of scenarios and challenges encountered in the data-oriented development of digital editions. Therefore, the generalizability of the findings to a wider range of scenarios might be limited. To ensure a comprehensive understanding of how data-oriented programming performs in various contexts, it is necessary to incorporate a broader spectrum of test cases representing different data sets and project requirements.

Additionally, the research has primarily focused on the application-level concepts of data-oriented programming, and the architecture layer concepts have not been fully explored

or implemented in the projects. While the application-level concepts have demonstrated positive results, further investigation is needed to address the architecture-level concepts. Exploring how data-oriented programming can be integrated into the architectural design of digital edition platforms can provide valuable insights into scalability, technical constraints, and the overall effectiveness of data-oriented design at a higher level.

The handling of deep and complex data structures presents a potential limitation. While data-oriented programming emphasizes efficient data manipulation, the performance impact of working with heavily nested and intricate structures should be considered. Processing such structures may introduce computational challenges, potentially impacting the efficiency and responsiveness of the digital edition development process. Performance issues arising from the complexity of data structures need to be carefully addressed to ensure smooth execution of data-oriented programming concepts.

Lastly, human error is an inherent limitation in any development process, including the implementation of data-oriented programming concepts in the Digital Humanities. Despite the guiding principles of data-oriented programming, the design and implementation of digital editions involve human decision-making and interpretation. Mistakes or oversights during the development process, such as errors in code or data organization, can have an adverse impact on the effectiveness and accuracy of the digital editions. Rigorous quality assurance measures should be employed to minimize the potential impact of human error.

Considering these limitations, it is important to approach the utilization of data-oriented concepts in web technologies for the Digital Humanities with caution. Acknowledging the specific constraints of individual projects, incorporating a diverse range of test cases, addressing performance issues related to complex data structures, and implementing robust quality assurance processes can help mitigate these limitations and enhance the effectiveness of data-oriented programming in digital edition development.

## 7.4 Future Work

The utilization of data-oriented programming concepts in web technologies for software management and data accessibility in the field of Digital Humanities has generated valuable insights. However, several unresolved questions remain, calling for further investigation.

Firstly, it is important to consider the applicability of data-oriented programming concepts to different types of digital humanities projects beyond the specific use cases and project-specific requirements analyzed in this research. Exploring how these concepts can be adapted and applied to diverse digital edition development scenarios will provide a more comprehensive understanding of their potential benefits and limitations. Future research should aim to explore the generalizability of data-oriented programming concepts across a wider range of digital edition projects.

Secondly, while the application-level concepts of data-oriented programming have shown positive results, the exploration and implementation of architecture-level concepts have been limited. Investigating how the data-oriented paradigm can be integrated into the architectural design of digital edition platforms can provide valuable insights into scalability, technical constraints, and the overall effectiveness of data-oriented design at a higher level. This would contribute to a more holistic understanding of the benefits and challenges of adopting data-oriented programming concepts in the Digital Humanities.

Furthermore, addressing the handling of deep and complex data structures is crucial. While data-oriented programming emphasizes efficient data manipulation, the performance impact of working with heavily nested and intricate structures should be considered. Developing strategies to optimize the processing of such structures and mitigate potential performance issues will enhance the overall effectiveness and responsiveness of data-oriented programming in digital edition development.

In addition, incorporating a diverse range of test cases representing different data sets and project requirements is essential to ensure a comprehensive evaluation of data-oriented programming concepts. This will help uncover a wider range of scenarios and challenges encountered in the data-oriented development of digital editions, allowing for a more robust assessment of the generalizability and effectiveness of these concepts.

Finally, staying abreast of emerging trends and technologies in the field of Digital Humanities is crucial for the future application of data-oriented programming concepts. Integration of artificial intelligence and machine learning techniques, leveraging cloud computing and distributed systems, and adopting semantic web technologies and linked data principles are areas that hold promise for enhancing data analysis, information retrieval, and data interoperability in digital edition development. Exploring these

avenues will open up new possibilities for studying and analyzing cultural and historical information in the digital age.

In conclusion, while the research on data-oriented programming concepts in web technologies for software management and data accessibility in the Digital Humanities offers significant insights, there are unresolved questions, methodological refinements, alternative perspectives, and emerging trends that require further exploration. By addressing these areas, the field can advance and unlock new possibilities for the effective utilization of data-oriented programming in the study and analysis of cultural and historical information.





## Bibliography

- [Ada06] ADAMKÓ, Attila: UML-Based Modeling of Data-oriented WEB Applications. *J. Univers. Comput. Sci.* (2006), Bd. 12(9): S. 1104–1117
- [Ani14] ANICHE, Maurício F; OLIVA, Gustavo A und GEROSA, Marco A: Are the methods in your data access objects (DAOs) in the right place? A preliminary study, in: *2014 Sixth International Workshop on Managing Technical Debt*, IEEE, S. 47–50
- [Bar04] BARBOSA, Denilson; MENDELZON, Alberto O; LIBKIN, Leonid; MIGNET, Laurent und ARENAS, Marcelo: Efficient incremental validation of XML documents, in: *Proceedings. 20th International Conference on Data Engineering*, IEEE, S. 671–682
- [Bau11] BAUER, Florian und KALTENBÖCK, Martin: Linked open data: The essentials. *Edition mono/monochrom, Vienna* (2011), Bd. 710: S. 21
- [Bay22] BAYLISS, Jessica D: The data-oriented design process for game development. *Computer* (2022), Bd. 55(05): S. 31–38
- [Bri20] BRITO, Gleison und VALENTE, Marco Tulio: REST vs GraphQL: A controlled experiment, in: *2020 IEEE international conference on software architecture (ICSA)*, IEEE, S. 81–91
- [Bro02] BROWN, Alan; JOHNSTON, Simon und KELLY, Kevin: Using service-oriented architecture and component-based development to build web service applications. *Rational Software Corporation* (2002), Bd. 6: S. 1–16
- [Ced16] CEDERLUND, Mattias: Performance of frameworks for declarative data fetching: An evaluation of Falcor and Relay+ GraphQL (2016)
- [Cum13] CUMMINGS, James: The text encoding initiative and the study of literature. *A companion to digital literary studies* (2013): S. 451–476
- [Dah06] DAHL, Mark V; BANERJEE, Kyle und SPALTI, Michael: *Digital libraries: integrating content and systems*, Elsevier (2006)
- [Dar22] DARMAWAN, Irfan; RAHMATULLOH, Alam und GUNAWAN, Rohmat: Web Service Modeling for GraphQL Based College Data Service Access, in: *2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)*, IEEE, S. 1–6
- [Dau16] DAUGIRDAS, Kestutis: *Die Anfänge des Sozinianismus: Genese und Eindringen des historisch-ethischen Religionsmodells in den universitären Diskurs der Evangelischen in Europa*, Vandenhoeck & Ruprecht (2016)

- [Deu18] DEUTSCH, Alin und PAPAKONSTANTINOY, Yannis: Graph data models, query languages and programming paradigms. *Proceedings of the VLDB Endowment* (2018), Bd. 11(12): S. 2106–2109
- [DFB22] DE F. BORGES, Marcos V; ROCHA, Lincoln S und MAIA, Paulo Henrique M: MicroGraphQL: a unified communication approach for systems of systems using microservices and GraphQL, in: *Proceedings of the 10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, S. 33–40
- [Fab18] FABIAN, Richard: Data-oriented design. *framework* (2018), Bd. 21: S. 1–7
- [Fan09] FANG, Cheng: A new dao pattern with dynamic extensibility, in: *2009 Second International Conference on Information and Computing Science*, Bd. 1, IEEE, S. 23–26
- [Fed20] FEDOSEEV, K; ASKARBEKULY, N; UZBEKOVA, AE und MAZZARA, M: Application of data-oriented design in game development, in: *Journal of Physics: Conference Series*, Bd. 1694, IOP Publishing, S. 012035
- [Fla15] FLANDERS, Julia und JANNIDIS, Fotis: Data modeling. *A new companion to digital humanities* (2015): S. 229–237
- [Fla18] FLANDERS, Julia und JANNIDIS, Fotis: Data modeling in a digital humanities context: An introduction, in: *The Shape of Data in the Digital Humanities*, Routledge (2018), S. 3–25
- [Fou18] FOUNDATION, The GraphQL: GraphQL, <https://graphql.github.io/graphql-spec/June2018/> (2018)
- [Fur18] FURCHERT, Almut: Hildegard of Bingen. In *A Dictionary of Philosophy of Religion*, edited by Charles Taliaferro and Elsa Marty (2018), Bd. 125
- [Gar11] GARBATOV, Stoyan und CACHOPO, João: Data access pattern analysis and prediction for object-oriented applications. *INFOCOMP Journal of Computer Science* (2011), Bd. 10(4): S. 1–14
- [Göh04] GÖHLERT, Ines und HÖNIG, Timo: Open Source Content Management Systems for Small and Medium-Sized Enterprises (2004)
- [Gui17] GUIA, José; SOARES, Valéria Gonçalves und BERNARDINO, Jorge: Graph Databases: Neo4j Analysis., in: *ICEIS (1)*, S. 351–356
- [Har18a] HARTIG, Olaf und PÉREZ, Jorge: Semantics and complexity of GraphQL, in: *Proceedings of the 2018 World Wide Web Conference*, S. 1155–1164
- [Har18b] HARTINA, Dewi Ayu; LAWI, Armin und PANGGABEAN, Benny Leonard Enrico: Performance analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University, in: *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)*, IEEE, S. 237–240
- [Jam12] JAMIL, Hasan M: Design of declarative graph query languages: On the choice between value, pattern and object based representations for graphs, in: *2012 IEEE 28th International Conference on Data Engineering Workshops*, IEEE, S. 178–185

- [Jos07] JOSHI, Rajive: Data-oriented architecture: A loosely-coupled real-time soa. *Whitepaper, Aug* (2007)
- [Kuc] KUCZERA, Andreas: TEI Beyond XML—Digital Scholarly Editions as Provenance Knowledge Graphs
- [Kul12] KULAK, Daryl und GUINEY, Eamonn: *Use cases: requirements in context*, Addison-Wesley (2012)
- [Law21] LAWI, Armin; PANGGABEAN, Benny LE und YOSHIDA, Takaichi: Evaluating GraphQL and REST API Services Performance in a Massive and Intensive accessible information system. *Computers* (2021), Bd. 10(11): S. 138
- [lib12] Hildegardis Bingensis: Liber epistolarum, <https://liberepistolarum.mni.thm.de> (accessed 2023-04-12)
- [Llo11] LLOPIS, Noel und TOUCH, Snappy: High-performance programming with data-oriented design. *Game Engine Gems* (2011), Bd. 2: S. 251–261
- [lox] Smart Home Brochure, Online, URL <https://www.loxone.com/enen/info/smart-home-brochure/>, accessed 2023-04-24
- [Mal01] MALAN, Ruth; BREDEMEYER, Dana ET AL.: Functional requirements and use cases. *Bredemeyer Consulting* (2001)
- [Mat04] MATIC, Danijel; BUTORAC, Dino und KEGALJ, Hrvoje: Data access architecture in object oriented applications using design patterns, in: *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (IEEE Cat. No. 04CH37521)*, Bd. 2, IEEE, S. 595–598
- [Mei03] MEIER, Wolfgang: eXist: An open source native XML database, in: *Web, Web-Services, and Database Systems: NODe 2002 Web-and Database-Related Workshops Erfurt, Germany, October 7–10, 2002 Revised Papers 4*, Springer, S. 169–183
- [Mij19] MIJAC, Tea; JADRIĆ, Mario und ČAKUŠIĆ, Maja: In search of a framework for user-oriented data-driven development of information systems. *Economic and Business Review* (2019), Bd. 21(3): S. 6
- [Mos06] MOSELEY, Ben und MARKS, Peter: Out of the tar pit. *Software Practice Advancement (SPA)* (2006), Bd. 2006
- [Nei19] NEILL, Iian; KUCZERA, Andreas; LIEGEN BEI DEN AUTOREN, Medienrechte und ALLER VERWEISE, Letzte Überprüfung: Zeitschrift für digitale Geisteswissenschaften (2019)
- [Nei12] NEILL, Iian: Standoff Properties Editor, <https://github.com/argimenes/standoff-properties-editor> (accessed 2023-04-12)
- [Nel01] NELLHAUS, Tobin: XML, TEI, and Digital Libraries in the Humanities. *portal: Libraries and the Academy* (2001), Bd. 1(3): S. 257–277
- [Neo23] NEO4J: Neo4j Documentation, <https://neo4j.com/docs/> (2023)
- [Ngu14] NGUYEN, Hung Viet; NGUYEN, Hoan Anh; NGUYEN, Anh Tuan und NGUYEN, Tien N: Mining interprocedural, data-oriented usage patterns in JavaScript web applications, in: *Proceedings of the 36th International Conference on*

- Software Engineering*, S. 791–802
- [Noc04] NOCK, Clifton: *Data access patterns: database interactions in object-oriented applications*, Addison-Wesley Boston (2004)
- [Ore07] OREN, Eyal; DELBRU, Renaud; GERKE, Sebastian; HALLER, Armin und DECKER, Stefan: ActiveRDF: Object-oriented semantic web programming, in: *Proceedings of the 16th international conference on World Wide Web*, S. 817–824
- [Pie16] PIERAZZO, Elena: *Digital scholarly editing: Theories, models and methods*, Routledge (2016)
- [Pri17] PRISMA: Execution | GraphQL, <https://graphql.org/learn/execution/> (2017)
- [QM23] QUIÑA-MERA, Antonio; FERNANDEZ, Pablo; GARCÍA, José María und RUIZ-CORTÉS, Antonio: GraphQL: A Systematic Mapping Study. *ACM Computing Surveys* (2023), Bd. 55(10): S. 1–35
- [Red13] REDDY, Ch Ram Mohan und RAO, RV Raghavendra: QoS of Web service: Survey on performance and scalability. *Computer Science and Information Technology* (2013), Bd. 3(9): S. 65–73
- [Sha22] SHARVIT, Yehonathan: *Data-oriented programming unlearning objects*, Manning (2022)
- [Tos] TOSCHKA, Patrick; JAROSCH, Julian und KUCZERA, Andreas: The Socinian Correspondence: A Graph-Based Digital Scholarly Edition
- [VB14] VAN BRUGGEN, Rik: *Learning Neo4j*, Packt Publishing Ltd (2014)
- [Vog18] VOGEL, Maximilian; WEBER, Sebastian und ZIRPINS, Christian: Experiences on migrating RESTful web services to GraphQL, in: *Service-Oriented Computing–ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13–16, 2017, Revised Selected Papers*, Springer, S. 283–295
- [Wai15] WAIKAR, Manoj: *Data-oriented Development with AngularJS*, Packt Publishing Ltd (2015)
- [Wie] WIEGAND, Frank: TEI/XML Editing for Everyone’s Needs. *The Linked TEI: Text Encoding in the Web*: S. 231

## List of Figures

2.1	The Socinian Correspondence: Project Workflow. . . . .	6
2.2	Example of over- and under-fetching in REST & GraphQL [QM23]. . .	8
2.3	Principal components interaction of the GraphQL paradigm [QM23]. . .	9
2.4	GraphQL document definition [QM23]. . . . .	10
2.5	GraphQL type system example [QM23]. . . . .	14
2.6	GraphQL type system example [QM23]. . . . .	15
2.7	Response to mutation and query operations executed in Figure 2.6 [QM23].	15
2.8	An example of a labeled property graph in Neo4j. . . . .	17
3.1	Floorplan of a well-lit smart home [lox]. . . . .	22
3.2	Simple light switch. . . . .	23
3.3	Labeled light switch. . . . .	23
3.4	Principles of data-oriented programming: An Overview [Sha22]. . . . .	24
3.5	Class diagram of a fictitious library management tool [Sha22]. . . . .	27
3.6	Data-oriented architecture for loosely coupled applications [Jos07]. . . .	33
4.1	Project-oriented needs for an architecture layer: use case diagram. . . .	36
4.2	Project-oriented needs for an application layer: use case diagram. . . .	41
4.3	Concept: Processing of data-oriented requirements. . . . .	43
4.4	Concept: Service Access Patterns. . . . .	45
4.5	Concept: Service traversal with controller. . . . .	47
4.6	Concept: Example web application. . . . .	48
4.7	Concept: Cyclic data dependencies. . . . .	52
5.1	Placentinus: An interconnected annotation entity in the text view of the new Socinian Correspondence scholarly digital edition. . . . .	53
5.2	Scholarly Digital Edition: Components and specifics. . . . .	55
5.3	Unified Neo4j data model for the Socinian Correspondence. . . . .	58
5.4	The Socinian Correspondence: Extended view of node structures. . . . .	59
5.5	The Socinian Correspondence: Example of nested sections. . . . .	61
5.6	The Socinian Correspondence: Entity mentions listing. . . . .	62
5.7	The Socinian Correspondence: Communication view. . . . .	62
5.8	The Socinian Correspondence: Search for relations. . . . .	63



## Listings

2.1	Example code block in Cypher. . . . .	18
3.1	An example of object-oriented programming following this principle. . .	25
3.2	An example of functional programming following this principle. . . . .	25
3.3	An example of decoupling for isolated testing. . . . .	26
3.4	Using maps to create generic structures for players. . . . .	28
3.5	Improving data immutability with spread operations in JavaScript. . . .	29
3.6	Example implementation of a GraphQL schema. . . . .	30
4.1	Static method implementation in a Text Data Access Object. . . . .	51





## A Use Cases

<b>Use Case:</b>	Approve and publish new versions
<b>Description:</b>	This use case describes the process of a Publisher approving and publishing new versions of the scholarly digital edition.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Publisher
<b>Assumptions:</b>	<p>Publisher has access to the latest version of the edition.</p> <p>Publisher has authority to approve/publish new versions.</p> <p>New version has been reviewed and edited by the Editor.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. Publisher reviews new version of the edition.</li> <li>2. Publisher ensures that new version meets quality standards.</li> <li>3. Publisher approves new version for publication.</li> <li>4. Publisher publishes new version of an edition.</li> </ol>
<b>Variations:</b>	If the new version does not meet the quality standards, the Publisher sends it back to the Editor for revisions.
<b>Non-Functional Requirements:</b>	<p><b>Reliability:</b> The Publisher should be able to approve and publish new versions of the edition with 99.9% uptime.</p> <p><b>Security:</b> The new version should be published securely to prevent unauthorized access.</p> <p><b>Usability:</b> The approval and publication process should be simple and easy to use for the Publisher.</p>
<b>Issues:</b>	Complicated and time-intensive publishing process.

**Table A.1:** Use case: Approve and publish new versions.

<b>Use Case:</b>	Check status of published materials
<b>Description:</b>	This use case describes the process of checking the status of published materials in the scholarly digital edition.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Publisher, Editor, Maintainer
<b>Assumptions:</b>	<p>Publisher, Editor, and Maintainer have access to the latest version of the digital edition.</p> <p>Publisher, Editor, and Maintainer have permission to view the status of published materials.</p> <p>Publisher and Editor have certain methodologies to mark published materials with different statuses.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. Actor views the digital edition.</li> <li>2. Actor navigates to the section of the digital edition that displays the status of published materials.</li> <li>3. Actor reviews the status of the published materials.</li> <li>4. Actor takes any necessary action based on the status of the published materials and the role of the actor.</li> </ol>
<b>Variations:</b>	Actor can view a collection of published materials and their statuses.
<b>Non-Functional Requirements:</b>	<p><b>Reliability:</b> The status of published materials should be displayed accurately and reliably.</p> <p><b>Usability:</b> The process of checking the status of published materials should be simple and easy to use for all actors.</p>
<b>Issues:</b>	Different status codes per project.

**Table A.2:** Use case: Check status of published materials.

<b>Use Case:</b>	Export text, metadata and annotations
<b>Description:</b>	This use case describes the process of exporting text, metadata and annotations from the scholarly digital edition.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Publisher, Editor, Visitor
<b>Assumptions:</b>	<p>Publisher, Editor, and Visitor have access to the latest version of the digital edition.</p> <p>Visitors may export text, metadata, and annotations for scientific work, research, personal reference, or other purposes.</p> <p>Visitors may require the exported materials to be in a specific format to suit their needs.</p> <p>Visitors may export metadata to assist with citation and referencing.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. Actor navigates to the section of the digital edition that contains the text, metadata and annotations they want to export.</li> <li>2. Actor selects the text, metadata, and annotations they want to export.</li> <li>3. Actor exports the selected text, metadata, and annotations in the desired format.</li> </ol>
<b>Variations:</b>	<p>Visitors may export text, metadata, and annotations specifically for scientific work, research, personal reference, or other purposes.</p> <p>Visitors may export metadata to assist with citation and referencing.</p>
<b>Non-Functional Requirements:</b>	<p><b>Performance:</b> The export process should be completed in a reasonable amount of time.</p> <p><b>Usability:</b> The export process should be simple and easy to use for the actors.</p>
<b>Issues:</b>	Lack of consistency in versioning.

**Table A.3:** Use case: Export text, metadata & annotations.

<b>Use Case:</b>	View text, metadata and annotations
<b>Description:</b>	This use case describes the process of viewing text, metadata, and annotations in the scholarly digital edition.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Visitor, Editor
<b>Assumptions:</b>	The Visitor and Editor have access to the latest version of the digital edition.  The Visitor and Editor have permission to view text, metadata, and annotations.
<b>Steps:</b>	The Visitor or Editor navigates to the section of the digital edition that contains the specific text, metadata, and annotations they wish to view.  The Visitor or Editor views the text, metadata, and annotations in the digital edition.
<b>Variations:</b>	Visitor wants to view or search for a single text or different annotations.  Visitor wants to view all annotations in a text.  Visitor wants to view a collection containing all texts.  Visitor wants to view occurrences of annotations in different texts.  Visitor wants to search by text metadata.
<b>Non-Functional Requirements:</b>	<b>Usability:</b> The viewing process should be simple and easy to use for the Visitor and Editor.
<b>Issues:</b>	Recursive data relationships.

Table A.4: Use case: View text, metadata &amp; annotations.

<b>Use Case:</b>	Create automated workflows
<b>Description:</b>	This use case describes the process of creating automated workflows beyond the scholar edition, within the digital edition architecture.
<b>Source:</b>	Software Engineering Requirements
<b>Actors:</b>	Developer
<b>Assumptions:</b>	<p>The Developer has access to the digital edition architecture.</p> <p>The Developer has the necessary permissions and tools to create automated workflows.</p> <p>The Developer has the necessary knowledge to create automated workflows.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. The Developer navigates to the section of the digital edition architecture where they can create automated workflows.</li> <li>2. The Developer creates a new automated workflow or modifies an existing one according to the requirements.</li> <li>3. The Developer tests the automated workflow to ensure that it functions correctly.</li> </ol>
<b>Variations:</b>	None at this time.
<b>Non-Functional Requirements:</b>	<p><b>Scalability:</b> The automated workflow creation process should be scalable to accommodate an increase in the number of workflows or their complexity.</p> <p><b>Maintainability:</b> The automated workflow creation process should be easy to maintain and update.</p>
<b>Issues:</b>	Different workflows per project.

**Table A.5:** Use case: Create automated workflows.

<b>Use Case:</b>	Develop new features and enhancements
<b>Description:</b>	This use case describes the process of developing new features and enhancements in both the architecture and application layers of the scholarly digital edition.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Developer
<b>Assumptions:</b>	<p>The Developer has access to the digital edition architecture and application layers.</p> <p>The Developer has the necessary permissions and tools to develop new features and enhancements.</p> <p>The Developer has basic knowledge of programming to develop new features and enhancements.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. Developer identifies a new feature or enhancement required by the stakeholders.</li> <li>2. Developer creates a design document for the new feature or enhancement, describing the requirements, architecture, and implementation plan.</li> <li>3. Developer codes the new feature or enhancement, adhering to the design document.</li> <li>4. Developer tests the new feature or enhancement to ensure that it functions correctly.</li> <li>5. Developer deploys the new feature or enhancement to the digital edition architecture without any issues.</li> </ol>
<b>Variations:</b>	None at this time.
<b>Non-Functional Requirements:</b>	<p><b>Scalability:</b> The new feature or enhancement development process should be scalable to accommodate an increase in the number of features or their complexity.</p> <p><b>Maintainability:</b> The new feature or enhancement development process should be easy to maintain and update.</p> <p><b>Compatibility:</b> The new feature or enhancement should be compatible with existing software and architecture.</p>
<b>Issues:</b>	Specific knowledge is required for specific parts of a project.

Table A.6: Use case: Develop new features and enhancements.

<b>Use Case:</b>	Integrate support for third-party tools
<b>Description:</b>	This use case describes the process of integrating support for third-party tools in both the architecture and application layers of the scholarly digital edition. These third-party tools could be web services or other components that need to be easily integrated into the digital edition.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Developer
<b>Assumptions:</b>	<p>Developer has access to the digital edition architecture and application layers.</p> <p>Developer has the necessary permissions and tools to integrate third-party tools.</p> <p>Developer has basic knowledge of programming to integrate third-party tools.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. The Developer identifies a new third-party tool required by the stakeholders.</li> <li>2. The Developer evaluates the tool to ensure it meets the digital edition's requirements and integrates seamlessly into the architecture and application layers.</li> <li>3. The Developer integrates the third-party tool into the digital edition architecture and application layers.</li> <li>4. The Developer tests the integration to ensure that it functions correctly.</li> <li>5. The Developer deploys the third-party tool integrated digital edition.</li> </ol>
<b>Variations:</b>	None at this time.
<b>Non-Functional Requirements:</b>	<p><b>Scalability:</b> The integration process should be scalable to accommodate an increase in the number of third-party tools.</p> <p><b>Maintainability:</b> The integration process should be easy to maintain and update.</p> <p><b>Compatibility:</b> The third-party tool should be compatible with existing software and the digital edition architecture.</p>
<b>Issues:</b>	Tightly-coupled components in different projects.

**Table A.7:** Use case: Integrate support for third-party tools.

<b>Use Case:</b>	Backup and restore data
<b>Description:</b>	This use case describes the process of backing up and restoring data in the scholarly digital edition. The process is critical to ensure that data is not lost in the event of a system failure or data corruption.
<b>Source:</b>	Software Engineering Requirements
<b>Actors:</b>	Maintainer
<b>Assumptions:</b>	The Maintainer has the necessary permissions and tools to perform data backup and restore.  The digital edition has a backup and restore system in place.
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. The Maintainer initiates the data backup process at regular intervals or as needed.</li> <li>2. The backup system creates a copy of the data and stores it in a secure location.</li> <li>3. The Maintainer tests the backup to ensure that the data can be restored in the event of a system failure or data corruption.</li> <li>4. If data needs to be restored, the Maintainer initiates the data restore process.</li> <li>5. The restore system retrieves the backup data and restores it to the digital edition.</li> </ol>
<b>Variations:</b>	None at this time.
<b>Non-Functional Requirements:</b>	<p><b>Reliability:</b> The backup and restore system should be reliable and ensure that data is not lost in the event of a system failure or data corruption.</p> <p><b>Performance:</b> The backup and restore process should be optimized to ensure that it completes within a reasonable amount of time.</p>
<b>Issues:</b>	None at this time.

**Table A.8:** Use case: Backup and restore data.



<b>Use Case:</b>	View project description and details
<b>Description:</b>	This use case outlines the process for Visitors to access the project description and details for the scholarly digital edition. The process enables Visitors to learn about the project's scope, purpose, and objectives, as well as any other relevant information.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Visitor
<b>Assumptions:</b>	The digital edition has a project description and details section.  The Visitor has a device with an internet connection and a web browser.
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. The Visitor navigates to the project description and details section of the digital edition.</li> <li>2. The digital edition displays the project description and details, including its purpose, scope, and objectives.</li> <li>3. The Visitor can access additional information related to the project, such as project team members, timeline, and financial support sources.</li> </ol>
<b>Variations:</b>	Variations pending on different project requirements.
<b>Non-Functional Requirements:</b>	<p><b>Usability:</b> The digital edition should be user-friendly and easy to navigate to ensure that Visitors can access the project description and details without difficulty.</p> <p><b>Availability:</b> The digital edition should be able to provide project description and details when needed.</p>
<b>Issues:</b>	None at this time.

**Table A.9:** Use case: View project description & details

<b>Use Case:</b>	View project in different languages
<b>Description:</b>	This use case outlines the process for Visitors to view the scholarly digital edition project in different languages. The process enables Visitors to access the project in their preferred language.
<b>Source:</b>	Project Requirements
<b>Actors:</b>	Visitor
<b>Assumptions:</b>	<p>The digital edition supports multiple languages.</p> <p>The Visitor has a device with an internet connection and a web browser.</p> <p>The Visitor is aware of the available languages for the project.</p>
<b>Steps:</b>	<ol style="list-style-type: none"> <li>1. The Visitor navigates to the digital edition project page.</li> <li>2. The digital edition displays the project page in the default language.</li> <li>3. The Visitor selects a preferred language option from the language menu.</li> <li>4. The digital edition displays the project page in the selected language.</li> </ol>
<b>Variations:</b>	None at this time.
<b>Non-Functional Requirements:</b>	<p><b>Usability:</b> The language menu should be easy to find and use, ensuring that Visitors can easily switch between languages.</p> <p><b>Availability:</b> The digital edition should be able to provide different languages.</p>
<b>Issues:</b>	None at this time.

Table A.10: Use case: View project in different languages