Tampere University

Hung Anh Pham

DEVELOPMENT OF DATA ADAPTERS TO SUPPORT SERVER-SIDE DATA PROCESSING IN WEB APPLICATIONS

Bachelor of Science Thesis
Faculty of Information Technology and Communication Sciences
Supervisor: Prof. Kari Systä
December 2021

ABSTRACT

Hung Anh Pham: Development of data adapters to support server-side data processing in web applications
Bachelor of Science Thesis
Tampere University
International Bachelor Programme in Science and Engineering
December 2021

The age of fast Internet, smart portable devices, instant communication and services has ushered a tremendous rise of web applications, thanks to their cross-platform nature and ever-expanding capabilities. Alongside this growth is the rising amount, significance, and varieties of data and its purposes. In many software projects, data from multiple sources has to be served in different front end clients with different use cases, which complicates the development of such projects. This research studies and implements a data processing layer called the data adapter, in order to solve that challenge and prevent other issues regarding scalability, maintainability, and application performance.

Data adapters provide uniform interfaces to communicate with the databases as well as the front end clients, alongside a processing unit whose purpose is to transform raw data from databases into suitable formats to be used directly on the front ends. In this thesis, a data adapter is designed and built for an existing, in-development project, as part of its data pipeline. As this project is yet to have a sufficiently functioning front end application at the time this thesis is written, a front end is also implement to test the functionality of the data adapter.

The data adapter implementation successfully provides the data pipeline with the ability to handle the two-way synchronisation of data from multiple sources to multiple front ends. Observations gathered during this process also reveal the importance of architectural and technological choices, which can significantly improve or impede the functionality, flexibility and scalability of the adapter and the project as a whole. Since the data pipeline and front end client of the project, at the time of writing, are not fully implemented, the data adapter is yet to handle the amount and complexity of data it is designed to do. However, the preliminary results realised in this thesis signal that as the application scales, data adapters will be able to achieve their full potential and play a vital role in the system.

Keywords: web applications, data adapters, data processing

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

PREFACE

This thesis mainly discusses data adapters and their roles in the architecture of a web application. To better assess their impacts, a data adapter will be implemented for an existing application, and its performance will be tested.

I would like to express my gratitude to my supervisor, Professor Kari Systä for his guidance and support during the process of my research and writing. I would also like to thank Ville Heikkilä for providing me with the necessary tools and knowledge to conduct the research implementation and experiment.

Tampere, 3rd December 2021

Hung Anh Pham

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface, an interface that allows multiple applications or software to interact and communicate with one another
DOM	Document Object Model, a standard interface that describes the HTML document as a tree structure
Git	An open-source version control system that helps developers efficiently coordinate, collaborate, and manage projects
REST	Representational state transfer, a set of architectural constraints for APIs. There are multiple criteria, for example stateless client-server communication, interface uniformity, etc.
SPA	Single page application, a type of web application that only downloads one single HTML page alongside other required assets that allows it to run on a web browser
UI	User interface, where users can see and interact with the application. With regards to web applications, the user interface is what users are seeing on the screen
URL	Uniform Resource Locator, the web address that points to a web resource on a network, or the Internet

1. INTRODUCTION

Web applications are playing an increasingly larger role in the way people use the Internet. As more companies are looking for ways to increase their presence on the Web and digitalise their products and services, the demand for reliable, performant web applications is currently on the rise. Couple that with the increasing amount of data collected that needs to be utilised to better serve customers, a big challenge in web applications development is to design an architecture that could process large amounts of data without compromising the usability of the application.

While web applications may differ in their scale, purpose, or development stack, the architecture of a standard web application will have three main parts: the front end, the back end, and the database. Each of these parts may function differently and have slightly different roles from application to application, but the general purposes are largely the same. The front end is where users interact with the application through their devices. The back end handles the business logic and user requests, acting as a bridge between the front end and the database. The database, as its name suggests, stores the data that the application needs to function properly. The database, back end and front end correspond respectively to the data tier, application tier, and presentation tier of the three-tier architecture model [1]. Designing an appropriate architecture and optimising the aforementioned parts is paramount to the performance and stability of web apps, especially in the presence of large amounts of data.

Processing data fetched from multiple sources, to be served on multiple front ends poses a challenge to web applications development that requires conscious decisions on architectural designs and implementation. This thesis focuses on developing data adapters as a server-side data processing solution which solves that challenge. Its roles, impacts, potential benefits and drawbacks in web applications will be discussed from multiple angles. The main research task of this thesis will concern designing and implementing such a data adapter in an existing application.

The remaining chapters are structured as follows. Chapter 2 talks in depth about data adapters in the context of server-side data processing for a web application. Chapter 3 provides an architectural overview and the development stack of a project which will benefit from the implementation of data adapters. The process of designing and implementing a data adapter for this project will be discussed in Chapter 4. To test this data adapter, a front end application will also be developed, and the process is detailed in Chapter 5. Observations from this experiment will lead to general conclusions on the impacts of data adapters, alongside important things to take into account when designing and implementing them - all of which will be discussed in Chapter 6.

2. DATA PROCESSING AND DATA ADAPTERS

A web application requires the data stored in the database to work correctly, however it usually directly cannot use the raw data from the database. This data has to be processed, transformed, filtered, aggregated, etc. so that it exists in suitable formats that the application can use, and only the relevant data is included. Processing data can be done on either the client or the server, or in most cases, a combination of both.

2.1 Server-side processing and client-side processing

To clearly highlight the differences between these client-side and server-side processing, let us have an example of a web application that will be run on a mobile phone browser. In this example, the client is the mobile web browser running on that mobile phone, being served the application by a server managed by an external service that the application is hosted on. If data processing is done on the client, all necessary data has to be fetched from the server and then transformed into suitable formats through a sequence of tasks. The code that performs such tasks is run inside the mobile browser, and the performance of the operations depend on the processing power and system resources of the mobile phone [2]. This also means that different front ends, with different use cases and presentations, requiring the same data in different formats, will have to develop their own data models and interfaces. Two problems can already be identified: a performance issue caused by heavy calculations in the front ends, and an inconsistency issue where each front end (which is supposed to only handle presentation and user interaction) has to develop its own data model and interface, significantly increases development time and effort across multiple front-end clients; while decreases the scalability of the system [3].

Server-side data processing shifts the responsibilities above away from the client device to the server. Instead of forwarding the data directly from the database to the mobile client, the server would first process the data, transform it into suitable formats that the clients can use. After this is completed, the server will send the transformed, ready-to-use data to the different clients, which can then use it without further processing. The server is now responsible for taking into account multiple front ends and serving the corresponding data formats. The computing power of the server is also responsible for the processing performance [2]. This is better compared to the client-side alternative: developers can configure the performance and availability of the server much more easily then relying on client devices [4]. Letting the server taking care of data processing also provides a strong separation of roles between the server and the client: the server handles the logic and processing, while the client focuses on presentation and user interactions.

Most web applications adopt a hybrid method of data processing which performs these tasks on both the server and the client. In this way, developers can take advantage of the server-side resource and independence, while limits the number of requests to and responses from the server so that smaller, lightweight data transformation triggered by the user's inputs can be easily done within the browser [5]. There is no golden rule that dictates how much should the server or client handles data processing, since each application is different, and this separation of roles depends purely on how the application is designed. However, the server is generally responsible for the heavy lifting of data processing due to the performance reliability compared with the client, transforming and adapting the data to formats that different clients can use without high levels of discrepancies and data models. This is where data adapters come into the picture.

2.2 Data adapters to support server-side processing

As its name suggests, data adapters actively adapt the data they receive for the front end clients to use through processing operations running on the server. They bridge the database - whose main role is to store data, and the front end - whose main purpose is to handle user interactions through an interface. Data adapters shall therefore be part of the back end, or the application tier in a three-tier architecture [1]. This clear separation of duties between architectural components is a clear benefit for developers in terms of maintainability and scalability, as well as application performance reliability on a wide range of different client devices [1, 3, 4].

A data adapter shall accept incoming data from multiple sources, regardless of the syntactic data formats, or any other details specific to the source databases. The adapter outputs transformed and unified data - compiled from the aforementioned sources - in multiple different formats that are compatible with the front ends that need it, regardless of the front-end client's implementation details. In order to make this happen, the raw data from data sources need to be transformed in specific ways to arrive at the front-end-specific formats. Data from different sources can be grouped together, filtered, labeled, mapped, etc. inside the adapter to achieve the desired outcome.

The key difference between data adapters and a regular logic layer inside the back end should be highlighted: data adapters are specifically designed and implemented to handle data coming from different sources, transform it, and deliver it in different formats to different front end clients. While in simpler applications, the back end might only have to work with data coming from one source, built with one front end client in mind; data adapters need to be more flexible and resilient to the technological variety of both sides. This two-way adaptation is the crucial role of data adapters.

2.3 Data adapters components

With its purpose made clear, the design and implementation of such data adapters can now be discussed. Of course, details in these regards depend on the applications themselves: their development stacks, functionalities, scales, customers, etc. but certain high-level criteria must be fulfilled for the adapters to work properly with other components of the system. The more in-depth discussions regarding the characteristics of the application will be conducted in later chapters.

Looking at other works concerning data adapters reveals possible solutions to the challenge of designing them. Y.-T. Liao et al. [6] developed data adapters for querying and transforming data between databases of different types (RDB and NoSQL), both of which are used simultaneously in applications. The researchers focused on developing a general SQL interface to access different databases, which consists of query parsers, translators and connectors; alongside a database (DB) Converter, which takes care of database transformation with table synchronisation, allowing the application to handle data from the NoSQL databases without having to convert the queries itself. While it is clear that the use case of this data adapter is very different from ours, its design gives insights into what our adapter needs: an interface to communicate with other components, and a "converter" which processes the data.

Firstly, an interface for both the database and the front end client is required in the data adapter. A connection has to be established from the adapter to the database in order for the raw data to be queried, fetched, and possibly sent back to be stored. Connections to multiple databases might also be required, due to the role of the adapter to handle data from multiple sources, that the front end can use. Similarly, the data adapter has to allow connections from front end applications that enable them to retrieve the transformed data they require. This can be done by exposing an API which could have, for example, different endpoints for different data formats that the adapters can produce [7].

Secondly, data adapters need a component responsible for converting, or processing the data into suitable formats. This unit shall sit between the two interfaces above, as its job is to receive raw data from the database-facing interface, adapts it, then delivers to clients through the client-facing interface. Important technological choices have to be made for its implementation, as the languages and tools used depends heavily on the nature of the databases, the raw data formats, and the front ends. Performance will also have to be considered, as it directly impacts data availability and costs in the long run [4].

Based on the nature of different applications, other characteristics might also be required from the data adapter. For example, if the front end is a monitoring service showing graphs of an ongoing process that changes frequently, the adapter may continually query the database and emit frequent new data to the subscribed clients so that the data shown on the graphs is up-to-date. For larger applications, the adapter must also be elastic and scalable to handle an increasing amount of traffic without causing delays that affect the performance of the app. These topics, however, are beyond the scope of this thesis.

3. OVERVIEW OF THE RESEARCH PROJECT - VISDOM

This research aims to design and implement a data adapter for an existing web application, in order to clarify and possibly quantify the benefits that data adapters bring to the application. A data adapter will be implemented as part of the Data Management System for a project called VISDOM [8]. This in-development project offers users configurable dashboards that show data visualisations of multiple software engineering related tools and resources to different parties involved. VISDOM, therefore, will have to process large amounts of data from multiple sources, and transforms them into different formats which suit various kinds of visualisations - which are tasks that data adapters can help tremendously. As explained in the previous chapter, designing and implementing a data adapter has to depend on the architecture of the app, the overall architecture of VISDOM would now be discussed.

3.1 Overall architecture

From a high-level view, the architecture of the VISDOM project has three main components: data sources, a data management system, and visualisations. These components, as part of the whole technology value chain, are depicted in Figure 3.1

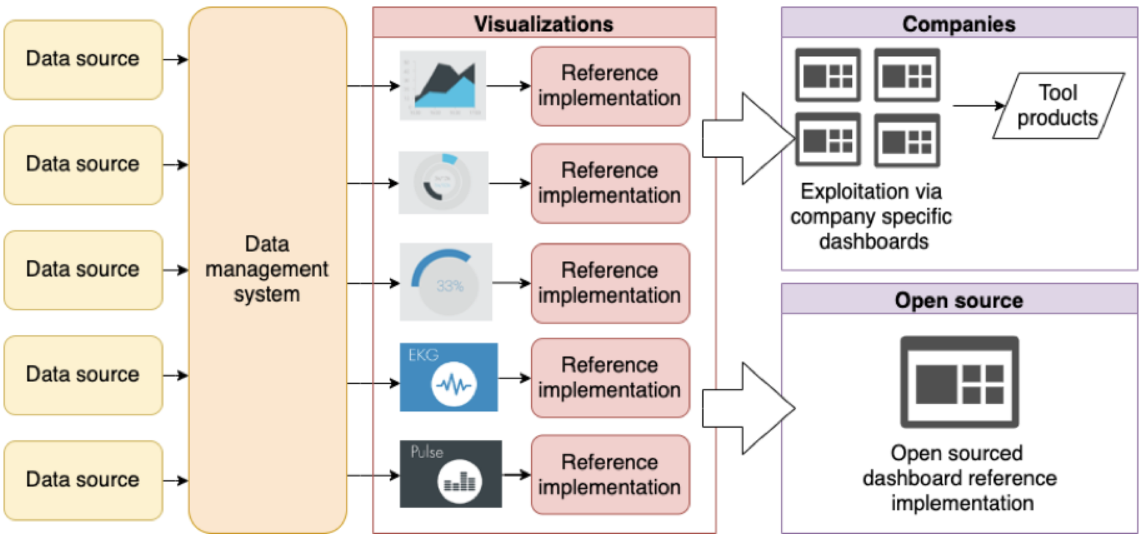


Figure 3.1. *The technology value chain of the VISDOM project*

The data sources are software engineering tools (Trello, GitHub, etc.) and code repositories, which contains different kinds of information relevant to the development of software projects. The Data Management System (DMS) takes care of collecting data from these sources, merging, linking, and storing them. Some required processing and transformations of this data will also be done by the DMS. The processed data will then be used by the third component: visualisations, displaying the data on interactive charts, graphs, etc. These visualisations are dynamic, and respond to a wide range of user inputs, from zooming in and out to selecting what properties and attributes should be displayed.

Among these components, the Data Management System is the main concern of this thesis. This is where server-side data processing is carried out, and this is where the data adapters should be implemented. Therefore, understanding different parts of its architecture is vital to the implementation of the data adapters.

3.2 Data management system architecture

Figure 3.2 illustrates different components that form the Data Management System and how they work together to perform the DMS' roles. There are four main parts: data fetchers, database, data adapters, and data broker.

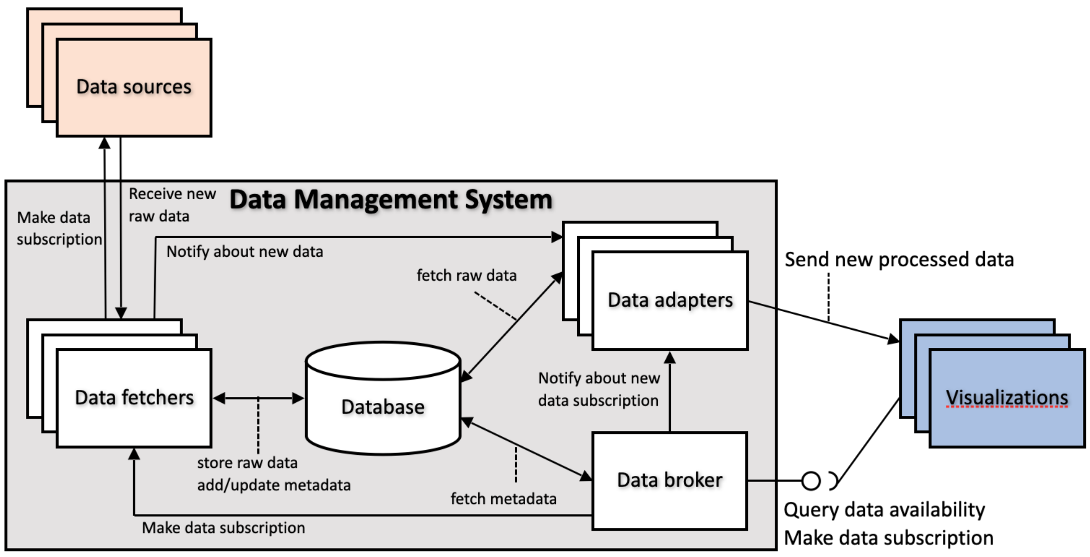


Figure 3.2. *VISDOM Data Management System architecture*

The responsibilities of each part can be inferred from their respective names. Data fetchers subscribe to the data sources and fetch data from them, which will be stored in the database. Data fetchers would also send notifications when new data is fetched to the data adapters. Data adapters are in charge of fetching raw data from the database, process and transform them from different formats to suitable, uniform ones that can be used in different visualisations. The last piece of the puzzle - data brokers - subscribe to data fetchers, pull metadata stored in the database and notify data adapters of new data subscriptions. The purpose of data brokers is to provide possible queries that visualisations can use with data adapters, information on the availability of data adapters and fetchers, monitoring them and restarts them when problems are detected.

In Figure 3.2, while there are only one database and one data broker, there are multiple data fetchers and adapters. The DMS was designed this way so that each data source has one dedicated data fetcher - this goes a long way in terms of maintainability and scalability, as each data source has different methods of accessing and querying, and developers can prevent multiple adapters experiencing issues when a single data source has problems. Data adapters, while only has one single source - the database itself - may also be split according to where the data they process came from. For example, the front end application needs data from GitHub and Trello, but the two platforms stored data in very different formats. Two separate data adapters can be designed so that one would be technologically compatible with GitHub raw data, while other specialises in working with Trello's data formats. The group of data adapters will then work with each other to provide a uniform interface from which data, regardless of its source, can be easily queried.

3.3 Database and visualisations technological stack

The next step before diving into developing data adapters is to understand the technologies used to build the database and Visualisations - two main components which data adapters directly communicate with. Data fetchers and data brokers, while also communicate with data adapters through notifications, are beyond the server-side data processing focus of this thesis. Keep in mind that at the time of writing, the development of VISDOM Data Management System is still in its early stages, therefore the current implementation might be lackluster compared with the specifications listed in the previous section.

The current database is built using MongoDB, a document-oriented database solution that stores data records as BSON documents - the binary representation of JSON documents [9]. MongoDB naturally supports JSON, and this is the format of data queried from MongoDB. Data records are stored in collections, which in turn are stored in databases. A MongoDB instance can have multiple databases for different purposes [10]. The database instance of this project has multiple databases, consisting of databases to store metadata, configurations, etc. alongside other databases, each of which stores data from a separate data source. For example, the database directly involved in this thesis is the `gitlab` database, storing data pulled from certain repositories hosted in Tampere University's GitLab. Inside this `gitlab` database contains multiple collections, for instance `commits`, which stores documents with information related to different commits; and `files` - storing information related to the files hosted in GitLab repositories.

The Visualisations are currently build using React - a front end library that quickly allows developers to quickly develop front-end, component-based web applications in a declarative manner [11]. The current proof-of-concept application is called Visu-App, primarily focused on displaying data from programming-related courses in Tampere University, which utilizes data from the University's GitLab and Plussa grading system. Visu-App has different charts and graphs that display such data, which accepts users' inputs to filter displayed data as they wish.

Both database and the visualisation application (as well as the data adapter) will run in their own separate Docker containers, which can communicate with each other through ports inside a VISDOM's Docker network. For this thesis, the current database will be used as the source for the data adapters, due to MongoDB's ease-of-use and the fact that the instance is already performing well at this development stage. Only data from the `gitlab` database is fetched, due to the fact that it is the only database that was sufficiently developed at the time of writing. In the future, as more databases are created inside the MongoDB instance, they can be connected to the data adapter to increase the variety of available data. The VISDOM data adapter that will be implemented in this thesis will therefore take scalability into account to prepare for the future. On the front-end side, Visu-App unfortunately cannot yet be utilised to test the data adapter, as its visualisations also require data from the Plussa grading system, which is not yet present in the MongoDB instance. Therefore, this research will build a simple React application with different graphs to visualise the GitLab data, in order to test the functionality of the data adapters.

With the DMS architecture and relevant techonological stack explained, the design and implementation of data adapters for VISDOM can be finally discussed. Chapter 4 will go in great details regarding this topic.

4. IMPLEMENTATION OF A DATA ADAPTER FOR VISDOM

From the discussions in previous chapters regarding possible components of a data adapter and the nature of VISDOM databases and front-end client, a more detailed adapter architecture can be formed - this is illustrated in Figure 4.1. One would now be able to choose the suitable technological stack to utilise. This part of the thesis elaborates on the criteria of the adapter, rationalises the choices of tools base on them, as well as provides a walk-through of how the adapter is implemented.

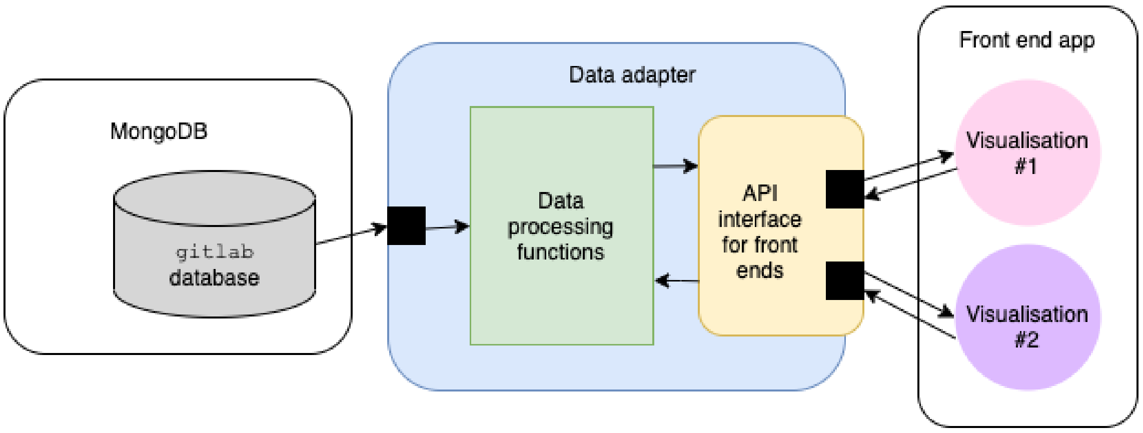


Figure 4.1. *Data adapter components and its connection to the VISDOM database and front end*

4.1 Development criteria and technological stack for the data adapter

Like other components in the VISDOM Data Management System, the Data adapter has to run in its own independent Docker container, and connected through other containers through open ports inside the VISDOM Docker network. Docker and Docker Compose will be needed to containerise the adapter, using a Dockerfile and Docker Compose YAML document - both declaring certain necessary configurations for the operations [12]. These configurations will include environment variables, such as the host, port and other credentials to connect to the MongoDB database.

The adapter itself will be developed as a NodeJS application, communicating with the database thanks to the help of Mongoose. Mongoose is an Object Data Modeling for MongoDB and NodeJS that helps us manage the data schema, relationships between them, and translation between JavaScript objects inside NodeJS and their respective representations inside the MongoDB databases [13]. The NodeJS application will run as a server, listens on port 4000 inside the Docker network. This port is also exposed, meaning the adapter can be reached from outside the VISDOM Docker network (from the local web browser, for example). This is very helpful in testing the data adapter during development.

From the front end client's point of view, besides exposing its port, the adapter also needs to expose an API that the client can call to fetch processed data from. As the data adapter has to be suitable with different visualisations by providing them with a uniform interface, its API needs to have different endpoints, one for each visualisation. By sending a GET request to a particular endpoint that corresponds to a visualisation, the client will receive the data object containing the data that can be used to draw the chart. To implement this API, Express - a popular web framework [14] will be used. The choice of this framework comes from the MERN (MongoDB, Express, React, Node) stack, a standard technological stack for web applications composed of the listed frameworks [15]. Taking advantage of the MERN stack means that developers have the chance to develop a full-stack application using platforms built on JavaScript and JSON - the representation of data stored in the database [16].

The last part of this section discusses what endpoints should be available on the data adapter interface. To define them, one has to look at what the front end client requires. For this thesis, a React front end application is developed to display the following two different visualisations:

- Visualisation A: A bar chart showing the number of commits made to a project each week, each bar is divided into color blocks, each block tells who the committer is.
- Visualisation B: A line graph describing the number of weekly commits made to different projects by the same author, each line represent a project.

At least two endpoints will therefore be available - one for each visualisation. Both visualisations will allow customisations that users can make from the front end client. Users can choose the time range the graphs cover, the authors/projects to be displayed, as well as whether other details related to each commit should be visible. These endpoints would therefore accept parameters such as time range, project name, and author name as query parameters; which would enable the front end to receive the data it needs.

4.2 Implementation steps

The following implementation steps serve as a brief documentation of how the adapter was implemented, not a step-by-step programming tutorial. The steps discussed in this section would generally follow a high-level fashion, without going into details of every step such as how to initialise a NodeJS application, how to install packages, etc. On the other hand, processes that are directly relevant to implementing the data adapter, like defining the data schema, connecting to database, and implementing data processing functionalities, will receive the focus.

4.2.1 Setting up the adapter

As mentioned in the previous section, the NodeJS application requires environmental variables declared in its containerisation configurations. The variables used in this thesis' implementation are listed in Listing 4.1 as key-value pairs. They are, of course, just examples; the exact values depend on the real-life use cases and the developers.

```
- APPLICATION_NAME = visdom-gitlab-adapter
- ADAPTER_NETWORK = visdom-network
- HOST_NAME = localhost
- HOST_PORT = 4000
- MONGODB_HOST = visdom-mongodb
- MONGODB_PORT = 27017
- MONGODB_USERNAME = harrypham
- MONGODB_PASSWORD = harrypham
- MONGODB_DATABASE = gitlab
- MONGODB_METADATA_DATABASE = metadata
```

Listing 4.1. *Environmental variables passed to data adapter Docker container*

The purpose each variable could be explained quite well from its name. `APPLICATION_-NAME`'s value is used as the name of the Docker container, while `ADAPTER_NETWORK` is the name of the Docker network which the data adapters, fetchers and databases are on - either an existing Docker network, or one that Docker will create. `HOST_NAME` and `HOST` define where actors outside the Docker network can access the application running inside the container. The remaining variables, whose names all start with `MONGODB_` are needed to connect to the database. The application uses `MONGODB_HOST` and `MONGODB_PORT` to form the connection address, where `MONGODB_DATABASE` is the name of the database that will be connected to. To authenticate itself, the app uses `MONGODB_USERNAME` and `MONGODB_PASSWORD` to authenticate through `MONGODB_METADATA_DATABASE`.

When a NodeJS application is run, the entry point - in this case, `index.js` - is run. Therefore, from the entry point file, all necessary steps in setting up the data adapter have to be initialised. The major steps include building the database interface by connecting to the database and defining the data schema, building the front end interface by setting up Express server and data routers, and implementing the data processing functions. Listing 4.2 shows an `index.js` file with placeholder comments for each step.

```
// index.js
require('dotenv').config(); // use dotenv to access environment variables

// STEP 1. CONNECT TO MONGODB DATABASE, DEFINE DATA SCHEMA

// STEP 2. INTERFACE FOR FRONT END: SETTING UP EXPRESS SERVER AND DATA ROUTER

// STEP 3. DATA PROCESSING FUNCTIONS
```

Listing 4.2. *Data adapter entry point with placeholder comments*

4.2.2 Connecting to MongoDB database and defining source data schema

Connecting to MongoDB inside the NodeJS application is done with the help of Mongoose. Listing 4.3 shows how to pass database credentials to Mongoose to establish the connection.

```
// connection.js
const mongoose = require("mongoose");
const url = `mongodb://${process.env.MONGODB_HOST}:${process.env.MONGODB_PORT}/
            ${process.env.MONGODB_DATABASE}`;

const connectMongo = () => {
 return mongoose.connect(url, {
    user: process.env.MONGODB_USERNAME,
    pass: process.env.MONGODB_PASSWORD,
    authSource: process.env.MONGODB_METADATA_DATABASE,
    useNewUrlParser: true,
    useUnifiedTopology: true
  });
};
module.exports = connectMongo;
```

Listing 4.3. *Connect to MongoDB inside NodeJS using Mongoose*

The code in Listing 4.3 constructs a URL that can be used to access the `gitlab` MongoDB database inside the Docker network. Authenticate is done in the `connectMongo()` function, where the username, password and the authentication database are given as parameters. This function is then used inside the application entry point `index.js` to connect to MongoDB.

To work with data objects inside MongoDB, its schema has to be defined. A Schema is basically a description of how the data looks like: what fields/attributes are included inside each data packet, and their types. As this data adapter works with raw commit objects in the MongoDB instance, the Commit schema is defined, which can be seen in Listing 4.4

```
// schema/commit.js
const mongoose = require('mongoose');
const { Schema } = mongoose;

const CommitSchema = new Schema({
  id: String,
  created_at: Date,
  title: String,
  message: String,
  author_name: String,
  author_email: String,
  committer_name: String,
  committer_email: String,
  committed_date: Date,
  // ... and other fields
}, { collection : 'commits' });

const Commit = mongoose.model("Commit", CommitSchema);
module.exports = Commit;
```

Listing 4.4. *Defining the Schema of the raw Commit data object inside the MongoDB instance*

All fields inside a Commit object and their corresponding types are declared. The collection that commit objects are stored in - collection (`commits`) - is also given when creating Schema, so that Mongoose knows where to look for the objects. In the end, a Commit model is created, using a given name `Commit` and the defined Schema. This model can be used to search and manipulate Commit objects in the database, thanks to a wide range of available Mongoose methods.

4.2.3 Initialising the interface for front ends

After establishing configuring the NodeJS application to work with the MongoDB database, it is time to initialise the Express application. This is a very simple process with a few lines of code, as shown in Listing 4.5. The Express server will listen on port number `4000` for incoming HTTP requests. The next step is to implement the data router (or data controller), which effectively provide endpoints the front end interface, listen for requests and respond with the processed data. Listing 4.6 shows how it can be done.

As its name suggests, the data router handles and responds to client requests, using a callback function. In the Listing, the router responds to a `HTTP GET` request directed at the `/commits` endpoint by fetching all commit objects and return them, with HTTP Status Code 200 (Success). If this router needs to be expanded with more endpoints, HTTP Methods, and callback handler functions in the future, it could be done similarly. This

```
//index.js

require('dotenv').config();
const connectDb = require("./src/connection");
const express = require('express');
var cors = require('cors'); // enable Cross-origin resource sharing

const app = express();
app.use(cors())

const port = process.env.HOST_PORT || 4000;

// Connect to MongoDB
connectDb().then(() => {
  console.log("MongoDb connected");
}).catch((err) => {
  console.log(err)
});

// Express application listens for HTTP requests
app.listen(port, () => {
  console.log(`Server listening at ${port}`)
});
// Implement data routers

// STEP 3. DATA PROCESSING FUNCTIONS
```

Listing 4.5. *Setting up Express server*

```
// src/controllers/gitlab.js

const gitlabRouter = require("express").Router();
const Commit = require("../schema/commit");

// GET request to /commits
gitlabRouter.get("/commits", async (req, res) => {
  const commits = await Commit.find({});
  res.status(200).json(commits);
})

module.exports = gitlabRouter;
```

Listing 4.6. *Implement commit data controller as Express routers*

router is then imported to `index.js` - shown in Listing 4.7, in turn used as a callback function that takes care of requests coming to the Express application's / endpoint.

At this point, the two interfaces are now initialised. The next steps is to extend the front end interface by providing more endpoints for specific visualisations, and implementing the data processing functions that transform raw JSON data objects from MongoDB into the formats which can be directly used by different visualisations. This requires looking at the front end, understanding how the visualisations will be built, and constructing the

```
//index.js
// ...
const gitlabRouter = require('./src/controllers/gitlab');

const app = express();
app.use(cors())

const port = process.env.HOST_PORT || 4000;

// ...
// Takes care of requests hitting /
app.use("/", gitlabRouter);

app.listen(port, () => {
  console.log(`Server listening at ${port}`)
});

// STEP 3. DATA PROCESSING FUNCTIONS
```

Listing 4.7. *Implement commit data controller as Express routers*

data formats they need.

4.2.4 Specifying front-end data formats

To be able to handle different types of visualisations that use different data formats, a dynamic, versatile, feature-rich yet non-verbose charting solution is needed. Since this system, from the database to the front end, follows the MERN stack that prominently takes advantage of the JSON data format, such charitng solution shall also support JSON as the main type of data.

For the React client that will be developed, Recharts is chosen as the charting library. Built for React, and built upon React components, not only does Rechart fulfill the aforementioned qualities, it also let developers construct their own visualisations from decoupled and reusable components, much like any other components in React [17]. Recharts provide a wide range of visualisations such as line graphs, bar charts, pies, area maps, tree maps and even more, each with its own variants and customisations.

Listing 4.8 shows an example how a simple bar chart is built, and Figure 4.2 is the rendered bar chart from the code. From these, it is easy to see that the `BarChart` component takes the data (through the `data` prop) in the form of an array of objects. Each object represents a section on the x-axis of the bar chart, each section has at least one bar, each bar corresponds to a numerical value of an attribute inside the object. Objects should have similar attributes, in which one has to serve as the `dataKey` for the `XAxis` (the label on the x-axis), while the others could be used to contain numerical values to be charted. In our example, two objects are defined, each has 3 attributes: `name`, `propertyOne`, and propertyTwo. The XAxis's dataKey is name, therefore its values (Page A and Page B) are displayed on the chart's x-axis. Both propertyOne and propertyTwo also serve as dataKeys for the Bar component, therefore their numerical values are visualised in coloured bars (the colours are given as props). Notice that the bars have to be declared and its dataKey specified, which means not all attributes inside an object have to be visualised.

```
import React from "react";
import { BarChart, Bar, XAxis, YAxis, Legend } from "recharts";

const data = [
  { name: "Page A", propertyOne: 4000, propertyTwo: 2400 },
  { name: "Page B", propertyOne: 3000, propertyTwo: 1398 }
];

export default function App() {
  return (
    <BarChart width={500} height={300} data={data}>
      <XAxis dataKey="name" />
      <YAxis />
      <Legend />
      <Bar dataKey="propertyOne" fill="#8884d8" />
      <Bar dataKey="propertyTwo" fill="#82ca9d" />
    </BarChart>
  );
}
```

Listing 4.8. *An implementation of a bar chart using Recharts*

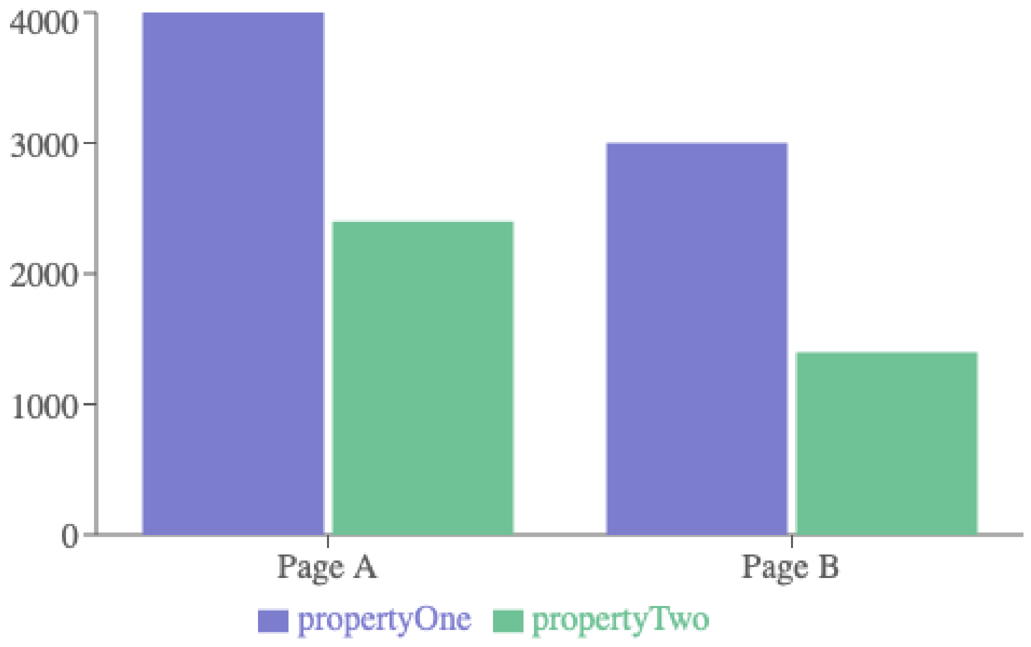


Figure 4.2. *A simple bar chart built using Recharts, using the example data object in Listing 4.8*

From the analysis above, the structure of the data object for our visualisations can be specified. This can be done even though the simple bar chart visualisation will not be used, since other kinds of visualisation also follow the same data format requirements.

To visualise the number of commits made to a project from multiple authors, the stacked bar chart will be used. It is similar to the bar chart above, the only difference is that the bars for each object (in different colours) are stacked on top of each other [18]. On the other hand, a line graph will be used to illustrate numbers of commits made to different projects by a same author [19]. Figure 4.3 illustrates how the stacked bar visualisation could look like.

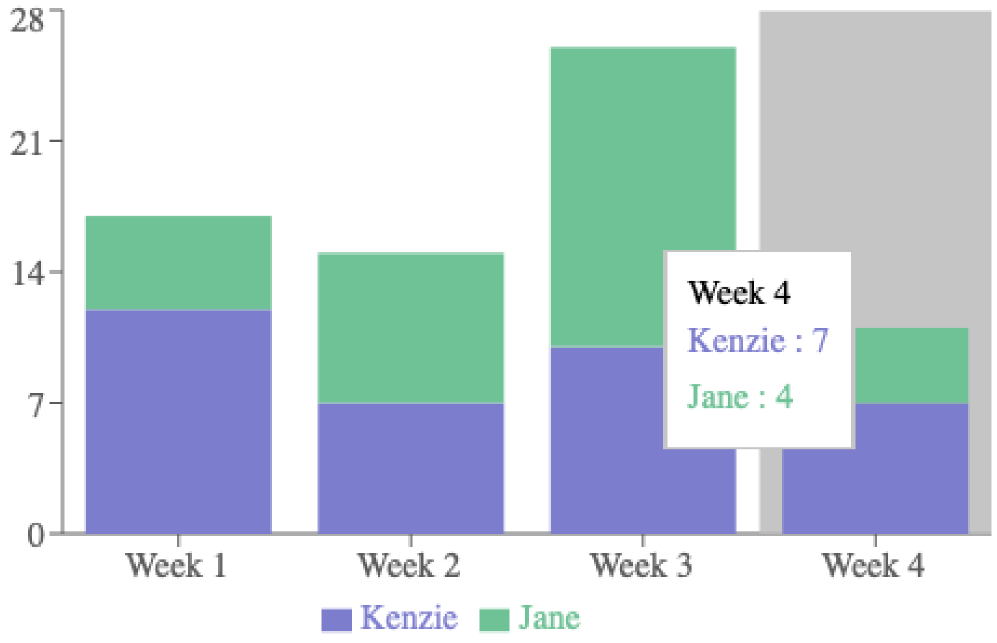


Figure 4.3. *A stacked bar chart visualises the number of weekly commits made to a project by two authors*

The bar chart illustrates well what data is needed, and how the commit data objects have to be processed. It shows the number of commits from each author in different weeks. Therefore, the commit objects need to be divided into weeks based on their committed dates, then divided by the commit authors. For each week, the number of commits belong to every author must be counted and returned - other details of each commit such as the project ID, files affected, etc. do not have to be included in this visualisation.

Listing 4.9 gives the JSON Schema of each data object that the bar chart uses. Each object inside the array represents a week (denoted by the `week` property), containing the number of commits each author has done (the authors' names are listed as the remaining properties). Note that there can be any number of properties representing authors, and its name needs to starts with `author_`. Since the visualisations span multiple weeks, the week objects are stored into a JavaScript array. This is what the data adapter must return to be used by the bar chart visualisation.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "week": {
      "type": "string"
    },
    // author properties start with author_,
    // but their names also depend on the author id
    "author_1": {
      "type": "integer"
    },
    "author_2": {
      "type": "integer"
    },
    // other authors
  },
  "required": [
    "week"
  ]
}
```

Listing 4.9. *The JSON Schema of each data object to be used by the stacked bar chart component*

The line graph component uses a very similar schema for its objects, but instead of having commit authors' names as attributes, they would be project names. Based on this, the next steps in the data adapter implementation - defining visualisation endpoints and implementing data processing methods - can be taken.

4.2.5 Implementing data processing methods and API endpoints

As discussed above, each data object of either visualisation will represent a week, each contains different attributes whose keys represent the commit author's name or the project name. Listing 4.10 is the implementation of the data adapter's processing function corresponding to Visualisation A. This function fetches all commits belonging to the project given as a parameter, filters them with the time range and author names, generates data objects in the correct format to be returned in an array.

The processing function of Visualisation B can be implemented very similarly, as the only difference is between projects and commit authors. With both functions done, they can be used right away to finish building the Express server. Listing 4.11 shows the functions being used inside the data controller implemented in subsection 4.2.1. The necessary query information is extracted from the request's query parameters, then passed to the data processing functions to be used.

As the scope of this thesis only concerns data coming from GitLab, this is the only con-

```
// src/processData.js
// ...
const buildWeeklyProjectCommitsNums = async (projectName, startDate, endDate) =>
{
  const result = []
  const projectCommits = await Commit.find({project_name: projectName})
  const allAuthorNames = await Commit.distinct('author_name');
  // divide the given time range into weeks. Each week has its own
  // start and end date.
  const weeks = divideTimeRangeIntoWeeks(startDate, endDate);
  let weekIndex = 1;
  for (const week of weeks) {
    let weekObject = {
      week: weekIndex,
    }
    const weeklyCommits = projectCommits.filter(commit => {
      return
        new Date(commit.created_at).getTime() >=
          new Date(week.startDate).getTime() &&
        new Date(commit.created_at).getTime() <=
          new Date(week.endDate).getTime();
    });
    for (const authorName of allAuthorNames) {
      const weeklyCommitsByAuthor = weeklyCommits.filter(commit => {
        return commit.author_name === authorName;
      });
      weekObject[`author_${authorName}`] =
        weeklyCommitsByAuthor.length
          ? weeklyCommitsByAuthor.length
          : 0;
    }
    weekIndex++;
    result.push(weekObject);
  }
  return result
}
// ...
```

Listing 4.10. *How a data processing function can be implemented*

troller that will be implemented. However, it can be easily expanded to provide support for data from other sources by implementing similar controllers and endpoints for them. For example, when data from Trello is available from the MongoDB instance, it can be fetched to the adapter, processed, filtered and aggregated, and returned through similarly. By adapting data from varying sources to one similar format, this adapter then achieves another one of its primary goals - a uniform interface for different visualisations.

The data adapter is now fully implemented. It is now able to be used from the React front end. The last piece of the puzzle is to implement the React application, connect it to the adapter, and start querying.

```
// src/controller/gitlab.js

// ...
const {
  buildWeeklyProjectCommitsNums,
  buildWeeklyPersonCommitsNums
} = require("../processData")

gitlabRouter.get("/weekly-project-commits", async (req, res) => {
  const projectName = req.query.projectName;
  const startDate = Number(req.query.startDate);
  const endDate = Number(req.query.endDate);
  const weeklyProjectCommitsNums =
    await buildWeeklyProjectCommitsNums(projectName, startDate, endDate);
  res.status(200).json(weeklyProjectCommitsNums);
})

gitlabRouter.get("/weekly-person-commits", async (req, res) => {
  const authorName = req.query.authorName;
  const startDate = Number(req.query.startDate);
  const endDate = Number(req.query.endDate);
  const weeklyPersonCommitsNums =
    await buildWeeklyPersonCommitsNums(authorName, startDate, endDate);
  res.status(200).json(weeklyPersonCommitsNums);
})

// ...
```

Listing 4.11. *Resolver functions used in Apollo's `resolvers` object*

5. USING THE DATA ADAPTER FROM A FRONT-END APPLICATION

A React application shall be implemented to use the data adapter implemented in the last chapter. As specified in the previous chapter, this front end client will have two visualisations: a bar chart showing the number of commits made to a project each week from different authors, and a line graph showing the number of weekly commits made to different projects by the same author. Each visualisation fetches its required data from its own corresponding endpoint in the data adapter API interface.

5.1 Setting up the React front end

To initialise the React client, Create React App, a tool which lets developers quickly build a React application that can be easily configured to other back-end and database platforms, will be used [20]. This creates a new directory, separate from the implemented data adapter, containing the necessary assets for the app. The entry point of the React application is the `index.js` file inside this directory.

Similar to the data adapter, the front end client will also be containerised separately inside the same Docker network (`visdom-network`). The host name and port number of the data adapter (`visdom-gitlab-adapter` and `4000`) are supplied to the front end as environment variables (`REACT_APP_ADAPTER_HOST` and `REACT_APP_ADAPTER_PORT`, respectively). The front end will be able to connect to the data adapter using an URI built from these variables.

5.2 Fetching data from the data adapter interface

Communicating with the data adapter is done with the help of `axios`, an HTTP client that can help developers handle requests and responses between the React application and the server [21]. Listing 5.1 shows how `axios` is used to communicate with the data adapter.

Note here that the `weeklyProjectCommits` function (which takes care of fetching data for the bar chart component) accepts 3 parameters: the project name whose commits will

```
// src/queries.js
import axios from 'axios';
const uri = `http://${process.env.REACT_APP_ADAPTER_HOST}:
                    ${process.env.REACT_APP_ADAPTER_PORT}`
export const allCommits = async () => {
  try {
    const response = await axios.get(`${uri}/commits`);
    return response;
  } catch (error) {
    throw error;
  }
}
export const weeklyProjectCommits = async (projectName, startDate, endDate) => {
  try {
    const response = await axios.get(`${uri}/weekly-project-commits`, {
      params: {projectName, startDate, endDate}
    });
    return response;
  } catch (error) {
    throw error;
  }
}
// other queries like allCommits, allAuthors, etc.
```

Listing 5.1. *Communication between React front end and data adapter using axios*

be processed, alongside the time range (start & end date) that concerns the commits. `axios` then transforms these parameters into query parameters passed to the HTTP GET request. The data adapter, as discussed in the previous chapter, will extract these parameters and use them to process the necessary data.

5.3 Building React visualisations

The final step is to use these functions inside the React components. Without going into the details of how React works and its terminology, Listing 5.2 is a piece of pseudocode on how data fetched from the data adapter (using the aforementioned functions in `src/queries.js`) is passed into React components. Note that this code does not represent a syntactically correct and functional React component.

The resulting React component can be seen in Figure 5.1. In addition to the visualisation chart itself, the application has other components: buttons to switch between charts, time pickers that let users choose the time range, as well as radio buttons to choose which projects/commit authors whose commits data is displayed.

The second visualisation, a line graph showing the number of weekly commits by an author to different projects, can be implemented very similarly. Fetching data is done by calling the `/weekly-person-commits` endpoint, and the visualisation component is implemented in a similar logic as the bar chart, just with different components - Figure 5.2

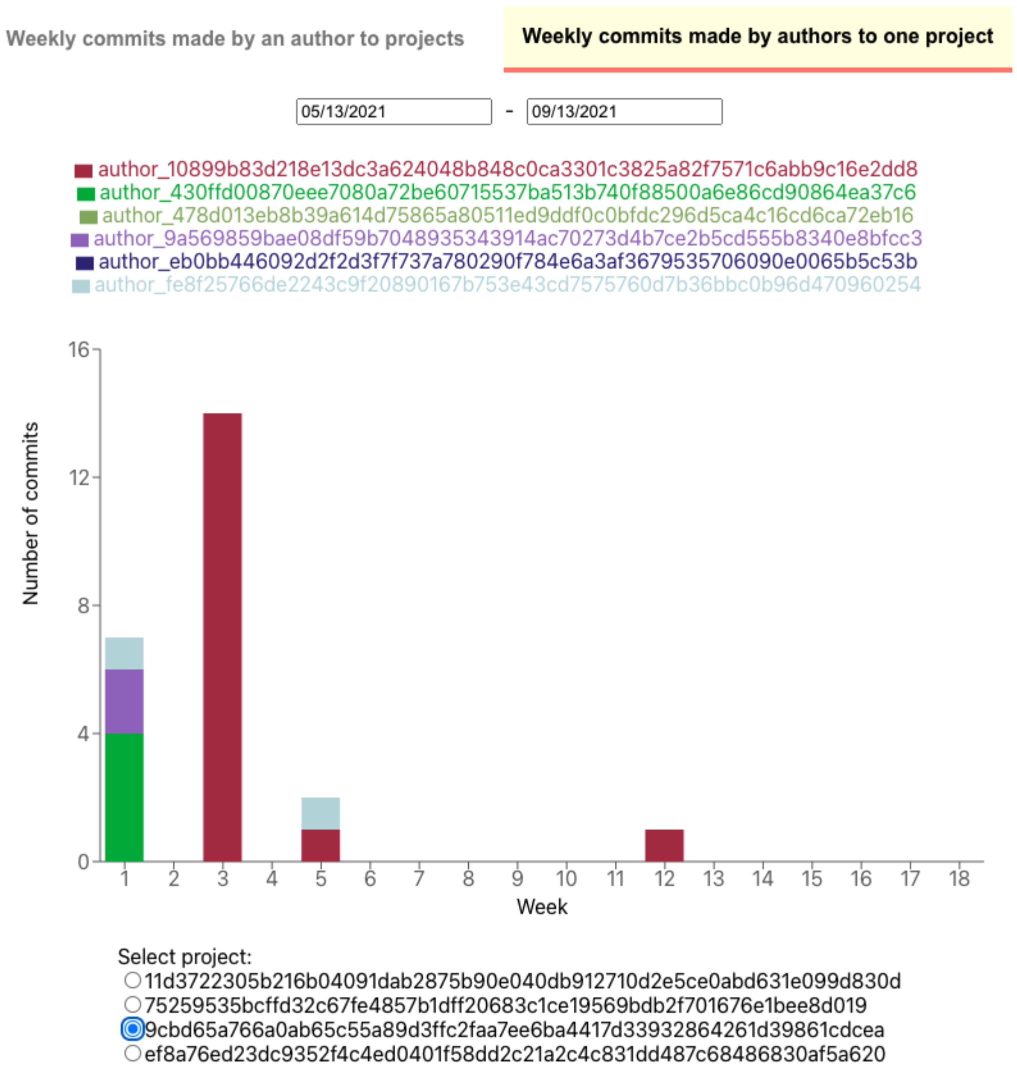
```
// src/view/WeeklyProjectCommits.js
import { BarChart, Bar, XAxis, YAxis, Tooltip, Legend } from 'recharts';
import { allProjects, weeklyProjectCommits } from '../queries';

export default function WeeklyProjectCommits() {
  const projects = allProjects();
  const visualData = weeklyProjectCommits(allProjects[0], startDate, endDate);
  return (
    <BarChart data={visualData}>
      <XAxis dataKey="week" />
      // other Recharts components
      {authors.map((author, index) => (
        <Bar stackId="a" dataKey={`author_${author}`} />
      ))}
    </BarChart>
  );
}
```

Listing 5.2. *Pseudocode of the React visualisation that uses data from the adapter*

shows the result. This is where the benefit of the adapter really shines - the uniform interface providing data regardless of source or visualisation drastically improves implementation, and therefore, scalability. Vastly different visualisations can be implemented as long as the data adapter provides data in correct formats through its API.

The visualisation uses data fetched directly from the implemented data adapter without any further front-end processing. All processing is done in the back end, while the front end simply receives the resulting data and plugs them in to the React components in a declarative manner. Different visualisations query from different endpoints, and get back data that is suitable for their use. This achieves our main goal of separating data processing from the front end, provide a uniform data model regardless of data sources or front-end clients, and unlocks the benefits discussed in Chapter 2. With the test implementation of data adapter and front end completed, it is now time to discuss and analyse the issues observed during the process.


Weekly commits made by an author to projects

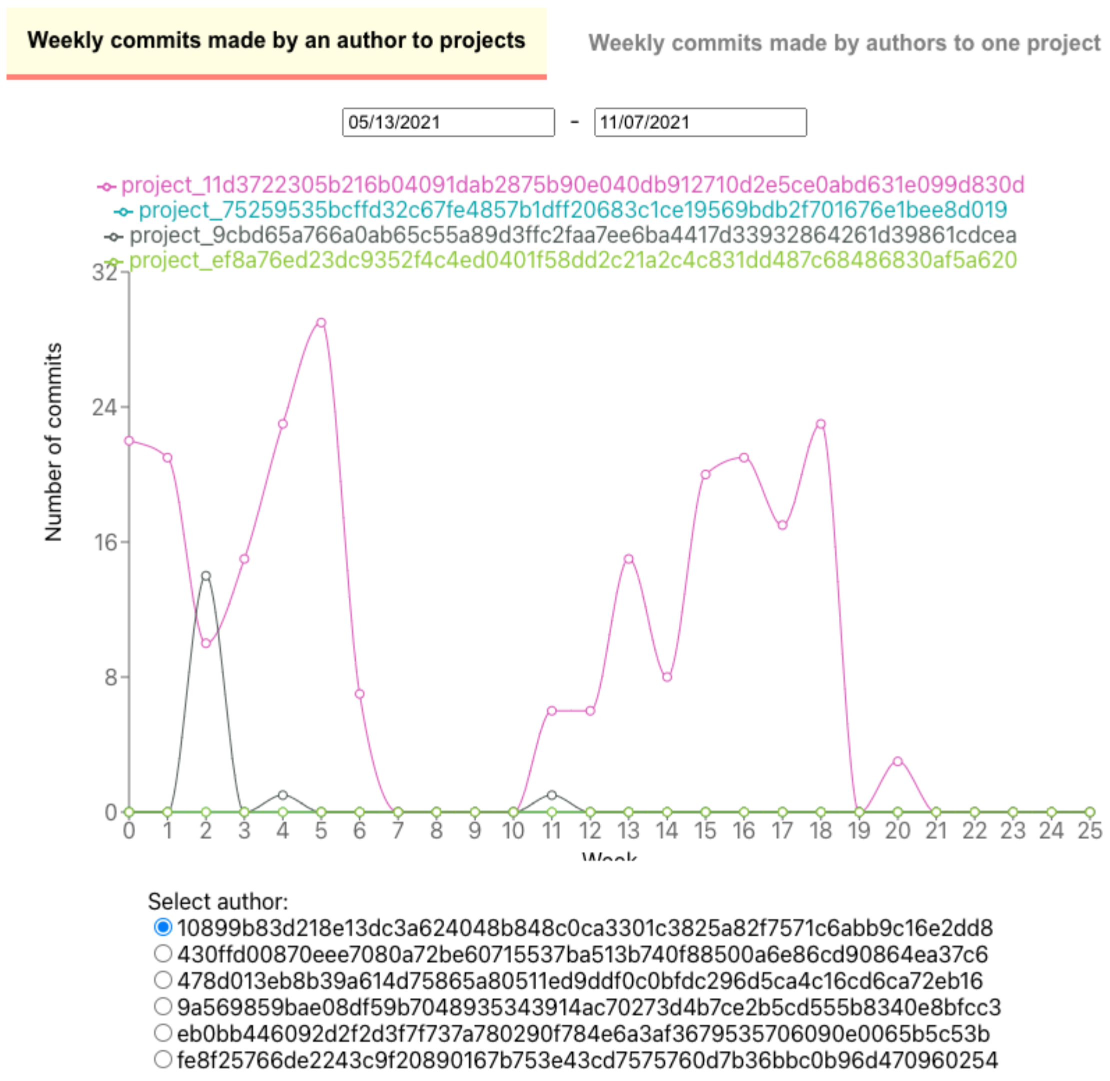
Weekly commits made by authors to one project

05/13/2021 - 09/13/2021

🟥 author_10899b83d218e13dc3a624048b848c0ca3301c3825a82f7571c6abb9c16e2dd8
🟩 author_430ffd00870eee7080a72be60715537ba513b740f88500a6e86cd90864ea37c6
🟩 author_478d013eb8b39a614d75865a80511ed9ddf0c0bfdc296d5ca4c16cd6ca72eb16
🟪 author_9a569859bae08df59b7048935343914ac70273d4b7ce2b5cd555b8340e8bfcc3
🟦 author_eb0bb446092d2f2d3f7f737a780290f784e6a3af3679535706090e0065b5c53b
🟦 author_fe8f25766de2243c9f20890167b753e43cd7575760d7b36bbc0b96d470960254

Select project:
○ 11d3722305b216b04091dab2875b90e040db912710d2e5ce0abd631e099d830d
○ 75259535bcffd32c67fe4857b1dff20683c1ce19569bdb2f701676e1bee8d019
[✓ 9cbd65a766a0ab65c55a89d3ffc2faa7ee6ba4417d33932864261d39861cdcea]
○ ef8a76ed23dc9352f4c4ed0401f58dd2c21a2c4c831dd487c68486830af5a620

Figure 5.1. *A React visualisation component that uses data from the adapter, whose pseudocode is in Listing 5.2. The visualisation contains a time selector at the top, followed by a bar chart. Each color represents a different author, whose identity is decoded into an alphanumerical string. Under the bar chart, users can select which project to show the relevant data - the projects' names are also decoded similarly.*


Weekly commits made by an author to projects
Weekly commits made by authors to one project
05/13/2021 - 11/07/2021
project_11d3722305b216b04091dab2875b90e040db912710d2e5ce0abd631e099d830d
project_75259535bcffd32c67fe4857b1dff20683c1ce19569bdb2f701676e1bee8d019
project_9cbd65a766a0ab65c55a89d3ffc2faa7ee6ba4417d33932864261d39861cdcea
project_ef8a76ed23dc9352f4c4ed0401f58dd2c21a2c4c831dd487c68486830af5a620
Number of commits
0
8
16
24
32
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Week

Select author:
[✓ 10899b83d218e13dc3a624048b848c0ca3301c3825a82f7571c6abb9c16e2dd8]
430ffd00870eee7080a72be60715537ba513b740f88500a6e86cd90864ea37c6
478d013eb8b39a614d75865a80511ed9ddf0c0bfdc296d5ca4c16cd6ca72eb16
9a569859bae08df59b7048935343914ac70273d4b7ce2b5cd555b8340e8bfcc3
eb0bb446092d2f2d3f7f737a780290f784e6a3af3679535706090e0065b5c53b
fe8f25766de2243c9f20890167b753e43cd7575760d7b36bbc0b96d470960254

Figure 5.2. *The second visualisation. Its components are similar to that of Figure 5.1*

6. RESEARCH OBSERVATIONS

In this thesis, the use of data adapters in server-side data processing gives ways to the implementation of a model data adapter for an existing project, as well as a front end for testing. This chapter discusses the observations the author had during the process and after the process was completed.

One of the biggest considerations that have to be taken is the design of the data adapter. As a bridge between the database and the front-end application, the data adapter has to be able to work well with both sides, at the same time not posing them any limits. Developers who wish to build an efficient, sufficient, and scalable data adapter have to understand the other components surrounding it. This project's database and front end choices are rather straightforward - having a MongoDB database and a React application calls for the MERN stack. An Express server serving as the adapter is therefore a simple choice to make [15]. This however, does not mean that this solution will work best for other projects and use cases. Other platforms can store, process, and use data in different formats using varying methods, while also having different pros and cons. These have to be taken into account before the implementation of the adapter begins.

Since technological platform choices are being discussed, it is only fair for the author to disclose a misstep taken while developing the data adapter. Initially, the data adapter was implemented as a GraphQL server, as opposed to the traditional resource-based API. GraphQL is a query language developed by Facebook for APIs, which provides descriptions of the API's resources in details [22]. It enables clients to query for data using simple, declarative strings that only ask for what it needs, and only receives what it asked for. This could be a great choice for, say, a library application that fetches books data. The GraphQL server provides a description about the `Book` data object's fields (for instance, `title`, `author`, `ISBN`, `genre`, etc.); while the client can list what fields it needs, and fetching only those, reducing network loads and the need to filter out unnecessary information. In our data adapter, however, this turns out to be an inconvenient choice. In contrast to the example `Book` data type above, the data objects used for visualisation do not have the same, fixed, predictable fields - each of them contains the `id` of either the commit author or the GitLab project. This information could not be known in advance since it has to be fetched from the database first. The data object specification is therefore inconvenient and not scalable, posing problems for the development and maintenance of the data adapter. The more traditional, resource-based API is then chosen as the architectural style which the adapter server follows. With this style, one does not have to define the resource data fields description, fixing the problem of GraphQL. Scalability, maintainability and code cleanliness of the data adapter therefore increases [23].

The choice of the API implemented with Express also brings forth the main feature of a data adapter: a two-way adaptation of data, a uniform data model that is independent of both data sources and front end visualisations. As long as the data exists in the MongoDB database, the adapter can easily manipulate JSON data from any source so that the end formats are standardised. This standard data format results in the process of building a uniform API interface being very straightforward. Different visualisations may have different endpoints for querying, each endpoint is able to provide its respective visualisation the data in a suitable format. With this standardised, uniformed nature, the data adapter makes implementing and integrating new databases and visualisations convenient, as the clients and data sources do not have to concern themselves with one another - but only the data adapter interface itself.

Another big advantage of having a data adapter in this project, as discussed in earlier chapters, is the separation of roles between different components in VISDOM. The front end now can focus on delivering the visuals and user interactions while the data adapter is responsible for data processing tasks. In turn, front-end and back-end developers can more easily handle the codebase and focus on utilising their skill set [24]. This notion was proven during the process of implementing the data adapter and the React front end: the development activities for each part was simple, straightforward, and focused. In chapter 5, writing React components is the big task, while connecting and fetching data is simply handled using an external library, and no special data manipulation is needed. One can easily imagine, for example, how this is a convenient and efficient setup for front-end developers to focus on their core competencies. In a large scale project involving a larger pool of individuals, the benefits of this separation will be even more pronounced.

On the other hand, having smaller, separate components/services responsible for different things is not without its challenges. In this project, a challenge in building such a system is initialising each new service. Setting up requires new Docker containerisation configuration, not only for the new service itself but also for its integration with other parts of the system. This can be quite simple with a handful of components, as it is in this thesis, but when the system grows large, the sheer amount of cross-integration links between them makes setting up new service challenging, and maintaining that intricate web a nightmare. Issues like this - inherent in the micro-services architecture, which VISDOM follows - are further discussed in other works, such as that of Mendonça et al [25]. Regular architectural reviews should be conducted to make sure that the software system has not grown too complex to maintain. When it is, an architectural revamp or platforms migration will be necessary.

7. CONCLUSION

This research studied, designed and implemented a data adapter to showcase its ability to provide a uniform data model and interface that can take into account different data sources and application consumers. The VISDOM data adapter accepts and processes data objects stored in the database, then transforms them and return the visualisation-tailored results through an API interface, providing different endpoints for different visualisations. Front end visualisations do not need to know the details of data from different sources, and vice versa. In the future, as more data sources and visualisations are added to the system, the data adapter can be easily scaled to be able to handle more complex data in larger amounts.

Other development-specific issues are also improved thanks to the data adapter. By having a dedicated service focused on processing data from the database, the front end of the system, for example, no longer has to be concerned with tasks unrelated to visual presentation and user interactions, making the front-end code lighter and simpler. This in turn reduces the amount of effort required to develop and maintain individual components of the project [23], providing developers with more focused and sensible workflows. By bringing data processing to the back end (typically running on a server), the front end code will therefore no longer run tasks of that kind, dramatically reduce the workload on client machines that run these apps [2]. This, of course, further enhances the user experience while using web applications, helping them reach a wider audience without having to heavily compromise on product features.

A few observations were brought to light with the implementation of the data adapter for VISDOM, as well as the front end application to test it. Decisions regarding the design and implementation platform of the data adapter play a big, long-lasting role in the system, and these choices depend entirely on the nature of the project, its data and usage. Using a more novel solution that works better in most solutions might be detrimental to others, such as the initial, premature choice of GraphQL as the platform to build the data adapter upon. Having separate components taking care of different purposes does indeed benefit the project as discussed in the previous paragraphs, further consolidating these arguments, however also introduced certain problematic patterns that a monolithic system might not have - multiple services initialisation and complex integration between components [25]. A large software project could have tens of separate components interconnected to one another, making the maintenance process very difficult to navigate. Each part of the system may have its own endpoints, ports and API interfaces, making new components challenging to be set up. As the number of components grow, so do the complex links at an even faster rate. This creates a cycle of problems that is hard to break out of, unless regular maintenance and revamping are utilised to keep the project in check.

With that said, however, the evident benefits that a separate data adapter brings to VIS-DOM far outweigh the possible drawbacks. In a project of this scale, the data adapter can dramatically improve the user experience using the web applications, as well as the developer experience in developing and maintaining the system. The role of the data adapter as a separate data processing entity is, therefore, a pattern that web applications should follow to limit the problems inherent in large, monolithic applications, and bring forth improvements for both users, developers, as well as the applications themselves.

REFERENCES

[1] *What Is Three-Tier Architecture | IBM*. URL: https://www.ibm.com/cloud/learn/three-tier-architecture. (accessed: 21.11.2021).

[2] *What do client side and server side mean? | Client side vs. server side*. URL: https://www.cloudflare.com/en-gb/learning/serverless/glossary/client-side-vs-server-side/. (accessed: 21.11.2021).

[3] Ejsmont, A. *Web Scalability for Startup Engineers: Tips & Techniques for Scaling Your Web Application*. New York: McGraw-Hill Education, 2015.

[4] Chelliah, Pethuru, Naithani, S. and Singh, S. *Practical Site Reliability Engineering*. Packt Publishing, 2018.

[5] *Client-side vs. Server-side vs. Pre-rendering for Web Apps*. URL: https://www.toptal.com/front-end/client-side-vs-server-side-pre-rendering. (accessed: 21.11.2021).

[6] Liao, Y.-T., Zhou, J., Lu, C.-H., Chen, S.-C., Hsu, C.-H., Chen, W., Jiang, M.-F. and Chung, Y.-C. Data adapter for querying and transformation between SQL and NoSQL database. *Future Generation Computer Systems* 65 (2016), pp. 111–121.

[7] Massé, M. *REST API Design Rulebook*. O'Reilly Media Incorporated, 2011.

[8] *VISDOM - Visual software diagnostics*. URL: https://iteavisdom.org. (accessed: 5.3.2021).

[9] *JSON and BSON*. URL: https://www.mongodb.com/json-and-bson. (accessed: 5.11.2021).

[10] *Databases and Collections*. URL: https://docs.mongodb.com/manual/core/databases-and-collections/. (accessed: 5.11.2021).

[11] *React - A JavaScript library for building user interfaces*. URL: https://reactjs.org/. (accessed: 5.11.2021).

[12] *Overview of Docker Compose*. URL: https://docs.docker.com/compose/. (accessed: 5.11.2021).

[13] *Mongoose - elegant mongodb object modeling for node.js*. URL: https://mongoosejs.com/. (accessed: 5.11.2021).

[14] *Express - Node.js web application framework*. URL: https://expressjs.com/. (accessed: 5.11.2021).

[15] Subramanian, V. *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node*. Apress L. P., 2019.

[16] *MERN Stack Explained*. URL: https://www.mongodb.com/mern-stack. (accessed: 5.11.2021).

[17] *Recharts*. URL: https://recharts.org/en-US. (accessed: 5.11.2021).

[18] *StackedBarChart | Recharts*. URL: https://recharts.org/en-US/examples/StackedBarChart. (accessed: 5.11.2021).

[19] *SimpleLineChart | Recharts*. URL: https://recharts.org/en-US/examples/SimpleLineChart. (accessed: 5.11.2021).

[20] *Create a New React App - React*. URL: https://reactjs.org/docs/create-a-new-react-app.html. (accessed: 5.11.2021).

[21] *Getting Started | Axios*. URL: https://axios-http.com/docs/intro. (accessed: 5.11.2021).

[22] *GraphQL | A query language for your API*. URL: https://graphql.org/. (accessed: 5.11.2021).

[23] Szőke, G., Antal, G., Nagy, C., Ferenc, R. and Gyimóthy, T. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software* 129 (2017), pp. 107–126. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.08.071. URL: https://www.sciencedirect.com/science/article/pii/S0164121216301558.

[24] Joshi, V. *Seven Reasons Why A Website's Front-End And Back-End Should Be Kept Separate*. URL: https://www.forbes.com/sites/forbestechcouncil/2018/07/19/seven-reasons-why-a-websites-front-end-and-back-end-should-be-kept-separate/?sh=73c0aebc4fca. (accessed: 21.11.2021).

[25] Mendonça, N. C., Box, C., Manolache, C. and Ryan, L. The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture. *IEEE Software* 38.5 (2021), pp. 17–22. DOI: 10.1109/MS.2021.3080335.