FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Object-relational mapping for database access in JavaScript
Student:	Bc. Ladislav Louka
Supervisor:	Ing. Jan Matoušek
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Object-relational mapping (ORM) libraries enable to naturally connect objects written in a given programming language with their representation stored in a database. There are lots of such libraries for JavaScript (JS) and TypeScript (TS), but each comes with its own set of compromises. Explore and describe available open-source frameworks, provide example implementations of application showcasing their advantages and downsides. In the benchmark application test primarily feature richness of framework, efficiency, type support for TypeScript and ability for relational data fetching. Discuss good and best practices for use with each library and what project they fit in.

Guidelines:
1) Research existing JS and TS libraries and explore their problems and benefits by gathering users' experiences.
2) Design a benchmark database and example application, implement the application in each framework.
3) Describe and test the frameworks with a focus on their functionality, efficiency of database usage, speed and usability.
4) Provide outcomes of the tests, analyze their results. Describe which framework has an advantage in each situation.
5) Recall on gathered experience, discuss findings, propose improvements and continuations.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 3 December 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Object-relational mapping for database access in JavaScript

Bc. Ladislav Louka

Department of Software Engineering
Supervisor: Ing. Jan Matoušek

April 2, 2023

Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on April 2, 2023 . . . .

Citation of this thesis

Louka, Ladislav. *Object-relational mapping for database access in JavaScript.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

TODO

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

TODO Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Replace with comma-separated list of keywords in English.

Contents

List of Figures

Introduction

In the modern world, data have become an essential aspect of almost every field. From e-commerce to healthcare, education to finance, data is everywhere and plays a critical role in decision-making processes. The advent of Web 2.0, which brought with it the concept of user-generated content, was largely supported by connecting the Web to databases. Social media platforms, for example, rely heavily on data to provide personalised recommendations, targeted advertising, and other features that keep users engaged. Even non-Web entities and applications often need significant data storage and, as a result, the ability to manage and manipulate data has become a critical skill for developers and organisations alike.

Relational databases such as SQL Server and PostgreSQL are by far the most popular databases for data storage used in business-level applications. These databases use the relational data model, which is based on tables with rows and columns, to store and manipulate data. However, there are also non-relational NoSQL alternatives like MongoDB and Firestore that use a document data model, key-value stores like Redis, or a graph data model (Neo4j) to manage data. Although these databases have their unique strengths and weaknesses, they are generally considered to be more flexible than relational databases and are particularly well-suited for managing unstructured data.

Object-oriented programming languages and languages incorporating parts of the paradigm, such as Java, Python, Ruby, and JavaScript, have gained popularity due to their ability to create complex software systems that can handle large amounts of data efficiently. Object-oriented programming (OOP) is a programming paradigm that represents concepts as ”objects” that have attributes (data) and behaviours (methods). This makes it easier to write, maintain, and reuse code, which is essential when working with large-scale software systems.

Despite the popularity of object-oriented programming languages, there is often a disparity between OOP languages and the relational data model used by many databases. OOP languages are designed to work with objects, whereas relational databases are designed to work with tables. This can make it challenging for developers to work with databases using OOP languages.

Object-Relational Mapping (ORM) has become a popular solution for developers who need to connect object-oriented programming languages with relational databases. ORM allows developers to work with relational databases using object-oriented programming languages, eliminating the need to write complex SQL queries. By abstracting away the details of the underlying database, ORM allows developers to focus on the application logic and reduces the amount of boilerplate code that needs to be written. This makes it easier for developers to work with databases and reduces the potential for errors.

The paper aims to conduct a comprehensive analysis of the most popular ORM packages and SQL query builders for Typescript. This analysis will provide an objective measurement of their relative strengths and weaknesses in terms of functionality, type support, performance, and package quality. Also included are noncomparative examples of syntax and usage to illustrate strengths and weaknesses and to showcase the functionality of the modules. By evaluating each package's performance in these key areas, the paper aims to provide a comprehensive comparison that will be useful to developers who are looking for the best ORM or SQL query builder package for their Typescript project.

Before we start with the full comparison of ORMs that support TypeScript, we must first define what counts as an Object-Relational Mapping Package, what are SQL Query Builders, and other technologies and terms used further in this work. Then we explain how the packages further analysed were selected and by which criteria they are ranked and reviewed.

Chapter 1

Terminology Used

1.1 Object-Relational Mapping

Object-relational mapping is a way to access relational data in an object-centred programming language. The primary purpose is manipulating data without switching concepts from object-oriented paradigms to the relational representation of data in which most databases operate. The scope of this translation layer can (as shown later in this work) vary. Different people define packages as ORMs while providing diverse levels of functionality.

At its base level, ORM provides an intermediary layer between applications' OOP model and database which is usually relational (but can be graph or document focused). The layer allows the developer to work with objects in the code, while the package translates it into a relational structure when saved to the database. These packages are often used on projects that are heavily connected to a database model, as ORMs are most beneficial when using a database is commonplace. When used only occasionally, it usually brings too expansive a setup to translate into gains in code readability and maintenance costs compared to executing premade SQL queries.

In addition to the basic functionality of translating between different styles of data representation, ORMs often include functionality such as connection pooling, support for read-only data replications, caching, or database migrations. When using such modules, developers can avoid writing boilerplate code that is typically required.

1.2 SQL Query Builder

SQL query builder is derived from its function to create SQL queries and OOP pattern, which it implements, called ”builder”. Object-oriented programming design patterns are reusable solutions commonly encountered during software development in OOP languages. These patterns propose interactions between objects and their internal structure. There is no single authority on how these patterns are defined, nor a comprehensive list of these patterns, as every author prioritises different patterns and functionalities.

The Builder pattern is one such pattern, providing API for the complex creation process of objects. This pattern is one of the 23 defined in ”Design Patterns” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, which has been highly influential in software engineering. Its purpose is to separate the creation of an object from its representation, allowing for the separation of parts that initially were parts of one construction method into several.

SQL Query is made up of several clauses, which each serve distinct functions. For example, the SELECT clause specifies which columns are to be retrieved, and the FROM clause specifies which table or tables the columns should be. Once the builder pattern is applied to the SQL query, each of these clauses (or even smaller fragments) can be created by calling the Builder’s methods, creating a programmatical way to create SQL queries. The Builder also allows developers to abstract minor differences between different SQL implementations.

As the ability to create queries based on multiple criteria is one of the basic functionalities of ORMs, they are almost always built on some query builder. These can be available standalone or fully integrated into the ORM package. Often, query builders are sufficient for the purposes of database access in most applications, so they were included in the comparison. While they certainly lack feature sets and, compared to ORMs, query builders usually require SQL knowledge, they can be easier to set up and maintain while also being faster and allowing fine-tuned adjustments to a query.

1.3 PostgreSQL

One of the most popular database engines today, PostgreSQL, is an open-source object-relational database management system. Originally developed under the name Postgres (short for Post-Ingres) as a new generation after a successful relational database called Ingress, it was first released under this name in 1989. After several years of development at the University of California at Berkeley, the name was changed to focus on SQL compliance. The whole project moved to open-source, community-focused development in 1996. Currently, the project is maintained by The PostgreSQL Global Development Group, and releases and source code are provided under an open-source BSD-style licence free of charge.

PostgreSQL is, at the time of writing, one of the most popular SQL databases available, owing to its widespread adoption to reliability and high scalability while supporting most of the SQL standard and being fully ACID compliant. ACID is an acronym for Atomicity, Consistency, Isolation, and Durability, which are four fundamental tenets specifying properties for reliability and consistency of transactions (explained in TODO).

Some of the features that often make PostgreSQL stand out amongst other RDBMS are its support for many different and advanced datatypes out of the box, such as the ability to natively store JSON objects and arrays, XML data or geometric types. With extensibility being a significant focus, a lot of functionality can be installed or optionally enabled, further improving the reach and applicability.

There are extensive implementations of the API for many programming languages, including C, Python, Java, and JavaScript. PostgreSQL also offers extensive documentation of both its API and internal functionality, which supports its growth and popularity.

1.4 Lazy loading

Lazy loading is a technique for optimising data retrieval to increase application performance. It is a strategy consisting of only loading data when it is needed rather than all at once, therefore, reducing the initial load in exchange for the need to do additional loading later. Lazy loading is usually achieved by breaking down larger datasets into smaller ones and loading each one only when necessary. Such practice is commonplace in web development, as asset sizes (such as images or JavaScript) have only grown in the evolution of the web.

Some of the most common implementations of Lazy Loading come in the way of only loading low-quality images unless the user is focused on them or splitting code into multiple files, which are fetched when necessary, providing quick first page-load time at the cost of adding additional requests.

When talking about ORMs and database access, lazy loading usually takes place as replacing data retrieval from an object with a call to retrieve the data from the database. In other words, the data of the database object do not need to be loaded when the object representation or its part is created in the programme. This technique is often used for loading relations so that only one table needs to be queried to create the essential representation of the object while skipping the need for additional fetches or joins only to be invoked once needed.

1.5 Eager loading

Eager loading is the programming practice of loading all the required data at once, optimising the number of requests that must be made to retrieve everything. This is done with the expectation that one significant request will minimise the amount of additional data that would need to be sent between the two parties. This can lead to faster load times and improved application performance.

It is usually achieved by sending a singular request and caching the data in memory, even though it might only be needed later. There are obvious downsides to this, such as higher memory usage or often loading more data than is necessary. The concept of eager loading is antithetical to the lazy loading approach, and that is on purpose. Each approach prefers a different focus, and thus each is fit for different usage; lazy loading is practical when a first look or first results matter the most, and eager loading is when the focus is on one large result, which would be slowed down by too many small requests, which would need to be made for the total result.

In the context of object-relational mapping frameworks, we are most likely to encounter eager loading when fetching related entities. This way, when there is an expectation for data about the currently approached entity, the ORM can optimise the query so that the data are already loaded in memory when it is requested later.

1.6 Circular dependence

A common problem in software development, circular dependency occurs when two or more parts of code depend on each other, making it impossible to resolve their dependence onto a dependency graph. Such a graph must conform to limits set out for tree graphs and, therefore, cannot contain a loop. Due to the way how modules are loaded in Node.JS, such a problem would lead to a deadlock and is therefore resolved by trying to run the modules in a specific order. However, such an approach is only sometimes feasible, so other solutions must be used. The issue of circular dependency is also present in the compilation because, while TypeScript does allow asynchronous references of types between files using ”import type ”, if we need to import not only the type but also the value, TypeScript will not be able to resolve the type, and the compilation will fail. There are many solutions to this problem, the most common being dependency injection or lazy loading.

In ORMs and database representation in OOP languages, this problem is generally connected to the bidirectional nature of relations, as its explicit representation will inevitably create circular dependency. Therefore, there needs to be a functionality built in that allows users to define bidirectional relations without sacrificing type safety or encountering a deadlock with importing modules.

1.7 Database transaction

Transaction isolation is a concept used in database management to represent a unit of work. The transaction is typically a series of one or more database operations that are supposed to be completed on the all-or-nothing principle.

In addition to performing database queries atomically, the transaction also needs to provide additional functionality, such as coordination of reads and handling operations in a reliable and recoverable manner.

As database transactions are some of the basic functionalities of modern RDBMS, their handling is essential when considering the ORM framework. Often an operation can only be performed when the previous one succeeded or has to be made strictly in order without another operation having access to the data in between. This can be achieved only through the database transaction, and support for them is necessary for many use cases.

1.8 Database connection pool

A database connection pool is a component that collects and manages several database connections and allocates them to individual requests to the database. It works by creating either a fixed number of connections at the beginning or scaling up the number of connections based on usage. In this way, querying the database does not have to wait for the connection to be established, and the request can be routed through the database connection pool to the currently unused connection. Additionally, due to having multiple connections, multithreaded and asynchronous applications can coordinate connections to the database. Single connection applications can be stalled while waiting for a single otherwise non-blocking request, while others could be served by the database. Such connections must be coordinated with transaction management, as the transaction is inherently connected with the connection that spawned it.

1.9 Read replica

A read replica is a special kind of database instance, a read-only instance of the database presenting additional query points for the applications accessing the database without having to resolve consistency between instances. With usual databases supporting multiple instances, concurrent writes to alternative machines could produce an inconsistent state in the database. With a read-only replica, consistency is not threatened; the only negative is the possibility that the connections will receive a state that is delayed when the replica is not synced to the latest consistent state of the primary instance.

Creation and usage of read replicas can significantly speed up database performance as queries are no longer constrained by single hardware, which usually bottlenecks query speed. Duplicating the data over two instances can double disk read speeds; if different physical devices are used, slow sequential scans over data can run independently and finish faster.

1.10 JavaScript

A high-level dynamically typed programming language developed in the mid-1990s at Netscape Communications Corporation to add dynamic content to web pages. Initially called Mocha, it was later renamed multiple times to finally settle on JavaScript to use the (at the time very high) popularity of Java.

Before JavaScript, websites were almost always purely static documents that were displayed in web browsers (such as Netscape at the time or Google Chrome or Firefox currently). The logic for any web application had to be handled purely on the server side. With the introduction of JavaScript, web pages were able to be more interactive and dynamic. While initially designed to be used when writing HTML documents and executed by web browsers, it outgrew its client-side roots and conquered large parts of the server-side development and even mobile app and desktop application environments. The advantage of JavaScript is that it can be a completely full-stack language that provides exact parity of logic between client and server and allows for significant code portability.

Until the last few years, JavaScript had an exclusive reign over interactive web content, which made it one of the most used programming languages in the world. With multiple deficiencies known and unfixable without massive problems with incompatibility, multiple additions which build atop JavaScript and even whole languages which compile into JavaScript were developed. Some complied languages are, for example, CoffeeScript, Dart or TypeScript. These languages exist to provide additional features and functionality that are not easily or at all possible in pure JavaScript.

1.11 ECMAScript

Soon in the usage of JavaScript for web pages, it became apparent that establishing standards would be a necessary step for compatibility between implementations in different web browsers. Following this consensus, Ecma (originally an acronym for European Computer Manufacturers until 1994) International standards association meeting was held, and the first edition of the document specifying the new standard specification was adopted in June 1997.

The document, coded under the name ECMA-262, is a comprehensive document that has gone over several versions over the years and specifies the syntax, semantics, and behaviour of the language. There is also an extensive description of data types, operators, flow control structures, built-in objects, and API.

ECMAScript is currently used primarily for client-side scripting, with primary implementations being those used in web browsers, such as SpiderMonkey (Firefox), V8 (Google Chrome, Opera) and JavaScriptCore (Safari). Increasingly with new revisions of the standard, even server-side applications and services have started migrating to ECMAScript from other standards (primarily CommonJS), but many constructs are not directly compatible or translatable.

1.12 CommonJS

One of the alternative specifications which reflected missing functionality in ECMAScript was CommonJS. Created to establish conventions on modularisation for JavaScript outside the web browser, it has also standardised several APIs and internal features.

Started in 2009 by an engineer at Mozilla, the project was initially called ServerJS, with its flagship feature being the synchronous loading of modules. This means that once a module is imported, its exported components are immediately available to be used. This simplifies working with modules and was necessary for the expansion of JS code into server-side development and is used widely today.

Since its conception, gripes with the ECMAScript specifications were largely fixed with further iterations, making it also usable in server-side development. Popular packages, including those exclusively used in server development, have migrated their codebase to ECMAScript.

1.13 TypeScript

A statically typed language built on top of the JavaScript foundation, TypeScript was developed by Microsoft Corporation with the focus on allowing developers to catch errors at compile time before the problem is encountered during runtime, which usually requires extensive testing. TypeScript code is written in enhanced syntax and then compiled into regular JavaScript, with several standards supported, including CommonJS and ECMAScript.

TypeScript was designed to address several shortcomings that have been present in the ecosystem for a lost time, especially when creating large-scale applications. JavaScript applications are very flexible with their dynamic and loosely typed nature and prototype usage, but with flexibility comes a large surface area for errors and mistakes.

Today, TypeScript is widely used for web development and JavaScript server-side development. Most popular frameworks provide at least partial support for TypeScript, and some (such as Angular and React) have even switched to it as the preferred language. TypeScript has support in many JavaScript-integrated development environments, such as Microsoft's Visual Studio Code or JetBrains WebStorm. With solid typing comes the ability for more substantial and consistent code completion, guaranteed automated refactoring, and error checking.

Other projects have tried to fix the same issues as TypeScript fixes. For example, Dart, which is developed by Google, works in the same way, although further from the traditional syntax, it is also compiled into standard JavaScript. However, it never gained the same traction, and its focus was changed from alternative to JavaScript to the primary language for development in the multi-platform framework Flutter.

1.14 Node.js

Node.js is an open-source, cross-platform JavaScript based on the V8 engine developed by Google for Google Chrome. It is designed to allow for server-side usage of JavaScript. Released by Ryan Dahl in 2009, it has since become standard for server-side JavaScript development, especially web applications. The framework has gained popularity thanks to its alternative execution model, which separates it from traditional server-side languages. Instead of spawning different threads or workers for connections, it works with a non-blocking asynchronous I/O model, where many concurrent connections can be handled with only a small overhead.

This is achieved through asynchronous programming, where multiple tasks can be executed concurrently without blocking the main execution. Node.js supports asynchronous programming through the concepts of callbacks and promises. Callbacks are functions passed as arguments that are executed in finished or failed states, ensuring that logic can be applied sequentially after the asynchronous operation is finished. Promises provide a more structured and object-focused way to handle asynchronous operations and have become the preferred way. A promise is a representation of value which might not be available yet, containing a status variable and reference for the result once achieved, allowing for code execution while the operation status is updated in the background. When the value of the promise is necessary, the promise can be checked or waited for using the async/await constructs.

While Node.js is currently the most dominant, there are other alternatives available with their own approaches and focuses. The most popular one is Deno, also developed by Ryan Dahl, intending to address some of the security and design issues of Node.js. Deno, for example, contains extensive tools and utilities within its standard library or uses better sandboxing between modules as supported by V8, the engine on which both it and Node.js run.

1.15 npm

One of the key benefits of the Node.js ecosystem is the large number of third-party packages that can be incorporated into projects. For example, many popular web frameworks, such as Koa or Express.js, are built for Node. Database drivers are also provided in module form, and therefore there needs to be a tool which allows users to incorporate such modules into their projects efficiently.

Originally an acronym for Node Package Manager, the three-letter name has been officially checked to the abbreviation of 'npm is not an acronym'. The first release was published in 2010, and it has since become the default Node.js package manager. Npm consists of a command line client, which is also called npm, and an online database of packages called the npm registry, which is hosted at `www.npmjs.com`.

Although npm is the default package manager, alternatives that were created with different focuses and compromises exist, for example, yarn.

1.16 JSON

JavaScript object notation (JSON) is a lightweight data-interchange format that is widely used in web development. Introduced as an alternative to the complex XML format that was previously used, it is based on a subset of JavaScript representation of values. It consists of key-value pairs in objects, arrays, and primitive types. One of the main benefits is its simplicity and readability for humans, which makes it useful for places where data could need to be interpreted by both humans and machines.

JSON has been standardised in the ECMA-4040 document by Ecma International. The document specifies syntax and semantics, ensuring its reliability, consistency, and portability throughout systems and applications.

1.17 Unit of Work

Unit of Work is a software design pattern used most commonly in ORMs and similar frameworks to manage persistence and consistency between application and database state. The pattern is used to group all database operations relating to a single transaction or process and only execute the final state, ensuring they can be performed atomically without requiring lengthy and expensive locking of database rows or tables or risking deadlocks through database transactions.

The main idea is to track changes across the object in memory, and instead of committing every change into the database, only the last state change is executed. This can be applied not only across one object instance but also across whole swathes of objects. While atomicity is undoubtedly necessary on many occasions, and unit of work on the ORM side can significantly reduce the number of requests to the database, it can also lead to inconsistency when multiple applications access the database and data which are currently loaded in memory on one machine are modified by a different one.

1.18 Active record

The Active Record pattern is a design pattern defined by Martin Fowler in his book "Patterns of Enterprise Application Architecture" and is commonly used to represent database records in an application.

The goal of the pattern is to encapsulate logic for interacting with the database table into a single object. Each instance of the object represents a single record, and modifications made on it are then usually flushed with a method call into the database. The base class also provides static methods for CRUD (create, read, update, delete) operations and possibly additional business logic.

INSERT EXAMPLE CODE FROM FOWLER

The main benefit of the Active Record pattern is a simple and intuitive interface for objects and tables. Modifications of the object can be made right on the data in languages, which allow setters and getters on attributes, and static methods provide a simple gateway to work with the table.

Limitations of the pattern come in the tight coupling between the application and database logic, as the object instance is inherently tied to the database representation. This makes it harder to test the implementation and often requires additional abstraction or mocking. Additionally, the pattern does not easily allow for the management of relations, so a database schema with complex relations might not be able to represent the data easily.

1.19 Data mapper

The Data Mapper pattern, as described by Martin Fowler in his seminal work on enterprise application architectures, provides a clear separation between domain models and their underlying data storage. This approach enables developers to create complex and expressive domain models without being constrained by the relational database schema or various storage options. By decoupling in-memory representations from the data storage mechanisms, the Data Mapper pattern promotes a clean separation of concerns and enhanced flexibility in application design.

Distinguished from the Active Record pattern, the Data Mapper pattern ensures that business logic and data access responsibilities remain separate. In this approach, a single entity represents the table or collection, while distinct entities represent individual records. The Data Mapper serves as a data access layer that performs operations on the data storage representation without creating any direct bindings between in-memory objects and the database. This responsibility is solely managed by the Data Mapper, which takes care of any objects that utilise it.

This separation enables applications that employ the Data Mapper pattern to adhere to the Single Responsibility Principle, one of the SOLID principles of software design popularised by software engineer Robert C. Martin. By limiting the responsibilities class must service and ensure that it is not accountable for multiple unrelated tasks, the single responsibility principle aims to create more straightforward and more maintainable classes. Consequently, the Data Mapper pattern contributes to a more robust and modular software architecture that is easier to develop, maintain, and extend.

However, the Data Mapper pattern has drawbacks. One notable downside is the increased complexity introduced by the additional layer of abstraction. This added complexity could lead to a steeper learning curve for developers unfamiliar with the pattern, as well as the potential for increased development time. Moreover, the mapping process between domain objects and the persistence layer may introduce performance overhead, which can be a concern for applications with stringent performance requirements. Additionally, implementing the Data Mapper pattern often necessitates extensive configuration and mapping code, which can be time-consuming to write and prone to errors.

1.20 MVC architecture

The Model-View-Controller (MVC) architecture is a prevalent design pattern in software development, emphasising the separation of concerns by organising application components into three distinct roles. This architectural pattern, originating from the work of Trygve Reenskaug in the 1970s, has found widespread use in modern web development across various programming languages and frameworks.

The three components of the MVC architecture, Model, View, and Controller, each serve specific purposes. The model represents the application's underlying data structure and business logic, encapsulating core functionality, ensuring data consistency and handling the data storage and representation. In contrast, the view is tasked with rendering data and presenting them to users in an intelligible format. The controller functions as an intermediary between the model and the view, processing user input, manipulating the model, and updating the view as needed.

Separation of these components from the MVC architecture facilitates modularity, maintainability, and testability in software design. Each component can be developed, tested, and updated without interaction with the other layers, simplifying the development process and making it more manageable to identify and resolve issues. Furthermore, the separation of concerns allows developers to concentrate on a single aspect of the application at a time, resulting in more organised and efficient code.

However, the MVC architecture has drawbacks. One notable disadvantage is the added complexity resulting from the additional layers of abstraction, which might be challenging for inexperienced developers and could prolong the development process. Additionally, some critics contend that the strict separation of concerns can create a rigid structure that might need to be better suited for applications with rapidly changing requirements or unconventional designs. Additionally, the structure may be too complex for many projects, which would benefit from more concise and flexible architecture.

Chapter 2

Framework selection

Selecting the optimal framework for any project can be difficult with many parameters and options, and quite often, there are better options than the most popular. The JavaScript ecosystem is rich in choice, as throughout the years, many developers and companies have aimed to create packages in their image. Mainly due to this plethora of choices, there is a need for an overview, which would present advantages and disadvantages. However, only some frameworks can be reviewed; therefore, at least essential criteria need to be established.

The selected packages were selected for their support of TypeScript, with varying levels of compatibility, which will be shown in further detail later. Additional criteria considered were popularity and support as separate factors, leading to the inclusion of widely-used packages with currently limited support and development and lesser-known packages with solid support.

2.1 Typescript support

The primary selection criterion for the packages was TypeScript compatibility. Each package had to have at least a basic functionality working and typed, requiring only reasonable effort to integrate. The degree of support varies among the packages, and their level was also measured in comparison, but the base level was necessary to be considered.

The functionality considered essential is not easy to define either, but as the level of type support varied, the minimum settled on was package and connection setup and simple querying. The package had to have connection options typed, at least for primary usage, as listing all options for all connections is not necessary for most uses. Querying and updating database records is the most common activity for which ORMs and connection builders will be used, so the types they provide are some of the most useful. The result of a simple non-joined query on one table should be able to return exact and correct types, and an update of the record should also at least suggest the attributes which can be changed.

2.2 Popularity and Support

Popularity was inherently a factor in the selection of packages; if the package was known more, its likelihood of being found was smaller. We researched popularity in several ways; the primary source was searching by name and keyword ORM on the npm repository. Secondary sources were articles on ORM and database access in Node.js. The npm repository provides statistics about the packages listed on it, the most prominent being weekly downloads. The statistic is good for basic orientation but is not a great indicator of the exact number of users, as users can download the package multiple times, most packages are cached by third parties, which automatically download a version when it is released and many more ways, which skew the number. Additional input for popularity was the number of issues and stars the project currently holds on GitHub.

Support is a secondary attribute that is highly linked to popularity. Although all packages reviewed are open-source, only maintainers can merge code into the main branch or release versions onto the registry. If they are no longer active, the project effectively stops. While they can be released under a new name if the licence permits such a thing, no packages missing implementation into the benchmark have forks that would relieve the issues encountered. High-quality support is crucial for addressing issues, incorporating new features and compatibility with changes in underlying technologies.

2.3 Implementation criteria

Although some packages were initially selected for comparison, as previously mentioned, problems that needed to be more severe were encountered during their implementation into the included benchmarks. They will still be introduced, and the issues explained; however, they will only be included in comparisons within the basic summary.

Chapter 3

Ranking and Grading of the Frameworks

This chapter outlines and explains the criteria for evaluating ORM and SQL query builder packages chosen for the comparison. These criteria will be the core points which will be considered, but other specific notes will be made about each package. The main criteria were the level of TypeScript support, range of compatible database management systems, popularity, support, documentation quality, dependency count, and performance in different scenarios.

3.1 Quantifiable Criteria

The main section of the evaluation criteria focuses on technical aspects of the frameworks, specifically their usage of TypeScript, support for different databases, and difficulty composing queries. As these qualities are quantifiable, they were given the highest priority in comparing the packages.

3.1.1 TypeScript Support

The quality and extent of TypeScript support vary among the packages, with some offering better integration and type safety without the need for casts. In contrast, others only provide basic typing or require result type definitions to be written into each request, which amounts to the same behavior as if the result was cast. Such functionality often comes when the package initially written for JavaScript is not rewritten in TypeScript but is only provided with a `types` file, which specifies call signatures, but cannot provide other assurances.

3.1.2 Database Compatibility

Database compatibility is not necessary when working with a large project that may encompass many services or when choosing a toolchain for a team, as the one database may not satisfy all the needs the team might have, and building experience with multiple frameworks could be considered unnecessary spending. Providing a unified API over multiple databases can be one of the benefits of query builders or object-relational mapping frameworks.

3.1.3 Performance

Performance in different scenarios is abstract, but flexibility and performance are crucial in a database access framework. Suppose the package would restrict the ability to access the data, requiring roundabout ways to deal with basic operations. In that case, there are better ways to simplify development, just as if the framework creates excessively suboptimal queries or adds excessive overhead. One of the requirements for a comprehensive ORM framework is the ability to support many use cases and represent and work with many different data models. If ORM cannot allow use cases or cannot represent commonly used database design patterns, it is lacking in some ways compared to one that does.

3.2 Package Properties Criteria

However, technical criteria are only some that should be considered when selecting a framework. Many of these factors are interconnected; often, success in one is either caused by or preceded by doing well in others. For example, while the popularity of the package can show the reliability and usability of the package, it also often results in more issues reported and fixed, and more users are more likely to create community resources supplying or improving official documentation.

3.2.1 Popularity

Popularity measures usage, as indicated by package downloads, the number of issues, and the number of users on GitHub who favourited or followed the repository. While all imperfect measures for absolute popularity, they help compare popularity between packages by their relative difference.

3.2.2 Support

The number of resolved and still open issues is connected mainly to the popularity metric of issues. With such a metric, support can be measured, and the statistic will be included; however, more important than that is the patterns of behavior which maintainers have shown previously. If the release schedule is predictable, bugs and security issues are fixed quickly, hesitant adopters can be assured that this pattern will continue, and the framework is a safe investment. On the contrary, a project which is officially or probably no longer supported can be assumed to be a wrong choice, as it cannot react to newly found errors and problems with dependencies and might be unusable due to changes with TypeScript or Node.js runtime.

3.2.3 Dependencies

As dependencies require maintenance due to their changes and vulnerable versions, their amount should also be manageable. Otherwise, it might increase the maintenance cost for the package and application size. Even though data storage is less critical than previously, having a more storage-conscious package is still beneficial.

3.2.4 Documentation Quality

Documentation quality is critical for new adoption and onboarding for working with the framework. It also cannot be measured with reasonable objectivity. Perceived quality depends on language understanding and users' previous experience with the programming language and similar frameworks. Evaluation of documentation will therefore summarize clarity, extensiveness and whether features such as Javadoc annotations are used to contain or link to the documentation.

3.2.5 Performance

Performance is often secondary when choosing an ORM framework, as quite often, even frameworks adding significant overhead and creating suboptimal queries are not noticeably slowing down the application. As the application grows, the performance can become significantly more critical, and the resources needed can be more expensive to scale. A high-performing package can support this growth by maintaining efficacy under load and effectively using its available resources.

Performance was measured in multiple ways; the first metric was the execution time of a single query to measure the latency added by using the framework, compared to using other frameworks or plain database drivers. Benchmarking this way provides information about the amount of overhead the framework requires to function, and if the connection pool is well initialized, connections are assigned optimally, and data are correctly retrieved. The second benchmark run repeats the test multiple times to eliminate any inconsistency that could occur in a single run.

The following chapters aim to provide a comprehensive and in-depth analysis of packages compared by evaluating each package by these comprehensive criteria with additional added when.

Chapter 4

Benchmark database schema design

4.1 Introduction

This chapter describes the database used for performance testing of the ORM and query builder packages. The database is designed around imaginary data collection about cats, their home domiciles and toys found within these houses, and the toys' manufacturers. The database comprises six main entities - `cat`, `cat colours`, colour `hex codes`, `houses`, `toys` and `toy producers`.

4.2 Cat Entity

The `cat` entity instances represent individual cats which we want to monitor. Each has a unique identifier, name and date of birth, all of which are nullable except for the identifier. This entity aims to represent the basic database table and to verify the correct handling of the data type from Postgres, as JavaScript Date time represents a moment, including time. In contrast, the database entry would only contain the date. Additionally, the `cat` entity uses big integer data type, and handling numbers beyond the standard range allocated in JavaScript is tested. The `cat colour` and `colour hex code` are two entities that represent the cat colour by its name and by its hex code. The entities are intentionally split in this way to use identifying relation - the primary key of the hex colour entity is also a foreign key referencing the id of the cat colour entity.

4.3 House and Toy Entities

The `house` entity represents domiciles where the cats spend their time at their behest. The relation must also account for ambitious cats using several houses as their homes. The main aim is to test the difficulty of implementing and using simple many-to-many relations. The only attribute that provides new data type or behaviour is the simple `has_dog` attribute, specified as a Boolean. It is one of several attributes that test the frameworks' ability to correctly type and convert the data recovered from the database.

The houses can be equipped with many toys for the cats to use. The relation between houses and toys is modelled through a decomposition table which contains attributes representing the number of the same toy in the house. While the primary keys are the identifiers of the house and toy, the decomposition with the amount, rather than several records with an additional identifier, is designed to test the ability to insert a record if it does not exist or update the value referencing its previous state. If more toys are purchased, the owner of the house does not suddenly throw out all toys they already had; they will add them to their current pile. This operation is often called *upsert* - a combination of update and insert, and some database engines, such as CockroachDB, implement it explicitly under this name. PostgreSQL achieves it using the `ON CONFLICT` statement in `INSERT` query. It also tests the handling of composite primary keys, a standard paradigm in many databases.

4.4 Toy Entity

The `toy` entity purpose in testing is in numeric data type used in `price` attribute and usage of additional column attributes such as `CHECK` constraints or `DEFAULT` values in the column. Column `naughty` is focused on commonly problematic strings in software development, such as special Unicode characters, emojis and other issues that could come up in handling data from the database, especially if the encoding is not correctly handled. Toys producers host the JSON columns to test if it is possible to use advanced JSON traversal and query operators provided in PostgreSQL (and their equivalents in other database management systems).

Chapter 5

Benchmark Framework Design

The benchmarking process was designed to compare the performance of various ORM and SQL query builder packages. As such, it was important to ensure that the benchmarking framework was developed in the same environment as the packages themselves. To achieve this, the framework was implemented in TypeScript, the same language used by the packages being tested.

The benchmarking framework had to be designed to accommodate errors that could occur during development and testing of the packages. Additionally, it had to support testing of multiple database schemas and allow results to be exported in a variety of formats for further analysis. The resulting benchmarking framework provides a robust and comprehensive means of comparing database access packages.

5.1 Test Suite and Schema Separation

The benchmarking framework was designed to support separation of tests into multiple test suites, a common practice with JavaScript test frameworks such as Jest or Mocha. Test suite separation allows for organization of tests by subject and contains specifications about the database schema and data expected to be executed. Input and output parameters must be typed to test types support, and the framework should provide sufficient functionality to avoid the need for casting.

The tests are expected to be run simultaneously with snapshots of the database schema and should not interfere with data used by another test suite. As a deliberate choice, this limits the scope of each test's modifications over the database and data. However, it eliminates the need to reset the database to the original state after each test suite, reducing the time it takes to run the benchmark.

5.2 Test type and Error Handling

The framework needs to support multiple tests to ensure the validity of any results it produces. If performance is measured, multiple runs can reduce the impact of statistical anomalies, which can occur due to the innumerable number of external events.

Along with performance, the correctness of both query types, resulting runtime types, and the result value are essential. As types are only visible before compilation, and with typed test suite definitions TypeScript compiler would not compile the code, as it would raise type inconsistency. Even incorrect types will be necessary to be cast into their expected value. However, even just the need for such modification means the package must allow more type definitions.

Resulting runtime types and values are validated using the node module `node:assert`, which provides assertion functions. It is provided to function with testing frameworks such as mocha, which do not offer verification functions. Included are even deep equality checking functions. The main advantage, however, comes from being included in the Node.js standard library, meaning that no additional package has to be included.

Returning an incorrect result is one of many ways the benchmark test can be failed; the package can return an unexpected error, or the test is impossible to perform. Both are fail-states, which the benchmark suite must account for with error handling. One choice during the design process was that a single failure would mark the real test as failed, even though other iterations succeeded. If the package caused the issue, that means the package is not reliable enough.

5.3 Multi-Framework support

The benchmarking bootstrap is designed for sequential testing of multiple packages. This design, rather than separate execution, allows for comprehensive comparison under the same conditions, ensuring accurate results. As managing the dependencies could prove problematic if packages had different dependencies required, npm workspaces were selected as a project structure. That way, top-level dependencies of the framework can be separated from the individual implementations.

5.4 Reporters - Output options

An integral part of the design was the inclusion of reporters. Reporters provide an interface and implementation of multiple output options, enabling the results to be saved and shown in various formats. Standard test frameworks utilise reporters to make code coverage or detailed error stack inspectable. The reporters can interpret the data in different formats with a benchmarking framework. The reporter interface is designed to be extensible, allowing for the easy addition of other output options or data interpretations in the future.

Conclusion

Bibliography

[1] Hopcroft, J.; Ullman, J. D. *Introduction to automata theory, languages, and computation*. CNIB, 1995.

[2] Aho, A. V.; Sethi, R.; et al. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007, ISBN 0-321-48681-1.

[3] Melichar, B.; Češka, M.; et al. *Konstrukce překladačů*. Vydavatelství ČVUT, 1999, ISBN 80-01-02028-2.

[4] Delta live tables SQL language reference. Available from: `https://docs.databricks.com/workflows/delta-live-tables/delta-live-tables-sql-ref.html`

[5] What is Delta Lake? Feb 2023. Available from: `https://docs.databricks.com/delta/index.html`

[6] Introduction to data lakes. May 2022. Available from: `https://www.databricks.com/discover/data-lakes/introduction`

Appendix A

Acronyms

PDA Push-down automaton

DFA Deterministic finite automata

Appendix B

Contents of enclosed medium

```
readme.txt ........................ the file with CD contents description
exe ...................................... the directory with executables
src ..................................... the directory of source codes
  wbdcm ..................................... implementation sources
  thesis .............. the directory of LaTeX source codes of the thesis
text ........................................ the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
  thesis.ps .............................. the thesis text in PS format
```