Puja Heino

Node.js v23.5 in Database Handling Perspective

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

26 December 2024

Abstract

Author:	Puja Heino
Title:	Node.js v23.5 in Database Handling Perspective
Number of Pages:	34 pages
Date:	26 December 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Janne Salonen, head of ICT department

With the integration of JavaScript on the server side through Node.js, web application development has undergone a dramatic transformation. Modern applications now utilize JavaScript across the entire stack, simplifying workflows and improving consistency. Node.js continues to evolve with each version, introducing solutions to challenges such as performance, compatibility, and security.

This thesis examines Node.js v23, highlighting its significant advancements, including improved ECMAScript module handling and runtime performance optimizations. Through an analysis of these enhancements and a comparison with prior versions, the study identifies the benefits of adopting Node.js v23. Furthermore, a practical demonstration showcases how Node.js v23, when paired with SQLite, can facilitate the development of high-performance, scalable applications.

Keywords: Node.js, SQLite, JavaScript

Contents

List of Abbreviations

1 Introduction 1

2 About Node.js 3

2.1 History and Evolution 3
2.2 Key Features of Node.js 5
2.2.1 npm: The Heart of Node.js's Ecosystem 5
2.2.2 Single Threaded Model and Non-Blocking I/O 6
2.2.3 High-Performance V8 Engine 7

3 Node.js and Database Connection 10

3.1 Introduction to Database Connectivity in Node.js 10
3.2 Relational and Non-Relational Databases in Node.js 10
3.2.1 Relational Databases 10
3.2.2 NoSQL Databases 11
3.3 Popular Database Libraries and Tools 12
3.4 Asynchronous Database Operations 13
3.5 Security Considerations 14
3.5.1 Injection Attacks 14
3.5.2 Cross-Site Scripting (XSS) 15
3.5.3 Brute Force Attacks 15
3.5.4 Cross-Site Request Forgery (CSRF) 15
3.5.5 Securing Dependencies 15

4 Node.js v23 and SQLite module 17

4.1 Node.js v23 17
4.2 SQLite 18
4.3 Demonstration: Using SQLite with Node.js v23 20
4.4 Performance Comparison: Node.js v22 vs. v23 25

5 Conclusion 27

References 29

List of Abbreviations

ACID:	Atomicity, Consistency, Isolation, Durability
API:	Application Programming Interface
CLI:	Command-Line Interface
CSRF:	Cross-Site Request Forgery
CSS:	Cascading Style Sheets
ECMA:	European Computer Manufacturer’s Association
ESM:	ECMA Script Modules
HTTP:	Hyper Text Transfer Protocol
IoT:	Internet of Things
JSON:	JavaScript Object Notation
JIT:	Just-In-Time (Compilation)
LTS:	Long-Term Support
MFA:	Multi-Factor Authentication
npm:	Node Package Manager
NoSQL:	Not Only Structured Query Language
ORM:	Object-Relational Mapping
SQL:	Structured Query Language

XSS: Cross-Site Scripting

1 Introduction

In the past 20 years, JavaScript has changed significantly. It started as a small language used for basic tasks, but now it has grown into something very different. With the help of Node.js, JavaScript can now be used for both front-end and back-end tasks.

Node.js is compatible with many different systems and that allows JavaScript to run on the server-side. Before Node.js, JavaScript was only used in the browser for client-side work. Developers can nowadays use the same language for both front-end and back-end. This saves time and effort. It has helped developers to build large, scalable, and fast applications. Node.js is today one of the most popular tools for making web applications.

One important feature of Node.js is its non-blocking, event-driven I/O system. This helps it work fast and handle many tasks at once. Node.js also has a large number of libraries available through npm. These libraries make it easy to build different types of applications, like real-time chat apps and large APIs. Every new version of Node.js brings new features to solve problems with performance, compatibility, and security to name a few.

Figure 1 illustrates the architecture of Node.js, highlighting its core components, including the V8 JavaScript engine, the event loop, asynchronous I/O operations via Libuv, and worker threads. This architecture enables efficient handling of concurrent operations and ensures high performance.

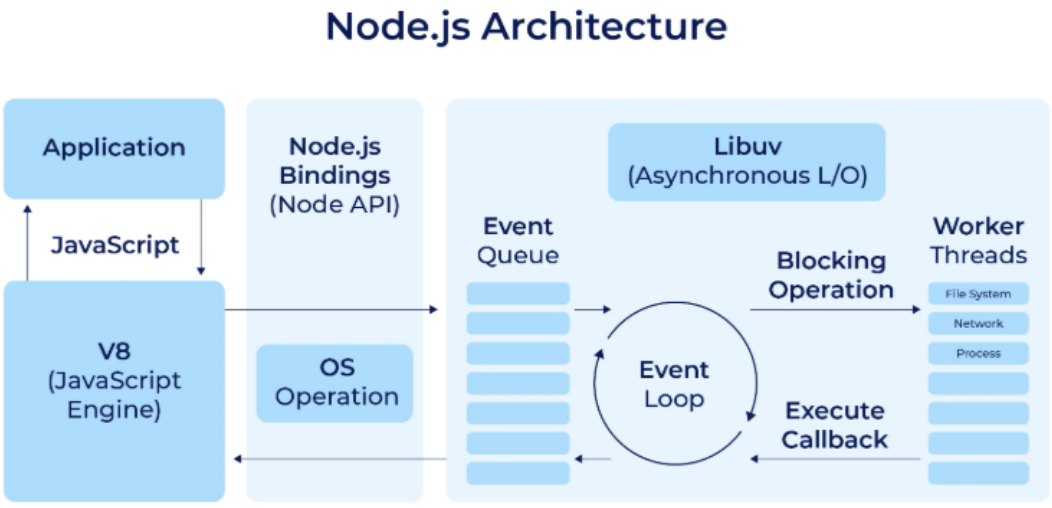


Figure 1: Node.js Architechture (copied from https://litslink.com/blog/node-js-architecture-from-a-to-z)

This thesis will focus on the most important updates in Node.js v23. This version came with some big improvements, like better handling of ECMAScript modules, faster performance, and easier development processes. It will also explain how Node.js works together with SQLite, which is a lightweight and fast SQL database. The thesis shows that using Node.js v23 with SQLite is more efficient for database management than older versions of Node.js.

To make this clear, the thesis will show a real-life example of how these technologies can be used together to build strong and secure applications. It will also explain why upgrading to Node.js v23 is important to enjoy the new features and improvements.

At the end of this thesis, readers will have an understanding of the new features in Node.js v23, why SQLite is a good choice for an embedded database, and how these tools can work together to build modern applications.

2 About Node.js

Node.js has revolutionized software development by introducing JavaScript on the server side. Before Node.js, JavaScript was primarily limited to client-side tasks, handling specific functions within the browser. Node.js made it possible for developers to use the same language for building high-performance applications for both the front-end and back-end. Node.js is built on Chrome's V8 engine and employs a non-blocking, event-driven I/O model. This makes it highly efficient and ideal for real-time applications or those serving large numbers of users. This chapter walks you through the history, development, and enduring relevance of Node.js in modern software development.

Node.js is one of the most commonly used server-side runtime environments in the software industry. It is utilized to create a variety of applications, including websites, desktop software, command-line tools, microservices, and Internet of Things (IoT) solutions. Its growth is fuelled by a robust developer community, which continuously enhances the platform by creating new modules and libraries.

At its core, Node.js is a lightweight, high-performance, cross-platform runtime designed to handle heavy traffic and large-scale operations. Its non-blocking, event-driven architecture ensures efficient memory usage and high throughput. These qualities make Node.js an excellent choice for applications requiring real-time updates or the ability to process large datasets efficiently. [1]

2.1 History and Evolution

The history of Node.js begins in 2009 when Ryan Dahl introduced it at the JSConf EU conference. At the time, traditional web servers like Apache HTTP Server couldn't quite manage to handle larger numbers of simultaneous connections efficiently. To tackle this problem, Dahl developed Node.js, a platform designed to create highly scalable, real-time web servers. On May 27,

2009, Node.js was released as an open-source JavaScript runtime, initially only compatible with Linux and macOS.

One of the key innovations of Node.js was its integration with Google Chrome's V8 engine, which allowed JavaScript to be compiled into machine code for faster execution. Combined with a non-blocking, event-driven model, Node.js enabled developers to manage a large number of connections simultaneously, making it unique among back-end technologies.

In January 2010, npm (Node Package Manager) was introduced, revolutionizing how developers shared and managed code. npm enhanced collaboration and accelerated the adoption of Node.js. By 2011, Node.js expanded its compatibility to include Windows, further increasing its popularity.

Figure 2 provides a timeline of Node.js milestones, showcasing its key developments from its inception in 2009 to the release of Node.js v20.0 in 2023. It highlights significant events such as the introduction of npm, the formation of the Node.js Foundation, and the adoption of ES modules.

Figure 2: Node.js Milestones (copied from https://www.bacancytechnology.com/blog/nodejs-statistics)

The rapid growth of Node.js didn’t come without its challenges. In 2014, governance conflicts led to the creation of io.js, a community-driven fork of Node.js. To resolve these issues, the Node.js Foundation was established in 2015, and by 2016, io.js was merged back into Node.js under the Foundation’s guidance. The Node.js Foundation later combined with the JS Foundation to form the OpenJS Foundation, which continues to oversee its development.

Since then, Node.js has introduced several important updates, such as Long Term Support (LTS) releases starting in 2015, HTTP2 in 2018, and native ECMAScript module support in 2019. Other enhancements, like .env file support and experimental garbage collection improvements, demonstrate the platform’s commitment to staying relevant and addressing the needs of modern developers. These milestones underline Node.js’s ability to evolve with the ever-changing software landscape. [2]

2.2 Key Features of Node.js

2.2.1 npm: The Heart of Node.js’s Ecosystem

One of the major strengths of Node.js is npm (Node Package Manager), which has been a cornerstone of its ecosystem since its introduction in 2010. By providing a centralized repository for sharing and managing code, npm revolutionized software development, fostering collaboration and efficiency. Today, it is one of the largest software registries in the world, with millions of packages available for projects of all sizes.

npm simplifies dependency management, enabling developers to easily install, update, and manage libraries within their projects. This makes it a vital tool for teams across the globe, facilitating the sharing and reuse of code. Through npm, developers can access a wide array of tools for building front-end applications, back-end APIs, testing, debugging, and more.

On top of all that, npm offers advanced features like version control, automated scripts, and security audits, further enhancing the development process. These capabilities have made npm an indispensable resource for developers seeking to create efficient, scalable, and flexible applications with Node.js. [3]

2.2.2 Single Threaded Model and Non-Blocking I/O

Node.js stands out for its non-blocking I/O model combined with a single-threaded event loop. This design allows Node.js to handle multiple connections simultaneously without creating a separate thread for each one. Instead, incoming requests are placed into an event queue, and the event loop processes them efficiently in a non-blocking manner.

Figure 3 illustrates how traditional multi-threaded servers allocate separate threads for each connection, resulting in idle time and resource inefficiency. This highlights the contrast between Node.js's single-threaded, non-blocking approach and traditional methods.

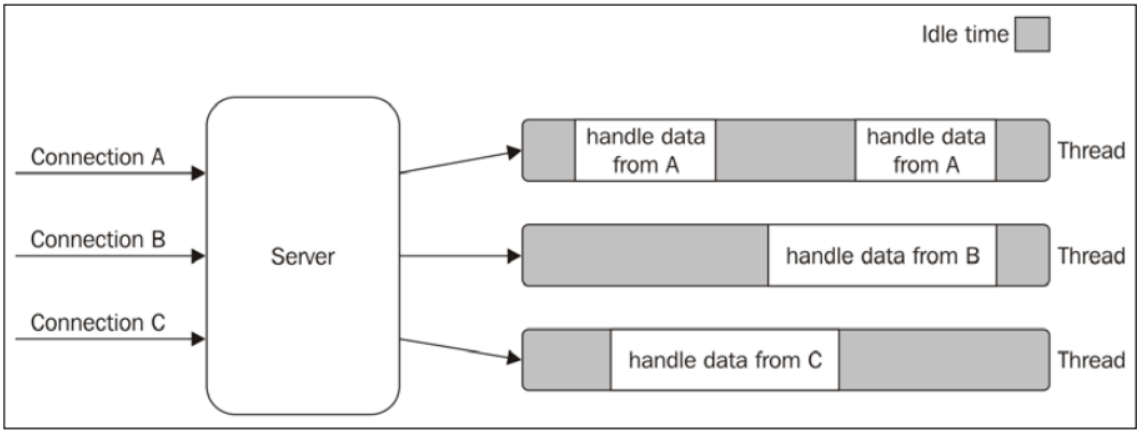


Figure 3: Using multiple threads to process multiple connections (Node.js Design Patterns, Mario Casciaro)

Unlike blocking operations—such as synchronous file reads—that force an application to wait, non-blocking functions allow other tasks to proceed while waiting for a response. For example, when using fs.readFile(), Node.js reads

the file asynchronously, continuing with the next operation while the result is processed by a callback function.

This approach is particularly beneficial for applications requiring constant connections, such as chat systems, online gaming, or live streaming platforms. These environments demand low latency and high concurrency, which Node.js excels at due to its lightweight single-threaded architecture.

Figure 4 demonstrates how Node.js's single-threaded, non-blocking architecture efficiently handles multiple connections by processing tasks in an asynchronous manner. This eliminates idle time and ensures that resources are optimally utilized, making it well-suited for real-time, high-concurrency applications such as chat systems and live streaming platforms.

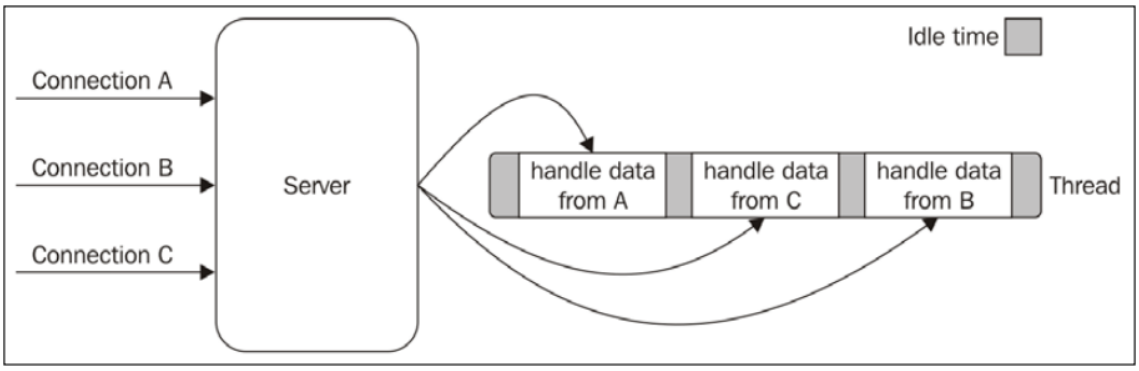


Figure 4: Using a single thread to process multiple connections. (Node.js Design Patterns, Mario Casciaro)

By using a single thread and an efficient event-driven model, Node.js achieves high performance and scalability, making it an excellent choice for handling high-traffic or real-time applications. [4]

2.2.3 High-Performance V8 Engine

The V8 JavaScript engine is one of the key factors that provide Node.js with high performance. V8 was initially created for the Chrome browser and uses JIT (Just-In-Time) compilation, where JavaScript code is dynamically compiled into machine code, enabling applications to run very fast and efficiently.

Figure 5 illustrates the internal process of V8, starting with parsing JavaScript source code into an Abstract Syntax Tree (AST). The Interpreter Ignition converts this AST into bytecode for initial execution, while the TurboFan compiler optimizes frequently executed code into machine code for better performance. This dual approach ensures both fast startup times and efficient execution.

Node.js also uses V8 for its functionality, providing the same performance for JavaScript on the server side as it does in the browser. Written in C++, V8 is available on platforms including macOS, Windows, and Linux. V8 is a modular component that can be regularly enhanced to suit different contexts.

One of the key benefits of V8 is that it supports the current version of ECMAScript, including ES modules, while also maintaining support for legacy features. This ensures that developers can use the latest language features while writing code that is efficient and fast. Additionally, V8 is used in desktop applications developed with Electron, demonstrating that its applications extend beyond Node.js.

Therefore, the V8 engine is a critical part of Node.js that provides speed, optimization, and compatibility. It enables developers to build fast and efficient applications that are scalable and comply with the latest JavaScript standards.[5]

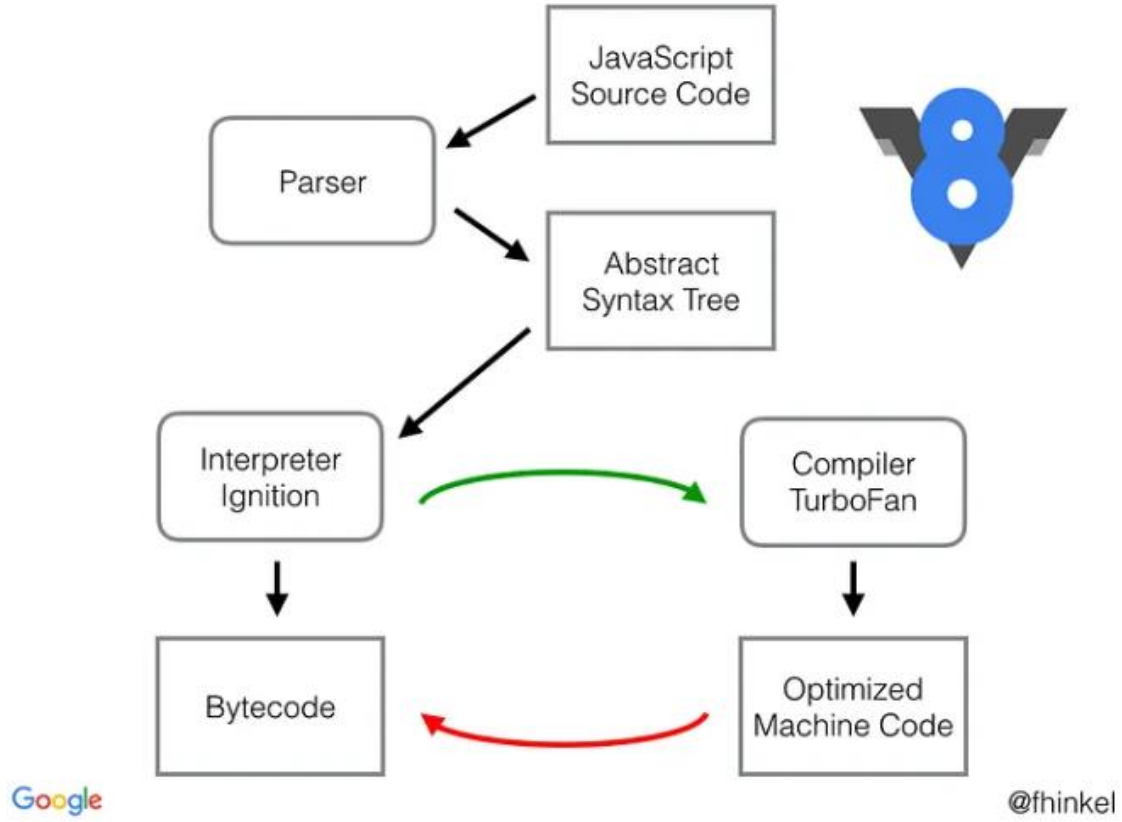


Figure 5: Javascript V8 Engine (copied from https://bashiralhanshali.medium.com/javascript-v8-engine-9682c216d89f)

3 Node.js and Database Connection

3.1 Introduction to Database Connectivity in Node.js

The use of data has become a crucial part of today's application development. Whether it is user details, product information, or analytics logs, database connectivity is essential for any application to function smoothly. Due to its non-blocking, event-driven architecture, Node.js is an excellent option for working with databases and developing scalable, high-performance applications.

Database connectivity in Node.js facilitates the storage, retrieval, and manipulation of data, enabling dynamic content generation, scalability, and secure transactions. For instance, applications like e-commerce sites and social networks rely heavily on robust back-end databases to provide personalization, monitor inventory, and deliver real-time features such as notifications and chat functionality. [6]

3.2 Relational and Non-Relational Databases in Node.js

Node.js works well with two major types of databases, which fit different purposes:

3.2.1 Relational Databases

Relational databases are those that organize data in the form of tables with clearly defined structure and ensure that data is always consistent and accurate. Examples include:

- MySQL: Being very reliable and highly scalable, MySQL is commonly employed in Node.js apps including CMS and e-commerce websites.

- PostgreSQL: Offering a comprehensive list of features and enhancements, PostgreSQL offers JSON data type and geospatial query, which can be utilized for any application that demands flexibility.

3.2.2 NoSQL Databases

NoSQL databases are most versatile databases as they store data in a flexible manner and can be retrieved in any format. Examples include:

- MongoDB: MongoDB is one of the most popular NoSQL databases that works with Node.js. Due to the fact that it uses JSON like structure to store data, makes it a great choice for start-ups as well as small projects with constantly changing requirements.

- Cassandra: Cassandra is known for its excellent scalability and high availability and is commonly employed in distributed systems such as analytics systems.

Figure 6 compares relational and non-relational database models. Relational databases, such as MySQL or PostgreSQL, organize data into structured tables with relationships between rows (e.g., “Accommodation” linked to “Reviews”). In contrast, non-relational databases like MongoDB store data in a more flexible format, embedding related information within a single document for ease of retrieval.

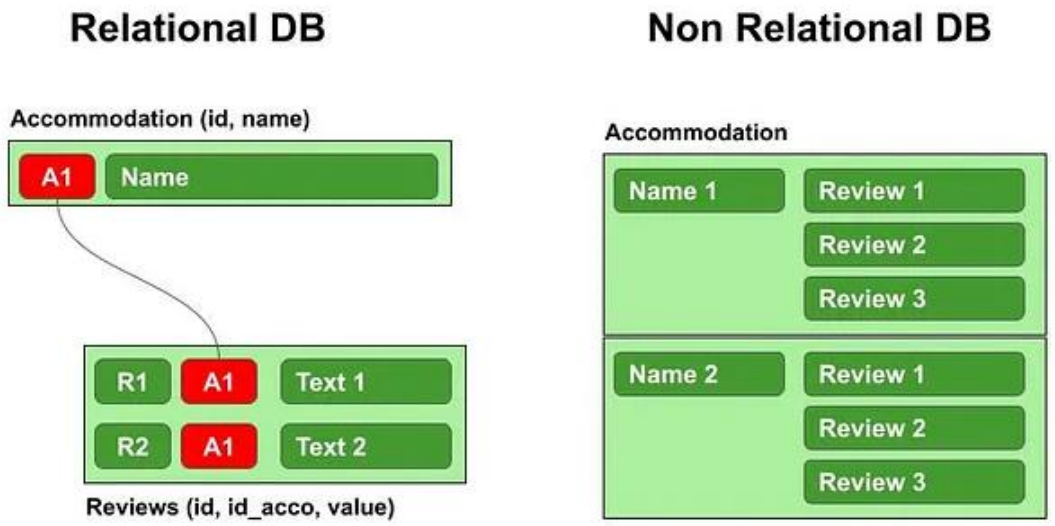


Figure 6: Relational vs Non-Relational Databases (copied from https://towardsdatascience.com/relational-vs-non-relational-databases-f2ac792482e3)

It is important to select the right database based on factors such as data model, performance, scalability and the nature of the application. [6]

3.3 Popular Database Libraries and Tools

There is a large number of libraries and tools for Node.js that makes it easy to work with both relational and NoSQL databases to connect to. These libraries help in enhancing the efficiency of connection and productivity by providing features like connection pooling, query construction and object-relational mapping. Popular options include:

- mysql2 and pg: Commonly used for relational databases especially the MySQL and PostgreSQL, these offer efficient and strong query protocols.

- mongoose: One of the most popular libraries for MongoDB, it provides Schematically model and validate documents for NoSQL database.

- sqlite3: A lightweight library which is used for using SQLite with Node.js, mainly used in small projects.

Such libraries enable developers to work more efficiently in handling database functions and operations and at the same time, concentrate on the core application. [6]

Figure 7 illustrates the popularity of various database technologies based on a survey with over 70,000 responses. Relational databases like MySQL and PostgreSQL dominate the chart, reflecting their reliability and widespread adoption. SQLite is also notably popular, ranking third, further emphasizing its lightweight and versatile nature, especially for small-scale applications.

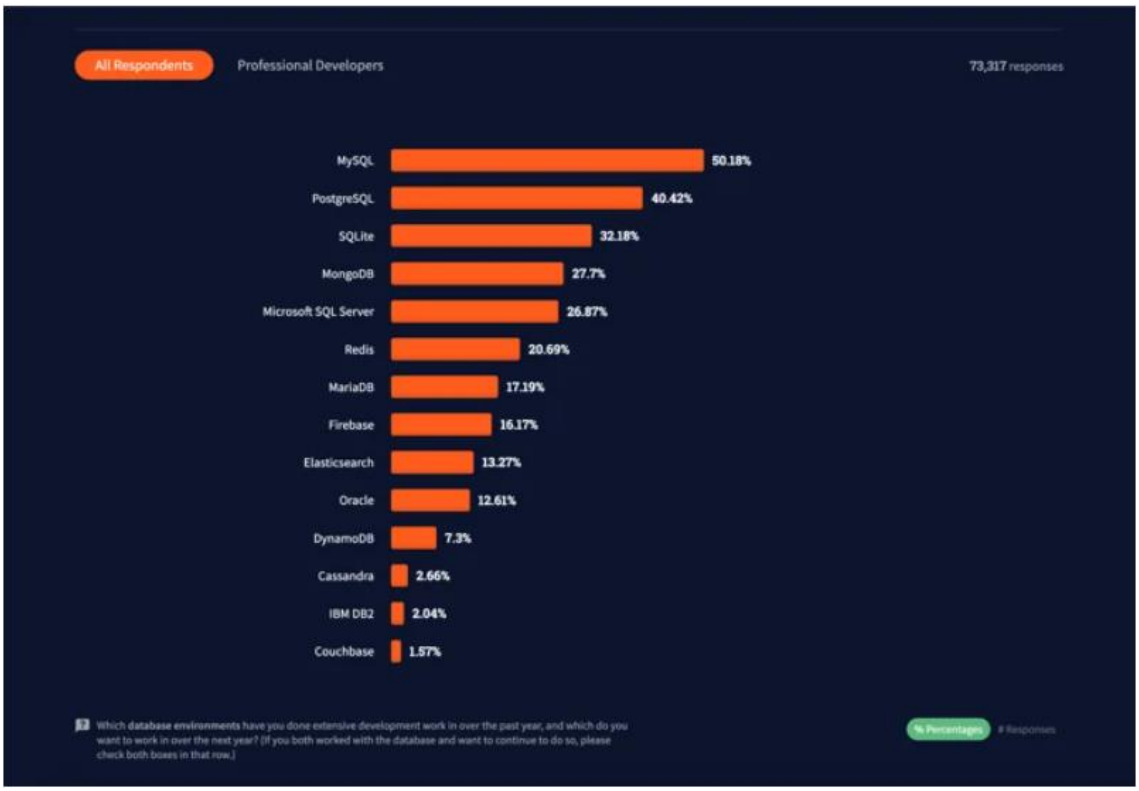


Figure 7: Most popular databases (copied from https://www.bitovi.com/blog/postgresql-dynamodb-mongodb-choosing-a-database-for-a-node.js-app)

3.4 Asynchronous Database Operations

Asynchronous database operations in Node.js allow applications to handle long-running tasks, such as database queries, without blocking the main execution thread. This ensures smoother application performance by enabling other tasks

to continue while waiting for the database response. Node.js achieves this through its non-blocking I/O architecture, supported by promises and async/await, which simplify asynchronous workflows.

Libraries like sqlite3 and mysql2 provide built-in asynchronous methods that make it easier to interact with databases. For example, using async/await enables developers to write cleaner, more readable code for database queries and operations.

To ensure secure and efficient database access, prepared statements and query parameters are commonly used. These mechanisms prevent SQL injection by separating user inputs from query execution. Organizing database logic in a dedicated repository layer further enhances code maintainability and allows for easy migration or updates to database solutions.

Real-time applications, such as chat systems or live notification platforms, particularly benefit from efficient asynchronous database handling. By offloading query execution and maintaining non-blocking operations, these systems can deliver responsive user experiences without bottlenecks or delays. [7]

3.5 Security Considerations

As one of the most popular server-side platforms, Node.js faces several security threats, including injection attacks, cross-site scripting (XSS), brute force attacks, and cross-site request forgery (CSRF). Addressing these risks is crucial to safeguarding data, maintaining user trust, and ensuring application integrity.

3.5.1 Injection Attacks

Injection attacks, where malicious queries or code are introduced through user inputs, pose a significant threat to applications. These attacks can compromise sensitive data or corrupt databases. To mitigate this risk, developers should use parameterized queries or prepared statements to separate SQL queries from

user inputs. Libraries such as sqlstring (for MySQL) and pg-format (for PostgreSQL) help sanitize queries effectively. Additionally, Object-Relational Mapping (ORM) tools like Sequelize and Mongoose offer built-in protections against injection attacks.

3.5.2 Cross-Site Scripting (XSS)

XSS attacks occur when attackers inject harmful scripts into web pages viewed by other users, potentially leading to unauthorized access or data theft. To prevent XSS, developers should validate inputs and encode outputs to avoid code interpretation. Libraries like xss and frameworks such as Helmet, which sets protective HTTP headers, can further enhance protection.

3.5.3 Brute Force Attacks

Brute force attacks exploit repeated attempts to guess login credentials. These attacks can be mitigated by implementing rate-limiting tools like express-rate-limit, locking accounts after multiple failed attempts, and using Multi-Factor Authentication (MFA) for added security.

3.5.4 Cross-Site Request Forgery (CSRF)

CSRF attacks occur when unauthorized commands are executed on behalf of authenticated users. These can be countered by using CSRF tokens, which validate the authenticity of requests. Libraries like csrf provide robust solutions for generating and validating these tokens.

3.5.5 Securing Dependencies

Dependencies in third-party packages can introduce vulnerabilities. Tools such as npm audit and Snyk enable regular dependency checks and facilitate the resolution of potential risks. Evaluating the reputation and maintenance of libraries before integration also helps ensure application security.

By implementing these practices, developers can build secure Node.js applications that not only perform well but also effectively mitigate prevalent security threats. [8]

4 Node.js v23 and SQLite module

4.1 Node.js v23

Node.js v23 introduces several significant upgrades aimed at improving performance, compatibility, and developer workflows. These updates continue to refine the platform, ensuring it stays aligned with modern JavaScript standards and developer needs.

A key enhancement in v23 is its improved support for ECMAScript modules (ESM). This update simplifies module integration, allowing developers to work seamlessly with modern JavaScript syntax without requiring substantial changes to their code. By aligning more closely with browser-based JavaScript, Node.js v23 streamlines both development and maintenance.

The stabilized node --run command is another notable addition, making script execution more efficient and reducing dependence on external shell commands. Alongside this, improvements to the Node.js test runner, such as support for glob pattern, enhance test coverage management across large projects, increasing productivity.

Performance improvements are a highlight of Node.js v23. Integrating the latest V8 engine and adding support for C++20 features result in faster execution times, reduced memory usage, and optimized garbage collection. Applications relying on real-time processing also benefit from improved event signal management, offering better handling of aborted states and synchronized communication.

However, v23 introduces changes that may require developers to adapt. The removal of 32-bit Windows support necessitates updates to legacy systems, and the deprecation of unused CLI flags and legacy features encourages developers to modernize their codebases to meet current standards.

Node.js v23 marks an important step forward with its focus on performance, usability and compatibility. Its advancements empower developers to build efficient, scalable applications while keeping up with the latest JavaScript trends. [9]

4.2 SQLite

SQLite is a lightweight, serverless, and versatile SQL database management system renowned for its simplicity. Unlike traditional databases that require separate servers and complex configurations, SQLite is an embedded library that integrates directly into applications, eliminating the need for an elaborate setup.

Developed by D. Richard Hipp in 2000, SQLite was designed as a less complex and faster alternative to databases like MySQL and PostgreSQL. Its ease of integration has made it popular across various industries, including mobile app development, web browsers, and small to medium-sized websites.

Figure 8 illustrates the internal architecture of SQLite, showing the flow of operations from the API to the file system. The process begins with the tokenizer, which breaks down SQL queries, and proceeds through components such as the code generator and virtual machine. On the backend, structures like the Btree and pager handle data organization and storage.

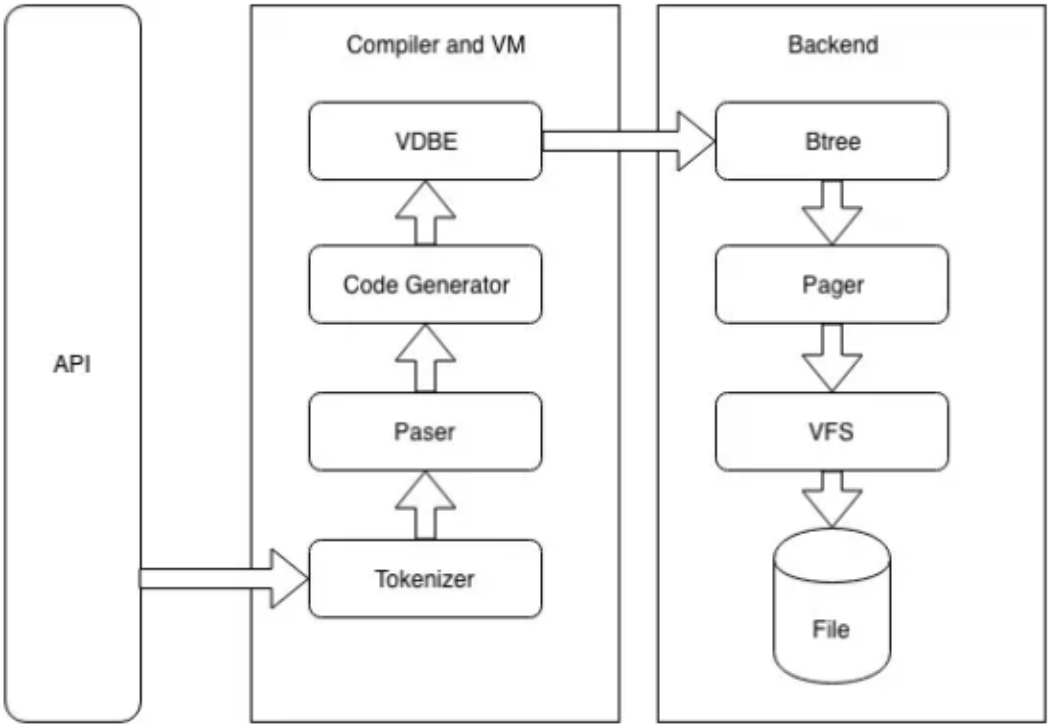


Figure 8: SQLite Architecture (https://medium.com/technology-in-essence/how-sqlite-database-works-b10ac80e4f07)

As an ACID-compliant database, SQLite ensures data consistency and reliability even during system failures or power outages. It stores all database data in a single file, simplifying backup and management tasks. Additionally, SQLite provides a robust set of SQL features along with modern capabilities like JSON storage and partial indexing, making it suitable for current application development needs.

Despite its strengths, SQLite does have some limitations. It offers limited concurrency compared to server-based databases and lacks support for stored procedures or advanced data types. However, its portability, reliability, and ease of setup make it an excellent choice for applications requiring an embedded database solution. [10]

4.3 Demonstration: Using SQLite with Node.js v23

This chapter provides a practical demonstration of integrating SQLite into a Node.js v23 application. The example highlights the advancements in Node.js v23 and explores their impact on database operations. By walking through the steps of setting up SQLite, performing basic operations, and analyzing performance improvements, the demonstration underscores the practical benefits of combining these technologies for secure and efficient application development.

The demo will cover:

- Setting up SQLite in a Node.js v23 application: Learn how to configure SQLite in a project using the latest features of Node.js

- Performing basic database operations: See how to create tables, insert data and query results using SQLite with Node.js

- An analysis of performance differences between the latest iteration of Node.js (v23.5.0) and its preceding version (v22.12.0)

This hands-on example will highlight the real-world use-cases of Node.js v23 and SQLite, and their effective integration. At the end of the demo, you will gain a clearer perspective on how the two can be combined to develop highly secure and efficient applications.

Figure 9 shows the entire process of creating a database (or connecting to an existing one) in code. The process will be broken down into smaller pieces with explanations of every step along the way.

app.js > ...

```js
// Import the SQLite module
const sqlite3 = require('sqlite3').verbose();

// Open or create the SQLite database
const db = new sqlite3.Database('./database.db', (err) => {
  if (err) {
    console.error('Error opening database:', err.message);
  } else {
    console.log('Connected to SQLite database.');
  }
});

// Create a "users" table
db.serialize(() => {
  db.run(
    `CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        email TEXT UNIQUE NOT NULL
    )`,
    (err) => {
      if (err) {
        console.error('Error creating table:', err.message);
      } else {
        console.log('Users table created or already exists.');
      }
    }
  );

  // Insert a sample user
  db.run(`INSERT INTO users (name, email) VALUES (?, ?)`, ['John Doe', 'john.doe@example.com'], function (err) {
    if (err) {
      console.error('Error inserting data:', err.message);
    } else {
      console.log(`Inserted row with ID: ${this.lastID}`);
    }
  });

  // Query and display users
  db.all(`SELECT * FROM users`, [], (err, rows) => {
    if (err) {
      console.error('Error querying data:', err.message);
    } else {
      console.log('Users:');
      rows.forEach((row) => {
        console.log(`${row.id}: ${row.name} - ${row.email}`);
      });
    }
  });
});

// Close the database
db.close((err) => {
  if (err) {
    console.error('Error closing database:', err.message);
  } else {
    console.log('Closed SQLite database.');
  }
});
```

Figure 9: Code example of creating/connecting to a database

These following steps cover the full lifecycle of typical SQLite operations in a Node.js application and are accompanied by proper error handling for each operation.

Figure 10 shows how to import the sqlite3 library in verbose mode, which provides detail debug information and makes it easier to troubleshoot issues during development

```
1   // Import the SQLite module
2   const sqlite3 = require('sqlite3').verbose();
```

Figure 10: Import SQLite module.

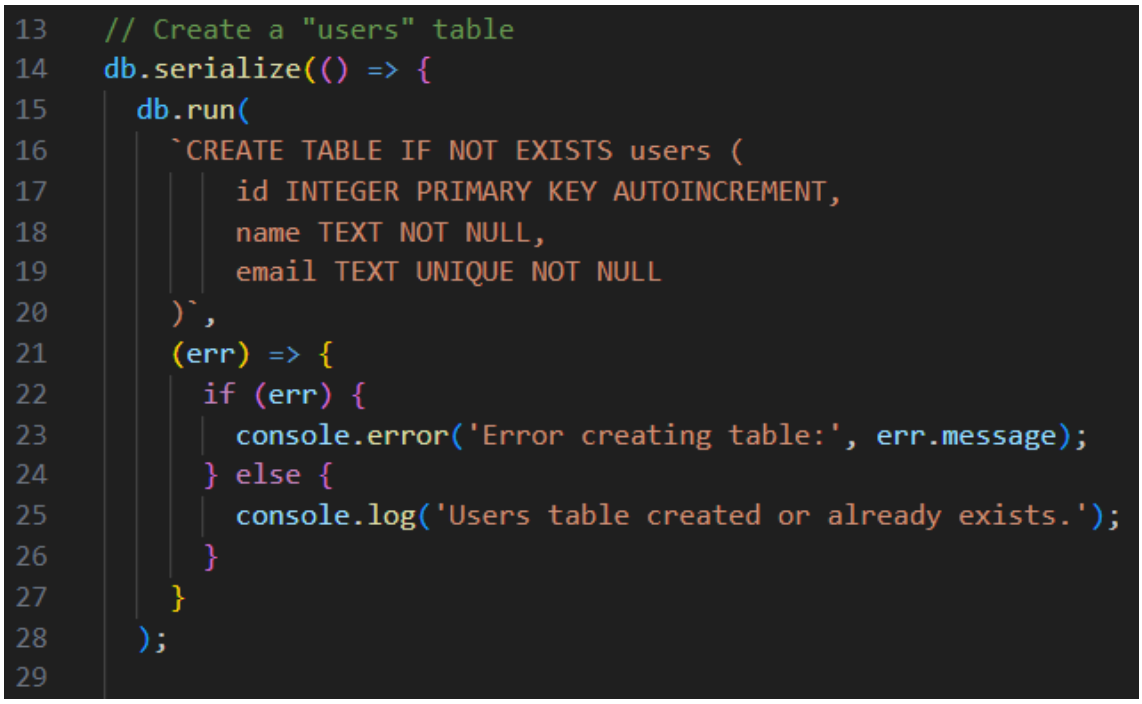
Figure 11 shows how to open (or creates if it doesn't exist) a database file named database.db in the current directory. The sqlite3.Database constructor connects to the database, and an error message is logged if the connection fails.

```
4   // Open or create the SQLite database
5   const db = new sqlite3.Database('./database.db', (err) => {
6     if (err) {
7       console.error('Error opening database:', err.message);
8     } else {
9       console.log('Connected to SQLite database.');
10    }
11  });
```

Figure 11: Open SQLite database connection

Figure 12 demonstrates how to create a table called "users" if it doesn't already exist. The table has three columns:

- id: A unique primary key that auto-increments.
- name: A TEXT column that cannot be null.
- email: A TEXT column that must be unique and cannot be null. If the table creation fails, an error message is logged.



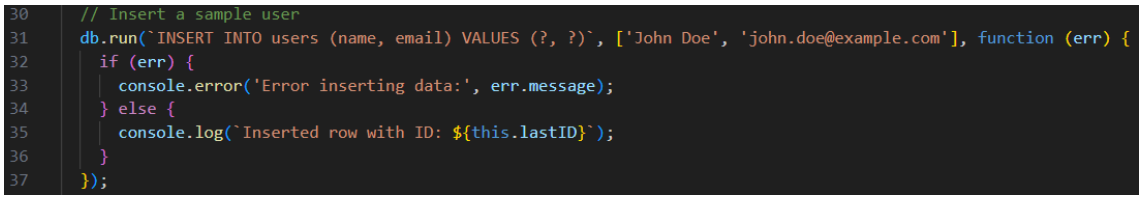
```
// Create a "users" table
db.serialize(() => {
  db.run(
    `CREATE TABLE IF NOT EXISTS users (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      name TEXT NOT NULL,
      email TEXT UNIQUE NOT NULL
    )`,
    (err) => {
      if (err) {
        console.error('Error creating table:', err.message);
      } else {
        console.log('Users table created or already exists.');
      }
    }
  );

```

Figure 12: Create a table called "users"

Figure 13 demonstrates how to inserts a sample user into the users table using a parameterized query.

- The '?' placeholders prevent SQL injection by safely binding user inputs.
- The this.lastID variable holds the ID of the newly inserted row. If the insertion fails, an error message is logged.



```
// Insert a sample user
db.run(`INSERT INTO users (name, email) VALUES (?, ?)`, ['John Doe', 'john.doe@example.com'], function (err) {
  if (err) {
    console.error('Error inserting data:', err.message);
  } else {
    console.log(`Inserted row with ID: ${this.lastID}`);
  }
});
```

Figure 13: Insert data into "users" table

Figure 14 show the code that queries all rows from the users table and displays the results.

- The db.all method retrieves all matching rows.

- Each row is logged in the format <id>: <name> - <email>. If the query fails, an error message is logged.

```
39    // Query and display users
40    db.all(`SELECT * FROM users`, [], (err, rows) => {
41      if (err) {
42        console.error('Error querying data:', err.message);
43      } else {
44        console.log('Users:');
45        rows.forEach((row) => {
46          console.log(`${row.id}: ${row.name} - ${row.email}`);
47        });
48      }
49    });
50  });
```

Figure 14: Query and display data

Figure 15 shows how to close the database connection to release resources. If the closure fails, an error message is logged.

```
52  // Close the database
53  db.close((err) => {
54    if (err) {
55      console.error('Error closing database:', err.message);
56    } else {
57      console.log('Closed SQLite database.');
58    }
59  });
60
```

Figure 15: Close SQLite database connection

When running the program shown above, your result should look something like the following example in figure 16:

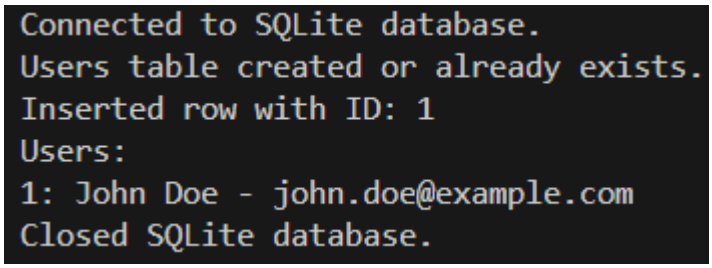


Figure 16: Result after running ‘node app.js’-command in cmd

4.4 Performance Comparison: Node.js v22 vs. v23

To evaluate the performance improvements in Node.js v23, a benchmark was conducted using a SQLite database interaction script. The script performed the following operations in a loop over 20 runs:

1. Open a database connection.
2. Insert a row into the database.
3. Query and retrieve data from the database.
4. Close the database connection.

The average execution times were recorded for Node.js v22.12.0 and v23.5.0, as per seen in Table 1:

Table 1: Node.js v22 vs. v23 Execution time comparison

Node.js Version	**Average Execution Time (20 runs)**
v22.12.0	23.30ms
v23.5.0	20.78ms

If we break the key findings in this test execution, we can state the following:

- Node.js v23.5.0 outperformed v22.12.0 by approximately **2.52ms per run**, demonstrating a **10.8% improvement** in execution time.

- These results highlight the benefits of the performance optimizations introduced in Node.js v23, particularly the updates to the V8 engine and other runtime enhancements.

While this comparison specifically focuses on versions 22 and 23, it is reasonable to assume that the performance gap would be even more pronounced if compared to earlier Node.js versions. As Node.js has continually evolved over the years, older versions lacked many of the critical enhancements to the event loop, ECMAScript module handling, and native module execution efficiency that are now standard in v23. This progressive refinement underscores the importance of staying updated to leverage these advancements in both performance and developer productivity.

5 Conclusion

The aim of this thesis was to analyze the capabilities of Node.js v23, particularly its interaction with SQLite, to determine its effectiveness in handling modern database operations. Through an exploration of Node.js's evolution and key features, alongside an in-depth look at SQLite's strengths as a lightweight, embedded database, this study has highlighted how these technologies complement one another to address contemporary application development needs.

Node.js v23 has brought significant advancements, including better ECMAScript module support, the stabilization of the node --run command, and performance improvements driven by the updated V8 engine. These enhancements not only simplify workflows but also ensure that Node.js continues to meet the growing demands of developers working with scalable and efficient applications.

SQLite, with its simplicity, ACID compliance, and versatility, remains a compelling choice for developers seeking an embedded database solution. Its integration with Node.js v23 demonstrates how lightweight database systems can be effectively paired with modern runtime environments to deliver reliable, high-performing applications.

The performance comparison between Node.js v22 and v23 underscores the tangible benefits of the latest version. With a 10.8% improvement in average execution time, the results validate the continuous optimizations made to the Node.js runtime, offering developers a faster and more efficient platform. This improvement is especially valuable for real-time and resource-intensive applications where performance is critical.

The demonstration of SQLite operations in a Node.js environment showcased the ease of setup, the simplicity of database interactions, and the measurable efficiency gains in Node.js v23. By combining theoretical insights with practical implementation, this thesis has provided a clear view of how Node.js and SQLite can be leveraged to build robust and efficient systems.

In conclusion, Node.js v23’s improvements reaffirm its position as a leading choice for server-side development, while SQLite’s embedded nature continues to make it a strong contender for lightweight database needs. The findings from this study not only emphasize the advancements in these technologies but also serve as a reminder of the importance of staying updated with the latest developments to fully harness their potential. Future studies could expand on this work by exploring additional database integrations or benchmarking Node.js against alternative runtime environments to provide a broader perspective on its capabilities.

References

1 Ulises Gascón. *Node.js for Beginners: A comprehensive guide to building efficient, full-featured web applications with Node.js.* Date of retrieval 10.12.2024.

2 Luis Llamas. *Node.js Course.* Date of retrieval 11.12.2024. https://www.luisllamas.es/en/nodejs-course/.

3 ChatGPT 4o. *What is npm?* Date of retrieval 11.12.2024.

4 Yamini Panchal & Ravi Kumar Gupta. *Building Scalable Web Apps with Node.js and Express.* Date of retrieval 14.12.2024.

5 *The V8 JavaScript Engine.* Date of retrieval 14.12.2024. https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine.

6 ChatGPT 4o. *Introduce the importance of database connectivity in Node.js applications.* Date of retrieval 15.12.2024.

7 Adam Freeman. *Mastering Node.js Web Development: Go on a comprehensive journey from the fundamentals to advanced web development with Node.js.* Date of retrieval 17.12.2024.

8 Omonigho Kenneth Jimmy. *Security Best Practices for Your Node.js Application.* Date of retrieval 17.12.2024. https://blog.appsignal.com/2024/07/03/security-best-practices-for-your-nodejs-application.html.

9 Chintan Gor. *What's New in Node.js 23.* Date of retrieval 19.12.2024. https://www.esparkinfo.com/blog/whats-new-in-nodejs-23.html.

10 *Introduction to SQLite.* Date of retrieval 23.12.2024. https://www.geeksforgeeks.org/introduction-to-sqlite/